PM animator

FOR ATARI HOME COMPUTERS

OWNER'S MANUAL AND TUTORIAL

DON'T ASK INC
COMPUTER SOFTWARE

2265 Westwood Blvd., Suite B-150, Los Angeles, CA 90064
(213) 477-4514

pm ANIMATOR

OWNER'S GUIDE/TUTORIAL

by

Roger Bush

and

Don't Ask Computer Software

DISCLAIMER AND LIMITED WARRANTY

This software product and the attached instructional materials are sold "AS IS" without warranty as to their eprformance. The entire risk as to the quality and performance of the computer software program is assumed by the user. The user, and not the manufacturer, distributor or retailer assumes the entire cost of all necessary service or repair to the computer software program. DO'T ASK COMPUTER SOFTWARE, Inc. shall have no liability or responsibility to the purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption in service, loss of service, loss of business and anticipatory profits or consequential damages resulting from the use or operation of the software, hardware, or documentation portions of this product.

However, to the original purchaser only, DON'T ASK COMPUTER SOFTWARE, Inc. warrants that the medium on which the program is recorded will be free from defects in materials and faulty workmanship under normal use and service for a period of ninety (90) days from the date of purchase. If a defect in the medium should occur during this period, the medium should be returned for repair or replacement to DON'T ASK or to its authorized dealer. After this ninety day period, media inoperable **for any reason** may be returned to DON'T ASK **only** along with $5.00 for prompt replacement.

The above warranties for goods are in lieu of all other express warranties and no implied warranties of merchantability and fitness for a particular purpose or any other warranty obligation on the part of DON'T ASK shall last longer than ninety (90) days. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

NOTICE

Don't Ask Computer Software, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice. DON'T ASK can have no responsibility for errors, whether factual or typographical, in this documentation.

TABLE OF CONTENTS

NOTICE OF NON-EXCLUSIVE LICENSE

Purchase of pm ANIMATOR grants to the purchaser a non-exclusive license to incorporate the TOTAL.LST package into his own programs without paying a licensing fee. Further, if the purchaser desires to include all or some of the TOTAL.LST subroutines into commercially marketed software, he may do so if he includes the following insertion on all packaging materials and manuals: "This product was produced using pm ANIMATOR, copyright 1983 by Don't Ask Computer Software, Inc., 2265 Westwood Blvd., L.A., CA 90064." No other compensation is required.

INTRODUCTION

What is pm ANIMATOR?

pm ANIMATOR is a powerful set of tools for the Atari computer that allows the straightforward incorporation of Player-Missile Graphics (PMG) into BASIC programs.

The Atari computer has the built-in ability to manipulate a series of up to eight complicated graphic objects (four "players" and four "missiles") as simply as most other computers can deal with a single line in a graphic display. The location of these players and missiles, their movement, and their ability to collide or pass in front of one another are all controlled by Atari hardware. PMG is a prime reason why the Atari computer has achieved such preeminence in computer games and graphics of all kinds.

There are four major components to the pm ANIMATOR package:

1) THE GRAFIX EDITOR AND FILE EDITOR

These two programs allow you to create the images you want to use in your programs.

The Grafix Editor is a multi-featured tool kit for the creation of player-missile graphic images. For sophisticated animated graphics, you use the editor to create a sequence of graphics frames – each one slightly different from the previous one just as in the successive frames of a movie – and to view the animation you have created. You can then easily store these animation frames as player-missile data files to be called up in your own programs.

The File Editor is a graphics spreadsheet that will allow you to manipulate your Grafix Editor files visually in order to create the sequences you want. You can combine smaller files into larger ones, or select individual frames from several different files to create a new file. The File Editor makes it easy to get the results you want.

2) TOTAL.LST

The highest quality graphics are seldom implemented from BASIC because BASIC is

too slow to rapidly manipulate the large amounts of data that make up elaborate graphic images, and therefore it produces slow, uneven motion, or even worse, long delays between different graphics displays. It you are an assembly language programmer, you can easily get around these problems. But if you are not, pm ANIMATOR gives you another way out.

Included in the package are some powerful machine-language subroutines that speed up the "slow spots" in implementing PMG such as loading in ASCII data, clearing out areas of memory, and vertical motion. Most of the necessary routines are built into a BASIC subroutine called TOTAL.LST. Even a special routine for multiplayer animation is included.

TOTAL.LST is a "LIST-format" BASIC file that you "ENTER" into your own program. You don't have to know any machine language to use the TOTAL.LST program; you access the routines through the USR statement of BASIC.

All the subroutines in the package are in the "LIST" format so that you can enter them directly into your own programs. All the subroutines are described in detail in this manual. The better you come to understand them, the more you will be able to make use of them.

3) DEMONSTRATION PROGRAMS

There are numerous Atari BASIC demonstration programs on the pm ANIMATOR disk. There is the main demo which uses everything the package has to offer, but there are also seven simpler demonstrations of various features and subroutines. Each of these is fully documented with REM statements in BASIC code, and you are encouraged to borrow as many techniques as you can from these demonstrations.

4) THE TUTORIAL

This manual serves as an instruction manual for the features and use of pm ANIMATOR, as an introduction to PMG itself, and as an introduction to some of the basic concepts underlying computer graphics. You do not have to learn all the technical information and terminology in order to use this package, but becoming familiar with it can only help you become a better programmer.

Therefore, the first six chapters in the manual deal with some of the basics of computer graphics and the main features of player-missile graphics. If bits and bytes, bit maps, memory maps, and players are unfamiliar concepts to you, take the time to go through these chapters.

If you are comfortable with these ideas, go on to the second section of the manual which deals with how to use the pm ANIMATOR package. Chapters 7, 8 and 9 explain how to use the Grafix Editor and the File Editor. Chapters 10 and 11 explain how to implement pm ANIMATOR animation in your own programs; the use of machine language subroutines via the USR function is explained at this point, as well as the function of each of the utilities and subroutines provided. Chapter 12 discusses the demonstration programs provided in this package.

The third section of the manual deals with techniques of player-missile animation as well as advanced programming topics. The techniques discussed in Chapter 13-16 will be extremely helpful to you when you have mastered the earlier material.

The manual concludes with two appendices. One summarizes all the hardware registers in the Atari that deal with PMG, and contains important information to be used when programming with PMG. The second appendix is a list of suggested references on programming and the Atari computer.

Some friendly advice: pm ANIMATOR is a *programming tool*, not a game that is ready to play. Despite its power and useful features, it still requires *you* to come up with all the clever programming ideas. If you had the finest woodworking tools available, you would still need to do a lot of carpentry before you could design and build fancy furniture. Similarly, while pm ANIMATOR puts some powerful tools in your hands, you will have to work with it awhile before you'll be ready to write the next STAR RAIDERS. Have fun! It'll be worth it!

SECTION I
Player-Missile Tutorial

Chapter 1

What's So Great About Player-Missile Graphics Anyway?

Since you decided to get the pm ANIMATOR package, one of two things must be true: you know all about PMG and are anxious to find a more convenient way to use it, or you've heard that PMG is hot stuff and you want to get in on it. If you fall into the first category, you can probably skip this chapter as well as the next five. Otherwise, we will try to point out just what *is* so great about PMG.

Players and missiles get their names from a highly successful application of Atari's advanced graphics system: arcade games. But players and missiles aren't necessarily part of games at all. They are simply special graphic images, designed for rapid movement on the Atari graphics screen, that are independent of any other graphics or text information on the screen. There are four players available for use at any one time as well as four missiles. Players and missiles are limited in width but extend over the full height of the screen. As you will see, missiles are essentially just narrow versions of players.

There are several advantages to using players and missiles for Atari graphics. First, they are independent of everything else on the screen. The usual graphics modes on the Atari (and on most other computers as well) deal with the screen as a grid of individual boxes called pixels. Pixels are the "atoms" of the television picture. To put an image on the screen we have to map out a pattern of pixels, and to move that image we have to map out a whole new pattern. This kind of graphics is called bit-mapped graphics; some PMG programmers call it "playfield graphics". (We will learn more about bits and bit-mapping in the next two chapters.) In player-missile graphics, on the other hand, the graphics image is treated as a single entity that we can move around the screen at will regardless of what else is there. We will also find that each player and its associated missile (missile 1 has the same color as player 1 and so on) has its own color that is independent of any other color(s) on the screen. Thus, the four players give us four additional colors that can be on the screen at any one time

(actually, the four missiles can be combined to form a fifth player with its own fifth color). Chapter four discusses the fine points of defining players and missiles. For now, it suffices to say that players and missiles give us independent graphics images to incorporate into our programs.

Besides this useful independence of screen location and color, PMG has the important advantage that its graphics images can move quickly. In bit-map graphics or character graphics, when we want things to move on the screen, a lot of microprocessor time is spent erasing the previous graphics images and redrawing the new graphics images. All this work for the 6502 leaves little time for other tasks, which limits the complexity and/or execution speed of programs containing elaborate graphics. PMG comes to the rescue because it is built into the hardware of the Atari in such a way that most of the work is done by the dedicated video chips ANTIC and CTIA (or GTIA) instead of the 6502 using a neat trick called Direct Memory Access (DMA). This leaves much more time to execute complicated programs.

So now we know that PMG gives us a graphics system of up to 8 objects that can move around (or even under!) whatever background we create. It gives us many more colors on the screen at once, faster motion, and time for much more complicated programs. Fantastic! So where's the catch?

Well, PMG is an intrinsic part of the Atari hardware and, compared to the PLOT/DRAWTO graphics of BASIC, is much more on the nuts-and-bolts (actually bits-and-bytes) level of the machine. So, to do it right, we need to have a more sophisticated understanding of the inner workings of the computer than we need to execute bit-map or character graphics from BASIC. pm ANIMATOR will allow you to take advantage of the power of PMG without a thorough grounding in the computer's inner workings. The information contained in this manual will give you enough knowledge to easily incorporate the pm ANIMATOR routines into your BASIC programs, and to put sophisticated animated graphics into your own software without knowing all the ins and outs of the hardware. The following five chapters will give you enough background information so that you won't be lost in the language of player-missile graphics.

Chapter 2

Bits, Bytes, and their Arithmetic

It is not necessary to read the following five chapters to use pm ANIMATOR to generate amazing arcade-style graphics with Atari BASIC. However, an understanding of how player-missile graphics works will allow you to make the fullest use of the "graphics power" that your Atari computer can muster. The following chapters will start out with a few general programming concepts, and will build upon these concepts to explain player-missile graphics and its use.

On to the topic at hand; bits and bytes. When information is stored in a computer's memory, it is broken down into small pieces which are represented by numbers. Bits are the smallest units of comuter memory, and they are used to represent the smallest pieces of information that can be stored in a computer. A bit can take one of two values, 1 or 0, and these values are used to represent the pieces of information. For example, suppose we want to represent the information that something is either true or false. We can represent true with "1" and false with "0". A single bit can represent any *two* pieces of information. Thus, for a screen display, "1" can represent the color red and "0" can represent the color blue. However, a single bit cannot represent red, blue, *and* green. To represent more than two things, we need to use more than one bit.

Bits are the building blocks for storing information in a computer. By using more than a single bit we can represent more than 2 things. A collection of 8 bits turns out to be a very convenient object to consider and has a name of its own: a byte. Bytes are important to us because the Atari home computer uses a 6502 microprocessor, which is an 8 bit (1 byte) machine. That means that each memory location in the Atari holds 1 byte (remember that a byte is 8 bits). A byte can be written as an eight-digit number composed of only 1's and 0's, each of which is a bit. We will write down a byte and learn some useful terminology.

10001000

Each bit in a byte is numbered, so the rightmost bit (also called the least significant bit) is BIT 0. The leftmost bit is BIT 7. Notice that although there are 8 bits to each byte, the bit numbers range from 0 to 7. So in the above byte, bits 3 and 7 are true (or have a value of one). You can also say that bits 3 and 7 are turned on. A nibble is 4 bits. BITS 0-3 are the low nibble of a byte and BITS 4-7 are the high nibble.

This (or any) byte can be expressed as a normal decimal number. The trick is to realize that bytes are actually base 2 or binary numbers. The number 1234 decimal (base 10) actually means

(1X10x10x10)+(2x10x10)+(3x10)+(4x1)=1234

Similarly, the number 10001000 in base 2 means

(1x2x2x2x2x2x2x2)+(0x2x2x2x2x2x2)+(0x2x2x2x2x2)+(0x2x2x2x2)+(1x2x2x2)+(0x2x2)+(0x2)+(0x1)=10001000

or to simplify things

(1x128)+(0x64)+(0x32)+(0x16)+(1x8)+(0x4)+(0x2)+(0x1)

If we do the addition, we find that the binary number 10001000 means 136 in decimal.

We have just *converted* the binary number to a decimal number. We will do more conversions shortly.

Base 10 means that each numeral represents some number times a power of ten. The rightmost number is the number of ones, next to it on the left is the number of tens, then the number of hundreds, and so on. Each column is a power of ten. Binary (base 2) works the same way except that the digits represent powers of two instead of powers of ten.

The largest byte that can be written is 11111111, which translates to 255 in base ten. Since the smallest byte, 00000000, simply means 0, a given byte can have 256 different values. Just as one bit can be used to represent two different things, one byte can be used to represent 256 different things.

How can we tell if a number is base 10, base 2, or even hexadecimal (base 16)? The following standard rules will apply throughout this manual. If the number is preceded by a percent sign (e.g. %1000100), then that number will be in base 2. Hexadecimal (base 16) numbers – hex for short – will be preceded by a dollar sign (e.g. $77). (Base-16 numbers will be very useful later on.) If there is nothing in front of the number it is decimal (base 10).

We will now do some more number conversions. We start with the decimal number 254 and convert it to binary. Remember that since 254 is between 0 and 255 the binary number will be 1 byte long (8 bits). Each bit stands for a power of 2:

Bit-7	Bit-6	Bit-5	Bit-4	Bit-3	Bit-2	Bit-1	Bit-0
128	64	32	16	8	4	2	1

So, we do the conversion:
254 is larger than 128 so the binary number is now

1xxxxxxx + 126 remainder

(we put x's in the places we don't know yet)
126 is larger than 64 so we now have

11xxxxxx + 62 remainder

62 is larger than 32 so we get

111xxxxx + 30 remainder

30 is larger than 16 so

1111xxxx + 14 remainder

14 is larger than 8 so

11111xxx + 6 remainder

6 is larger than 4 so

111111xx + 2 remainder

2 is equal to 2 so

1111111x + 0 remainder

since 0=0 the last bit is 0, so 254=%11111110

The same type of routine will convert any number from one base to another. The powers of 16 are

16x1=16
16x16=256
16x16x16=4096

So the number $8765 would convert to decimal as follows:

(8x4096)+(7x256)+(6x16)+(5x1)=34661 decimal

Next, we consider how the number 255 converts into hex. 255 is less than 4096 and is less than 256, so we can immediately write the first two digits as zeros ($00xx) or simply

write $xx. There are 15 sixteens in 255 with a remainder of 15. We now have a problem. We have to somehow write a fifteen in each digit of our hex number. How do we represent 15 in hex? We use the normal decimal numbers 0 through 9 for the numbers below ten and we use the beginning letters of the alphabet to represent the numbers 10-15 as follows:

10=$A
11=$B
12=$C
13=$D
14=$E
15=$F

So the number 255 is equal to $FF, a fifteen in each place. We can now write any number in hex. For example, $ED is equal to (14x16)+13, or 237 decimal.

Now we know how to use base-16 numbers. Why bother? It is very convenient to use hexadecimal numbers to represent binary numbers because there is a shortcut for converting from one to the other, and it is certainly easier to write or talk about numbers like $FF instead of %11111111. If you want to convert from binary to hex, here is the easy way to do it. Remember that a nibble is four bits long and there is a high nibble and a low nibble in each byte. The number %10100101 has a high nibble of %1010 and a low nibble of %0101. To convert %10100101 to hex, convert each nibble to hex separately. Thus, in our example, %1010=10=$A and %0101=5=$5. (We treat the high nibble as if it were a low nibble for the conversion; i.e., think of its columns as ones, twos, fours, and eights.) Now just as the two nibbles side-by-side make up the byte, the two hex numbers side-by-side give the hexadecimal conversion of the whole byte. Thus, %10100101=$A5. It's that easy to convert from binary to hex. (If you doubt this, try it the long way to convince yourself that you get the same results.)

All of this strange arithmetic becomes important when we wish to draw a player missile image because it is in binary form that information is stored by the computer. Therefore, it is important that you become familiar with binary and hex if you want to really understand how PMG works, although you do not need to know this information to use pm ANIMATOR.

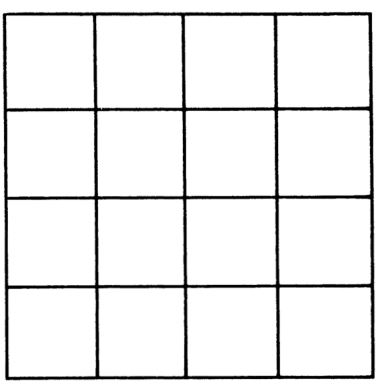
Chapter 3

Bit Map Graphics

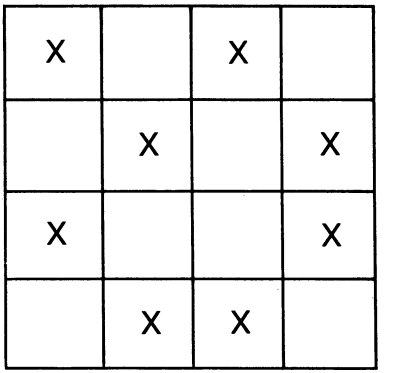
This chapter will cover bit-map graphics and describe how several of the different Atari graphics modes work.

Bit-map graphics is very similar to a technique sometimes taught in high school art classes. In this technique, you place a grid pattern over a picture to break it up into many little boxes, each containing a piece of the overall picture. Each little box contains a far simpler image than the picture it comes from. By copying the little boxes one by one, any student could do a fair job of copying even the most complicated picture. Bit-map graphics is the computer's way of storing pictures in a simplified form by breaking up the picture into little boxes – pixels on the TV screen – and representing the boxes with bits and bytes. Some examples will show how this works.

We start out with a field that has 16 boxes on it arranged like this:



Now we will fill some of these boxes with an X as shown:



X		X		1	0	1	0
	X		X	0	1	0	1
X			X	1	0	0	1
	X	X		0	1	1	0

On the right side of this diagram we have represented each box with an X in it with a "1" and each empty box with a "0". If we look at each of the rows we see that a binary numer is formed and we can assign an equivalent decimal number to each row.

Row	Binary	Decimal
1	1010	10
2	0101	5
3	1001	9
4	0110	6

If this isn't clear to you, you might try rereading the previous chapter about bits and bytes, or look into some of the further readings suggested in Appendix II.

In the above example, each box is represented by 1 bit, either a "1" if it has a X in it or a "0" if it is empty. Each row is made up of 4 boxes, so we can represent where the full and empty boxes are located in each row with 1 nibble (remember that a nibble is 4 bits). If you think about it, you'll see that the single nibble (it represents one number) uniquely defines the pattern of full and empty boxes in its row. You might try drawing your own pattern in the grid and evaluating the four numbers from the picture. The four numbers (nibbles) that describe the bits in this little picture are exactly like a map of the grid. You have "bit-mapped" the picture.

In our example, each box can have only 2 different values, 1 or 0. If 1 stands for red and 0 stands for black, then the boxes that have 1's in them will be red and the boxes with zeros will be black. What we have just described is roughly the way Graphics 8 works on your Atari.

We will alter our example to better describe Graphics 8. Imagine a field that is made up of 192 rows and 320 columns. Again we'll let 1 bit represent each box. We can easily see that we can have 2 colors on the screen, a color represented by a 1 (for example, red) and a color represented by a 0, which we will call the background color. All boxes that are red will be considered to make up "playfield" 1. On the Atari computer a

playfield is defined as all of the boxes that have the same value (determined, for example, by the COLOR statement in BASIC). All the boxes in a playfield necessarily have the same color, but two boxes might have the same color and not be part of the same playfield. If the background color was the same as playfield 1, then you would not be able to see which boxes were playfield 1 and which boxes were background.

In graphics mode 8, each row is made up of 320 boxes, as we said. Since there are only two colors allowed, each box can be represented by only a single bit. Since there are 8 bits in a byte, there are 320/8 bits or 40 bytes required to represent each row. The total amount of RAM required to display the entire Graphics 8 screen is 192x40 or 7680 bytes. Your Atari BASIC manual states that Graphics 8 uses 7900 bytes. This discrepancy is because of the display list, which is a program that tells the Antic chip exactly how to display the screen. (A full explanation of display lists is beyond the scope of this manual. Please consult some of the other sources in the reference list for further information.)

Now, suppose that we are designing a hypothetical computer, and we want it to have a graphics mode that will display 320 boxes by 192 boxes in 4 different colors. We therefore need to figure out the minimum number of bits that we must use to represent each box. We need 2 bits for each box. Remember that 2 bits can represent 4 different objects:

%00 = 0
%01 = 1
%10 = 2
%11 = 3

If we represent each box with 2 bits, then there are (320/8)x2 bytes needed to represent each row, or 80 bytes per row. Over a full screen this will require 15360 bytes, not including the display list! So this graphics mode will be very expensive as far as RAM usage is concerned.

So far we have assumed that a box can only be in one of two states ("on" or "off"). We also know that the number of different colors that a box can assume is dependent on the number of bits that represent each box. This sort of graphic display is called a bit-map display mode. In bit-map display modes, an area on the screen (a box or pixel) is entirely lit up. The size of the boxes is dependent on the resolution. If the screen is divided into 16 (4x4) boxes, the area of each box will be very large. In our Graphics 8 example above, the boxes will be very small; they will look like dots on the television screen. However, this technique of lighting up little squares on the screen (bit-map display) is not the only way to do computer graphics.

Another method is called character graphics. We can use Atari graphics mode 0 as an example. In this mode we can display 24 rows of characters with each row being made up of 40 characters. Each character that appears can be any one of 256 different characters. How many bits must we use to display 256 different characters? The answer is 8 bits or one byte (remember there are 256 possible values for a byte). Each box on the screen that can contain a graphics character is called a graphics cell, and it can have values between 0 and 255, represented internally by 8 bits. How much memory is used by a graphics mode 0 screen? We have 40 columns per row, 24 rows per screen, and 1 byte per cell: 40x24x1, or 960 bytes, not counting the display list.

In character graphics, we no longer merely turn on or off a little box (pixel). Now we can place one of 256 characters in the box. How are the characters themselves stored in the machine? As bit-maps. Each character has a unique set of numbers that describes how to form the character from individual pixels. The ATASCII code 0-255 for a character tells the computer which bit-map to look up in its table in memory. We will see how to bit-map characters shortly.

All of the Atari graphics modes work along the principles outlined in this chapter. See the Technical Users Notes for technical information about the 14 other Atari graphics modes.

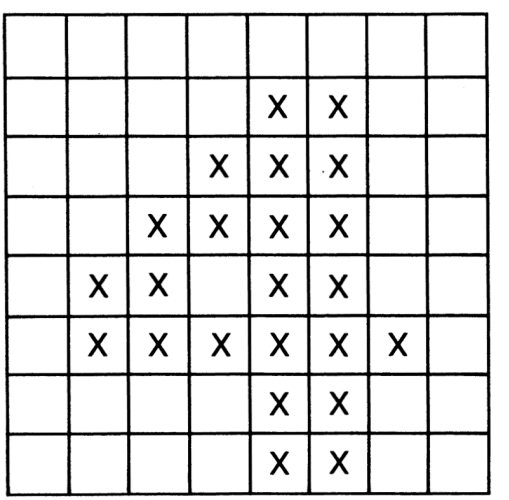
In this chapter, we have seen how a pattern of full and empty boxes in a grid can be represented by simple binary numbers. The set of numbers corresponding to a given pattern is called (for obvious reasons) a bit-map. It is important that you have a firm understanding of the concept of bit-mapped graphics to understand how player-missile graphics is implemented on the Atari. The following chapter will show how such a bit-map is used in PMG. If you have trouble with the first example in the next chapter, come back to this chapter and try to clear up your confusion before going on.

Chapter 4

Principles of Player-Missile Graphics

In this chapter we will begin the discussion of player-missile graphics itself. For the moment, we will ignore the actual commands used to initiate it and concentrate on the underlying operating principles of PMG.

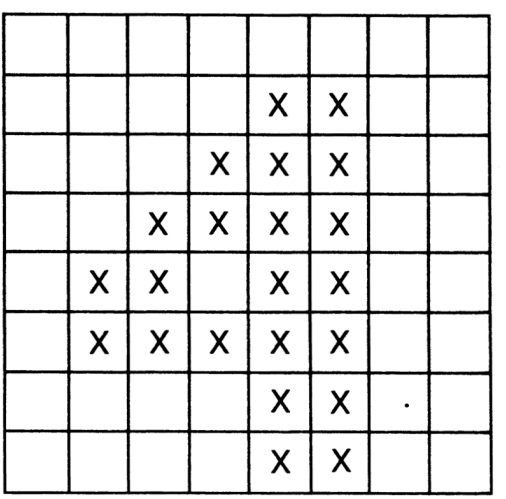
As in the previous chapter, we will use a grid – on a computer screen, a colored background – divided into 64 squares: 8 columns and 8 rows:



We have filled some of the squares with X's to create a "4", which in a more compact mode looks like this:

```
    X X
   X X X
  X X X X
X X   X X
X X X X X
    X X
    X X
```

From the previous chapter, we know that each box can be represented by a bit, with a 1 representing an X and an 0 representing an empty box. In this case, each row is made up of 8 boxes which will be represented by 8 bits or 1 byte. If we write the numerical representation to the right of our grid, we produce the following bit-map:



%00000000 = 000
%00001100 = 012
%00011100 = 028
%00111100 = 060
%01101100 = 108
%01111110 = 126
%00001100 = 012
%00001100 = 012

So if we were given the decimal numbers:

0
12
28
60
108
126
12
12

we could convert them into binary, bit map them out, and see that these numbers, when bit-mapped, would look like a "4". This is how the characters of character graphics are produced. This is also the idea behind drawing a player-missile image.

The PMG images we can store as bit-maps and move around on the television screen are either players or missiles. Each player is 8 bits wide (missiles are narrower). This means that the horizontal width of a player is represented by 1 byte, or 1 memory location. We will now consider how to determine where each player will appear on the field (or television screen). First consider the vertical position of the player.

Imagine that your television set has a grid on it that has 8 columns and 128 rows. Each row can therefore be represented by 8 bits or 1 byte. We can number each row starting from 0 at the top down to 127 at the bottom (memory locations usually start from 0 rather than 1). If we wanted our "4" from the example above to be near the top of the screen, we would place the bytes that represent the "4" in rows 0-7. If we wanted the "4" near the bottom, we would place the numbers that make up the "4" in rows 120-127. This is exactly how the vertical location of the player is controlled. We set aside some memory in the Atari which will hold the player-missile data. 128 bytes of this memory controls the vertical location of one player (this is double line resolution which will be explained later).

We now want to move our "4" up the screen. Suppose that the "4" extends from row 100 to row 107 (this places the "4" about three quarters of the way down the screen). To move the "4" up one row, the data that was in row 100 must be put in row 99, row 101's data in row 100, 102's in 101, and so forth, and a 0 must be put into 107 (we now want nothing there instead of the old data). This could be accomplished in BASIC by the following program. We will assume that memory reserved for our player missile data starts at memory location 1000

```
10 FOR X=1000 TO 1007
20 POKE X-1, PEEK(X)
30 NEXT X
40 POKE 1007,0
```

Likewise, if you wanted to move the player down by 1 row you could use the following program:

```
10 FOR X=1007 TO 1000 STEP-1
```

```
20 POKE X+1, PEEK(X)
30 NEXT X
40 POKE 1000,0
```

The higher in memory that you move a player image the lower on the television screen that player will appear.

You should think of each player as extending from the top of the screen to the bottom of the screen. We could make the player take up the full height of the screen if we so desired by putting data in all 128 rows. Contrastingly, if there is no data in the player area, the player would not be seen at all. In a game using player-missile graphics, although we see a figure that looks small, the player actually extends from the top of the screen to the bottom. We just can't see these other areas because there is no player data there.

What has been described above is called double line resolution, because each row that makes up the player is 2 lines thick. It is also possible to have single line resolution where the screen will be divided into 256 rows; however, this mode uses up twice as much memory.

The horizontal position of each player is controlled by a single memory location, set aside for that purpose, called the horizontal position register. Each player has its own. To control a player's horizontal placement on the screen you POKE a number between 41 and 200 into the horizontal position register belonging to that player. If you want a certain player in the middle of the screen, for example, you POKE 120 into its horizontal position register.

Aside from the players, we also have four missiles to work with. Like players, missiles should be thought of as extending from the top of the screen to the bottom, although, of course, a given missile might not have data in all 128 (or 256) rows. Recall that players are 8 bits wide, which means that the data in one row of a player is represented by one byte. Missiles, unlike players, are only 2 bits wide – the data in one row of a player is represented by 2 bits. For a given row on the screen, the data for all four missiles are represented together in a single byte. This byte is divided up as follows:

Bit Assignments

7	6	5	4	3	2	1	0
Missile 3		Missile 2		Missile 1		Missile 0	

So bits six and seven contain data for missile 3, and so on. For example, if the byte in row 63 of the missile data is 204=%11001100, then missile 3 and missile 1 each have their complete width turned on at the vertical middle of the screen. Just as for players, each missile has its own horizontal position register. However, unlike the players, each missile does not have its own color register. In fact, a missile assumes the color of its associated player, i.e., missile 2 has the same color as player 2 and so on.

Collisions, playfields, priority, and other considerations

Now that we know (in principle) how to place a player or missile on the screen, we can even think about what happens when players and missiles collide, as they often do in games. Before we deal with the way collisions are recorded, we need to return to the concept of a playfield.

In graphics modes 1 and 2, the Atari computer can have up to 4 playfields and 1 background on the screen at any one time (this is different in other modes). Remember the question from the last chapter: how many colors can be displayed on a screen at one time? The answer is dependent on how many bits we use to represent a pixel. Pixels are the atoms of the video screen. A video image is divided into no finer parts than the pixels that make it up. All of the boxes that we have been using as diagrams in this tutorial could also be called pixels.

If we use 2 bits to represent each pixel then we can only display 4 different colors on the screen, as follows:

%00 = Color 1 (background)
%01 = Color 2 (Playfield 0)
%10 = Color 3 (Playfield 1)
%11 = Color 4 (Playfield 2)

This is how Graphics mode 7 is set up on the Atari. Each playfield is made up of boxes that have the same value (color), e.g. %10 is playfield 1 in Graphics 7. All pixels in any one playfield have the same color which can be controlled by a single memory location. However, two different playfields can have the same color. Each playfield has its own color register so that its color can be varied independently of the others.

With this in mind, we can consider collisions. Collisions occur when two graphics objects – a missile, player, or playfield – want to occupy the same spot on the screen. Imagine a missile "hitting" an airplane. Both the missile and the airplane will occupy the same spot on the screen at the same time, and a collision is said to have occurred. The Atari automatically records each collision that occurs and stores this information in memory. By looking (PEEKing) at a specific memory location, we can determine, for example, if player 2 collided with missile 3. The following collisions are monitored by the Atari:

Missile-to-Playfield : did one of the four missiles collide with any of the playfields, and exactly which missile and which playfield.

Player-to-Playfield : did one of the four players collide with any of the playfields, and exactly which player and which playfield.

Missile-to-Player : did any of the 4 missiles collide with any of the 4 players, and exactly which player and which missile.

Player-to-Player : did any of the 4 players collide with each other, and which players actually did collide.

The following collisions are not monitored: playfield-to-playfield collisions, and missile-to-missile collisions. All of the collision bits can be cleared by writing to a specific location, and then new collisions can be monitored.

The Atari was designed to display 4 players and 4 missiles. Each missile assumes the color of its player, so that missile 1 will have the same color as player 1 and so on. It is possible to combine the 4 missiles into a fifth player which can have its own separate color.

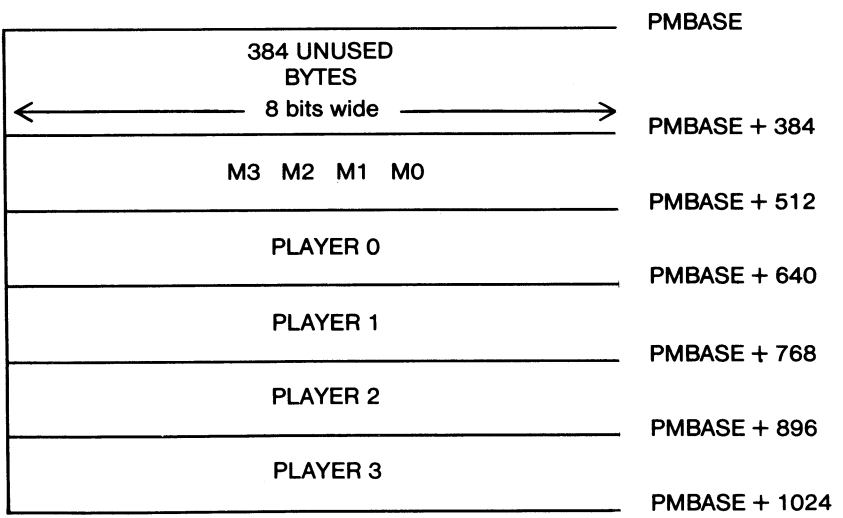
The last thing that we have to consider is priority. When 2 objects, either players or playfields, want to occupy the same spot, the Atari needs to be told what it should display. Imagine a card passing in front of a curtain: you will see the card and you cannot see the curtain that is behind the card. You could then say that the card has a higher priority than the curtain. On the Atari we have 4 players and 4 playfields to consider, and by writing to the appropriate address we can control which objects will "pass in front" and which objects will "pass behind" each other. The possible priorities that can be set are listed in Appendix I.

This chapter has served as a brief introduction to some of the fundamentals of player-missile graphics. Before we can embark upon the implementation of PMG into real programs, there is a final general topic to be discussed: the placement of PMG data in the memory of the Atari computer. This is the topic of the next chapter.

Chapter 5

Memory Maps and The Shadow

In this chapter two further concepts will be covered: memory maps and shadow registers. A memory map is exactly what it sounds like. It shows how the memory locations in the computer are being used. It can be very specific and display each byte (or bit), or it can be general and show large areas of memory. In this case we will look at a general memory map of double line resolution player-missile graphics.



How can we understand all this? We first define PMBASE. PMBASE is a variable that defines where in memory the player-missile memory begins. Player-missile memory is an area in memory containing data that will be placed on the screen in the manner described in chapter 4. Although PMBASE could theoretically be anywhere in memory, it turns out that PMBASE must be on a 1k boundary for double-resolution PMG or on a 2K boundary for single resolution PMG. A 2k boundary means that PMBASE must be a number that is divisible by 2048 (2 kilobytes means 2048 bytes, not 2000 as you might expect); a 1k boundary must be divisible by 1024.

For demonstration purposes only, we'll assume that PMBASE=20000. The top line that is opposite PMBASE now represents memory location 20000. Notice that in that top box it says that 384 bytes are unused. Therefore, memory locations 20000 thru 20383 are not used by PM graphics and can be used by you for any other purpose (like storing your PM images). The next area contains 4 boxes that start at PMBASE + 384 (20384) and go to PMBASE + 512 (20512). This area of memory is where the missile data is mapped onto the screen. This area is 128 bytes long and is mapped out as explained in Ch. 4 (i.e. bits 6 and 7=missile 3, bits 4 and 5=missile 2, bits 2 and 3=missile 1, bits 0 and 1=missile 0). If missile data is located at 20400 then the missile will be located near the top of the screen. If the data is at 20500 then the missile will appear at the bottom of the screen.

Player 0 vertically maps into 128 locations from 20512 (PMBASE + 512) thru 20639 (PMBASE + 639). Player 1 vertically maps into locations 20640 (PMBASE + 640) thru 20767 (PMBASE + 767). Player 2 vertically maps into locations 20768 (PMBASE + 768) thru 20895 (PMBASE + 895). Player 3 vertically maps into locations 20896 (PMBASE + 896) thru 21023 (PMBASE + 1023). Notice that each player area and the missile area are 128 bytes in length. The screen is then divided vertically into 128 rows. Remember, where the missiles and players are located in memory will determine the vertical placement of the missiles (or players) on the screen. By checking the memory map, you can tell exactly where in memory each player is located.

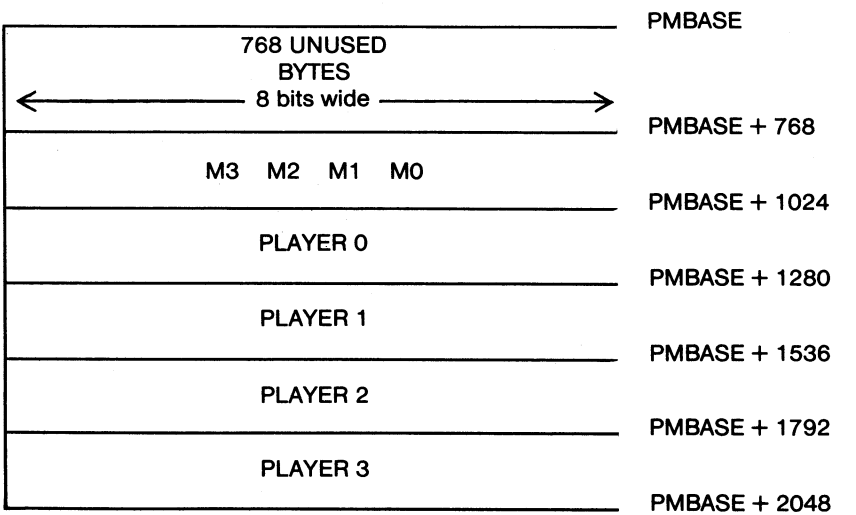
If you are following all this, you should be able to answer these questions. We will tell you where the data is located using the above memory map and you should be able to tell which player or missile is involved and where on the screen it will appear. Answer with the player or missile number and whether it appears on the top, middle, or bottom of the screen.

question	memory	data
1	20390	48
2	20890	255
3	20200	255
4	20520	100
5	20704	200
6	20448	192
7	21000	52
8	20510	195
9	20832	255
10	20770	200

The correct answers are:

1) missile 2, top of screen
2) player 2, bottom of screen
3) in unused area, so nothing on screen (this was a trick question)
4) player 0, top of screen
5) player 1, middle of screen
6) missile 3, middle of screen
7) player 3, bottom of screen
8) missile 3 and missile 1, bottom of screen (195=%11000011)
9) player 2, middle of screen
10) player 2, top of screen

The memory map presented above was for double line resolution player-missile graphics. This means that the screen is divided vertically into 128 rows. There is also single line resolution on the ATARI computer. Single line resolution means that the screen is divided vertically into 256 rows. Here is the memory map:



In single line resolution the television screen is divided vertically into 256 rows, so each of the above boxes represent 256 memory locations (bytes). The entire amount of memory the player-missile data base takes up is 1024 bytes in double line resolution and 2048 bytes in single line resolution. Since the data must be stored on a 1K or 2K boundary, we see that it exactly "fits" into a 1K or 2K section of memory.

Before we can start working with PMG and before we can make use of the PMG registers listed in the first appendix, we have to discuss one more principle: shadowing.

The Atari computer has more than a 6502 microprocessor in it. It has a video microprocesor called ANTIC, a video interface chip called CTIA (or GTIA in later machines), and a sound chip called POKEY. Each of these chips has registers that allow us to issue instructions directly to the chip. These registers are mapped directly into the 65536 bytes of memory that are addressed by the 6502.

However, there is one small problem. Some of these registers are "read only" and some are "write only". This can lead to the following situation: we might write (POKE) a certain value into register x, and then read (PEEK) the same register x and get a completely different value. Using a real example, we can tell GTIA (or CTIA) that player 0 should be color x by POKEing x into GTIA register COLPM0 ($D012). This GTIA register is located at memory location $D012 or 53266 decimal. However, if we PEEK into 53266 we don't find the color of player 0 at all. Instead, it turns out that we are reading the register which tells if the trigger has been pressed on joystick 2. How do we find out the value of register COLPM0, a write-only register? The answer lies in a shadow register.

The people who designed the Atari use a system called shadowing, so that we can find out what color player 0 is. In shadowing, a different memory location contains the value in the hardware register. It works like this. The shadow of COMPM0 ($D012) is location 704 ($2C0). Sixty times every second the value that is in location 704 is POKEd into location 53266. This is done for most of the Atari hardware "write only" registers. Not all the registers have shadows; two write-only registers that do not have shadows are the player and missile horizontal position registers. It is up to you to keep track of the horizontal positions of the players and missiles; there is no shadow register to PEEK into. In case you are wondering, the updating of the register from its shadow occurs during the Vertical Blank Interrupt (VBI), the time during which the electron beam in the television is shut off while it goes from the bottom right of the screen back up to the upper left to start a new frame of the picture. This is "free time" for the ANTIC chip. (A full description of VBI's is beyond the scope of this manual. Please consult the Technical Users Notes for more information about interrupts). In any case, this updating of the hardware registers is transparent to the user, meaning that we are not aware that it occurs, and do not have to do anything to insure that it occurs.

If we want to write anything to a hardware register, we first check to see if that register has a shadow, and if it does, we write to the shadow instead of the hardware register directly. If we wrote to the hardware register directly, we would effect no change because the vlaue that was stored in the shadow would be POKEd back into the hardware register as soon as the next VBI occurred. Therefore, the only way to change the value in a "shadowed" register is poke the new value into the shadow first, and then into the hardware register.

This sums up most of the technical background needed for the use of player-missile graphics. The next chapter demonstrates the cumbersome steps to be taken to implement PMG into a BASIC program. Fortunately, you have a sophisticated set of

tools in the pm ANIMATOR package that removes much of the burden in using PMG effectively. Now that you know what a bit-map is, you can let the Grafix Editor create one for you. Now that you know about setting up the PMG memory area and the hardware registers involved, you can let the TOTAL.LST subroutine do all the dirty work for you. In Chapter 7, we will move on to the use of the pm ANIMATOR package.

Chapter 6

A Real Example (No Machine Language)

This example will show you how to use player-missile graphics strictly from BASIC. Although this example is slow and rough it accomplishes its purpose. This is the way you would set up and use player-missile graphics if you didn't own a powerful graphics system like pm ANIMATOR.

First we must clarify that there are only six steps required by the computer itself to put a visible player on the screen. However, to make player-missile graphics faster and easier, and to add animation requires additional set-up steps.

Set-Up Steps For Player-Missile Graphics

These are the steps necessary for the computer to set up player-missile graphics.

(1) Enable player-missile 'Direct Memory Access' (DMA) by POKEing location 53277 with a 3. (This allows for a section of memory to represent the player areas; anything poked into these areas will be seen in the player area on the screen.)

(2) Enable player-missile DMA control by POKEing 559 with a 62 for single line resolution.

(3) Store the PMG images (pm ANIMATOR files or other bit-mapped data) somewhere in 'protected' RAM (strings, page six, etc.).

Note: The next steps can occur in any order.

(4) POKE the bit image into the desired locations in player-missile memory from protected memory.

(5) POKE the appropriate player horizontal position register (53248 to 53251) with a position on the screen (usually from 30 to 225, varying with the television or monitor).

(6) Give the appropriate player a color by POKEing its color register with the desired color value (see Atari BASIC manual).

These set-up procedures are those that are absolutely required by the computer. However, these steps alone do not provide high speed animation or vertical movement; in fact, they make player-missile graphics tedious and slow to use.

The example below will blank the screen out, draw the playfield and then put a green flying saucer (player zero) on the screen. The horizontal and vertical motion is controlled by the joystick.

```
10 REM SAUCER DEMO, TOTALLY IN BASIC.
20 REM THIS IN NO WAY REFLECTS THE QUALITY OF pm ANIMATOR!!!
80 GRAPHICS 8:POKE 710,0:POKE 559,0:COLOR 1:REM SET UP COLORS AND SHUT OFF
SCREEN.
85 FOR I=1 TO 100:X1=INT(RND(0)*280)+1:Y1=INT(RND(0)*180)+1 :PLOT X1,Y1:NEXT I:REM
DRAW STARS.
90 PM=PEEK(106)-40:PMB=256*PM:REM FIND WHERE TOP OF RAM IS AND PUT PMBASE
JUST BELOW IT.
100 POKE 53277,3:POKE 54279,PM:REM LINES 100 TO 110 ARE THE SET-UP STEPS TALKED
ABOUT EARLIER.
110 HP=53248:POKE 559,62:POKE 623,1
120 POKE 704,200:POKE HP,100
130 DATA 24,126,255,255,126,60,36,66,129:REM DATA FOR FLYING SAUCER SHAPE
140 X = 100:Y = PMB + 1024 + 100: REM X IS THE HORIZONTAL POSITION, Y IS THE EXACT
LOCATION OF THE SAUCER IN MEMORY (VERTICAL POSITION)
150 FOR I=1 TO 9:READ A:REM DRAW SAUCER IN MIDDLE OF SCREEN
160 POKE PMB+1024+100+I,A:NEXT I
500 T=STICK (0): IF T=15 THEN GOTO 500:REM CHECK TO SEE IF JOYSTICK IS BEING MOVED
510 ON T GOTO 500,500,500,500,500,500,520,500,500,500,530,500,540,550: REM IF JOYSTICK
IS MOVED IN THE RIGHT DIRECTION GO BELOW, IF IT ISN'T THEN GOTO 500
520 X=X+1:POKE HP,X: GOTO 500:REM MOVE ONE SPACE TO THE RIGHT
530 X=X-1:POKE HP,X:GOTO 500:REM MOVE ONE SPACE TO THE LEFT
540 RESTORE 130:Y=Y+2:POKE Y-1,0:POKE Y,0:FOR I=1 TO 9:READ A:POKE Y + I,A:NEXT
I:GOTO 500:REM MOVE TWO SPACES DOWN (HIGHER IN MEMORY)
550 RESTORE 130:Y=Y-2:FOR I=1 TO 9:READ A:POKE Y+I,A:NEXT I:POKE Y+11,0:POKE
Y+10,0:GOTO 500:REM MOVE TWO SPACES UP (LOWER IN MEMORY)
```

Now that you have seen player-missile graphics work from BASIC, you are ready to see what pm ANIMATOR can do.

SECTION II
Owner's Guide to pm ANIMATOR

Chapter 7

Introduction To Using
The Grafix Editor And File Editor

The Grafix Editor is an all-purpose editing tool that enables you to build and edit pictures for animation. It allows you to animate them as they progress, in any sequence you wish, and to see and perfect your animation before it is put into a program, so there is almost no need to 'hand-map' these figures. The Grafix Editor also has many special commands for manipulating the pictures you are creating to achieve the best possible results in the least time. And it includes many convenient DOS-like commands, such as DELETE, FORMAT, LOCK, etc.

To understand the concept of animation and how the Grafix Editor works, visualize a film projector. If you were to look at the actual film it would appear to be a sequence of small transparent pictures or 'frames', each a little different from the one before. When we view the film, the motion our eye sees is created by the individual pictures flipping by so quickly that we perceive them as one picture that is moving instead of separate still frames. The same sort of thing happens in computer animation. From here on we will refer to the separate pictures which create the animation as *animation frames*.

Before we go on, it is worthwhile to discuss the difference between motion and animation. Since animation and motion seem almost synonymous, this difference may seem subtle at first but it is an important one.

Animation is a type of motion which is created by altering a frame. Take the example of a bird in flight. If we were able to stop the bird's forward progress, we would still see the motion created by the bird flapping its wings. This type of motion is called animation because it is created by changing the shape of the animal; that is, the bird changes its body position as it moves. In computer graphics, animation is the motion made by changing frames only; other types of movement will simply be called "motion".

Motion denotes direction, progress and a change of position relative to surround-

ings. Consider a ball tossed into the air. It is constantly changing position and direction and it is making progress by moving from one place to another. If we did not see the ball's surroundings, we would not be able to notice any motion. In PMG, this type of player motion is controlled by changing the horizontal and vertical position registers, not by changing frames.

The Grafix Editor works on one file of sixteen frames at a time, each frame being 16 blocks tall by eight wide. We can think of the file as a strip of film that will be used for the purpose of animation. We can view these files as they are animated, save files, and later edit them. An important difference between these pictures and a strip of film is that with the Grafix Editor we can show the frames in any order we like, as many times as we like in that order. This means that identical frames do not need to be repeated by drawing them over; we only need to tell the program that they will be used over again in our sequence.

Three-colored frames can also be made by using an overlay technique supported by the Grafix Editor. This is particularly easy with figures that are only one frame tall. Although they are not directly supported by the Grafix Editor, larger frames for larger animated figures are possible.

Using The File Editor

Once you have created a series of files, you may want to append one file to another, cut out parts of a file, or reorder the frame sequence in a file. This helps with the organization of a program and it also helps to conserve program memory. These are exactly the functions of the File Editor. The File Editor will also display and create files of up to fifty frames, which comes in handy when trying to find exactly which frames are in some large file.

You might think of the File Editor as the cutting room for a film. After the film has been shot (i.e. the frames designed), it must be put together, certain parts must be cut out, and other parts repositioned. In addition to all of the various DOS commands, the File Editor has the ability to simultaneously display three 16-frame files plus part of a fourth.

It is a good idea to design each piece of animation separately with the Grafix Editor, rather than cramming each sixteen-frame file with as many figures as will fit. Keep these original animation designs on a separate disk from which you can take whichever figures suit your needs, and then design a custom file for your specific application. In this way you can build an organized animation library that you can refer to for animation segments.

Generally, this system almost totally does away with the need for graph paper, and bit-mapping by hand. However, there are times when graph paper is helpful, particularly when very large animated figures are required. Despite this small limitation, the pm ANIMATOR will improve the quality of your animation and reduce the time spent on designing animation by up to 95%. (For those of you who haven't done any sort of animation it will improve the quality of your animation by 100%!)

The next chapter discusses the features of the Grafix Editor in detail.

Chapter 8

Using The Grafix Editor

To use the pm ANIMATOR graphics system you must have at least 32K, a disk drive and a BASIC language cartridge. To boot the disk, insert the BASIC language cartridge into the (left) cartridge slot of your Atari computer. Next turn on the disk drive and insert the pm ANIMATOR master diskette. Turn on the computer and wait for the menu to boot in.

After several seconds, a menu with the pm ANIMATOR logo should appear. Under the logo are three selections: (1) Grafix Editor (2) File Editor and (3) Demonstration. Make a selection by pressing the number corresponding to your desired choice. To run the Grafix Editor press the '1' key on your keyboard. The screen will go blank momentarily and then come on again with the Grafix Editor's screen layout: a large square window in the upper left corner, a smaller window in the middle left, two greenish bars near the top middle, and a section on the right listing the editor's special commands. The large square window is the magnified editing window. It displays the figure currently being edited. To edit a 'frame' (an individual player shape) you move the cursor displayed in this window, using either the joystick or the keyboard. (This will be explained in detail later).

To the immediate right of the editing window is the actual frame area. It is here that the frame you are currently editing is displayed exactly as it will look when put into your program. The green bars border the frame that is currently being edited to remind you of which one it is. Each of the three windows that display the frames are independent of each other; they can each display a different frame or all can display the same frame. In this manner, several frames may be viewed simultaneously for comparison. The frame's number in the file (1 to 16) is displayed to the right of each window, along with the current player color value. The smaller window just below the large editing window is the animation window. Player animation is displayed here.

Editing Mode:

When the Grafix Editor loads in, it will be in the editing mode. Think of this as the drawing mode of the editor. Each frame you draw will be made up of large blocks, as seen in the large editing window. These look like dots in the actual frame, which is what they will look like when implemented in a program.

To draw and edit a frame, use the joystick to position the cursor inside the large editing window. The joystick will move the cursor in any of the standard eight directions within the border of the editing window. If the space the cursor occupies in the large editing window is black, pressing the joystick button will light up a block in that position. If the space the cursor occupies is already lit, then pressing the joystick button will erase it. If you wish to use the keyboard to edit your figures, then press the 'K' key. When using the keyboard to edit, use the four arrow keys (located near the {RETURN} key on the keyboard) to position the cursor, and use the space bar to light or erase the blocks. To jump back and forth between the joystick and the keyboard, hit the 'K' key again.

Command Mode:

To use the special functions of the Grafix Editor, press the letter keys that correspond to the first letter of the command you want. These special functions are provided to save time and expand the versatility of the Grafix Editor. When you select a command, one of three things will happen. Either:

(1) The function will be immediately carried out.

(2) A message will appear asking for additional required input (when appropriate, the range of acceptable values will also be displayed). Answer these prompts with a number value, a filename, or whatever input is requested. Single key input, such as a single number from 0 to 9, will not require that you press {RETURN}, while most other entries will.

(3) An error message will appear, and you will exit from the command mode back to the editing mode.

Commands:

ANIMATE: This function allows you to view animation in any sequence using the frames that are currently in memory. The first prompt will ask for the number of frames your sequence requires. The next prompt will ask you how many times you wish this sequence to be repeated. Do not confuse the number of frames needed with the number of steps in the animation. Notice that a sequence like: 1,2,3,4,3,2,3,4,4,4 takes 10 steps but requires only 4 separate frames. After the first two prompts have

been correctly answered with number input, a prompt will appear asking for the actual sequence. You will see:

SEQUENCE (<RETURN> FOR DEFAULT) HIT ANY OTHER KEY TO CONTINUE.

Defaulting will cause the program to automatically follow a sequence in numerical order (i.e. 1,2,3,4,5,6...) for as many frames as you indicated, and always starting from one. Pressing any other key will tell the program that you wish to enter your own sequence. It will then ask you for the number of the first frame in the sequence, then the second, then the third, and so on. After entering your sequence the program will ask you to choose a speed of animation. A zero will allow the fastest animation with only a very small pause between frames, while a speed of nine will be slow enough for you to examine the animation critically and see if it needs correction. The program will display the message "PROCESSING..." and then, after a few seconds, the animation will occur in the animation window.

COLOR: This command will allow you to change the color of any of the frames. This is useful when you wish to see if another color would be more suitable for your figure, or when using the MULTI function for three- colored players (see MULTI function and Multicolor animation section in Chapter 16). Note that the figure in the animation window will be the same color as the frame in window one. After you select this option the program will ask for the number of the window whose color you wish to change. After selecting this, every frame displayed in this window will be the color selected for the window. Next you must designate which method of color change you want. The cycle method lets you use the right and left arrow keys to change the color, and you see the colors as they change. When you are satisfied with the color, press <RETURN> to exit this command. With the normal method you merely designate the color and luminance for that window.

DATA: This command allows you to see the actual decimal or hexadecimal values for each row that make up the frame currently being edited. (See Chapters 3 and 4 for examples of such bit-map data.)

INVERSE: This function will light all of the unlit blocks and erase all of the lit blocks in the figure currently being edited, much like the Atari inverse video key does for text.

KEY/STICK: This command jumps between joystick and keyboard control of editing. With the keyboard the four arrow keys control the cursor and the space bar is used to draw and erase. The joystick uses all eight directions to move the cursor and the trigger button to light and erase blocks.

LOAD FILE: Use this function to load an already existing data file from a disk. After you select this command the program will request the file's name. Give the complete file name and hit <RETURN>. The file will load in and you will see the first three frames, one in each successive window. Editing will default to the first frame, in window one.

MULTI: This function is used to obtain a three-colored player by overlaying an existing frame of a different color on top of the frame in either window one or window two. To get three different colors, windows one and two must have different colors. A particular block in the frame will have the third color if the same block in both frames coincides (see the appendix on Multicolor animation and GPRIOR register).

The program will first ask for the number of the frame you wish to overlay, and then the window you wish to overlay it on. The overlaid frame will remain there unchanged until the MULTI function is again selected.

NEXT: This function allows you to choose the window whose contents you wish to edit. You may select from the three small windows to the immediate right of the large editing window. For example, if you were editing a figure in window 1, you could use the NEXT command to begin editing a figure in window 2 and leave your previous work behind in window 1. This is useful for jumping back and forth between frames for comparison.

PHOTOCOPY: This function will copy the picture from the frame you are *currently* editing to any other frame in the file. Notice that this will erase an old frame and replace it with the new one if an old frame occupies that space.

ROTATE TO: This function is used to start editing a new frame. It will not erase the old frame or change the window. For instance, if you were editing frame one in window 1, you could rotate to frame two, edit it (still in window 1), and then rotate back to frame one and resume work on it.

SAVE FILE: This function will save the file currently in memory as it is. The program will ask for a name for the new file and then save it to disk. Caution: using the same filename as another unlocked file on the disk will erase the old file and replace it with the new one.

UNDO: This function erases the current frame being edited and clears the editing window.

VOLUME: This function changes (or turns off) the volume of the various noises that this system makes, such as the noise the cursor makes when it moves.

WIDTH: This function displays all the frames as they would look in a different width by changing the player size registers. All frames will be the same width, including the ones displayed in the animation window. Size one is normal, size two is twice as large, and size three is quadruple size (see player width register). Note that although a size two player is 16 pixels wide, there are still only 8 bits of data defining the row of the player.

Z:OPTIONS: This function displays the submenu and allows you to use its DOS-like commands.

QUIT: This command exists the Grafix Editor and returns to the pm ANIMATOR selection menu.

Commands From Submenu:

These commands mostly deal with file manipulation and are like those you would use from the DOS MENU. Select these the same way you make selections from the main menu.

CATALOG: This command will display the filenames of the files currently on the disk. It is equivalent to the directory command in DOS.

DELETE: This command will delete a file, provided the file is not locked. The program will ask for the name of the file to be deleted. If you change your mind, do not enter a filename and just hit <RETURN>.

FORMAT: This command will format a disk (and wipe out any information you have stored on it). The program will ask if you really want to go through with the format process; if so, type 'Y' followed by a <RETURN>.

LOCK: This command locks a file so that it will not be accidentally erased. Caution: This command will not protect a disk from being FORMATTED, which will wipe out all of the data on the disk.

UNLOCK: This command unlocks a file that was previously locked so that it can be deleted or changed.

QUIT: This command exits the Grafix Editor and returns to the pm ANIMATOR selection menu.

RENAME: This command gives the designated file a new name you provide.

Z:OPTIONS: Returns you to the main Grafix Editor menu.

Chapter 9

Using The File Editor

To use the File Editor, select number 2 from the pm ANIMATOR menu. The screen will blank out momentarily while the program initializes. When the screen reappears it will be divided into a grid of five columns and ten rows. To the left is a list of the File Editor commands. Unlike the Grafix Editor, the File Editor only has one mode.

The columns of the grid are numbered 1 to 5 and the rows are numbered 0 to 9. When files are loaded into any of the five columns, any portion of the screen may be saved as a file. For instance, to append one file onto another you would first load in one file, then load in the next file where the last one left off, and finally save a file that is 32 frames long. To customize a file, parts of a new file can be substituted for parts of an old file. The catalog also contains many DOS commands for greater convenience.

Once the program is loaded from the menu and the screen is turned back on, it is ready for use.

COMMANDS:

CATALOG: This is the same as the catalog function on the Grafix Editor. It displays the filenames that are on the diskette and is the equivalent of the DOS directory command.

DELETE: See Grafix Editor "DELETE".

FORMAT: See Grafix Editor "FORMAT".

LOAD: The format for this command is different from the load command in the Grafix Editor. With this load, any part of a file can be loaded into any particular column and row. First, the program will ask for the name of the file from which you wish to load. Next, it will ask for the location on the grid where you want the first frame of the file to

be placed: first the column (1 5) then the row (0-9). Then it will ask how many frames you wish to load and the frame number from which to start loading. The load command should be used to edit and position files in a more desirable orientation. The results can then be saved.

MULTI: This command will allow you to view up to ten three-colored frames. When you design a set of three-colored frames, the MULTI function will enable you to see these frames combined. This function operates using columns 2 and 3, or 4 and 5, but not both at one time. To use it, position one file in column 2 (or 4) and the file that is to be another color in column 3 (or 5). The program will then ask for the color and luminance values for both columns and then automatically superimpose the second column over the first. When you are done looking at them, type <RETURN> and the columns will return to their normal places and colors.

It is easiest to use this function when the frames which are to be a certain color are all in one column and are separated from those that are to be the other color. (For more see Chapter 16).

DISPLAY DATA: This command will show you the decimal data of any frame on the screen, or of an entire column (selecting 10 will print the entire column). The prompt will first ask which column to print from. You may then specify a single frame by selecting a row (0-9), or ask for data from the whole column (10).

QUIT: This command will return you to the selection menu.

RENAME: See Grafix Editor "RENAME".

SAVE: This command will allow you to designate a starting point and a file length of up to fifty for the new file. After asking for a filename, the program will ask for the starting frame of the new file in the column-row format. Next it will ask for the length of the file from that point.

UNLOCK/LOCK: See Grafix Editor "UNLOCK/LOCK" functions.

Z:MOVE OR COPY: These two functions are used for moving individual frames to different areas on the grid in order to arrange and save a custom file. The copy function will put a duplicate of any frame onto the screen while the move function will move a frame from its present position to a new one. The prompts will ask you for the coordinates of the frame to move or copy and then the coordinates of its destination.

Chapter 10

Incorporating pm ANIMATOR Animation Into Your Basic Programs

Part I – The USR Statement in BASIC

After a little practice, there is no reason that creative animation should be difficult to implement in your BASIC program. If you have found that using pm ANIMATOR's editors is easy, you should find the supplied routines just as easy to use. This section will explain the steps necessary to set up player-missile graphics and to use all the subroutines for animation that will make PMG faster and easier to use. In addition, there is a reference section describing the function of each subroutine, its limits, the format of the USR statement needed to use it, and much of the source code.

Fast and clean player-missile animation cannot be produced from 'pure' BASIC simply because of the slowness of the language. However, do not be alarmed if you have had no background with machine language. All of the machine language routines are provided as well as full instructions on how to use them. To use these routines proficiently requires that we understand the BASIC USR command.

Although BASIC's slow speed of operation makes it unsuitable for most high speed graphics purposes, its ease of use makes it attractive to the beginning programmer. To overcome this problem of speed, many programmers resort to the machine language subroutine, as opposed to actually programming in assembly language. A machine language subroutine is a piece of machine language code designed by a programmer to perform a specific task in machine language, thus speeding up the execution time for this task. Machine language subroutines are accessed via the BASIC USR command, which in effect tells the computer to jump to the desired machine language routine, execute it and return to BASIC. A USR statement has the following form:

A=USR(ADDR,A1,A2)

where ADDR is the address of the machine language routine in memory, and A1 (argument 1) and A2 (argument 2) are defined by the actual machine language routine. In fact, you can have as many of these parameters as you want *provided the machine language subroutine allows for them.*

Note: An 'argument' is simply a real number expressed in any form. It may be expressed in variables, constants, operations (+,-,*,/) or specific BASIC commands.

A USR command is always used in conjunction with an existing machine language subroutine. Suppose a USR command of the above form will be used with a routine whose function is to clear an area of memory very quickly. This is especially useful with PMG because 'garbage' (unwanted residual data) often clutters up the visible player areas (on the screen this will look like random explosions throughout the player area!). The machine language routine will reside at the location stored in the variable ADDR of our USR command. The first argument in this USR command will specify the exact starting place in memory we wish to clear out, the second argument will specify how many memory locations after the first are to be cleared. For purposes of comparison, here is how this would be done with a BASIC subroutine:

```
5 REM BASIC 'CLEAR' ROUTINE
10 FOR I = ADDRESS1 TO ADDRESS2
20 POKE I,0
30 NEXT I
```

This routine will work well enough, but if we have to clear thousands of locations in memory, this routine will take a few minutes to execute. However, if we were to perform this task from machine language, the screen would be almost instantaneously cleared. Our machine language routine will be executed when the following command is encountered in our BASIC program:

A=USR(ADDR,PMB+4*256,4*256)

ADDR = memory location where machine language clear routine starts.

PMB = the address of our player-missile base (see Chapters 4 or Appendix I). Therefore, PMB+4*256 will be the address of the beginning of player zero.

4*256= the number of memory locations to clear. It is expressed this way because there are four player areas, each 256 bytes long (in single line resolution).

The A in this USR command acts as a dummy variable (that is, it serves no function but is required by BASIC syntax).

To use a machine language subroutine from BASIC we must first put it somewhere in memory. We must choose a location the computer does not use for some other purpose so that the routine will be safe from tampering. Usually the best way to accomplish this is to store it in a string, because strings are stored in memory in a location where the computer will not accidentally erase them. To store a machine language routine as a string, first obtain the decimal values of the actual machine language coding. Then use the CHR$ command to convert these numbers into characters, and assemble the characters into a string. There is one problem, though: so far we have no way of knowing where in memory the computer is storing our string, and we need this information to access the routine. Furthermore, strings are occasionally moved from one place to another. To determine where in memory a string is stored, use the BASIC ADR command, which gives the numerical address of the string's starting place (in this case, the starting place of our machine language routine; the computer does not distinguish between numbers stored in memory that simply represent characters in a string and those that represent machine language coding). Then you can access the routine. Here is a sample BASIC routine that sets up a machine language routine like the one discussed above for clearing a large area of memory. This BASIC program obtains the machine language code from DATA statements, stores the routine as a string, and accesses the routine with a USR statement. It is just like the BASIC program you would write to set up the actual memory-clearing routine, except that, for brevity, the routine's long machine language coding has been replaced by a short list of numbers.

```
10 DIM MACH$(10):FOR I=1 TO 10:REM MACH$ IS THE ROUTINE.
20 READ A:MACH$(I)=CHR$(A):REM CONVERT VALUES TO CHARACTERS.
30 NEXT I: Z = 256
40 DATA 104,11,23,45,44,34,56,112,152,96:REM THIS DATA IS THE MACHINE
LANGUAGE CODE (THIS IS ONLY SIMULATED DATA FOR THE PURPOSE OF
ILLUSTRATION AND IS NOT REAL MACHINE LANGUAGE CODING).
50 XX=USR(ADR(MACH$),1536,Z): REM EXECUTIVE MACHINE LANGUAGE
SUBROUTINE.
```

Note the first expression in the USR command (ADR(MACH$)); this is the starting place of MACH$ in memory.

To use the pm ANIMATOR routines you only need to memorize the format of the USR statement, and the various customized formats for each subroutine (these are fully described in chapter 11). With these statements we can, in a sense, add commands to ATARI BASIC to make it a more powerful language.

Part II - Using The TOTAL.LST Subroutine

Note: This section gives a brief overview and a listing of the entire package of machine language subroutines. Part 3 of this chapter covers the details of how to produce various types of motion and animation. The complete description of each machine language subroutine, its format, and how to use it may be found in Chapter 11.

TOTAL.LST is a self-supportive subroutine; all you need to do is to put your code in between lines 10 and 31000 and TOTAL.LST does the rest. In about 7 seconds it sets up a vertical motion register, an animation routine, a fast clearing routine, a routine to load files from disk (pm Animator data files as well as other ATASCII stored information), and also the things that the computer itself needs to run player missile graphics. After this routine executes, all you need to do to have a fully-prepared PMG system is to load a pm Animator data file from the disk, read one into memory from DATA statements, or get the data into the program in some other way.

There are a few things you must keep in mind after TOTAL.LST has initialized:

(1) Any GRAPHICS command given after TOTAL.LST has initialized will cause the player areas to go haywire. If you need to switch graphics modes, do it before TOTAL.LST initializes. If this isn't possible, remove the players from the visible portion of the screen (by way of horizontal position registers) and reinitialize TOTAL.LST. (The only things that need to be redone are the steps required by the computer itself and the steps that put the player data into the player area (See Chapter 5).)

(2) A large part of 'page six' (locations 1536 to 1791 in memory) is used for the vertical blank routine. Therefore only the upper part of this page of memory may be used for your own purposes. (See Chapter 11 for exact used and unused locations). It is important that the vertical blank routine in page six remain undisturbed. If the vertical blank routine is disturbed in any way, the computer will probably lock up and will have to be turned off and restarted.

Here is a complete listing of the TOTAL.LST file. Important: this listing is different from the actual version on the disk. Although both will perform the same tasks, the disk version sets up faster and uses less memory. The major difference is that the machine language routines are read from data statements instead of from their character string form. This printed version was designed for clarity and ease of printing.

TOTAL.LST Listing

```
10 GOSUB 32025
20 STOP : REM REMOVE THIS LINE
30999 REM VERTICAL BLANK DATA
31000 DATA 162,3,189,244,6,240,89,56,221,240,6,240,83,141,254,6,106,141,255,6,142,253,6,24,169,0,109
31005 DATA
253,6,24,109,252,6,133,204,133,206,189,240,6,133,203,173,254,6,133,205,189,248,6,170,232,46
31010 DATA
255,6,144,16,168,177,203,145,205,169,0,145,203,136,202,208,244,76,216,6,160,0,177,203,145,205
31015 DATA 169,0,145,203,200,202,208,244,174,253,6,173,254,6,157,240,6,202,48,3,76,131,6,76,98,228
31020 DATA 104,169,7,162,6,160,129,32,92,228,96
31030 REM DATA FOR CLEAR ROUTINE (NONE OF THE FOLLOWING LINES THAT CONTAIN DATA STATEMENTS ARE IN THE ACTUAL VERSION OF TOTAL.LST)
```

```
31040 DATA 104,104,133,209,104,133,208,104,133,211,104,133,210,24,
165,210,101,208,133,210,165,211,101,209,133,211
31050 DATA 169,0,168,145,208,166,208,228,210,208,7,166,209,228,211,208,1,96,230,208,208,2,
230,209,162,1,208,231,65
31060 REM DATA FOR ANIMATION (ROT$)
31070 DATA
104,104,133,204,104,133,203,104,133,207,104,133,206,160,0,177,206,145,203,200,192,16,208,247,96
31080 REM DATA FOR MACHINE LANGUAGE LOAD ROUTINE (LD$)
31090 DATA 104,104,104,170,104,157,69,3,104,157,68,3,104,157,73,3
31095 DATA 104, 157, 72,3,32,86,228,192,0,48,1,96,132,203,96
32025 DIM CLER$(54),ROT$(25),LD$(31),SET$(11),NAME$(15):REM DIMENSION STRINGS FOR
MACHINE LANGUAGE ROUTINES
32026 REM LINES 32030 – 32040 APPEAR DIFFERENTLY HERE THAN IN THE ACTUAL TOTAL.LST.
32030 RESTORE 31040:A=ADR(CLER$) :FOR I=A TO A+53:READ I2: POKE I,I2:NEXT I
32035 A=ADR(ROT$) :FOR I=A TO A+24: READ I2:POKE I, I2:NEXT I
32040 A=ADR(LD$):FOR I=A TO A+30:READ I2:POKE I,I2:NEXT I
32045 PM=PEEK(106) – 16:PMB=256xPM:REM THIS FINDS A PLACE FOR TOTAL.LST. THESE TWO
VARIABLES (PM,PMB) ARE VERY IMPORTANT. PM= PAGE NUMBER (HI BYTE) OF PLAYER-MISSILE
BASE WHILE PMB= EXACT LOCATION IN MEMORY OF THE PLAYER-MISSILE BASE.
32050 HP=53248:VP=1780:LE=1784:POKE 559,62:POKE 623,1:POKE 1788,PM+4:POKE 53277,
3:POKE 54279,PM:POKE 704,110
32051 REM LINE 32050 SETS UP THE VARIOUS PARAMETERS NEEDED FOR PLAYER-MISSILE
GRAPHICS (DISCUSSED IN CH. 6 OF TUTORIAL).
32055 FOR I=0 TO 3:POKE LE+I,16:NEXT I:REM SET UP OUR LENGTH REGISTERS (CREATED BY
VERTICAL BLANK ROUTINE, SEE CH. 5).
32060 RESTORE 31000
32061 REM LINE 32065 SETS UP THE VERTICAL BLANK IN THE UPPER HALF OF PAGE SIX. SET$ IS ONLY
USED ONCE, IT GETS THE VERTICAL BLANK ROUTINE STARTED.
32065 FOR I=1665 TO 1769:READ A:POKE I,A:NEXT I:FOR I=1 TO 11:READ A:SET$(I)=CHR$(A): NEXT I: RESTORE
32070 ZZ=USR(ADR(SET$)): REM FIRE UP THE VERTICAL BLANK ROUTINE.
32075 I2=0:DIM PNT(16):FOR I=PMB TO PMB+254 STEP 16: I2= I2+1:PNT(12)=I:NEXT I:REM THIS
LINE SETS UP A CRUDE POINTER SYSTEM (16 FRAMES 16 LINES HIGH). IT MAY BE REMOVED AND
REPLACED WITH ANOTHER POINTER SYSTEM(S).
32076 REM THIS IS THE MACHINE LANGUAGE LOAD ROUTINE (LINES 32080 AND 32085 ARE ONLY
USED ONCE, THE ROUTINE STARTS AT 32090.)
32080 START= PM*256:LOD=32090:NUM=256
32085 RETURN
32090 OPEN #1,4,0,NAME$:REM YOUR FILENAME (I.E. NAME$="D:RUNNPUSH).
32095 GET #1,A
32100 POKE START, A:START=START+1:NUM=NUM-1
32105 XX=USR(ADR(LD$),16,START,NUM):REM LOAD THE FILE INTO RAM.
32110 CLOSE #1:RETURN
```

Part III - Producing Motion and Animation With the TOTAL.LST Subroutine

Note: Particulars such as the size, color, and priority of players are not discussed here as they are discussed in Appendix I.

HORIZONTAL AND VERTICAL MOTION:

Horizontal motion of any of the players is achieved by poking the horizontal positioning shadow register with the value corresponding to the position on the screen where you wish to place the player. To simplify matters, TOTAL.LST has assigned the horizontal position register a variable name: HP. To move player 0 to position 100, you would POKE HP, 100. To move player 1 to position 100, you would POKE HP+1,100. Since the horizontal register is a write-only register (you can't find its value by PEEKing into its location), you should use a variable in your own program to update the position of your player; this way you will know where it is on the screen.

Vertical motion of players, which is fairly complicated to do without pm ANIMATOR, is made easy by TOTAL.LST's vertical blank routine. This routine creates a simulated vertical position register for each player comparable to the horizontal position registers built into the Atari. It gives you a single location into which to POKE new values, making vertical motion as simple as horizontal motion. The important thing to remember about the vertical blank routine is that you need only one USR statement to set it up; you don't need a separate USR statement every time a vertical move must be performed. This is made possible by the use of the vertical blank interrupt (VBI), a function of the Atari operating system that can run a small machine language program automatically every 60th of a second. In this case the machine language program being run every 60th of a second is the VBLANK routine, which moves the player data up and down in memory to produce vertical motion of the player. Each time it is run, the routine moves the player to the vertical position whose value is stored in the memory location VP (actually VP+P, where P is the number of the player, as will be seen below). If you POKE a value into location VP, the next time the VBLANK routine is executed it will move the player to that location on the screen. Thus a software 'register' has been created whose memory location is represented by the variable VP.

To use this vertical position register, first set up the vertical blank routine by loading it and doing a USR command. Then specify the height, in scan lines, of each of the players you will use, and POKE this value into the memory location represented by the variable LE. Each player can have a height of up to 256 scan lines. For player n, POKE the height into location LE+n. To assign player 0 a length of sixteen scan lines, for example, POKE LE,16. To give player 3 a length of 20 scan lines, POKE LE+3,20. Next, choose the vertical position where you want the top of the player to first appear and POKE that value into the vertical position register. The vertical position value can be any number from 0 to 255, although not all these positions are visible on the screen because of overscan. For player p, POKE the position value into memory location VP+p. So to start the top of player 0 at 100, POKE VP,10, and to start the top of player 3 at 219, POKE VP+3,219. Now load the actual player data (the player image) into the player area of memory, making sure that the first byte of data is POKEd into the player's current vertical position, as stored in VP. (Recall that the value in VP is the current

location of the top of the player, and the value in LE determines how far down the player extends. Thus, if the player data starts above or below the VP location, some of it will be outside the boundaries defined by VP and LE, and that part of the player image will not be moved by the vertical blank routine.) The best way to make sure your first byte of data is going into the VP location is to POKE a 0 into VP and then load the image into the top of memory for the player you want (player 0 memory, for example) starting at the top. However, you can also insure that your first byte goes into the right place by POKEing the player data into the player memory location plus a vertical offset. For instance, if the player's vertical position on the screen is to be 100 (approximately the middle), you could start loading your data into player 0 by POKEing the first byte into PMB+1025+100. Your routine to load in your data might look something like this:

```
10 POKE VP,100 : FOR I=PMB+1025+100 TO PMB+1025+116: REM PMB IS
THE PLAYER-MISSILE BASE. THIS LINE ALONG WITH LINE 20 WILL PLACE THE
PLAYER DATA INTO PLAYER ZERO, IN THE MIDDLE OF THE SCREEN.
20 READ A : POKE I,A:REM THIS ROUTINE USES DATA STATEMENTS ONLY
FOR ILLUSTRATION. NORMALLY THESE VALUES WOULD BE STORED IN
MEMORY OR IN STRINGS AND THE IMAGE WOULD BE PLACED BY USING ROT$
(See Chapter 11 on ROT$).
30 NEXT I: DATA 1,2,3,4,3,2,1,2,2,22
```

Now the vertical position register for your player is working, and you can move the player image vertically by simply POKEing a new value into VP+n. For example, to move player 2 to position 56 on the screen, POKE VP+2,56.

ANIMATION:

To incorporate animation into a program we must use a USR statement and have a basic understanding on how the routine works.

The type of animation routine in TOTAL.LST is called a 'MOVE' routine and it does what its name says: it moves data from one place to another. Actually, it duplicates it from one place to another, as the data that is moved is still in its original place as well as its desired destination. In the case of player-missile animation, the routine moves frame data from their protected area in memory to the *exact* position of the desired player in memory. The routine is set up to move only sixteen separate data items – a Grafix Editor frame of standard height. By moving this data from the protected area to a player area that is visible on the screen we can produce animation. All that is necessary is a FOR-NEXT loop with a USR statement that uses ROT$ sandwiched between the FOR and the NEXT. Although this routine is set up to move only sixteen bytes of data, it can be modified to move more. (This and the format of ROT$ are explained in Chapter 11.)

CLEARING OUT AN AREA FAST

Many times it is necessary to clear out a player-missile area because it is littered with 'garbage' (residual data that still occupies an area in memory). This can be

accomplished with CLER$ which will clear out any number of memory places. So be careful: it will even clear itself out if told to do so.

LOADING A FILE FAST WITH LD$

After the subroutine TOTAL.LST has initialized, it will have set up a machine language load that will load ATASCII files directly into memory. This is very convenient since pm Animator files are in ATASCII. For more information on this, run the demonstration programs and see Chapter 11.

LARGE PLAYERS USING SUMOVE$

Although each player has a set horizontal resolution of eight bits, players can be combined to give increased resolution. To do this you simply place the players side by side to form one large 'player'. This works nicely until movement or animation of any type is required. Even a move with our existing machine language routines is jerky. This is because of the interval between each successive BASIC command. To create smooth multiplayer motion and animation we must use the routine in SUMOVE$. (This is a file on the disk that should be ENTERed into TOTAL.LST on a separate disk.)

The new routine is nice in that it uses the already existing machine language routines to perform its own functions. First it loads the horizontal registers with new values, then the simulated vertical registers, and then it does the animation. The reason that the motion is smooth is that all of these changes are done almost simultaneously (in contrast with BASIC).

Before using SUMOVE$ you must specify the height of each player to be animated (similar to changing the number of scan lines in ROT$).

Another important thing to remember is that SUMOVE$ always moves all of the players and animates all of them. With this in mind, you must sometimes use dummy values and constants in the USR function when you only want two or three players (see DRAGON.DEM).

Chapter 11

Description and Format of Machine Language Routines in TOTAL.LST

This chapter is designed to teach the uses of the machine language routines in TOTAL.LST as well as to be a quick reference section for the format of each routine. The routines in the subroutine "D:TOTAL.LST" are installed by including TOTAL.LST in your BASIC program. To use TOTAL.LST, simply use the command: ENTER"D:TOTAL.LST". The last (and hardest) step is to write your program between lines 10 and 31000 (where TOTAL.LST begins).

We can best understand how and why these routines are used by thinking of each of them as an 'additional' BASIC command. For example, if ATARI BASIC had no For-Next statement, we could simulate the statement by using other BASIC commands to build a subroutine. Consider the next few lines of code:

```
10 FOR I=ADDR1 TO ADDR2
20 POKE I,INT(RND(0)*254+1)
30 NEXT I
40 REM THIS PROGRAM POKES RANDOM NUMBERS INTO ALL MEMORY
LOCATIONS BETWEEN ADDR1 AND ADDR2.
50 REM FEEL FREE TO USE THIS ABSOLUTELY USELESS SUBROUTINE IN
YOUR OWN BASIC PROGRAMS!
```

The above program would load a random value from 1 to 255 into all of the memory locations between ADDR1 and ADDR2. If we had no For-Next statement in ATARI BASIC we could duplicate this program in this fashion.

```
10 I = ADDR1
20 I=I+1
30 POKE I,INT(RND(0)*254+1)
40 IF ADDR1=ADDR2 THEN END
50 GOTO 20
```

This program accomplishes the exact same thing as the previous one only without the use of the For-Next statement.

It would be a real advantage if ATARI BASIC included commands like ANIMATE, ROTATE, and MOVEFIGURE. However, since ATARI BASIC has no such commands, we have augmented it with the next best thing: machine language subroutines.

The following is a detailed description of each routine, its function and format, and special notes on each routine.

Name of routine: VBLANK – Vertical Blank Routine

Function: Moves any of the four players vertically via a single fixed register (VP). This is similar in effect to the horizontal position register but is for vertical motion.

Format: POKE VP+P,Y1

POKE = a BASIC command used to change the value of a memory location.

VP = the memory location of the first vertical position register. This is created by the Vblank routine in TOTAL.LST and is not a feature inehrent to the ATARI computer! VP's actual decimal value is 1780; it is represented by this variable so you will not have to remember the number.

+P = the variable P is the player number (0-3). The vertical position registers are at locations 1780 to 1783, so we could use the actual location instead of its variable representation (VP+P), but it is easier to remember VP+2 (player two's vertical register) than location 1782.

Y1 = the intended vertical position of the player.

Example: POKE VP+1,102

As indicated above, VP+1 is location 1781 in memory, player one's vertical position register. This example would instantly move player one to the vertical position of 102 on the screen. It is important to remember that vertical position is measured from the top of the player.

Special notes: This routine is set up automatically by including TOTAL.LST in your program. The machine language routine is set up so that it automatically runs once every 60th of a second. This explains why our command to move a player vertically has no USR statement in it (i.e. POKE VP+1,102).

When animating a player, make sure that the data (no matter how it is being put into the player area) is being placed starting from VP+P. That is, the vertical blank routine cannot tell the difference between a blank player area and one with important data in

it, so it will move the data from the old vertical position (in memory location VP+P) to the new position whether it is blank or not. To make sure that the frame data is in the right place, you can use this statement in conjunction with ROT$ (explained later):

A=USR(ADR(ROT$),PMB+1025+PEEK(VP),20000)

Notice that the first argument contains the BASIC PEEK command to find out the vertical position of the player (for more on this see the explanation of ROT$ and the demonstration programs).

It should also be noted that the height of a player can be adjusted. If you wish to have a player of a different height, it can be accomplished by POKEing to another location. These locations are 1784-1787 and correspond to each of the four players. Currently they are set up for sixteen scan lines (16 memory locations, 16 blocks high), but they can be changed like this:

CHANGING A PLAYER'S HEIGHT

POKE LE+P,HI

LE = location 1784, the height register used by Vblank.

P = the player number whose height you desire to change. Again, these variables may be replaced with the appropriate constants.

HI = the desired height in scan lines (blocks) of the player.

A real example of this might be POKE LE+2,20. This would change player two's height to 20.

This command to change a player's height is only half of the job. You must also design a player twenty lines high instead of sixteen.

The vertical blank routine uses memory locations 1665 to 1791. In addition to these 'page six' locations, it also uses 'zero page' locations 203 to 207 (locations $CB to $CE). Do not use these locations as it may cause the routine to malfunction.

Name: ROT$

Function: This machine language string is the basis for all animation in this package. It is a simple 'move' routine, and it does exactly that. It moves data from one place to another. More precisely, this routine will copy data (it will not disturb this data, but merely duplicate it somewhere else) from a protected area into an area where it will be visible. In this way we can store an entire file in memory somewhere and 'copy' each frame into the desired player area. By continually copying one new frame after another into the player area we achieve animation.

Format: A=USR(ADR(ROT$),PADR,DAT)

ADR(ROT$) = the beginning of the string ROT$ which is a machine language routine.

PADR = the exact address in memory to which you wish the data to be copied. This will be a visible player area (explained in the example below).

DAT = the exact address in memory from which you are copying data. To better organize these addresses, a pointer system is required (see Appendix VI).

Example: A=USR(ADR(ROT$),PMB+1024+Y,PNT(1))

PMB+1024+Y = the area we are copying the data to. In this case we are copying the data to player zero. PMB is a variable used in TOTAL.LST that holds the memory location of PMBASE (see Chapter 5 or Appendix I). The 1024 is added to get past the unused RAM area immediately below PMBASE; you may remember this from the single line resolution memory map. The added Y variable is the vertical position at which to draw the player on the screen. These variables could just as well be replaced with constants; however, it is easier to remember the variables instead of numbers.

PNT(1) = the item in the array PNT that is the memory location of the data to be copied. PNT has been dimensioned ahead of time; you will notice that TOTAL.LST sets up an array by this name and holds the memory location for the data to be copied. Arrays are convenient for holding memory locations because you don't have to remember the memory location, just whether it is the first item in the array, or the second, and so on. (See Chapter 16 on pointer systems).

The above statement, provided the variables were given legitimate values, would copy sixteen memory locations (the data for one frame) into player zero's area starting from the location given by the value of PNT(1). The value of the variable Y determines exactly how high or low on the screen the frame is positioned. It will only do this once; in order to produce animation, you must repeatedly use this machine language routine and change the data to be copied. To do this, change the second argument in the USR statement (PNT(1)) to the memory location of the next frame to be displayed. (For more on this, see the animation demos on the disk.)

Notes: Since this USR routine allows you to give it any memory locations you wish, it is highly flexible. With this in mind, there are several things you must remember when using it. First, remember to use variables and arrays wherever possible. This way you will not be confused by numerous and nearly meaningless constants and you will be free to concentrate on your program. This also helps when debugging as you do not need to check every single constant. Also remember that if your player moves vertically you must take this into account in your USR statement, when you indicate where in memory ROT$ should copy the data. For instance, if a player moves towards the top of the screen it is moving down in memory. This change in memory locations must be accounted for (see demos on disk).

As it stands now, this routine is set up to copy the contents of sixteen consecutive memory locations. If you wish to change this number to something else, include the following somewhere in your program:

ROT$(22,22)=CHR$(X)

X = number of data items to be copied (height of a single frame).

This command will change the ROT$ machine language routine so that it will copy more or less data, allowing for animation with players larger than sixteen frames. The variable X can be any number between 1 and 255. For instance, if you wanted to use frames that were 32 blocks high (height of two normal frames), you would include this statement in your program:

ROT$(22,22)=CHR$(32)

however, SUMOVE$ does it much more smoothly. It not only does the animation for each player at the same time, but it also moves the players horizontally and vertically at the same time.

Format: A=USR(ADR(SUMOVE$),P0,D0,P1,D1,P2,D2,P3,D3,X0,X1,X2,X3,Y0,Y1,Y2,Y3)

ADR(SUMOVE$) = starting address for machine language subroutine SUMOVE$.
P0-P3 = the exact locations to copy the frame data to.
D0-D3 = the exact memory locations from which to copy data.
X0-X3 = new horizontal positions.
Y0-Y3 = new vertical positions.

Since the USR statement is so long, it is necessary to use several one letter variables to fit the statement on one line. (See KNIGHT.DEM and DRAGON.DEM.)

Each player's frame height may be changed independently. Frame height here is changed for the same reason that it was in ROT$: to produce animation with larger size frames. To do this, use the same type of command you used to achieve this in ROT$. To change the routine so it would animate frames of 32 lines high you would do this:

SUMOVE$(22,22)=CHR$(32):REM changes height of frames for animation for player zero.

SUMOVE$(45,45)=CHR$(32):REM changes height of frames for animation for player one.

SUMOVE$(68,68)=CHR$(32):REM changes height of frames for animation for player two.

SUMOVE$(91,91)=CHR$(32):REM changes height of frames for animation for player three.

This would set up the routine so that each of the four player's frames used in animation would be 32 blocks tall (the height of two normal frames). Also, be aware that this routine uses the vertical position register. This means that whatever was necessary to set up for the vertical blank routine (player height, etc.) is also necessary here if you wish to use vertical motion.

Note: This routine is on a separate file on the pm ANIMATOR disk. The file "D:SUMOVE.LST" contains the entire routine as well as the means to set it up. To use this file, merely install it in TOTAL.LST by using the ENTER command. First ENTER "D:TOTAL.LST"then ENTER "D:SUMOVE.LST". SUMOVE.LST is already set up to be merged with TOTAL.LST.

This routine is undoubtedly the most complicated to use. With this in mind, you should not attempt to use it until you have a full understanding of the other routines and how player-missile graphics works.

For more information on this, you are encouraged to examine the various demonstration programs on this disk.

Name: High-speed clearing routine - CLER$

Function: This routine clears out an area of memory extremely fast. It is most useful for clearing garbage (residual data) from a player area. If the animation quickly changes, such as in the main demo for this program, it is necessary to clear the previous data from the player area.

Format: A=USR(ADR(CLER$),ADDR1,BYTE)
ADR(CLER$) = The address of CLER$, the machine language subroutine.
ADDR1 = the memory location at which clearing starts.
BYTE = the number of memory locations to be cleared.

Example: A=USR(ADR(CLER$),PMB+1025,256)
PMB+1025 = the memory location to start clearing from; in this case it is the address of player zero, since PMB = the memory location of PMBASE.
This routine will clear 256 memory locations starting at the beginning of player zero's area. It will clear out any data that was previously in player zero.

Name: Fast File Loader - LD$.

Function: To load any ATASCII file, such as a pm ANIMATOR data file, from the disk into memory.

Format: NAME$="D:FILENAME":NUM=number of bytes:START=starting address: GOSUB LOD
"D:FILENAME" = the name of your file along with the device and colon ("D:").
NUM = the numebr of bytes of data in the file to be loaded. For a normal file this would be the number of frames times sixteen. This can be thought of as the number of memory locations the file will occupy in memory.
START = the exact starting location in memory to load the file into.
GOSUB LOD = this command runs the loading subroutine with the values and filename you provide it with.

Example: NAME$="D:RUNNER":NUM=96:START=PMB:GOSUB LOD
This example will load the file "D:RUNNER", which is 96 bytes long, into the area just below PMB (PMBASE). In this case, the loaded file will occupy memory locations PMB to PMB+95.

Name: SUPERMOVE ROUTINE - SUMOVE$

Function: To produce any type of multiplayer animation in which several players must be combined to form a large one. This can be accomplished using the other routines;

Chapter 12

About The Various Demonstration Programs on the Disk

Although this utility is entirely menu driven, there are several demos and various subroutines that are not accessible via the menu. These routines are transferable and should be copied to another disk. All of the demonstration programs are in the 'SAVE' format, while the utilities are in the 'LIST' format. To use the demonstration programs, make sure the BASIC cartridge is in, insert a diskette with DOS on it into the drive, and turn on the computer. Then put in the pm ANIMATOR master diskette. (Booting the pm ANIMATOR master diskette will load in the main menu.) Next, run the demonstration by typing:

RUN "D:filename"

where "filename" is the name of the demonstration. These demonstration programs will give practical examples of animation and motion from the very basic to the very complex (e.g. multiplayer, multicolor animation) using the routines supplied with this system. The demonstration programs are listed below with a short description of what each one demonstrates.

The utilities in the package are in the 'LIST' format to allow you to enter them into your programs directly. All of these routines are integrated into a system in "TOTAL.LST" which will set itself up first and can be easily integrated into your own programs (for more on this see chapters 10 and 11). This small subroutine takes less than ten seconds to set up the player area, initialize the machine language subroutines, completely clear the player area, set up a pointer system, and so forth.

DEMONSTRATION PROGRAMS:

All of these demonstration programs have REMark statements to improve readability. However, lines 50,55,56, and 60 are repeated in each demonstration and are

not REMarked. Instead, they are explained here. Line 50 determines the place the player-missile base will be located, lines 55 and 56 set up a clearing routine and clear the 'garbage' (previous data) from the player-missile area, and line 60 sets up the various player-missile parameters necessary for operation. For more information see the section on implementation.

IMPORTANT: Most of these demonstrations require some additional data files from the disk containing the animation frames themselves.

(1)MOTION.DEM: This demonstrates motion routines and should be run while reading the section on motion routines.

(2)HELICOPT.DEM: This demonstrates the helicopter example discussed in Chapter 13.

(3)RUN1.DEM: This demonstrates how to use the routine ROT$ to do quick animation, and how to load a file into memory very quickly with the LD$ routine.

(4)VERT.DEM: This demonstrates how to set up and use the vertical blank routine in TOTAL.LST that makes vertical motion quick and easy.

(5)RUN2.DEM: This demonstrates simultaneous horizontal and vertical motion as well as animation.

(6)DRAGON.DEM: This demonstrates the use of SUMOVE$ to create a large moving animated figure.

(7)KNIGHT.DEM: This demonstrates the use of SUMOVE$ for multicolor player animation.

ABOUT THE MAIN DEMO AND AWESOME

Creating flashy animation scenes like those in the main demonstration is really not that difficult after you are familiar with the various routines and the formats for using them. First you should practice doing small things – just to become familiar with the included routines – then work up to the bigger animation, and eventually up to a full-sized program, like a game.

The most importat step is to have a plan. Decide exactly what you want to do and sketch it out on paper. If what you are doing requires a complicated playfield, such as a modified display list, it is a good idea to draw the playfield on a piece of graph paper. After making up this plan you can start designing the actual players.

Sometimes players will be so large that you will need to sketch them out on graph paper and cut them up into 16X8 blocks to be designed with the editor. Even though this takes longer than designing a small player, it is still easier than bit-mapping the player by hand.

Remember that an ounce of ingenuity is worth a pound of programming knowledge (ancient Silicon Valley proverb). The pm ANIMATOR programmer thought of at least five ways to make the earth turn in the demonstration, all of which would have taken well over 48 hours of programming time! But, with a little ingenuity, he thought of a way to make a rotating earth that took only about three hours, from the design time to the actual implementation! In fact, the earth was probably one of the simplest pieces of animation in the whole demonstration. We won't give away the secret of the rotating earth, but you are free to examine the program. The earth demonstration bears the filename "AWESOME", but you may no longer agree after you figure out how easily it's done (hint: it has something to do with priority). The machine language code in P$ is a simple horizontal move that uses all of the players and all of the missiles and its format is: AA=USR(ADR(P$),MISSILES,P0,P1,P2,P3), where the arguments represent the intended horizontal position.

SECTION III
Advanced Animation Techniques

Chapter 13

Creating Animation
Section I: An Overview

Whether a program achieves the right effect will be determined in large part by the quality of its graphic presentation. With this in mind, the design of the animation is often just as important as the ability to implement animation into a program. Once you understand the various routines included with pm ANIMATOR and how player-missile animation works in general, you are ready to concentrate on the quality of the animation you produce. This section provides many guidelines and hints for creating the best possible animation.

The first step is to decide what the figure will be – not necessarily how it will look but what subject it will represent. This may seem to be obvious, but it is important to start at the beginning.

Once you have chosen the subject, you must decide upon the 'tone' your figure should have. For instance, if a comical figure is desired such as the wild dancer in the demonstration, then the movements can be jerky and somewhat random. The jerky motions of the dancer are not realistic; however, the tone of the dancer was not intended to be realistic. On the other hand, the small runner was designed to set a 'realistic tone', so its movements are quite realistic. Animation that is realistic takes considerably more effort to produce than other types of animation. When trying to create realistic animation, you should have pictures of the object that you are trying to create.

After you decide upon the tone of the animation, you must decide the size, color and resolution of the figure (does it use more than one player?). If the figure is to be two or more players wide or in three colors, you must allow for this initially or it will cause problems later on.

After all of these steps have been completed, you are ready to create the actual animation.

Simple Animation

The simplest form of animation is 'two-step' animation – animation composed of only two different pictures. Two-step animation is often used to give the illusion of blinding speed and also to simplify the program it is in. Some examples of this type of animation might be: a "Pac-Man" figure, a bird beating its wings, a crab opening and closing its claws, etc. Usually, small objects that require fast open-close, up-down, or side-to-side types of motion should have this form of animation.

When designing a figure of this type, one frame should be in strong contrast with the other frame, or else the animation may not be noticeable. For instance, a jumping frog should be very compact in the first frame with its legs tucked in, and in the next frame its legs should be fully extended. This would make the frog appear to jump very quickly. It is important to note that two-step animation can be very impressive if it is combined with a good motion routine. For instance, a spring that compressed when it hit the ground and then stretched out as it left (in two steps) would be impressive if the motion routine simulated a realistic bouncing spring, subject to gravity, loss of momentum, etc. (See routine to make ball bounce in DEMOTION and also in Chapter 14).

Although this type of animation is the easiest to perform, if done well it can give outstanding results.

Advanced One-Player Animation

Advanced animation includes any type of animation with more than two separate pictures. Usually, animation of this type involves four or five separate pictures, several moving parts, and a special sequence that must be integrated into the code of the program. The reason that animation involving more than two steps is so much more difficult is that it requires many intermediate steps that are not so easily defined. For instance, an animated Pac-Man would be incredibly simple because of the nature of its open-close movement (directly opposite pictures). However, a piece of animation like the runner involves frames that are not direct opposites of one another. It is these 'transitional' animation frames that are hard to define, and it is also these frames that determine whether or not your animation will be smooth.

The problem of these 'transitional' frames can be simplified using several techniques. First, it is a good idea to sketch the figure and each phase of its animation. Start by sketching the first and last pictures of the animation. Next, fill in the middle frame which will be the only transitional frame for the time being. Design the three frames with the Graphix Editor. Then animate the figure to see which transitional frames, if any, are necessary. Add the necessary frames and perfect the animation by cutting out frames or simplifying the figure. Do this until the animation is smooth and achieves the desired effect. The best way to learn how to edit and smooth out animation is to use the pm ANIMATOR system often.

Complex animation will often have several things moving at one time. The runner, for example, is moving his legs at the same time he is swinging his arms. To simplify

designing a piece of animation that has several parts moving at once, design the animation to move only one part. When the animation of that one part is satisfactory, add the other parts of animation one by one. Sometimes the animation of one part will require more steps than the animation of another part of the figure. For instance, it may take five steps to move the runners legs while it takes six for him to fully swing his arms. Adding another frame to compensate for this may make the animation look awkward. The best thing to do is to rework the animation so that either the legs move in slower jumps from one frame to another (allowing another transitional frame), or the arms move at a faster rate.

The eight-pixel resolution and the single color of a single player is sometimes a problem when trying to represent an object. Many times we cannot overcome this problem and we must resort to using two combined players. Though we can never totally overcome this single-color limited-resolution problem, there are some partial remedies. One is to add or leave out something for the purpose of definition. For instance, the runner has a thin band left out to represent a belt, which would otherwise need to be done in another color. Here is a hypothetical situation to illustrate the importance of this leaving-something-out/putting-something-in method. Consider a helicopter that has to be designed in the space of one player. The shape of the helicopter's body is not important here; the main point is to have realistic rotor blade motion. From a front view the blade should appear to get smaller as its edge rotates toward the viewer; and then get larger again. This seems simple, but upon experimentation, we find that the rotor blade does not appear to be rotating at all. It is merely getting larger and smaller. Where did we go wrong? In real life, depth perception and color shading help us to perceive motion as it really is. Unfortunately, for our players we have only one color and limited resolution to work with. The best solution is to add a small raised bump on the edge of one side of the rotor blade, or to leave a part out somewhere on the moving blade (run the demo "D:HELICOPT.DEM). This additional part, although it may not seem obvious to you, will enhance animation greatly.

Another type of animation definition is a sort of simulated three-dimensional animation that can be performed quite easily. First design a shape that is symmetrical – a square or circle works nicely. Keep in mind that with this type of animation, the symmetrical shape will appear to be spherical if it is a circle, tubular if it is a square, and vase-like if it is any other shape. The next step is to design a face or some other design that is to rotate across the figure. The last step is to animate the figure by shifting the design by one space to the right on the object until the entire design has rotated 'off' the figure; it will appear to have gone in back of the figure. Next, shift the design to the left in the same way. Finally, arrange the frames in the order of extreme left to the middle to extreme right. This type of animation usually takes about nine or ten frames. The result is a rotating three-dimensional object.

These are just some of the techniques used in overcoming the limited resolution problem. If you run into a problem making an object, keep trying a new approach and eventually you will produce some animation that will satisfy you.

Chapter 14

Creating A Realistic Motion Routine

As you go through this section, you should run the demonstration program "D:MOTION.DEM" which demonstrates all of the motion routines listed here. You should also be familiar with the VBLANK routine included in TOTAL.LST (especially the way the variable VP is used in the routine.)

Motion routines produce the motion of the object; this is separate from animation, the movement created by changing the object's form. For example, an animated bird flaps its wings because its actual size is altered; however, it moves across the screen in some direction because its position is altered by a motion routine. Most motion routines are simple and require a simple incrementation or decrementation of the x and y (horizontal and vertical) positions. However, there are some types of motion that are harder to represent.

Consider the problem of creating a realistic ball that bounces up and down. The worst way to do this would be to position the ball a little further up in each frame of animation. This method is awkward because it uses up six or seven frames of animation and the height of the bounce is limited to several spaces. A better method would be to increase the x and y positions by increments until the ball reached the highest desired position and then incrementally decrease the y position while increasing the x position. However, this method moves the object up and down at a steady rate, while a real ball would slow down as it neared the top of its bounce and then speed up on its way down.

The correct motion routine should make the ball bounce in an arc-like path instead of a steady line. Here is a correct motion routine for a bouncing ball.

Motion routine:BALL

```
10 B= 0:REM B IS THE X POSITION THAT WILL BE PLUGGED INTO THE HORIZONTAL POSITION FOR PLAYER ZERO.
20 FOR X=1 TO 6:REM SIX BOUNCES
30 FOR X2=-10 TO 10 STEP .5:REM THIS IS THE LOOP THAT CALCULATES THE Y POSITION.
40 Y=X2*X2+40:B=B+1:REM CALCULATE THE VALUE FOR Y RELATIVE TO X2, ADD ONE TO THE X POSITION (B).
50 POKE HP,B:POKE VP,Y:NEXT X2:NEXT X:REM HP IS THE X HORIZONTAL REGISTER (HP=53248), VP IS THE SIMULATED Y REGISTER (SEE SECTION ON VBLANK ROUTINE)
60 REM THIS PROGRAM EXAMPLE ASSUMES THAT THE READER IS FAMILIAR WITH THE VBLANK ROUTINE AND HOW TO USE IT! (SEE SECTION ON VBLANK ROUTINE)
```

But a more realistic ball bouncing across the screen, unlike this one, would not continue to bounce just as high on each consecutive bounce. A real ball would lose height with each bounce until it finally stopped. The new routine must produce the same arc motion and it must also change the height of the bounce. To do this, we will use the old routine and add some changes. The most obvious change is to give the for-next statement two variables as limits. The new lines look like this:

```
FOR X2= -K TO K STEP 0.5
Y = X2*X2+Z: B=B+1
```

Z represents a displacement we must calculate every time the ball bounces. The displacement will be equal to the old value of the variable plus the difference between the old bounce and the new bounce. This difference can be found by subtracting the highest y value of the next bounce from the previous one. This must be calculated and added to the old Z value at every bounce. The new line at 50 looks like this:

```
NEXT X2:Z=Z+K*K-((K-1)*(K-1)):NEXT X
```

For more clarification, see the listing for this motion in "D:MOTION.DEM"

Next, consider the motion of a falling object dropped from a building. The object at first has a small speed that builds up rapidly until it hits the ground. Obviously, a simple routine that increases the y position of the object by one will not do. However, there are two solutions that will work. The easiest method is to gradually increase the number of spaces at a time that the y position is incremented at a time (increment the rate of increase in y). This will appear to make the object gain speed as it approaches the ground. Our routine will look like this:

```
Motion routine: FALLING OBJECT
10 POKE HP,100:Y=32:REM SET X POSITION AT 100 ON SCREEN, SET INITIAL Y POSITION AT 32.
```

```
20 FOR X=1 TO 148:REM LOOP THROUGH 148 TIMES.
30 R=R+.02:REM R IS THE RATE THAT IS ADDED TO Y AT LINE 40; HERE IT IS CHANGED TO MOVE OBJECT FASTER.
40 Y=Y+R:REM CHANGE Y POSITION BY MOVING IT R SPACES FURTHER DOWN THE SCREEN.
50 POKE VP,Y:NEXTX:REM ACTUALLY MOVE PLAYER BY CHANGING VP VALUE FOR NEW Y, LOOP BACK TO LINE 20.
```

The other way to simulate a realistic falling object would be to decrease the amount of time between each change in the y position. This method would be smoother than the previous method; however, it would be considerably slower. To produce the desired effect we must implement a for-next loop, used in this case as a timer, with a variable parameter. Our routine will look like this:

```
Motion routine: FALLING OBJECT II
10 Y=32:K=40:POKE HP,100:REM START THE OBJECT AT 32; K IS THE VARIABLE THAT DETERMINES HOW MUCH TIME THE LOOP AT LINE 30 TAKES; POSITION PLAYER AT 100 HORIZONTALLY.
20 FOR X=1 TO 100: REM MOVE THE OBJECT ONE HUNDRED TIMES.
30 FOR TIME = 1 TO K:REM TIME DELAY LOOP.
40 NEXT TIME:REM LOOP BACK TO 30
50 Y=Y+2:POKE VP, Y:REM MOVE OBJECT DOWN BY TWO SPACES.
60 K=K-.5: NEXT X: DECREMENT K BY .5 SO THAT TIME LOOP AT 30 IS CONSTANTLY DECREASING.
```

The computer's sine and cosine functions (SIN(X),COS(X)) can be used in a variety of ways to produce complex motion. Consider, for instance, a spaceship that orbits a tube shaped object in a spiral pattern. The spaceship must go steadily upward and slow down as it reaches the top of the arc, much like the motion of the ball. Then it must make the same arc-like motion upside down. The motion would follow a smooth wavelike pattern in which the object slowed down as it reached the very top or bottom of a wave. This happens to be the graph of the sine function. By adding a couple of lines that change the priority register (GPRIOR), we can make the spaceship appear to be behind the tube and then in front of it to give a three-dimensional effect. The code to produce this motion looks like this:

```
Motion routine:SPACESHIP
10 FOR I=0 TO 37 STEP.1:REM I WILL DETERMINE THE X AND Y POSITION IN LINE 20.
20 POKE HP,I*5+30:POKE VP, 100 + SIN(I)*40:REM CHANGE X POSITION (X POS. INCREASES AS I IN LINE 10 INCREASES), CHANGE Y BY TAKING THE SINE OF I
30 IF PEEK (VP)>139 THEN POKE 623,4 :REM IF THE VERTICAL POSITION BECOMES GREATER THAN 139 THEN CHANGE THE PRIORITY REGISTER SO THAT THE OBJECT WILL BE BEHIND THE TUBE.
```

40 IF PEEK(VP)<64 THEN POKE 623,1:REM IF THE VERTICAL POSITION BECOMES LOWER THAN 64 THEN CHANGE THE PRIORITY REGISTER (GPRIOR, A SHADOW REGISTER AT 623) SO THAT THE OBJECT WILL BE IN FRONT OF THE TUBE.

50 NEXT I

Using equations to give objects motion is an easy way to perform complex animated graphics. One drawback of this method is that it could greatly reduce the speed at which the object moves. A complex computation could take the computer several seconds which would make the motion routine useless. To solve this problem, we can use an equation to compute all of the moves ahead of time, store the (x,y) positions in strings and then reference them. Strings are perfect for our purposes because they store characters that are represented in memory as numbers 0 to 255, which are the extreme positions of a player-missile object; a player's x and y positions are never lower than zero and never higher than 255. To demonstrate this technique, we will use the motion routine for the spaceship and calculate and store all of the moves in a string ahead of time. The code to do this looks like this:

Motion routine: SPACESHIP II

10DIM M$(126):Z=0:REM MOVES WILL BE PRECALCUALTED AND STORED IN CHR$ FORM TO BE RETRIEVED AS ASCII VALUES.

20 FOR I=0 TO 6.2 STEP.1:Z=Z+2:REM ADD TWO TO Z WHICH IS AN INCREMENTING VALUE IN LINE 30.

30 M$(Z-1)=CHR$(INT(I*5+30)) :M$(Z)=CHR$(INT(100+SIN(I)*40):REM THIS LINE FIGURES OUT EACH SUCCESSIVE POSITION AND STORES THEM IN (X,Y) ORDER.

40NEXT I:A=ADR(M$):REM LOOP BACK TO CALCULATION PORTION OF ROUTINE. A IS THE POSITION IN MEMORY WHERE M$ IS STORED; WHEN READ FROM MEMORY THE CHARACTERS APPEAR AS ATASCII VALUES.

50 FOR I2=0 TO 190 STEP 31:REM THIS LINE FIGURES OUT THE STARTING PLACES FOR THE HORIZONTAL POSITION ON EACH SUCCESSIVE SINE WAVE CYCLE.

60 FOR I=1 TO 126 STEP 2:POKE HP,PEEK(A+I-1)+I2:REM THIS IS THE POSITION READING LOOP. THE POKE ADDS I2 TO THE NEXT HORIZONTAL MOVE VALUE AND THEN ACTUALLY MOVES THE PLAYER.

70 POKE VP,PEEK(A+I):REM READ Y VALUE AND POKE IT IN.

80 IF PEEK(VP)>130 THEN POKE 623,1:REM THIS LINE CHANGES THE PRIORITY TO GIVE 3D EFFECT AS IN LAST ROUTINE.

90 IF PEEK(VP)<70 THEN POKE 623,1:REM CHANGE PRIORITY BACK TO NORMAL

100 NEXT I: NEXT I2

110 REM NOTE THAT THE LOOP IN LINE 20 IS INCREMENTED BY A VERY SMALL VALUE (ie .1). IN MANY FUNCTIONS, SUCH AS SINE, THIS IS DESIRABLE.

Designing fairly complex motion routines of your own involves just a couple of steps. First, decide upon how the object's motion will change over a period of time – whether it will change its pattern of motion, speed up, etc. Next, design a simple routine that performs the basic motion of the object. Then add in the additional features such as the height of the next bounce, the speed, timing, etc. The only other step involved is perfecting the routine so that it performs as you want it to.

Chapter 15

Multiple Players and Multicolored Players

Sometimes the 8 x 256 resolution and tiny size of a player are just too limited to produce the kind of graphic effect that we desire. Also, a player is only one color and we may have wanted to use several colors. The only way to overcome these limitations is to use several players as one. This is done easily enough; simply position two players side by side on the screen, one containing the data for the right half of the figure and the other the data for the left half. A more challenging venture is to animate and move this two-player figure. With pm ANIMATOR this is accomplished by the SUMOVE$ routine which is discussed in Chapter 11. In this chapter, we will discuss how to design the overall two-player shape, as well as how to design the animation.

Let's assume that we want to design a large flying dragon that would flap its wings as it flew across the screen. Simply by the nature of this figure (large and high resolution), we can tell that we must use two or more players; for this example we will use three. The first step is to sketch what the dragon will look like on a piece of paper. If you can't sketch, then find several pictures of what you wish to create in animation. Next you must either obtain some bit-mapping paper or make your own. With this bit-mapping paper we are going to draw, by hand, our large multi-player figure. To make your own bit-mapping paper, design a master copy from which to make xeroxed copies. Take graph paper and divide each row of squares into two rows by adding a horizontal line. Each one of the newly formed rectangles (half a square) will represent an unlit block (a bit) of a player, just as the rectangles in the Grafix Editor's large editing window does. Now you can make a scale drawing of your large figure. If you wish to draw multi-player figures using a different width, then use the following proportions to make your bit-mapping paper:

PROPORTIONS FOR BLOCK (PIXEL) SIZE IN PLAYER-MISSILE SHAPES: HEIGHT COMPARED TO WIDTH OF A SINGLE PIXEL.

SINGLE LINE RESOLUTION:

At normal width a pixel is 2 times as wide as it is high.
At double width a pixel is 4 times as wide as it is high.
At quadruple width a pixel is 8 times as wide as it is high.

DOUBLE LINE RESOLUTION:

At normal width a pixel is just as wide as it is high.
At double width a pixel is 2 times as wide as it is high.
At quadruple width a pixel is 4 times as wide as it is high.

Note: for more information on the above, see Appendix I under player width.

After drawing this special graph paper and making several copies of it, we must begin designing our dragon.

We carefully sketch the dragon on one of the pieces of bit-mapping paper we have made and then fill out the blocks to outline its shape. However, suppose we find out that three players are too small for our figure. We can use a different player width to increase the horizontal size of the player. Double width will be sufficient for this shape, so we design a piece of bit-mapping paper to the right proportions (see above chart). Although the player's horizontal size is now twice as big, there are still only 24 pixels across. They are each just twice as wide (3 players * eight bits = 24 total blocks across). Soon enough, after sketching the dragon on the special graph paper (bit-mapping paper) and filling in the appropriate blocks, we find we have created an appropriate design. We then divide the dragon into three columns, each eight blocks wide (the width of a single player), and into rows sixteen blocks high (the height of one frame). In this way we will design the entire figure piecemeal, like a jigsaw puzzle, by designing each individual part of the multi-player figure as one frame. Since the data file created is saved by the Grafix Editor as a continuous record of data, we can divide the file into frames of any height we want in our program.

With multi-player figures we must remember that we can have several widths and colors as opposed to only having one. With this in mind it is sometimes to your advantage to mix player widths (i.e. to make some of the players regular width and some double, etc.). Our dragon, for example, may need a large body and a small detailed head. For this we could make the players that would be the body of double width, and the players that would be the head normal width for higher resolution. It is also advantageous to animate as little of the figure as possible. For instance, in the dragon in the demonstration the wings and tail (upper half of the body) are the only things that are animated. This means that we only have to have one 'frame' of data for the head and lower body, because these parts do not change; they remain the same throughout the dragon's flight. (For more on this see "D:DRAGON.DEM" and examine the file "D:BIG"; use the File Editor, as this is a Grafix Editor data file).

Multi-color players work much the same way that large multi-player figures do. You

cannot get a multi-colored single player per se without using any hardware tricks such as display list interrupts (see Chapter 16 for a note of explanation). Multicolored players are actually two players positioned on top of each other. Using this overlay method, four colors can be obtained: the background color, the color of player zero, the color of player one, and a special color caused by an overlapping of sections of player zero and one. This last special color is obtained by setting the fifth bit in the register GPRIOR (see Appendix I for more on this). The special color occurs when one pixel overlaps a lighted pixel of the other player, causing that particular pixel to become a different color. This only works with certain two-player combinations: players zero and one, and players two and three. It will not work, for instance, with player two and player three.

To generate multicolored players using the Grafix Editor follow this procedure. First, design the animation using one color (i.e. one player). After you have done this, design the player in three colors using the MULTI command (see Chapter 8 under MULTI). It is best to devote half of the file to the frames that will be in player zero, and the other half to the frames that will be in player one. Since this requires two players, it will require two frames of data for every one multi-colored frame you want to have. One frame of data will have the parts of the figure that are in the first color, the next frame will have the parts of the figure in the other color and both will have the parts that are to be in the special color. The main demo makes use of these techniques.

Chapter 16

Final Notes

This section covers miscellaneous tips that will help advanced users to implement player-missile graphics in their BASIC programs.

MODIFYING TOTAL.LST TO WORK WITH DIFFERENT GRAPHICS MODES.

If you attempted to use TOTAL.LST in conjunction with a high resolution graphics mode like 7 or 8 you would get 'garbage' on the screen. This is because the player-missile area has been placed on top of the screen data area. Thus, when you were putting player frame data into the player area it was also being put into the screen data area, causing the 'garbage'. To overcome this, position PMBASE a little lower in memory (screen memory is at the very top of user RAM). This is accomplished by changing line 32005 of TOTAL.LST to look like this:

```
32005 PM=PEEK(106)-40:PMB=256*PM
```

Remember that single line resolution player-missile graphics operate on a 2K boundary. That is, PMBASE can only be located at a location that is a multiple of 2K. Since RAMTOP (the Atari name for location 106) holds the page number of the highest useable RAM location (a page is a block of 256 consecutive memory locations), we can determine the very highest user-RAM location. It is always best to use the highest location in memory for the player-missile base since the ATARI uses the lower locations first. Since single line resolution operates on a 2K boundary, and 2K=8 pages of memory (2K = 2048 bytes = 8 pages x 256), we must subtract a number that is a multiple of 8 from the number in location 106. The highest number you will have to use is 40, and subtracting 8 usually works on the text modes and lo-res graphics modes.

DATA FILE STRUCTURE

The data files that you create with the Grafix Editor are standard ATASCII files. There are no gaps between successive frames in the file; they are merely placed top to bottom in successive order. If you do not wish to use LD$ (the quick loading routine) you can load a file using the OPEN and GET command. Suppose we wished to load a file of ten frames (10 frames x 16 = 160 bytes of information, 160 memory locations required) into the area just below the player-missile base. The BASIC coding for this would look like this:

```
10 REM LOADING A GRAFIX EDITOR FILE USING BASIC GET, OPEN COMMANDS.
20 OPEN #1,4,0,"D:RUNNER"
30 FOR I=PMB TO PMB+159:REM PMB = LOCATION OF PMBASE.
40 GET#1,A:POKE I,A
50 NEXT I: CLOSE #1
```

USING A POINTER SYSTEM

A pointer system is simply a systematic way of remembering certain memory locations. For example, say we wish to animate a player in our program. The ROT$ routine requires that we remember the exact position in memory from which to get the frame data. If we had to compute this every single time it would slow the animation down.

Our pointer system would be used to remember the exact position in memory of every frame. In this way we will have a sort of bookmark that tells us where the data for each separate frame resides in memory. We store the needed memory locations in an array. Suppose that we wish to make a pointer system that will point to sixteen frames that are stored just behind PMBASE. The code to do this would look like this:

```
10 DIM PNT(16):A=0:FOR I=PMB TO PMB+256-16 STEP 16:A=A+1:PNT(A)=I: NEXT I
```

Notice that our for-next loop is 16 less than 256. This is because the first pointer is located at PMB; if we were to have 16 more that would make 17 pointers. By employing this pointer system technique, we can use a simple for-next loop for animation. (For more information see the demos on the pm ANIMATOR disk).

MEMORY CONSERVATION TECHNIQUES

There are two memory conservation techniques that you may want to use. The first of these is to use the large space just behind PMBASE for storing frame data. This area is protected unused RAM which will not be used by the computer, so you might as well take advantage of it. This is also a nice place to store machine language routines that must remain in a fixed position, such as a vertical blank routine or a display list interrupt routine.

The other memory conservation technique is simple. Store as much data as you can on disk and load it directly into memory by disk. Do not use DATA statements to read frame data or other data into memory unless you have to, because it takes up a lot of memory.

OTHER TRICKS WITH PMG

There are a few other interesting tricks that one can do with player-missile graphics that we haven't really covered. One of them is the use of display list interrupts with player-missile graphics. With a DLI, it is possible to produce a figure with many colors or a figure with rotating colors using only one player.

Another trick that can be done is to simulate more players on the screen. To do this one can use a vertical blank routine that moves a player from one horizontal position to another, back and forth. Each player would move back and forth horizontally so quickly that it would look like a separate player.

These tricks are beyond the scope of this manual and are only for advanced programmers with a thorough knowledge of machine language and of the ATARI computer itself. You are encouraged to experiment with pm ANIMATOR and PMG, but you are now on your own!

APPENDICES

Appendix I

PM Graphics – The Hardware Registers

In this Appendix we will describe all of the hardware registers that are used when accessing player missile graphics as well as the software registers created by the TOTAL.LST subroutines. All of the following information on hardware registers can also be found in the Atari Technical Users Notes.

We will give each register the name that Atari has assigned to it, e.g. DMACTL is location $D400. Where applicable, we will also list the shadow memory location for that hardware register. The following shorthand will be used in describing the bits that make up each register: Bit 0 will be labeled B0, bit 1 will be B1 . . . through bit 7 which will be B7. The format of this Appendix is as follows:

ATARI REG. NAME Functional Description

Read or Write Only? Address in hex (decimal) – Shadow if present-bit map of register (function of each bit)

See chapters 2-5 for explanation of terminology used here.

ATARI HARDWARE REGISTERS

DMACTL – Direct Memory Access Control

Write $D400 (54272) – shadow at $22F (559)
B6 and B7 – Not used
B5 1= Enable Instruction: Fetch DMA
B4 1 = Single line PM resolution
B4 0 = Double line PM resolution .
B3 1 = Enable player DMA (turn on players)
B2 1 = Enable missile DMA (turn on missiles)

B1 and B0

0,0 = No playfield DMA
0,1 = Narrow playfield DMA
1,0 = Standard playfield DMA
1,1 = Wide playfield DMA

Default value = $22 (34)

GRACTL – Graphics Control
Write $D01D (53277) – No shadow
B7-B3 – Not used
B2 1 = Latch triggers (Trig0-Trig3) remember if triggers were pressed
B1 1= Enable player DMA
B0 1 = Enable missile DMA

PMBASE – Player-missile Address Base Register
Write $D407 (54279)
Single line resolution
B0-B2 – Not used
B3-B7 – The most significant byte of the address of the player-missile area

Double line resolution
B0-B1 – Not used
B2-B7 – The most significant byte of the address of the player-missile area

PRIOR – Priority
Write $D01B (53275) – shadow at $26F (624)
B7-B6 = Not used
B5 1 = Multiple player color enable
B4 1 = Fifth player enable (causes all missiles to assume the color of playfield 3)
B3-B0 = Priority select (see table below)
Check the Hardware Manual for further details about PRIOR

Bit-Map for PRIOR

B3	B2	B1	B0	
PF0	PF0	P0	P0	Highest Priority
PF1	PF1	P1	P1	
P0	PF2	PF0	P2	
P1	PF3+P5	PF1	P3	
P2	P0	PF2	PF0	
P3	P1	PF3+P5	PF1	
PF2	P2	P2	PF2	
PF3+P5	P3	P3	PF3+P5	
BAK	BAK	BAK	BAK	Lowest Priority

COLPF0-COLPF3 and COLBK – Playfield and background colors
Write $D016,$D017,$D018,$D019,$D01A (53270-53274)
Shadows at $2C4,$2C5,$2C6,$2C7,$2C8 (708-712)
Playfield 0 = $D016
Playfield 1 = $D017
Playfield 2 = $D018
Playfield 3 = $D019
Background = $D01A

COLPM0-COLPM3 – Player Missile Color
Write $D012,$D013,$D014,$D015 (53266-53269)
Shadows at $2C0-$2C3 (704-707)
Player and missile 0 = $D012
Player and missile 1 = $D013
Player and missile 2 = $D014
Player and missile 3 = $D015

All color registers have the following bit map:

B7-B4 = Color
B3-B1 = Luminance (Brightness)
B0 = Not Used
Luminance can be even numbers from 0 (black) to 14 (brightest).

Colors are:

B7	B6	B5	B4	
0	0	0	0	
0	0	0	1	Gold
0	0	1	0	Orange
0	0	1	1	Red-Orange
0	1	0	0	Pink
0	1	0	1	Purple
0	1	1	0	Purple-Blue
0	1	1	1	Blue
1	0	0	0	Blue
1	0	0	1	Light-Blue
1	0	1	0	Turquoise
1	0	1	1	Green-Blue
1	1	0	0	Green
1	1	0	1	Yellow-Green
1	1	1	0	Orange-Green
1	1	1	1	Light Orange

GRAFP0-GRAFP3 – Player graphics register
Write
Player 0 $D00D (53261)
Player 1 $D00E (53262)
Player 2 $D00F (53263)
Player 3 $D010 (53264)

These addresses write data directly into the player graphics registers, independent of DMA. If DMA is enabled, the player graphics register will load automatically from the area defined by PMBASE.

GRAFM – Missile graphics register
Write $D011 (53265)

This address writes data directly into the missile graphics register, independent of DMA. If DMA is enabled, the missile graphics register will load automatically from the area defined by PMBASE.

SIZEP0-SIZEP3 – Player Size
Write
Player 0 $D008 (53256)
Player 1 $D009 (53257)
Player 2 $D00A (53258)
Player 3 $D00B (53259)

B7-B2 Not used
B1-B0 0,0 Normal size (8 color clocks wide)
0,1 Twice normal size (16 color clocks wide)
1,0 Normal size
1,1 4 times normal size (32 color clocks wide)

SIZE – Missile size
Write $D00C (53260)
B7-B6 Missile 3
B5-B4 Missile 2
B3-B2 Missile 1
B1-B0 Missile 0

Each pair of bits maps the same as for player size.

HPOSP0-HPOSP3 – Player horizontal position
Write
Player 0 $D000 (53248)
Player 1 $D001 (53249)

Player 2 $D002 (53250)
Player 3 $D003 (53251)

Values POKEd into these locations control the horizontal position of the appropriate player. $30 (48) is the left edge of a standard width screen. $D0 (208) is the right edge of a standard width screen.

HPOSM0-HPOSM3 – Missile horizontal position
Write
Missile 0 $D004 (53252)
Missile 1 $D005 (53253)
Missile 2 $D006 (53254)
Missile 3 $D007 (53255)

See HPOSP0 for description.

M0PF,M1PF,M2PF,M3PF – Missile to playfield collision
Read
Missile 0 $D000 (53248)
Missile 1 $D001 (53249)
Missile 2 $D002 (53250)
Missile 3 $D003 (53251)

B7-B4 Not used
B3 1 = Collision with playfield 3
B2 1 = Collision with playfield 2
B1 1 = Collision with playfield 1
B0 1 = Collision with playfield 0

P0PF,P1PF,P2PF,P3PF – Player to playfield collision
Read
Player 0 $D004 (53252)
Player 1 $D005 (53253)
Player 2 $D006 (53254)
Player 3 $D007 (53255)

B7-B4 Not used
B3 1 = Collision with playfield 3
B2 1 = Collision with playfield 2
B1 1 = Collision with playfield 1
B0 1 = Collision with playfield 0

M0PL, M1PL, M2PL, M3PL – Missile to player collision
Read

Missile 0 $D008 (53256)
Missile 1 $D009 (53257)
Missile 2 $D00A (53258)
Missile 3 $D00B (53259)

B7-B4 Not used
B3 1 = Collision with player 3
B2 1 = Collision with player 2
B1 1 = Collision with player 1
B0 1 = Collision with player 0

P0PL, P1PL, P2PL, P3PL – Player to player collision
Read
Player 0 $D00C (53260)
Player 1 $D00D (53261)
Player 2 $D00E (53262)
Player 3 $D00F (53263)

B7-B4 Not used
B3 1 = Collision with player 3
B2 1 = Collision with player 2
B1 1 = Collision with player 1
B0 1 = Collision with player 0

HITCLR – Collision clear register
Write $D01E (53278)

POKEing any value to this register clears all of the collision registers (makes them 0).

These are all of the hardware registers that are used with player-missile graphics. Further information can be found in the Atari Technical Users Notes.

pm ANIMATOR REGISTERS

The following are variables which are used by TOTAL.LST, the heart of the pm ANIMATOR runtime package. They are not ATARI register names but actual variable names that can be referenced in your programs.

HP – This variable represents the player's horizontal position register. This short notation is easier to remember than 53248 (or was it 53428??.)

VP – This is the location of the vertical position register of player zero, created by the vertical blank routine in TOTAL.LST. The vertical position registers for the players are as follows:

VP+0 - PLAYER ZERO - 1780
VP+1 - PLAYER ONE - 1781
VP+2 - PLAYER TWO - 1782
VP+3 - PLAYER THREE - 1783

For more information see chapters ten and eleven of the pm ANIMATOR MANUAL.

LE - This is the location of the player height or 'length' register for player zero. The vertical blank register in TOTAL.LST requries that you first define the length of a player by POKEing the desired length into the appropriate player's length register. These registers are as follows:

LE+0 - PLAYER ZERO - 1784
LE+1 - PLAYER ONE - 1785
LE+2 - PLAYER TWO - 1786
LE+3 - PLAYER THREE - 1787

For more information see Chapters ten and eleven.

PM - This is the high byte or the 'page number' of PMBASE. This is often useful in calculating exact memory locations needed for the machine language routines. For more information see Chapters ten and eleven.

PMB - This is the exact memory location of PMBASE. This is often useful in calculating exact memory locations needed for the machine language routines. For more information see Chapters ten and eleven.

Memory location 1788 - The high byte of the address of player zero (PM+4) is POKEd into this register. This register is used by the vertical blank routine in TOTAL.LST.

Appendix II

References

The following references contain material relevant to the discussions in this manual in some cases at a simpler level than is presented here and in others at a more advanced level.

DE RE ATARI
Atari, Inc.
Sunnyvale, CA 94086

ATARI 400/800 BASIC REFERENCE MANUAL
Atari, Inc.
Sunnyvale, CA 94086

TECHNICAL USER'S NOTES
Atari, Inc.
Sunnyvale, CA 94086

WHAT TO DO AFTER YOU HIT RETURN
Albrecht, Finkel, and LeBaron
Hayden Book Company
Rochelle Park, N.J.

THE ATARI ASSEMBLER
Don Inman and Kurt Inman
Reston Publishing Co., Inc.
Reston, VA

YOUR ATARI COMPUTER
Ion Poole, Martin McNiff, and Steven Cook
OSBORNE/McGraw-Hill
Berkeley, CA

6502 ASSEMBLY LANGUAGE PROGRAMMING
Lance A. Leventhal
OSBORNE/McGraw-Hill
Berkeley, CA

6502 APPLICATIONS BOOK
Rodnay Zaks
Sybex
Berkeley, CA

The following periodicals have also contained valuable information on PMG and general programming techniques from time to time:

A.N.A.L.O.G.
Cherry Valley, MA 01611

ANTIC
San Francisco, CA 94107

BYTE
Peterborough, NH 03458

COMPUTE!
Greensboro, NC 27403

CREATIVE COMPUTING
Morris Plains, New Jersey 07950

MICRO
Chelmsford, MA 01824