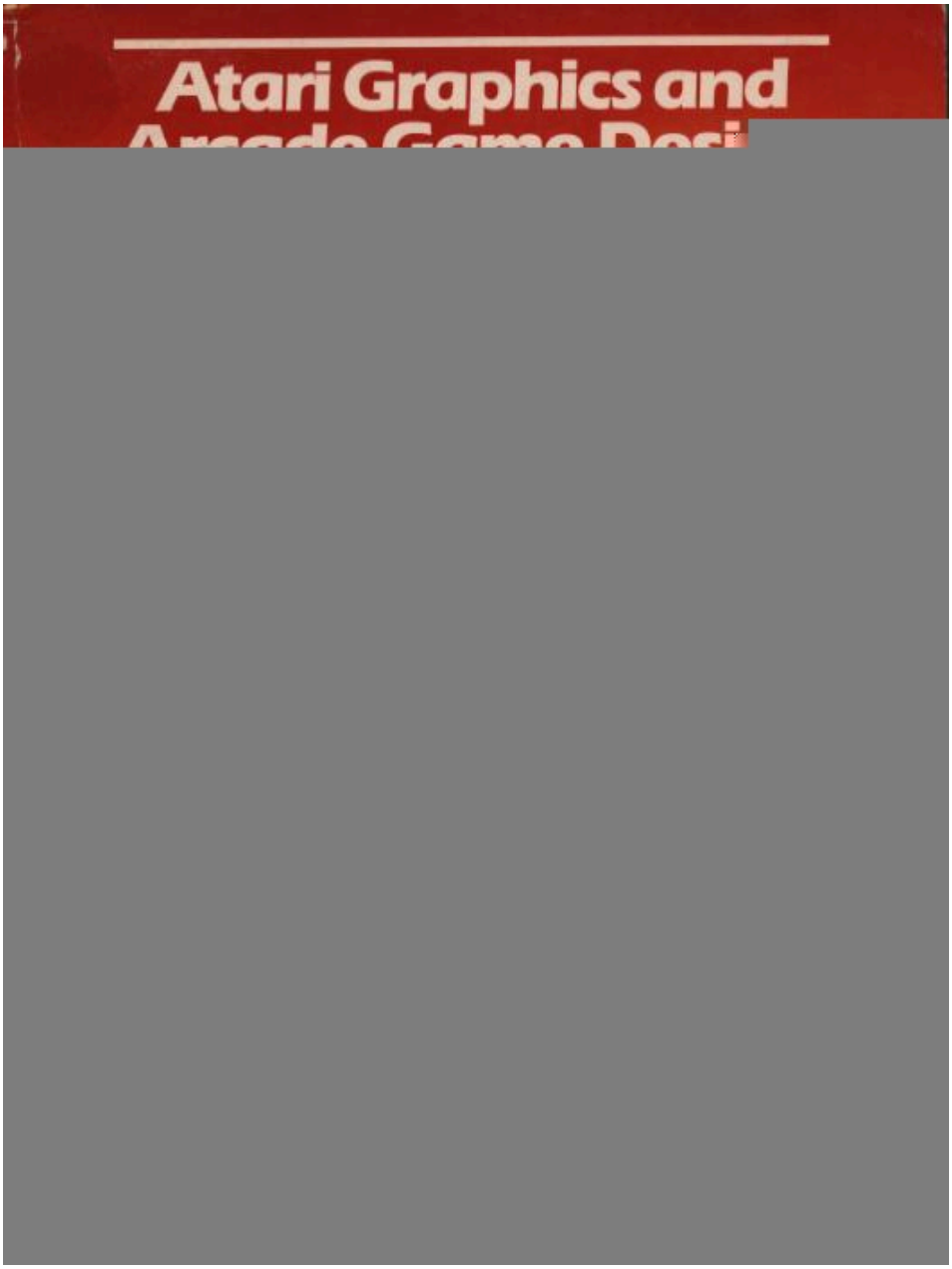
Search



Return to Table of Contents | Next Chapter

Front Matter

Atari Graphics & Arcade Game Design
by Jeffrey Stanton & Daniel Pinal
Arrays, Inc.
The Book Division
1223 S. Hindry Ave. Los Angeles, CA 90045

Acknowledgements

A technical book like this was a long, time-consuming undertaking. The book took much longer than we ever anticipated because we wanted to write the definitive book on the subject with both meaningful examples and technical accuracy. We wanted a book that was free of errors and with listings that worked,
We are indebted to the authors of De Re Atari and the A tart Personal Computer Hardware Manual (Atari Computer, Inc.) who shed light on the inner workings of the computer, and to the author of Mapping the Atari (Compute Books) who produced a comprehensive and valuable reference to all of the memory locations inside the Atari computers. Thanks to Michael Mellin who edited the manuscript without meddling with the content, and to Estela Montesinos who without complaint redid the numerous changes in my diagrams again and again until we got it right.

ISBN 0-912003-05-7

Webification notes by Kay Savetz (Jan 1 2001)

On behalf of the Atari community, I would like to thank Jeffrey and Dan for allowing their classic book to be posted on the Web. Please respect the copyright and do not redistribute, mirror, or copy this online book. Thanks to the team of Atari devotees who helped digitize the book text and program code: Willi Kusche, Erhard Puetz, Sysop Fox-1, Allan Bushman, and Ron Hamilton.
Send comments or questions about this site to Kay Savetz.

Return to Table of Contents | Previous Chapter | Next Chapter

Preface

Computer owners, who view stunning graphic effects or play visually exciting games, rarely consider the effort or techniques required to achieve those images. They don't realize that a programmer's ability to create Atari graphics can be compared to an artist's ability using a sketchpad or an animator's skill using animation techniques. In fact, an arcade game is basically an interactive cartoon in which the player controls a character or object that influences the action of the remaining computer-controlled objects on the screen. This action, like in a movie, consists of individual frames viewed at high speed to produce fluid motion. The objects in a game can be animated by either moving them from one screen position to another without changing their shape, or by changing their shapes between frames. In either case, the effect is the same -- a feeling of motion.

Atari computers are wonderful graphics machines capable of extraordinary visual effects. Unfortunately, few of these can be implemented directly from Atari BASIC without a thorough knowledge of Machine language and the architecture of the machine. Those who understand the techniques and have mastered them are mostly too busy writing programs to share their knowledge.

This book will allow you to enter the world of Atari graphics in which your most imaginative ideas can be animated. The various chapters will present a comprehensive course in both Atari graphics and high-speed arcade animation techniques. While at least half of the book requires the ability to program in Assembly language, we were careful to bigin the book with the simplest graphics concepts 'in Atari BASIC. The book alms to increase the novice programmer's skill. It assumes no prior knowledge of either Atari graphics or Assembly language. Since we know that many of our readers will be young teenagers, we made every attempt to include BASIC program examples, some with Machine language subroutines, in most of the chapters. We felt that concepts like custom display lists, color indirection, scrolling, character set animation, and player-missile graphics can be learned by beginners, but we didn't neglect the advanced programmer either. We cover the most advanced topics possible on a Machine language level. We discuss vertical blank and display list interrupts, kernals, bit-mapped graphics, sound, scrolling, and player-missile graphics, and use these techniques to develop four complete Assembly language games.

The only requirements for this book are an inquisitive mind, perseverance, and a good Assembler. Although prior Assembly language programming experience isn't necessary, you won't be able to write code without an Assembler.

We will attempt to explain the ideas in this book through a combination of text, drawings, flow charts, and working code. The concepts in this book may seem easy at times, and somewhat difficult at other times. The Atari is a complex machine with many idiosyncrasies. The hardware sometimes makes game design relatively easy, yet the concept of an interrupt-driven machine with its timing problems can make advanced programming frustrating. Out advice is to read the book in stages and try the examples. Learn how they work.

While our goal for presenting the material was to educate a new generation of arcade game designers, I dread the proliferation of copy cat games. The world doesn't need an eighth Pac-Man or a tenth Q*Bert. They have been done. We hope that programmers both young and old will use their imaginations to create something novel and exciting.

JEFFREY STANTON
VENICE, CALIFORNIA

JULY 14,1984

PROGRAM LISTINGS AVAILABLE ON DISK

Note: program listings are no longer available on disk from this address. You can download them from this Web site.

All of the code listed in this book is available on diskette to readers who disdain typing long computer programs. We barely managed to cram all of the listings without DOS on three disk sides, one side BASIC and two sides assembly language. These disks and files are unprotected. We decided to offer readers a choice of buying all three sides, just the BASIC programs, just the Assembly language source code and game object files, or a disk containing all of the games for those who only wish to play them.

The cost of these disks is nominal and can be ordered using the card in the back of the book from ~~Stanton Products, 3710 Pacific Avenue #16, Marina del Rey, California 90292.~~ The prices are as follows:

1) BASIC program listings only $10.00
2) Assembly language listings only $15.00
3) All program listings (2 disks) $20.00
4) Playable games only $12.50

Include $1.50 postage. California residents add 6 1/2% sales tax.

The F-S Macro Assembler 40/80, a completely compatible upgrade to SYNASSEMBLER, is also available from Stanton Products under a licensing agreement with Funsoft and S-C Software. It is a disk based co-resident assembler, well suited to both beginners and professional programmers. It comes in two versions on the disk; 40 and 80 column achieved entirely through software. Its powerful macro and conditional assembly features makes it one of the most powerful assemblers available for both the newer XL series of Atari computers and the older 400/800 computers. There are 25 pseudo-ops and 31 commands, designed to make life easier for the programmer. With the ability to chain source files and assemble object code to disk, the programmer is only limited by the amount of online disk storage. It is available, as all development tools should be, on an unprotected disk. A 45-page manual is included. It sells for $50.00 plus $1.50 postage and any applicable sales tax. It can also be ordered using the coupon in the back of this book.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 1
Graphics Modes And Color Registers

The ATARI 400/600/800/1200/1400 home computers are some of the most impressive graphics machines available at prices under $1000. Each of these very special machines, in addition to containing a 6502B microprocessor that runs at nearly twice the speed of many competing computers, also contains a number of custom hardware chips. One of these is a separate graphics microprocessor called ANTIC. These powerful chips free the 6502B to do what it was designed to do, calculate.

The computer was built to be extremely flexible with multiple graphics modes, redefinable character sets, indirect color registers, player-missile graphics, collision registers, display list interrupts, fine scrolling, and a built-in sound generator. These features give a polished, smooth, colorful look to the display, almost an arcade look. The effect isn't a coincidence, for the computer is nearly a clone of the hardware used in some of Atari's arcade machines like Missile Command.

Skilled Assembly language programmers can harness many of the Atari's capabilities, but less skilled or even beginning programmers may find the machine's flexiblity downright intimidating. Atari BASIC limits the programmer's choices, but it also eliminates much of the error prone graphics initialization that might produce wacky displays. Fortunately, capabilities like player-missile graphics and four-voice sound can still be accessed by PEEKing and POKEing to the machine's hardware addresses.

Most computers limit their graphics display to one or two modes because they are hardware dependent and memory specific. The Apple IIe, for instance, has Lo-Res and Hi-Res graphics. The 8K block of Hi-Res graphics is hard wired to specific memory locations $2000-$3FFF. Each byte (7 pixels) in display memory contains both pixel and color information that hardware uses to raster color dots on the television screen. If you move an object, you must change all of the bytes at both the old and new locations in screen memory without erasing the background. Worse yet, if you want to change an entire blue screen to red, you need to rewrite all 8K bytes of screen data. Similarly, scrolling requires a memory shuffle and at best results in slow, jerky motion.

The Atari, on the other hand, uses the ANTIC graphics microprocessor to interpret the graphics data in screen memory and another chip, the CTIA/GTIA, to plot the color information. Each of the fourteen different graphics modes (six text/character and eight graphic) uses differing amounts of screen memory for display. As a rule of thumb, the higher the resolution and the more colors available for display, the more memory required. Essentially, you need one bit of information for each pixel displayed plus additional bits for color information. Required screen memory, including the accompanying display list, can be as little as 261 bytes for double width, double height text, or as much as 7891 bytes for a hi-resolution 320 x 192 pixels screen. The screen can be placed nearly anywhere in memory as long as you tell ANTIC where to obtain its data.

The programmer can mix graphics modes because he can instruct ANTIC to display data from a specific part of memory in a particular graphics mode. This means you can have medium resolution graphics at the top of the screen, text in the middle and hi-resolution graphics at the bottom, if you like. The display is stacked in horizontal bands that stretch across the entire width of the screen. Any combination of display modes can be chosen as long as the entire display does not exceed 192 scan lines.

You are not even limited to using just the graphics characters in the ROM character set for your playfield graphics. A new character set can be customized with a character set editor. The computer will substitute the new set when you set the character set pointer to its address.

The unique way that the Atari computer handles color information through a series of addressable color registers gives the programmer added flexiblity. Each screen pixel is assigned to one of four playfield color registers when it is plotted. Simply changing the color value in one of these registers changes the color of every pixel assigned to that register. An entire background can be changed from red to blue with one simple POKE.

Although you can choose from 128 colors, you can work with only four at a time. Fortunately, you can gain extra colors either by adding different colored players to the display, or by changing the colors in mid-display using display list interrupts. Moreover, the addition of three GTIA modes allows either sixteen colors with one luminance, or one color with sixteen luminances, or nine colors. Although these modes sacrifice resolution to some extent, they are useful for three-dimensional shading, or adding a lot of color to the screen easily.

Perhaps the most powerful and useful graphics feature on the Atari is the playermissile graphics. Since these objects are completely independent of playfield memory, they can move smoothly over the display without affecting it. If the programmer wishes, he can set the priority so that one or more of the four players (five without missiles) can pass behind other players or playfield objects. Each player can be of different color, height and width, with a maximum width of eight pixels. In addition each player and each missile has individual collision registers that keep track of any collisions between each other and the playfields.

The Atari computer was the first home computer with player-missile or sprite graphics. It unfortunately doesn't allow true X,Y positioning. While the horizontal position can be set with a single POKE, the player's data must be shifted within a 128 or 256 (depending on the resolution) block of player memory to obtain true vertical positioning. In addition, although players can be plotted in double or quadruple width, the basic width of eight pixels imposes further limitations on the programmer.

Certainly, the use of player-missile graphics makes programming smooth animated graphics light years easier than on non-sprite-oriented computers like the Apple. But some of the newer computers like the Commodore 64 have eight players, each 24 x 21 pixels with true X,Y positioning, and both Coleco's Adam and the Texas Instrument's TI-99A have thity-two sprites, each 32 x 32 pixels with true X,Y positioning.

While it may be easier to program these computers using players exclusively, each has only three or four graphics modes that can't be mixed, and none can smooth-scroll the playfield without extensive memory shuffling.

The Atari's ability to fine scroll the graphics display either horizontally or vertically sets the Atari apart from all other home computers. The Atari does this easily because the ANTIC chip can begin with data anywhere in memory and automatically map to the screen sequential memory bytes on a line by line basis. To vertically rough scroll a 40 character per line text screen you only need to change by 40 bytes the start of the memory area from which ANTIC gets display data. You can scroll the screen less, just a few scan lines at a time or a portion of one character, by setting the vertical fine scroll register. Horizontal scrolling is done similarly, but the data structure is much more difficult to set up because the data lines do not follow sequentially in memory but have gaps to store the off-screen image. To move any row requires only resetting the memory pointer by one byte; however, the reset must be performed for each row or mode line that you want to scroll.

Since the Atari computer is interrupt driven, time critical program code can be executed during the computer's vertical blank period at a rate of exactly 60 times a second. It is also the perfect time to scroll the screen, move players, and check collisions, for discontinuous screen jumps can't occur when the electron beam is between frames.

The Atari computer has many powerful graphics features that overall far surpass any home computer on the market today. We will explain each of these features in greater detail through working arcade game examples in the appropriate chapters in this book.

Display Modes

The Atari computer's fourteen display modes includes six text or character graphics modes, and eight bit-mapped graphics modes of varying resolution. In addition, computers containing the GTIA color chip have three extra graphics modes which are special cases of the highest resolution bit-mapped graphics mode.
Programmers using the Atari BASIC cartridge from non-XL machines can easily access only nine of these fourteen display modes. Although XL machines can access four of these five additional modes directly from BASIC, these extra modes, which include several multi-colored character set modes requiring custom character sets, are best left to the Assembly language programmer.

The character graphics modes 0, 1, and 2 are easy to understand. The normal text display is graphics 0. It consists of twenty-four rows of forty characters. Each character is eight dots wide by eight dots or television scan lines high. Graphics mode I characters are just graphics mode 0 characters stretched twice as wide; only twenty of these characters fit on any row. Graphics 2 characters are as wide as graphics one characters but are twice as deep. Since each character is sixteen scan lines high, only twelve rows would fit on the screen in the full screen mode.

Normally a text window occupies the bottom four rows (32 scan lines) in all BASIC graphics modes except the GTIA modes. If you were to set the display for graphics mode 2 characters, ten rows of these characters would be displayed above four rows of graphics 0 characters. The display can be set to full screen by adding 16 to the graphics mode. Thus, invoking GRAPHICS 2+16 sets up a full screen display consisting of twelve rows of graphics 2 characters.

Graphics modes 3 thru 8 are bit-mapped graphics modes that don't involve characters. Instead they plot colored dots or pixels of varying resolution. A pixel can be as large as eight dots by eight dots in graphics mode 3, be four dots by four dots in graphics mode 5, two dots by two dots in graphics 7, or be as small as one single dot in graphics mode 8. The finest resolution in graphics mode 8 gives us complete control over every dot on the screen.

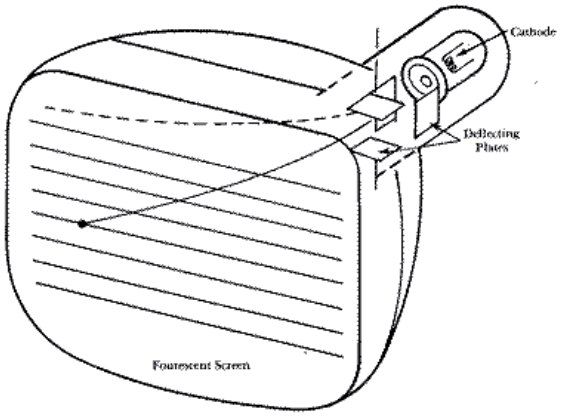
Obviously, there are more reasons to use each of these modes beyond the varying degrees of resolution. As the graphics modes increase in resolution, more memory is required for display. Bigger pixels simply fill up the screen faster and therefore require less memory. For example, a graphics 3 screen that has pixels the size of a character and a screen resolution of forty pixels by twenty-four, requires only 480 bytes for screen memory. A graphics 8 screen that stores groups of eight pixel-sized dots in one byte, requires 7680 bytes of screen memory. A programmer using this mode on a 16K Atari 400 or 600XL, would have very little room for his program.

The number of colors available is another reason for choosing a particular graphics mode. As a rule, more memory is required to display more color because the color information must be encoded within the screen data. For example, the only difference between the two color graphics modes 4 and 6, and the four color modes 3, 5, and 7 is in the amount of memory required. Graphics modes 4 and 5 have the same resolution (80 x 48), but graphics mode 4 uses only half as much memory. Graphics mode 5 must use two bits to encode the color for each pixel, where only one bit is needed to tell the computer how to display graphics mode 4 pixels. A similar relationship exists between graphics modes 6 and 7.

You have probably noticed that each graphics mode is displayed in a series of rows. A full-screen graphics 5 screen consists of forty-eight vertically stacked rows of pixels, each four scan lines high. A graphics seven screen consists of ninety-six vertically stacked rows of pixels, each two scan lines high. In each case there are 192 scan lines displayed on the screen. This number is no accident but is determined by the way television sets draw or scan a picture.

How Televisions Work

Most television sets, including the ones on which you actually watch TV, are raster scan devices. A moving electron beam strikes the individual phosphors that are painted on the inside front surface of the television tube. Each time an electron strikes a phosphor, it glows momentarily. If the beam continues to strike the same phosphor it will glow, continously.



In order to draw an entire screen full of glowing dots the electron beam has to move in a series of accurate sweeps across the picture tube. A charged deflection plate bends the electron beam so that the stream of electrons strikes a series of adjacent phosphors along a straight horizontal line. These closely packed individual dots appear to be a solid line. The electron beam starts at the top of the screen and scans from left to right. When it finishes a line, it shuts down briefly while it returns to the left edge and drops down to the next scan line. This period is known as the "horizontal blank." The electron beam does this 192 times in a process known as "raster scan." When the beam finishes it returns to the top of the screen in a time period known as "vertical blank."

Obviously, if the pixels are to remain lit for longer than a brief moment the screen will have to be refreshed quite often. The electron beam retraces its 192 line path sixty times a second. A television set actually scans 262 scan lines, but the average set can only display slightly more than 200 lines. The area above and below your rectangular playfield contains some of these extra lines.

In order to get an image other than solid white, the electron beam's intensity is varied while it is scanning. By varying the intensity or the number of electrons that hit individual phosphors, different brightness levels are achieved. This gives us shading on black and white television sets that ranges from black through various shades of grey to pure white.

The process of producing a color image is only slightly different. The electron beam scans as usual from left to right over the playfield's 192 scan lines at sixty times per second. The difference is in the phosphors on the screen's surface. Each dot consists of a triangular pattern of three sub dots; one blue, one green and one red. Instead of one electron beam there are three beams, one for each color. Each beam is aimed very precisely through a mask at the subdots containing its color. Thus, when the Atari sends to the television set or color monitor color (frequency) and luminance (amplitude) information at a particular time during the scanning process, the electron beams will produce the desired display. The Atari doesn't tell the electron beams where to display information on the screen, but instead when to display information. The Atari's special hardware chips wait until the electron beams reach a particular point on the screen before immediately sending the proper color and luminance information.

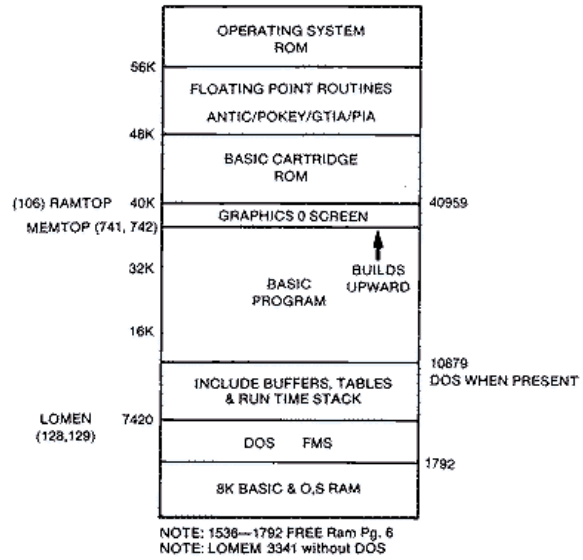
The time unit used to determine when to send color information is called the color clock. This is the amount of time it takes to change the frequency between the different colors. The electron beam has time to send 228 color clocks on one scan line. Again some of the information is plotted off screen so that the Atari displays only 160 color clocks. The color information is transmitted to the TV in the form of a square wave. When the signal is high during one clock cycle, you get one color, and when it is low you get the complementary color. Other colors are obtained by phase shifting the signal slightly.

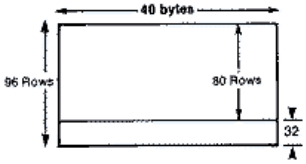
Memory Map

It is best at this time to develop an understanding of where the Atari computer stores your BASIC computer program and the location of display memory. All 6502-based Atari computers, address 64K bytes of memory whether physical memory exists or not. The computer is divided into RAM (Random Access Memory) and ROM (Read Only Memory). You can store things like your BASIC program in RAM memory. Atari 400's and 600XL's contain 16K of RAM memory, while the larger 800's, 1200's and 1400's contain 48K or 64K of RAM memory. Most of the area above 48K, from 52K to 64K is reserved for the computer's Operating System (OS), and the use of various hardware chips ANTIC, POKEY, CTIA/GTIA and PIA. The OS and all of these chips are in ROM. Locations in them can be read, but only a very few hardware locations in some of the chips can be written to. Since many of these writable hardware locations can't hold information for longer than 1/60 of a second, they are "shadowed" by RAM memory locations in the lower portion of the computer's memory. These "shadowed" memory locations, that contain information such as joystick/paddle values for each of the registers, are copied to the appropriate RAM locations during the vertical blank period between television frames. Color information "shadowed" in RAM is copied to the hardware registers at the same time. This not only saves you the trouble of refreshing the hardware each cycle, but allows you to read the "shadowed" values.
The lowest section of memory contains zero page, BASIC and OS system RAM, the stack, and the keyboard buffer. All of this occupies the space between 0 and 1535 ($000-$5FF). The area between 1536 and 1791 ($600$6FF) known as page six is free for user Machine language subroutines. The area above 1792 to location 7420 or LOMEM is reserved for the DOS File Management System. LOMEM drops to 1792 in computers that don't use a disk drive or interface module.

The 8K ROM BASIC cartridge occupies memory from 40K to 48K regardless of memory configuration. When the cartridge is engaged, RAMTOP or HIMEM drops to 40959 in 48K and 64K machines and remains at 16384 in 16K machines. The memory between LOMEM and HIMEM is the area available for running and storing your BASIC language programs.

BASIC programs are stored beginning at LOMEM. The program along with its buffers, tables, and run time stack build upwards in memory. When a graphics mode is invoked, BASIC reserves the area just below HIMEM for the graphics screen and text window if there is one. It stores a small program called the display list just below the graphics screen. This display list, which we will discuss in greater detail in the next chapter, tells ANTIC where to find display data and in which graphics mode to display it. In brief, the list contains an instruction for each row of graphics data, plus

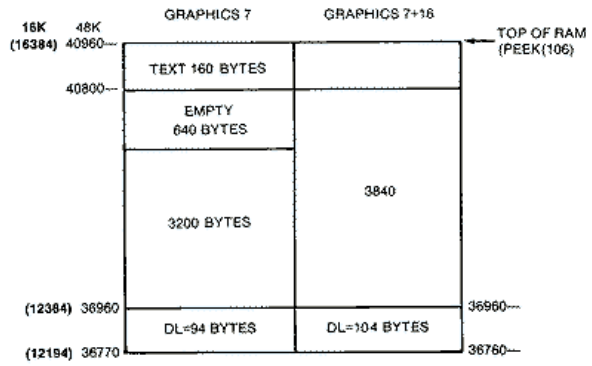




instructions about the number of blank lines to skip before plotting plus the locations of the screen's data and the beginning of its own display list.

For example, if we set up a full screen graphics 7 screen (40 x 96), BASIC places the beginning of screen memory automatically at 36960 for a 48K machine. The screen's memory is 3840 bytes. The 104-byte display list is placed at memory location 36760. If we choose instead a graphics 7 screen with a text window (40 x 80), only 3200 bytes of screen memory are required. BASIC puts screen memory at the same location but leaves the top 640 bytes empty. It stores the data for the 160 byte (4 line) text screen beginning at 40800. The display list is 10 bytes shorter and begins at location 36770 for 48K machines. The diagram below shows the appropriate addresses for 16K machines.

The advantage of having BASIC set up your graphics screen and display list is that it avoids errors that might produce weird displays. This includes displaying the proper screen data in the wrong graphics mode, or displaying a section of memory that isn't your screen data. The disadvantage is that you can't mix graphics modes or custom design a display. You are limited to displaying a single graphics mode with or without a four line text screen at the bottom.



Color

The Atari uses a very flexible method to display color. Instead of storing the color of each pixel directly in display memory, the Atari refers the color information to a specific color register. Each pixel has the color register number stored rather than a specific color. This method, is extremely flexible, but it allows only a maximum of five colors on the screen at any one time. In an effort to save screen memory at most only two bits are used to specify a pixel's color. On the other hand, if you wish to change the background color or the color and luminence of all the pixels referring to a particular color register, you need only to change the color value in that one color register.
There are five available color registers numbered 0 through 4. Color register 4 is also known as the background color register because it specifies the color and luminance for any place on the screen where nothing else is written. In the bitmapped graphics modes 3-8, this means the color between any of the plotted pixels. In text mode 0 it means the border area, not the color behind the character.

The color registers are each one byte long. The upper four bits determine the color (0-15), and the lower four specify the brightness. Only the highest three of the four luminance bits are used, so that there are eight levels of brightness. Sixteen different colors and eight levels of brightness create 128 shades of color. The arrangement for each color is in groups of sixteen. Values 0- 15 are for color #0 in different intensities, 16-31 for color #1, etc. The table below lists the values for many of the common colors.

As we said, there are five different color registers. Think of these as paint pots. You can put a color into any one of these color registers and then draw points (pixels) and lines using that color register. It is similar to drawing on a canvas with a brush. The difference is that if you draw a green line five pixels long with color register 0, screen memory doesn't store it as green, green, green, green, green, but as a series of bits 01 01010101. When ANTIC fetches screen data and feeds it to the CTIA/GTIA chip, this color chip looks to the appropriate color register to determine which color is to be put on the screen for each separate pixel. For most of the four color graphics modes the bit pattern is as follows:

00 BACKGROUND

01 REGISTER #0

10 REGISTER #1

11 REGISTER #2

Thus, if we had the following screen data for eight pixels, the colors drawn by the color chip would reflect the values stored in the different color registers. In the example below, black is in the background color register and red, blue and yellow are in the other color registers.

These five color registers are located in hardware in the actual CTIA/GTIA chip. Each time ANTIC feeds it data, it looks to these hardware locations before plotting the color. Even what appears to be a blank empty screen is still generated from the CTIA/GTIA's interpretation of the data. Even all zero bits indicate that the entire screen is just background. The chip looks to these registers thousands of times during the refresh process of updating the screen.

The operating system also maintains copies of these color registers in RAM memory. These are called shadow color registers. They are maintained because the hardware locations are "write only" locations. Since they can't be read, we need RAM locations where they can be read. At the beginning of each refresh cycle, these five shadowed registers are copied into the hardware locations.

CTIA REGISTER		O.S. SHADOW REG.	
PLAYFIELD #0 /COLOR REG #0	708 ($2C4)	53270	($DO16)
PLAYFIELD #1 /COLOR REG #1	709 ($2C5)	53271	($D017)
PLAYFIELD #2 /COLOR REG #2	710 ($2C6)	53272	($D018)
PLAYFIELD #3 /COLOR REG #3	711 ($2C7)	53273	($D019)
PLAYFIELD #4 /COLOR REG #4 (BACKGROUND)	712 ($2C8)	53274	($D01A)

BASIC uses the SETCOLOR command to set up the color registers. It is in the form of SETCOLOR (color reg #), (color #),(luminance #). A direct POKE to the O.S. shadow register is equivalent to SETCOLOR and is faster. For example SETCOLOR 1,3,8 is the same as POKE 709, (3*16)+8 or POKE 709,56.

COLOR VALUES FOR COLOR REGISTERS

	VALUE	HUE	LUMINANCE		VALUE	HUE	LUMINANCE
BLACK	0($00)	0	0	MAGENTA	82($52)	5	2
DARK GREY	4($04)	0	4	PURPLE	96($60)	6	0
GREY	6($06)	0	6	LAVENDER	102($66)	6	6
WHITE	14($0E)	0	14	BLUE	116($74)	7	4
GOLD	20($14)	1	4	LTBLUE	120($78)	7	8
YELLOW	24($18)	1	8	TURQUOISE	164($A4)	10	4
BROWN	34($22)	2	2	GREEN	196($C4)	12	4
TAN	36($24)	2	4	LIGHT GREEN	200 ($C8)	12	8
ORANGE	54($36)	3	6	YELLOW GREEN	214($D6)	13	6
RED	66($42)	4	2	OLIVE GREEN	228($E4)	11	4
PINK	72($48)	4	8	PEACH	246($F6)	15	6

Atari BASIC's COLOR command, used to specify a particular playfield register that plots points or draws lines, is probably the most confusing aspect of Atari graphics. When you choose a COLOR #, it selects a color register assigned to a playfield. It uses that color register to plot with

until a new color register is chosen. The problem is that the COLOR # often doesn't correspond to the color register and varies with the graphics mode.

There is a logical explanation for the discrepancy, but it is more apparent to the Assembly language programmer than to the casual BASIC programmer. Remember that in most modes two bits are used to specify the color. This is true in all of the bit-mapped graphics modes. Color #0 is usually background because a Machine language 00 written into display memory usually plots nothing. If COLOR # is I it writes a 01 in display memory for that pixel, if equal to a 2 it writes a 10, and if equal to a 3 it writes an 11. Unfortunately, if you refer to the table above, bits 01 corresponds to color register #0, bits 10 to color register #1, and bits I I to color register #2. On the other hand, the two color modes use only COLOR #1 to plot points because just a single bit is used to direct the CTIA to the color register.

The character graphics modes, 0, 1, and 2 are even more confusing. The upper two bits in each character not only determine which color register is selected, but effect which character is displayed. Essentially, parts of the character set appear in different colors. For example, if you are using the computer's default colors and you are in character mode #1, a 20 character per line mode, upper case letters appear in orange, lower case in light green, inverse upper case characters in dark blue, and inverse lower case graphics characters in red.

	ATASCII	COLOR REGISTER
Uppercase alphabet (A-Z) numbers, punctuation	39-90	0
Lowercase alphabet	61-122	1
Inverse uppercase alphabet numbers, punctuation	160-218	2
Inverse lowercase alphabet	225-250	3

Since the relationships between the COLOR # and the playfield it refers to differ in many of the graphics modes, beginners will do best by refering to the table below or the one in their BASIC reference manual. The concept of drawing lines and shapes using color registers as paint pots is best illustrated with the following demonstration. We will draw in graphics mode 7 full screen. This is a four color mode with three foreground colors and one background color. Initially, we will fill our paint pots or color registers with dark gray, green, blue and red.

```
COLOR REG. #0   GREEN       COLOR #1
COLOR REG. #1   BLUE        COLOR #2
COLOR REG. #2   RED         COLOR #3
COLOR REG. #4   DARK GREY   COLOR #0
```

We will draw our rectangular paint pots at the bottom of the screen and one shape in its color above each pot. We will also draw another shape or line above one of the other paint pots. When we are finished we will have six shapes above three paint pots. The solid shapes are filled in using the standard XIO color fill command in BASIC.

The XIO fill command is designed to work with four-sided figures, but if you are careful it will work with triangles. In general you plot a point in the lower right hand corner of the figure and use DRAWTO statements to reach the upper right hand corner of the figure. You next use the position statement to move the cursor to the lower right hand corner and use the POKE statement to place a number, equal to the COLOR number to be used for plotting into memory location 765. Last, you perform a XIO 18, #6,0,0,"S". The fill commands works from left to right and will fill the figure until it encounters any illuminated pixel between the left and right sides of the figure being filled. For example:

```
20 GRAPHICS 7+16
50 SETCOLOR 1,7,2:REM BLUE
80 COLOR 2
140 PLOT 35,90:DRAWTO 35,80:DRAWTO  5,80
150 POSITION 5,90
160 POKE 765,2
170 XIO 18,#6,0,0,"S"
180 GOTO 180 180
```

When you fill a triangle, two sides of the figure are drawn. While you may not be at the top of the figure when you finish, you will be on the left side when you reposition the cursor to the bottom right. Usually, the color fill works properly; but, if you look at the blue triangle, you will notice that we repositioned the cursor at the bottom one pixel to the left. This is because when the last line fills on the bottom it must have a right boundary to fill to. If the boundaries are equal, the line will begin filling from there to the right edge of the screen.

The colored bar above the paint pot indicates which paint pot or color register can be adjusted by the joystick. You can select an individual paint pot with the select key. When the bar is to the far right beyond the last paint pot, it points to the paint pot used to color the background. You can adjust the color in the paint pot by moving the joystick up or down. If you begin at the first paint pot, which is intially green, and change the color, you will notice that the two shapes that were green changed to the new color in the paint pot. It doesn't matter if the shape is above the paint pot or somewhere else on the screen. What matters is which color register or paint pot was in effect when the shape was drawn, for those pixels contain data that point to a particular color register.

You can play with the four paint pots and watch the various shapes change color. When you change the background color one or more of the other shapes will vanish if the two paint pots are identical. The shape is still there, but the pixels instruct the CTIA/GTIA chip to produce the same color as the background. The shape just blends in to become invisible.

Download PAINTPOT.BAS (Saved BASIC)
Download / View PAINTPOT.LST (Listed BASIC)

Graphics Modes

ATARI computers can display fourteen graphics modes of which nine can be directly accessed from BASIC on older machines and thirteen on the newer XL units. This section of the book will explain each of the various graphics modes, their resolution or size, the method by which they are mapped to the screen, and how colors are generated.

Graphics Mode 0 (ANTIC 2)

This is the normal-sized character or text mode that the computer defaults to on start up. Being a character mode, screen memory consists of bytes that represent individual characters in either the ROM or a custom character set. ANTIC displays forty of these 8 x 8 sized characters on each of twenty-four lines.
Graphics 0 is a 1 1/2 color mode. Color register #2 is used as the background color register. Color register #1 sets the luminance of the characters against the background. Setting the color has no effect. Bits within a character are turned on in pairs to produce the luminace color. Otherwise single bits tend to produce colored artifacts on the high resolution screen. These colors depend on whether the computer has a CTIA or GTIA chip, and the color of the background.

Graphics 1 (ANTIC 6)

This is one the expanded text modes. Each characters is 8 x 8 but the pixels are one color clock in width instead of the 1/2 color clock mode of Graphics 0 making the characters twice as wide. Only twenty characters fit on any line. A graphics I screen has twenty rows while the full screen mode has twenty-four rows of characters.
The two high bits of each ATASCII character, that normally identify lowercase or inverse video text in Graphics 1, set the color register for the 64 character set. Decimal character numbers 0-63 use color register zero, while those same 64 characters if given character numbers 64-127 use color register #1. If you are typing from the Atari keyboard, the uppercase letters A-Z ATASCII 65-90 (Internal # 33-58) are assigned to color register zero, while the lowercase numbers 97-122 (Internal # 97-122) are signed to register #1.

Graphics 2 (ANTIC 7)

This text mode is basically the same as the previous mode except that each row of pixels is two scan lines high. Thus 12 rows of 20 characters are displayed on a full screen. Only ten rows fit on a split screen.

Graphics 3 (ANTIC 8)

This four-color graphics mode turns a split screen into 20 rows of 40 graphics cells or pixels. Each pixel is 8 x 8 or the size of a normal character. The data in each pixel is encoded as two bit pairs, four per byte. The four possible bit pair combinations 00, 0 1, 10, and I I point to one of the four color registers. The bits 00 is assigned to the background color register and the rest refer to the three foreground color registers. When the CTIA/GTIA chip interprets the data for the four adjacent pixels stored within the byte, it refers to the color register encoded in the bit pattern to plot the color.

Graphics 4 (ANTIC 9)

This is a two-color graphics mode with four times the resolution of GRAPHICS 3. The pixels are 4 x 4, and 48 rows of 80 pixels fit on a full screen. A single bit is used to store each pixel's color register. A zero refers to the background color register and a one to the foreground color register. The mode is used primarily to conserve screen memory. Only one bit is used for the color, so eight adjacent pixels are encoded within one byte, and only half as much screen memory is needed for a display of similiar-sized pixels.

Graphics 5 (ANTIC A or 10)

This is the four color equivalent of GRAPHICS 4 sized pixels. The pixels are 4 x 4, but two bits are required to address the four color registers. With only four adjacent pixels encoded within a byte, the screen uses twice as much memory, about IK.

Graphics 6 (ANTIC B or 11)

This two color graphics mode has reasonably fine resolution. The 2 x 2 sized pixels allow 96 rows of 160 pixels to fit on a full screen. Although only a single bit is used to encode the color, screen memory still requires approximately 2K.

Graphics 7 (ANTIC D or 13)

This is the four color equivalent to GRAPHICS mode 6. It is the finest resolution four color mode and naturally the most popular. The color is encoded in two bit-pairs exactly the same way as in GRAPHICS 3. The memory requirements of course is much greater as there are 96 rows of 160 - 2 x 2 sized pixels. It requires 3840 bytes of screen memory with another 104 bytes for the display list.

Graphics 8 (ANTIC F or 15)

This mode is definitely the finest resolution available on the Atari. Individual dot-sized pixels can be addressed in this one-color, two-luminance mode. There are 192 rows of 320 dots in the full screen mode. Graphics 8 is memory intensive; it takes 8K bytes (eight pixels/byte) to address an entire screen.
The color scheme is quite similar to that in GRAPHICS mode 0. Color register #2 sets the background color. Color register #1 sets the luminance. Changing the color in this register has no effect, but, this doesn't mean that you are limited to just one color.

Fortunately, the pixels are each one half of a color clock. It takes two pixels to span one color clock made up of alternating columns of complementary colors. If the background is set to black, these columns consist of blue and green stripes. If only the odd-columned pixels are plotted, you get blue pixels. If only the odd-columned pixels are plotted, you get green pixels. And if pairs of adjacent pixels are plotted, you get white. So by cleverly staggering the pixel patterns, you can achieve three colors. This method is called artifacting. This all depends on background color and luminance.

The following five graphics modes have no equivalent in BASIC on older machine but if indicated do correspond to an equivalent graphics mode on the newer XL models.

Antic 3

This rarely used text mode is sometimes called the lowercase descenders mode. Each of the forty characters per line are ten scan lines high, but since each of the characters are only eight scan lines high, the lower two scan lines are normally left empty. However, if you use the last quarter of the character set, the top two lines remain blank, allowing you to create lowercase characters with descenders.

Antic 4 (Graphics 12-XL computers only)

This very powerful character graphics mode supports four colors while using relatively little screen memory (1 K). In addition its 4 x 8 sized characters have the same horizontal resolution as GRAPHICS 7, yet twice the vertical resolution. A large number of games with colorful and detailed playfields use this mode.
These characters differ considerably from ANTIC 6 (BASIC 2) characters, in that each character contains pixels of four different colors, not just a choice of one color determined by the character number. Each byte in the character is broken into four bit pairs, each of which selects the color register for the pixel. That is why the horizontal resolution is only four bits. A special character set generator is used to form these characters.

Antic 5 (Graphics 13-XL computers only)

This mode is essentially the same as ANTIC 4 except that each character is sixteen scan lines high. The character set data is still eight bytes high so ANTIC double plots each scan line.

Antic C (Graphics 14-XL computers only)

This two-color, bit-mapped mode the eight bits correspond directly to the pixels on the screen. If a pixel is lit it receives its color information from color register #0, otherwise the color is set to the background color register #4. Each pixel is one scan line high and one color clock wide. This mode's advantages are that it only uses 4K of screen memory and doesn't have artifacting problems.

Antic E (Graphics 15-XL computers only)

This four-color, bit-mapped mode is sometimes known as BASIC 7 1/2. Its resolution is 160 x 192 or twice that of GRAPHIC 7. Each byte is divided into four pairs of bits. Like the character data in ANTIC 4, the bit pairs point to a particular color register. The screen data, however, is not character data but individual bytes. The user has a lot more control, but this mode uses a lot more memory, approximately

GTIA

There are three additional graphics modes in the GTIA chip that are actually special interpretations of ANTIC mode $F, a high resolution graphics mode. They are designed to add a lot more color to the screen without sacrificing too much resolution. Graphics mode 9 is a one-color, sixteen luminance mode. Mode 10 is a nine-color mode with independent luminance setting, and mode 11 offers sixteen colors set at one luminance. Each mode has a resolution of 80 columns by 192 rows. There is no split screen in these modes.
The GTIA modes are selected by the upper two bits in the priority register shadowed at location 623 ($26F). BASIC programmers can reach any of these GTIA modes by normal graphics statements GRAPHICS 9, 10, or 11. Others will need to POKE the correct bit. Graphics mode 9 can be activated by turning on bit 6 with a POKE 623,64. GRAPHICS 10 is activated by turning on bit 7 with a POKE 623,128. Both bits 6 and 7 need to be set with a POKE 623,192 to activate graphics mode 11. These values will disturb the other bits in GPRIOR that set various functions, so take care if you have set any other bits previously by POKEing a value that combines both. These other bits allow the combination of all four missiles into a fifth player, establish player-missile and playfield priorities, and enable multiple-colored or overlapping players.

The GTIA chip was not standard equipment on Atari computers until December, 1981. Those with older computers may wonder if this chip is installed. If you are on the text screen (GR.0) and do a POKE 623,64, the screen will go black and become unreadable with the GTIA chip. If nothing happens, you have the CTIA chip. Your machine can be updated to the newer GTIA chip by your Atari dealer if you desire.

Since GTIA modes allow a lot more color while using the same screen memory as GRAPHICS 8, more bits are needed to keep track of the color. In fact sixteen colors require four bits, so that only two pixels are encoded within a byte instead of the usual eight. This is the reason the horizontal resolution drops from 320 pixels to 80 pixels per scan line. The pixels are elongated instead of square. Also, since there are only nine color registers in the computer, the sixteen colors are bit mapped to the screen as in other computers, instead of by the Atari method of color indirection.

Graphics 9

GRAPHICS mode 9 produces up to sixteen different luminances of the same hue. This is quite useful for drawing pictures that require alot of shading, or for digitizing pictures. The main color is set by the background color register #4. You can use the SETCOLOR command to set the color value in the upper four bits (nybble), and the luminance in the lower four bits to zero. The COLOR command is used to vary the luminance. What actually happens is that the pixel data from ANTIC is logically ORed with the lower nybble of the background color register to set the luminance that appears on the screen. A quick little program that will demonstrate the mode is listed below.

```
10 GRAPHICS 9
20 SETCOLOR 4,1,0
30 FOR I=0 TO 15
40 COLOR I
50 PLOT I+20,10
60 DRAWTO I+20,40
70 NEXT I
80 GOTO 80
```

Graphics 11

GRAPHICS 11 is a one luminance, 16 color mode. The luminance this time is set by the background color register #4. The SETCOLOR command is used to set up the single luminance value in the lower nybble of this register, while zeros representing the hue are placed in the upper nybble. The COLOR command is used in this mode to select the various colors. This time ANTIC's pixel data is logically ORed with the upper nybble of the background color register to set the hue that appears on the screen. A typical example follows:

```
10 GRAPHICS 11
20 SETCOLOR 4,0,6:REM LUMINANCE = 6
30 FOR I=0 TO 15
40 COLOR I
50 PLOT I+20,10
60 DRAWTO I+20,40
70 NEXT I
80 GOTO 80
```

This sixteen-color mode doesn't use the four playfield color registers for color indirection. The colors on the screen are determined directly by the data bits stored in memory. Each pixel has the value for the hue stored in memory. For example, a blue hue, which is number 7, has a bit value 0 111. This is what is stored in the nybble. Two adjacent pixels, which are stored in the same byte have a value of 01110111 or 119 decimal ($77). All have the luminance assigned to the background color register.

There is a slight advantage to not using the color registers in either GRAPHICS 9 or 11: even more color can be added to the screen by adding players of different colors. The disadvantage is that the collision registers are useless in both modes.

Graphics 10

GRAPHICS 10 is probably the most versatile of the GTIA modes. While it only has nine colors with variable luminance, it does use color indirection to produce its color. This means that the screen data points to one of the five playfield-color registers and the four player-color registers to obtain its color information rather than storing the color data on the screen as in the other two GTIA modes. Since players use the same color as part of the background, you must be careful. They will blend in when they coincide with that portion of the playfield using the same color. Collision registers don't work in this mode either. We believe ANTIC's collision registers are bypassed in all GTIA modes. A typical GRAPHICS 10 example follows;

```
10 GRAPHICS 10
20 FOR I=0 TO 8
30 N=I
40 POKE 704+I,N*16+6
50 COLOR I
60 PLOT I*2+20,10
70 DRAWTO I*2+20,30
80 NEXT I
90 POKE 704,0:REM FOR BLACK BACKGROUND
100 GOTO 100
```

When you choose the color register via the COLOR statement in BASIC, 16 different values can be used but only the first nine are valid. The remainder just repeat various playfield registers. The chart is listed below. You will notice that COLOR 4 no longer sets the background color. Instead it is set by player O's color in register number 704. It is best to use POKEs to set the color registers rather than SETCOLOR.

COLOR STATEMENT	REGISTER #	PLAYFIELD
0	704	PCOLOR0
1	705	PCOLOR1
2	706	PCOLOR2
3	707	PCOLOR3
4	708	PLAYFIELD 0
5	709	PLAYFIELD 1
6	710	PLAYFIELD 2
7	711	PLAYFIELD 3
8	712	PLAYFIELD 4

You can achieve some really nice animation effects when using this mode through the power of color indirection. If you don't count the background, you can rotate eight colors through the color registers to create a sense of motion.

The example below draws a very colorful rectangular box in perspective. Part of one side was left open so that you can see more of the left and back sides. The different color registers or paint pots are set up in a bucket brigade. Since location 713 isn't used for anything, we used this as a temporary storage location. The color or paint from register 705 is first put into this temporary bucket or storage location, then shift the rest of the colors by moving them from the higher color register to the next lower one. This is done in a FOR ... NEXT loop. We POKE the lower color register with the value we find in the next higher color register.

Download GR10DEMO.BAS (Saved BASIC)
Download / View GR10DEMO.LST (Listed BASIC)

GTIA Trick

The GTIA modes, because they are special cases of GRAPHICS mode 8, require a considerable amount of display memory, approximately 8K. Graphics mode 0, a text mode, in many respects is very much like our high resolution mode. It is a 1 1/2 color mode, and the individual pixels within a character are the same.
If you turn on GTIA mode I I from GRAPHICS 0 by POKING 623,192 the screen turns black and you get weird colored pixel patterns where your characters were. Recall from the above discussion that the color pixel patterns are directly bit mapped in screen memory in pairs of four bits, or nybbles, two per byte. If we could rewrite the character set so that the sixteen possible nybble pairs were in the first sixteen characters in the set, we could plot colored blocks the size of characters. While it wouldn't be as fine a resolution as in normal GRAPHICS 11, it would only require 960 bytes of screen memory, quite a substantial savings.

The bit pattern that is used to set color registers is the same one that we need to set up our pair of nybbles in each row of our character. The chart is listed below.

BITS	VALUE	COLOR
0000	0	grey (no color)
0001	1	light orange
0010	2	orange
0011	3	red orange
0100	4	pink
0101	5	purple
0110	6	purple blue
0111	7	blue
1000	8	blue
1001	9	light blue
1010	10	turquoise
1011	11	blue green
1100	12	green
1101	13	yellow green
1110	14	orange green
1111	15	light orange

We need to POKE a $00 into each of the bytes of the Oth character, a $11 (decimal 34) into the bytes for the 1st character, a $22 (decimal 34) into the bytes of the third character, etc. The values are seventeen apart so that it is simple to put the correct color nybble values into an array CT(16). Then it is a straightforward affair to method of POKEing the values in eight at a time into the proper character position in the new character set.

Any of these special GTIA color characters can be plotted to the screen at a specific position by calculating the offset from the beginning of screen memory. This location is stored at locations 88 and 89.

SCREEN PEEK(88)+PEEK(89)*256
OFFSET (40 x row #) + column #
LOCATION = SCREEN + OFFSET

So if you want to plot a purple pixel at character location (2,2) you do a POKE SCREEN + 82, 85. The luminance is set by the background color register 712.

An example of putting a row of sixteen different-colored GTIA pixels in GRAPHICS 0 is shown below.

Download GTIATRIC.BAS (Saved BASIC)
Download / View GTIATRIC.LST (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 2
Display Lists

We introduced you briefly in chapter I to a graphics microprocessor called ANTIC that is capable of displaying any of fourteen graphics modes. Since any screen actually consists of a collection or vertical stack of these individual graphics modes, ANTIC looks to a program called the display list to determine in which graphics mode it should display the screen data. The fact that there is something resembling a graphics display instruction set makes the computer extremely flexible. It becomes possible to display any collection of graphics modes from data in screen memory that can be stored virtually anywhere within the computer's RAM memory. This flexibility allows the user to mix graphics modes and even scroll the screen in any direction by altering the portion of screen memory displayed.

ANTIC, like most true microprocessors, has an instruction set that is used to write the display list program. The display list specifies three things: where the screen data is located, what display modes to use to interpret the screen data, and what special display options, if any, to implement.

Antic Instruction Set

ANTIC has a simple instruction set with only four basic instruction types. -there are map mode instructions, character mode instructions, blank line instructions, and jump instructions. Map mode instructions instruct ANTIC to display a mode line as colored pixels, while character mode instructions tell ANTIC to display a mode line with character data either from its internal ROM or from your own custom designed set. Blank line instructions instruct ANTIC to display a number of horizontal scan lines with solid background color. Like GOTO statements in BASIC, jump instructions change the value in ANTIC's program counter so that it looks for its next opcode somewhere else.

Special Options or Modifiers

ANTIC also has a number of special options or modifiers to its map and character mode instructions. These are specified by setting one of four high bits in the instruction set. These options are load memory scan (LMS), display list interrupt (DLI), vertical scroll, and horizontal scroll.
The load memory scan option is the most frequently used option for it occurs at least once in every display list. It specifies where the screen data is stored in memory. Technically, only one of these instructions is actually needed in any series of display modes because screen memory is usually continuous. However, if memory isn't continuous, either within a particular display mode or at the boundry between different modes, an LMS instruction is needed each time a new section of memory is used. Another instance where an additional LMS instruction is required, is in ANTIC modes E and F (GRAPHICS 8) where continuous screen memory crosses a 4K boundary. The load memory scan option is invoked by adding a decimal 64 ($40) to the map or character in the mode instruction. This is equivalent to setting the sixth bit in the instruction. LMS instructions are three bytes long. The first byte is the opcode specifying the mode and the last two bytes contain the address of screen memory in low byte, high byte order.

The other three modifiers are sometimes used by Assembly language programmers to achieve special effects. Setting bit 7 or adding 128 to the opcode enables a display list interrupt. 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 in memory locations 512,513 decimal ($200, 201). Display list interrupts are often used to change the colors in the color registers over part of a screen or to change between character sets midway down the screen. We will discuss these uses in detail in chapter 6 and 4, respectively.

Horizontal scrolling can be set up by adding 16 to the opcode or setting bit 4. Likewise, you enable vertical scrolling by adding 32 to the opcode or by setting bit five. These modifiers allow you to fine scroll the screen in either direction. Naturally if you are planning to scroll through memory by changing the start of screen memory,

you will need to combine your scroll modifier with a load memory scan modifier. This technique will be shown in more detail in chapter 7.

Note:
1) Display List Interrupts can be enabled by adding 128 decimal, $80 Hex to above values
2) Since it is impractical to do horizontal scrolling without a LMS instruction values are not referenced.

Blanking Instructions

The blanking instructions generate a certain number of blank scan lines in the color and luminance of the background or border color. From I to 8 blank scan lines can be generated by these opcodes. They are primarily used to correct the overscan on a television set.

Jump Instructions

There are two jump instructions. The first (JMP) tells ANTIC to continue looking for instructions at a different address. It is equivalent to a GOTO in the display list. It is a three byte instruction with the address in low byte, high byte order following the opcode. Its only function is to provide a solution to the display list's inability to cross a I K boundary. If for any reason your display list must cross a 1K boundary, then it must use a JMP instruction. Otherwise, don't worry about this instruction.

The second jump instruction (JVB)-Jump and wait for Vertical Blank-is used in every display list. It is a three byte instruction; the address in low byte, high byte order follows the opcode. JVB tells ANTIC to jump to the start of the display list and wait for a new screen refresh to begin. Surprisingly, this address doesn't have to be accurate, because the OS keeps track of the top of the display list and passes it to ANTIC during the vertical blank. However, you should try to maintain the real address because if you use any SIO functions such as the disk drive or the printer, ANTIC won't be updated properly and the jump will be to the address that you specify in the instruction.

ANTIC INSTRUCTION SET

Instruction Decimal	Hex	Comment
0	0	1 Blank Line
16	10	2 Blank Lines
32	20	3 Blank Lines
48	30	4 Blank Lines
64	40	5 Blank Lines
80	50	6 Blank Lines
96	60	7 Blank Lines
112	70	8 Blank Lines
1	1	Jump to Location
65	41	Jump & Wait for VBlank

Typical Graphics 0 Display List

Let us look at the display list for a typical Graphics #0 screen, the standard text mode. If you look at the chart below, you will see that this is ANTIC mode 2. It is a character mode with forty bytes per line. Each row is eight scan lines high, and there are twenty-four rows of characters. To produce an entire screen of text, twenty-four mode lines of ANTIC mode 2 will be required.

In BASIC the display list is setup automatically in memory just below screen memory. The memory pointers to the top of the display list are shadowed at locations 560, 561 decimal in low byte, high byte order. Thus:

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

It is possible to look at the display list if, we write a short program to print the entire list to either the screen or a line printer. Display lists are easy to view if there are at least four lines of text 0; but viewing display lists for full screen graphics modes, and especially custom graphics modes, can be a problem without a printer. The reason is that you must PEEK the values in the display list while you are in the appropriate mode, and not when you return to text mode 0. A method to avoid the problem is to store the PEEKed display list elsewhere in memory and print it after you return to the text mode. A safe spot depending on circumstance might be 256 or more bytes below the desired display list.

The following program will print the display list for GRAPHICS 0 to the screen.

```
10 GRAPHICS 0
20 DLIST=PEEK(560)+256*PEEK(561)
30 FOR I=0 TO 40
40 PRINT PEEK(DLIST+I);" ";
50 NEXT I
60 GOTO 60
```

If you choose to print the list to the line printer then change line 40 to;

```
40 LPRINT PEEK(DLIST+I)
```

It is quite useless to attempt to get an 80-column printer to print the display list across several rows instead of in a vertical column. Putting in a semi-colon at the end of the print line does little more than place two values on one line and this becomes confusing. The fault lies in the Operating System's printer driver.

The 32 byte long display list appears as follows for a 40K or larger computer with BASIC present;

```
112     Blank 8 scan lines to provide for overscan
112
112
66      \Display ANTIC mode 2 (BASIC 0) 64+2
64      |Screen memory starts at
156     /64+156*256 =40000
2       \Display ANTIC mode 2 for second mode line
2       |
2       |
2       |
2       |
2       |
2       |
2       |22 more ANTIC mode 2 modes
2       |
2       |
2       |
2       |
2       |
2       |
2       |
2       |
2       |
2       |
2       |
2       /
65      \JVB-Jump and wait for Vertical Blank
32      |to display list address which starts
156     /at 32+256*156=39968
```

The first thing you notice is that every display list begins with three "blank 8 lines" instructions. This is to defeat the television's overscan by starting the display twenty-four scan lines down. The next instruction is a load memory scan (LMS). The ANTIC mode number is added to 64. The next two bytes are the low, high byte address to the

beginning of display memory. Since the LMS instruction counts as the first display mode, only twenty-three more display ANTIC mode 2 instructions are needed. Finally there is a JVB instruction that resets the program counter to the top of the display list at location 39968.

The display list for a 16K machine is similar. The differences are in the locations of the top of the display list and the start of screen memory. The start of screen memory is at 15424 decimal. The low byte after the LMS instruction is 64 and the high byte is 60. The top of display list is at 15392 decimal. The low byte after the JVB instruction is 32 and the high byte is 60.

Mixing Graphics Modes

You are not always stuck with a homogenous stack of display modes. After all, splitting text and graphics is mixing modes. It is very easy to mix display modes just by changing a single display mode instruction in the display list. For example, we could change the twelfth row of GRAPHICS 0 text in the program below to a GRAPHICS I elongated text mode (ANTIC 6) characters by changing the 12th display instruction in the display list.

```
20 GRAPHICS 0
30 DLIST=PEEK(560)+PEEK(561)*256
40 FOR I=1 TO 24
50 PRINT "LINE";I;" THIS ROW HAS FORTY CHARACTERS";:IF I<24 THEN ?
60 NEXT I
1000 GOTO 1000
```

If we count down the display list starting with the Oth byte, a POKE DLIST+ 16,6 would change the text characters to GRAPHICS I characters. Do this by adding; 70 POKE DLIST + 16,6. The trouble is that everything below our new mode line is offset by half a row or twenty bytes. To understand why this occurs you need to understand what happens when ANTIC receives instructions to display a particular mode.
When ANTIC gets an instruction to display a particular graphics or character mode it automatically goes to display memory and gets the precise number of bytes of data necessary to display that mode line. When it sees an ANTIC 2 (GRAPHICS 0) display mode it retrieves forty bytes and interprets them in the proper display mode. When it sees another ANTIC 2 display mode it retrieves the next forty bytes in sequence from display memory. If instead it sees an ANTIC 6 (GRAPHICS 1) display mode, it only retrieves twenty bytes. Upon encountering the next ANTIC 2 display mode ANTIC retrieves another forty bytes. The trouble is that each of our print statements start at intervals of exactly forty bytes from the beginning of screen memory. When ANTIC only retrieved twenty bytes, it displayed only half of our printed line. The next ANTIC 2 display mode retrieves memory beginning with the last half of our line of text and displays it on the next line. Each of the ANTIC mode 2 lines on subsequent lines are also off by twenty bytes. You could correct this by adding another ANTIC 6 mode instruction immediately below the first one. Add line 80 POKE DLIST+17,6 to the program. The text for the twelfth row is now split between the two display lines of elongated text. Since we retrieved forty bytes less memory in producing the screen, the last row in memory isn't displayed.

You can experiment by changing any display instruction to any desired mode. If you try substituting BASIC GRAPHICS 2 (ANTIC 7) characters for those same two lines by POKEing a 7 instead;

```
70 POKE DLIST+16,7
80 POKE DLIST+17,7
```

the bottom of the display gets pushed downward, partially off screen.
What has happened is that we are now displaying more than 192 scan lines, 208 lines to be exact. While this isn't a major problem, it is possible to confuse the television screen's timing and the picture may begin to roll. As a rule of thumb, you shouldn't have more than 192 scan lines in your display. Displaying fewer scan lines will cause no problems. In fact it will decrease the 6502 execution time by reducing the number of cycles stolen by ANTIC to display the screen data.

Moving The Text Window

In BASIC, whenever you specify mixed text in any graphics mode, the four lines of text are automatically placed at the bottom of the screen. This is rarely the best location for it. It is often preferable when designing games to put your scoring information at the top. The text window is easy to move if you rewrite the display list.
If you look at a GRAPHICS 3 display list you will obtain the following values. The display list on the left is the one BASIC defaults to, and the one on the right is the display list we need to have four lines of text at the top and a GRAPHICS 3 screen below.

```
112 Blank 8 lines
112
112
 72 \LMS display ANTIC 8
112 |Screen memory starts at
158 /112+(158*256)
  8 Display ANTIC 2 (GR.0)
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
 66 \LMS display ANTIC 2
 96 |Text memory starts at
159 /96+(159*256)
  2 Display ANTIC 2
  2
  2
 65 \JVB
 78 |Address to top of DLST
158 /78+(158*256)
```

```
112 Blank 8 lines
112 to provide for overscan
112
 66 \LMS display ANTIC 2 (GR.0)
 96 |Text memory starts at
    |
159 /96+(159*256)
  2
  2
 72 \LMS display ANTIC 8 (GR. 3)
112 |Screen memory starts at
158 /112+(158*256)
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
  8
 65 \JVB
 78 |Address to top of DLST
158 /78+(158*256)
```

There are two ways to modify the display list. The first method changes only the groups of bytes that need to be modified. If the change is simple, you need rewrite only a fraction of an entire display list. In this example we are only shifting sections of the display list in order to change the position of the text window, so we can move blocks of display list data around if we are careful not to overwrite data. The diagram shows the sequence of the three moves. The actual move is accomplished in three short FOR-NEXT loops. The three data bytes at DLIST+3, 4, and 5 are read and POKEd into locations DLIST+9, 10, and 11. The other two moves are similar.
Download DLIDEMO1.BAS (Saved BASIC)
Download / View DLIDEMO1.LST (Listed BASIC)

BASIC doesn't even notice the change. It keeps internal pointers to the locations of both text memory and screen memory. Since we didn't change these values text is printed correctly to the text area, and graphics are plotted correctly on the shifted screen. You might observe that when we set the colors for plotting in GRAPHICS 3, we affected the background of the text window. This is because SETCOLOR 2 affects the text background. We avoided plotting with SETCOLOR I because this register affects the luminance of the letters. We won't have any problem if we set this color register as long as we keep in mind that the luminance must be 6 or greater for readablity.

The second method and sometimes the easier method when rewriting the display list is to put the new list in data statements. Then it is only a matter of reading the list and POKEing the values into memory starting at the top of the old display list. Since the screen does act funny during the change, ANTIC can be temporarily disabled by writing a zero into SDMCTL at location 559 (32217). After the list has been POKEd into position, you turn ANTIC

back on with a POKE 559,34. If you did it right, the screen will appear exactly as you set it up. If for some strange reason you decide to move the display list, remember to tell the operating system by POKEing the address to the top of the list in locations 560 and 561 ($230,231), low and high bytes respectively.

Download DLIDEMO2.BAS (Saved BASIC)
Download / View DLIDEMO2.LST (Listed BASIC)

Custom Display List For Mixing Graphics Modes

The last example is a custom designed display list with two graphics modes split by a row of expanded text. Whenever you decide to design any custom screen it is best to lay out the design on paper, and translate it into a sequence of mode lines. Once you have looked up the number of scan lines required for each mode line, you must double check the line count so that it does not exceed 192 scan lines. You then translate the sequence of mode lines into a sequence of ANTIC mode bytes.

Our display will consist of sixteen mode lines or rows of GRAPHICS 5 (ANTIC 10) pixels followed by a row of enlarged GRAPHICS 2 (ANTIC 7) text, followed by fourteen mode lines or rows of GRAPHICS 3 (ANTIC 8) pixels. Since each of the GRAPHICS 5 pixels are four scan lines high, this portion of the screen requires 16 x 4 = 64 scan lines. The one line of GRAPHICS 2 text requires sixteen scan lines. GRAPHICS 3 mode lines are eight scan lines high. The fourteen rows at the bottom require 14 x 8 = 112 scan lines. This gives us a total of 192 scan lines.

In BASIC it is sometimes easier to let the system set up the memory area for your screen and display list. Since you need to be careful that the screen memory requirements don't collide with the top of memory, you always choose a graphics mode that uses at least as much screen memory as the one you are custom designing. Obviously a GRAPHICS mode 5 screen is larger in memory than one that is partially GRAPHICS 5 and partially GRAPHICS 3. GRAPHICS 5 screens require 960 bytes of memory while GRAPHICS 3 screens require only 240 bytes.

The location of screen memory for any GRAPHICS mode can be found at locations 88 and 89. SCREEN = PEEK(88)+PEEK(89)*256. The individual values of the low byte and high bytes for a GRAPHICS 5 screen in a 40K+ machine are 160 and 155 respectively. This is the lowest address of screen memory corresponding to the upper left corner of the screen. Memory builds upwards towards the top of memory.

Each of our sixteen mode lines of GRAPHICS 5 pixels requires twenty bytes. Therefore that portion of the screen requires 16 x 20 = 320 bytes. If we add 320 bytes to the beginning of screen memory, the next byte (the beginning of our text memory) in low byte/high byte order is at:

```
High Byte      Low Byte
 155            160
                320
```

```
     ___             ___
     155             480       reduce to a value (0-255)

or   156             224
```

The reason for the apparently nonsensical decimal math is that when you complete the addition of 320 and 160 in the low order byte the resulting decimal value 480 is larger than the storage capacity of one byte (255 decimal or I I I I I I I I binary). The carry bit is set and the high order bit of the low byte is carried to the low order bit of the high byte. The resulting address is the beginning of text memory for our GRAPHICS 2 text. Since it requires only twenty bytes of display memory, the beginning of our GRAPHICS 3 memory is at 244 for the low byte and 156 for our high byte. With this information it is now possible for us to create a display list. That display list is shown below.

```
        112     Blank 8 lines
        112
        112
        74      \LMS Display ANTIC mode 10 (GR.5)
        160     |Screen memory starts at
        155     /160+(155*256)
        10      Display ANTIC mode 10
        10
        10
        10
        10
        10
        10
        10
        10
        10
        10
        10
        10
        10
        71      LMS Display ANTIC mode 7 (GR.2)
        224     \Text memory starts at
        156     /224+(156*256)
        72      \LMS Display ANTIC mode 8 (GR.3)
        244     |Screen memory starts at
        156     /244+(156*256)
        8       Display ANTIC mode 8
        8
        8
        8
        8
        8
        8
        8
        8
        8
        8
        65      \JVC (jump and Wait for VBLANK)
        104     |Address of top of display list
        155     /104+(155*256)
```

Great care should be taken when designing a display list. One of the most frequent sources for error is forgetting that the LMS instruction is your first mode line for a particular graphics type. Forgetting this will cause you to have too many scan lines. The second source of error is incorrectly calculating the address for display memory. Once you begin looking at the wrong portion of memory, nothing appears that you print, plot, or POKE to.

Plotting Points & Lines Using A Custom Design

Plotting GRAPHICS 5 pixels on the top portion of our display is straightforward. All that you need to do is choose a color register, then PLOT. But on the lower portions of the screen, the Operating System must first be told which graphics mode to plot in and where screen memory is located. The current display mode is stored in memory location 87. Many programmers use this location to fool the OS into thinking that it is in a different

GRAPHICS mode by POKEing it with a number from 0 to 11. This value is the same as the BASIC graphics mode number. In addition, the lowest address of screen memory stored at locations 88 and 89 must be changed. In our example, when we wanted to plot to the GRAPHICS 3 portion of the screen, we POKED a 3 into location 87. We then put the beginning of the screen address into locations 88 and 89. These are the same values as our LMS address for this portion of the screen. If you don't perform this operation the OS will incorrectly calculate the memory addresses to plot your pixel or line of pixels. Similarly the OS needs to be informed when we print our message in GRAPHICS mode 2 characters. A POSITION statement must also be included because the OS automatically does a position each time it plots. The #6 type print statement must be used because this is screen memory, not text memory.

In summary, the ability to design a custom display list is very valuable. It gives you flexiblity that is not available on other computer systems. By customizing the display you can give the screen a personality all your own.

Download DLIDEMO3.BAS (Saved BASIC)
Download / View DLIDEMO3.LST (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 3
Character Set Graphics

The Atari computer is one of the few machines endowed with enormous flexibility when it comes to handling character set graphics. Most computers have a set of character shapes stored permanently in Read Only Memory (ROM). The Atari is no exception. In addition to the usual upper and lowercase letters and numbers, twenty-nine control keys represent graphics symbols. These include a heart, diamond, spade, and club useful for creating playing cards, and a number of triangles, small squares, and diagonal and edge lines suitable for creating geometric graphic displays. While special characters can be useful in some playfield designs, it is impossible to achieve widely varied playfield designs using characters on most computers without resorting to slower bit-mapped character shapes.

The Atari, fortunately, takes the concept of character graphics one step further and allows the user to redefine the character shapes to create colorful and varied backgrounds. A hardware register and its shadow register in RAM keeps track of the location of the character set currently being used. Normally the register defaults to the set stored in ROM at $E000, but you can give it any RAM address that falls on a 1K boundary. When ANTIC is given instructions to put character data on the screen it fetches character data from the character set stored at this location.

A character set consists of a maximum of 128 eight-byte shapes. Each character is represented on the screen as a group of 8 x 8 dots or pixels. When reading text on the screen it appears that there are gaps between characters, but there aren't. The gaps are provided by leaving blank space near the edge of the block alloted to the individual characters. Since there is no space otherwise defined between characters, it is possible to create larger shapes consisting of groups of adjacent characters, or background with no breaks in it.

If we take a look at a single character like the capital letter A from the ROM character set, it forms a pattern of onoff dots as illustrated below. The pixels in each of the eight rows, numbered here from 0 -7, are represented in the character set as a value from 0 -255 ($00-3FF). The positions of the individual pixels in the row determine the value.

Pixel values are calculated using a base two numbering system. If a pixel is in the rightmost column of a row, the bit is set, and it has a value of one. If the lit pixel is in the next column to the left, that bit is set, and it has a value of two. The values increase by a factor of two until we reach the left most column which has a value of 128. Thus, if you want to determine the value of the row in a character you add up the individual bit values for the lit pixels. If we look at row #1 in the letter A we find that the fourth and fifth pixels are lit. The value of the set bits is 8+16 = 24 ($18). If all the pixels were lit we would have 128+64+32+16+8+4+2+1=255 ($FF).

Each character is stored sequentially in the character set as groups of eight values. The first character is stored in the first eight memory locations (0-7) in the character set, the second character is stored in the second eight memory locations (8-15), etc. The first character is called the 0 th internal character, the second the first, etc. Eight values times 128 characters requires 1024 bytes of memory. If you look in your BASIC book you will see that internal character values 128-255 produce inverse characters. These characters are the same as the ones in the internal character set except the seventh or leftmost bit of the character number is set or turned on. This is equivalent to adding 128 to the internal character number. If the high bit is set, ANTIC automatically interprets the character as inverse, and it plots it on the screen.

Since the computer can only store numbers from 0-255, all characters are assigned ATASCII (Atari ASCII) numbers. For example the letter A is assigned the ATASCII value 65. If you look at the individual characters stored internally from 0- 127 in the character set and then compare them to the ATASCII values you will find them out of order. In fact the order is as follows:

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

Essentially, Atari moved the graphics characters and placed them between the uppercase and lowercase letters. It may seem illogical at first since it complicates the calculation of the memory locations for any ATASCII character value, but it actually enables you to choose between upper and lowercase graphics in modes one and two. In both of these enlarged text modes only sixty-four characters are available, and the set is only 512 bytes long. They use the leftmost two bits of each byte to point to the color register for that character. The uppercase characters in internal positions 0-63 use color register #0. If you attempt to print lowercase characters to the screen, you still get uppercase characters but using a different color register. The letter "A" is internal character 33 while the letter "a" is internal character 97. This is equivalent to 33 + 64 or toggling one of the two high bits. The same is true for inverse. This can be demonstrated in the following program.

```
10 GRAPHICS 2+16
20 PRINT #6;"HELLO hello"
30 PRINT #6;"HELLO hello" ; REM THIS LINE IN INVERSE
100 GOTO 100
```

The result is the word HELLO printed in capital letters to the screen in four different colors. This occurs because the half-size, 512-byte character set doesn't contain any inverse or lowercase letters. The computer interprets only the lower six bits of the ASCII character as the character number and the upper two bits as the color register. Uppercase letters use color register 0 so they appear in orange. Lowercase characters are interpreted as color register I and appear in aqua. Inverse uppercase characters, which appear in the second line, use color register 2 and are blue, while inverse lowercase uses color register 3 and are light red. Remember that these colors are the computer's default values and the user can change them. The PRINT #6 prints to the graphics portion of the screen. Regular PRINT statements print to the text window, if there is one.

There is a method of obtaining lowercase letters on the screen, but if you use it you can't have uppercase characters, too. Since lowercase characters are in the upper 512 bytes of the character set, you can tell the computer that this is your new character set by changing the character set address pointer to an address that is 512 bytes higher. Location 756, the character base shadow register, normally points to the character set at $E000 and has a value for the high byte of 224 ($E0). If you add the line

```
40 POKE 756,226
```

the characters will be printed from this alternate set. They will all appear as lowercase in different colors. If you want to mix upper and lowercase letters you will need to move the character set into RAM and copy the lowercase characters into the locations occupied by the numbers and symbols.

Finding Your Character Within a Set

If you are going to move or change characters you have to be able to find them in memory by either internal or ATASCII value. Finding them by internal value is simple. Each character is 8 bytes long. The formula is;

MEMORY LOCATION = START+ 8 * INTERNAL VALUE

Finding them by ATASCII value is more difficult because the graphics characters have been sandwiched between the upper and lowercase letters. The three formulas are as follows. Please remember that the starting value of the character set defined by START should be on a 1K boundary. That value is 57344 if you are using the ROM character set.

ATASCII Value (AV)	Starting Memory Location
32-95	START+(AV-32)*8
0-31	START+(AV+64)*8
96-127	START+(AV*8)

Changing the Character Set

The easiest method of customizing a character set is to copy the ROM character set to RAM and change individual characters within it. To do this from BASIC you will need to reserve 1K or four pages of memory at the top of memory so that the set will reside in a safe place and not be wiped out by either your program or the display area of memory. The top of memory for any computer is found at decimal location 106. The actual value is PEEK(106)*256. If we move this high byte pointer down four pages there will be a new top of memory. We then POKE this value back into location 106 so that BASIC won't put anything above this. The RAM character set begins at this new top of memory. Do a graphics call after changing location 106.

We must now inform ANTIC of the new location for the character set. Location 756 ($2F4) is the character base shadow register. The value in this location is copied into the character base hardware register at location 54281 ($D409) every sixtieth of a second during the vertical blank period. We can't store this value directly in the hardware address, or it will be overwritten during the next vertical blank period.

Moving the character set requires reading a byte from ROM and storing it in the appropriate position in RAM. In our case CHROM is 57344 ($E000) and CHRAM=NMEMTOP*256. To accomplish this PEEK the values in ROM and POKE them into RAM during a FOR-NEXT loop from 0 to 1023. Once you have copied the set you can then modify specific characters and use these in PRINT statements to the screen. Or if you wish, you can alternately POKE their internal character numbers directly into screen memory. This will require a calculation to determine the correct position.

In order to show the power of a redefined character set, we will create a large Halloween pumpkin that consists of four adjacent lowercase characters, the letters a, b, c and d. The diagram below shows a magnified view of the pumpkin's lit pixels along with the screen arrangement of the four characters or letters that they represent. The letter "a" when redefined shows only the upper left quarter of our pumpkin. Similarly, the letter "b" shows only the upper right portion of our pumpkin.

Once the pixel positions in the eight rows of each character are translated into numerical data equivalent to our drawing, they are then POKEd into the appropriate position in the RAM character set as replacements to the existing characters. The four lowercase letters are internal characters 97-100. Since the four are next to each other, we can POKE in all four characters together in a FOR-NEXT loop. The starting location of the ninety-seventh character is the starting location of the RAM character set offset by 8 bytes x 97 characters. Therefore

START = NMEMTOP*256 + (8*97)

Copying the character set into RAM using a FOR-NEXT loop is quite slow and requires ten seconds. Normally, you copy the set first and then change the character set pointer at decimal location 756. 1 thought it might be more fun to watch the set being copied so I listed the program to the screen and changed the character base pointer before the copy loop. At first the listing appears fine but when the character set points to a garbage section of RAM it degenerates into random dots. As the ROM set is copied more and more of the listing becomes legible. The numbers and symbols appear first, followed by the uppercase letters. The lowercase letters form last, since they are in the last quarter of the ROM character set. If you stare at the lowercase letters in our two PRINT statements listed on lines 160 and 170, you will see them change into a pumpkin. The pumpkin will then be printed in the upper left corner of the screen. Each time that you wish to draw a pumpkin you have to use two PRINT statements on the screen. The POSITION statement can be used to PRINT the four characters that represent the pumpkin in the correct spot, but you will need a POSITION statement for each of the two lines representing the upper and lower halves.

Download PUMPKIN1.BAS (Saved BASIC)
Download / View PUMPKIN1.LST (Listed BASIC)

Many games use character set playfields as backgrounds in either BASIC mode 2 (ANTIC 7) or ANTIC mode 4. The latter allows small but detailed four-color characters four pixels wide by eight deep but requires nearly 2K of screen memory. While individual characters are limited to a single color in BASIC mode 2, they have an advantage in that these large characters don't require much memory. With twelve rows of twenty characters a screen requires only 240 bytes of memory. Each character position can be plotted in one of four colors, determined by toggling the two high bits of the internal character number.

The next example is similar to the last, except that the redefined characters are plotted as BASIC mode 2 characters. Again, space is reserved above the top of memory by moving down the top of memory pointer four pages (1024 bytes). The ROM character set is copied as before, but only half of the set, or 512 bytes, is used in this graphics mode. The shortened set doesn't contain both upper and lowercase letters. If you use the lower half of the set, you get uppercase letters only, even if you print lowercase letters. To print the four redefined lower case letters you will need to change the character set pointer to the upper half of the RAM character set. This can be

accomplished in line 100 by adding two pages to NMEMTOP and POKEing it into the character set pointer at location 756.

Download PUMPKIN2.BAS (Saved BASIC)
Download / View PUMPKIN2.LST (Listed BASIC)

The pumpkin is printed to the screen in four separate positions. By using all of the uppercase and lowercase normal and inverse combinations for the letters. a, b, c, and d, the pumpkin will appear in four different colors determined by the playfield color registers 0-3. The background is controlled by playfield 4 and can be changed by a POKE 712, COLOR VALUE. You will notice that the pumpkins are completely surrounded by yellow colored hearts. This occurs because screen memory contains zeros everywhere except where we placed pumpkins. ANTIC interpretes the Oth character in the lower half of the character set as a heart. The hearts can be removed from sight by POKING the color in playfield 0 to zero, but one of the pumpkins will also fade from sight. Therefore, it is best to create a blank character by POKEing zeros into this Oth character. To do this, add the following lines.

```
234 START=(NMEMTOP+2)*256
235 FOR I=0 TO 7
236 POKE START+I,0:NEXT I
```

Copying the ROM character set is the slowest part of the program. This operation could be speeded up a hundred or more times by substituting a Machine language subroutine that can be called by BASIC's USR function. In order to make the routine versatile, the user is given the option of either relocating the full 1024 byte ROM character set to RAM, or just the upper 512 bytes as might be needed in a Graphics 2 display. The routine is also fully relocatable and presently resides in the upper half of page six in memory. Its calling format is A= USR(1664,CHRAMH, 1 or 2). The number 1664 is the starting location of the Machine language subroutine. CHRAMH is the high byte value of the relocated RAM character set. This should be on a 1K page boundary for full sized (1024 byte) character sets and on a 1/2-K page boundary for half size (512 byte) character sets. The value one tells the routine to move all 1024 bytes of the ROM character set to the new RAM location, while a two tells the routine to skip the first 512 bytes of the ROM character set and just move the last half of the set to the new RAM location. The Graphics 0 mode pumpkin example is repeated below to show you the obvious speed advantage in using the Machine language subroutine.

Download PUMPKIN3.BAS (Saved BASIC)
Download / View PUMPKIN3.LST (Listed BASIC)

Note: Assembly language listing of subroutine in appendix.

Character Editor

Customizing a character set can be rather tedious, if you don't use a character set editor. We have furnished a simple one that you can operate by either keyboard or joystick control. The editor allows you to design single-color characters and afterwards save your custom set to a disk for later use in your own program.
The screen displays an enlarged 8 x 8 grid at the top for editing or designing your characters. The entire 128 character set is displayed below. Newly edited characters are also displayed in their proper position within the set. A character is chosen for editing by giving the internal character number. You should use the appendix in the back of this book to choose the correct character. The CTRL CLEAR key will erase the bit pattern of the present

character if you wish to design rather than modify an old character. A white square above the enlarged grid indicates that you are in the Draw mode. It can be turned off and will erase by pressing the U key for Undraw. It can be toggled back by pressing the D key. You can move the cursor without affecting anything by moving the joystick or by using the arrow keys. Each time that you want to draw a pixel you can either press the joystick button or the space bar. Since pressing the space bar for each pixel is tedious, I recommend that you edit using a joystick. When you are finished editing the character, just hit return to enter it. You can avoid entering a newly edited character by pressing the ESC key.

Once you are completely satisfied with your custom set, you can save it to the disk by pressing the S key for Save. You only need give it a file name, and it will be saved to disk. If you are re-editing a previous set, you can load one from disk using the L key for Load. You don't have to worry about loading a set to edit the first time you use the utility, because the program automatically defaults to the ROM character set which it has copied into RAM.

Download CHSETED.BAS (Saved BASIC)
Download / View CHSETED.LST (Listed BASIC)

Character Set Loader

In order to use these custom character sets you will need a Machine language routine to load your custom character set into memory from disk. The subroutine can be accessed through a USR call. The format is U=USR(CALL,IOCB, SET,LENGTH,CMD). If you are placing it into page six, CALL = 1536. IOCB = I for a read. The device # that was opened is for the file (like open #1, etc.). The set is placed at SET=CB*256 where CB is the high byte location of the character set, and LENGTH =1024. The format CMD=ADR("L") is somewhat unusual. The subroutine was designed to accept either an "L" or "S" letter command. The statement physically passes the address where the letter is and the computer looks and finds the actual letter stored at that address. The program asks for the name of the file. The "D:" doesn't need to precede the name because this string is included at the beginning of F$. The input name string is Q$. The statement F$(3)=Q$ tacks the name of the file to the "D:". The one last thing that you have to do before you call the subroutine is to open the channel to access the drive. This is in the form OPEN #IOCB, I0,0,F$. IOCB= 1, IO=4, and the 0 is unused. I won't go into details of the actual Machine language routine. The listing, however, is provided for machine language programmers; my comments accompany it.

Download CHSETLOD.BAS (Saved BASIC)
Download / View CHSETLOD.LST (Listed BASIC)

Note: Assembly language listing in appendix.

Multi-Color Characters

There are two important character graphics modes that are not supported directly by OS. Both ANTIC modes 4 and 5 use characters that are only four pixels wide instead of the usual eight pixels. Each byte is broken up into four bit pairs. The advantage is that each pixel can be of four different colors, counting the background color. Since each pixel is addressed by two bits instead of one, the value of each bit pair can determine from which color register the pixel derives its color. This becomes a very clever use of color indirection. Note that if the high bit of the character number is set, you get a fifth playfield color. The bit pair order and its associated playfield color register are as follows:

```
BIT PAIR          PLAYFIELD    COLOR REGISTER
0 0               4 (Bkd)      712
0 1               0            708
1 0               1            709
1 1 /character#   2            710
1 1 \high bit set 3            711
```

Multi-colored characters in ANTIC mode 4 can be used to create colorful and detailed playfields. They are also useful in games in which a large number of multicolored animated objects are required. While it is always tempting to use player-missile animation because it is easier, large detailed multicolored players aren't always available unless you overlap or combine your relatively few available players.
Most multi-colored ANTIC 4 shapes require a number of adjacent redefined characters since 4 x 8 pixel sized characters are tiny and don't allow for much detail. This group of characters is called a matrix.

To give you an example of the detail possible we are going to create a multicolored boat using four ANTIC mode 4 characters. The boat is 8 pixels high by 16 pixels wide and consists of three colors; yellow, red and green. The black background is actually the fourth color in our character. Each group of 4 horizontal

pixels makes up an individual character. Each column of colored pixels is then expanded into a double column bit pair. These eight columns make up a normal byte of character data. Each color pixel is then translated into its appropriate bit pairs so that they point to the proper color registers. Green pixels set the low bit of the pair, red pixels set the high bit of the pair, and yellow pixels set both bits. No bits are set when using the background color register. While the background is blank in our example, it can be set to a non-black color if the background register is part of the character shape. This occurs quite frequently if you are producing a map from a character set.

The first column of the first character of our ship contains only one red pixel in the second row. Since the left column of the character uses the two high bits of each byte, only the high bit is set in the second row. If the pixel would have been yellow, both high bits would have been set in that row. You will notice that after the appropriate bits in the bit pairs have been set for each character, the resultant pattern doesn't necessarily resemble the original image. Once you have finished the translation the final character data is then obtained by adding up all of the set bits in each byte as you would ordinarily do if they were single color characters.

It is very easy to modify a BASIC 0 (ANTIC 2) display list to produce an ANTIC 4 screen. Both text modes consist of 24 rows of 40 characters. Only the interpretation of the character data by ANTIC is different. Since both display lists are identical in length, all that has to be changed is the LMS instruction and the 23 mode line instructions that follow. The LMS instruction for an ANTIC 4 screen is 64+4 =68. This follows the three instructions that blank 8 scan lines. Therefore we need only POKE this new value into location DLIST+3 where DUST is the beginning of the display list. The following two bytes in the display list are just the address of the first byte of screen data. We

ignore these two bytes. The 23 bytes following this are mode instructions. We need only change these to the value 4 in a FOR-NEXT loop in which we go from DLIST+6 to DLIST+28

The rest of the example is essentially the same as in the pumpkin example. We copy the character set into RAM, change the character set pointer, then read in the four ANTIC 4 characters. They are placed in the 97th -100th internal character positions so that they represent the letters "a" through "d". The ship is placed on the screen by printing the string "abcd" at position 5,5.

Download ANTIC4.BAS (Saved BASIC)
Download / View ANTIC4.LST (Listed BASIC)

Designing an ANTIC 4 character set is quite laborious and prone to error when done by hand as in this example. It is best accomplished by using a character set generator like the one furnished in this book for single color characters. If you are planning to use multi-colored character sets frequently, we would suggest that you buy one of the commercial packages. The best one that we have found is Datasoft's "Graphic Generator" for $24.95. It will design character sets in all text modes including single color, two color and four color sets. It allows you to design a block of characters called a matrix on the screen at one time. The matrix can be set to any group of characters that you choose.

Character Graphics Animation

There are two methods for achieving character set animation. The easier, but least memory-efficient, method is to rotate through a series of different character sets. The characters that are printed to the screen change, if there are distinct differences between the character data for a particular character number in each of the sets. You don't have to reprint the individual characters to the screen since the animation occurs by substituting the character data from another set rather than by changing the character itself. The change requires only a single POKE to the character set pointer at decimal location 756. The disadvantages is that each set required 1K of memory. Since animation sequences require at least 4K of memory to store the various character sets. This is a waste of space unless you are using a large number of characters. In that event this method certainly becomes the best one.

Rotating Character Sets

We have written a very simple example to illustrate the simplicity of the technique. The example rotates through four different character sets. In order to create the first three from the ROM character set, we just offset them by varying degrees during the copy stage. The goal was to create different sets in which the four graphics characters, which have a small dot in four different corners, line up. These four characters are internal characters numbers 9,11, 12, and 15. If we offset the lowest set in memory by six characters of forty-eight bytes, internal character #9 from that set will line up with character #15 from the ROM set. The other two sets need to be offset by only two and three characters respectively for the graphics symbols to all line up.

```
ROM      0 1 2 3 4.5 6 7 8 9 10 11 12 13 14 15
RAM1                 0 1 2 3  4  5  6  7  8  9
RAM2               0 1 2 3 4 5  6  7  8  9 10 11
RAM3             0 1 2 3 4 5 6  7  8  9 10 11 12
```

Now when you print CHR$(15), internal character #15, the equivalent of internal character #9, will appear when the character set pointer points to the set at RAM I, and at #I I when using the RAM2 set, and at # 12 when using the RAM3 set. This arrangement also causes internal numbers 17,19,20, and 23 to line up in the various sets when you print CHR$(23).

The various sets are copies into RAM memory in an area made safe by effectively lowering the actual top of memory by twelve pages or 3K. The starting address is set at the new top of memory and the others are offset by 1K each. Since the first character set is offset by eight characters or fortyeight bytes, it is best to zero out these missing characters or garbage may appear in the 0th character, which is a blank or null character, and clutter most of the screen in one or more of the RAM character sets. Copying the sets takes thirty seconds, so be patient.

The actual animation is remarkably simple once you have printed several characters to the screen. You just cycle through the various frames by changing the character set pointer at decimal location 756 ($2F4) to each of the four different character sets. A brief delay is placed between set changes, of the animation would be too fast to see.

Download ANIMDEMO.BAS (Saved BASIC)
Download / View ANIMDEMO.LST (Listed BASIC)

Animation Using Different Characters

The second and more common method of achieving character set animation is to rewrite different characters to the screen in the same location. Of course this either requires a new print statement each time or a new character value POKEd directly into screen memory. The advantage of this method is that you can have as many variations as you like in the animation cycle without requiring many different character sets.

As a first example we have designed a simple ANTIC 4 character demonstration that places a random number of eggs on the screen. Each of these eggs begins to hatch into one of five different bug shapes before it eventually explodes. The complete life to death cycle requires thirteen animation frames, or fourteen, if you count the blank character needed to erase it at the end. The eggs, insects, and explosions consist of character pairs set side by side, and they are basically printed to the screen as character strings in a fixed position.

The program setup is quite similar to earlier examples. Space is reserved for the RAM character set by moving the top of memory downward, and a Graphics 0 display list is modified to ANTIC 4. Once the characters are read in through data statements and written into the RAM character set, they are transferred into strings to facilitate printing them quickly.

H$, BUG$, and BOOM$, contain the various character pairs for hatching, bug shape. and explosions respectively. Both the hatching and explosion animation is produced by printing two characters out of the string to the screen. This occurs in a loop in which the character pair shifts down the string until the sequence is completed. The egg-hatching string consists of the following characters;

H$ = "/0123456789:;<=>"

They are printed immediately to the screen as pairs without pause in order to have the eggs look like they are wobbling. If we examine the series of characters printed in the loop containing lines 330 and 340, they will be as follows:
1st cycle "/0"
2nd cycle "12"
3rd cycle "34"
4th cycle "56"
8th cycle "=>"

There are five random bug pairs contained in the string BUG$. BUG$ = "!"#$%&()*'". Lines 370-379 print one random bug on the screen.

The routine that steps through the character strings BOOM$ to print each of the character pairs that make up the explosion sequence is quite similar to the egghatching one. There is a little more lag, since we are only printing one set at a time. If we examine the sequence of characters printed in the loop in line 460 they are as follows:
BOOM$ = "?@ABCDEFGH"

1st cycle "?@"
2nd cycle "AB"
3rd cycle "CD"
4th cycle "EF"
5th cycle "GH"

After the bugs disappear, a caterpillar-like bug dashes across the screen. This is the same shape as the ship from our previous ANTIC 4 example in the last section of this chapter. This shape is four characters wide. The string M$ = " +,-." contains a blank as the first character so that is erases the left character of the shape as it moves rightward across the screen. Essentially we print the string on top of the previous one but shifted slightly to the left. As we repeat this pattern in a continuous loop, the result is animation.

Download BUGDEMO.BAS (Saved BASIC)
Download / View BUGDEMO.LST (Listed BASIC)

Animated Birds

The final example is one of animated birds. The animation sequence consists of four different wing positions, so that the birds appear to be flapping their wings in flight. To give the feeling that the birds are actually hovering rather than glued in place, they moved randomly about the screen.
Each of the shapes consists of an array of characters, six across by three high. Since each shape uses it takes an array of eighteen characters, each of the four animated figures are placed eighteen characters apart in the

character set.

An animation sequence like 0-1-2-3-0-1 ... which we would call circular, would produce a discontinuity in the motion. The bird's wings are moving downward during the four cycles but if we went back to the 0th frame again, the bird's wings would suddenly be at the top. Therefore, the motion must be oscillatory so that the wings begin moving upward after reaching the bottom. The sequence is 0-1-2-3-2-1-0-1-2.... Line 40 in the program accomplishes this.

Printing a matrix of characters is usually tedious using normal print statements. If there is more than one row of characters involved, you will need several print and position statements. Certainly, an easier method would be to develop a Machine language subroutine in which you only had to specify the first character in the matrix, the height and depth of the matrix, and the position on the screen that you wish to print it.
The subroutine is called MCYCLER and it is stored in this example as a string, Since some users may prefer to POKE it into page six or even somewhere else, it is written as fully relocatable code. It is called with a USR statement. X = USR(MCYCLER, ROW, COL, MATRIXH, MATRIXV, STARTING CHARACTER #). MATRIXH and MATRIXV are just the horizontal and vertical size of the matrix. In the bird example they are six and three respectively. MCYCLER is the starting address of the subroutine, but since it is stored as a string, the value is ADR (MCYCLER$). The subroutine will print the matrix to the screen, except when the starting character # is zero. This feature is needed to erase the previous matrix in its last position before you draw the next frame.

Erasing one frame and drawing the next is accomplished in lines 80 and 90. The old location of the matrix is at BX (L1), BY (L1), and the new location is at NX (L1), NY (L2). Notice that the starting character during the draw stage depends on which frame, B(L1), we are drawing, and that each matrix is eighteen characters apart in the set. The value of one is added since the Oth character is actually a blank or null character. After the subroutine draws the matrix, the old location is transferred to the new location. This is necessary since the last figure drawn must be erased from its old position BX(L1), BY(L1) before the next one is drawn at a new random position NX(L1), NY(L1).
We thought it might be instructive to see which internal characters are actually being printed to the screen. You can toggle between the ANTIC 4 bird set in RAM and the normal ROM set by pressing the SELECT key. The OPTION key resets it.

Download BIRDDEMO.BAS (Saved BASIC)
Download / View BIRDDEMO.LST (Listed BASIC)

Chapter 4
Assembly Language Applied To Game Design

The words, Machine language and/or Assembly language, evoke visions of indecipherable code to the novice BASIC programmer. The code looks unfamiliar. But so was BASIC when you were first learning it. While BASIC has its roots in the English Language and algebraic expressions, Assembly language appears to consist of unfamiliar op codes or mnemonics that are used in conjunction with an unfamiliar base 16 number system called hexadecimal.

It is our intent in this chapter to teach you the fundamentals of Assembly language programming by comparing it to similar code written in BASIC. Rather than teach you all aspects of the language, we will concentrate only on the operations needed to do simple game graphics.

A good Assembler is needed to write Assembly language programs. An assembler merely translates mnemonics like JMP, which is equivalent to a GOTO, into hexadecimal opcodes that the computer understands. Most Assemblers have an editor, an Assembler, and a debugger. The editor allows you to enter Assembly language code usually by line number and later edit, delete, or insert particular lines. The Assembler portion converts your source listing into Machine Code in a two-pass operation. Since any line of code can have a label in its first field, the Assembler will automatically calculate the branches or GOTOs to lines referenced with these labels. Also, if you want to store a variable called ZAP, the Assembler which assigns a memory storage location for the variable will automatically furnish the correct memory address for any subsequent store or load operations using that variable. Last, there is a Machine language monitor or debugger that helps locate errors. It allows you to examine and change both memory and internal registers. It also includes step and trace features that allow you to step through your code one instruction at a time.

Readers who already own assemblers may use the one they have. We have provided a translation table in the Appendix in the back of this book to aid you in converting our SYNASSEMBLER source code to that used in your assembler. We chose SYNASSEMBLER when we began this book in the Spring of 1983 because it was co-resident (screen editor, assembler, and debugger are in memory simultaneously) and was available in cartridge form.

For those of you who are new programmers, or are unhappy with their present assembler, we recommend either the *F-S Macro Assembler 40/80* from Stanton Products (See coupon in back of book), an enhanced disk version of the now discontinued *SYNASSEMBLER*, or *MAC 65* from Optimized Systems Software. Both of these assemblers are fast (2000 lines/minute), are co-resident assemblers, allow source files to be chained, and offer a choice of assembling to either disk or memory. Both of these are professional packages and are used as development tools in various software houses. The F-S Macro Assembler 40/80 and the discontinued SYNASSEMBLER are both derived from the S-C family of assemblers on the Apple II computer. Whereas the new F-S Macro Assembler 40/80 is compatible with the new XL series of computers, unpatched versions of SYNASSEMBLER are not. The F-S Macro Assembler 40/80 is completely compatible with SYNASSEMBLER source files with the exception of the way it handles ATASCII string data. A simple global replace will suffice. (see note in Appendix on assemblers differences.)

Our readers will certainly want to know why we don't use the more popular *Atari Editor Assembler* cartridge. First, it is very, very slow, often taking ten minutes to assemble a 1000 line program. Second, it doesn't allow chaining of files, nor assembly to the disk. Third, it is full of bugs. It remains popular mostly to beginner programmers who want to try to write a very short Assembly language subroutine that will interface to their BASIC programs.

Basic Assembly Language

The Atari computers contain a central processing unit (CPU), a 6502A microprocessor that operates at 1. 8 Mhz. It accepts instructions to perform various operations, like taking a value and storing it somewhere in memory, adding

a number to another number located in one of its internal registers, or comparing two values. What makes programming in Assembly language rather difficult (or at least tedious) is that the computer can only execute one tiny instruction at a time, and only perform its operations in three internal registers. These three addressable registers are known as the X register, Y register, and Accumulator. Each can hold eight binary digits called bits, which are individually valued at 0 or 1. The eight bits, collectively called a byte, have values ranging from 0 to 255 decimal or ($00 to $FF in hexadecimal notation).

Essentially, the computer, which is an eight-bit microprocessor, can manipulate data whose values range from all eight bits off (00000000) to all eight bits on (11111111). The average person has great difficulty in thinking of values represented by 0's and 1's. Fortunately, someone invented a number system called hexadecimal, which is base 16 instead of binary or base 2.

Hexadecimal Numbers

Since 16 is 2x2x2x2, we can divide our eight bits into two four-bit groups. If you determine each of the decimal equivalents of all the combinations of base two representations, you obtain the following table. These values range from 0 to 15 decimal. In the hexadecimal numbering system, values above 9 are represented by the letters A-F. In order to prevent confusion between decimal and hexadecimal numbers, hexadecimal numbers are preceded by a "$".

```
BINARY    DECIMAL  HEXADECIMAL
 0000        0         $0
 0001        1         $1
 0010        2         $2
 0011        3         $3
 0100        4         $4
 0101        5         $5
 0110        6         $6
 0111        7         $7
 1000        8         $8
 1001        9         $9
 1010       10         $A
 1011       11         $B
 1100       12         $C
 1101       13         $D
 1110       14         $E
 1111       15         $F
```

Hexadecimal numbers are very much like decimal numbers. They can be added and subtracted in like manner. The only difference is that instead of having units, tens, hundreds, etc, the hexadecimal numbers have units, sixteens, 256's, and so forth. Each successive digit is sixteen times the position to the right instead of ten times as in our decimal system.

```
DECIMAL        HEXADECIMAL
1 6 5          $ 1 3 A

1 HUNDRED        1-256
  6 TENS           3 SIXTEENS
     5 ONES           A ONES

  1 x (100) = 100     1 x (256) = 256
+ 6 x (10) = 60      +3 x ( 16) = 48
+ 5 x( 1) = 5        +A x (  1) = 10

          165 DECIMAL             $13A = 312 DECIMAL
```

Hexadecimal numbers are used to address the Atari's 48000+ memory locations. Each group of 256 bytes ($00 - $FF) is called a page, starting with page zero. In 48K Atari computers, memory is directly addressable from locations $0000 to $BFFF (0 -49151). Locations above $BFFF are also addressable, but these locations don't contain RAM. The area from $D000 to $D7FF contain custom hardware chips such as the GTIA, POKEY, PIA, and ANTIC microprocessor. Some of these hardware locations can be read and some written to. The area above that, $D800 to $FFFF, contain the 10K operating system ROMS.

Memory Considerations in Assembly Language

The bottom of RAM, pages 0 thru 5 ($0000 - $05FF) are generally off limits for program storage. Zero page ($00 - $FF) is a very special area. There are a number of zero page addressing instructions that execute faster because

they require only two instructions instead of the usual three. This is because they only need to address a memory location from $00 to $FF instead of $0000 to $FFFF. These locations are used extensively by the Operating System.

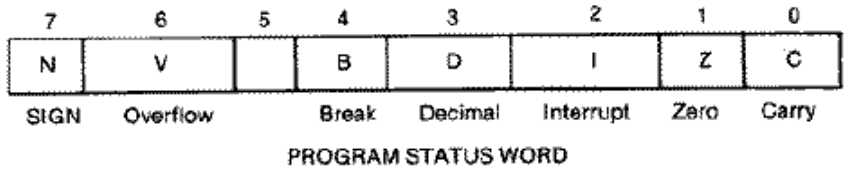
Only the last few bytes of zero page are available to the user. In fact, if you are using Synassembler only locations $F0 - $FF are totally free. You can also use $D6 - $EF, if you don't mind if your data is altered by the floating point package each time arithmetic operations are performed by BASIC. And if you are writing a subroutine to be accessed from BASIC, only locations $CB - $D1 (203-209) are available. $D4 and $D5 can be used to send variables back to BASIC via the USR function.

Page one of memory ($100 - $1FF) is reserved for the stack. It is used by a special purpose register in the 6502A microprocessor for keeping track of return addresses when calling subroutines. This scratch area for the Stack Pointer is sometimes used for temporary register Storage.

Pages two and three are used for various I/0 operations, and operating system shadow registers, page four for the cassette buffer, page five for the keyboard buffer, and pages seven through twenty-eight for DOS 2.0's file management system. Essentially the area below 7420 ($1CFC), with the exception of page six, is off limits to programmers using DOS. However, if DOS isn't resident, you can begin storing at 1792 ($700) safely. A pointer to the low end of memory, MEMLO, can be read at locations 743,744 ($2F7,2F8).

Program Counter & Program Status Word

When a microprocessor processes a Machine language program, it keeps track of which instruction it is executing with an internal 16-bit register called the program counter. The program counter contains the current address of the instruction that is being processed. When the computer finishes with an instruction, it sets a flag or condition in a 7-bit, Program Status Word, which is another register. For example, if you want to test if a value in the Accumulator is equal to zero, you compare the value in the Accumulator to zero. If this value is equal to zero, the zero flag will be set and the next instruction, Branch Equal to Zero (BEQ), will be executed. Other flags that can be set are the carry flag, and the negative flag. A diagram of the Program Status Word is shown below.



PROGRAM STATUS WORD

OP Codes

The 6502A microprocessor accepts only Machine language instructions. These are called opcodes. When the computer encounters a $4C, it performs an equivalent to a GOTO in BASIC. The Machine language instruction $4C 00 08 tells the computer to jump to memory location $800. (Remember, addresses require two bytes. The low order byte in this case contains $00 and the high order byte, $08--in effect, the reverse order of the actual values.) Unfortunately, Machine language is difficult to remember, so programmers invented a substitute called Assembly language, wherein each opcode is assigned a mnemonic such as JMP, BRK, or LDA. The above example looks like this: JMP $0800.

If you were to type the following Machine Code into the monitor in your Assembler, you would see how the monitor disassambler interprets the code, as in the following example:

4000: A9 30 8D 00 41 CE 00 41 AD 00 41 C9 00 D0 F6 60 [CR]

If you enter a 4000L from the Synassembler monitor you will see the following:

```
4000: A9 30    00030      LDA #$30
4002: 8D 00 41 00040      STA $4100
4005: CE 00 41 00050      DEC $4100
4008: AD 00 41 00060      LDA $4100
400B: C9 00    00070      CMP #$00
400D: DO F6    00080      BNE $4005
400F: 60       00090      RTS
```

The disassembler translates the Machine Code to more easily understood mnemonics. In the first line of code, LDA is the mnemonic for Load Accumulator. It is the instruction for the 6502 to load the Accumulator with an immediate value--in this case, $30. The # sign signifies that it is an "immediate" instruction; the ($30) is the data

portion of the instruction. The STA in line two is an "absolute" instruction. It specifies the address in memory for storing the byte of data that is in the Accumulator.

The difference between "immediate" and "absolute" instructions is an important point. Let us take the example LDA #$30. In this "immediate" instruction, the computer takes the operand ($30) as a value and places it in the Accumulator. However, LDA $30 is an "absolute" instruction, so the computer takes the operand as an address from which to load data into the Accumulator. In both cases, we get a value in the Accumulator. You can tell the modes apart because "immediate" instructions have a # sign before the operand.

You might wonder, what does this code do? It is a time delay subroutine. It puts a decimal 48 in memory location $4100. Line two stores it there, then the value stored at that memory location is decremented by one in line three. It is then reloaded into the Accumulator to be compared against the value zero. If it is zero it falls through to the return-from-subroutine instruction and ends; but if it isn't zero it branches back to memory location $4005. That location tells the computer to decrement the value in $4100 once again. The code will perform this small loop until the value in $4100 becomes zero. At that time, the test for a zero becomes true and the program returns to the line after the JSR in the program that called it.

Does it work? First type 400E:00 to change the RTS to a BRK. This will return us to the monitor when we are finished. Then type 4100:AA to place something in that memory location so that if you look at it later you will believe the program did something. Finally, do type 4000G to start the routine. The code returns you back to the monitor when it finishes a split second later. Now type 4100 and a 00 is returned. This is the value in memory location $4100. You can do a 4000S and an S each time to watch the code single step, or you can trace the entire operation by typing a 4000T . The strange numbers that appear below each line of code are the values in the internal registers. A is for Accumulator, X for X register, Y for Y register, P is the Program Status Word, and S is the Stack Pointer.

This program has a direct analogy to the following BASIC program:

```
10 X=48
20 X=X-1
30 IF X<>0 THEN 20
40 RETURN
```

The major differences between the two programs is that in Assembly language there are no line numbers used within the code (line numbers are used only by the editor to place your text in order, and you have to take care of every minute detail. BASIC automatically assigns the storage locations of all variables and the location of each instruction in memory. In Assembly language programming, we have to assign the X variable to memory location $4100, and have to calculate the relative branch or GOTO so that it references the memory location $4005. This is done by branching back $F6 bytes or -8 bytes to the proper address. Yet many of these details can be greatly simplified if we use an Assembler to do our programming.

The same program using an Assembler looks like the following:

```
LINE LABEL INSTRUCTION COMMENT
         FIELD FIELD    FIELD

00010          .OR $4000  ;ASSEMBLE CODE AT $4000
00020 X        .EQ $4100  ;X IS STORED AT $4100
00030          LDA #$30
00040          STA X
00050 LOOP     DEC X      ;X=X-l
00060          LDA X
00070          CMP #$00   ;DONE?
00080          BNE LOOP
00090          RTS
```

The Assembler generates identical Machine Code, but many of the tedious details are simplified. Once X is equated to the memory location in line 2, references to that variable in lines 4 through 6 are handled automatically. If X were assigned to a different memory location because we lengthened our program, you would only have to change line 2. Also, labels act like line numbers in BASIC. Since the Assembler assigns the line of code labeled LOOP to a particular memory location, it can calculate the correct branch automatically when it encounters line 8 during assembly. The .OR in line 1 is a pseudo-op, understood only by the Assembler. This does not generate code but tells the Assembler where the code is to be run and stored. The pseudo-op .TF causes the generated code to be stored to the disk rather than to memory.

Addressing Modes

Now that you have had a taste of Assembly language programming and have seen that it isn't as bad as you thought, there are a number of fundamental operations that must be learned. The most important operation is to move numbers from one memory location to another. This can be accomplished by loading a value into any one of three internal 6502 registers--the Accumulator, X, or Y registers--and storing that number somewhere in memory. A LDA (Load Accumulator) instruction can be carried out in several different ways depending on its addressing mode. First we can load the Accumulator with a real hexadecimal value (LDA #$05). This is called Immediate Mode Addressing. Sometimes we need to be able to load the Accumulator with a variable stored in a memory location (LDA $4100). This is called Absolute Addressing.

The only other addressing method that we will discuss for the time being is the Indexed Addressing mode. It takes the form of LDA $4100,X or LDA $4100,Y depending on whether the X or Y register is used as an index. If, for example, the X register contains a #$05, then the instruction above loads the value from location $4100 + $05 or $4105. This addressing mode is used primarily for indexing into tables stored at particular memory locations. There is no problem with the tables crossing page bounderies. For example, if your table began at $4080 and the X-register contained a $90, then the instruction LDA $4080,X would fetch the value in memory location $4080 + $90 or $4110.

EFFECTIVE ADDRESS = ABSOLUTE ADDRESS + X

EFFECTIVE ADDRESS = ABSOLUTE ADDRESS + Y

Store operations are similar to load operations. You can store a value into an "absolute" memory location, or you can store indirectly into a memory location, offset by the value contained in either the X or Y register.

In summary, the table below shows the various load and store operations.

```
        ACCUMULATOR     X REGISTER      Y REGISTER

LOAD    LDA #$05        LDX #$05        LDY #305
        LDA $4100       LDX $4100       LDY $4100
        LDA $4100,X     LDY $4100,X
        LDA $4100,Y     LDX $4100,Y

STORE   STA $4100       STX $4100       STY $4100
        STA $4100,X                     STY $4100,X *
        STA $4100,Y     STX $4100,Y *
```

*Both indirect operations involve zero page addressing only.

Incrementing & Decrementing

Sometimes it is necessary when counting cycles, or looping through code to increment or decrement a value directly similar to a FOR-NEXT loop in BASIC. In Assembly language, either the X and Y registers or any memory location can be incremented or decremented. If the X register contained a $FE, then it would contain $FF when incremented. But if it contained a $FF, it would wrap around to become $00. The computer informs you by setting a zero flag in its Program Status Register.

```
                ACCUMULATOR     X-REG   Y-REG   MEMORY LOCATION

INC BY 1        NOT AVAILABLE   INX     INY     INC $4100
DEC BY 1        NOT AVAILABLE   DEX     DEY     DEC $4100
```

Stack Instructions

There is a special area in the computer ($100 - $1FF) that is used quite frequently by an internal register called the Stack Pointer. The computer uses this area to save return addresses when handling either interrupts or subroutines. The stack is like a dish dispenser. Bytes are pushed on the stack in order, and pulled off in reverse order. The first byte stored is the last byte to be pulled off. The Stack Pointer always points to the next free byte in the stack. Since the stack is only 256 bytes long, only 128 address pairs can be stored at any one time.

Normally the stack would be of little interest to programmers except that it can also be used to temporarily store data. If you were worried about your three registers being altered in a subroutine, you could push all three values

onto the stack before calling the subroutine, and then pull them back off when you return from the subroutine. BASIC also uses the stack to transfer data in the USR function when calling a Machine language subroutine. The top byte in the stack contains the number of variables being passed. The values follow in two byte pairs in hi byte low byte order.

Two basic Machine language instructions provide key tools for using the stack. PHA pushes the value in the Accumulator on the Stack. PLA pulls the top value of the stack and places it in the Accumulator. Since these instructions only involve the Accumulator, you would need to transfer the value in the X register to the Accumulator (TXA) in order to save the X register on the stack. Similarly you would transfer the Y register to the Accumulator (TYA) first before a PHA to the stack. Be careful when working with the stack. For instance, if you push data onto the stack while in a subroutine and don't pull it back off, when the subroutine reaches the RTS instruction it will return to the main program at the wrong address.

Altering Program Flow

Program flow can be altered, as in BASIC, with instructions that resemble GOTO, GOSUB, and IF ... THEN statements. The JMP instruction is equivalent to a GOTO statement; it can transfer control to any location in the machine to continue executing code. JMP $8D6C instructs the computer to continue executing code beginning at address $8D6C. The GOSUB statement is identical to a JSR (jump Subroutine) in Machine language. When the computer reaches the instruction $5A83, it pushes the two-byte memory address of the instruction onto the stack, so that when it returns from the subroutine via an RTS (ReTurn from Subroutine), it will know the address where it will continue the program. When it returns, it pulls the return address off the stack and increments it by one so that it points to the next executable instruction.

The IF ... THEN statement is analogous to a number of branch instructions which test the Program Status Register to see which flags are set. Usually, you use compare operations to set flags. You can compare a value against the value stored in either the Accumulator, the X or the Y Registers. The mnemonics are CMP, CPX and CPY, respectively. For example,

```
        LDA $4100 ;LOAD ACCUMULATOR WITH VALUE AT $4100
        CMP #$05
```

Different flags are set depending on the result.

Branch instructions are very similar to a JMP instruction (which is an unconditional branch), except that only under certain circumstances will they cause program flow to continue at a different location. For example, if we were to test for that wraparound case when we incremented the X-register that contained $FF, we would want to test the Zero Flag with a Branch Equal Zero (BEQ) instruction, and go to some label if the condition is true.

```
            LDX $4100    ;LOAD X REGISTER WITH VALUE IN MEMORY
            INX          ;INCREMENT X - REGISTER
            BEQ SKIP     ;TEST IF 0, AND IF TRUE GOTO SKIP
            RTS          ;RETURN TO MAIN PROGRAM
SKIP        LDA #$04
             .   .
             .   .
```

This short example loads a value from the memory location into the X register, then increments it. If wraparound occurs, the test for a zero flag causes the program to jump to a label called SKIP, and the code does not return to the program that called it via the RTS. There are numerous tests on each of the flags in the Program Status Register. A summary is shown below.

```
BCS     -       Branch if the carry flag is set.            C = 1
BCC     -       Branch if the carry flag is clear.          C = 0
BEQ     -       Branch if the zero flag is set.             Z = 1
BNE     -       Branch if the zero flag is clear.           Z = 0
BMI     -       Branch if minus.                            N = 1
BPL     -       Branch if plus.                             N = 0
BVS     -       Branch if overflow is set.                  V = 1
BVC     -       Branch if overflow is clear.                V = 0
```

Most Assemblers offer alternative mnemonics for BCC and BCS. Since, during comparisons, the carry flag is set when the value in the appropriate register is equal or greater than the value compared, BCS might be called BGE (Branch Greater or Equal). Likewise, BCC is equivalent to BLT (Branch Less Than). Why use these alternatives? Because they are easier to remember and visualize, and they make it clear that you are doing logical comparisons, rather than testing the results of an addition or subtraction.

There is one other important concept that should be understood when doing comparisons. I implied that the subsequent branch was like a GOTO in BASIC or like a JMP in Assembly language. This is not entirely true, since the range of the branch cannot exceed -126 to +129 bytes. This is because the branch instruction is only two bytes long. The first byte is the instruction code and the second the relative address. It takes a two byte address to branch to any place in memory (Except Page Zero). The JMP instruction has the advantage that it is three bytes long. In most cases, this limitation will not cause problems. But if a "branch out of range error" occurs, you must reverse the test so that it will reach the required destination via a JMP instruction.

Example: If BEQ SKIP is out of range then substitute the following:

```
BNE *+$5     or     BNE B
JMP SKIP            JMP SKIP
.                   B NOP
.                   .
```

This change causes the program to drop through the JMP instruction if the zero flag was set, and then jump to location SKIP. However, if the zero flag is not set, it will advance ahead five bytes to the instruction following the JMP. All other branch instructions work in a similar manner. This gives the equivalent of a Long Branch.

Addition & Subtraction

Simple addition and subtraction of unsigned numbers is easily accomplished in Machine language. All additions and subtractions must be performed one byte at a time. Thus, large numbers or multi-byte numbers (those that exceed $FF), must be added or subtracted one byte at a time, and the carry flag must be accounted for. It's actually not much different from addition of two multi-digit decimal numbers. Those numbers have a digit in the ones column, another in the tens, etc. If you add 65 to 78, you add the ones column first. Five plus eight equals 13. The value in the ones column is 3; you then carry the one "ten" into the tens digit column before you add the two numbers in the tens column. Hexadecimal addition is similar. You clear the carry before you add. If the sum of the two values exceeds $FF, the carry is set. Since you don't clear the carry when adding the next higher byte, the resultant answer will be the sum plus the previously computed carry, as in the following example:

```
EXAMPLE:        +CARRY
                    63    F4
                   +02   +16
                   ---   ---
                    66    0A ;SETS CARRY
```

The code for addition and subtractions is as follows:

ADDITIONS

```
CLC               ;CLEAR CARRY
LDA #$F4          ;LOAD LOW ORDER BYTE
ADC #$16          ;ADD WITH CARRY
STA LOW           ;STORE LOW BYTE
LDA #$63          ;LOAD HIGH ORDER BYTE
ADC #$02          ;ADD WITH CARRY (NOTE DON'T CLEAR CARRY)
STA HIGH          ;STORE HIGH BYTE
```

SUBTRACTIONS

```
SEC               ;SET CARRY FLAG
LDA #$F4          ;LOAD VALUE
SBC #$16          ;SUBTRACT WITH CARRY
STA VALUE         ;STORE RESULT
```

You should be aware that the rules for subtraction are different from the ones for addition. The carry must be set first. This is equivalent to a borrow in subtraction. After the subtraction operation, the carry will be clear if an underflow (borrow) occurred. The carry will be set otherwise. Setting the carry is very important, a step that many beginners forget. The results are invariably incorrect if this step is skipped--and possibly even "random," since the status of the carry flag can be on or off when the subtraction operation is performed. This can make debugging difficult.

Breakout Game (BASIC)

The "Breakout" game involves the simplest animation technique available on the Atari, moving individual pixels from one position to a new position. We have a Graphics 5 pixel-sized ball that bounces around the screen. It will ricochet off a movable paddle, the walls, or any of the 2 pixel-high by 5 pixel-wide colored bricks. Movement is accomplished by erasing the ball at its old position and redrawing it at its new position. The ball is very predictable. It changes direction only upon collision, and in all cases (except contact with the paddle) simply reverses direction. The point of contact with the joystick-controlled paddle determines the ball's direction. Balls striking the left end travel upwards and to the left at a 45 degree angle, while balls striking the inside left travel in the same direction but at a 60 degree angle. Balls striking the paddle's right side travel at similar angles, but to the right.

Once you have the design description, in this case a game that is an old classic, the next step is to translate it into a logical sequence of events and their consequences. This can best be accomplished by drawing a flow chart that shows the possible pathways for each module in the program. Each of these modules can be as small as a single statement, or can consist of entire subroutines. No matter how detailed or general you make it, the flowchart must accurately represent the game's logic. While it is a good tool for learning to think logically, a flowchart isn't necessary or required in all cases. Many good programmers have never drawn one. They obviously have the ability to flowchart unconsciously in their minds.

The game should be programmed in small steps rather than as a complete entity. This way you get to see results early. Besides, it is easier to debug a small section, such as the ball bouncing off the paddle and moving around the screen, than to attempt to debug a complete program that is full of errors. The most successful programmer will be one who can debug by watching what goes wrong on the screen.

Paddle Position

Determining where the ball strikes the paddle is easy in our "Breakout" game. The paddle is always drawn two-pixels wide at row 36 decimal or $24, and the first pixel begins at PX, a variable controlled indirectly by the joystick. Actually the new paddle position is P = P + D where D depends on the direction of movement and whether the button is being pressed. If the joystick is pushed to the left, D=-1, while if it is pushed to the right, D=1. When the button is pushed, D=D*3, and the paddle moves at triple speed. The Boolean logic in line 230, ((P+D)>0 AND P+D<76) gives a value of true=1 or false=0 depending on whether the paddle has exceeded the screen bounds after movement. If it hasn't, the result is P=P+D*(1), and there is a new paddle position. If it has, the result is P=P+D(0) and the paddle remains stationary.

Ball's Position & Velocity

It is easy to compare the ball's new vertical position NX to that of the paddle's leftmost position PX. The difference NX-CX is C. You can use this value to index into a table to obtain the new horizontal velocity; DX = C(C). These values vary with position. The two outside blocks give a DX of + 1 or 1, and the two inside blocks give a DX of

+1/2 or -1/2. The vertical velocity, DY is equal to -1 since the ball is always travelling upwards after striking the paddle.

In order to update the ball's position, we take the old ball's position and add the change in position or its directional velocity. The format is:

NEW POSITION = OLD POSITION + CHANGE IN POSITION

NX = BX + DX
NY = BY + DY

Incrementing or decrementing the ball's position by 1/2 in the X direction is not physically possible since screen positions are whole numbers. The ball's position is truncated to the nearest integer value with the INT function. The result is that the ball remains stationary in the X direction during one frame, then moves one whole pixel position during the next frame or cycle.

Collisions with Bricks

As the ball bounces around the screen it will soon collide with one of the colored 2 by 5 pixel-sized bricks at the top of the screen. It is possible to test for a collision by using the LOCATE function. This function, which returns the color register at the ball's position, works only in BASIC Graphics modes 3-8. Non-zero values in this example indicate a collision with one of the three colored bricks (Playfields #1-3).

If there is a collision, the correct block needs to be removed. This is quite simple to calculate for the X direction:

C INT(NX/5)*5

You still need to determine if the ball hit the brick in an even or odd pixel row. It might appear that the ball would always collide with the bottom or odd row of pixels first, but if there are gaps between bricks as occurs later in the game, the ball can approach from the side and strike the brick along the top or even row of pixels. If the ball strikes the bottom row, you will need to adjust the position to the brick's top row in order to erase one complete brick. The test is a very simple Boolean function in line 320. For example if the ball's new vertical position, NY=9, then NY/2 <> INT(NY/2) would reduce to 9/2 <> 4 which is true. We would then decrement NY to an even number in order to erase the complete block. The top left corner of the 2 pixel by 5 pixel brick is C,NY. Five pixels are erased from C,NY to C+4,NY in each of its two rows.

The brick's score depends on its playfield color. SCORE = SCORE + SCORE(C), where C is the value returned by the locate function. The yellow (playfield #1) bricks at the top are worth ten points, the green (playfield #2) bricks in the middle are worth five points, and the blue (playfield #3) bricks at the bottom are worth only three points.

The ball's vertical direction of travel reverses upon collision with a brick. It continues in the horizontal direction until it reaches either the left or right playfield boundary at BX=0 or BX=79. It reverses direction there so that DX = -DX. If the ball reaches the top of the playfield at BY = 0, it will reverse its vertical direction. But if the ball reaches the bottom it is lost and we begin again with a new ball. The game will end when we have run out of either bricks or balls.

Download BREAKOUT.BAS (Saved BASIC)
Download / View BREAKOUT.LST (Listed BASIC)

Breakout Game (Assembly Language)

The "Breakout" game is quite easy to translate into Assembly language once you understand how BASIC handles its graphics commands. The Operating System (OS) implements each of these commands through the CIO (Central Input/Output) subroutine located at $E456. When a program calls the OS through this location, the OS expects to be given the address of a properly formatted IOCB (Input Output Control Block). There are eight of these, each sixteen bytes long. These are located from $340 to $3BF. The appropriate IOCB number times 16 is passed to the subroutine in the X-register. The full details of how the internals actually work are really not important, especially to the beginning Assembly language programmer. Let's just say that we developed a set of graphics subroutines that mirror their BASIC language counterpart. We have commented on each of these in the listing for anyone who would like to study them.

Graphics Commands

The five graphics commands that we need for our game are: GRAPHICS #, POSITION H,V; PLOT H,V; DRAWTO H,V; and LOCATE H,V,Color. We set up each by inputting certain parameters into the Accumulator, X-register, and Yregister. Once you've set up the registers you need only JSR to that subroutine. The table below shows what you need to input into each of the registers.

```
Function        Accumulator     X-register      Y-register

GRAPHICS        Mode #          -------         -------
POSITION        Vertical        Horizontal      Horizontal
                                High byte       Low byte
PLOT            Vertical        Horizontal      Horizontal
                                High byte       Low byte
DRAWTO          Vertical        Horizontal      Horizontal
                                High byte       Low byte
LOCATE          Vertical        Horizontal      Horizontal
                                High byte       Low byte
                Has color
                value on
                return.
```

For example, if we wish to set up a Graphics 5 screen and draw a blue (playfield #3 default color) line from 10, 15 to 30,15 our program would be as follows:

```
LDA #$05        ;GRAPHICS 5 SCREEN
JSR GRAPHICS
LDA #$03        ;PLAYFIELD #3
STA COLOR
LDA #$0F        ;VERTICAL=15
LDX #$00        ;HORIZONTAL HIGH BYTE
LDY #$0A        ;HORIZONTAL LOW BYTE
JSR PLOT        ;PLOT PIXEL
LDA #$0F        ;VERTICAL=15
LDX 000         ;HORIZONTAL HIGH BYTE
LDY #$IE        ;HORIZONTAL LOW BYTE
JSR DRAWTO      ;DRAW LINE
```

Breakout Game

Once you understand the simplicity of duplicating the BASIC graphics statements in Assembly language you can proceed with developing the game.

The "Breakout" game is a very close translation of the BASIC version with a few subtle differences. One of the problems in working with Assembly language is that all numbers are whole integer numbers. In the BASIC version the ball's horizontal direction (DX) became +1/2 or -1/2 when it hit the inner portion of the paddle. Since incrementing the ball's position by +1/2 would be impossible in Assembly language, DX and BALLX, a temporary value for the ball's horizontal position, are doubled in value. If we then divide BALLX by two before plotting the ball's true position, TX, the fractional part, will vanish. In essence the ball will move horizontally every other frame.

BALLX = BALLX + DX (doubled values)
TX = BALLX / 2

ASL and LSR Instructions

Multiplication and division by powers of two is easy in Machine language. The mnemonic ASL is used for multiplication by two. The Arithmetic Shift Left (ASL) instruction shifts all of the bits in the Accumulator one position to the left. Thus, bit 0 is shifted into bit 1, bit 1 into bit 2, etc. Bit 7 is shifted into the carry bit so that you can use the BCC and BCS instructions to test for overflows. For example, if only bit 2 was on (4 decimal) and we did an ASL, the bit would be shifted to bit 3 (8 decimal). Thus, it is easy to multiply by powers of two by performing repeated ASL instructions.

Conversely, division is performed by the Logical Shift Right (LSR) instruction. Bits are shifted to the right and the bit 0 is shifted into the carry. This is equivalent to dividing by two with loss of the fractional part.

```
       LDA #$05    ;LOAD ACCUMULATOR WITH 5
       LSR         ;DIVIDE BY 2
       STA $4000   ;VALUE STORED IN $4000 IS 2
```

Ball's Direction After Paddle Collision

The table of directional values for the four possible collision positions with the paddle are stored in VX. The two negative values in the table are stored in their two's complement form because it is easier to add two positive numbers rather than to test for a negative number and subtract.

```
                              0th   1st   2nd    3rd
                        VX    $FE   $FF   $01    $02
```

For example, #$FE (-2)+ #$03 = #$01. The offset position from the paddle's left edge is placed in the X register to get the new horizontal velocity.

```
LDA TX       ;COMPARE PADDLE HORIZ. WITH BALL HORIZ.
SBC PX       ;DIFFERENCE
TAX
LDA VX,X     ;FETCH VELOCITY VALUE FROM TABLE
STA DX       ;THIS IS DOUBLED VALUE
```

We calculate the ball's new position as follows;

```
CLC
LDA BALLX    ;OLD BALL POSITION (DOUBLED)
ADC DX       ;NEW HORIZ. VELOCITY DOUBLED
STA BALLX
LSR          ;DIVIDE BY 2
STA TX       ;BALL'S TRUE HORIZ. POSITION
```

Scorekeeping

The scorekeeping routine also deserves an explanation. It differs substantially from the routines used in the other Machine language games in this book. It takes advantage of the 6502's ability to work in a numbering system called Binary Coded Decimal, or BCD. This system uses the lower four bits or low-order nibble to represent the low-order decimal digit, and the high-order nibble to represent the high-order decimal digit. The advantage is that the numbering system resembles decimal. The disadvantage is that it requires some advanced programming technique to isolate the digits in order to print them to the screen.

```
  DECIMAL     BINARY      HEX
                         (BCD)

     07      0000 0111     $07
     10      0001 0000     $10
     16      0001 0110     $16
     42      0100 0010     $42
```

To get to this mode you must set the decimal flag with a SED (Set Decimal Mode) command. It remains in effect until it is cleared by a CLD (Clear Decimal Mode) command.

A pair of bytes, SCORE and SCORE+ 1 are used to store the four score digits. These are updated by adding POINTS,X to SCORE+ 1 each time a brick is removed. The X-register contains the color value of the block hit so that we need only index into a table of point values. We didn't clear the carry when we added #$00 to SCORE (highbyte). However, if there was an overflow in SCORE+1 (low byte) during the first addition, the carry would be included in the resulting value in SCORE. Each of the four nibbles must be separated, translated into an internal character #, and finally placed into the appropriate position in the text window. The byte's high nibble is first shifted to the low nibble by four successive LSR instructions and then translated into an internal character number. Digits in the internal character set begin at #$10. Internal character #16 decimal = 0, 17 decimal = 1, etc. The ORA #$10 instruction, which combines the individual bits in its operand with those in the Accumulator, is just a fancy way of adding $10 to the value of our digit. The value of the low nibble is isolated by ANDing it with #$0F. It is then ORed with #$10 to obtain the internal character and stored in the next screen position. We have effectively stored the thousands and hundreds digits in the screen window. The code loops back again to obtain the value for the two nibbles in SCORE+ 1. These contain the tens and units digits. All of the store operations are done using indirect indexed addressing of the form STA(WINDOW),Y. We will discuss this at greater length in later chapters. Meanwhile, it allows us to index rapidly into a memory area whose two-byte address is stored in zero page.

If you're confused or lost at this point, don't worry. Just read on. Our intention was merely to show how a simple game like "Breakout" could be translated into Assembly language using graphics subroutines. It is not necessary to understand all of the details but to be able to roughly follow the code as it pertains to the game's flow chart. Many of the subtle tricks we mentioned in the previous discussion we will discuss in much greater detail in subsequent chapters.

Download BREAKOT.EXE (Executable program)
Download BREAKOT.OBJ (Object code)
Download / View BREAKOT.LST (Assembler listing)
Download / View BREAKOT.S (Source file)
Download / View BREAKOT.RAW (As printed in book)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 5
Player-Missile Graphics

Player-missile graphics differ from playfield graphics in two distinct ways. First, the players, which appear as semi- transparent overlays against the background, are memory independent of the playfield graphics. Thus, these sprites, as playermissiles are sometimes called, don't overwrite playfield memory and don't destroy the background graphics as they move around the screen. Second, since they are directly mapped to the screen by the CTIA/GTIA and the ANTIC hardware chips, positioning a sprite on the screen is extremely fast and accurate. They were designed with fast smooth motion in mind.

As we learned from character graphics, the screen is a two-dimensional image; the screen RAM is organized one-dimensionally as one long string of bytes. Animating a character requires erasing the old character image, calculating a new position, then writing a new character image at the new position. Unfortunately, if the motion is vertical, the new position must be some multiple of twenty or forty bytes apart from the original in memory. The necessary calculations are time consuming. Moreover, character graphics animation with 8 x 8 pixel sized characters isn't smooth unless you use numerous transitional shapes. Also the background needs to be restored after each move.

Atari engineers thought of a simpler and better method to achieve smooth animation. They created a graphics image that was both one-dimensional on the screen and one-dimensional in memory. This image appears as a vertical stripe one byte or eight pixels wide extending from the very top of the screen to the very bottom. The image or player appears in RAM as a table of bytes that is either 128 or 256 bytes long. Each byte, depending on the programmer's choice of resolution, is mapped into either one or two horizontal scan lines.

The two hardware chips, the CTIA/GTIA and ANTIC, automatically place a sprite on the screen. While the GTIA is busy putting color pixels on the screen based on graphics information furnished by ANTIC, it simultaneously keeps track of its current horizontal position on the screen. If it finds that the current horizontal position equals the value of the horizontal position register for the player, it asks ANTIC to give it a byte from the player-missile area of memory corresponding to the current scan line. It then interprets that byte as a series of on-off pixels or points, starting with the high bit on the left, and plots them in the selected player color. If the bit in the byte is on, it illuminates or plots a pixel, and if it is off it skips plotting the pixel. Areas of the player stripe that contains zeros or no player data are transparent to the background or playfield graphics. Scan lines that contain data form a solid image that overlays but does not affect the background image.

Player data is mapped in much the same way as character data is mapped in a character set. Where a character is limited to eight rows of data, a player stripe, depending on player resolution, consists of either 128 or 256 rows. Each byte corresponds to a row, and each of its eight-bit positions corresponds to the eight pixels that collectively form part of the player image. A bit that is on or has a one in it lights the corresponding pixel which maps from left to right, high bit to low bit. For example

This data table, which is stored in the player-missile area of memory, is 256 bytes long. Actually, only 192 bytes corresponding to the screen's 192 scan lines are effectively used. A 256-byte area was chosen since it is one page of computer memory, and it is easier to find the start of a player's data when it begins on a page boundary.
The vertical starting position depends on the position in player memory. However, the 0th or first byte is mapped automatically by hardware to an area offscreen past where ANTIC is generating display list scan lines. Therefore, the first twenty or so bytes and the last thirty or so bytes are beyond the normal raster scan of a television set. A player that begins in the thirty-fourth position in player-missile memory will map to the screen starting at the second scan line. Players move vertically by moving the player data through the 256-byte page of player memory. The higher a player image is stored in memory, the lower it appears on the screen.

Atari's player-missile system consists of four players and four missiles that reside in a 2K block of memory known as the player-missile memory area. Think of

missiles as narrow players two pixels wide instead of eight pixels wide. Each of the four players has its own playermissile area of memory. Single-resolution players use 256 bytes or one page of memory, while double-resolution players use 128 bytes or half a page of memory. They are arranged sequentially so that player one follows player zero, etc. The four missiles, on the other hand, are stored in the same block of memory just below player zero. They are arranged in two-bit pairs with the 0th missile occupying the rightmost or lowest twobits, and the third missile the leftmost or highest two bits. While this arrangement is handy if you want to combine all four missiles to enable a fifth player, it presents a problem when moving a single missile vertically on the screen. If the missile data is moved in memory to correspond with a missile's screen movement, then portions of data for the other missiles that reside in those same bytes will also be moved. There is a Machine language solution to the problem that requires masking the bytes during movement, but this technique is unavailable from BASIC except by a complex USR function. Fortunately, horizontal movement is much easier. The designers incorporated a separate horizontal position register for each player and each missile. Even if all missiles are combined to enable a fifth player, each two-bit-wide missile band must be still set individually. Essentially, the fifth player is kept as a unit when it is moved, if each missile is positioned two units apart in the horizontal axis when it is moved.

The color and size of each player and its associated missile can also be specified. Size only affects the horizontal width of the player or missile. Widths can be normal, double, or quadruple size. For example, in double width, the GTIA chip begins double plotting the bytes on bits when it reaches the horizontal position register of the player or missile. Color is assigned to four shadowed player-missile color registers. Since these four additional color registers are independent of the playfield color registers, more colors can appear on the screen at any one time. While each player can have a different color, each missile is assigned the color of its corresponding player. Thus, if player three is green, missile three will also be green. The only exception to the rule is that if a fifth

player is enabled, the four combined missiles use the color in playfield three's color register. There is rarely a conflict since playfield three is only used in four graphics modes. Since the GTIA plots both the player and playfield color pixels simultaneously, it can detect any overlap between graphics images on the screen.
Player priority can be set so that all or half of the players are in front of the playfield graphics, all are behind the playfield graphics, or some of the playfield colors are in front of the players with the rest behind. Any overlaps in position are of course returned in a series of read-only hardware locations called collision registers.

These are quite useful in game design. You can also enable bit 5 in the priority register (POKE 623,32) so that you obtain a third color when players 0 and 1 or 2 and 3 overlap. If you don't set the overlap option, the area of overlap will be black.

NOTE: Above values will set all missiles the same size.
NOTE: If missile sizes are set Individually, then add the four different values. The combination is poked into ($D00C).

Enabling Player-Missile Graphics

There are numerous steps involved to enable player-missile graphics from both BASIC and Machine language. Beginners see it as a long list of mysterious POKEs, but there is a logical and explainable reason for each. First there are two electrical switches between Antic, GTIA, memory and the world, which tell them whether to do DMA (Direct Memory Access) automatically. The first, called DMACTL (Direct Memory Access Control), is shadowed at location 559 decimal ($22F). It enables ANTIC to fetch bytes automatically and plot them to the

screen. When it is turned off with a zero, the screen is turned off because ANTIC can no longer fetch bytes from memory. It defaults normally to standard playfield graphics with double-line resolution for player-missiles turned off. This value is 34 decimal. The table below, which depends on which bit positions are set in DMACTL, summarizes the various modes.

The second switch GRACTL (Graphics Control) physically enables playermissile graphics. If we POKE a three into decimal location 54279 ($D01D), both players and missiles are turned on. In addition, we will need to set color, width, and horizontal positions for each player.
Next we need to reserve space for our 2K player-missile area in memory assuming we are using single-resolution P/M graphics. Since it needs to be on a 2K memory boundary, it is best to reserve space and put it above the screen and display list areas beginning at eight pages (2K) below the top of memory. This is accomplished by simply adjusting the top of memory pointer at location 106 decimal with a new value that is eight less than the original value. The computer now thinks top of memory is 2K lower than it actually is, and assigns screen memory and its accompanying display list below that when any graphics mode is set up.

The 2K player-missile area (single-line resolution) begins at the new top of memory which we just POKEd into location 106. Since ANTIC needs to know where we have put our P/M storage area, we need to POKE this value into location 54279 ($13407) known as PMBASE. This is the high byte value of the location. The low byte value is assumed to be zero since it is on a 2K page boundary.

The memory area assigned to player-missile graphics may or may not contain miscellaneous data which will appear as garbage in the P/M stripes. The 2K area should be cleared to zeros. However, since a long series of POKEs in BASIC is rather slow, it is faster to clear only the P/M memory that you will actually use. In the example below, only player #0 is used. Therefore, only the area from PMBASE+ 1024 to PMBASE+ 1280 is set to zero. In the worst case where you were using the missiles and all the players, the unused first 768 bytes in singleline resolution or the first 384 bytes in double-line resolution need not be cleared.

Space Ship Example

Player #0 is a spaceship originally five scan lines high. To make it appear larger each scan line is plotted twice and the width is doubled. In addition, a pair of zeros has been placed at both the top and bottom of the player image for a total of fourteen bytes. The reason for this will shortly be clear. The fourteen player-missile data bytes are then POKEd into the proper P/M memory area for player #0. By POKEing the data starting at SPOT = PMBASE+1024+100 the player will begin 100 scan lines from the top. Since the first thirty-two scan lines are above the top line of text on the screen, our image begins sixty-eight lines from the top. Accounting for the two blank or zero lines of our image, our ship actually begins at the seventieth scan line.

For a player to move vertically, all of its data needs to be shifted in memory. Players can move downward on the screen by shifting the data upward in memory. Conversely a player moves upward on the screen by moving its data downward in memory. The laver case is definitely the easier direction because data can be moved from top to bottom without the danger of overwriting any of the bytes during the move. All fourteen bytes of data are shifted upward two scan lines from SPOT+I to SPOT+I-2. The two extra zero bytes on the end serve to erase the bottom two bytes of the ship's shape in its previous position. If this isn't done, the bottom sliver of the ship will remain as screen garbage after the move.

Moving the ship down the screen or upward in memory is slightly harder. If we start at the top of the shape as before, the first byte (0) would replace the third byte (153), and the second byte (0) would replace the fourth byte (153). Fine, but when we try to move the third byte to the fifth position, the original data (153) has been overwritten by the first move. Eventually all of the bytes will contain zeros and the shape will disappear. The solution to this dilemma is to start at the bottom of the shape and move each of the bytes upward in memory two places. Since you don't overwrite any shape information during the move, the shape remains intact. Again, the two extra zeros at the top serve to erase the tiny two-line sliver that would have remained after the move. It might seem strange that we deliberately added two zero bytes at the top and two at the bottom of our player when the

rest of the storage area is obviously full of zeros. However, these could only be used if more bytes than that of our actual shape were moved, and only if we adjusted where the top or bottom of our shape begins. If we need to go to this trouble, it is easier to add extra leading and trailing zeros.
Download PMEXAMP1.BAS (Saved BASIC)
Download / View PMEXAMP1.LST (Listed BASIC)

A Player-Missile Machine Language Move Subroutine

It becomes apparent from this simple example that moving a player vertically using POKEs in BASIC is very slow. However, moving a player horizontally is fast because a horizontal position register was built into the hardware.
One clever approach to moving players fast vertically is to take advantage of BASIC's fast string-handling capabilities. These routines are just high speed Assembly language copy routines. The trick to using these routines is to fool the Atari into assigning the player-missile string data to the memory area assigned to player-missile graphics. Then player-missile graphics data can be moved simply with P$ = S$ type statements. While this technique allows the programmer to avoid Machine language subroutines, it is not the easiest method to learn or understand. Since the example and explanation would be intimidating to beginners, we will delay it until much later in this chapter.

Player-missile graphics can be speeded up tremendously if Machine language subroutines are used. There have been numerous vertical blank player-missile subroutines published in the magazines. While most are effective in rapidly moving players vertically by one of several techniques, nearly all have neglected missile movement. This is unfortunate since moving one missile without disturbing the others requires using AND and ORA machine instructions on a bit level to mask off the other missiles during vertical movement. This is something that is beyond the capabilities of BASIC.

We have developed an easy-to-use Machine language subroutine that can handle both players and missiles simultaneously. The subroutine resides in page six of memory but is relocatable to virtually any free area of memory.

There has been much controversy on whether page six of memory is actually a safe area for placing user subroutines for BASIC programs. While it is true that inputting more than 128 characters into the text buffer will overwrite the beginning of page six, any program-especially a game that remains in control throughout-has no problem. In any case, the second half of page six $680-$6FF is always safe.

Our subroutine was designed for single-line resolution player-missile graphics. It requires the input of five variables: player number, player length, the old Y (vertical) position, the new Y position, and the new X (horizontal) position. The format is:

A = USR (1536, PLAYER #, PLAYER LENGTH, OLD Y, NEW Y, NEW X)

The number 1536 ($600) is the starting location of our subroutine. The players are numbered 0-3 and the missiles 4-7. Player #4 in our subroutine is actually missile #0, etc. The player length can only be as large as sixtyfour lines. This limitation will become clear when we explain how the subroutine works. The values for both the X and Y positions can be anything from 0-255. However, you should be aware that only X values in the range of 48-207 and Y values in the range 32-223 are within the bounds of the 40 by 24 text screen or playfield graphics area. The range is actually slightly larger, but visibility depends on the overscan of your television set. While the OLDY value is the programmer's choice upon initialization, it must be set equal to the NEWY position after each USR call. If OLDY isn't set to NEWY immediately after the subroutine call, unpredictable graphics will begin to appear in the player stripe when the subroutine is used again. The reason is that the subroutine zeros out the shape at its previous position OLDY before drawing the shape again at its new position NEWY. If OLDY isn't set to the player's last vertical position, the subroutine will miss zeroing the shape at the old position. Most likely, you will get two shapes or, worse yet, a sliver left at the previous position.

A Player-Missile Example

The next example is a simple demonstration of the speed, versatility and ease of use of our player-missile subroutine. This two-part example is instructive in several regards. First, it shows how a simple spaceship capable of firing missiles can be joystick controlled. Even its simple inline code, which allows either the ship to move or the missile to move, but not both simultaneously, gives some insight into the difficulty in designing even

the simplest games. Second, the concept of priority, or which player or playfield is to be drawn in front of another is demonstrated. But first I think a discussion of how objects move might be helpful, especially to beginners.

Dynamics of Objects in Motion

Any object in motion, whether it is simulated on a video screen or moves in the real world, is subject to the laws of physics. Readers will immediately cringe at the thought of advanced mathematics, mainly calculus. But calculus is merely a method of calculation that involves the summation of many small bits and pieces of a body's velocity and acceleration to determine the actual distance an object travels. Fortunately, the computer automatically divides our time frame into analogously small units, or animation frames.
Let's examine an object in simple linear motion. The object is initially at rest. It is then given a horizontal velocity of one unit to the right. Thus the velocity is +1 unit/time frame. During each animation frame, the object moves +1 units to the right.

An object's direction of travel and its magnitude is represented by a line segment called a vector. An object's velocity always points in the direction of travel. Our object shown below has a velocity of +1 units/time frame, so that the velocity is pointing to the right. Since the velocity vector is to the right, the object moves to the right +1 unit/frame.

Similarly, an object that moves diagonally downward and to the right has a positive velocity vector in both the X and Y directions. This velocity vector is a combination of the velocity components in both the X and Y directions. The object will continue moving in the direction of the velocity vector until either VX or VY changes. For example, if VX becomes zero, the object will begin to move straight down in the direction of the new velocity vector.
This can be formalized into equations for each of the two screen directions X and Y.

VX = +1 Velocity is constant in X direction.
X = X + VX New position is the old position plus the change in position (velocity).

Likewise:

VY = +1 Velocity is constant in Y direction.
Y = Y + VY New position is the old position plus the change in position (velocity).

While this simplistic method of moving objects by instantaneous changes in direction and velocity works on the video screen, objects in the real world don't behave this way. Take the family car for example. You step on the gas pedal, and the car's velocity begins to climb steadily. The distance traveled each second begins to increase as the velocity increases. When the car is only going 15 MPH the car travels only 22 feet each second, but when the car reaches 60 MPH it travels 88 feet each second.
The driving force that speeds up our car is called acceleration (V = V + A). Acceleration can be constant as in a rocket's thrust, or like gravity which pulls a falling object to Earth. For a screen object's motion to appear realistic, especially in the case of a falling bomb, acceleration must be taken into account. When a bomb drops, its vertical velocity increases with time. If there were no wind resistance, the bomb's vertical velocity would increase until its impact on the target.

If a constant force was suddenly applied to a stationary object such that it accelerated downward with an increase in velocity of one unit/ frame, the distances moved would grow substantially.

```
TIME    VELOCITY  POSITION (distance)
 0        0          0
 1        1          1
 2        2          3         VX = VX + 1
 3        3          6          X = X + VX
 4        4         10
 5        5         15
 6        6         21
```

The plot of the trajectory of a falling bomb is shown below. The trajectory, neglecting wind resistance, forms a curve that is called "parabolic." There are two components to the velocity vector. VX in the X direction is a constant set equal to the plane's forward velocity. VY in the Y direction grows larger with time as gravity accelerates it in the downward direction. This same effect can be observed by dropping a ball from the second or third story of a building. At first, the ball falls slowly, but then it begins falling faster. Observers at ground level will note the acceleration of the moving ball just before it bounces. The summation of the two velocity vectors determines the resultant direction of an object's motion for each animation frame. Since the VY vector grows larger with each frame, the total velocity vector begins to point downward. Eventually, the bomb will be falling almost straight down. Thus:

VX=CONST X=X+VX

VY=VY + GRAVITY Y = Y + VY

In the above cases, the acceleration was non-existent or constant. However, as we will see at the end of this chapter, even simple games involving a steerable spaceship that can be thrust in the direction that it is headed, must use variable acceleration. The rate or value may remain constant but the direction changes. While only the beginning of the discussion above is necessary to follow the second example, I hope it will give you some insight into how an object's motion is simulated on the Atari's video screen.

Program Initialization

Our first example, a joystick-controlled single ship capable of firing missiles in eight directions, takes advantage of our player-missile Machine language subroutine. As in the first example in this chapter, a 2K block of memory needs to be reserved for player-missile graphics. The top of memory pointers are moved down eight pages, and

the player-missile base PMBASE is set equal to the top of memory. BASIC then sets up the screen area and its display list just below. Graphics I was chosen because it uses little screen memory and because we will need to write some large characters to the screen for the second part of our example.
Next, our two Machine language subroutines need to be POKEd or placed into memory. The first subroutine is the player-missile subroutine. It is 151 bytes long. Since it begins at the start of page six ($600) or 1536 decimal, a simple FOR ... NEXT loop that reads the data statements and POKEs memory, will do the job. The second Machine language subroutine is a clear memory routine. It will automatically clear player-missile memory to zero in a fraction of a second. It replaces a slow thirty second long FOR ... NEXT loop which would POKE a zero into each sequential memory location from PMBASE+0 to PMBASE+2047. It also resides in page six of memory, beginning at $6A0 or 1696 decimal. It is twenty-six bytes long.

We must activate player-missile graphics next. We will set DMACTL for singleline resolution and GRACTL for both players and missiles. Player #0 is set to double width by writing a zero to location 53256, and player #2, which will be used in the second part of this example, is set to double width by writing a one to location 53258. The missiles are set to regular width, and the color of each of the players is selected. The priority GPRIOR shadowed at location 623 decimal is set so that player #0 is in front of the playfield graphics and player #2 is behind. We will talk more about this location during the discussion of the second half of this example. Last, we tell ANTIC our player-missile address by writing it to PMBASE at decimal location 54279. Since our subroutine also needs the location of PMBASE, its high byte is. POKEd into location 1686 which is the last memory location of our subroutine. We could have passed it to the subroutine via the USR function, but then you would have had to type the value in each time you used the subroutine.

Player Shapes Using the P/M Subroutine

We then store the two player shapes in our player shape storage area. You will recognize the first shape from the first example. It is our spaceship. The second shape is a solid block ten bytes high. Normally, you would want to POKE your shapes into the proper place in the player-missile memory area. However, when we designed the subroutine, we felt that the best and fastest approach would be to erase the shape at its old location before redrawing it at the new location. The traditional approach was to do a memory move of the entire shape including leading and trailing zero bytes, to insure pieces aren't left behind. If a shape were moved vertically more than one or two scan lines, a series of small memory moves had to be done sequentially.
In order to use our faster subroutine, we had to find a safe place to store our shapes in memory, no matter where you put our subroutine, the player-missile area, and screen memory. Fortunately, the first 768 bytes of the playermissile memory area are unused. We decided to use the first page or 256 bytes for the player storage area. This limits the size of each of the four players to sixty-four bytes. Since sixty-four bytes or scan lines make up nearly one-third of the screen, this limitation really doesn't cause any problems.

Before any shapes are stored in this "safe" area, player-missile memory should be cleared or zeroed with the Machine language subroutine stored in page six of memory at 1536 decimal. The only parameter passed via the USR function is the starting memory address PMBASE. This routine was specifically designed to clear eight pages of memory, precisely that of a 2K block of player-missile memory. If you neglect to clear it, random patterns will most likely surround your shapes in the player missile stripes. The first shape, our spaceship, which is player #0, is POKEd into the first ten bytes of the player-missile storage area from PMBASE+0 to PMBASE+9. The second shape, the solid block, which is player #2, is stored beginning at PMBASE+128.

The spaceship (player #0) must have an initial starting position and starting velocity. While it is obvious that the present position X0=100, Y0=80 is important, its previous Y axis position must have an initial value despite the fact that there couldn't have been a previous one. Setting Y0OLD=80, the value of our present Y position, will suffice. Likewise, the same is true for player #0's missile. YM0OLD=10, a value that insures that it is off screen.

Joystick Controlled Ship Movement

The spaceship's velocity is joystick-controlled. Pushing the stick in any direction instantaneously changes the ship's velocity. If the stick is pushed to the right, the ship is given a horizontal velocity of 2 units/frame. Since there is no vertical component to the ship's velocity, the ship will move to the right 2 units/frame. It will continue in that direction until either it collides with the screen's boundary, or a new joystick input gives the ship a new velocity vector. The equations that control the ship's X and Y position are as follows.
X0 = X0 + VX0
Y0 = Y0 + VY0

There is a separate pair of velocity vectors for each of the eight possible joystick directions. The absolute values on each of the axis are not the same because pixel size is rectangular rather than square. Objects with equal velocities along both axis tend to move faster along the horizontal axis than along the vertical axis. To compensate for this, we set VX = 2 and VY = 3. Diagonal velocities, which are the vector sum of the two velocity components, are faster than anticipated. While it appears that you could correct this by using fractional values for both velocity vectors on the diagonals, you can't move part of a pixel position in either axis. However, if the velocity vectors were much larger you could correct the diagonal velocity by using smaller whole numbers. The diagram below shows the velocity component values for each of the joystick directions.

The only way to set the ship's two velocity vectors for each joystick direction is to test the value of Z0 = STICK(0) against each of its possible values. When the stick is centered (Z0=15) the velocity remains the same and we skip the remainder of the tests. The other eight possiblities are tested in a series of IF statements. If a match is found for any direction, new velocity vectors are substituted. While it appears to be a cumbersome method, there is no better method.

I'm sure many readers wonder why Atari BASIC returns such a strange and illogical set of values for joystick directions. STICK(0) through STICK(3) reflect values returned in the PIA chips locations 54016 and 54017 ($D300,$D301). When the joysticks are centered, each of the four bit locations for each joystick are on (set to one). When the joystick is pushed in any direction, its appropriate bit position is turned off (set to zero). For example, if the joystick is pushed to the right, the fourth bit from the right is turned off. The remaining three on bits add up to seven. Diagonal joystick movements have some combination of two bits turned off. The diagram below shows all of the possible bit patterns.

After the joystick is read, the player's position is updated and checked so that it does not exceed the screen boundaries in either direction. A USR call to our playermissile Machine language subroutine will move the player shape into the proper place in memory so that it appears at the choosen X,Y screen coordinates. The format as discussed earlier is A = USR (1536, PLAYER #, PLAYER LENGTH, OLD Y, NEW Y, NEWX). The player length is 10, OLDY equals Y0OLD in this example, NEWY equals Y0, and NEWX equals X0. Immediately after the USR call, update the OLDY position with the NEWY position so that you don't have to worry about losing this value when you calculate your new position during the next frame.

Missile Movement

Pressing the joystick button fires a missile. STRIG (0) is normally set to a one value when the button isn't pressed. But when the button is pressed it becomes zero and the program branches to a second joystick read routine at line 185. It is nearly identical to the first, with the exception that the velocity vectors are nearly double in value; VXM0=5 and VYM0=6. This ratio produces nearly identical apparent speeds in both the horizontal and vertical axis. Since the missile needs to be given a direction to fire, it branches past the missile routine if the joystick is centered.
The missile's initial starting position is at the center of our ship. Since X0, Y0 are the coordinates of the ship's missile at the center of the ship, the missile's initial position equals the current ship's position plus the correction. Thus, XM0=X0+3 and YM0=Y0+4. The missile's position is updated each frame the same way the

ship's position is updated each frame. The new position equals the old position plus the missile's velocity vector.
XM0 = XM0 + VXM0
YM0 = YM0 + VYM0

Unlike the ship, however, the missile's velocity vector can't be changed once the missile is fired. Missile movement is in a closed loop that exits only when the missile reaches the boundaries of the screen. This closed loop not only prevents you from firing a second missile before the first has reached the end of its travel, it also prevents the ship from moving while the missile is in motion. This is a good example of how simplistic code that branches to either one event or the other creates an unrealistic effect. The solution, which will be explained in the next example, requires the ship's movement code to be placed in line with the missile's movement code, and conversely the missile's movement code to be placed in line with the player's movement code. This apparent overkill, in fact, allows the ship to be steered once the missile is launched.

The missile is plotted via the USR call to our player-missile subroutine. Missile #0, which is two pixels wide, is player #4 and its height is set to two scan lines to make it square shaped. Its OLDY position is YM0OLD, and its new position NEWX, NEWY is XM0, YM0. Again, once the subroutine is called via the USR call, it is important to immediately set the missile's old position equal to its present position.

Finally, if the missile does reach the screen boundaries, it is necessary to place it out of view beyond the scan of the television set. Since the missiles can't be simply shut off, you should set the horizontal register to either a small number or a large number. Since XM0=10 is offscreen, it will suffice.

Priority Demonstration

The second part of the example is a demonstration of player versus playfield priority. Whenever two independent objects such as two players or a player and a playfield object appear at the same spot on the screen, the GTIA must decide who gets displayed first and who second. This is known as priority. There is a register called GPRIOR shadowed at location 623 decimal ($26F) that selects which screen object is in front of the others. There are actually only four possible selections. Either all of the players are in front of the playfield, all of the playflelds are in front of the players, players #0 and #1 are in front of the playfields with players #2 and #3 behind, or playfields #0 and #1 are in front of all the players with playfields #2 and #3 behind.
Lower number players take precedence over higher number players and, likewise, lower number playfields take precedence over higher number playfields. Actually, there is no way for two playfields to occupy the same space, since each color pixel is assigned to a specific color register. In fact, only the on bits, or those portions that show in the player-missile stripe, actually are involved in the priority conflict. The off bits are ignored by the system. Obviously, if a player has the shape of a donut, the background or another higher numbered player set behind it will show through the hole.

Unfortunately, player #0 always has priority over the other three players. There is no method to change the priority between the individual players other than to switch the player data between player-associated areas of memory.
The great advantage of priority control is that you can control what happens to players when they meet the background color display. Perhaps you have an aerial combat game. You may want the plane to fly behind clouds, yet always remain in front of the scrolling ground far below. Setting bit three in GPRIOR will give two of the playfields higher priority than the players, and the remaining two playfields the lowest priority.

In our example, we have the word ATARI written in playfield #0, a spaceship (player #0), and a moving block (player #2). Initially, bit I is set in GPRIOR (POKE 623,2) so that player #0 moves in front of the letters while player #2 moves behind the letters. This part of the example can be reached by pressing the START key. The IF statement in line 175 tests for this possibility. The spaceship is joystick-controlled exactly like the first part; however, the missiles have been disabled.

Pressing the SELECT key sets the color of the letters to the same color as the background. However, since the pixels in the letters still refer to playfield #0, and player #2 had a lower priority than all of the playfields, that player is masked by the invisible playfield object as it moves behind it. Our much larger player shows through the spaces around the individual letters, thus illuminating the darkened object.
Pressing the OPTION key does two different things. First, it restores the playfield #O's color to its default color and luminance. Second, it changes the GPRIOR to I so that all players have priority over all playfields. Now both our block and our spaceship are in front of the letters. The ship (player #0) still has priority over the moving block (player #2).

Download PMEXAMP2.BAS (Saved BASIC)
Download / View PMEXAMP2.LST (Listed BASIC)

Explanation of Player-Missile Subroutine

The player-missile subroutine was designed to handle both players and missiles simultaneously. In order to do this while retaining compact and relocatable code, several compromises were made. First, only single-line player resolution is offered. We felt that the coarser, double-line resolution could be simulated by setting the player-missile stripe at double width, then double plotting each of the shape's bytes. An attempt to give you a choice of resolution modes would have created a long and messy algorithm since the memory requirements of each are quite different mathematically. In single-line resolution the actual memory storage areas for each player are exactly one page or 256 bytes apart. This makes setting the pointers to the memory area easy since only the high

byte is involved. But in double-line resolution the memory areas are 128 bytes apart. A much more complicated multi-byte add is required to set the pointers.
Second, we decided to erase the old shape completely before drawing the player shape in its new position. It is a faster technique since the more traditional method needs to do a series of block memory moves to shift the shape more than one line at a time. Our method requires storing the shapes in a permanent safe spot. We choose the first 256 bytes of the player-missile area. Since there are four player shapes, each shape is limited to 64 bytes or scan lines.

Advanced Assembly language programmers will notice that we did not write the subroutine in VBLANK. We felt that BASIC was slow enough without adding small delays while waiting for the VBLANK interrupt to begin our subroutine. Moving several players and their corresponding missiles on the screen at the same time could slow the game down. Motion smoothness is another reason for using VBLANK for player-missile graphics. However, as you will see in the following game examples, the animation frame rate is not fast enough to produce smooth animation with or without VBLANK.

The subroutine is divided into three parts: a routine to interpret and store the passed parameters in the USR function; a routine for erasing and drawing the player shape; and a routine for erasing and drawing a missile. Once the parameters are stored, a test on the player number determines if it is actually a missile, and if so, branches to that part of the code.

Interpreting USR Parameters

A USR function in BASIC pushes all of its parameters onto the stack before it enters the Machine language subroutine. The stack is like a dish dispenser in that the last value placed on the stack must be pulled off first. The first byte contains the number of passed parameters, a useless value, and one to be discarded. The other parameters are stored in two-byte pairs, high-byte first, in the order that you passed them, The high-bytes in our example are useless since none of our values exceeds 255. The five pairs are pulled off the stack in an indexed loop using the X register. They are pulled two bytes at a time. Only the second, or low byte is stored sequentially at PLAYNUM,X ($691, X). Thus location $691 contains the player number, $692 the player length, $693 the value for OLDY, etc.

Erasing and Storing Player Shapes

The pointers to move our player or missile shape from their storage area to the proper player-missile memory area are set up next, The shape is stored at SHAPEL, SHAPEH. The high byte BASE is just the beginning of the P/M storage area which was POKEd into location 1686 decimal ($696) from BASIC. The low byte is obtained from a table called INDEX. Each of the values in this four-byte table are sixty-four bytes ($40) apart. The actual memory location that the shape is to be stored at in the P/M area is SHPML, SHPMH. Since each player is 256 bytes or a page apart in memory, we added #$04 to BASE in order to set it for player #0. We then did a cute little trick. We put it in a loop and incremented SHPMH for each player number.

Erasing the old player shape from memory could be handled quite easily with simple indexing in the form STA ADDRESS,Y if only one single player were involved. ADDRESS would be the absolute address of that player's P/M memory, and the Y register would contain the value YOLD. Unfortunately, the high byte of each player's P/M memory area is different. Rather than try to update ADDRESS each-time, it is more efficient if the two-byte page address is stored in zero page. Then indirect index addressing in the form STA (SHPML),Y can be used either to erase or plot player-missile data.

If the computer finds a $00 in location $CE (SHPML), and a $9C in location $CF (SHPMH), then the base address is $9C00. The Y register contains the value of the OLDY position. If the shape was thirty-two scan lines down the screen, then the Y register = $20. If the computer wished to erase the shape, then it would store a #300 in the Accumulator into memory location $9C00 + $20, or $9C20 as shown:

The actual code to erase the player shape is reiterated in a loop PLAYLEN times. Since it is easier to increment the Y register from zero until it is equal to PLAYLEN, the vertical screen offset is stored in SHPML in zero page. The address SHPML, SHPMH is now the address of the beginning of the player-missile shape. The Y register offsets into the shape. The code is shown below.

```
ERASE   LDA OLDY     ;Y VALUE
        STA SHPML
        LDA #$00     ;ERASE WITH 0
        TAY          ;Y REGISTER = 0
.1      STA(SHPML),Y ;STORE IN PROPER P/M AREA
        CPY PLAYLEN  ;DONE?
        BLT .1
```

The code for drawing our shape is quite similar, except that instead of storing zeros in P/M memory, we transfer the player shape from its storage area to its proper place in P/M memory. The pointers to its storage area, SHAPEL, SHAPEH were previously set up in zero page. We need update only SHPML, the low byte pointer to the place we are actually moving our player, with the NEWY position. The high byte SHPMH remains the same. The code follows:

```
        LDA NEWY         ;NEW Y VALUE
        STA SHPML        ;SETUP POINTER TO PLOT
        LDY #$00
DRAW    LDA (SHAPEL),Y;LOAD BYTE FROM PLAYER SHAPE TABLE
        STA (SHPML),Y    ;STORE IN PROPER P/M PLAYER AREA
        INY              ;NEXT BYTE
        CPY PLAYLEN      ;DONE?
        BLT DRAW
```

Handling Missiles

Player numbers 4-7 refer to missiles 0-3 respectively in our player-missile subroutine. If we didn't use different numbers, the user would have to type in a letter M or P to differentiate between the two different kinds of sprites.

While missiles are just narrow players, all four reside in the same 256-byte block or page of memory. The missiles, each of which is two bits wide, are arranged parallel to each other. Essentially, all four two-bit pairs for each scan line are in the same byte of data. This makes moving one missile but not others somewhat difficult.

Missile #0 uses the first two or lowest two bits of the byte, missile #1 bits three and four, missile #2 the fifth and sixth, and missile #3 the two highest bits. While the data in one byte determines which pixels are lit for all four missiles on any scan line, each missile has an independent horizontal position register. Thus, each of the missiles can have movement completely independent of the others.

As we mentioned, moving one missile vertically without changing the other missiles' data. can be a problem. If we had just two missiles initially on the same scan line, and we attempted to move missile #0 downward one scan line, either a memory move or an erase before redrawing would affect the missile #1 as well. In the first case, missile #1 would move in tandem with missile #0, while in the second case missile #1 would be erased. The solution is to mask off the other missile's bits during the erase, and to draw the missile's new position using another masking operation that will not affect the remaining bits.

ORA Instruction

This drawing technique uses the OR memory with Accumulator (ORA) instruction. It works on the bit level. If the bits in either memory or the Accumulator are on, then the result is one. If neither is on, the result is zero.

```
     ACCUMULATOR MEMORY RESULT BIT IN
         BIT      BIT    ACCUMULATOR
          0        0          0
ORA       1        0          1
          1        0          1
          1        1          1
```

If missile #0 was already on a particular scan line and you wanted missile #1 to be on the same scan line you would ORA missile #1 with the byte in P/M memory.

```
        0 0 0 0 0 0 1 1 MEMORY
ORA     0 0 0 0 1 1 0 0 MISSILE SHAPE
        0 0 0 0 1 1 1 1 RESULT
```

AND Instruction

Another selective drawing technique is the And Memory with Accumulator (AND) instruction. It too works on a bit level and is used to filter or mask out certain bits in the Accumulator. Both the memory bit and the Accumulator bits must be set (on) for the result to be one. If either memory bit is off, or both bits are off, the result is zero. We put ones where we don't want to change bit values, and zeroes where we do. We can erase one missile at a time without affecting the others.

```
     ACCUMULATOR MEMORY RESULT BIT IN
         BIT      BIT    ACCUMULATOR
          0        0          0
AND       0        1          0
          1        0          0
          1        1          1
```

If you wish to erase only one of the two missiles on the same scan line, you AND the data byte with a mask that always produces a zero bit result in the missile bits to be erased, and a one in all the other bits. For example, to erase missile #0 and not missile #2, the mask with which you AND the data bit is $FC.

```
        0 0 1 1 0 0 1 1 MEMORY
AND     1 1 1 1 1 1 0 0 MISSILE SHAPE
        0 0 1 1 0 0 0 0 RESULT
```

There are four different missile masks:

Missile mask #0 1 1 1 1 1 1 0 0 $FC
Missile mask #1 1 1 1 1 0 0 1 1 $F3
Missile mask #2 1 1 0 0 1 1 1 1 $CF
Missile mask #3 0 0 1 1 1 1 1 1 $3F

Likewise, there are four different missile shapes. For simplicity and visibility we made each missile two pixels wide. The height is controlled by the length, which is nominally two bytes, However, missiles can assume tall, thin dimensions if the user chooses a length of four or more bytes.

Missile shape #0 0 0 0 0 0 0 1 1 $03
Missile shape #1 0 0 0 0 1 1 0 0 $0C
Missile shape #2 0 0 1 1 0 0 0 0 $30
Missile shape #3 1 1 0 0 0 0 0 0 $C0

The data for the above two tables appears in our player-missile subroutine as MASKS and SHOTS.

The heart of the missile routine is the erase and draw code. Each is a mere three lines long. The selected missile is erased by ANDing memory with the proper mask.

```
LDA (SHPML),Y ;LOAD OLD MISSILE DATA
AND MASKS,X ;ERASE IT BUT DON'T DISTURB OTHER MISSILES
STA (SHPML),Y ;STORE RESULT BACK IN MEMORY
```

The selected missile redrawn in its new position with the following three lines of code.

```
LDA (SHPML),Y ;LOAD OLD MISSILE DATA
ORA SHOTS,X ;MERGE SHOT DATA WITH OTHER MISSILES
STA (SHPML),Y ;STORE NEW COMBINED MISSILE BYTE IN MEMORY
```

The technique is best illustrated with an example where we have three missiles on the screen. Missiles #0 and #1 are on the same two scan lines. Missile #2 is on the same scan line where we wish to move our missile.

Download PMINTB.EXE (Executable program)
Download PMINTB.OBJ (Object code)
Download / View PMINTB.LST (Assembler listing)
Download / View PMINTB.S (Source file)
Download / View PMINTB.RAW (As printed in book)

Two Ship Example

We can write a simple two-player shoot-'em-up game in BASIC if we use our player-missile subroutine to provide enough animation frame or speed for playability. The code for the second ship is nearly identical to that for the first ship. In fact, with the exception of using variables having a I at the tail end of the variable names, the code is the same, line for line. The code for player #2 follows the code for player #1.

The code for each player includes: a joystick read routine to determine the ship's new velocity vector and to update its position; a similar routine to determine the missile's velocity and direction; and logic to prevent either the ship or the missile from leaving the screen boundaries. The ship in our previous example stopped dead while the missile moved. These ships, however, not only continue in their present course during the missile flight; they even alter course to evade enemy missiles.

In the previous example, if the missile movement code was executed, it branched past the player movement code. In this example, the missile code updates the player position, and the player position code updates the missile position. This enables the player to continue moving or maneuvering while the missile is in flight.
If you look at the game's flow chart for each player, you will notice that the two possible paths are decided by whether the joystick button is pressed. Pressing the button fires a missile in the desired direction, only if there isn't another missile already on the screen. Of course, if you don't give it a direction (joystick centered), it skips firing the missile and just updates the ship's position based on its current heading. When it fires the missile, it turns the missile flag on (M0FLAG = 1). To prevent it from firing again before the missile reaches the edge of the playfield, the program tests M0FLAG in line 430. If the flag is on, it just updates the missile's position based on its current trajectory. Finally, once the missile reaches the screen edge, it turns the flag off (M0FLAG = 0) and plots the missile offscreen.

When the joystick button isn't pressed, the program code reaches line 110, the beginning of the joystick read routine which determines the ship's new velocity vector. This enables the ship to change direction. Thus, if a player wishes to change direction immediately upon firing his missile, he must release the button quickly, so that it reaches this code on the next animation frame.

Once a missile has been launched, it will continue on its path even while the ship is being maneuvered. Therefore, the missile's position needs to be updated in this pathway, too. This is done immediately after the ship's position has been updated, but only if the missile is on the screen.

Collision and Explosions

A game wouldn't be complete if we couldn't detect if one or the other ship were killed in combat. There are two ways to die in this type of game-by collision with the opponent's ship or by missile fire. The collision register at decimal 53260 ($D00C) detects player #0 to player collisions. If it returns a value greater than zero, two players have collided. Likewise, collision registers at 53256 ($D008) and 53257 ($D009) detect collisions between missiles and players. The first will return the value 2 if missile #0 collides with player #1, and the latter will return the value I if missile #1 collides with player #0. It is important that these collision registers are cleared to zero before plotting players and missiles on the screen. You do this via the HITCLR register at decimal 53278

(DO1E) in line 90. This line at the beginning of the animation frame loop clears all of the collision registers before any players or missiles are placed on the screen.
The rather simplistic explosions are linked to the sound routine. Each ship brightens from luminance 4 to luminance 14 in its own color, then blacks out quickly. The luminance changes within a low, rumbling sound loop. The formulas in lines 1570 and 1580 were designed to prevent INT (10-I*0.66) from becoming larger than the value 10. If the ship's luminance of 4 when added to this value became larger than 15, the ship's color and luminance would change as the color value would wrap to the next higher color with a low luminance.

Recall that the sound statement is SOUND (Voice, Pitch, Distortion, Volume). The pitch is set to 250, a very low tone, with a distortion value of 4. The even numbered distortion levels 0,2,4,8,12 introduce different amounts of noise into the pure tones, 10 and 14. The FOR ... NEXT loop (lines 1150-1580) decreases the volume level very slowly.

This example's slow animation frame rate produces a game lacking smoothness. BASIC is slow even with the use of Machine language subroutines. There are a lot of IF ... THEN statements that slow the game down. The game, however, will run a little faster if all of the REM statements are deleted. Arcade games really need to be written entirely in Assembly language to achieve fast, smooth animation. The Space War game that is developed at the end of the chapter is a very similar game, but smoother and more controllable.

Download TWOSHIP.BAS (Saved BASIC)
Download / View TWOSHIP.LST (Listed BASIC)

Shoot Bricks Game

The next example uses both playfield and player-missile graphics in a timed game in which the object is to survive the longest between two crushing brick walls. The joystick-manueverable player can use a pistol to shoot any of the bricks out in the ten rows on either side of him. The bricks are replenished randomly at a rate slightly faster than the player can remove them. Thus, it becomes a matter of strategy and endurance to last for any reasonable length of time.
When designing a game like this, you must anticipate the player's strategy. The player, when confronted with the impossiblity of keeping the entire wall back, will retreat to either the top or bottom and shoot just at the blocks immediately surrounding him. If the bricks were still placed on the screen randomly even after a row of bricks closed completely, the player would have plenty of time to shoot at the few random blocks appearing in the rows adjacent to him. However, if the random blocks are not put on the screen in the already closed rows, bricks would appear more rapidly in the few open rows. This makes the game fast-paced and somewhat challenging. The game itself has little play depth, so don't expect it to hold your interest as a game. It was primarily designed to teach programming technique.

It is desirable to have a different color for each of the ten rows of bricks, but there is a maximum of only four color registers in the non-GTIA graphics modes. Therefore, we use display list interrupts to change a single color register while ANTIC is drawing the screen. It would be easier to use GTIA mode 11, but, unfortunately, this mode does not support collision registers.

We choose to do the display in ANTIC 5, one of the non-BASIC graphics modes, because the four-color characters are sixteen scan lines high. Each of the lines contains forty characters. The blocks are four color dots wide by 16 scan lines tall. The bricks are added and removed in adjacent pairs (two characters) so that they appear to be square-shaped. The availability of the extra colors had little to do with the initial programming design, but when we put in a scorekeeping timer at the bottom of the screen, we were able to draw the characters using a different color register from that of the blocks. A collision is never detected when the player touches the score line.

Setting Up Display List

The screen resides just below the top of memory which has been lowered twelve pages to make room for both the player-missile memory and new character set. We choose to modify the display list for a Graphics I screen because the display memory (20 columns x 24 rows) is the same as ANTIC 5 (40 columns x 12 rows), and our display list is slightly shorter. The start of the display list, which is shadowed at locations 560 and 561, is exactly the same for both graphics modes. Therefore, it is easy to POKE in our new display list at DUST = PEEK(560)+PEEK(561)*256. This value is 37216 for 48K machines. If we substitute the following 20-byte display list for the original, we will have an ANTIC 5 screen.

```
112     These three instructions print
112     24 blank scan lines at the top
112     of the screen
69      ANTIC 5 with a LMS instruction added
128     Address of the first line of screen data
145     145*256 +128 = 37248
133     Display the rest of the data in
133     ANTIC 5 with a DLI added
133     We have a total of 11 Antic 5 lines
133
133
133
133
133
133
```

```
133
7       Text mode 2 for timer display
65      jump and wait for vertical blank
96      Address of vertical display list itself
145     145*256+ 96 =37216
        (return to the top of this list)
```

Initialization

The initialization for this game resembles our previous example. The playermissile, clear memory, and display list interrupt subroutines are each POKEd into memory in their appropriate places in page 6. The code is stored as DATA statements. The first half of the character set (512 bytes) is then moved to its new location, just above the top of memory, starting at location CHRSET which is equal to PEEK(106)*256. Only the first two characters are used to generate the screen. The 0th character is a blank and is used where there are no blocks. This includes our empty 0th row at the very top. The first character is rewritten as a solid shape using color register 1. The bit pattern for each line in the character is 10 10 10 10 decimal 170 or $AA. Using this bit pattern for color register I is a deliberate choice. The characters in the score line use color register 2. Thus, a collision with our score letters and numbers (playfield 2) will not produce the same value as a collision with a block (playfield 1).
Blocks are placed initially on the screen in rows I through 10 for columns 0 to I I and 28 to 39. There are six pairs of blocks situated on each side of the player on each row. The offset into screen memory for any block is OFFSET = 40*R + C, where R and C are the row and column respectively. The location of the screen is shadowed at locations decimal 88 and 89. With SCREEN = PEEK(88) + PEEK(89)*256, the actual memory location of any block is SCREEN + OFFSET.

Player-missile graphics appear to be double-line resolution set at double width, but they are actually single-line resolution with each byte doubled. Two different players are used for the man. Player #0 faces left, and player #I faces right. They are the same color and have the same vertical position. Player #1 is placed on the screen initially at X=125, Y=50.

The two levels of difficulty are selected with the SELECT key. It toggles between putting blocks on the screen at one-half second intervals and one second intervals. An asterisk (*) at the bottom right of the screen denotes the easier level. The START key starts the game.

Main Game Loop

The game loop tests when to place a block randomly on the screen at one-half or one second intervals. VBLANK increments the timer at decimal 20 every sixtieth of a second. When it overflows (TIMER >255), location 19 is incremented. If we set TIMER = 195, after sixty cycles occur(one second), location 19 becomes 1. This makes a convenient test to determine when one second has elapsed. Likewise if TIMER is set to 225 with the SELECT key, after one-half second location 19 would be incremented. The'test at line 210, IF PEEK(19)=O THEN 344, skips putting a new block on the screen and updating the scoring timer unless the proper timing interval has elapsed.

There are two pointer arrays, L(10) and R(10), that keep track of the inside wall boundaries closest to the player for each row. Initially each of the L(I) elements are set to I I and each of the R(I) elements are set to 28. As blocks are added and subtracted, these values begin to change by multiples of two. For example, if a block were added to both the left and right sides of row 3, then L(3)=13 and R(3)=26. If blocks were added just to the left side, eventually the sides would touch when L(3)=25 and R(3)=26. Since we don't want to add a block to a row that is already touching, we could test if R(I)L(I)=I. If it equals 1, then we choose a different random row and random side to try to place another block. Eventually, we find a place even if it is in one of the rows to the right or left of our player. The gun fighter can be maneuvered around the vacant area of the playfield by a player using a joystick. When the joystick is positioned up or down, the man moves vertically up or down by four units. When the joystick is pushed left, a left-facing figure appears and moves two units leftward. Similarly, a right facing figure appears and moves rightward two units when the joystick is pushed right. The routine also supports diagonal movements incorporating combinations of vertical and horizontal movement. Each of the joystick movements sets the variable PLAY to zero or one to indicate which player is to be placed on the screen through our player-missile subroutine. The player that does not appear on the screen is placed offscreen at X=10. Again remember to set the old Y position equal to the new Y position (YOLD=Y) just after the subroutine is used.

Collision Test

A collision between either player and the playfield #1 blocks has to be tested in two different places in the game code. Obviously the test must be done just after the player moves, but it also has to be done just after anew random block is placed on the screen. A collision is detected in either case if the value 2 is set in player #0's collision register 53252 ($D004) or player #1's collision register 53253 ($D005). Since the bricks appear to be crushing our man, it is effective to squash him by setting the player width back to normal. When this happens, it appears that the man is compressed towards the left. This occurs because the player image is always plotted from left to right and begins at the value in the horizontal position register. The GTIA just double plots player pixels when set to double width. The player struck by the left wall remains in contact with the wall when it is compressed, but the player struck by the right wall will shift to the left, or away from the wall, eight pixels. If we just correct the X position by adding 8, it will remain in contact with the right wall and look like it was also crushed by the block.

While a collision between a missile and a block isn't difficult to detect, the program must determine which block was hit. Obviously, the block must be in the row directly in line with the pistol. If you look at the diagram below, you will see that the pistol is sixteen scan lines below the top of the man. Since the top left position of the man is at X,Y, then YM = Y+ 16. The bullet fires from the tip of the gun at XM = X when facing left, and at XM = X + 8 when facing right. The movement routine prevents the pistol from going beyond the top of the first row of blocks. When the pistol is at the top of the first row of blocks (YM = 48) the man is at Y=32. Therefore, the formula ROW=INT((YM-32)/16) determines which row the bullet travels. If we substitute YM = Y+ 16 the expression simplifies to ROW=INT((Y-16)/16). For example, if the man is at the very top (Y=32) then his pistol is aimed along

ROW=INT(32-16)/16 or ROW=1 as expected. The bullet moves 2 pixels horizontally each frame until it eventually collides with a block. The pairs of blocks are then removed from the side the player faces. Since the player can't move while the bullet is in motion, there is no ambiguity possible. For example, if the player (PLAY=O) were facing left on row 2, and the left block pointer L(2)= 9, then blocks 2,9 and 2,8 are removed. You need only POKE a 0 into locations SCREEN+OFFSET and SCREEN+OFFSET-1 where OFFSET 40*ROW + L(ROW). Likewise, if a player were facing right (PLAY= 1), the two blocks that need to be removed are at locations SCREEN+OFFSET and SCREEN+OFFSET + 1 where OFFSET = 40*ROW + R(ROW).

The game naturally ends when the player runs out of space. The stopped timer indicates the time elapsed. At this point everything has to be reset for the next game. The blocks on the screen are reset for the next game. The blocks on the screen are reset slowly due to Atari BASIC's naturally slow implementation of the POKE statement. This does give a breather between games. The timer is reset, then the player. A button press is all that you need to begin a new game.

Extra Colors via a Display List Interrupt Subroutine

Some of the advanced programmers might find the display list interrupt subroutine to be of interest. It changes the color value in color register 1 ($2C5) each time ANTIC calls it via a display list -interrupt. Since it keeps an internal counter called PLACE for its X register index, it must know when to reset its pointer. It does this by checking the value in the vertical line counter which increments by one for every two scan lines. When it is equal to line 50, which is two scan lines beyond the beginning of row # 1, it resets PLACE to 0. It loads PLACE into the X register, indexes into the color table, and stores it in the color register. Now PLACE is incremented by one. The next DLI just loads the next color in the table into the color register. The pushes and pulls on the stack at the beginning and end of our subroutine save the current X register and Accumulator so that they are restored upon return to your program.

Download SHOOT.BAS (Saved BASIC)
Download / View SHOOT.LST (Listed BASIC)

Space War Game

Space War, the first game with a fully steerable spaceship, was developed at MIT. While most of the newer computer owners won't remember this game, practically everyone is familiar with Asteroids. Most versions of this game have a fully steerable spaceship that can be thrust in the direction that it is headed. Although some versions invoke an automatic deceleration mode, some Asteroid games require the player to turn his ship around so that it thrusts in the opposite direction to slow down.

Dynamics of Motion with Acceleration

We demonstrated earlier in this chapter that objects move in the direction of their velocity vector. An object's new position is its old position plus its change in position due to velocity.
Using the Atari's screen coordinate system for the example above, VY is negative and VX is positive. Therefore,

X=X+VX
Y=Y+(-VY)

While the velocity vector may remain constant for many animation cycles, so that a ship will continue to move in the same direction, sooner or later a new velocity vector will be input to change the object's course. This new velocity is the vector sum of the old velocity vector and the new velocity vector.

Those readers who have taken Physics will recall that the velocity of a body in motion changes due to external forces on it while it is in motion. In spaceships, that force is thrust. Thrust causes an acceleration of the object's mass as shown in the equation.

F=m*a=m* delta V

When thrust is applied to a spaceship, it accelerates. If a ship is light and has a big engine with considerable thrust, it will accelerate quickly. But if it is heavy, it will accelerate much slower. Acceleration is essentially caused by a change in the object's velocity if you ignore the object's mass,

Unless you are doing an actual simulation, in which the values of thrust or force and an object's mass are important, only acceleration values need to be considered. Suitable values for arcade games are small and scaled, so that objects don't move fast relative to their size, or fly off the screen in the blink of an eye.

If we consider a spaceship that is in motion for three frames, then thrust only during the fourth frame, it will change direction depending on the vector sum of its old and new velocity vectors. This is illustrated below. The applied thrust is straight upwards, so that VX = 0 and VY = -2. The ship's new velocity vectors for each direction are calculated as follows:

VX = VX(old) + delta VX = 2 + 0 = 2
VY = VY(old) + delta VY = -1+(-2) = -3

Likewise, the ship's new position is equal to its old position plus its change in position due to velocity for that animation frame. This breaks dowq into components for the X and Y directions.

X = X(old) + VX
Y = Y(old) + VY

The ship's new velocity vector causes it to move two units in the X direction and three in the negative Y direction during each frame until a new thrust vector is applied. The resultant position can be summarized in the table below.

```
Frame     X     Y     delta VX    delta VY
  0      10   100        2           -1
  1      12    99        2           -1
  2      14    98        2           -1
```

```
  3     16   97       2         -3  Thrust applied here
  4     18   94       2         -3
  5     20   91       2         -3
```

Now, when the acceleration on an object, is sustained over many animation frames, the increase in velocity is cumulative. For example, if thrust were applied to a stationary object in the positive X direction with a force of 1 unit/frame, the new VX would increase from zero by units of one for each animation frame.

```
         CYCLE   VX    X                   CYCLE    VY     Y
           0     0     0                    0       0      0
           1     1     1                    1       2      2
VX=1       2     2     3   Similarly VY=2   2       4      6
           3     3     7                    3       6     12
           4     4    10                    4       8     20
```

Our example makes it clear that if you accelerate for too many animation frames, the spaceship will be moving too fast. A limit should be set on the velocity vector for both directions. just what the limit should be depends on the effect you wish to achieve. Obviously, if your animation rate is that of VBLANK, 60 frames per second, a ship velocity of 5 units/cycle will race across the screen in one-half second.

Joystick Routine

A joystick will control the ship's direction in our game. Pushing to the left or right rotates the ship to one of eight directions. Pushing up thrusts the ship. The joystick is read in the VBLANK routine every 1/60th of a second. While a Machine language subroutine is more than equal to the task of keeping up with the update, its speed creates a small problem. If a player turns his ship by pushing left or right on his joystick, the ship will spin rapidly, making 1/8 of a turn every 1/60 of a second, That translates to 7 and 1/2 turns per second, a speed that is obviously uncontrollable. The joystick must be read less often, say every sixth VBLANK cycle if the ship is to turn slowly enough to steer.
Machine language joystick routines are inherently simpler because only four bit positions for up, down, left and right, need to be tested. Diagonals are usually ignored because they nearly always represent combinations of commands. For example, a diagonal up and right command in our game means thrust while turning right. The joystick read subroutine will detect the bit pattern twice, once when testing the up bit to determine if it should reset the velocity, and a second time when testing the right bit to determine if it should turn the ship. The bit pattern is as follows.

```
  X    X    X    X RIGHT LEFT DOWN  UP
128   64   32   16     8    4    2   1
```

All of the bits are normally set when the joystick is centered or in the neutral position (I I 1 1). When the joystick is pushed in any direction its particular bit position is turned off, as illustrated in the diagram below.
To test a particular bit position you only need to AND it with that bit. For example, to test if the left bit is turned off:

```
00001010     Diagonal left & up
00000100     AND #$04
00000000     Result=0
```

The result is zero if the bit is turned off when the joystick is being pushed in that direction. If the joystick were in the neutral position, the third bit would be on, and the result after the AND operation would be non-zero. When

the result is zero we want to decrement DIR, the ship's direction, and make sure that if it becomes negative ($FF) it is reset to #$07. The code is as follows. All of the code is indexed with the player number in the X register.

```
CHKLF   LDA STICK,X     ;READ JOYSTICK
        AND #$04        ;LEFT BIT
        BNE CHKLF       ;BRANCH IF NOT ZERO
        DEC DIR,X       ;DECREMENT THE SHIP'S DIRECTION
        LDA DIR,X       ;CHECK IF NEGATIVE
        CMP #$FF
        BNE .1
        LDA #$07        ;SET DIR=7
        STA DIR,X
.1      JMP CHKFD       ;CAN'T BE RIGHT IF PUSHED LEFT
CHKRT   LDA STICK,X
```

Pushing forward on the joystick thrusts the spaceship. ANDing the joystick value with #$01 tests the up bit. If the result is zero, the joystick has been pushed up. The acceleration or thrust vector depends on the ship's direction. If the ship points to the right, DIR = 2, then VXT = 1, and VYT = 0. If the ship points diagonally upward and to the left, DIR = 7, then VXT = -1, and VYT = -1.

Ship's Direction and Velocity Vectors

Note that many of our ship's directions produce negative velocity values, while others produce positive values. Separate routines are required for adding and subtracting in Machine language. BASIC, however, just adds a negative number (X = 5+ (-1)). That's the clue. Adding a negative number is exactly the same as adding a positive number in Machine language. The difference is that negative numbers, like -1, are represented by the two's complement which for -1 is $FF. There is a limit for signed numbers of + or -127, because the BMI instruction tests the carry bit and considers the value if set. With the simplification of our thrust vector addition problem, we can construct a table of velocity vectors for each DIR value.

The equations for the ship's two velocity vector components are as follows:
VXP = VX(old) + VTX(DIR)
VYP = VY(old) + VTY(DIR)

A speed brake can be incorporated into the algorithm to prevent the velocity from exceeding a preset value. This would be analogous to wind resistance on a fastmoving automobile. It prevents a vehicle's speed from increasing infinitely. I choose a maximum velocity of 5 units per frame. It is based on keeping the animation smooth and the speed in bounds. The code for the X direction is as follows:

```
        LDA DIR,X       ;GET PLAYER DIR
        TAY
        CLC
        LDA VTX,Y       ;GET X(DIR) THRUST VECTOR
        ADC VX  ;ADD OLD VELOCITY VECTOR
        CMP #$FA        ;IS IT -6?
        BNE .5
        LDA #$FB        ;CLIP TO -5
.5      CMP #$06
        BNE .6
        LDA #$05        ;CLIP TO 5
.6      STA VX
        STA VXP,X       ;STORE NEW SHIP VELOCITY IN X DIRECTION
```

Addressing the Correct Shape Table

Now that we can control our ship in eight directions, we need shape tables for each of these directions. That means eight separate shapes, each twelve bytes long. The plot subroutine that places the shape in the player-missile memory area is virtually the same as that used in our player-missile subroutine discussed earlier in the chapter.
The zero page pointers to our shape and to its eventual storage location in player-missile graphics memory are set up in a subroutine called PLOTSET0 for player #0 and PLOTSET1 for player #1. Since our drawing routine takes the direction into consideration in order to obtain the correctly rotated shape from our shape table, we can find the correct low byte of the shape by the following formula:

SHPL = SHPLO(DIR)

The shape number DIR, which is also our direction, is placed in the Y register so that we can find the low byte pointer to our shape stored in a table called SHPLO. Each of the values in that table are twelve bytes apart starting at #300. The high byte is constant for all shapes.

```
        LDY DIR         ;VALUE FOR DIRECTION OF ROTATED SHAPE
        LDA SHPLO,Y     ;AS INDEX TO PROPER LOW BYTE OF SHAPE
        STA SHPL        ;STORE LOW BYTE POINTER IN ZERO PAGE
        LDA /SHAPEO     ;HIGH BYTE
        STA SHPH        ;STORE HIGH BYTE IN ZERO PAGE
```

If the ship were turned so that it was pointing right, then DIR = 2 and SHPLO(2) = $18. This low byte of the shape table is stored as SHPL. The drawing routine will now plot the second shape from our shape table.

Smoothing the Ship's Movement

Recall that when we updated our ship's position in our last BASIC example, movement wasn't very smooth because of the slow animation frame rate. Potentially, we face a very similar problem here because the ship's velocity vector that controls its next position is only updated every sixth frame. If the ship's position is updated in the same loop as the ship's velocity, it will appear to have very jerky movement, sometimes moving as much as five pixels per frame. It would be better to move the ship in smaller increments more often, even as fast as the scan rate of sixty times per second. Of course, at slower velocities the ship would have to be moved less often. The trick is to control the rate.

If the ship were moving in a direction at full speed, VX = 5, we would want to update the position every frame, but if the ship were moving slowly, VX = 1, we would want to update the screen every fifth or sixth cycle. This means that we should skip plotting the ship at its new position until a number of frames has passed. We could use a counter called SKFLAG to check when we should plot. Naturally, it would be different for each velocity and could be stored in a lookup table for easy access. Its values would nearly be the reciprocal of the velocity. At VXP = 5 we would want to replot each frame so SKFLAG = 1, and when VXP = 1 we would want to replot every fifth or sixth cycle or SKFLAG = 6. The relationship between the ship's movement and its actual velocity VXP isn't exactly linear. If you look in the table you will notice that it is fairly linear at slow speeds, but jumps from twenty pixels every sixty frames at VXP = 3, to thirty pixels every sixty frames at VXP = 4, to a whopping sixty pixels every sixty frames at VXP = 5. Fortunately, this is only a game, and the discrepancy is not noticeable.

The code that determines when the ship is to be moved and replotted for each direction is quite simple. The number of frames to skip, SKFLAG, is obtained from our table based on the ship's current velocity and direction. Five has been added to the velocity so that negative VXP, and VYP values can be indexed, too, using the Y register. Counters for each axis, SCRCNTX and SCRCNTY, that keep track of the number of frames elapsed since the last screen update, are incremented. They are then compared against the values from our table. When it matches, that counter is reset to zero, and the player's position for that axis is updated and plotted. The code for the X axis is listed below.

```
UPDATEX CLC             ;LOAD PLAYER'S HORIZ. VELOCITY
        LDA VXP,X       ;SO NEG #S APPEAR IN TABLE TOO
        TAY             ;USE AS INDEX
        LDA SKFLAG,X    ;GET VALUE FROM TABLE
        INC SCRCNTX,X   ;INC. COUNTER FROM LAST UPDATE
        CMP SCRCNTX,X   ;AT UPDATE TIME?
        BGE .1
        JMP EE          ;NO! DON'T UPDATE
.1      LDA #$00        ;RESET COUNTER
        STA SCRCNTX,X
        LDA VXP,X       ;BEGIN UPDATING PLAYER POSITION
```

Each time the player's position is updated in either axis, the player moves one pixel position in the proper direction. If the velocity is negative, the player moves either to the left or up depending on which axis is being updated. If the velocity is positive, the player moves down or to the right. Remember we decided to move the ship only a single pixel at a time because moving the player in finer increments at higher frame rates produces smoother animation than discontinuous jumps at slow frame rates. The ship's overall velocity is the same, but the ship doesn't strobe as it moves.

The screen has a wraparound feature in this game. This means that when a ship reaches the right side of the screen and exits, it reappears on the left side with its velocity and direction the same. This is true in the vertical direction, too. A ship leaving the top of the screen will reappear at the bottom. All that is needed is a simple check to determine if the ship has reached the screen boundary. If it has, its position is reset to the opposite screen boundary. Since the boundaries of the playfield screen don't quite reach the edges of the television set, we extended the screen boundary coordinates by eight pixels in all directions. The ship's vanishing point should be nearly at the edge. If a ship is traveling to the right, it will vanish at X = 216 ($D8) and reappear at X = 40 ($28). Don't confuse playermissile coordinates with the screen coordinates used in PLOT and DRAWTO statements. The top left corner of the playfield in player-missile coordinates is X = 48, Y = 32. Coordinate 0,0 is offscreen.

Missile Movement

The missiles in this game are controlled entirely by a separate subroutine. The subroutine needs only the player number inputted in the X register to operate correctly. It decides if a new missile can be fired or if one is already on the screen, then moves the missiles appropriately. There is no need to read a joystick for missile direction as in our earlier game, because the missile fires and moves in the direction, DIR, that our ship faces. Also we don't need to worry about updating the ship's position while in the missile subroutine, since the main program loop takes care of this.

The subroutine's initial decision is whether the joystick trigger STRIG0,X ($284,X) is being pressed. When the button is pressed, the subroutine returns a zero, and, if untouched, a one. There are also timers TMIS,X for each missile, to count the number of screen cycles that a missile travels. The actual values are unimportant, except whether they are zero or not. When TMIS,X is greater than zero, the missile is already on the screen and merely has to be moved while checking for screen edge boundaries. However, if TMIS,X is zero, the missile has to be placed initially at the ship's position, and the firing sound started.

Since we only have one missile per player, we can't have rapid fire missiles or more than one missile on the screen at a time. Games that allow a train of five or six missile tracks on the screen at one time are using character set animation or bit mapping, not missiles. We could have allowed the player to reset his missile track each time he pressed the trigger, but instead we decided that you couldn't refire until the missile reached the screen's edge or hit its target. Since many players might hold the trigger down indefinitely we had to test in both branches for whether the missile was already on the screen. If we had omitted this test, the missile track would reset each time a player pressed the trigger. If you wish to change the game to this mode of firing, just remove the test.

Once a missile is launched, it must continue to move in the initial direction, regardless of a change in course of the mother ship. We accomplish this by storing its initial direction at launch DIR,X as DIRMO,X. This direction is used to look up the missile velocity vectors VTX and VTY from tables. The position is updated by the following formula:
XMIS(new) = XMIS(old)+VTX(DIRM0)
YMIS(new) = YMIS(old)+VTY(DIRM0)

The code is as follows:

```
.2      LDY DIRMO,X     ;LOAD Y REG. WITH MISSILE'S DIRECTION
        LDA VTX,Y       ;LOOK UP X DIRECTION VELOCITY VECTOR
        ASL             ;DOUBLE VELOCITY VECTOR
        CLC
        ADC XMISO,X     ;ADD MISSILE'S OLD X POSITION
        STA XMISO,X     ;STORE NEW X POSITION
        LDA VTY,Y       ;LOOK UP Y DIRECTION VELOCITY VECTOR
        ASL
        CLC
        ADC YMISO,X     ;ADD MISSILE'S OLD Y POSITION
        STA XMISO,X     ;STORE NEW Y POSITION
```

These missile shapes are not the square 2 by 2 blocks of our earlier example. They are two pixels set vertically, horizontally, or diagonally in the direction of travel. A chart of their shapes follows:

First, notice that each missile has a different set of numerical values. Missile shape #0 uses the first two bits in the byte, and missile #1 uses the third and fourth bit positions. Therefore, there is a shape table for each missile. The shapes are stored in two-byte pairs for the two scan line high shapes. There is also an index to the low byte to each of these shapes. They are in two tables called MISLO and MISLO1 for each missile set respectively. The missile setup subroutine sets up the zero page pointers to the correct shape and to the storage location in player-missile memory. This area is just below the players or.75K from the start of player-missile memory. In addition, the proper mask for that player is stored as MASK,

Plotting the missile requires erasing the old missile first by ANDing the byte containing all four missiles with the proper mask. This step erases only the desired missile because the mask contains zeros in the missile bits to be erased and ones elsewhere. We then plot the missile on the screen by ORAing with a byte in the missile memory that may or may not contain other missiles. For example, if we wish to erase missile #I in a byte that contains missile #0, we AND it with the mask $FC.

```
    00001010 MEMORY
AND 11111000 MASK
    00000110 RESULT
```

If we move missile #1 to a scan line that contains missile #3 then we ORA its shape with the byte in missile memory for that scan line.

```
    00000100 MEMORY
ORA 00000010 SHAPE
    00001000 RESULT
```

s

The code is shown below:

```
MPLOT   LDY #$00
.1      LDA (SHPMOL),Y;LOAD OLD SHAPE
        AND MASK        ;MASK WITH PROPER MISSILE BEING MOVED
        STA (SHPMOL),Y;STORING IT DOESN'T ERASE OTHER MISSILES
        INY     ;NEXT BYTE
        CPY #$02        ;MISSILE 2 BYTES HIGH
        BLT .1
        LDY #$00
.2      LDA (SHPL),Y ;GET BYTE FROM CORRECT SHAPE TABLE
        ORA (SHPML),Y ;ORA AGAINST OTHER MISSILES ON SCAN LINE
        STA (SHPML),Y ;PUT IN MISSILE AREA
        INY     ;NEXT BYTE
        CPY #$02
        BLT .2
```

Derez Style Explosion

There are many types of explosions suitable for destroying a killed spaceship. One of the more effective methods is to de-rez the object. This technique makes the object appear to slowly disintegrate by randomly flickering the individual pixels until most of the pixels vanish.
The trick is to take the ship and store it into a temporary shape table called DEREZ. We load a random number, then ORA it with another random number to insure that we don't get a number with a few of bits "on". We then AND it with the shape in DEREZ and store it there. If we then AND the original ship's shape with this value three times, we get a significantly degraded image of the ship. We then ORA this with our degraded temporary shape in DEREZ so that we get a less degraded shape than we would get if we performed only the first of the two previous steps. Since we wanted this effect to last at least 30 cycles or half a second, the routine was largely experimentally determined. The image begins to degrade slightly during each cycle but isn't quite steady. Any byte's image can improve slightly but randomly in any frame, but the overall effect is continued degradation. The loop is 48 cycles but the image usually vanishes completely after 30 cycles. An example of two separate passes for a single byte is shown below.

The explosion sound gives the effect of a slow rumble that slowly decreases in volume. We are working with a fairly low frequency and a distortion of zero in channel one and a distortion of 8 in channel 2. The use of two channels with these distortion and frequency values was determined experimentally to produce the best effect. ORing SBANG/4 with $E0 gives us a range of frequencies $EO-$EF. These are stored in AUDF1 and AUDF2. Since both AUDCI and AUDC2 use the lower nibble for volume control, the value for AUDC1 can be obtained by ANDing the frequency with #$OF to mask out $EO. Since the upper nibble is the distortion, then a distortion of 8 can be obtained in AUDC2 by ORing AUDCl with #$80. The equivalent sound routine in BASIC is as follows:

```
10 FOR SBANG=64 TO 0 STEP -1
20 X=INT(SBANG/4)
30 SOUND 0,224+X,0,X
40 SOUND 1,224+X,8,X
50 NEXT SBANG
```

Scorekeeping

The score line is Written directly into GRAPHICS 1 (ANTIC 2) playfield memory beginning at the top of the screen memory. These locations SCREEN to SCREEN+ 19 contain internal character code valves stored initially in a table called SCLINE. During initialization they are moved into screen memory and the scoring digits, SCREEN+7, SCREEN+8 (tens digit, ones digit) for ship #0 and SCREEN+18, SCREEN+19 (tens digit, one's digit) for ship #1 are updated during the game. A diagram showing the score line and its internal character data is illustrated below.

Flow Chart & Game Code

Download SPACEWAR.EXE (Executable program)
Download SPACEWAR.OBJ (Object code)
Download / View SPACEWAR.LST (Assembler listing)
Download / View SPACEWAR.S (Source file)
Download / View SPACEWAR.RAW (As printed in book)

Player Missile Editor

We have included a player-missile editor that will make designing player shapes easy. The grid on the left hand side of the screen supports a shape twenty-two scan lines high in the single-resolution mode. Each of the enlarged blocks is the same shape as the individual pixels on the screen.
The cursor appears in the upper left hand corner. The message at the upper right initially is set in the NOPLOT mode. To put the cursor in the PLOT mode, press the letter P, for the ERASE mode, press the letter E, or return to the NOPLOT mode by pressing N. Pressing any of the arrow keys without the control key moves the cursor. Move the cursor to your desired starting position, and press P for plot. A block fills for each position of the shape, and the byte's decimal and hexadecimal value appears to the side.

As your shape forms, you will begin to see an actual sized player shape emerge in the lower right portion of the screen. You can toggle the player width by pressing the W key. It will progress from single width to double width to quadruple width before returning to single width. Likewise, toggle the player resolution from singleline resolution to double-line resolution by pressing the R key. If you wish to clear the entire screen of your shape, just press SHIFT-CLEAR.

The editor, which is written in BASIC, is somewhat slow. Most of its slowness is due to our decision to write it in graphics mode 8, so that we could plot accurately while simutaneously incorporating text on the screen. Machine language programmers need the hexadecimal values of the player shapes for their shape tables. Owners of the ABC BASIC compiler by Monarch Data Systems will find that the program compiles with no modification, and runs significantly faster.

Download PMEDITOR.BAS (Saved BASIC)
Download / View PMEDITOR.LST (Listed BASIC)

Player-Missile Movement Using Strings

An alternate method of moving player shapes vertically at close to Machine language speed is to take advantage of Atari BASIC's string manipulation routines. One in particular in the form of PO$(N)=SO$ will take a smaller string such as S0$="ABCD" and place it starting at the Nth position of the larger string P0$. Thus, if N=6, P0$ would be as follows:

```
BYTE    VALUE
1       0
2       0
3       0
4       0
5       0
6       ASCII A
7       ASCII B
8       ASCII C
9       ASCII D
10      0
11      0
```

The shorter string could just as easily be our player shape. It would be very easy to move it just by changing the value of N. The only hitch is that we would have to erase its previous position before we moved it, or the player string would begin to repeat itself, or parts of itself, within the larger (256 byte) player area string. Since there is no need to move the player more than two bytes at a time, we can easily accomplish the erase by overwriting the trailing or leading edge of our last position with two trailing null or zero characters at each end.

How BASIC Stores Strings

The biggest problem in this method is -to fool Atari BASIC into assigning the player-missile display area as a string variable. In order to understand how to do this, it is necessary to know how Atari BASIC stores variables in memory. Two areas are set aside in memory. The first, the Variable Value Table Pointer (VVTP), stores eight bytes of information for each variable declared in your BASIC program. The second, the String Array Table, reserves space in memory according to the size specified when you dimension an array. The VVTP table is arranged as in the following example. If it were for a string dimensioned as P0$(255) it would have these values:

Every time BASIC has to access information in a string it goes to the VVPT to find the current address of the string. Byte I contains the variable type which is 129 for a string. Byte 2 contains the variable number. BASIC tokenizes variables and refers to them by numbers, not names. Bytes 3 and 4 contain the value of the offset from the string/array area (STARP) where BASIC reserves memory for all dimensioned variables. The values are in low byte, high byte order. Thus the address of your string is STARP + Byte 3 + 256*Byte 4.
If we change the pointers to bytes 3 and 4 so that they now point to the playermissile area for player #0, we could use string move commands to move our player. To do this we will need to calculate the difference between the player-missile area and where BASIC was planning to store our string in STARP.

VVTP= PEEK(134) + 256*PEEK(135)
STARP = PEEK(140) + 256*PEEK(141)

PLAYER0 =PMBASE*256 + 1024
so
OFFSET=PLAYER0 - STARP

If we take this OFFSET value and divide it into its low and high byte values, then POKE them into bytes 3 and 4, BASIC would think that our strings were actually in the player-missile area. Since we have four players, we have to do this for each group of eight bytes in the VVTP table. Your dimensioned player strings should be the first variables in the program; otherwise, you will have to scan through the table looking for a match.

The following example uses this technique to move four players vertically. Horizontal movement is of course done by setting the player's horizontal position register. The example has four randomly moving, bouncing balls which reverse direction upon collision with another ball or the playfield boundary.

Just to give you a rough idea of the actual memory locations and the change we need to make in the VVTP table for player #0, the values for the examy4e are listed below:

VVTP = 7760
STARP = 10932
PMBASE = 38912 (High byte = 152)
PLAYER0 = 39936

so that OFFSET = PLAYER0-STARP = 29004 then LO = 76 and HI = 113 and they are POKED into bytes 3 and 4 respectively.

Download PMDEMO.BAS (Saved BASIC)
Download / View PMDEMO.LST (Listed BASIC)

While this is a clever method of obtaining fast vertical motion for players, there is no easy way to animate more than a single missile when using strings. The technique is shown more for completeness than for usefulness. We suggest that you use our player-missile subroutine for your games and avoid the complications of using string manipulation.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 6
Vertical Blank & Display List Interrupts

In this chapter you will learn to use Vertical Blank and Display List Interrupts. These interrupts are a powerful aid to the game programmer, who can use them to smooth animation, to enable players to be re-used in the bottom portion of the frame, to allow character sets and color registers to be changed mid-screen, and of course much more.

As you learned in the first chapter, "Vertical Blank" refers to the period of time the television set takes to reposition the electron beam back to the top left edge of the screen after raster scanning or drawing one television frame. Since television frames are generated every 1/60th of a second, there are sixty Vertical Blank periods a second, one for each frame.

Since nothing is happening on the television screen during the Vertical Blank period, it presents an ideal opportunity for a program to change the positions of objects on the screen in order to animate them. If you only knew when the Vertical Blank period occurred, you could take advantage of this time to create smooth and flicker-free animation.

The ANTIC chip in the computer lets you do just that. Since it and the GTIA/ CTIA chip generate the television image, they need to know what is happening on the screen at all times. When Vertical Blank occurs, an interrupt is generated on the Atari's 6502 CPU.

An interrupt is simply a way of telling the computer to stop what it is doing, handle something more important, then return to what it was previously doing. The 6502 checks the interrupt status after executing an instruction. If an interrupt occurred, it saves the Program Counter (marking its current place in the program) and the processor status register onto the stack. It then jumps through one of a number of preset vectors, depending on the type of interrupt that occurred. Once the interrupt has been handled, the program will RTI (return from interrupt instruction) back to the instruction that was to be executed before the interruption.

ANTIC also generates Display List Interrupts. If you recall the discussion on display lists, you remember that the display list is really a program for ANTIC. Each byte in the display list is part of a series of instructions that lets ANTIC know what type of information is to be displayed. Setting the high bit in an instruction in the display list will trigger an interrupt when it is time for the television beam to display the information for the last scan line in that graphics mode line. This type of interrupt is sometimes called a "Raster Interrupt." Remember that the interrupt does not stop the TV raster process but causes the 6502 to execute a series of instructions at this specific time.

Vertical Blank and Display List Interrupts (DLI's) are sent to the 6502 NonMaskable Interrupt (NMI) vector at $FFFA. It is called non-maskable because these interrupts cannot be disabled on the 6502 CPU as can other types of interrupts. Three interrupts go to the NMI vector. They are Vertical Blank, DLI's, and System Reset. Although these interrupts cannot be masked off on the 6502, ANTIC filters the first two. A memorymapped hardware register in ANTIC, NMIEN ($D40E), allows the programmer to enable or disable Vertical Blank and Display List Interrupts. Another register, NMIST ($D40F), shows which interrupt actually occurred. When an NMI occurs, the 6502 goes to an OS routine to find out what caused the interrupt. If a DLI occurred, this routine vectors through page-two vectors to a DLI service routine that changes one or more graphics registers which control display. On the other hand, if a VBI occurred, the OS routine saves the Accumulator, X and Y registers on the stack, then jumps through the appropriate page-two vector to the Vertical Blank routine. And if System Reset was pressed, it goes directly to the OS warm start vector for system reset ($E474).

Vertical Blank Interrupts

The Atari computers use the Vertical Blank Interrupt for many housekeeping chores. When the Atari is first powered up under normal conditions, the Vertical Blank Interrupt is enabled. A special list of vectors is placed in page two of memory. The operating system places the list of vectors in page two because these are RAM locations and the user can change their contents to point to his own interrupt service routine. System Reset is the only interrupt that cannot be routed through page two.
The operating system's Vertical Blank Interrupt (VBI) service routine is a twostage process. When an interrupt occurs, the computer is sent to a vector in the operating system, which in turn sends it to a routine to determine what type of interrupt has occurred. It then jumps through the appropriate vector on page two. It is called two-stage because it goes through two vectors on page two. This allows the programmer to use either his own VBI routine or that of the OS, or both.

The OS uses the VBI routine to transfer data from some of the special hardware registers in Atari's custom chips to shadowed locations in RAM and vice versa. For example, it reads the color register information, the location of the display list, and other graphics control information placed in RAM by the user, and writes them to their hardware addresses. It also reads many hardware registers such as game controllers and light pen and places them in RAM for the user. It updates the clock at locations 18-20 ($12-$14), and updates the timer for the attract mode which cycles the colors if a key has not been pressed for several minutes. It even takes care of the repeat action on the keyboard keys. Because the VBI routine updates many registers that are used by the average program, most programmers don't wish to replace it entirely by their own routine, since in many cases it would mean duplicating at least some of the procedures that are handled automatically by the OS.

The vectors are called Immediate VBlank and Deferred VBlank. Atari engineers split the Vertical Blank period into two separate sections because a VBI can extend for about 20,000 cycles or nearly all the time available before the next VBI. The Immediate VBlank portion refers to the time critical period when the electron beam is actually offscreen. It is here that shadowing and hardware registers are updated. Once the OS VBI routine has completed its housekeeping chores, it jumps through the Deferred VBlank vector, which is set by the OS to point to a routine that will restore the registers and return the computer to its position before the interrupt. The OS VBI routine will abort if the computer was in the middle of a time critical I/0 routine such as sending data to a cassette or disk drive. In such a case, the Deferred VBlank vector is bypassed. Unless you are using disk 1/0 where your code must be short enough not to delay shadowing updating, you can use Deferred VBlank without being concerned.

A major question asked by many programmers is: How much time do I have in the VBI routine to execute my code? To answer this we must examine the TV process again.

A television image is composed of 525 lines. Because of what is called interlacing, only half are drawn per frame, and some of these are offscreen due to normal vertical overscan. It takes the TV 63.5 microseconds to draw a single line. This includes the time it takes to shut the electron beam off and reposition it back on the left edge one scan line below. The length of Vertical Blank is equivalent to 22 scan lines or roughly 1400 microseconds. With the Atari's 6502B CPU running at 1.79 MHz, 1400 microseconds is equivalent to approximately 2500 machine cycles (1400 X 1.79 = 2506). So, how much can you do "inside" the Vertical Blank period? Not very much, if you are trying to accomplish things like moving players and scrolling the screen solely inside the Vertical Blank period. The important thing to remember is that one television frame is 1/60th of a second, which is equivalent to 16,666 microseconds or just about 30,000 machine cycles. This means that there is a maximum of approximately 30,000 machine cycles BETWEEN Vertical Blanks. ANTIC delays processor time in order to fetch screen data and look up character data during the screen drawing process. A programmer using an extensive Vertical Blank Interrupt routine will want to be sure that the routine is finished before the next VBI occurs; otherwise, his routine will be aborted in the middle unless very special precautions are taken. A programmer will also want to be sure that his VBlank routine is not so long that it causes serious delays to the program code that is running outside VBlank.

Programmers often ask if the 2500 cycle VBlank time constraint seriously limits the amount of time for smooth offscreen animation and updating. The answer is No! True, all of the graphics updating must be performed while the beam is offscreen, but you can also gain some additional time. Remember what a normal display list looks like. It begins with 24 blank lines. That gives 24 by 63.5 microseconds/line or another 2728 cycles that can be considered offscreen. High scores, player scores and messages generally cover the top few lines too. This adds additional time offscreen. Obviously if your Vertical Blank routine is even longer, you should be OK as long as you do all of your graphics updating within the first 5200 cycles of your routine. Figure that you have a maximum of 20,000 cycles (4500 instructions) in Deferred VBlank, but you must finish before the next VBI or your program will crash.

While programmers have written entire games in Deferred VBlank, only certain operations should be put in VBlank. All graphics updating, including scrolling the screen, moving player-missile objects, and changing color registers, should definitely be done in VBlank. In addition, collisions should be checked, and joysticks or paddles

should be read. This is also the best place to implement time critical sound routines. Everything else, including calculations, should be in your main code outside VBlank.

Setting up a Vertical Blank Interrupt is a simple process since the OS has a routine to set up the vector for you. All it involves is storing the address of your VBI routine in the vector table on page two of RAM. If a VBI has occurred between the time the two bytes that make up the vector were stored in the table, the 6502 would jump through an erroneous address, and the program would become erroneously lost. The OS setup routine automatically assures this never happens.

Setting up the routine is simple. Load the Accumulator with 7 if you are setting a Deferred VBlank routine, and 6 if you are setting an Immediate VBlank routine. The X register is loaded with the high order byte of your routine, the Y register with the low order byte. You then JSR to the OS SETVBV routine at $E45C.

Example:

```
SETVBV  .EQ $E45C
        LDA #$07        ;DEFERRED
        LDX /VBLANK     ;HIGH BYTE OF USER ROUTINE
        LDY #VBLANK     ;LOW BYTE
        JSR SETVBV
```

There are two possible exit points from a VBlank routine depending on whether Immediate or Deferred VBlank is used. If the programmer uses the Immediate one and still desires to use the OS VBI routine, the vector is $E45F (SYSVBV). If the Deferred VBlank is used or the programmer does not want the OS VBI routine to execute the vector, then it is $E462 (XITVBV). The XITVBV routine pulls the registers off the stack and does a RTI (Return from Interrupt).

Display List Interrupts

Display list interrupts are generally used to change the color registers midscreen or switch character sets in use. These changes can be made very rapidly since they are short and usually modify only a few bytes. Take care so that changes of this type are offscreen during Horizontal Blank, or they will appear crude and annoying. When ANTIC encounters the DLI instruction, it completes the last scan line for the mode it is drawing, then services the interrupt. This means in effect that the interrupt must be set during the mode line above the one you want the interrupt to effect.
The period of Horizontal Blank is seven microseconds. Horizontal Overscan is another 3.31 microseconds. These 10.31 microseconds mean approximately eighteen machine cycles offscreen. In order for an interrupt routine to synchronize with Horizontal Blank, a special hardware register in ANTIC freezes the 6502 until Horizontal Blank occurs. Thi's register at $D40A is known as WSYNC. Writing to this location pulls down the ready line on the CPU until Horizontal Sync. If you insert a STA WSYNC instruction, then change the value in a color register, color won't be changed in The Middle of the current line but will go into effect when the beam is off the left edge of the screen one scan line lower.

What if eighteen cycles are not enough time to make all the changes you need? There are several approaches that can be taken, and the proper one depends of course upon the situation.

As with the VBI, you need not worry about crashing the program because the code does not fit in the Horizontal Blank time or even within the time it takes to do the entire scan line. Only if the code is so long that another DLI occurs before the previous one has finished would you be likely to run into problems and crash the program. There is no reason to believe that you must complete the interrupt routine by the end of the scan line. There are some programs that have one DLI set at the top of the screen and do not return from the interrupt until the entire screen has been drawn, hundreds of scan lines and thousands of machine cycles later. The interrupt routine is used to control graphics information to the screen, line by line. A DLI routine written in this manner is called a kernel.

Programmers using DLI's for simple screen changes should decide if all of them must be made on a single line. Perhaps only a few changes need be made immediately. The rest can be made on the next line by waiting again for Horizontal Sync. All the changes must be made on a single line, if there is a major screen division between the score line and the playfield requiring different colors and character set. In this case, it might be practical to insert a zero in the display list. A zero is the blank line instruction in a display list. Changes could be made past Horizontal Blank on this line while the raster beam is drawing it and still be invisible to the viewer.

There is one more important point for those readers who still need to count cycles. Although a horizontal line takes 63.5 microseconds, you do not have 113 cycles per line (63.5 x 1.79 = 113.6). This is because ANTIC's Direct Memory Access (DMA) ability allows it to freeze the 6502 CPU and steal cycles in order to get screen data from screen memory, player/missile data from player/missile memory, and to look up character set data. It even steals cycles to look at the display list so it knows what type of information to display. The amount of time stolen per scan line can vary. ANTIC steals a cycle for each byte in memory it must access. This DMA is controlled by a hardware register called DMACTL at $D400 (54272 decimal). The OS VBlank routine rewrites the value found at its shadow (SDMCTL) at $22F (559 decimal) every VBI. By setting bits in SDMCTL, the program can control screen width from either 128, 160, or 228 color clocks. Selecting screen width tells ANTIC it may steal cycles from the 6502 to access the display list and screen memory in order to display information. Bits are also set here so that ANTIC may steal cycles to look up player and/or missile data in memory.

If you want an idea of how much time you lose from DMA, try the following example in BASIC.

```
10 FOR LI=1 TO 1000:NEXT LI
```

Then try:

```
10 POKE 559,0:FOR LI=1 TO 1000:NEXT LI:POKE 559,32
```

The second example is approximately 30% faster. The first POKE disables ANTIC's DMA, and the screen becomes black. The second POKE restores DMA after the loop is completed so we know how long it takes. If you were to try the first example in different graphic modes, you would find Graphics 8 the worst case, This is because Graphics 8 uses much more memory, so ANTIC must steal more cycles in order to access more screen memory. However, the Graphics 0 text mode isn't much faster because ANTIC also steals cycles while retrieving the characters set.

Display List Interrupts are even simpler to set up than Vertical Blank Interrupts. Since there is no DLI enabled when the program takes control, the program simply has to store the low byte-high byte address of the routine into the vector on page two (VDSLST-$200,$201). Once the vector has been stored and the display list is set for an interrupt, NMIEN is set to enable DLI's. Bit 7 of NMIEN enables DLI's. Bit 6 enables VBI's. Storing a $CO (192 decimal) is all that is needed to enable the DLI.

Example:

```
VDSLST  .EQ $200        ;DISPLAY LIST INTERRUPT VECTOR
NMIEN   .EQ $D40E
        LDA #VBI        ;LOW BYTE OF DLI ROUTINE
        STA VDSLST
        LDA /VBI        ;HIGH BYTE OF DLI ROUTINE
        STA VDLIST+1
        LDA #$CO
        STA NMIEN       ;ENABLE DLI'S
                        ;WITHOUT DISABLING VBI'S
```

Since the OS does not save the Accumulator, X and Y registers for a DLI as it does for a VBI, the user must save any registers used in the DLI routine on the stack upon entering the interrupt routine and restore them before exiting. If this is not done, the program will unquestionably crash.

The power of DLI's is obvious. With a DLI, a program can easily switch character bases from the standard ROM text set to a redefined character graphics set. Without a DLI, the programmer could only redefine characters that would not be used in the text display. DLI's enable you to change color registers mid-screen. A good example is a game like Sea Wolf (a submarine game) in which the background color changes from sky blue to ocean blue partway down the screen. Without a DLI to change the background color, one other playfield color register would have to be used for either the sky or ocean, thus limiting further the amount of color available for the screen. Multiple DLI's can be used to control screen scrolling in games like Frogger in which different bands on the screen scroll in different directions at different speeds.

DLI's can change player/missile horizontal positions and colors so players can be reused down the screen as long as vertical positions do not need to overlap in the same player. A player, which is actually a long vertical stripe, can have several different smaller images. The DLI allows you to snip the strip apart and reposition those pieces in the lower portion by changing the player's horizontal position register. If vertical movement is required, you must be careful that the different images don't cross the boundary. Since collision registers are set by the hardware when

collision occurs, collision registers can be read within the interrupt routine so that hardware collision detection is not lost by reusing the players. A good example of reusing the players is the Atari Galaxian cartridge.

BASIC can use a DLI to change one or more color registers at a particular spot mid-screen, if a short Machine language routine is added. The example below changes the background color register in the text mode to orange and darkens the color of the characters so they show up against the background. This is controlled by the color in playfield 2. The change occurs in line 12 so that the interrupt is set in the mode line preceding the change. The modified display list is as follows.

```
        112
        112     Blank 24 scan lines
        112
        66      |LMS for BASIC 0 (ANTIC 2) (64+2)
        64      |Low byte of screen memory
        156     |High byte of screen memory
        2       second mode line
        2
        2
        2
        2
        2
        2
        2
        2
        130     Eleventh mode line + DLI (128+2)
        2
        2
 .
 .
```

Since the display List is virtually identical, except for the modification in the eleventh mode line, we need only change that byte to activate our DLI. This is at the DLIST+15 byte.

```
5 REM FIND DISPLAY LIST
10 DLIST=PEEK(560)+256*PEEK(561)
15 REM INSERT INTERRUPT INSTRUCTION
20 POKE DLIST+15,130
25 REM READ IN DLI SERVICE ROUTINE
30 FOR I=O TO 19
40 READ A:POKE 1536+I,A:NEXT I
50 REM POKE IN INTERRUPT VECTOR
60 POKE 512,0:POKE 513,6
70 REM ENABLE DLI
80 POKE 54286,192
90 DATA 72,138,72,169,38,162,90
100 DATA 141,10,212,141,26,208
110 DATA 141,24,208,104,170,104,64
```

The actual Machine language DLI service routine is as follows:

```
D01A:           00010   COLBK   .EQ $DO1A
D018:           00020   COLPF2  .EQ $D018
D40A:           00030   WSYNC   .EQ $D40A
4000: 48        00040   PHA             ;SAVE ACCUMULATOR
```

```
4001: 8A        00050   TXA
4002: 48        00060   PHA             ;SAVE X-REGISTER
4003: A9 52     00070   LDA #$52        ;DARK COLOR FOR CHARACTERS
4005: A2 26     00080   LDX #$26        ;ORANGE BACKGROUND
4007: 8D OA D4  00090   STA WSYNC       ;WAIT
40OA: 8D 1A DO  00100   STA COLBK       ;STORE COLOR
40OD: 8D 18 DO  00110   STA COLPF2      ;STORE COLOR
4010: 68        00120   PLA
4011: AA        00130   TAX
4012: 68        00140   PLA
4013: 40        00150   RTI
```

Care must be taken when using multiple DLI's. There is only one vector for a DLI Setting NMIEN enables DLI's immediately and therefore does not wait for the next frame. Your program must insure that the proper routine will be executed. Three methods are available. The first method is to use a variable as an index that is incremented by each DLI The program then branches to the appropriate routine depending on the value of the index. The second method is to read the vertical line counter and branch to the appropriate routine. The third method is the cleanest. Each DLI routine resets the DLI vector on page two (VDSLST) to point to the next. The DLI is enabled within Vertical Blank and VDSLST is reset to point to the first DLI routine in VBlank.

The following example is a simple DLI routine to change the background color of a text screen similar to the Sea Wolf example.

```
00010   VDSLST  EQ $200 ;DISPLAY LIST INTERRUPT VECTOR
00020   SDLSTL  EQ $230 ;STARTING ADDRESS OF DISPLAY LIST
00030   WSYNC   EQ $D40A
00040   NMIEN   EQ $D40E
00050   COLPF2  EQ $DO18
00060   PAGEO   EQ $FO
00070   LDA SDLSTL      ;FIND DISPLAY LSIT
0080    STA PAGEO       ;DISPLAY LIST LOW BYTE
00090   LDA SDLSTL+1
00100   STA PAGE0+1     ;SAVE HIGH BYTE
00110   LDY #$08        ;3RD TEXT LINE INA A GRAPHICS 0 DISPLAY LIST
00120   LDA (PAGEO),Y   ;GET DISPLAY LSIT INSTRUCTION
00130   ORA #%10000000;TURN ON HIGH BIT ($80)
00140   STA (PAGEO),Y   ;AND STORE IT BACK IN THE DISPLAY LIST
00150   LDA #DLI        ;DLI LOW BYTE
00160   STA VDSLST      ;STORE LOW BYTE OF OUR ROUTINE IN DLI VECTOR
00170   LDA /VBI        ;GET HIGH BYTE OF OUR DLI ROUTINE
00180   STA VDSLST+l    ;STORE HIGH BYTE OF VECTOR
00190   LDA #%11000000; ENABLE DLI AND KEEP VBI ($CO)
00200   STA NMIEN
00210   RTS     ;AN RTS SHOULD RETURN THE PROGRAM
00220   ;BACK TO THE DEBUGGER WHEN RUN
00230   DLI     PHA     ;SAVE ANY REGISTER THAT WILL BE USED
00240   LDA #$A2        ;DEEP BLUE
00250   STA WSYNC       ;WAIT FOR SYNC FOR NEXT LINE
00260   STA COLPF2
00270   PLA     ;RESTORE ACCUMULATOR
00280   RTI     ;RETURN TO MAIN PROGRAM
```

Binary representation was used for clarity in this example.

Kernels

A kernel is a special DLI routine designed to control graphics information on a line-by-line basis for the entire screen. It does this by monitoring the VCOUNT (vertical line counter) register. The most frequent use is for producing multi-colored players. Even the player width can be changed on a line-by-line basis. For example, a cowboy image could be made out of one player; a broad white hat defined at double width changes a few scan lines down to a slim pink face, then changes a few scan lines down to a brown cowboy suit and finally to black boots; four colors and two different resolutions in a single player. Another example is the players in Atari's Basketball cartridge. You cannot create multi-colored players like these without Display List Interrupts because DLI's are keyed to playfield vertical positions, not player positions.

Kernels are difficult to use because graphics information changes only during Horizontal Blank. Kernels also drastically reduce the amount of time available for program logic, since most of the 6502's time is spent writing graphics information and waiting for Horizontal Sync on the next line. Since virtually no computation time is

available during display time, and only about 4000 cycles are available during Vertical Blank and overscan time, kernels are limited to simple skill and action games.

Multi-Colored Joystick Controlled Player

The following example illustrates the use of a kernel to produce a joystickmovable, multi-colored player. The player is 15 bytes high, but it also includes two zero bytes on each side so that it erases itself as it moves. There is no actual erase feature in this example. The player is controlled by a joystick. This is not the standard routine but an alternate one that uses table lookup to determine its horizontal and vertical movement. The formulas are as follows:
PLAYRH = PLAYRH + HOFF(STICK)
PLAYRV = PLAYRV + VOFF(STICK)

For example, if the joystick points to the right and up, STICK has the value 6. HOFF(6)=$02, and VOFF(6)=$FE or a -2. The player position changes to a location two scan lines up and two pixels right.

The kernel itself is designed to change the player's color four different times. To accomplish this it must wait for the vertical line counter to reach the player's position. Since this counter, called VCOUNT, actually increments every other scan line, we divide the player's vertical position by two. We add one because we actually want to start one line later. We have to wait for each horizontal scan to test for whether VCOUNT has reached our player's position. If it has, we then begin fetching color to put in the player's color register. Since the color changes every three scan lines, we must wait for three more horizontal scan lines before changing colors. The fact that we have to do a WSYNC for each scan line prevents us from doing any calculations outside of Vertical Blank.

Download MULTI.EXE (Executable program)
Download MULTI.OBJ (Object code)
Download / View MULTI.LST (Assembler listing)
Download / View MULTI.S (Source file)
Download / View MULTI.RAW (As printed in book)

Splitting Screen Horizontally Using DLIs

Our second example of a kernel produces a split color screen, but this time the split is across the vertical axis with the left portion of the screen black and the right pink. The kernel takes advantage of a careful timing loop after a WSYNC instruction. It waits until the beam is offscreen for each scan line, so the beam always reaches the same horizontal position before changing the background color register to pink. The loop is a simple countdown from six to zero. The background is restored to black for the next scan line immediately after the WSYNC. Notice that the color change is fed directly to the hardware color register. Remember that the kernel operates outside of VBlank, and the shadowed background color register doesn't get copied until VBlank. Thus, color changes must feed directly to hardware. We do have a joystick-controlled player-missile routine operating in VBlank in order to show that the kernel and it are entirely independent.

Download MIDSCRN.EXE (Executable program)
Download MIDSCRN.OBJ (Object code)
Download / View MIDSCRN.LST (Assembler listing)
Download / View MIDSCRN.S (Source file)
Download / View MIDSCRN.RAW (As printed in book)

Using DLIs to Create Animation

The third example shows a motorboat crossing water. A kernel controls an eight-scan line segment at the surface so that it appears that the water has waves. The interrupt occurs on the fourth text line. Essentially, each bit of an eight-bit random number is sequentially shifted right into the carry bit while in a loop. If the bit is set, the routine changes the background color register to light blue. If it isn't, it remains dark blue. Once the eight scan lines aye drawn, the routine sets the background color to dark blue for the remainder of the screen. A new wave pattern is picked every eight VBlanks: otherwise, the waves would change too quickly.

The white motorboat consists of two players, get side by side. It is moved in the VBlank interrupt routine which also provides the new random number every eighth cycle. The sequence is clocked through the internal clock at $14 called RTCLOC. ANDing the value with #$07 produces a positive value if you. don't have an exact even multiple of eight because some combination of the lowest three bits (1 -7) is set. In that case, the routine branches to a new random number.
There are many other possible examples using kernels. Advanced programmers might attempt to modify the second example so that a non-joystick-controlled player might have its horizontal position changed during the horizontal scan for each line. It is possible, using tight timing loops, to reuse the player on the same scan line.

Download WAVES.EXE (Executable program)
Download WAVES.OBJ (Object code)
Download / View WAVES.LST (Assembler listing)
Download / View WAVES.S (Source file)
Download / View WAVES.RAW (As printed in book)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 7
Games That Scroll

An effective means of showing a much larger environment than can actually fit on the screen at any one time is to scroll the screen. Examples where only a portion of the total information can be shown are numerous, and range from the simple listing of a long BASIC program to the scanning of a large terrain map such as those used in the war game Eastern Front.

Perhaps its best use, however, is the dynamic feel that it gives to scrolling arcade games like Super Cobra, Zaxxon, and Caverns of Mars. These games have multi-screen worlds which scroll on or off the screen as a player's ship moves. These games show only a window or part of the entire background world at one time. They differ from games that have background stars and aliens that appear to be traveling toward you from top to bottom. Scrolling games have objects or terrain in relatively stable positions within the game's world. They can be reached by traveling to that particular section of the world.

There are two ways to scroll screen data. Most conventional micro-computers move the data through a fixed screen area. This requires an enormous memory shuffle involving many thousands of bytes. In the case of an Apple computer, rough horizontal scrolling can be achieved by shuffling all 8K of screen memory data. Other computers that are endowed with character set graphics can reduce the workload to a manageable 1K of screen data. In nearly all cases, scrolling is coarse and jerky because individual bytes of data represent either seven or eight individual pixels. Moving one byte moves many pixels at one time.

An easier method, and the one the Atari engineers chose, is to move the screen window or screen area over the data. Luckily, the ANTIC graphics microprocessor uses the Load Memory Scan (LMS) instruction to determine where it finds its screen data. A normal display list will have one LMS instruction at its beginning. The RAM area that it points to has the screen data in linear sequence. If we change the starting screen address by manipulating the operand bytes of the LMS instruction, a primitive coarse scroll can be achieved. In effect, we have moved the screen window over the data just by changing two address bytes. This is exactly equivalent to the conventional method of moving all of screen RAM.

Atari computers are also equipped with both vertical and horizontal fine scrolling registers. These enable the computer to scroll the screen in steps smaller than character or pixel size. While the effect is impressive, the technique requires implementation at a Machine language level and will be discussed later.

Coarse Vertical Scrolling

Coarse vertical scrolling, similar to the way a long BASIC program listing scrolls, is quite easy to implement from BASIC. If we consider a Graphics 0 playfield, each row of character data consists of forty characters. The second row of data begins forty bytes beyond the first row of data. When ANTIC begins fetching display data from screen RAM, it begins with the first row, using data beginning at the address in the operand of its LMS instruction. It fetches forty bytes sequentially for each of the twenty-four lines of character data.

It would be very easy to modify the two-byte operand of the LMS instruction so that it begins fetching screen data beginning with the RAM screen address of the second line or forty bytes later. To do this we need only modify the fourth and fifth bytes of our display list. The 0th, first, and second bytes are blank 8 scan-line instructions, the third is the LMS instruction, the fourth and fifth bytes are the low byte and high byte address of screen data respectively. Each time we scroll the screen upward one mode line, we need to advance the LMS address by forty bytes. You need only make sure that the low byte doesn't exceed a value of 255. If it does, you adjust the low byte by subtracting 256, and increment the high byte by one. The example below begins its scrolling at the very bottom of memory and scrolls until near the top of memory.

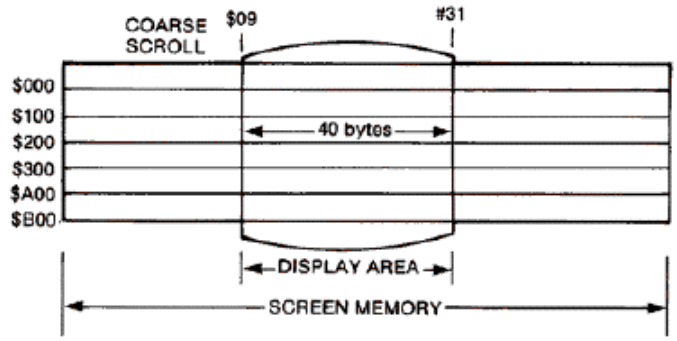
Download VERTSCR.BAS (Saved BASIC)
Download / View VERTSCR.LST (Listed BASIC)

As you watch the screen scroll upward, you will notice sometimes that the scrolling becomes momentarily discontinuous. This is a problem that occurs only in BASIC when scrolling is done outside the Vertical Blank period. Sometimes BASIC hasn't had time to POKE both the high and low bytes into the display list before the interrupt occurs. In the event that only one byte has been POKEd, the starting address in the LMS instruction is incorrect and the screen jerks. A more serious discontinuity occurs at 4K boundaries because ANTIC can't cross a 4K boundary using a single LMS instruction. Using LMS instructions for each mode line should solve the problem.

Coarse Horizontal Scrolling

Horizontal scrolling is much more difficult to do than vertical scrolling. The problem is that screen data is organized serially. Attempting to scroll to the right causes the computer to try to display data in the first line that would normally belong in the second line. We begin to get a snaking scroll throughout the entire display as the leftmost byte on each line will be scrolled into the rightmost position of the next higher line, and the following bytes shift one to the left.

The solution is to expand the screen data area and break it up into a series of independent horizontal data areas for each mode line. Each RAM area is still one-dimensional and serial in nature, but it extends much further than the screen shows. Since the purpose in scrolling a screen is to show more information than the screen can hold, we have allotted extra RAM for each mode line to hold this information. Now, if we move the screen window over the screen data, the data moves into view without affecting the data in any of the subsequent mode lines.



As a first step, you will need to organize your screen data by allocating a certain number of bytes of RAM for each mode line. Since the LMS operand consists of a low and a high byte, it is much easier to calculate addresses if each subsequent mode line is exactly 256 bytes long. This simplifies calculations since the high byte screen address for each mode line is a page, or one unit, apart. Since each mode line accesses a different page of screen memory, a special display list must be constructed that has an LMS for each mode line. As an example, we will horizontally scroll a BASIC mode 2 (ANTIC 7) screen. There are twelve mode lines on the screen, each using 256 bytes of memory, or a total of 3K of RAM memory. Since ANTIC displays twenty bytes per line, our world consists of nearly twenty-three screens of data arranged end to end horizontally. If we choose to use the lowest portion of RAM beginning with zero page which we know has interesting data, the display list is as follows:

```
        112
        112     8 blank scan lines
        112
        71      LMS for 0th row BASIC 2 (64+7)
        0       Low byte screen memory
        0       High byte starting zero page
        71      LMS for first row
0       low byte
1       High byte starting page one
71
0
2
.
.
71      LMS for eleventh row
0       Low byte
11      High byte
65      JMP to beginning of display list
```

```
0       Low byte
6       High byte page six
```

To execute a horizontal scroll, each and every LMS operand in the display list must be incremented for a rightward scroll and decremented for a leftward scroll. Since we set up screen memory one page per mode line, only the low byte address need be changed. Also, since the entire screen scrolls as a unit, at least in our case, all of the low byte operands are changed at the same time and have the same value. Program logic should also insure that the image doesn't scroll beyond the limits of the allocated RAM areas, otherwise the display will become garbaged.

The following example uses the preceding display list to scroll our screen horizontally through the lowest twelve pages of memory used by the OS and DOS. We have also placed our display list in page six (sixth row of our display) so that you can watch the data in the list change as the screen actually scrolls. Changes are made to each of the low byte operands of the LMS instructions via a FOR ... NEXT loop. They are the fourth, seventh, tenth, etc., positions in the display list. This is at DLIST+(3*J)+1 where J goes from 1 to 12. Also, since we can't scroll into screen RAM beyond the end of each page without messing up the display, program logic dictates that the right edge of our window does not exceed 255. Therefore, the left edge must not exceed 255 - 20 = 235 bytes. The screen is scrolled from 0 to 235.

Download HORIZSCR.BAS (Saved BASIC)
Download / View HORIZSCR.LST (Listed BASIC)

Obviously, the technique doesn't produce smooth wraparound, although the program repeats itself by beginning again at the left edge of screen RAM. The fault isn't with the technique but in the screen data. If you want wraparound, the data on the last screen must match that of the first screen. This would mean in our example that the last twenty bytes of each line exactly match the first twenty bytes of each line.

It wouldn't be difficult to combine horizontal scrolling with vertical scrolling to get diagonal scrolling. Since we achieve scrolling by adding or subtracting one from the LMS operand, and vertical scrolling by adding or subtracting the line length from the LMS operand, diagonal scrolling occurs when both operations are done simultaneously. If we wish to scroll down and to the left we would add 256 bytes to the high byte and 1 byte to the low byte of each scan line. While this might appear to be a simple procedure in this special case, any other configuration of screen RAM will involve two-byte additions.

Fine Scrolling

We can create much finer scrolling in steps smaller than pixel or character size by enabling the fine scrolling registers. There are two of these, one for horizontal scrolling (HSCROL) at $D404, and one for vertical scrolling (VSCROL) at $D405. They are enabled by setting appropriate bits in the display list instruction bytes for the mode lines in which we want fine scrolling. Vertical fine scrolling is enabled by setting bit 5 in the instruction bit. Similarly, horizontal fine scrolling is enabled by setting bit 4 in the instruction byte.

Function	set bit	add decimal	add hex
Load Memory Scan	6	64	40
Vertical scroll	5	32	20
Horizontal scroll	4	16	10

For example, if we were to enable fine horizontal scrolling in our example above, each of the LMS instructions would be 64+7+16 = 87.

The two fine scroll registers each have a limited range equal to 16 scan lines (0- 15) in the vertical direction, and 16 color clocks (0-15) in the horizontal direction. If we attempt to scroll beyond these values, ANTIC simply ignores the higher bits of the scroll registers. In order to achieve fine scrolling over a wider range, we need to combine fine scrolling with coarse scrolling. The technique is to fine scroll the image until the amount of fine scrolling equals the size of the pixel or character. Then you reset the fine scrolling register back to zero and coarse scroll the screen one unit. An example of fine scrolling in the vertical direction is shown in the following picture:

A minor problem occurs when you attempt to fine scroll data into the bottom mode line. Images tend to pop in suddenly rather than scroll in smoothly because there is no buffer of data for the next line. To get proper fine scrolling you will need to dedicate one mode line to act as a buffer. This can be done by simply not setting the vertical scroll bit in the display list instruction in the last mode line. The window will now scroll without the unpleasant jerk, but will be shortened by one mode line.

Fine scrolling in the horizontal direction is complicated by the fact that ANTIC sets aside a buffer on either end of the mode line so that portions of the pixel or character can scroll on or off the screen smoothly. When fine scrolling is enabled in the horizontal direction, ANTIC fetches more information (8 bytes/mode line) than the normal playfield (160 color clocks wide. It sets aside a buffer of sixteen color clocks on each side and retrieves forty-eight bytes of information per line rather than the usual forty bytes. This is usually not a problem if the programmer has organized his screen data in long horizontal rows. If, on the other hand, the fine scroll register is set to zero, the window is actually looking at the sixteen color clock buffer rather than the first byte in the display for that mode line. For example, in a BASIC mode two (ANTIC 7) line where each character is eight color clocks wide, ANTIC places the first two bytes of the mode line in the buffer so that the first two bytes or sixteen color clocks aren't displayed in the screen window. BASIC mode 0 (ANTIC 2) and ANTIC mode four displays, that use characters only four color clocks wide, will be offset by four bytes. This problem can be virtually corrected by advancing the vertical fine scroll register by fifteen color clocks. You will still be off by one color clock, but since you are planning to scroll the screen anyway, it doesn't matter.

The example illustrated below shows what is involved in fine scrolling ANTIC 2 (BASIC 0) or ANTIC 4 characters horizontally. Each of these characters is only four color clocks wide so that it requires only four fine scroll increments before you need to coarse scroll and reset the fine scroll register. The fine scroll register goes backward from 15 to 12 before being reset to 15. The top row shows that ANTIC actually places the first character into the buffer rather than into the first visible screen position if the fine scroll register is set to zero. The second row shows the correction obtained by setting the fine scroll register to 15. While it seems that a value of 16 should correct it totally, in fact the high bit would be ignored, and you would obtain the result of the top row.

Whether the movement be vertical, horizontal, or diagonal, the most difficult aspect of using Machine language to scroll the screen is calculating the LMS addresses for each of the mode lines. Multiplication of numbers other than powers of two, requires a complicated and time-consuming subroutine. Fortunately, several special or contrived scrolling cases make the calculation fairly easy. We will discuss these situations first.

Vertical Scrolling

Pure vertical scrolling is obviously the easiest case. It requires only one LMS instruction because screen memory is continous throughout the display. Still, if we had to calculate the starting address of the display from scratch each time we scrolled, we would still need an elaborate multiplication subroutine because multiplication by forty bytes or #$28 is not an easy feat. Luckily, rough scrolling is usually done a line at a time. Since we know the current address of the screen, we need only add forty bytes to this address to scroll the screen upwards, or subtract forty bytes to scroll the screen downwards. This only requires a double-byte add or double-byte subtraction.

Horizontal Scrolling

Pure horizontal scrolling, on the other hand, can become very complicated and require dozens of multiplications, if the screen size isn't a special case. The most common special case is one where each mode line is exactly 256 bytes or one page in memory. The LMS address for the subsequent mode lines are exactly $100 apart in memory. This becomes very convenient since you don't need to do an addition or subtraction on each of the current LMS addresses in order the rough scroll. Instead you can merely store the new value of the horizontal rough scroll offset into each of the low byte addresses for the LMS instructions. If you are in GRAPHICS 1 (ANTIC 6), this means that you have to complete twenty-four store operations in a simple loop. Perhaps the best example of this is our scrolling game example in the final section of this chapter. There are twenty-two rows of ANTIC 6 characters, each one page in length for a total of 5 1/2K of screen memory. The variable XS determines the rough scroll position and the variable FS, the fine scroll position. Fine scroll naturally progresses from 0 to 7 before there is a need to increment XS. However, as we discussed earlier, the actual value placed into the fine scroll hardware register is HSCROL = 15 - FS. Wraparound in this example is at XS=235. When XS exceeds 235, XS is reset to 0.

All twenty-two low byte addresses are updated in the display list during VBlank. The first of these addresses is at NDLIST+8. The next address is three bytes later. We can take advantage of indirect addressing using the Y register if we increase the Y register by 3 between store operations. The code below illustrates the technique.

```
        LDY #$00          ;COUNTER
.4      LDA XS            ;POSITION AT SCREEN LEFT
        STA NDLIST+8,Y    ;LOACATION OF FIRST LOW BYTE ADDRESS
        INY               ;LOW BYTES ARE THREE APART
        INY
        INY
        CPY #$4B          ;END OF LIST?
        BNE .4            ;NEXT ELEMENT
```

Obviously, if we choose to include vertical scrolling as well, we need only increment each of the high byte addresses of the LMS instruction to rough scroll the screen upwards by one mode line. The main problem is that a screen 256 bytes wide (six plus screens) uses a very great amount of screen memory. A depth of two screens in GRAPHICS 1 (ANTIC 6) would require 12K, and six screens would require 36K. This is an enormous amount of screen memory in addition to the game code, even for a 48K Atari.

Eight Way Scrolling - Special Case

A simple but contrived eight-way scrolling example could be developed that has a width of 128 bytes. The depth of course will be determined by the amount of screen memory available. The advantage of a 128-byte width over a 256-byte width is that it will allow a deeper scrolling playfield without requiring a complicated multiplication subroutine. Even so, the calculation is not as simple as the example above. First, it requires calculating the display address of the initial LMS instruction based on the rough scroll position YS. Each subsequent LMS address is determined by adding $80 to the previous one, then adding the horizontal rough scroll position XS to that.

ADDRESS = SCREEN + (YS*$80) + XS

We used a trick to avoid the initial multiplication usually needed to find the first LMS address. Since the high byte of the address increases every time our vertical rough scroll position (YS) increases by two, then:

SCHI = SCREEN/256 + YS/2

Odd values of YS set the carry bit after the division. If that occurs #$80 is added to the low byte of the first instruction. Afterwards the horizontal rough scroll offset XS is added.

SCLO = Screen address low byte + #$80 + XS (if carry set after division)

SCLO = Screen address low byte + XS (if carry clear after division)

Since each of the mode lines are scrolled equally horizontally, you only need to add #$80 in a double-byte addition to find the starting address of the next mode line. Again, this whole operation can be performed using indirect indexing. The first LMS low byte address is at NDLIST+7, and the high byte address is at NDLIST+8. We can index both of these addresses using the Y register, then increment the Y register by 3 in a loop to reach subsequent LMS address pairs. The routine exits the loop when all of the mode line addresses have been calculated or when the Y register equals (#ROWS * 3)-3.

Download 8WAYTEST.EXE (Executable program)
Download 8WAYTEST.OBJ (Object code)
Download / View 8WAYTEST.LST (Assembler listing)
Download / View 8WAYTEST.S (Source file)
Download / View 8WAYTEST.RAW (As printed in book)

The subroutine that updates the display list is driven by an eight-direction joystick routine. This routine calculates both the fine scrolling and rough scrolling values. The rough scrolling values XS and YS are inputted to the subroutine. The boundaries of the screen are indicated by the variables TOP, BOTTOM, LEFT, and RIGHT. Both TOP and LEFT are equal to zero and RIGHT is #$80. BOTTOM is user definable but must be equal or less than screen memory in bytes divided by 128. The flow chart is shown below.

Use of Lookup Tables to Determine LMS Screen Addresses

Another technique that would give you more flexibility in sizing your screen would be to use lookup tables to determine the LMS addresses for any given vertical scrolling offset YS. You can use each of the two tables, one containing the low byte address, the other the high byte address for each of the possible scrolled positions. By indexing into the start of the tables with YS, you can lookup each of your LMS addresses, then add the horizontal offset. The two tables are LMSHI and LMSLO. Both are formed by calculating all possible LMS addresses for the entire screen memory of your scrolling range.

```
        LDY #$00
        LDX YS          ;ROUGH VERTICAL SCROLL OFFSET
LOOP    LDA LMSHI       ;HIGH BYTE ADDRESS FROM TABLE
        STA NDLIST+5,Y  ;HIGH BYTE LMS
        LDA LMSLO       ;LOW BYTE ADDRESS FROM TABLE
        CLC
        ADC XS          ;ADD HORIZONTAL OFFSET
        STA NDLIST+4,Y  ;LOW BYTE LMS
        LDA NDLIST+5,Y
        ADC #$00        ;REST OF DOUBLE BYTE ADD
        STA NDLIST+5,Y
        INY             ;LMS INSTRUCTIONS-THREE BYTES APART
        INY
        INY
        CPX #$16        ;FINISHEDWITH # OF MODE LINES?
        BLT LOOP
```

General Case Eight-Way Scrolling

We have developed a general case eight-way scrolling example in which the programmer can define his own screen dimensions. Screens can be any width or height as long as the product of the height and width does not exceed the memory in the computer, in this case 64K. The following example is in GRAPHICS 0 (ANTIC 2), but there is no reason that you can't modify the display list to use a different graphics mode. Each mode line in the example is 1024 bytes in width and there are 64 rows of data. Thus, you are able to observe the entire memory of the Atari computer by scrolling with a joystick.

The routine is not exceptionally fast because it is the general case. It makes extensive use of sixteen-bit multiplication, thus wasting considerable time doing calculations. We don't recommend it for standard arcade games in which the screen is updated sixty times a second, but it is quite useful in tactical war games or other types of displays in which the screen changes less frequently. Both the flowchart and code follow.

Download 8WAY.EXE (Executable program)
Download 8WAY.OBJ (Object code)
Download / View 8WAY.LST (Assembler listing)
Download / View 8WAY.S (Source file)
Download / View 8WAY.RAW (As printed in book)

Strike Force--A Scrolling Game

Since horizontally scrolling shoot-'em-up arcade games like Super Cobra and Scrambler are immensely popular and not difficult to implement on the Atari, we included one called Strike Force as an example. The game involves flying an attack ship in a mission of destruction against an enemy attack fleet and their ground installations. The joystick-controlled ship is armed with bombs and lasers. It can fly at two speeds forward but can't reverse direction. Up and down joystick movements control altitude, while pushing the stick forward doubles the speed. The trigger fires the ship's lasers, except when the stick is pushed to the left. That drops bombs.

The continuously scrolling terrain stretches over thirteen screens. The last screen is a duplicate of the first to allow for wraparound. The tan-colored terrain consists of redefined Graphics 1 (ANTIC 6) characters using playfield register #0. The screen data for each of the twenty-two mode lines is 256 bytes or one page of memory. Screen memory requires 5 1/2 K.

Four active laser bases and seven missile bases populate the mountainous terrain. These redefined characters reference playfield register #1. They are red in color. While the missile bases don't launch their rockets, the laser bases produce deadly laser fire which should be avoided.

The player's ship, player #0, is double-width and light blue. Its single-width bombs use player #3. The aliens, one green and one red, use players #1 and #2 respectively. Since the ship's laser fire consists of quadruple-width missiles and the aliens use single-width projectiles, we can't combine the missiles to make an extra player. This limits the number of aliens on the screen at any one time to two.

SCROLLING GAME

While a two-prong alien attack can be handled quite skillfully by seasoned players who quickly learn patterns, the aliens in this game are chosen randomly from five different shapes and five different pre-programmed attack patterns. Thus, an alien shape doesn't necessarily correspond to a specific attack pattern. This subtlety makes learning the game difficult. Also, since games should increase in difficulty as they progress, alien firepower increases at 400 points, and again at 2000 points, until the game becomes nearly impossible for the average player.

Memory Layout

Once you have defined the basic design concept of your game, you need to decide where to put your program code, screen memory, display list, character set, and player-missile area. There aren't many constraints to where you put things in the Atari, except that the player-missile area must be on a 2K boundary, and the character set must be on a 1K boundary, and you should avoid the lower portion of memory, especially if DOS is used to load the program. The fact that Synassembler resides from $9C00 to $BFFF forces us to locate everything below that area. Therefore, we choose to locate our player-missile area at $8000, the character set above that at $9000 and our display list at the top at $9400. The two lines of scoring information begin at $6F00 and the 5 1/2K of screen area extends from $7000 to $8700.

We assembled our code at $3000, but we could have moved it higher in memory. It really doesn't make any difference in this case since our program code contains the data for the screen, character set, and display list. It wouldn't be any shorter if everything were pushed closer together in memory.

Display List

The display list is quite similar to the one developed earlier in the chapter for the rough horizontal scrolling example in BASIC. Separate LMS instructions are required for each of the twenty-two scrolling mode lines. In addition, there are two stationary ANTIC 6 mode lines used for scoring data at the top of the display. Since these lines use the ROM character set while the remainder of the display uses a custom character set, we need to do a Display List Interrupt on the second line in order for it to take effect in the third mode line. Each scrolling mode line is 256 bytes or one page apart so that the high-byte operands of each LMS instruction are one apart.

```
$70  8 blank scan lines
 70
 70
 46  |LMS ANTIC mode 6 -no scroll
 00  |Low byte score screen area
 6F  |High byte
 86  2nd score line -no scroll & DLI to take place next line
 56  |LMS ANTIC mode 6 horiz. scroll
 00  |Low byte of Ist or top row scrolling terrain
 72  |High byte
 56  LMS
 00  Low byte of 2nd row
 73  High byte
 .
 .
 56  |LMS
 00  |Low byte of 22nd row scrolling terrain
 86  |High byte
 41  |Jump and wait for VBlank
 00  |Low byte address of display list
 94  |High byte
```

The overall flowchart of the game is divided into Vertical Blank code and a main code loop outside Vertical Blank. The two sections are synchronized so that the main loop code will execute once, then wait in a tight loop for a flag to be set, signaling that the Vertical Blank code is finished. Although in retrospect it appears that the entire game could have been placed in Deferred VBlank, doing it this way assured that in the event the code became too long, the next Vertical Blank Interrupt wouldn't occur while the program was still in the previous one, thereby crashing.

The Vertical Blank code includes code that reads the joystick, controls all the ship's functions, including lasers and bomb drops, and interprets and executes the programmable code for the two aliens. The code in the main loop checks collisions in all combinations and takes appropriate action. This includes removing shot aliens and ground targets, derezing the player's ship when necessary, and updating the score. In addition, laser base fire, and the checks that control and increase the difficulty, are controlled here.

Unscramble Screen Data Routine

Normally, one would use some sort of homemade scrolling map editor to create the screen data necessary to form a 12-screen world. But these editors, if designed properly, automatically save the data to disk as a data file. In order to allow the reader to type in the required 5 1/2K of screen data without supplying an editor, the data was compacted to a mere 550 bytes. We simply took into account that the upper half of the screen was blank, and that large sections of the lower portion had sequences of similar characters.

Two tables were set up. One called VALUE contains the value of a particular sequence of blocks, and the other table called BLOCKS contains the number of number of those characters in a row. The small four-row sample illustrated below shows how the data for the two tables is obtained.

The unpacking routine that places the character data into screen memory is extremely sensitive to data errors in which the length of a single row isn't exactly 256 bytes, the length of one mode line. If a mistake occurs, data in the following rows isn't just offset, but it often doesn't even appear. The routine assumes that it will finish storing the number of repeat character bytes at exactly the same time that the Y register, which has meanwhile been keeping track of its position along the row, reaches the end (zero). At that point it jumps to the next row. Unfortunately, the Y register counter is being incremented in a loop while the number of characters in a row is being decremented. in the X register. For example, if the number of blocks in a row were one too many for the 256-block row, the Y register would have passed the zero point and the test for the end of the row would be missed. The row wouldn't be incremented. and the rest of the data would overwrite previous character data on the same mode line. While I thought of fixing it by testing whether the Y register became zero while in the loop, I'm not sure which case is worse, knowing that you made a mistake in the data for that row, or observing a completely weird display afterward.

Custom Character Set

The display uses a custom character set composed of twelve combinations of sloping terrain. Character #0 is blank. Since these are internal characters 0-11, they use color register #0 to set their color. We chose tan. The rockets, laser base, and its moving laser beam are also custom characters. Since we wanted to use a different color, we placed them in the upper half of the character set. Internal characters 64-127 refer to color register #1. The two high bits of the character number select the color register. The rockets are internal characters #63 and #64, the laser bases are characters #60 and #61, and the beam is character #62. We chose red primarily so that the laser beam showed up as red. Since the remainder of the character set was of no

interest, it wasn't copied from ROM and then modified. Instead, the character data was copied from tables directly into the specified positions in character set memory.

However, since we did need the letters for the GAME OVER message which scrolls across the playfield at the conclusion of the game, the data for these letters are stored as characters $65-$71, They too, appear in red. All of these tables occupy only 192 bytes of program memory.

Ship Operation

The joystick-controlled spaceship, while stationary on the horizontal axis during scrolling, can be moved vertically to adjust altitude. Pushing forward or to the left on the stick doubles the speed and consequently the rate of scrolling and horizontal speed of the attacking aliens. The latter is necessary to maintain the alien's position above the faster scrolling terrain. When the stick is pushed forward, the variable SPEED is set to one, otherwise it remains zero. There are also two different engine

sounds depending on the speed. Keeping the ship stationary in the horizontal direction was supposed to keep the programming simpler. Still, it wouldn't have altered the code much if the ship were free to move in that axis, as long as it was restrained from approaching too closely to the right edge of the screen. This is necessary to prevent the bombs from falling beyond the screen boundary. Introducing this motion might make an interesting exercise for those who understand the program.

Ship's Laser

Pressing the joystick trigger activates the ship's lasers. This works in any of the neutral or forward stick positions. The laser is a quadruple-width missile #0. The size can be set independently of the normal-size alien missiles by setting both of the lower two bits in SIZEM ($D00C = 3). This hardware register is divided into bit pairs for each of the four missiles. The missile shape is just one pixel high by two pixels wide.

Only one ship's missile or laser beam can be on the screen at any one time. To prevent it from refiring before it either strikes its target or exits screen right, a flag called TMIS0 is set. TMIS0 = 0 the first time through the routine so that it adjusts the missile's position to fire initially from the ship's nose and plots it there. It then resets the flag to 1 and turns on the laser fire sound timer. During subsequent cycles through the routine, the TMIS0 flag causes it to branch to the code that moves the

beam 2 units/cycle to the right. The position is tested against the screen boundary on the right side. If it hits it, the missile is removed and placed offscreen to the far left and TMIS0 is set to zero. Collisions, of course, cause the same effect, but they are tested elsewhere. To prevent the laser from being fired while the ship is offscreen after a derez, a DELAY flag is tested. The missile can't be fired if the flag is set, but a missile fired previously will continue along its track.

Bomb Drop

In a realistic bomb drop, the bomb arcs slowly at first and then accelerates rapidly downward until it falls almost entirely vertically. You can implement such a drop on the Atari with only a minimal knowledge of physics. Gravity or acceleration only acts on the bomb in the Y direction. While the velocity in the Y direction increases with time, the velocity in the X direction remains constant, if we neglect air resistance. An object that has a velocity in a particular direction will move in that direction. Its new position is equal to its old position plus its change in position during that period (velocity). We can summarize this as follows:

VY = VY + GRAVITY
YB = YB + VY
VX = CONSTANT
XB = XB + VX

Choosing a realistic acceleration value is largely experimental. Even an acceleration of + 1/frame would cause the vertical velocity to grow enormous over as little as 15 animation frames (1/4 second). If the bomb is to arc slowly, VY will have to be somewhat smaller than VX = 2 during the first few frames, yet not grow too much larger later, or the bomb will drop like a lead weight. In order for VY to increase slowly every fourth frame, the acceleration must be less than unity. A clever approach is to increment a variable called VTEMP and divide by 4 each time. It will take four cycles to increase VY by one unit/frame. This keeps the bomb from accelerating too fast, but it still will fall too rapidly from great heights. The solution is to clip VY so that it doesn't become too high. While the values are largely experimental, a maximum value of VY = 3 produces realistic-looking bomb trajectories.

The bomb subroutine uses a flag called BOMBON to determine if it should plot the bomb initially directly beneath the plane, or accelerate and move it as it falls. The bomb must begin its descent from the center of our plane because bomb bays are located at the plane's center of gravity. Therefore, the bomb needs repositioning from the ship's coordinates XPMO, YPMO at the tip of the tail. YB = YPMO+10 and XB = XPMO+5. The bomb is plotted there, and the BOMBON flag is set to 1.

When the bomb subroutine is entered on subsequent frames, it branches to the bomb falling code where the bomb's velocity and position for each direction is calculated. It is then plotted in that position. Of course, the bomb is removed during the collision test if it strikes either the ground or any of the laser base or missile targets, but that occurs elsewhere in the main line code. The BOMBON flag is reset to zero there, so that another bomb can be dropped if the trigger is pressed while the stick is pushed back or to the left.

Ship's Explosion and Delay

The player's ship explodes upon collision with the terrain, an alien, or enemy fire from either the laser bases or alien craft. Although there are many methods to explode a ship, we choose to use the deresolution method employed earlier in the Space War game in Chapter 5. The appearance is of a ship that is slowly disintegrating. As you recall, it uses a random number generator to degrade a duplicate of the ship's shape. It is a fairly complicated routine that uses a series of AND and ORA instructions to control the image's rate of degradation so that the random flickering of the individual pixels lasts at least forty-eight cycles. We aren't going to explain the routine again, so we suggest you look at it in the last section of Chapter 5.

Since their should be a several second rest between the ship's deresolution and its reappearance to battle once again, many of the subroutines that operate on a per cycle basis should be shut off during this period. These include the start of another enemy attack once they clear the screen, and the ability to fire lasers and drop bombs once your ship is killed. While it might be easier to set one delay flag during the explosion and branch past all of the code, you need to be selective in what is shut down, otherwise laser beams and falling bombs might vanish in mid-flight and the screen would suddenly stop scrolling with aliens frozen in position. The game should go on. After all, it is only your ship that has been destroyed; everyone else is still alive.

We need to set several flags in the XSHIP subroutine that is called whenever the ship collides with anything. Setting REZFLAG = 1 turns on the actual explosion subroutine. Setting DELAY = 1 starts the four second delay. In addition, this lengthens the timer delays, NDELAY1 and NDELAY2, that control when the next set of aliens reappear. If we are out of ships, the words "Game Over" are written into screen memory just beyond the right edge of the screen.

There is a section of code at the beginning of the Vertical Blank Interrupt routine that actually monitors all of these flags so that the ship is derezed first and then removed from the screen until the four second delay ends. It keeps track of the timer at location $13. This location is incremented roughly every four seconds. When it is

non-zero, the routine checks if any ships are left, and if so, puts the ship back and resets the delay timer to zero. Obviously, if we are out of ships, the game is over, and it is time to jump to the very beginning of the game code.

Long before the timer at $13 ever becomes non-zero, the routine determines where it is in the derez cycle via a counter called EXCOUNT. The explosion subroutine increments EXCOUNT after each cycle. When EXCOUNT reaches 48 cycles, the ship is virtually disintegrated. This is done one cycle earlier than the cycle that resets REZFLAG = 0. If at least one of the ship's pixels is in contact with the playfield, a collision value will be returned even though we just ordered the ship offscreen. The computer updates it's collision registers before we have time to move our ship. If we don't delay setting REZFLAG = 0 by one cycle, our collision test may detect a bogus collision and the player loses another ship. The ship is then moved off the screen. This is done one cycle earlier than the cycle that resets REZFLAG = 0 because if at least one of the ship's pixels is in contact with the playfield, a collision value will be returned even though we just ordered the ship off screen. The computer updates it's collision registers before we have time to move our ship. If we don't delay setting REZFLAG = 0 by one cycle, our collision test may detect a bogus collision and the player loses another ship.

Scrolling

The screen scrolls leftward at a constant rate of one color clock per second except when the control stick is pushed to the right. It then scrolls at two color clocks per second so that it appears that the ship is flying at twice the speed. To scroll the screen smoothly you need to alternate between adjusting the fine scroll register over eight color clocks, and rough scrolling the screen by adjusting the LMS operands of all twenty-two ANTIC 6 mode lines.

In our case, we increment a variable called FS, short for fine scroll, each cycle. This variable goes from 0-7, and is backwards from the horizontal fine scrolling register HSCROL at $D404. HSCROL is initially set at 15 when FS equals 0 and counts down as FS increases. Thus, HSCROL = 15-FS. The fine scroll register is reset every eight

clock cycles and the rough scroll variable XS is incremented. When the screen has been rough scrolled 235 times, it is time to stop and reset the rough scroll register XS back to zero. If we try to go further, for example to XS = 236, ANTIC will fetch the first nineteen bytes of that scan line correctly, but the twentieth will be data for the following scan line. This obviously produces a faulty display that becomes worse the further we move into the wraparound zone. This end zone should be an exact duplicate of the first screen so that when we jump from XS=235 to XS=O, the screen will remain unchanged.

Programmable Aliens

It usually requires an extensive amount of program logic to achieve a variety of enemy patterns within a game. Some games, like this one, use preset patterns that are independent of the player's action, while others react to evasive techniques used by the player. Obviously, the computer's reaction to the player's actions produces more challenging games, but if the programmer isn't careful he may design a game in which it is impossible to survive.

For example, in this game aliens follow some sort of zig-zag pattern, change their speed, and shoot in predetermined directions as they close on your position. It would be quite easy to program them to home in on your position with their guns or just on your vertical position for eventual collision. There would be only two possible results: you would either be able to shoot them easily if they matched your vertical position quickly for they would always be directly in your line of fire; or they would home in on you at the last split second and you would have no chance of survival. Likewise, you could never evade their guns, if they were programmed to shoot with uncanny accuracy. Perhaps a better method might be to make random variations on a predetermined path.

We chose a slightly different tack to solve the problem of easy programmability, while still offering enough variations to make learning the alien attack patterns difficult. We actually developed five different patterns for each alien player, and to make it easy to change wrote a scheduling routine that bases all enemy actions on an internal timer and a set of tables.

The first three bytes in the table contain the X1, Y1 starting position of the alien, followed by the value NDELAY1, which is the delay in cycles before the next alien appears after the current one is killed or exits the screen. Each group of six bytes that follows is an instruction containing the high and-low-byte time to read the next instruction (TIME1L,TIME1H), the alien velocity (VX1,VY1), whether to shoot or not (SHOOT1), and the direction of the shot (DIR1). We are limited to eight different instructions for each programmable alien because five different patterns are stored in one 256-byte page of memory. Thus, each complete program occupies 6*8 + 3 = 51 bytes of memory. We have set aside two blocks of memory, one for each player.

If you look at the data in our table ENEMY1 for the 0th shape, it is as follows:

```
X1        Y1        NDELAY1
$D0       $50       $30

TIME1L     TIME1H         VX1       VY1       SHOOT        DIR
$28        $00            $FF       $00       $00          $00
$50        $00            $FF       $01       $00          $00
$70        $00            $FF       $00       $01          $07
$FF        $00            $FF       $00       $00          $00
```

If you match it against the diagram below, you will see that the alien enters the screen at X=$D0, Y=$50. These are player-missile screen coordinates and are completely independent of the scrolling playfield. The value NDELAY1 = $30 means that there will be a delay of forty-eight screen cycles between one programmable alien using player #2 leaving the screen, and the next one entering. The alien begins moving leftward in step with the scrolling terrain below until the internal timer (TIMER1) reaches $28. It then reads in the next six-byte instruction. This instruction tells it to begin moving diagonally downward. VX1 =$FF and VY1 =$01. It also obtains a new timer value of $50 so that the next instruction isn't read until its internal timer reaches $50. At that time VY1 = 0 and the alien continues its path along the horizontal axis. Its gun is turned on, and it shoots in a direction of 7, up and to the left. It reads its last instruction at TIME1=$70 and shuts off its guns. It will continue moving horizontally until it exits screen left.

There is a delay timer, TDELAY1, that prevents a new alien from appearing immediately after the first. The value of NDELAY1 that was looked up in the tables by the prior alien is transferred to TDELAY1 after that alien exits the screen. If TDELAY1 is positive when it enters the routine that reads the programmable alien code, it branches past it and just decrements this timer once each cycle. There is a secondary flag called ONSCRN1 that is set to one during alien initialization after the

delay has ended. Once this flag is set, the program compares its timer, TIMER1, to that of TIME1 which it obtained from the first set of instructions. If it is equal, the program looks up the next set of instructions. It then moves the alien, jumps to a separate subroutine to fire the missile, checks if the alien is still on the screen, and finally plots the alien.

During initialization, one of five sets of pointers to the programmable data is chosen randomly. Then, one of five alien shapes is randomly picked. This prevents the player from predetermining the pattern from the alien's shape. The ONSCRN flag is then set and the initial values for X1,Y1, NTDELAY1, VX1, VY1, TIMEL1, TIMEH1, SHOOT1, and DIR1 are obtained.

The obvious advantage of developing this routine was that we were able to create an attack pattern, test it, then modify it until it became sufficiently challenging. Of course, when we did this, we hadn't put in the CHANCE subroutine. Therefore, it was easier to control which programmable alien code we were working with. Since the X-register determines which program is loaded, it is fairly easy to skip the CHANCE subroutine and load the X-register with a value from 0 to 4.

The game turned out to be much more difficult than anticipated because the level of alien firepower was too intense. It was relatively easy to cut the firepower at the beginning of the game, and then raise it as the player's score increased. The game shifts to intermediate level at 400 points, then to expert level at 2000 points. Most of the shoot flags in the programmable tables are set to zero at the beginning of the game and then restored as the game progresses. Consequently, it was quite easy to develop a game that increases in difficulty with the player's skill.

Ground Lasers

The four ground lasers use animated character graphics to move their laser beams. The segment of the beam uses internal character #62. Each base uses two flags to keep track of the status of the base and two timers to slow the beam down and delay it between firings.

The main loop code performs a simple test to determine if it should call the subroutine LASER. A timer called LDELAY is set to #$50 after each shot to insure that it rests slightly more than one second before refiring. The timer decrements once each cycle, so eventually it will reach zero and call the LASER subroutine.

Since it is possible to destroy one or more bases, the GALIVE flag keeps track of active bases. Bases are alive when GALIVE = 1. The LASON flag is used to prevent the routine from reinitializing the beam after it begins its track from just above and to the left of the laser base. When LASON = 1, it skips the initialization and proceeds

with the movement each time LCOUNT reaches zero until the beam, which is plotted at GROUNDL, GROUNDH, reaches the top mode line at GROUNDH = #$70.

The beam, which is traveling at a forty-five degree angle, is first erased at its old position GROUNDL,GROUNDH by writing a blank character #0. The new position is one character to the left and one mode line (256 bytes) lower in memory. Therefore, the new position is at GROUNDL-1,GROUNDH-1, and we plot internal character #62 in this position of screen memory. The beam would move much too rapidly if it were moved every cycle. We can slow it down and move it every third cycle by our usual countdown timer method. LCOUNT is reset to 3 each time the beam is moved and then decremented with each cycle. The beam is moved only when LCOUNT reacbes zero, thus slowing the beam down.

Collision Tests

There is a long series of collision tests in the main code loop that cover every possible interaction between any two screen objects. The tests can be performed outside the VBlank because the main code only executes once every VBlank cycle and will do the test after the collision registers have been set from the previous frame.

The first few tests involve collisions of our ship with the playfleld, enemy aliens, and their missiles. Ground-based laser fire is considered playfield in the tests. In all cases, collisions result in the elimination of our ship via the XSHIP subroutine. We only score points in the case where our ship collides with an enemy ship. A kill of any nature is worth points even if it costs you dearly. The explosion sound timers are also set here.

The second series of tests involve collisions of the two aliens with the playfield and our ship's laser. All cases result in the elimination of the alien ship. Points are only scored if the alien craft are killed by our ship's laser fire.

The final series of tests involve collisions of bombs and our ship's laser fire with the playfield. Collisions with the ground (playfield #0) are simple and only result in the removal of the weapon. Collisions with ground targets

(playfield #1), however, are more complicated because we need to calculate which target is to be removed in the subroutine RTARGET.

Fortunately, all of the targets are widely spaced so that we basically only need to compare the position of our laser or bomb with that of the playfield beneath. To convert the player position of the missile or bomb (POS) to that of the playfield requires us to first determine how many character positions are between it and the playfield's left screen edge, and add the rough scrolling offset into the playfield map, XS. The formula is:

BTARGET = XS + INT(POS/8)

Using this formula produces inaccuracies because we don't take into consideration the possibility that either the fine scrolling register or the left edge of the bomb may clip the far edge of the target in its arc, and thus throw off our calculation. Obviously, if we are testing for an exact horizontal match between the weapon's position and all of the available ground targets, we are going to miss occasionally. If we expand the test to include the BTARGET-1 and BTARGET+1, a match would always be assured.

Our subroutine decrements BTARGET and first compares it to horizontal positions of each of the four laser bases. It removes it if it finds a match and sets a flag STRIKE = 1. It continues to do the same test with the seven missile bases. If STRIKE doesn't equal one when it is finished, it increments BTARGET and tries the tests again. It will eventually find its target on one of the three passes, remove it, trip the explosion sound time, score some points, and exit the subroutine.

Scoring

The scoring subroutine differs little from those of many of the others used in this book. It and others keep track of the different decimal digits separately so that they don't have to convert an internal hexadecimal score to decimal. There are separate counters for the tens digit, hundreds digit, thousands digit, etc. Carrys are performed into the next higher digit whenever one or more of these digits exceeds a value of nine. The actual positions of each of these counters in the score line is shown below.

The scoring values for each target in the game are worth differing point values. Missile bases are worth 10 points, alien ships 20 points, and laser bases 30 points. Fairly high scores can be achieved by players since there are an unlimited number of aliens, and missile bases are replenished after all seven have been destroyed.

There are three separate entrance points to the subroutine. Notice that the first three blocks of each in the flowchart are identical. Each of these small sections increments the tens digit SR10 by one, and takes care of a carry to the hundreds digit if necessary. When 20 points are awarded for a target it enters at SCORE2, a lower point in the subroutine, and increments this digit twice. By entering at SCORE3, 30 points are scored since it passes through all three sections of the same code.

Retaining the player's high score turned out to be especially important in Strike Force. Many players had difficulty with the game, scored low, and refused to play it after two or three tries. When the high score feature was added, they would play for much longer periods to beat their last high score. Since the game resets automatically through the System Reset Vector, it was necessary to place the initialization for the high score digits or variables prior to that address in the game. The score variables for the last game aren't reset until after the START key is pressed. This makes it possible to see both the high score and the score for the previous game while in the "attack mode." Space was tight in the two-line display so that the two scores alternate in the display's slow-blinking cycle.

The high score is updated in the subroutine XSHIP when the game is over. Each of the current high score digits, starting with the highest, is subtracted from each of the score digits. If any become negative in the process, it means we haven't reached a new high score and the routine aborts. It updates only when it detects a positive result. Zero results are ignored. Once a positive or negative result is obtained, the rest of the lower digits are ignored for they would only confuse the routine. For example, if SCORE were 00630, and HISCORE were 00580, it would get a positive number in the hundreds digit. This means we have a new high score. If we were to continue the test we would get a negative number in the tens digit. This would mean we don't have a new high score. Since only the first positive or negative result is meaningful, it is best to ignore the remaining digits.

Sound

The game uses all four sound channels. Channel one is used for the laser fire sound, channel two for the short explosion sounds of the aliens and ground targets, channel three for the engine sound, and channel four for the sound of the ship's longer explosion. Four separate channels are required so that one sound doesn't interfere with another in the event that several occur simultaneously.

The sound subroutine resides and operates in the VBlank routine, They are essentially always on but require setting a positive value in a flag to trip them. For example, the laser sound can be tripped by setting SLTIME = 1. Each of the routines, which are described more fully in the sound section in chapter eight, use internal countdown timers to gradually lower the tone and volume. The engine sound, on the other hand, just produces a continuous sound using some distortion. The frequency is changed to a higher pitch when the ship speeds up.

Download SCROLL.EXE (Executable program)
Download / View SCROLL.LST (Assembler listing)
Download / View SCROLL.S (Source file)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 8
Raster Graphics & Sound

Raster graphics is a term we very rarely use in connection with the Atari computer system. It is a term that describes how individual pixels are mapped on a highresolution screen. The technique is about the only one possible on computers such as the Apple 11 and the IBM PC. Atari programmers like to use easier and more colorful techniques like character graphics and playermissile animation, but there are certainly a number of valid reasons for animating with raster graphics. The two best reasons are that Graphics mode 8 (ANTIC mode F) screens have the highest resolution (320 x 192 pixels), and that very large shapes can be smoothly animated. The biggest disadvantage is that you can have shapes with three colors at best.

Graphics 8 screens produce color by a method known as artifacting. On computers with a GTIA chip, pixels in even columns appear blue and those in odd columns appear green when the background color register is set to black. Obviously, we could obtain other color combinations by varying the background color register. If you wish to draw a shape entirely in blue, you need only plot the shape's individual pixels in the even columns. Similarly, you will obtain an all-green shape if you plot pixels only in the odd columns. When a blue pixel is next to a green pixel, the pair appears as white.

You get these alternating stripes of color because the Atari sends its color signal as a series of square wave pulses. One complete cycle is called a color clock. you get blue, and when it is low you get green. Other colors are produced by phase shifting the square waves. These colors have nothing to do with the positio television tube.

Pixel information is encoded eight pixels per byte. The screen is made up of 192 rows of forty bytes. Forty bytes times eight pixels per byte gives a horizontal r working in color you are really only talking about half that, or 160 pixel pairs horizontally.

Plotting pixel data is quite anolagous to plotting character data to the screen. In fact, if we took the character data for the letter "A" and plotted it on the screen l a particular column in the first eight rows, the character would appear as expected. Of course, it is a lot of trouble to just plot character data on a Graphics 8 sc technique.

A Graphics 8 screen fortunately can be mapped sequentially in memory if you are clever. The problem is that a display list cannot address screen memory that additional LMS instruction is needed on the other side of the boundary. You could fit 102 lines in the lower portion, but that would leave a 16-byte gap between decided to place 102 lines in the following example in the top 4K of screen memory and butt the first ninety lines against it in the lower 4K section. This leaves beginning and ample room to place the display list. The display list begins at $6000, and screen memory at $61F0. The 0th or top line starts at that address an later at $6218. Each of the 192 lines is offset in memory by an additional 40 bytes.

To plot a byte at a particular X,Y coordinate on the screen requires you to calculate a memory address based on the starting address of the row and the offset i
MEMORY ADDRESS = SCREEN MEMORY +(Y*40)+(X/8)

It isn't a difficult calculation, except that in Machine language, multiplication and division other than by multiples of eight require an enormous number of steps. would be alright. Unfortunately, you need to perform the calculation at least once for each row of the shape. If you were trying to do a Galaxian-type game whe you would never have enough time to move and draw them all and achieve a fast enough animation frame rate. A better method is to look up the starting addre just calculate the horizontal offset based on the shape's Y coordinate.

Plotting a pixel on the screen at a particular X,Y coordinate, based on its row and horizontal offset, will work only if the Y coordinate was a multiple of eight. The left end of the byte and would have a value of $FO. If you want to plot a pixel one unit further to the right, you need a byte with an entirely different pixel pattern byte just anywhere, like a "tad" to the right. The value of that byte is $80. Now, you begin to realize that you need eight different bytes just to cover all of the po also true of larger shapes, we have a difficult problem.

Color shapes have another problem. If you move them right or left one pixel, they shift colors. Therefore, you must move them two pixels left or right at a tim complicated, it actually reduces the number of shifted shape tables to four. It has one additional advantage in that there are only 160 possible horizontal positi the arithmetic to single-byte operations.

Bit Mapping the Shapes

Drawing a bit-mapped shape table anywhere on the Graphics 8 screen is a simple procedure, once you understand the basic concept. The shape table is stored by rows or columns. The technique, therefore, is to load each of these bytes, one at a time, into the Accumulator, find the position in memory for the screen lo that byte, then store it in that location.

Memory Location by Table Lookup

The difficulty, as we showed earlier, lies in finding a particular memory location, given an X,Y screen coordinate. Table look-up is obviously the fastest meth for the first position (leftmost) or 0th offset for each of the 192 lines. If the screen started at $61F0, the first line or line #0 would begin at this address, and the $6218. Each address takes two bytes. The first part is the high byte which in the latter case is $62. The second part, $18, is the low byte. These values can be s containing the lower order address of each line (call it YVERTL), and the other containing the higher order address of each line (call it YVERTH). Each table order that these tables not become specific to a particular screen address, the values are merely offset values from a zero starting address. The GETADR subro starting screen address to obtain a specific memory address. Our only constraint is that the screen start on a page boundary.
You can access any element in either table by absolute indexed addressing. The effective address of the operand is computed by adding the contents of the Y instruction. The format is:

EFFECTIVE ADDRESS = ABSOLUTE ADDRESS + Y REGISTER

If our YVERTL table were stored at $4000 and we wanted to find the starting address of line I (remember lines are numbered 0-191), we would index into the ta value into the Accumulator.

4000:F0 18 40 68 90 B8 ..... YVERTL TABLE

So LDA YVERTL,Y, where the Y register =$01, will fetch the value $18 from memory location $4000 + $01 = $4001, and place it in the Accumulator.

Similarly, if YVERTH were stored in the next page following our first table, then:

4100:01 02 02 02 02 02 ..... YVERTH TABLE

If the Y register = $01, then a LDA YVERTL,Y will take the value $02 stored in memory location $4100 + $01 = $4101, and place it in the Accumulator.

Storing the Shape in Screen Memory

Eventually we will want to store the first byte from the shape table into a memory location. This can be done efficiently if the two-byte address is stored seque the low-byte half of the address, HIRESL, at location $F2, and the high-byte half, HIRESH, at location $F3 in zero page:

```
LDY #$01        ;Y REGISTER CONTAINS LINE
LDA YVERTH,Y    ;LOOKUP HIGH BYTE OF START
                ;OF ROW IN MEMORY
STA HIRESH      ;STORE IN ZERO PAGE OF MEMORY
LDA YVERTL,Y    ;LOOKUP LOW BYE OF ROW IN MEMORY
STA HIRESL      ;STORE IN ZERO PAGE OF MEMORY
```

If the computer finds a $00 in location $F2 (HIRESL) and a $60 in location $F3 (HIRESH), then the base address is $6000. The Accumulator stores a value int
or lacation #6001, as shown on the following page.

The final addressing mode that we must consider is Indexed Indirect Addressing. The format is:

LDA (SHPL,X)

It is very similar to the Indirect Indexed addressing mode except the index is added to the zero page base address before it retrieves the effective address. Its p
effective addresses stored in zero page, but in the form we are going to use it, the X register is set to 0. Thus, it simply finds the base address.

We must use this second form of indirect addressing because there is a shortage of registers in the 6502 microprocessor. We are already using the Y register in
isn't an indirect indexed addressing mode of the form LDA(SHPL),X. Thus, we must go to the alternative addressing mode LDA(SHPL,X).
What this all boils down to is that we want to load a byte from a shape table into the Accumulator and store it on the screen with the following instructions:

```
LDA (SHPL,X) ;LOAD BYTE FROM SHAPE TABLE
STA (HIRESL),Y ;STORE BYTE ON HI-RES SCREEN
```

We can index into the shape table by incrementing the low byte SHPL by one each time, then store that byte into the next screen position on a particular line by
zero page method is faster than performing the equivalent code with absolute index addressing, because twobyte addresses can be handled with fewer instruc
less memory space.

Obviously, a generalized subroutine must be developed to find the screen memory address (HIRESL and HIRESH), given a line number and a horizontal displa
subroutine GETADR, short for Get Address.

Each time a row of shape-table bytes is transferred to succesive memory locations in screen memory, the program will call the subroutine GETADR. The line
offset by the horizontal location of the shape on the screen. Our table of line addresses is only an offset, so it will need to add the actual starting address of the
Memory address = Line # starting address + horizontal offset

```
GETADR  LDA YVERTL,Y    ;LOOKUP LOW BYTE ADDRESS OF LINE
        CLC
        ADC HORIZ       ;ADD HORIZ. OFFSET
        STA HIRESL      ;STORE LOW BYTE OF SCREEN ADDRESS
        LDA YVERTH,Y    ;LOOKUP HIGH BYTE ADDRESS OF LINE
        ADC /SCREEN     ;ADD HIGH BYTE OF SCREEN ADDRESS
```

```
        STA HIRESH      ;STORE HIGH BYTE SCREEN ADDRESS
        RTS
```

where the Y register has a vertical screen value (0-191).

If you are designing an arcade game, you will probably have several different shapes on the screen at one time. Keeping track of each shape's variables, whicl drawing routine, is generally easier if a set-up routine is incorporated into your program. This assures that you haven't forgotten to initialize anything before ent few variables need to be defined in the set-up routine: the location of the shape table; the horizontal displacement on the screen; and the width and depth of th

The drawing routine becomes more efficient the fewer times it accesses the GETADR subroutine. Therefore, it is much faster to load and store on the same sc shape's width is reached. Drawing our balloon a byte at a time across its width will only require calling GETADR 31 times. But if we plotted down instead, GETA byte, or 279 times, an unnecessary waste of time.

As we load and store across a particular screen line, we decrement SLNGH, the ship's width, until SLNGH equals zero. When we are finished with a row, we ir screen line down and decrement the DEPTH. When DEPTH reaches zero, we have plotted all rows of the shape and we are finished.

```
DRAW    LDY VERT        ;VERTICAL POSITION
        JSR GETADR      ;FIND BEGINNING OF SCREEN ADDRESS ROW
        LDX #$00
        LDA TEMP
        STA SLNGH       ;RESTORE VALUE OF WIDTH FOR NEXT ROW
        LDY #$00
DRAW2   LDA (SHPL,X)    ;GET BYTE OF SHAPE TABLE
        STA (HIRESL),Y  ;PLOT ON SCREEN
        INC SHPL
;NEXT BYTE OF SHAPE TABLE
     BNE .1

;IF CROSS PAGE BOUNDARY?
     INC SHPH

;INCREMENT TO NEXT PAGE OF SHAPE
.1   INY

;NEXT POSITION ON SCREEN
     DEC SLNGH   ;DECREMENT WIDTH
     BNE DRAW2   ;FINISHED WITH ROW YET?
     INC VERT

;IF SO, INCREMENT TO NEXT LINE
     DEC DEPTH   ;DECREMENT DEPTH
     BNE DRAW

;FINISHED ALL ROWS?
     RTS

;YES, END
```

Although the first row of the shape can be plotted at any VERT (0-191) position, if VERT began at 190, the computer would attempt to plot the third line at V that far would most likely produce garbage, as you would index beyond the end of the table. You should always be careful that:
TVERT <= 192-DEPTH

A simple test somewhere before the draw subroutine would suffice, but it might be incorporated into your joystick read routine.

XDrawing Shapes

Objects that move on the screen are shifted in position by erasing the object's first position before drawing it at its new position. The simplest method is to dra it before shifting it.

EOR Instruction

The Exclusive-OR instruction, EOR, is primarily used to determine which bits differ between two operands, but it can also be used to complement selected A works is elementary. If neither of two particular memory bits is set or their values are zero, the result is zero. If either one is set, then the result is one. But if b result is zero.

```
       MEMORY BIT    ACCUMULATOR     RESULT BIT IN
                         BIT          ACCUMULATOR
           0             0               0
EOR        0             1               1
           1             0               1
           1             1               0
```

If we take a byte on the screen and EOR it with the same byte

```
0 1 1 0 0 1 1 0     SHAPE ON SCREEN
0 1 1 0 0 1 1 0          SHAPE
_______________
0 0 0 0 0 0 0 0         RESULT
```

From the shape table, the result is zero or a screen erase. A similar effect would occur if a blank screen were EORed with a shape, then EORed again.

```
    0 0 0 0 0 0 0 0    BLANK SCREEN
EOR 0 1 1 0 0 1 1 0    WITH SHAPE
    _______________
    0 1 1 0 0 1 1 0    RESULT IS SHAPE ON SCREEN
EOR 0 1 1 0 0 1 1 0
    _______________
    0 0 0 0 0 0 0 0    RESULT IS BLANK SCREEN
```

It doesn't damage the background if a shape is EORed on the screen, and then off again. However it does distort the shape slightly.

```
    0 0 0 0 0 0 0 1    BACKGROUND
EOR 0 0 1 0 1 1 0 0    WITH SHAPE
```

```
    ---------------
    0 0 1 0 1 1 0 1    RESULT ON SCREEN (SHAPE
                       DISTORTED LAST BIT)
EOR 0 0 1 0 1 1 0 0    WITH SHAPE

    ---------------
    0 0 0 0 0 0 0 1    GET BACKGROUND BACK
```

In the above example, an extra pixel in the shape's last bit position distorts the shape drawn on the screen. In the example below, the fourth bit position becom

```
    0 0 0 1 0 0 0 0    BACKGROUND
EOR 0 1 0 1 1 0 0 0    WITH SHAPE

    ---------------
    0 1 0 0 1 0 0 0    RESULT ON SCREEN
          ^---------hole here
EOR 0 1 0 1 1 0 0 0    WITH SHAPE

    ---------------
    0 0 0 1 0 0 0 0
```

There are some techniques to avoid distorting the shape when the background is likely to interfere during the drawing process. This involves a combination o with the background stored in an alternate screen memory. An alternate method is to store the screen memory bytes in a temporary table equal in size to your shape. When erasing, you replace the shape with the background stored in your temporary table.

OR Instruction

The OR memory with Accumulator (ORA) instruction differs from the EOR instruction in that if both memory and Accumulator bits are on, then the result is

```
       MEMORY BIT    ACCUMULATOR     RESULT BIT IN
                         BIT           ACCUMULATOR
           0              0                0
ORA        0              1                1
           1              0                1
           1              1                1
```

If the background were as follows, and you ORed it with the shape, the shape remains correct.

```
    0 0 1 0 1 0 1 0    BACKGROUND
ORA 0 1 1 1 1 0 0 0    WITH SHAPE

    ---------------
    0 1 1 1 1 0 1 0    GET SHAPE + BACKGROUND WITH NO HOLE IN SHAPE
```

Unfortunately, if you EOR this result with the shape again, the background is flawed.

```
    0 1 1 1 1 0 1 0    SHAPE + BACKGROUND
EOR 0 1 1 1 1 0 0 0    WITH SHAPE

    ---------------
    0 0 0 0 0 0 1 0    FLAWED BACKGROUND
```

We can incorporate the Exclusive-OR instruction in our XDRAW routine. If we EOR the shape we had previously on the screen, nothing remains.

```
00010 XDRAW     LDY TVERT       ;VERTICAL POSITION
00020   JSR GETADR
00030   LDA TEMP
00040   STA SLNGH               ;RESTORE VALUE OF WIDTH FOR NEXT ROW
00050   LDX #$00
00060 XDRAW2    LDA (SHPL,X)    ;GET BYTE FROM SHAPE TABLE
00070   EOR (HIRESL),Y  ;EOR WITH BYTE ALREADY ON SCREEN
00080   STA (HIRESL),Y  ;DRAW ON SCREEN
00090   INC SHPL        ;NEXT BYTE OF SHAPE TABLE
00100   INY
00110   DEC SLNGH       ;DECREMENT WIDTH
00120   BNE DRAW2       ;FINISHED WITH ROW?
00130   INC TVERT       ;IF SO, INCREMENT TO NEXT LINE
00140   DEC DEPTH       ;DECREMENT DEPTH
00150   BNE DRAW        ;FINISHED ALL ROWS?
00160   RTS             ;YES, END ROUTINE
```

Now that we know how to DRAW and XDRAW a bit-mapped shape anywhere on a Graphics mode 8 screen, the principle for animating them is simple. A sh new position is calculated, then it is redrawn at its new position. The procedure is outlined below.

A delay has been inserted between the DRAW and the XDRAW to allow the object to be on the screen longer than it is off. Without the delay, the object is er drawn. This does not give the shape's image sufficient time to remain onscreen during one animation frame. The result is a badly-flickering image. A small de the internal clock at $14 for so many jiffies. Experiments show that 3/60 seconds is a good value.

Whenever a shape is moved horizontally, the bit pattern within a screen memory byte shifts and sometimes even intrudes into the adjacent byte. To avoid color column artifacting, shapes must be moved horizontally two columns at a time. If we consider the four-pixel-wide shape in the diagram below and move it right t same shape and color pattern, but the value in its shape table changes. If we shift the shape far enough, some of the bits run into the next byte. By the time we the pattern repeats itself as if it were in the same starting position but one byte over. So if we are going to be able to move a shape anywhere on the screen, w tables each one byte wider than the original shape. And since we need to move the shape horizontally two pixels or color clocks at a time, it would be easier to that goes from 0-159 instead of 0-319.

It would be nice if there were a relationship between the horizontal position (X) and the shape #. The mathematical relationship is as follows:
TEMP = INT (X/4)
SHAPE# = X-TEMP*4

Actually, it is a lot faster to look the value up in a table called XOFF. This table has the shape table # for each possible X position. You can retrieve the shape ta into the table with the X position in the Y-register. We use another small table SHPLO to store the low byte starting positions of each of the shape tables, and a combined length of the four shapes crosses a page boundary. The code to set up the pointers to the proper shape is as follows:

```
LDY X           ;HORIZONTAL POSITION (0-159)
LDX XOFF,Y      ;INDEX TO FIND SHAPE #
LDA SHPLO,X     ;INDEX TO GET LOW BYTE OF SHAPE TABLE
STA SHPL        ;STORE LOW BYTE IN ZERO PAGE
LDA SHPHI,X     ;GET HIGH BYTE OF SHAPE TABLE
STA SHPH        ;STORE HIGH BYTE IN ZERO PAGE
```

The drawing routine is exactly the one described earlier. Once the pointers to the proper shape table are inputted with both the shape's vertical position and ho transferred to screen memory from the appropriate shape table.

The XDRAW subroutine differs from our drawing routine in only one instruction. Instead of just fetching a byte from our shape table and placing it directly in scr with the byte already on the screen before storing it there. The bits are effectively erased if the screen image byte and the shape table byte are a match.

```
LDA (SHPL,X)   ;GET BYTE FROM SHAPE TABLE
EOR (HIRESL),Y ;EOR WITH SCREEN IMAGE
STA (HIRESL),Y ;PLOT ON SCREEN
```

Collision Detection

Detecting collisions between raster shapes isn't easy. There aren't any collision registers to query as you can when working with player-missile shapes. Instead must simultaneously test for any other pixels within that byte's (or pixel's) screen location. The test is performed using the AND instruction.

The AND Instruction

The truth table for the AND instruction is as follows:

```
        ACC.    MEMORY  RESULT
        0       0       0
        0       1       0
        1       0       0
        1       1       1
```

Both Accumulator and memory must be on (set) for the result to be on (set).
If we take a screen memory location that has an object in it and AND it with a byte from our shape table, any duplication in any bit location where something is non-zero result.

```
        0 1 1 1 1 0 0 0    Background
AND     0 0 0 1 1 1 1 1    Shape
        ---------------
        0 0 0 1 1 0 0 0    Result $18 >Zero
```

Drawing While Testing for Collision

Usually, in any game, if a collision is detected, the object is to be removed. Your first instinct is to stop drawing the object since it is to be removed anyway. B (EORing) the screen and you stop in the middle of your shape, you are going to leave a mess. It is much better to set a collision flag, finish drawing the shape completely EORing the shape off the screen.

```
        LDA (SHPL,X)      ;GET BYTE FROM SHAPE TABLE
        AND (HIRESL),Y  ;AND WITH SCREEN IMAGE
        BEQ DRAW          ;BRANCH ON NO COLLISION
        LDA #$01          ;SET COLLISION FLAG
        STA ESET
DRAW    LDA (SHPL,X)      ;GET BYTE FROM SHAPE TABLE
        EOR (HIRESL),Y  ;EOR WITH SCREEN IMAGE
        STA (HIRESQ,Y     ;PLOT ON SCREEN
```

Collision Detection - A Special Case

Any two objects of byte size or larger should have no problem with collision detection, especially if you are working with solid white objects. But there is a s
in which collision detection would not work. Let us assume that we have a blue spaceship and a green alien that appear to collide. If ,we examine their bit pat
never coincide.

```
        B G B G B G B G B G B G

        ---------------------
        0 0 1 0 1 0 1 0 1 0 1 0 SHIP
AND     0 0 0 1 0 1 0 1 0 1 0 0 ALIEN
        ---------------------
        0 0 0 0 0 0 0 0 0 0 0 0 RESULT O
```

The solution is to test the ship against screen memory with what is called a "mask" of the ship's shape, as if the ship were solid white. We take this mask of th
green pixels lit, and AND it against the alien occupying the same screen locations. A collision will be detected in this case. We set a flag, and then take the ap
shape table and EOR it against the screen.

Blimp Example

A good raster graphics example would be one that would be difficult or impossible to do with either animated character graphics or player-missile graphics. T
this example is eight bytes wide (nine if you count the byte needed for the offset shapes), and thirty-one scan lines deep. By artifacting, we are able to produc
colors: blue, green, and white. The shape is outlined below.
The blimp is joys tick-con trolled and therefore free to move anywhere on the screen. You have to be very careful that you don't try to plot the shape beyond the
plotting bytes beyond the right edge would produce a wraparound effect at the left edge one scan line lower, plotting beyond scan line 191 could create severe
portion of memory. If we exceed the bounds of our YVERTL, YVERTH tables, unknown pointers to our plotting position in screen memory would be placed in z
position cannot exceed 192-31 = 161. All of these tests are incorporated in the joystick subroutine.

Flickering Problem with Large Raster Shapes

The first time we attempted the example, we placed the raster drawing code outside of VBlank and the music routine in VBlank. Unfortunately, the rastered ir
image, after remaining on the screen for several frames, has to be erased at some point before it is redrawn at its new position. Since this takes place when the
there is a slight gap, possibly as long as a frame, before the redrawn image is in place. This never seems to be a problem on other computers like the Apple 11
techniques to produce their television images while the Atari does not.
We then felt that the animation should become flicker-free if we moved the raster drawing routines inside VBlank. This way the rastered image could be erased
beam was mostly off-screen. We arranged the code so that the shape is drawn initially, remains stationary on the screen for three television frames, then is era
fourth frame. A timer called TDELAY is set to zero after each erasure, and increments with each frame. A test will cause a branch past the erase-moveredraw c
three. When it is equal, it will erase the shape, read the joystick, calculate its new position, then redraw it in that position.

We feared that the code might be too long to fit within one Deferred VBlank cycle because the shape was nine bytes by thirty-one scan lines. Having never enc
became confused with the buggy results. The code was arranged differently within the Vblank at the time. The sound routine was last, and I was drawing the ra
routine invariably drew the shape then hung the first time it was run after assembly, but would actually execute the code after a system reset. A more serious pr
scan lines directly beneath the shape were garbaged. The routines worked when the code was outside VBlank.

Testing Whether Code Finishes Before VBlank Ends

After many wasted hours, we decided that a test would be needed to determine if and when the end of the VBlank code was ever reached. Obviously, if we re
problem; however, if we didn't, we would have to finish it on the next cycle. Let's assume that we haven't finished it when the computer says it's time for a ne
saves all of the registers and its position within the code just like it was outside VBlank. Now when a new VBlank Interrupt occurs, it begins again from the t
routine but when it tests if VBFLAG = 0, it discovers that it never finished the last VBlank and exits through the exit VBlank subroutine at $E462. The compu
position in the code when it was interrupted. It then finishes the VBlank routine. While this is a good example of how to correct the problem of VBlank routin
completely smooth out the animation. However, it is slightly better than when the raster code was completely outside VBlank.
If you would like to observe what happens if the above method isn't incorporated within your program, try removing the JMP $E462 statement. The result is a s
rastered shape is plotted in pieces on different scan lines, and the display begins to roll.

Background Sound

The background sound throughout our raster example is a familiar tune. The sound routine, which is explained in the next section, reads the individual notes a
runs in VBlank because the length of the notes uses the system timers.

Download RASTER.EXE (Executable program)
Download / View RASTER.LST (Assembler listing)
Download / View RASTER.S (Source file)
Download RASTER.OBJ (Object file)
Download / View RASTER.RAW (As printed in book)

Sound

Sound complements graphics in nearly all arcade-style games. While most people think of sound effects as the only necessary sound, the addition of an origin
contribute greatly to a game's overall popularity. In either case, the Atari, with its four-voice sound chip, is well-suited to the task.
The Atari computer has four independent voices that can vary in pitch by more than three octaves. The tone can vary from very pure to highly distorted. In addi
loudness level, completely independent of the television's volume setting.

BASIC's Sound Statment

In BASIC, the SOUND statement takes the following form:
SOUND Voice, Pitch, Distortion, Loudness

The first parameter Voice is simple. There are four voices or channels whose numbers range from 0-3. It takes a separate sound statement to activate each ch
they are all off at anytime, but anyone can be selectively turned off by setting Pitch, Distortion, and Loudness for that voice to all zeros.

Pitch can vary between 0 -255. The value'N' is used in a divide circuit. For every N pulses coming in, one pulse goes out. As N gets larger, the output pulses be
lower note. A value of 121 produces a middle C tone. A pitch of 60 produces a C tone one octave higher, and a pitch of 243 produces a C tone one octave lowe
approach the edge of human hearing and may not be audible on a television speaker that lacks a tweeter.

The Atari computer produces both pure and distorted tones. The term distortion is actually a misnomer. All of the sound waves on the Atari are square waves. D
of a degradation of the wave form like in harmonic audio, but by selectively removing pulses from the waveform. A more appropriate term would be noise. Distc
generate pure tones. Other even-numbered distortion values (0,2,4,6, and 12) introduce different amounts of noise into the pure tone. The quality of the sound
the distortion. Some combinations, mainly distortion 12, combine to produce an undistorted secondary tone with harmonic overtones.

Loudness is controlled by the fourth number in the SOUND statement. The value varies from 0 (silent) to 15 (loudest) and is fairly linear for a single voice. The
pitch. High-pitched sounds seem quieter than low-pitched sounds. If you are working with multi-channels, the sum of all four channels should not exceed thirty-
audio output. The sound produced tends to actually lose volume and assume a buzzing quality.

Sound Duration

Since the SOUND statement lacks a duration parameter, sound can be turned on and then off by using an empty FOR ... NEXT loop as a delay. It is largely ex
NEXT loops iterate at approximately 450 times per second. A loop that goes from 1-225 would cause a delay of half a second. Thus, the following three lines
for onehalf second, then turn it off.

```
100 SOUND 0,121,10,10
110 FOR I=1 TO 225:NEXT I
120 SOUND 0,0,0,0
```

Sound Effects

Simple sound effects are created largely by trial and error. Many use FOR ... NEXT loops to either vary the pitch or vary the volume. Some do both. The pistc
Chapter 5 varies the volume. The bonk sound of the brick being removed is similar but at another low pitch. Both sounds use distortion or noise to achieve the

```
100 REM - PISTOL SOUND
110 FOR L=10 TO 4 STEP -0.25
120 SOUND 0,10,0,L
130 NEXT L

100 REM - BONK SOUND FOR KNOCKING OUT BRICK
```

```
110 FOR L=15 TO 0 STEP -0.5
120 SOUND 0,20,2,L
130 NEXT L
```

It is also possible to vary both the pitch and the volume simultaneously in a loop. The following example simulates the sound of a falling bomb. It begins witl changes to a low pitch, followed by the thumping sound of an explosion.

```
100 REM - FALLING OBJECT
110 FOR L=30 TO 200 STEP 3
120 SOUND 0,L,10,L/25
130 FOR K=1 TO L/10:NEXT K
140 NEXT L
150 SOUND 0,20,0,14
160 SOUND 1,255,10,15
170 FOR K=1 TO 150:NEXT K
180 SOUND 1,0,0,0
```

Most sound effects have to be placed in a larger loop with the graphics or player-missile commands, or motion will stop while the sound routine runs. The probl the time delays and preset durations of the sound effects. Worse yet, the location of these routines within the program changes the result. This occurs because number list whenever it encounters a branch or GOTO instruction. Obviously, it finds line numbers at the beginning of the program before it finds line numbers solution to the problem is to run your sound routines in the VBlank period, and this approach requires Machine language programming skills.

Sound-Assembly Language

The POKEY digital I/0 chip controls the audio frequency and the audio control registers for all four sound channels. The AUDF# (audio frequency) locations AUDC# (audio control) locations are used to control distortion and volume. The sound locations are as follows, and they are write registers only:
AUDF1 = $D000 AUDC1 = $D001
AUDF2 = $D002 AUDC2 = $D003
AUDF3 = $D004 AUDC3 = $D005
AUDF4 = 3D006 AUDC4 = $D007

Frequency values range from $00 to $FF. POKEY actually increments this number by one before sending it to its divide by "N" circuit. For every N pulses comi the higher the value of N, the lower the tone. The rate of the pulses depends on the POKEY clock.

AUDC1-4 Sound Registers

The AUDC1-4 locations control both distortion and volume. The bit pattern is as follows:

The lower four bits control the volume level (0-15). Zero means no volume, while 15 means as loud as possible. The only constraint here is that the total volu does not exceed thirty-two. Bit 4 is a volume-only control. Turning this bit on will force the speaker cone out. Trouble is that this by itself won't produce a tor repeatedly forcing the cone in and out rapidly. This bit can be useful to advanced sound programmers.
The upper three bits control the distortion. Distortion is produced by first dividing the clock value by the frequency, then masking the output using the various pc pattern. The result is finally divided by two. Poly counters or polynomial counters are actually shift registers that produce various degrees of distortion in randor Since they are repeatable, they are predictable and are useful for generating sound effects. In general, the tones become more regular or recognizable with fev masking the output. The 17bit poly counter is useful for white noise effects like a waterfall, while the 4-bit poly counter is useful for a motor sound.

```
BIT     7       6       5
        0       0       0       5-bit, then 17-bit polys
        0       0       1       5-bit poly only
        0       1       0       5-bit, then 4-bit polys
        0       1       1       5-bit poly only
        1       0       0       17-bit poly only
        1       0       1       no poly counters (pure tone)
        1       1       0       4-bit poly only
        1       1       1       oo poly counters (pure tone)
```

AUDCTL Register

In addition to the independent channel control bytes (AUDC1-4), there is one other register, AUDCTL at 53768 or $D208, that affects all of the channels. Eac specific function.

```
Bit     Description
7       Makes the 17-bit poly counter into a 9-bit poly
6       Clock channel one with 1.79 MHz
5       Clock channel three with 1.79 MHz
4       join channels one and two (16 bit)
3       join channels three and four (16 bits)
2       Insert high pass filter into channel one, clocked by
        channel two
1       Insert high pass filter into channel two, clocked by
        channel four
0       Switch main clock base from 64 KHz to 15 KHz
```

Shifting the 17-bit poly counters to 9-bit poly counters by setting bit 7, will create more repeatable sound patterns rather than white noise-type patterns. Settin frequency (setting bits 5 and 6), will produce higher tones. Likewise, setting the bit 7 from 64 KHz to 15 KHz will produce much lower tones.
If you couple two of the sound channels by setting either bits 3 or 4, you reduce the number of channels to two but gain increased tonal range. Normally, you g eight bits of a single channel, but the combined 16-bit register increases the tonal range to nine octaves.

Sometimes you may encounter problems POKEing sounds in BASIC or in Machine language without initializing the sound registers. BASIC requires a null sou 0,0,0,0. In Machine language you need to store a 0 at AUDCTL ($D208), and a 3 at SKCTL ($D20F).

Background Music

One of the most pleasing uses of sound is to play musical tunes quietly in the background, during many games or at least use them to enhance an animated titl
run in the Vertical Blank period so that the note lengths remain accurate. Generally, you store the notes and durations of the tune in a table. The Atari reads th
duration from the table. It then turns on the note and sustains it until the timer at $14, which counts in jiffies, reaches the value set by the duration. At that poi
computer reads the next note and duration. Two consecutive notes with the same pitch sound like they run together as a much longer note. It is often necessar
jiffies between the notes. With this method, it is very simple to play an entire musical score without affecting the play mechanics or speed of a game. Be caref
that timer elsewhere in the game.
We use the value of $FF for the note as a flag to indicate the tune is finished. While it could be used to conclude the piece, we use it to reset the pointers to the
endlessly.

The lengths of the different notes is summarized in the table below:

```
NOTE            JIFFIES
Sixteenth       8
Eighth          15
Quarter         30
Half            60
Rest            60
```

(The book indicates a code sample should be inserted here, but it is missing.)

Sound Effects

Explosion sounds are simple to implement in Machine language within the Vertical Blank routine. Basically, you need a very irregular rumbling sound that sl
Setting the distortion to zero sets up 17bit poly counters that produce quite irregular sound. The duration of the sound is controlled by a timer that counts dow
control the volume level so that it decreases as a function of the value of the timer. For instance, if the sound is to take one second, SEXTIME, short for Set E
64 and is decremented every jiffy. If VOLUME = SEXTIME / 4, then the volume will decrease from 16 to 0 as SEXTIME counts down to 0. The code follow

```
SOUND3  LDA SEXTIME     ;CHECK EXPLOSION TIMER FLAG
        BEQ .1          ;IF AT 0, NO SOUND
        DEC SEXTIME     ;COUNTDOWN
        LSR             ;DIVIDE BY 4 TO GET VOLUME 16-0
        LSR
```

```
        STA AUDC4       ;TELL POKEY NEW SOUND VOLUME
                        ;UPPER NIBBLE (DISTORTION = 0)
        LDA #$40        ;TONE
.1      RTS
```

Laser fire can be simulated by rapidly changing the frequency from a high pitch to a lower one in discontinuous jumps while using a distortion set at 6. This p
than a smooth frequency transition. You can implement this effect by making the timer, SLTIME, short for Set Laser Timer, a function of the frequency. If Fre
each time SLTIME is incremented, the tone will jump in increments of 16. Remember, the higher the N in the divide by N he lower the tone. The problem her
short if to increment simply from 1 to 15. Therefore, a secondary loop delays each tonal jump by 4 cycles. The entire sound routine takes 60 jiffies rather than
code are below:

```
SOUND       LDA SLTIME      ;CHECK LASER TIMER FLAG
            BEQ .3          ;IF 0 EXIT
            CMP #$0F        ;TIMER GOES FROM 1 TO 15
            BNE .1
            LDA #$00        ;TURN SOUND OFF
            STA AUDF1
            STA AUDC1
            RTS
.1          LDA SLTIME1      ;CHECK DELAY TIMER
            BNE .2           ;IF NOT 0 COUNTDOWN TILL IT IS
            LDA DELAY1       ;GET NEW DELAY VALUE
            STA SLTIME1      ;STORE IT
            INC SLTIME       ;INCREMENT MAIN TIMER
                             ;(THIS IS ALSO OUR FREQUENCY VALUE)
.2          DEC SLTIME1      ;OUR FREQUENCY VALUE
            ASL              ;MULTIPLY BY 16
            ASL
            ASL
            ASL
            STA AUDFI        ;NEW TONE VALUE
            LDA #$86         ;DISTORTION 8, VOLUME 6
            STA AUDC1
.3          RTS
```

Chapter 9
Advanced Arcade Techniques

Maze Games

Maze games achieved the height of popularity with the debut of eat-the-dot games like Pacman and Ms Pacman. They weren't the first eat-the dots games: that honor goes to a car game called Head-on. They were, however, the first games to transpose the usual open field chase or pursuit games to the narrow confine of maze corridors.

Topographically, a maze is merely a network of interconnecting paths. These pathways constrain the player's movement. In a sense each individual section of the maze gives the player a set of movement rules. The passage walls that are open allow the player to reach the next section, while the closed walls block movement in other directions.

The maze can be divided into a number of small blocks, each the height and width of the passageway. Depending on the graphics mode, each of these blocks could assume the size of a character. That way each character becomes one of the blocks in the maze. The characters can be open blocks, with various combinations of open exterior walls, or they can be floors and ladders for use in a climbing, jumping arcade game. While most people do not think of games like Donkey Kong and Apple Panic as maze games, they too require a set of movement rules to keep the player confined to floors and ladders.

The instructions that guide a player about the maze can be very complicated, or they can be quite simple. They can take thousands of bytes or just several hundred. Naturally, each individual block needs to be a part of the playfield and requires one byte per block. If we drew out maze in graphics one characters (eight dots by eight rows), we could have a maze twenty blocks wide by twelve rows deep. That takes 240 bytes of memory. Next we need instructions to tell the player if he can move up, down, left or right from the center of the block where he is. That could take as many as four bytes per block. These could all be placed in tables so that if we knew which block we were in, we could index into each table look up the legal moves for that block.
Storing a zero for an open pathway and a one for a closed pathway doesn't use a memory location to its capacity. Only the lowest bit is used. If we could combine all four directions into one byte, with each direction using one bit, we would only use a fourth as much memory and still have half of the byte left. With Left using the lowest bit or 0th bit, Right the first bit, Up the second, and Down the third, we can test each of these bits by ANDing with a MASK that contains a I bit in the bit position we wish to test and 0 bits everywhere else.

```
LEFT    CLOSED  0000 0001
RIGHT   CLOSED  0000 0010
UP      CLOSED  0000 0100
DOWN    CLOSED  0000 1000
```

Thus to test if the up direction is closed we AND the instruction byte with $04. In the following example only the right direction is open.

```
0 0 0 0 1 1 0 1 INSTRUCTION BYTE
0 0 0 0 0 1 0 0 AND #$04

_______________
0 0 0 0 0 1 0 0 RESULT IS POSITIVE IF UP DIRECTION IS CLOSED
```

We could set a flag for the gates that we find open. If we did the above test and found the result positive, or the up direction closed, we could set FLAGU = 1. The four flags FLAGU, FLAGD, FLAGL, and FLAGR, would be set to 0 if found open and to I if closed.
Since we need to design a character set to visually show which walls are open and which ones are closed, it would be beneficial to mirror the byte value of the block's instruction byte. Thus, if a character had the only the left wall closed, its instruction byte would have a value of one. Therefore, we will design the first character in the character set to have only the left wall closed. The Oth character has no walls closed, and the fifteenth character in our table has all four walls closed. The instruction byte that reflects this has a value of 15 ($0F). Each of the characters in our set is shown below.

The playfield consists of twelve rows, each with twenty of these characters. As a whole they make up an entire maze. A player in the top left hand block in the maze is in the Oth position of the character data that ANTIC uses to generate the playfield or maze. If the player moves one row down to position XB,YB (0, 1) it is at the twentieth position of the character data; and one position to the right (1,1), the twenty-first position position in the data. This can be formalized as:
BLOCK = (YB*20) + XB

If we index into the screen data, SCREEN,Y where the Y register contains the value of the character at the player's position. That number is the same as the sum of the individual instruction bits for legal movement in each of the four directions. The fact that only one table is needed for both the screen data and the legal movement instructions is not a coincidence. It is simply a clever way to condense data.

The design of any maze game should include a simple premise with a simple set of rules. The object of PacMan is to eat all of the dots on the maze floor in order to advance to the next level. The object of the maze game is for the player's letter to chase two higher letters in the alphabet. The player catches the next higher letter in order to become that letter as it advances toward the letter Z and a harder maze. just as the four ghosts are the adversaries in PacMan, the red minus sign is the adversary here. It constantly pursues your player and if it catches it, your letter value decreases by one. Thus, the game begins with the letter A chasing letters B and C. If it catches the B, it becomes a B and continues chasing a C and D. If the minus sign catches your player at that point, it reverts back one letter to the letter A.
There is no point scorekeeping system for this game. The progress you make during the game becomes a visual scorekeeping system. The goal is to progress forward, not to gain points. If we were to award 100 points for each letter gained, and 50 points for each letter lost, players might prolong the game indefinitely, advancing several letters than losing a few.

All the players move at exactly the same speed. This forces strategic play, since you can't catch the fleeing letters by simply tailing them. The game must appear winnable, so we never make the player start the game over when he is caught by the minus sign. The game would become frustrating if a player advanced to the letter T and reverted back to an A just because the minus sign caught him repeatedly. Beginners would also find the game frustrating if the minus sign caught them and put them in another random position in the maze, just as they were about to trap a letter.

The maze game code is much too long to run within the time frame provided by vertical blank. Since I had tested my theory of maze game logic with a single player within the vertical blank period, I decided to place the remainder of the game code outside the vertical blank time, but sychronized with it. Normally code entirely within VBLANK runs in a endless loop (FOREVER JMP FOREVER) and only leaves this loop during the interrupt period. When your code runs outside VBLANK it runs until the VBLANK interrupts it, then executes the code in the interrupt routine. At the end of the vertical blank period it returns to where it left off in the code. With the 6502 running at 1.8 MHz a moderately short program is likely to cycle through twenty or thirty passes between VBLANK interrupts. The only solution to synchronize the two pieces of code is to allow the code to run once then wait in a tight loop testing for some flag that can only be set if the program reaches the vertical blank routine. This is precisely what we did. Our VBLANK routine sets a flag call VBFLAG = 1. The main program code sits in an endless loop testing this flag. It can Only exit the loop to the beginning of the program code when VBFLAG = 1.

```
FOREVER LDA VBFLAG      ;TESTFLAG
        CMP #$01
        BNE .1
        JMP LOOPM       ;EXITS WHEN VBFLAG=1
.1      JMP FOREVER
```

Player Movement

The player is joystick guided. When a player orders his player to move in a certain direction a number of things happen. First the program checks if the player is at the center of a block and if so checks which movement directions are legal. For example if a player initially started at the top left corner, it can only move right or down. If the player began moving down, he could no longer move left, but only back in the direction he came from, at least until he reached the center of the next block. So once a player begins moving the legal direction flags must be reset to reflect the legal moves between adjacent blocks. Second, since player movement should be automatic once a player begins moving in the desired direction, a auto flags denoted as DR, DL, DU, and DD must be set. The auto flag is on when set to I and off when set to 0. So if our player is heading downwards, DD=1.

Joystick Subroutine

The joystick subroutine is similar to other joysticks routines except in its treatment of diagonals. Only a single bit has to be tested if it is off to determine the desired direction for the up, down, left and right positions. But in the four diagonal directions two bits are off. Since we can only move horizontally or vertically in the maze, a decision must be made to which direction the player means. If a player were moving right and he wanted to turn

up at the next intersection he would likely point his stick diagonally up and right. Since he is travelling right in an automatic mode, he probably doesn't mean to order his player right but instead up. So if the right auto flag is on DR=1, it means that he is already travelling right and intends to go up the next time he reaches a block that allows him to move in that direction. Similarly, if the is moving up, DU=1 so he would probably intend to go right at the next intersection.

If you look at the flow chart for the joystick subroutine, you will notice that in addition to setting auto flags and resetting legal flags for each of the four primary joystick directions, a flag called INHIBIT is set sometimes in the diagonal test. This is necessary because if for example we had an up and right diagonal and we were traveling the logic would say that we want to test for a right move. Since the right bit is off, the apropriate flags will be set correctly, but when it reaches the up bit test that bit is also off so that it will try to set latches for that direction too. But if it has to pass a INHIBIT = 1? test, it will prevent the program from reaching the second bit position test, The INHIBIT flags are only set for situations where the code will branch to the right and left bit tests first. Code that branches to either the up or down bit tests can not reach, the left and right bit tests to cause problems.

Whenever a joystick command is given to change the player's direction, the joystick routine checks to see if the player can move in that direction from its current position. It checks to see if the legal move direction flag is open or closed in the desired direction. If it is open (0), it shuts off all the auto flags except the one for the new direction. Finally it resets the legal move flags so that it can only go forward or reverse between blocks, and then moves the player one unit forward. If you try to command the player to move in a direction that it can't go, it will keep on going forward automatically if it can.

Auto Mode Code

The auto mode is what makes the joystick routine particularly smooth and responsive. If it didn't exist, you would have to exert more effort to get your man to steer properly. Besides always needing to push the joystick one way or the other, you would have to be careful to reach the exact center of a block before successfully negotiating a turn.

The auto mode code simply checks to see which auto flag is on and checks if it is legal to continue moving in that direction. If not the player stops until it receives a new command.

Legal Move Subroutine

A subroutine appropriately called LEGAL determines in which directions the player is allowed to move. You input the player # in the X-register and its current X and Y positions, and it sets the legal move flags FLAGL,X, FLAGR,X, FLAGU,X, and FLAGD,X for the appropriate player. It is a two step process. First it decides if the player is at the center of a block. Each block is 8 pixels wide by 16 pixels deep. For a player to be at the center of a block it must be at an exact multiple of 8 horizontally and an exact multiple of 16 vertically. The player's position is in player-missile coordinates. The top left corner of the maze is at location 48,32. And the 8 scan line high pla'yer is initially 4 scan lines lower in order to center it in a 16 scan line higher to center it in a 16 scan line high

block. We can get two values TEMPX and TEMPY by subtracting 48 and 36 from the player's horizontal and vertical positions respectively. The coordinates of the block XB,YB that the player is currently in are calculated as follows.

```
TEMPX = XP-48
        XB = TEMPX/8

TEMPY = YP-36
        YB = TEMPY/16

BLOCK = YB*20 + XB
```

It is quite simple to determine if the player is at the center of the block by checking the values in TEMPX and TEMPY. If any of the first three bit positions in TEMPX containing anything (a remainder) we have a value that is not an exact multiple of 8. You can AND with #$07 to check the first three positions of TEMPX for a non zero value. For example:

```
00001001 TEMPX=#$09
00000111 AND #$07
```

```
________
00000001 RESULT positive
```

Similarly we can AND with #$0F to test the first four positions of TEMPY for a non-zero value or a remainder.

If the above two tests prove that we are at the center of the blocks, we can test the individual direction bits for the character number for that block. We open all of the flags before testing each direction. The address to the screen data or maze map has previously been put in zero page at locations MAPL, MAPH. The code for testing if the block's left direction is open is shown below.

```
        LDY BLOCK       ;BLOCK WE TEST IS USED AS INDEX INTO TABLE
        LDA (MAPL),Y    ;GET VALUE OF BLOCK
        AND #$01        ;TEST LEFT DIRECTION
        BEQ             ;IF RESULT 0 THEN LEAVE FLAG OPEN
        LDA #$01        ;SET FLAG CLOSED
        STA FLAG,X      ;X REG. CONTAINS PLAYER
.2      LDA (MAPL),Y
```

Computer Controlled Players

The three other players, the two letters and the minus sign, are computer controlled. The two letters must continually flee from the player's joystick controlled letter, while the minus sign homes in on the position of the player's letter.
Each of the computer controlled sprites must determine the direction or directions that they wish to travel relative to the player's letter in order to either seek it or flee from it. If the playfield were completely open the logic would be rather simple. The minus sign would approach the player in one of two random directions, horizontally or vertically until it was even in that axis, then move towards it in along the opposite axis. A letter fleeing from the player's letter would would move away from the player along one axis until it reached the playfield boundary, then move towards the corner opposite that of the player's letter. Of course it would get trapped in the corner.

Mazes have lots of corners where fleeing letters and pursuing minus signs could become trapped. A pursuing player that found itself in a parallel corridor would closely follow your motions as you moved back and forth safe but next to it. There might not even be an escape since you and it would reach the next turn at exactly the same time. On the other hand, a player that fled would often get stuck in a corner awaiting for you to become parallel with it along one corridor before fleeing along the other. As many maze game designers will testify, the only practical solution is to force both the pursueing and fleeing computer controlled players to always move forward in the corridors, never in reverse.

The logic needed to keep a player moving forward is not always simple, for sometimes it disobeys the general rules given to each type of player to either pursue: or flee from the joystick controlled letter. In general, if a player is at a decision making point, the center of a block, it must decide if there is a direction available, other than the direction it is traveling in or reverse. If it doesn't have a choice it just continues in its current direction. But if there is a choice it must decide if one of those directions is a better choice than the direction it is currently travelling.

A set of four relative direction flags, RELL,X, RELR,X, RELU,X and RELD,X, can be set to indicate whether the computer controlled letters and minus sign should move towards our letter or not. Each of their X and Y coordinates are checked against the joystick controlled letter's X,Y coordinate. Obviously if the player's letter is to the left of the minus sign, the minus sign would want to move left so RELL,X is set to 1. The minus sign is player #3 (X-register = 3) so that RELL,3 = 1. The minus sign would also like to move up so that RELU,3 =1. On the other hand, the letter B, which has a similar relative positioning as the - sign in the diagram below, wants to escape and move in the exact opposite direction. The testing algorithm is the same for both types of players but instead of setting RELU,1=1 the letter B would like RELU,1=-1. If it wants to go the opposite direction it is much simpler to just set the flag for the opposite direction than to test for negative relative flags. ThusRELD,1=1 is the exact equivalent and the letter subsequently flees.

The next step is to decide if any of these relative direction flags match any of the legal direction movement flags. For example if RELL,1=1 and FLAGL,1=l then either the fleeing letters or the pursuing minus sign can take the turn. If the move is possible it sets a move flag MFLAGL,1=1 for that direction. The reader at this point is probably muttering, "Not another flag!" It is actually necessary because there can be more than one possible direction to move. In that case the program will have to choose one direction to move. There is a counter called NUM that increments each time it sets a move flag. If NUM >1 we will have to choose a direction randomly. The choice of directions are arranged in pairs left-right and up-down. Depending on the random number value, the test will be on either the vertical or horizontal direction first. There is no danger that the program will miss finding a set move flag if it branches past the horizontal axis since it only occurs when NUM=2. The two flags

that have been set can't be both left and right since a fleeing or pursuing player would choose only one direction to move in any axis. Once it determines which move flag has been set it resets the auto flag and moves in the appropriate new direction.

There are cases especially in corners where the relative flags and the legal flags don't match. In these cases where the player would become trapped, the player must be forced to make the only legal turn even if it means moving the wrong way towards the joystick controlled letter. I called the subroutine CORECT. The routine, realizing that the player was moving forward when it became stuck, tests which auto flag is set and compares it with the legal move flag for that direction. If it can continue moving forward it goes to AUTOP, the automatic mode for the player, and bypasses the code to force it to make the corner against its will. However, if it becomes stuck, it, it tests the legal move flags for the two directions along the opposite axis. For example if it were heading right and became stuck it would check FLAGU,X and FLAGD,X. There is no need to check the FLAGL,X because the player is not allowed to reverse itself. Once we have determined the new direction we reset the auto flags and move it one pixel in that direction.

Player Collisions

Eventually, either the joystick controlled letter is going to catch the next higher letter or the minus sign is going to catch it. When any two players overlap a collision register is set which we can test. If the joystick controlled letter catches the next higher letter it must become the next higher letter and the letter that was just caught becomes a letter two higher than it was. Thus letter A which catches B becomes a letter B, letter C remains the same, and letter B becomes a D. The caught letter must be repositioned somewhere else on the screen preferably at the opposite end of the

screen. If our hero is at the bottom left of the maze, we put the new letter at the top right. The four possible placement positions aren't random but in specific spots with specific starting directions. The actual repositioning is accomplished in a subroutine called PLACE. It is very simple and clearcut routine. It does randomize the left-right starting positions for the bottom two locations because occasionally two letters are caught almost simultaneously in either of the top two outer pathways.
A collision of your letter with the minus sign results in a decrement of one letter except when you are the letter A. In fact all three letters are decremented one letter so that if you are the letter C chasing a D, after the collision with the minus sign you are a B chasing a C in exactly the same position. Although the minus sign is repositioned at the bottom of the screen on the opposite side of the maze from the player, it was felt that penalizing your player by repositioning it at the top corner was frustrating to beginners who got caught all too often and felt that it was like starting over.

Each of the three lettered players has a variable that points to which letter it currently is. POINTO refers to the joystick controlled letter, and POINT1 and POINT2 refer to the two fleeing letters. This variable also controls which letter shape is taken from the shape table during the PLOTSET subroutine. The shapes are arranged in numerical order. Shape #0 is the minus sign. The 26 letters follow in sequential order. Two blanks are placed at the end so that when the player reaches Y it is chasing only a Z and one invisible player. When the player reaches Z there are two invisible players still on the screen,

While it is possible to test whether player #1 or player #2 has collided with our joystick controlled letter, you can't be sure that the collision is with the next higher letter because that letter alternates between the two players. Therefore a test comparing the values of the two colliding letters must also be done to determine if the two letters are one apart. Say our player is an E (POINTO=5) and it collides mistakenly with player #1 that currently is a G (POINTI=7). The test POINT1-POINTO gives a value greater than 1. Therefore we ignore the collision. But if we collide with player #2 that is an F (POINT2=6) then POINT2-POINTO is equal to 1. We then increment POINTO our player to a F and increment POINT2 twice to H. Now we have a F chasing a G and H.

Second Maze Level

Eventually the player reaches the letter Z in the game. There is a four second pause before a new maze appears. Setting up the screen is simple. The 220 bytes of character data for the second screen is moved into screen memory starting at location SCREEN. The zero page pointers MAPL and MAPH, which are used by the

subroutine LEGAL to obtain the value of a particular block, are also set to point to the character data at DSCREEN2.

```
       LDX #$00
SLOOP1 LDA DSCREEN2,X   ;LOAD NEW MAZE DATA
       STA SCREEN,X     ;STORE ON SCREEN
       CPX #$F0 ;DONE?
       BNE SLOOP1       ;NEXT BLOCK
       LDA #DSCREEN2    ;SETUP ZERO PAGE POINTERS TO DATA
       STA MAPL
       LDA /DSCREEN2
       STA MAPH
```

The layout of the second maze was designed to be quite harder. The only open pathways are along the bottom and two sides. The interior pathways are designed so that if you don't concentrate and choose your pathways with care, a wrong turn will nearly always give the pursued letter a bigger lead.
It is possible to add many more levels to the game. If there is only a few you could duplicate the above code and by testing which level you have just finished branch to the appropriate block of code to put the maze character data in screen memory. More advanced programmers should set up an indexed table for the hi byte pointer to each 256 byte block of character maze data if there are more than four maze levels.

Pause Feature

A pause game feature is important to most arcade games, especially one in which the game might last more than several minutes. The problem with an Atari is that game code within the vertical blank executes 60 times per second regardless of any pause control in the main program loop. If the game is to stop, the pause code must branch past the game code within vertical blank to XITVBK. In our case, if we branch past the VBLANK flag which we use to syncronize the non-vertical blank code, the rest of the game action will also stop. The pause code shown in the flow chart below is placed at the very beginning of vertical blank.

Download ALPHMAZE.EXE (Executable program)
Download ALPHMAZE.OBJ (Object code)
Download / View ALPHMAZE.LST (Assembler listing)
Download / View ALPHMAZE.S (Source file)
Download / View ALPHMAZE.RAW (As printed in book)

Tank Game

A tank duel between two or more tanks was one of the first few classic/games developed for coin-op play in the arcades. It was quickly translated to the Atari 2600 game system under the name Combat, and was supplied with the first four to five million game systems. Regrettably, it was never rewritten for their home computers. While the cartridge had some interesting variations like Tank-Pong, in which the shells bounced off walls for quite some distance, it was unrealistic in that the tank turret could only fire in the direction the tank pointed.

Basic Design of Game

Our game, Tank Battle, is designed to be much more realistic. The tanks, which can turn and drive in eight directions, are equipped with rotatable turrets so that they can fire in directions other than the one that they are traveling. The terrain features or barriers, which are useful for hiding behind, can be blown away by tank fire. It

takes two shots to completely fracture one of the terrain blocks. Tanks that are blown away are immune to enemy fire for several seconds upon reappearance. This is necessary since the tank always reappears in the same position and it would be only fair to allow the player to move his tank or fire back at a tank waiting in ambush.
Regrettably, the control system is a compromise. Game design would be so much easier if joysticks had two buttons. The single button serves a dual function for turning the turret and firing the gun. When the button isn't pressed, the tank is in the steering-drive mode. Pushing to the left or right rotates the tank, and pushing forward or back moves the tank in those directions. Since the tank can't penetrate barriers, it often needs to be backed up to turn away from the obstacle. Pressing the button puts it into the turret-fire mode. The tank can still be driven forward and backward, but pushing the stick left or right rotates the turret counterclockwise and clockwise respectively. The gun is fired when the button is released.

The use of two players for each tank, one for the body and one for the turret, thwarted any attempts to make it a four player game. We're not saying that it couldn't be done, but we could just see the flickering that would occur if two tanks using the same players ended up on the same scan lines. Actually, now that we think about it, it could have been done if we sacrificed the distinct second color of the turret. The turret might not show clearly, but it would reduce each complete tank to one player. Now you would need sixty-four tank shapes, one for each combination of tank direction and turret direction. As it is we use eight tank shapes and eight separate turret shapes for each of our eight joystick-controlled directions.
Only one missile can appear on the screen at a time for each tank. Each missile continues along its directed path until it exits the playfield or reaches either the enemy tank or a blow-away wall. Since the missile is armed upon pressing the trigger, it becomes increasingly important to be able to not only adjust the turret's direction but also the tank's forward and backward position before the shell is actually fired by releasing the trigger. The inablity to reload and fire before the previous missile completes its trajectory may be quite realistic in tank warfare, but can be a liability in an arcade game. Perhaps this forces the game to transcend the quick reflexes genre to one of strategic play.

The game was virtually designed to run in Deferred Vertical Blank. While collisions, explosions, and missile movement are performed for both tanks every VBlank cycle, reading the joystick and the actual calculation and updating of tank and turret positions alternate for each tank every VBlank cycle. Tank #0 is updated on even cycles and tank #I is updated on odd cycles. This was necessary for it appeared that there were far too many instructions to fit within the Deferred Vertical Blank period.

Updating the tank's position is one of the more intriguing problems in this game. Calculating the tank's next position based on the direction that it is moving is quite straightforward. The formulas are:

PLAYERH = PLAYERH + HOFFS(NEWD)
PLAYERV = PLAYERV + VOFFS(NEWD)

where NEWD is the tank's new direction.

Tank Collision with Playfield

The tank obviously has to be prevented from moving into a wall. We wanted to avoid the unsightly bounce that occurs in many games that test illegal positions using player-playfield collisions. The problem is that you can't test the collision until you actually move there, but once you are there you have to move back from the illegal position, or in this case, away from the wall.
The solution is to test for a collision at the tank's next calculated position before the tank is actually moved there. This can be accomplished exactly the same way collisions are detected when rastering shapes on the screen, by ANDing the player byte against the playfield byte beneath it. If any two bits overlap, the result is a positive number. Remember that to do the comparison, the tank doesn't have to be physically where it is supposed to be. You need only calculate which player and playfield character bytes intersect and compare the data for an overlap.

By calculating the tank's new position NEWH, NEWV, the offset into the character XOFF, YOFF can be determined simply by ANDing the position with #$07. This is equivalent to the expression NEWH -8 (INT(NEWH/8)). It masks off the higher bits above 8 so that only the remainder is left. The actual character involved in the intersection also has to be obtained. Its position is at CHRX, CHRY and is obtained by dividing the new player position NEWH, NEWV by 8 in each axis. Its position in memory is CHRHI*256 + (CHRX + (20*CHRY)).

When the player shape is offset horizontally into the character, part of its shape is in that character and the rest extends into the next character to the right. If we are going to make a comparison in the data, we will need to obtain data from that character also. The eight bits that we need to compare are beneath the player byte but span the two different character bytes. If we look at the diagram below, out player shape is offset into the first character by five bits. Thus, three bits of the first character byte need to be combined with five bits from the adjacent character. Once we have a byte we can AND the character data with that of the tank data. The data is tested for each byte of the vertical overlap or until we detect a collision. If we don't detect a collision, we clear the carry before exiting the subroutine. However, if we do, we set the carry before exiting. Upon return we need only perform a BCS instruction to determine if the desired move is indeed legal.

0th byte overlapped character data $FF intersects tank data $38
1st byte overlapped character data $FF intersects tank data $C6

```
LDA (P0TMP1),Y           $38     1 1 0 0 0 1 1 0
AND BYTE                 $FF     1 1 1 1 1 1 1 1
_____________________________________________________
RESULT >0 (Collision) $38        1 1 0 0 0 1 1 0
```

Updating Tank Position and Rotation

The two tanks are updated on different VBlank cycles. While it is true that tank #0 is updated on even cycles and tank #1 is updated on odd cycles, there is a rest of two additional cycles before they are updated again. Essentially, we are in a four-cycle loop with two null cycles. If you look at the code beginning with the label UPDATE, you will notice that we loaded the Accumulator with the clock timer value and then ANDed it with #$02. If you look at the bit patterns for the numbers 0-3 and perform the operation, you will get the following:

```
ACCUM.      00  01  10  11
AND #$02    10  10  10  10
            ----------------
            00  00  10  10
```

Notice that we obtain non-zero values for the second and third cycles. We could branch on a non-zero test and skip updating the tank on these two cycles. Incidentally, the clock doesn't need to be reset every four cycles. It continues to count to 255. However, as far as the last two bits are concerned, 4 is the same as 0, and 5 is the same as 1, etc. We can then determine which tank to update by ANDing the clock RTCLOC with #$01. The value is either going to be zero or one, and that can be placed in the Xregister to obtain the appropriate tank's variables. There are several more tests that have to be performed. Obviously, if the tank is dead, we don't have to worry about updating it. Next we need to decide if we are in the turret mode. This is important because we need to distinguish between a button that hasn't been pressed and one that has been just released to fire. In the latter case, the turret mode is set so it branches past the first button test that would have shunted it to code to move or rotate the tank, and instead reaches the second button test. If the button is actually released, it means that the button had just been released and it trips the gun to fire. The turret mode is turned of f in that event. However, if the button is still held down it branches to ROTATE where a new turret direction is calculated every fourth jiffy.

Tank Rotations

The calculations to rotate either the tank or turret are very straightforward. The new direction is either incremented or decremented depending on joystick direction. Both TURRETD, the turret direction, and PLAYERF, the tank direction, have values between 0 and 7. These direction values are used to obtain the correct shapes to plot in the player-missile area of memory for both tank and turret. The tank and turret rotations are coupled such that the turret's position relative to the tank is constant as the tank is turned. If the turret points out the front of the tank, it must remain in that position when the tank turns. We update both the tank and turret directions in the code when turning the tank. You'll notice that we are always keeping temporary values for movement, including new tank and turret directions. Since the act of rotating the tank could cause a collision with the wall, it may be necessary to ignore the new values and simply not rotate the tank or turret.

Missile or Tank Fire

The beginning section of the VBlank code concerns collisions between missiles and playfield, missiles and tanks, and between two tanks. When a missile strikes the playfield it has to distinquish between border characters and barrier characters. The character set is as follows:

```
0    -Blank
1-6  -Border characters
7    -Unused
8-15 -Walls (in pairs)
```

The missile track ends when it hits a border character. If you choose to change the maze so that border characters aren't used, you will need to remove the comment field from some of the statements in lines 2540 to 2760. These statements actually test boundary conditions rather than simply character numbers from 1 to 6 to determine if the missile reaches the border. When the missile strikes a character that makes up a wall, it decrements it if it is an odd numbered character, or erases it if it is an even numbered character. The characters are set in pairs. The higher numbered character is a complete shape while the lower numbered character is the fractured shape.

Tank Explosions

Of course, it is very likely that the player's missile will hit its intended target, the other player's tank. If it does, the hit tank begins to explode. In order to prevent the tank from reinitiating its explosion by another closely fired

missile, we need to test if CYCLES >0. We also need to determine if the tank is immune from enemy fire just after it reappears. Again we test for a value IMMUNE >0. Both CYCLES and IMMUNE are counters that count down every VBlank.
The tank's explosion pattern is a series of horizontal lines that rapidly expand vertically outward along the tank's 8-bit-wide player band. The tank shape is replaced by a series of horizontal lines. Their vertical positions are stored in two arrays, DZAP and UZAP. These two arrays each store the vertical positions of eight of these horizontal lines. The lines are moved at different rates by adding or subtracting different values to their vertical positions. For example, if we consider the eight lines in the DZAP array, they all begin initally at X=80. Then DZAP(INDEX) = DZAP(INDEX) + INDEX, where INDEX ranges from 0 to 7 for the eight values in the table. The higher values in the table move the fastest downward. Eventually all of the lines exceed the screen boundary at #$CO and the explosion ends. Similarly, the eight lines in UZAP move upward as UZAP(INDEX) = UZAP(INDEX) -INDEX. They, too, eventually exceed the screen boundary at #$20 and the explosion sequence ends.

The collision routine that detects collisions between the two tanks has to be more selective than usual. If we aren't careful, the debris from the exploding tank could blow up the other tank if it had a similar horizontal position. This can be avoided by checking the explosion cycle counter to see if it is greater than zero.

The blown up tank is put back on the screen once the CYCLE counter runs out after four seconds. The IMMUNITY counter is then set to 255 jiffies or four seconds so that the enemy tank can't sit in ambush. It does give the defeated tank a slight edge for several seconds but this can be compensated by having the other tank stay out of the line of fire.

Game Timer

The game is timed to last three minutes. This is set up initally as two minutes and sixty seconds, and can be changed to suit the player. The timer operates in the decimal mode. Both the ones and tens digits for seconds are stored in the lower and upper nibbles of the variable SECONDS. The nibbles are separated and converted into character data before being stored in score screen memory at locations LINE3+10 and LINE3+11. The value for minutes, which is in the lower nibble only, is easier to obtain. It is eventually stored at LINE3+8. The game ends, GAME=0, when the timer reaches zero. ENABLE, a VBI flag, is also set to zero at that time so that the entire VBI routine is skipped when the game isn't in progress.

Download TANK.EXE (Executable program)
Download / View TANK.LST (Complete assembler listing)
Download / View TANKMAST.SRC (Master source file)
Download / View TANK.EQU (Source file)
Download / View TANKDATA.SRC (Source file)
Download / View TANKSUBS.SRC (Source file)
Download / View TANKVAR.SRC (Source file)
Download / View TANKVBI.SRC (Source file)
Download / View TANK.SRC (Source file)
Download / View EQUATES (Source file)

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter 10
Game Design Theory

There is no sure-fire way to predict whether a game will be successful, but there are certain attributes that contribute to success. Certainly a game should have a goalwithout one, what is the point in playing? Rules should be straightforward and logical. The game should also be a challenge; if it requires no skill, you will quickly tire of it. A game should evoke either fantasy or your innate curiosity; if it isn't novel or puzzling, it becomes boring. And lastly, arcade games, that by definition have a lot of action, should be easily controllable.

Game objectives take two different forms. There are games where you gradually approach the goal, like destroying a fleet of invaders in Galaxian, eating all the dots in Pac Man, or rescuing all of the hostages in Choplifter. There are other games where the goal is to avoid catastrophe. Examples of this range from preventing a nuclear power plant meltdown in Scram, to saving your cities during a nuclear missile attack in Missile Command, or preserving all of your fuel canisters in Ripoff. Do not confuse these two kinds of game objectives with the simplistic and mindless act of scoring lots of points by shooting everything that moves.

Goals must suit a player's expectations or fantasies. This is why certain people like certain types of games better than others. The battle lines of good against evil lurk in the background of many space games, wherein evil, menacing invaders are bent on the destruction of Earth. It becomes the player's goal to protect the Earth as long as possible while scoring the most points for killing aliens. Other appealing goals range from accumulating the most treasure while exploring a dangerous cavern, to escaping from a crumbling building before it collapses or your food runs out.

Computer game fantasies derive some of their appeal from the emotional needs they satisfy. Different fantasies appeal to different people. Sometimes the fantasy is simply an adolescent emotional release as in Food Fight, where you battle piethrowing chefs with tables full of messy food. The fantasy of destroying objects during a game appeals to others. It can take the form of popping balloons by bouncing a clown off a teeter-totter, as in Clowns and Balloons, or breaking out bricks in a wall, as in Breakout. In each case, the partially-destroyed wall or rows of balloons presents a visually compelling goal and a graphic scorekeeping device as well.

Goals in most games imply an end point, either when the goal is reached or when you fail, It is often important to make sure the game doesn't just go on and on forever. Limits should be set. Sometimes these take the form of time limits or constantly diminishing amounts of ammunition, balls, or ships. The most widespread limiting factor, at least on home computers, is speeding up the game. it is also the most abused. A game tempo, where it is neither humanly nor mechanically possible to withstand the onslaught of the computer's forces, cheats the player.

For a game to be considered challenging, it should have a goal where the outcome is uncertain. If the player is certain to reach the goal or certain not to reach it, the game is unlikely to present a challenge, and the player will lose interest. It is very easy to introduce randomness into the game either by hiding important information or by introducing random variables that draw the player toward disaster. Be careful not to overdo this, since a totally random game lacks a skill factor. Players quickly discover that they have no control over the outcome.

One of the more important design elements in any game is a logical set of rules. The rules can be extremely simple or utterly complex, but they must make sense. Since the game must follow its theme, any rules or variations should stem directly from that theme. It is pointless to throw in game elements that simply don't belong just because you think that confusing the player would make the game more difficult. For instance, Donkey Kong, one of the best jumping, climbing arcade games, doesn't require the player to shoot everything in sight, just avoid obstacles to reach the goal. Similarly, a tough, shoot-'em-up game like Galaxian keeps its fluid alien attack uncluttered by distracting game elements.

A game like Galaxian is considered asymmetric. It is not a balanced game because both the player and the computer's alien fleet are unbalanced in strength. Yet the differences between the advantages and disadvantages of the two opponents are too similar to build triangular relationships that make an arcade game more interesting.

The triangular relationship is one in which each opponent can defeat one other or be defeated by the third. The relationship is often used in many games to lure the player into a trap by sacrificing a weakly-armed player. Battlezone is a good example. The computer maneuvers the saucer to entice the human into a poor position against the tank. Time Pilot lures you into a poor position more subtly by placing the bonus parachute directly in line with the incoming enemy fighter pack. Other games use the bonus to distract you. Sky Blazer nearly always drops a fuel canister just at the time that your target appears, and bomb targets come up in Xevious just when you are engaged in a heavy fire fight with the alien armada.

A variable difficulty level is often used to alter the game's level of play. These levels, often with ego-satisfying names like Star Commander or Pilot, can be set by the player. Many games are designed to become harder the further you progress. The increasing skill level requirement presents an added challenge, while preventing the player from growing complacent. Often the technique is to speed up the game or place additional enemy craft into battle. The player is required to play faster and better, honing his reflexes during the process. Another variation allows less time to complete your objective as the difficulty increases.

Care must be taken so that the game's level of difficulty progresses evenly from beginner to expert level, Players' scores should reflect a steady improvement in what is known as a positive monotonic curve. A game with a relatively flat curve is hard to learn, while a sharp jump means that there is some trick required to master the level. Games that don't have a positive monotonic curve frustrate players because they fail to provide reasonable opportunities to better one's score.

Any good game should offer a reward for reaching increasingly. difficult levels of play. Often, bonus points, extra balls, ships, or more ammunition are rewarded for exceeding score thresholds. It is important that the rewards for winning outweigh the disappointment of losing. A player's ego is involved. A person wants to beat a challenging game, not be humiliated each time he loses.

The ideal arcade game should foster the illusion of winnability at all levels of play. One important factor is a clean and simple game design. Too much detail or too many rules may intimidate the player. If a player believes that his failure was caused by a flaw in an overly complex game or by the controls, he will consider the game unfair and quit. On the other hand, if a player perceives failure to be attributed to correctable errors on his part, then he believes the game to be winnable and will play repeatedly to master the game. It's as if the player teases himself to play one more time.

Appealing to a player's curiosity effectively keeps a game interesting. While novelty is sometimes a crucial factor in the original purchase, if the game has little depth, it becomes repetitious and boring. One method that appeals to many game designers is to have the game progress to slightly different scenarios. Some games change the opposition, while others vary the scenery; some do both. The player has to excel if he is to satisfy his curiosity. Games like Threshold, which progresses through twenty-four sets of alien spacecraft, or Vanguard, in which both the scenery and alien craft changes, offer strong curiosity incentives.

These spurs to a player's interest in the game are called "Perks." They are most important just when the player thinks he has the game figured out. Perks must be carefully timed so that the player does not give up on the game because not enough happens soon enough or because everything the game has to offer has been seen too soon. The most common perk is an extra life. Consider these coin-op video games: Pac Man, Donkey Kong, Dig Dug, Joust, Marto Bros., and Tempest. These games have multiple screens. The different screens by themselves are a perk, but what these games have in common is the time at which an extra life is rewarded. The extra lives generally come to the average player at some point in his third screen. This is hardly coincidental. The screens are scheduled at a specific rate, somewhat dependent on the player's skill. The extra life on the third screen comes in just before the average player might become exasperated and so not put in another quarter. The novice player is usually out of lives at this point, too.

Some games use cartoon intermissions to perk up the game. The player's interest is renewed with each cartoon. For many players, seeing the next cartoon becomes a personal goal. Placing hidden features not even hinted at in the rules is another clever perk. These embellishments are left for the experienced player to discover. They can even brighten up the earlier levels of a game which has become dull for the expert. For example, in the coin-op version of Star Wars, players hear the voice of Obi-Wan Kenobi admonishing them to use the Force. Nothing in the game instructions tell them what to do. Only by experimentation will a player realize that he must fly through the trench without firing a shot to receive a substantial point bonus upon reaching the exhaust port. The high score feature can also be considered as a "perk." While it doesn't renew interest within the game, it is important because

it can renew interest in playing even a mediocre game again. The high score itself presents a personal goal to reach, whether it be to beat your own high score or someone else's.

Sometimes varying the player's emotional response during the game serves as a "perk." Tension can be relieved during a tense shoot-'em-up with occasional comic relief, while cute games sometimes need touching moments. Remember games as entertainment need contrast. In sum, the varieties of perks are endless, but their objective is the same: renew interest in the game before the player becomes tired of it.

A game's controllability is one of the more important considerations in design. It is sometimes referred to as human engineering. Designers usually choose between keyboard and paddle/joystick control. While eye/hand coordination is more effective with paddles or joysticks, programmers attempting to create games with too many control functions will opt for a keyboard control system. At times, they produce a game that requires nine or ten keyboard controls which, unfortunately, only a pianist can operate. Games whose controls require considerable time to master often prove frustrating to play.

The ability to accurately control the action is crucial in the I design. If the screen does not respond immediately to the player's input, the player may end up feeling out of sync and become frustrated with the game. The programmer must insure through careful design that the game responds properly to the player. The player shouldn't believe, for example, that the computer made a wrong turn for him in a maze game.

Apparently, Atari owners like games which pit them against a competitive computer opponent. In several multi-player games, groups of two or more simultaneously compete against each other. Most of these contests are sports or card games involving two to four players. The cooperative game is rarely seen, except in cases where the computer competitor is much too skillful. The arcade game Ripoff involves a computer opponent that is more than a match for two players playing simultaneously. The battle is so fast and fierce that the teammate's ship has to be protected from his partner's bullets. The home computer version of Wizard of Wor offers a choice of competitive or cooperative play. It is a tough game that really needs cooperative play if the more advanced game levels are to be reached, but cooperation in this game is by agreement, not by mutual invulnerablity. Your partner is even worth 1000 points if you mistakenly blunder. It is quite interesting to watch cooperation turn into a fiercely competitive game after one player inadvertently walks into the other player's line of fire.

So far, we have discussed theory and generalizations that should increase a game's playability and appeal to the public. Concrete examples of the more popular games should give you a much more solid foundation for your own designs.

Example Arcade Games

Space Invaders was the first really popular arcade game. The object is to defend your turf against an alien horde of ferocious invaders who attack your castles and gun bases with a barrage of undulating bullets. It is actually a timed game, since you only have a limited period to destroy the entire attacking wave before they descend to the ground in marching formations and overrun your lone gun base.
The elimination of each alien acts as a visual scorekeeping device. You can never win, only survive as long as possible (thus getting the maximum playtime for your quarter). Elimination of each attacking wave, however, is an intermediate goal that staves off your inevitable doom. Each successive level becomes more difficult since the aliens, which begin their attack increasingly closer to Earth each round, limit the amount of time that you have to destroy them. Their constant approach to your mobile gun base decreases the reaction time needed to avoid enemy fire.

Shoot-'em-up games like Sneakers, Galaxian, Threshold, and Galaga are actually spin-offs of the Space Invaders theme. Whether they are set in space or on the ground, each has a variety of targets bent on your destruction. The targets or attackers are no longer static. Either they appear to dodge your fire, or they resort to kamikaze-type attacks.

The strong appeal of these types of games is based on curiosity and game depth. You are inspired to do better with each game just to see what the attackers are going to look like in the next level and what their tactics might be. The design goal is variety, with each successive level slightly harder than the last. Although most offer an unlimited number of bullets, Threshold controls rapid, random, and wasteful firing by overheating your lasers. Thus, your firing must be more accurate and paced during the game.

The popularity of Pac Man can be attributed to the game's design. First, it satisfies the fantasy concept of a person's childhood dreams. As children, we dreamt that we were being chased by evil monsters or ghosts, and we felt powerless to stop them. We

wished that there were some way to turn the tables, if only for a few moments. Pac Man's four energy dots fulfill that fantasy. The game also offers the visual feedback of the number of remaining dots to be eaten at each level. And since clearing each individual level is an immediate goal, even beginners believe a level can be cleared. Because Pac Man is a game of consumption rather than one of destruction, it appeals to players of both sexes. The game becomes a learning experience for the more advanced player, since the ghosts follow a discernible, non-random pattern. A player is eventually able to predict their movements and, consequently, to develop a technique to clear all of the dots on a particular level. The long term goal is survival and the highest score. The game is designed so that you gain more pleasure as you get better. Thus, players are willing to devote the time and money to master the game.

Scrolling games, such as Super Cobra, Vanguard, and Tail of Beta Lyrae, wherein your ship travels over a multi-screen world, benefit strongly from player curiosity and visual variety. Vanguard, a shoot-'em-up game in which a variety of enemy vessels and creatures attack your ship, has an extremely long sinuous tunnel with various types of chambers. The game has so many sections, combined with scrolling directions which change from horizontal to diagonal to vertical, that it is like playing many different arcade games at once. The player is given the option several times during the game to enter battle with a time-limited, energized spacecraft equipped for ramming the enemy, or merely four plain, old directional lasers. A map displayed in the lower corner informs the player of his progress. The curiosity factor is so enticing in this game that thirty seconds are provided to lure you into inserting another quarter in order to continue from where you left off.

Super Cobra is a classic game wherein you fly a helicopter over scrolling alien terrain and through heavily fortified and obstacle-filled narrow tunnels. Initially, you have to survive ground-launched rockets and a few laser bases, but as the game progresses you must also contend with meteors and alien ships. Using either your bombs or lasers to clear the tunnels of protruding ground targets is crucial to your survival. Bombing accuracy is also important. If you don't replenish your fuel supply by hitting enough fuel depot targets, your game will soon be over.

Pole Position, a highly competitive game, appeals to many players because it mixes just the right amount of fantasy with reality. It fulfills the fantasy of being a race car driver without the inherent danger. Crashes are never fatal and do not end the game. The goal of the game is to qualify for and complete the race. In a sense, it is a very realistic simulation requiring shifting gears and precise steering on a scrolling roadway. The player has a three-dimensional view of the course and his car, as if he were following it from fifty feet behind, a sort of out-of-body effect.

Joust immediately comes to mind when discussing a pure fantasy game that traces its roots to the glamorous days of medieval chivalry. Instead of presenting two knights in shining armor dueling on horseback, Joust allows the player to fly his ostrich-like mount to do battle in midair. The player does not shoot his opponents but defeats them by ramming his mount and lance into theirs, sometimes delicately, sometimes violently. The higher mount always wins. The player gains the excitement of physical contact without a bloody nose.
The game constantly forces the player into action. He must keep hitting the action button to make his mount fly. When the player takes a short rest between screens, his

surrogate also rests and does not continue to fly along aimlessly as it might in other games. Two players can play simultaneously but are not forced into partnership. The Lava Troll on the bottom of the screen is an additional menace both to the player and his enemies. It attempts to grab at anything close enough and drag it into the lava. This sometimes works to the player's advantage, since the lava can imprison an enemy and make it easier to destroy. A more formidable enemy, the pterodactyl that appears on higher levels, requires the player to discover a way to defeat it. As an added perk, every fifth screen is a bonus level where the player need not fight anyone but simply pick up the eggs for additional points. Many times this earns the player an extra life or at least a temporary rest from the game's pace. Physical contact, originality, immediate player involvement, and monster interaction are key parts of this game's success.
Some of the most clever games can be classed as novelty games. These are often "cute" games involving human or animal characters with which the player can identify. These novelty games either follow the theme of rescuing someone, or require the player to develop good manual dexterity and precise timing skills in order to avoid catastrophe or the demise of the hero.

The rescue theme appears in games like Donkey Kong, Donkey Kong Junior, Fantasy, and the Adventures of Roby Roto. In many cases an actual rescue doesn't take place, but the theme carries the player from one portion of the game to the next. In both Donkey Kong and Fantasy, the girl is whisked away to the next screen just before the player reaches her. The objective isn't the rescue but to overcome the obstacles barring your way. Learning the patterns and precise timing through repeated play hones the player's skill.

Although playing the hero is rare in these games, two games have followed this theme: The Adventures of Roby Roto in the arcade, and Choplifter on most microcomputers. The latter is probably the purest in theme of the two. The rescue of sixty-four hostages is the one and only goal. Success is measured in the number of hostages rescued. The fact that the player may have destroyed twenty-seven enemy tanks and planes during the mission adds nothing to the score. Thus, while details

like the hostage's waving builds empathy for the hostages, the appeal is simply the ego-satisfying role of playing the hero.

In the final group of novelty games, the player must avoid the calamity of losing a life. The goals and obstacles in these games differ widely. Crazy Climber requires the player to scale a building while windows close to block the path, and angry tenants, attempting to knock the climber off the building, drop flower pots on his head. Frogger has the player brave traffic in a test of precise timing skills. And playing Tarzan in jungle Hunt requires dexterity and timing skills to swing from vine to vine like a trapeze artist, or risk death in the fall. In each of these games the cuteness is what first attracts the audience, but it is the development of the player's timing skills and game depth that keeps him playing. Again, the concept is variety, along with increasing levels of difficulty.

Arcade games have that indefinable ability to make you feel that your losing is just a fluke, and that if you play just one more time, you'll beat it. If you can design a game that is fun and exciting to play, and has that added quality, then you have designed an addictive game, and wealth beyond your wildest dreams may be yours.

What Can Go Wrong

The best piece of advice that we can give a game programmer is to carefully plan out your game before you begin programming it. First decide what results you want and work backwards to figure out what you have to do to get them. If it doesn't work, change your concept or goal until you get something you are satisfied with. Make sure the game follows real-world physical principle, so that it feels right on a gut level. For example, objects smash or bounce when they fall from any height.

Many novice programmers try to get something up on the screen immediately. Actually, there is nothing wrong with this technique. After all, it does give you some encouragement to continue. However, most develop their games on a piecemeal basis, adding something because it looks good or because they need more action. The result is that they soon run out of players or characters and are forced to do a very painful rewrite.

Everyone prefers to organize his game differently. Some, like me, prefer the tight- structured approach of a flowchart; others, like my partner, just write down a rough outline of the order of events in the game. Whichever approach you prefer, we strongly recommend that you develop many of your frequently used routines as independent subroutines. This approach simplifies the logic of the main code loop.

We carefully planned all of the games in this book before we wrote them. This means that we considered where items like screen memory, player-missile memory, the character set, and the actual game code were placed in memory. We roughly flowcharted the game's main logic loop. We then wrote the code in small chunks, but in such manner that it always ran, or at least was supposed to run.

The first priority was to draw the playfield. This generally means that we had to get the display list right and move the character set data into the correct section of memory. It may sound like a piece of cake, but some terrible things can go wrong. Sometimes the display list is too long because you forgot that the first LMS instruction is one of the mode lines. The screen rolls or goes wacko. Maybe you let the display list inadvertently cross a 1K boundary, or allowed screen memory to cross a 4K boundary in the middle of a mode line. Each of these mistakes can cause the screen to behave erratically. If you do get a stable display, and it isn't the one you specified, perhaps you forgot to tell ANTIC where either your display list or your RAM character set resides. It is possible that you didn't place your character set on a 1K boundary, or on a 1/2K boundary if you are in GR. 1 or GR. 2. The fastest method to troubleshoot the problem in Assembly language is to enter the monitor and look at the intended areas for the display list and character set and to see if they are actually there. It is quite possible your memory move routine is faulty.

The next step is usually to initialize the starting positions of your players. Sometimes they just don't appear, so you immediately check to see if the player shape is actually in the proper 256 bytes of player-missile memory. You should also check that the PMBASE is on a 2K boundary for singleline resolution players (1K boundary for double-line resolution), and that you told ANTIC where that is. If that isn't the fault, some programmers make the mistake of trying to read a player's horizontal position by looking at the ANTIC horizontal position register. You may be able to write a horizontal position to the hardware location, but you read collisions from these same hardware locations. If you are going to increment a player's horizontal position, you will need to update a RAM location before writing the value into the hardware register. If this doesn't appear to be the problem, there are two other possibilities. First, you may have forgotten to turn on player-missile graphics switches. But probably the most frequent mistake is to forget to set the player's shadow color register. If it isn't set, it defaults to the background color, blends in with the background and disappears. Always use the shadow color register to change or set color registers, or the change may only last one television frame because the hardware registers are updated every VBlank. The only exception to this rule is when you use a Display List Interrupt to change colors midscreen.

Display List and Vertical Blank Interrupts can sometimes cause unforeseen problems. Inexperience is usually the culprit. The first thing to remember is that you must have a program to interrupt from. Since it is easy to write a simple game entirely in Deferred Vertical Blank, the main loop can be as short as FOREVER JMP FOREVER. The machine will hang up if you don't have somewhere to jump back to at the end of the VBI. On the other hand, if the code is too long, it will be interrupted by the next VBlank before it finishes. Unexpected results, such as a garbaged screen, may occur. The most common problems with Display List Interrupts occur when you forget to save your registers before entering the routine or forget to restore them before exiting. A mistake here will lock up the machine. The other problem is when the interrupt seems to occur on the wrong mode line. Remember that the interrupt has to be set on the mode line before the interrupt is to occur.

BASIC programmers who use Machine language subroutines sometimes encounter strange problems. If you are going to incorporate a VBlank routine, make sure you clear the decimal mode at the beginning. This is especially important if your program uses decimal arithmetic internally. Another problem occurs when you pull the incorrect number of bytes off the stack. This can lock up the machine on the return if the return address on the stack is incorrect. Unfortunately, these subroutines are very difficult to test in Assembly language without constructing a setup routine to simulate the stack environment.

BASIC is usually very forgiving, so it is unlikely that you will lock up the machine if you aren't using Machine language subroutines. One of the most common display mistakes is forgetting to set a graphics mode after you lower RAMTOP to reserve space for your RAM character set and player-missile graphics. If you forget, you will still have a Graphics 0 display just below the old RAMTOP. The new graphics call will actually place the screen below RAMTOP.

We hope we have suggested adequate solutions for the most common errors that might occur in your games. We have learned many of these by bitter and frustrating experience. We will admit that these weren't the only errors that we encountered when programming the code in this book. However, most of the others were logic problems that one of us alone couldn't trace. For example, when I was programming the maze game, I programmed the manual mode for the joystick-controlled letter first. It worked fine, but became buggy when I added the auto mode. Sometimes the letter would behave properly, yet at other times it would escape the maze walls. I single-stepped the code repeatedly and the legal move flags were always set correctly. Days went by and I couldn't find any cause for the anomaly. My partner discovered that it only happened just after the stick was returned to neutral. While legal moves were reset at the center of each maze block, moving the stick was required to close pathways other than those in the direction of movement or in reverse. Nothing was done in the neutral position because the letter was stopped in the non-auto mode. Since I had neglected to close gates when I was in the neutral position, the joystickcontrolled letter was now traveling in some direction automatically, so it became possible to give it a new direction command while it was between blocks. For example, if it had just passed a block that said it could go right, pushing right from the neutral position would command it to go right even though there was a wall there. In short, I forgot to close gates when I was in the auto mode, because I assumed they were set by one of the four non-neutral positions. This is a fine example of misguided thinking.

In closing, we hope that we have provided you with enough programming techniques and game theory to create your own arcade games. Remember that originality, persistence and attention to detail are the keys to success in this industry. We hope some of our readers will join the ranks of successful Atari game designers. If you take the easy way out and program a quick game, the results will show in mediocrity.

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix A: Useful Peeks and Pokes

MEMORY CONFIGURATION

10,11 ($A,$B) Start Vector-Disk Based Software-(DOSVEC)

These locations hold the start vector or run the address for disk based binary load software. This is also the address BASIC jumps to when you call up DOS.

12,13 ($C,$D) Initialization Address for Disk Boot (DOSINI)

These locations hold the initialization address for the disk handler.

14,15 ($E,$F) Upper Limit of BASIC Program (APPMHI)

These locations contain the memory high limit for your BASIC program. Memory above that is used for screen display.

88,89 ($58,$59) Screen Memory Address (SAVMSC)

These addresses contain the lowest address of screen memory.

106 ($6A) Top of RAM Address-most significant byte (RAMTOP)

Gives the total number of pages (256 bytes) available. Peek (106)/4 gives the number of 1K blocks available.

741,742 ($2E5,$2E6) Free Memory High Address (MEMTOP)

This address is the highest free location in RAM for program and data. This value is updated when you press RESET, when you change GRAPHICS mode, or when a channel (IOCB) is OPENed to the display. The display list starts at the next byte above MEMTOP.

743,744 ($2E7,$2E8) Free Memory Low Address (MEMLO)

This is the address of the first free location of RAM for program use. This value which is normally 1792 ($700) is updated when DOS is present.

DISPLAY SCREEN

77 ($4D) Attract Mode On/Off (ATRACT)

Setting this location to 0 disables the attract mode, This happens automatically whenever a key is pressed. Normally this location is incremented every 4 seconds until the value reaches 127 ($7F), then set to 254 ($FE) until the attract mode is terminated. The attract mode protects the television screen by rotating the colors when the Atari is idle.

87 ($57) Display Mode (DINDEX)

This location contains the current BASIC graphics mode (0-11) in non-XL machines and (0-15) in XL machines. This can be used to fool the OS into thinking it is in a different graphics mode.

90 ($5A) Starting Graphics Cursor Row (OLDROW)

Used to determine the starting row for DRAWTO and XIO 18 (FILL) command.

91,92 ($5B,$5C) Starting Graphics Cursor Column (OLDCOL)

These locations are used by DRAWTO and XIO 18 (FILL) commands to determine the starting column of the DRAW or FILL.

96 ($60) Ending Graphics Cursor Row (NEWROW)

Row to which DRAWTO and FILL will go.

97,98 ($61,$62) Ending Graphics Cursor Column (NEWCOL)

Column to which DRAWTO and FILL will go.

660,661 ($294,$295) Split-screen Text Memory Address (TXTMSC)

Address of upper left corner of the split-screen text window.

708-712 ($2C4-$2C8) Playfield Color Registers (COLOR0-COLOR4)

Each of these locations determines the color for the various playfields. You can change these registers from BASIC via either a POKE or the SETCOLOR command.

752 ($2FO) Cursor Inhibit (CRSINH)

Zero turns the cursor on. Any other number turns the cursor off.

54276 ($D404) Horizontal Fine Scroll Register (HSCROL)

When horizontal fine scroll is enabled by setting bit 4 in the LMS instruction in the display list, POKEing this location with a value from zero to 16 clock cycles (depending on graphics mode) will fine scroll the screen.

54277 ($D405) Vertical Fine Scroll Register (VSCROL)

POKEing a value from zero to 16 (depending on graphics mode) will fine scroll the screen one or more scan lines. Vertical fine scrolling can only be used if bit 5 is set in the LMS instruction in the display list.

54282 ($D40A) Wait for Horizontal Synchronization (WSYNC)

This location allows the OS to synchronize the VBI's or DLI's with the screen display. Simply accessing this location halts the CPU until horizontal sync occurs.

54283 ($D40B) Vertical Line Counter (VCOUNT)

This register keeps track of the current line number currently be drawn on the screen. PEEKing here returns the line count divided by two; ranging from zero to 130 ($82). It is used during Display List Interrupts to change colors or when working with kernels.

CHARACTER SETS

755 ($2F3) Character Mode Register (CHACT)

This location defaults to a value of 2. Zero means normal inverse characters, one is blank (invisible) inverse characters, three is solid inverse characters. Four to seven is the same as zero to three, except the display is printed upside down. For example; POKE 755,4 will invert the standard character set.

756 ($2F4) Character Base Register (CHBAS)

This high byte location is used to tell ANTIC where the character set is located. It normally defaults to 224 ($E0) for upper case characters and numbers. Lower case and graphics characters can be selected in graphics modes I and 2 by POKEing CHBAS with 226 ($E2).

763 ($2FB) Last ATASCII Character or Plot Point (ATACHR)

Returns the last ATASCII character read or written, or the value of a graphics point. The FILL and DRAW commands use this location for the color of the line drawn.

DISPLAY LISTS

512,513 ($200,$201) Display List interrupt Vector (VDSLST)

These locations store the address of the instructions to be executed during a display list interrupt.

559 ($22F) DMA Control Register (SDMCTL)

This location enables or disables direct memory access by ANTIC. The normal default value is 34 ($22) which enables DMA for fetching normal playfield display data. You can turn the display off by POKEing a zero here.

560,561 ($230,$231) Display List Address (SDLSTL)

This is the starting address of the display list.

54286 ($D40E) Non-maskable Interrupt Enable (NMIEN)

This location is used to enable or disable vertical blank and display list interrupts. It is set to 64 ($40) upon powerup to enable VBI's. A value of 128 ($80) enables DLI's and 192 ($CO) enables both. A zero disables both.

KEYBOARD I/O

16 ($10) POKEY Interrupts (POKMSK)

This location enables and disables POKEY functions such as its timers, serial input/output data ready, and the BREAK key interrupt. The interrupt key can be disabled by POKEing a 112 to this shadowed location, and at 53774 ($D20E). It is better for the user to write his own routine for the BREAK key is reenabled whenever RESET is pressed or when PRINT or OPEN statements address the screen. You can store the location of your own interrupt routine at locations 566, 567 ($236,$237).

17 ($11) BREAK Key Flag (BRKKEY)

A zero means the BREAK key is pressed; any other number means it isn't.

764 ($2FC) Keyboard Character (CH)

This returns the value of the last key pressed. This value is neither internal or ATASCII but "raw" keyboard matrix code. A 255 ($FF) POKEd here clears it.

53279 ($D01F) Console Keys (CONSOL)

Used to determine if one of the three yellow console buttons have been pressed. It is best to clear it first with the value of eight to ensure an accurate reading. A value of 6 indicates the START key has been pressed, a 5 the SELECT key, and a 3 the OPTION key. The location reads a 7 when no CONSOLE keys are pressed, and a 0 when all are pressed simultaneously.

JOYSTICK/PADDLE I/O

624-631 ($270-$277) Paddle Game Controller (PADDL0-PADDL7)

A number between zero and 228 ($E4) is returned in these locations for each of the seven paddle potentiometers.

632-635 ($278-$27B) joystick Controller Port (STICK0-4)

These locations return one of nine possible joystick positions for each stick. These values range from 5-15 (35$F). These values are 15 when the joystick is centered or in the neutral position.

636-643 ($27C-$283) Paddle Trigger (PTRIG0-PTRIG7)

Used to determine if the button on each paddle is pressed. A zero is returned if it is pressed and a one is returned when it isn't.

644-647 ($284-$287) joystick Trigger (STRIG0-STRIG3)

These locations are used to indicate whether the joystick button is pressed. A zero is returned if it is pressed, and a one is returned when it isn't.

PLAYER-MISSILE GRAPHICS

623 ($26F) Player/Playfield Priorities (GPRIOR)

Priority options select which screen objects will be "in front" of the others. Players have precedence over all playfields when this location is one. A value of 4 gives the playfields priority over all of the players, and values 2 and 8 allow only some of the players to have priority over some of the players. In addition, the value 16 ($10) allows the four missiles to be combined into a fifth player, and a value of 32 ($20) allows players to be overlapped to produce a third color. This location is also used to enable the three GTIA modes.

704-707 ($2C0-$2C3) Player-missile Color Registers (COLPM0-COLPM3)

Each of these locations determines the color of a player and its corresponding missile.

53248-53251 ($D000-$D003) Player Horizontal Position Registers (HPOSP0-HPOSP3)

These write only registers determine the horizontal positions of each of the four players. Values range from 0-277 ($0- $116).

53248-53251 (D000-$ D003) Missile to Playfield Collision Registers (MOPF-M3PF)

These read only registers tell you which playfield the missiles have collided with. The values are as follows; Playfield #0- 1, Playfield #1 -2, Playfield #2 -4, and Playfield #3- 8.

53252-53255 ($D004-$DO07) Player to Playfield Collision Registers (P0PF-P3PF)

These read only registers tell you which playfield the players have collided with. The values are the same as with the missiles above.

53252-53255 ($D004-$D007) Missile Horizontal Position Registers (HPOSM0-HPOSM3)

These write only locations determine the horizontal positions of each of the missiles. Values range from 0-277 ($0-$116)

53256-53259 ($D008-$DOOB) Player Width Registers (SIZE0-SIZE3)

These write only locations control the widths of the players. Values 0 and 2 produce normal width players, 1 double width players, and 3 quadruple width players.

53256-53259 ($D008-$D00B) Missile to Player Collision Registers (M0PL-M3PL)

These read only registers determine collisions between missiles and players. The values are as follows; with player #0 - 1, player #1 -2, player #2 -4, and with player #3 -8.

53260 ($D00C) Missile Width Register (SIZEM)

This write only location controls the magnification of all four missiles. While a value of 0 or 2 displays all of the missiles at normal width, 1 displays all of the missiles at double width, and 3 displays all of the missiles at quadruple width, missiles widths can be set individually. Each missile is controlled in two bit pairs starting with the lowest. The bit values for the 0th missile is the same as above.

53260-53263 ($D00C-$D00F) Player to Player Collisions (P0PL-P3PL)

These read only registers return non-zero values when players collide. In general, the registers contain a 1 after a collision with player#0, a 2 after one with player #l, a 4 after one with player #2, and as 8 after one with player #3.

53277 ($DOID) Graphics Control Register (GRACTL)

This, when used in conjunction with DMACTL ($22F) enables player-missile graphics. A value of 2 enables player DMA only, a value of 1 enables missile DMA only, and a value of 3 enables both.

53278 ($D01E) Clear Player-missile Collision Registers (HITCLR)

You POKE this location with any number to clear the collision registers. This is normally done at the end of VBLANK after you have tested for all collisions.

54279 ($D407) Player-missile Base Registers (PMBASE)

This location contains the high byte starting address of your player-missile area.

SOUND

65 ($41) Input/Output Noise Control (SOUNDR)

Disk read/write operations can be silenced by POKEing a zero to this location.

53760,53762,53764,53766 ($D200,$D202,$D204,$D206) Audio Channel Frequency (AUDFI-3)

These locations control the frequency of the sound channels. The value N is used in the divide by N circuit. Thus the notes or tone becomes lower when N becomes larger. N can be in the range 1-256.

53761,53763,53765,53767 ($D201,$D203,$D205,$D207) Audio Channel Control (AUDC1-4)

These registers set the volume and distortion levels.

53768 ($D208) Audio Control Register (AUDCTL)

This location allows two channels to be joined for greater frequency range, and allows the user to vary the poly-counters which control noise, or to switch the clock base from 64KHz to 15KHz for a change in frequency range. To properly initialize the POKEY sound capabilities POKE AUDCTL with zero and POKE 53775,3 ($D02F). This is equivalent in BASIC of SOUND 0,0,0,0.

MISCELLANEOUS

18,19,20 ($12,$13,$14) Internal Realtime Clock (RTCLOK)

Locations 20 increments every VBLANK interrupt or 1/60 second until it reaches 255 ($FF). Location 19 is then incremented by one and 20 is reset to zero. Likewise, location 18 is incremented when 19 reaches 255. This happens every 18.2 minutes.

53770 ($D204) Random Number (RANDOM)

This location produces a random number from 0-255 ($0-$FF) when read.

58454 ($E456) Central Input/Output (CIO) Utility Entry (CIOV)

This is actually a subroutine in the OS ROM that is used to pass I/O operations to the correct device driver. Once parameters are set in the correct IOCB's you jump here to use them. While BASIC only supports one byte at-a-time I/O (GET and PUT), addressing CIOV directly allows the user to input or output a buffer of characters at a time.

58460 ($E45C) Set Vertical Blank (SETVBK)

This OS routine, which sets system timers during the VBLANK routine, insures that both bytes of the vector addressed will be updated while VBLANK is enabled.

58466 ($E462) Exit Vertical Blank (XITVBK)

This OS subroutine is used to restore the computer to its pre-interrupt state upon exiting the user's VBLANK routine. The computer can then resume normal processing.

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix B: ATASCII Character Set

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix C: Assembler Comparisons

SYN ASSEMBLER F-S MACRO	ATARI ASSEMBLER	MAC 65	ATARI MACRO	EASTERN HOUSE	MEANING
.OR $600	* = $600	.OR = $600 * = $600	ORG $600	.BA $600	Define Program Origin
.EQ	=	.EQ or =	EQU or =	.DE	Define equates
.BS 5	* = * + 5	.DS 5	DS 5	.DS 5	Reserves space for data
.HS FFFFFF	.BYTE $FF,$FF,$FF	.BYTE $FF,$FF,$FF	DB $FF,$FF,$FF	.BY $FF,$FF,$FF	Define Hexidecimal data
.DA #20,#40	.BYTE 20,40	.BYTE 20,40	DB 20,40	.BY 20 40	Defines bytes
.DA $E474 .DA START	WORD $E474 WORD START	WORD $E474 WORD START	DW $E474 DW STAR	.SI $E474 .SI START	Define a two byte word low byte, high byte order
.AT "HELLO" (NOTE)	.BYTE Using internal HEX values	.SBYTE "HELLO"	DB Using internal HEX values	.BY Using internal HEX values	Define string using internal character values
.AS "HELLO"	.BYTE "HELLO"	.BYTE "HELLO"	DB "HELLO"	.BY "HELLO"	Define string using ASCII values
# LABEL	# LABEL & $FF	#< LABEL	# LOW LABEL	#L, LABEL	Returns Low byte (LDA LABEL)
/ LABEL	# LABEL / 256	#> LABEL	# HIGH LABEL	#H, LABEL	Returns High byte (LDA LABEL)
BGE LABEL	BCS LABEL	BCS LABEL	BCS LABEL	BCS LABEL	Branch if > = (After a compare)
BLT LABEL	BCC LABEL	BCC LABEL	BCC LABEL	BCC LABEL	Branch it < (After a

					compare)
.IN "D:PARTZ"	----	.INCLUDE #D:PARTZ"	INCLUDE D:PARTZ	.FI "DI:PARTZ"	Include a file for Assembly

NOTE: The F-S Macro Assembler uses a .AS ^ "HELLO" where the ^ is a Shift *sign.

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix D: Binary File Autorun

Since assembly language files (object code) generated by SynAssembler and many other assemblers will not run automatically from the DOS menu without specifying the run address, we have provided a BASIC language utility that will fix the problem. Files run automatically from Atari DOS if the starting address in low byte, high byte order is appended to the file, DOS reads these two values into locations $2E0 and $2E1 (736, 737). Upon completion of the binary load, control is normally passed back to the DOS menu. However, if there is an address in these locations, the computer will jump to that location.

Our utility will append your file automatically, a function that used to be provided with the /A command in the old DOS 1.0 menu. The program only asks for the name of the file and its run address in decimal. Assuming the file in on the disk in the disk drive, the utility will open the file, append the run address, then close it. It will now run automatically when you load it from the DOS menu using the L command.

Download AUTORUN.BAS (Saved BASIC)
Download / View AUTORUN.LST (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix E: Source Code for Chapters 3 & 5

Character Set Move Routine

Download CHARMOVE.EXE (Executable program)
Download / View CHARMOVE.LST (Assembler listing)
Download / View CHARMOVE.S (Source file)

Load or Save Character Sets

Download / View CHSET.S (Source file)

Matrix Cycler

Download / View MCYCLER.S (Source file)

PM Clear Routine

Download PMCLEAR.EXE (Executable program)
Download / View PMCLEAR.LST (Assembler listing)
Download / View PMCLEAR.S (Source file)

Return to Table of Contents | Previous Chapter | Next Chapter

About The Authors

Jeffrey Stanton received a BME (1967) and a MSME (1969) from Rensselaer Polytechnic Institute. He worked as a control systems engineer and mechanical engineer for the aerospace industry in the early 1970's. His interest in computer game design sidetracked his career as a photographer and book illustrator in the late 1970's. In addition to writing several Apple arcade games and doing some occasional consulting, he is the author of Apple Graphics and Arcade Game Design, and one of the editor/reviewers for the books of Apple and Atari Computer Software. He currently divides his time between writing and reviewing software in the mornings, and operating a postcard stand on the Venice Beach boardwalk in the afternoons. He lives in Venice, California.

Dan Pinal, typical of many of the early computer hobbyists, is self educated. He was one of the first to own an Atari computer, and entered the micro-computer industry a year later. Dan consulted, taught and did game programming for two software houses at the peak of the game market in 1983. He has one Atari game currently on the market. Dan currently lives in Los Angeles, California.

Return to Table of Contents | Previous Chapter | Next Chapter

Atari Graphics And Arcade Game Design

Atari Graphics and Arcade Game Design is the most comprehensive book about designing arcade games on the market. More than just a book on designing games, it teaches all of the basic fundamentals of Atari graphics including scrolling, GTIA color, display lists, player-missile graphics, character animation, vertical blank and display list interrupts.

The book is understandable and informative to programmers on all levels ranging from intermediate BASIC to advanced Assembly language. Atari Graphics and Arcade Game Design addresses an audience of teenagers and young adults, so it presents graphics concepts simply through clear text, diagrams, and easy to follow examples. The book thoroughly flowcharts and explains the sometimes complex logic of the game code discussed. Subjects such as bomb drops, laser fire, millile and ship movement are covered in great detail as are the physics and mathematical concepts needed to make them work realistically. And useful tools such as character and player-missile editors are included as an added bonus.

Atari Graphics and Arcade Game Design teaches game design using advanced Atari graphics principles.As examples, it develops a wide variety of complete working games in both BASIC and Assembly language. The first two example games are first written in BASIC, then translated into Assembly language as a learning bridge between BASIC and the faster, more powerful Assembly language. A third BASIC game demonstrates that you can create playable moderate-speed games in BASIC with the help of machine language routines developed in this book. Finally, three commercial quality machine language arcade games - an alphabet maze, a scrolling game with ground laser bases and a programmable alien attack, and a two- player tank game with rotating turrets and blow-away walls-help advanced programmers pursue state-of-the-art game design.

ISBN 0-912003-05-7
$16.95 (U.S.A.)
Arrays, Inc.
THE BOOK DIVISION

Return to Table of Contents | Previous Chapter

Book Software Archive

If your browser tries to display the file rather than downloading it,
try pressing option (Mac) or control (PC) while you click the Download link.

Chapter 1
PAINTPOT.BAS: Color demonstration in GR. 7 (Saved BASIC) Download | Jump to text
PAINTPOT.LST: (Listed BASIC) Download / View
GR10DEMO.BAS: Draw a colorful box in perspective in GR. 10 (Saved BASIC) Download | Jump to text
GR10DEMO.LST: (Listed BASIC) Download / View
GTIATRIC.BAS: Shows 16 GTIA pixel colors in GR. 0 (Saved BASIC) Download | Jump to text
GTIATRIC.LST: (Listed BASIC) Download / View
Chapter 2
DLIDEMO1.BAS: Example of modifying the Display List (Saved BASIC) Download | Jump to text
DLIDEMO1.LST: (Listed BASIC) Download / View
DLIDEMO2.BAS: Example of rewriting the Display List (Saved BASIC) Download | Jump to text
DLIDEMO2.LST: (Listed BASIC) Download / View
DLIDEMO3.BAS: Custom screen with Display List Interrupts (Saved BASIC) Download | Jump to text
DLIDEMO3.LST: (Listed BASIC) Download / View
Chapter 3
PUMPKIN1.BAS: Character set graphics example in GR.0 (Saved BASIC) Download | Jump to text
PUMPKIN1.LST: (Listed BASIC) Download / View
PUMPKIN2.BAS: Character set graphics example in GR. 2 (Saved BASIC) Download | Jump to text
PUMPKIN2.LST: (Listed BASIC) Download / View
PUMPKIN3.BAS: Modify character set with machine language copy (Saved BASIC) Download | Jump to text
PUMPKIN3.LST: (Listed BASIC) Download / View
CHSETED.BAS: Character set editor (Saved BASIC) Download | Jump to text
CHSETED.LST: (Listed BASIC) Download / View
CHSETLOD.BAS: Load character set from file (Saved BASIC) Download | Jump to text
CHSETLOD.LST: (Listed BASIC) Download / View
ANTIC4.BAS: Example of Antic mode 4 screen (Saved BASIC) Download | Jump to text
ANTIC4.LST: (Listed BASIC) Download / View
ANIMDEMO.BAS: Character animation demo (Saved BASIC) Download | Jump to text
ANIMDEMO.LST: (Listed BASIC) Download / View
BUGDEMO.BAS: Animated bugs - character animation demo (Saved BASIC) Download | Jump to text
BUGDEMO.LST: (Listed BASIC) Download / View
BIRDDEMO.BAS: Animated birds - character animation demo (Saved BASIC) Download | Jump to text
BIRDDEMO.LST: (Listed BASIC) Download / View
Chapter 4
BREAKOUT.BAS: BASIC Breakout game (Saved BASIC) Download | Jump to text
BREAKOUT.LST: (Listed BASIC) Download / View
BREAKOT.EXE: ML Breakout game (Executable program) Download | Jump to text
BREAKOT.OBJ: (Object code) Download
BREAKOT.LST: (Assembler listing) Download / View
BREAKOT.S: (Source file) Download / View
BREAKOT.RAW: (As printed in book) Download / View
Chapter 5
PMEXAMP1.BAS: Player Missile example-moving a ship vertically (Saved BASIC) Download | Jump to text
PMEXAMP1.LST: (Listed BASIC) Download / View
PMEXAMP2.BAS: Player Missile example-priority demonstration (Saved BASIC) Download | Jump to text
PMEXAMP2.LST: (Listed BASIC) Download / View
PMINTB.EXE: Player Missile ML interface to BASIC (Executable program) Download | Jump to text
PMINTB.OBJ: (Object code) Download
PMINTB.LST: (Assembler listing) Download / View
PMINTB.S: (Source file) Download / View
PMINTB.RAW: (As printed in book) Download / View
TWOSHIP.BAS: Player Missile example-collision demonstration (Saved BASIC) Download | Jump to text
TWOSHIP.LST: (Listed BASIC) Download / View
SHOOT.BAS: Shoot Blocks game (Saved BASIC) Download | Jump to text
SHOOT.LST: (Listed BASIC) Download / View
SPACEWAR.EXE: Space War game (Executable program) Download | Jump to text
SPACEWAR.OBJ: (Object code) Download
SPACEWAR.LST: (Assembler listing) Download / View
SPACEWAR.S: (Source file) Download / View
SPACEWAR.RAW: (As printed in book) Download / View
PMEDITOR.BAS: Player Missile shape editor (Saved BASIC) Download | Jump to text
PMEDITOR.LST: (Listed BASIC) Download / View
PMDEMO.BAS: Player Missile-moving with BASIC strings demo (Saved BASIC) Download | Jump to text
PMDEMO.LST: (Listed BASIC) Download / View
Chapter 6
MULTI.EXE: Multicolored Player moves with joystick (Executable program) Download | Jump to text

MULTI.OBJ: (Object code) Download
MULTI.LST: (Assembler listing) Download / View
MULTI.S: (Source file) Download / View
MULTI.RAW: (As printed in book) Download / View
MIDSCRN.EXE: Change Player color midscreen using DLIs (Executable program) Download | Jump to text
MIDSCRN.OBJ: (Object code) Download
MIDSCRN.LST: (Assembler listing) Download / View
MIDSCRN.S: (Source file) Download / View
MIDSCRN.RAW: (As printed in book) Download / View
WAVES.EXE: Boat and Waves - Using DLIs to create animation (Executable program) Download | Jump to text
WAVES.OBJ: (Object code) Download
WAVES.LST: (Assembler listing) Download / View
WAVES.S: (Source file) Download / View
WAVES.RAW: (As printed in book) Download / View

Chapter 7

VERTSCR.BAS: Coarse vertical scrolling example (Saved BASIC) Download | Jump to text
VERTSCR.LST: (Listed BASIC) Download / View
HORIZSCR.BAS: Coarse horizontal scrolling example (Saved BASIC) Download | Jump to text
HORIZSCR.LST: (Listed BASIC) Download / View
8WAYTEST.EXE: 8-way scrolling - test case for modifying display list (Executable program) Download | Jump to text
8WAYTEST.OBJ: (Object code) Download
8WAYTEST.LST: (Assembler listing) Download / View
8WAYTEST.S: (Source file) Download / View
8WAYTEST.RAW: (As printed in book) Download / View
8WAY.EXE: 8-way scrolling (Executable program) Download | Jump to text
8WAY.OBJ: (Object code) Download
8WAY.LST: (Assembler listing) Download / View
8WAY.S: (Source file) Download / View
8WAY.RAW: (As printed in book) Download / View
SCROLL.EXE: Strike Force-a scrolling game (Executable program) Download | Jump to text
SCROLL.LST: (Assembler listing) Download / View
SCROLL.S: (Source file) Download / View

Chapter 8

RASTER.EXE: Raster graphics example with sound (Executable program) Download | Jump to text
RASTER.LST: (Assembler listing) Download / View
RASTER.S: (Source file) Download / View
RASTER.OBJ: (Object file) Download
RASTER.RAW: (As printed in book) Download / View

Chapter 9

ALPHMAZE.EXE: Alphabet Maze game (Executable program) Download | Jump to text
ALPHMAZE.OBJ: (Object code) Download
ALPHMAZE.LST: (Assembler listing) Download / View
ALPHMAZE.S: (Source file) Download / View
ALPHMAZE.RAW: (As printed in book) Download / View
TANK.EXE: Tank Battle game (Executable program) Download | Jump to text
TANK.LST: (Complete assembler listing) Download / View
TANKMAST.SRC: (Master source file) Download / View
TANK.EQU: (Source file) Download / View
TANKDATA.SRC: (Source file) Download / View
TANKSUBS.SRC: (Source file) Download / View
TANKVAR.SRC: (Source file) Download / View
TANKVBI.SRC: (Source file) Download / View
TANK.SRC: (Source file) Download / View
EQUATES: (Source file) Download / View

Appendix D

AUTORUN.BAS: Binary File Autorun (Saved BASIC) Download | Jump to text
AUTORUN.LST: (Listed BASIC) Download / View

Appendix E

CHARMOVE.EXE: Character Set move routine (Executable program) Download | Jump to text
CHARMOVE.LST: (Assembler listing) Download / View
CHARMOVE.S: (Source file) Download / View
CHSET.S: Load or save character sets (Source file) Download / View | Jump to text
MCYCLER.S: Matrix cycler (Source file) Download / View | Jump to text
PMCLEAR.EXE: Player Missile Clear routine (Executable program) Download | Jump to text
PMCLEAR.LST: (Assembler listing) Download / View
PMCLEAR.S: (Source file) Download / View

The Whole Enchilada

agagd.ZIP: All programs and source code from the book (298K!) (ZIP file) Download

---

Return to Table of Contents