Search

ATARI® Player-Missile Graphics in BASIC

Philip C. Seyer

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging In Publication Data

Seyer. Philip C.

Atari player missile graphics in BASIC.

"A Reston computer group book."

1. Computer games. 2. Atari computer-Programming.
3. Basic (Computer program language) 4. Computer graphics.

I. Title. GV1469.2.S49 1984 794.8'2 83-21223
ISBN 0-8359-0112-2

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

I would like to dedicate this book to my brother, Mark and my dad, Herman, both of whom gave me much encouragement and support.

Acknowledgments

Special thanks go to my eleven year old son. Dan Seyer, who helped design the game in Chapter 11. Dan also came up with several programming suggestions, including a clever way to handle "collision detection."

Thanks go to game programmer, Robert Sombrio for his innovative joystick reading routine. And thanks to Datasoft Inc., for their **Graphics Master** program, which was helpful in preparing some of the graphics for this book. Likewise, thanks to Robert Martin for his **MMG BASIC Debugger**, which proved quite helpful in preparing the programs in this book.

Thanks, too, to the people at Atari, Inc. for their help in answering technical questions and supplying supportive instructional documents.

Last, but not least, I would like to thank the people at Reston Publishing Company who helped put this book together, especially Nikki Hardin. Senior Editor; Nanette T. Edwards and Camelia Townsend, Production Editors; Al Pagan, cover artist.

Preface

This book is a self-instruction course in how to design and animate screen images on the ATARI Computer. The book centers around what is called "Player-Missile Graphics (PMG)".

The course will take you step-by-step through all of the fundamentals of PMG and give you "progress checks" so that you can check your understanding of the material presented. When you see a black circle in the margin (like this •), you will know I am about to ask you a question. Since the correct answer is given immediately below the question, you may wish to cover it up with a separate sheet of paper before you continue reading.

You will get the most from this book if you answer each question and then compare your response with the one given. If your answer is incorrect, I suggest you reread the material preceding the question and then answer it again. When you're sure you understand, go on to the next section.

Most chapters end with a sample program. I suggest you enter each program and save it under a unique name. Successive programs build on the previous ones, so when preparing a new program, first enter the one you typed in previously and then modify it.

Have fun! The programs all work. The one in Chapter 11 is quite involved and may contain some hidden bugs. I invite your help. If you find any bugs or come up with improvements, I would be delighted to hear from you. Write to me in care of Reston Publishing Company, Inc., Reston, Virginia.

Return to Table of Contents | Previous Chapter | Next Chapter

Introducing PMG

Surprise! Atari has a secret feature that sets it apart from most other personal computers. It's called **Player-Missile Graphics** (PMG for short). With PMG you can create all sorts of special graphic effects--effects that would be extremely difficult, if not impossible, with an Apple, IBM, or TRS-80.

EASY IMAGE CREATION

Once you know how, you can easily create your own "custom made" graphic images. Thanks to Atari's special direct memory access system (DMA), it's almost as easy as shading in squares on a piece of graph paper.

EXTRA COLORS

You can make your "babies" any color you want. These images (or "players," as Atari calls them) are independent of the usual screen images. Let's say you are using what is called "graphics mode 5." Normally, with this mode, you are limited to four colors. But with PMG you can have up to nine colors, because each player can be a different color.

THREE-DIMENSIONAL EFFECTS

PMG lets you simulate 3-D. Because players are independent of the other screen images, they can be set to hide behind (or go in front of) other objects. Also, it's really easy to change the size of a player and to make a player look like it is moving from a distant location into the foreground. Atari uses this technique effectively in STAR RAIDERS to animate the attacking spacecrafts. So can you!

ANIMATION

PMG greatly simplifies animation. With PMG you can move players quickly and smoothly with just a few commands. At the same time you can simultaneously create sound effects and carry out other processing. That's because PMG is controlled by two extra microprocessors called ANTIC and CTIA (or GTIA).

NOT JUST FOR GAMES

Atari first developed PMG to simplify game programming. But you can use PMG anywhere that animation or special screen images and colors are useful. For example, in an educational program you could grab the learner's attention with an animated arrow. The arrow could point out various parts of a diagram. In a music program you could use players for notes and have them dance across the screen along with the music. In a program that displays textual material, you could use players to highlight important words or sentences. Normally this is not so easy. Without PMG, graphic images cannot appear on the same line with text.

PMG could be useful in business programs, too. For example, you could design an animated warning routine to catch an operator's attention when a crucial entry was required. It might even make the job more fun and cut down on data entry errors.

LEARNING TO USE PMG

You may be saying to yourself: "If PMG is so great, why aren't more people using it?" Well for one reason, it has been really hard to learn--that is, up until now. PMG is perhaps the most powerful, yet least understood, feature of the Atari computer.

But relax. In this book we will take you step by step through everything you need to know to create sophisticated graphic effects far beyond the reach of most other micros.

OVERVIEW OF PMG SETUP

Here are the major programming tasks needed to set up PMG. (Don't try to understand all this right now. It's just an overview. Details will come later.)

- Design the graphic images you want to display. Then calculate the data needed in memory to produce these images.
- Allocate space in memory for PMG.
- Dimension strings that will hold player image data.
- Set the PMG memory to zero data.
- Read the player image into the player image strings.
- Set the color of the players.
- Tell Atari where the PMG memory starts.
- Tell Atari whether you want single- or double-line resolution.
- Turn on PMG.
- Set the initial position coordinates for players and missiles.
- Animate the players and missiles as desired.

These are the basic things you'll need to get started. I'll cover them first. Then after you've mastered them I'll show you some advanced programming tricks you can amaze your friends with. "How did you do that?" they'll say.

Now, let's get started with the first task--designing player images.

Return to Table of Contents | Previous Chapter | Next Chapter

2

Designing Player Images

In this chapter you'll learn how to design an original graphic image that you can animate using Atari's special PMG features. We'll call this image a "player" to distinguish it from other graphic objects that may appear on the screen. Specifically, when you finish this chapter, you'll be able to:

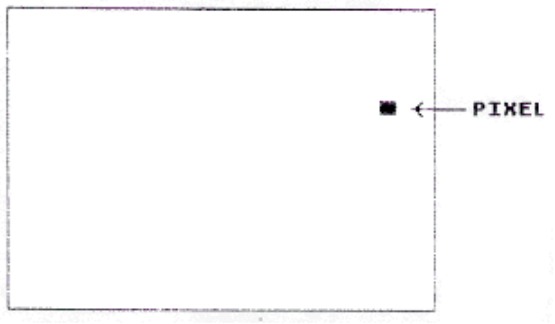
1. Lay out a player image on graph paper.
2. Figure out the binary data required in memory to create a player.
3. Calculate the decimal numbers needed to create a player image.
4. Sketch out the player images that would be created by various kinds of data in memory.
5. Explain what distinguishes player-missile (PM) memory from ordinary memory.

Let's begin by taking a look at the way graphic images are stored in memory.

HOW IMAGES ARE STORED

As you probably know, your Atari has several different graphics modes. In this discussion, I will be referring to graphics modes 3 through 8. These modes are for displaying pictures rather than words or letters.

An important graphics term is pixel, short for "picture cell." (Think of "pic.-cell.") A pixel is the smallest picture element possible. The size of a pixel depends on the graphics mode you are in. This next illustration shows the approximate size of a pixel in graphics mode 3.



If you'd like to see what a pixel actually looks like in different modes, trying running this program:

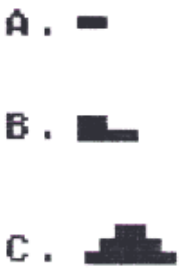
```
10 GRAPHICS 5:? "WHAT GRAPHICS MODE WOULD YOU LIKE? ...
   (ENTER A NUMBER BETWEEN 3 AND 8)"
20 INPUT GM:IF GM<3 OR GM>8 THEN 10
30 GRAPHICS GM
40 COLOR 3
50 PLOT 30,18
60 ? "THIS IS A PIXEL IN GRAPHICS MODE";GM
```

```
70 FOR PAUSE=1 TO 1000:NEXT PAUSE
80 GOTO 10
```

As you may know, all data in random access memory (RAM) is stored in consecutive memory locations called bits. (Eight consecutive bits are called a byte.) Each bit contains either a one or a zero.

A certain part of RAM is used to hold data that will be used to display screen images. Let's call that screen RAM. If a bit in screen RAM is set to one, then the corresponding screen pixel will light up. If a bit is a zero, then the pixel will be turned off.

- Suppose a byte (eight bits) contains these values: 00011000. Which of these diagrams correctly shows the screen image that would be displayed by that byte?



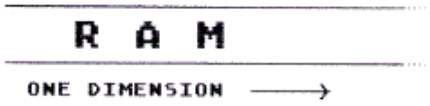
ANSWER

A (Remember that each bit set to one causes the corresponding screen pixel to be illuminated.)

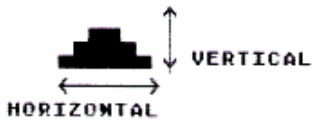
ONE- AND TWO-DIMENSIONAL IMAGES

Notice that image "A" above is not yet really a two-dimensional image. It has a horizontal dimension, but no vertical dimension. Actually, it is just a stubby line. Let's make it into a two-dimensional image by turning on some pixels directly beneath it on lines two and three.

Now we have a two-dimensional image. It has height and width. In Ordinary RAM it would be hard to display this image. We would have to do a lot of calculating to figure out which bits in RAM to set to one. Why? Well, RAM is linear; it consists of a series of bytes laid end to end. It is one-dimensional.



On the other hand, screen images are two-dimensional. They have a horizontal and a vertical dimension. Fortunately, the Atari engineers simplified



all this. They arranged for the second byte in PM memory to be displayed directly under the image of the first byte. Similarly, the third byte is displayed directly under the image of the second byte, and so on. This way the data for an image can be stored in consecutive bytes in RAM as illustrated here:

Of course, before you put any data into those consecutive bytes, you need to know exactly what image you want to create. So let's go over the procedure for designing a player image.

DESIGNING A PLAYER IMAGE

First, lay out a grid that is 8 columns wide with as many rows as you like up to 128. For example, you might start with a grid that has 8 columns and 11 rows, like this:

Next, make an image by shading in selected squares on the grid. (Actually, each square on your grid represents a pixel on the display screen.) Here's an example:

- In this example, how many pixels are lit up to draw the character's neck?

ANSWER

Just one.

LIGHTING UP PIXELS

The next step is to figure out which bits in memory to set to one so the proper pixels are turned on. This is really quite simple. All we do is convert each row in our grid to a number.

To do that we look at each square in our grid. If the square is shaded in, we write down a "one." If the square is not shaded, we write down a zero. For example, we would convert the first row of our player like this:

This number--00011100--is called a binary number. Notice that it is composed of just ones and zeros. As you may know, a binary 00011100 is not at all the same as eleven thousand, one hundred (11,100). I'll explain this shortly if you don't understand binary numbers. For now, just remember that we are translating each row of our player into a binary number.

- I did the first row; now you try the second and third. Convert them into binary numbers.

ANSWER

00011100 (row 2)
00001000 (row 3)

- Now would you like to do the rest?

ANSWER

a. 00011100
b. 00011100
c. 00001000

d. 00011100
e. 00111010
f. 01011001
g. 00011000
h. 00101000
i. 01001100
j. 01000100
k. 01000100

Now we have all of the binary numbers needed for our player image. But before we can put these values into Atari's memory, we need to convert them into decimal numbers. That's because BASIC was designed for inputing decimal numbers, not binary ones.

If you already know what decimal and binary numbers are and how to convert from binary to decimal, you can skip ahead to "Converting the Image to Decimal Numbers." Otherwise, read on and I'll explain.

Decimal numbers are the kind used by most normal people (You know what I mean--people who aren't assembly language programmers). The word part "dec" means "ten". For example, a decade is a period of ten years. The decimal number system contains ten different number symbols, which are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Similarly, the word part "bi" means "two." For example, bimonthly means "every two months." Binoculars are field glasses designed for use by **two** eyes. The binary number system uses only two kinds of number symbols: 1's and 0's.

Let's look at some binary numbers. To keep things simple, we'll start out with small binary numbers and put each number symbol in a box.

- So a binary 11 represents a decimal 3 since 2 + 1 = 3. What, then, would this binary number represent?

ANSWER

2 (2+0=2)

Now let's try a binary number with three number symbols. Again, we'll put each symbol in a box. This time, above each box, we'll show what the box is worth in decimal if it has a "1" in it.

So this binary "111" represents a decimal 7 since 4 + 2 + 1 = 7.

- Got that? Let's see. Give me the decimal equivalent of each of these binary numbers:

ANSWER

6 (4+2) 3 (2+1), 2

- Let's take away the boxes. What is the decimal equivalent of each of these binary numbers?

a. 101 _____
b. 010 _____
c. 001 _____
d. 111 _____

ANSWER

a. 5 **b.** 2 **c.** 1 **d.** 7

Now let's look at an eight-position binary number. We'll put the digits in boxes again to help you visualize the number.

This binary number is equal to 255 since:

128+64+32+16+8+4+2+1=255

- Now, you try one. How much is this in decimal? (Hint: compare this one with the previous binary number.)

ANSWER

253 (It is just two less than the previous example. Notice we now have a "0" in the "2 box" instead of a "1."

- How about this one?

ANSWER

20

- Try one more.

ANSWER

40 (32+8=40)

CONVERTING THE IMAGE TO DECIMAL NUMBERS

Now you're ready to convert your own player image to decimal numbers. Before you do that, however, you may wish to practice on some examples.

- Here's the image we worked on earlier. I've done the first three rows for you. Try the rest if you like:

00111100 (4+8+1632)	a. 60
00011100 (16+8+4)	b. 28
00001000 (8)	c. 8
00011100	d. _____
00111010	e. _____
01011001	f. _____
00011000	g. _____
00101000	h. _____
01001100	i. _____
01000100	j. _____
01000100	k. _____

ANSWER

a. 28
b. 28
c. 8
d. 28
e. 58
f. 89
g. 24
h. 40
i. 76
j. 68
k. 68

If you'd like more practice in calculating decimal numbers, you may wish to try this next one. Otherwise, on to the next chapter. I can't wait to show you a new, little-known method for allocating PM memory!

MORE PRACTICE

- Give the decimal numbers for creating this shape:

ANSWER

a. 60 (32+16+8+4)
b. 124 (64 more than the previous row)
c. 116 (124-8)
d. 127 (3 more than row 2)
e. 124 (same as row 2)
f. 112 (12 less than row 5)
g. 124 (same as row 2)
h. 96 (64+32)

Congratulations. You can now design your own player image and calculate the decimal numbers needed for it. Next, you learn how to reserve memory for your player-missile image data.

Return to Table of Contents | Previous Chapter | Next Chapter

3

Dimensioning Strings for PMG

In this chapter you'll learn an important secret--one not revealed in most other books and articles on PMG.* The secret is to use strings to store PM (Player-Missile) data. [If you don't know what a string is or how to dimension it, I suggest you see **Inside ATARI BASIC** by William Carris (Reston, Va.: Reston Publishing Company, Inc., 1982.]

Why is this such a big deal? Because (as I mentioned earlier) Atari has the ability to move string data super fast. So by using strings you can achieve fast vertical animation without resorting to machine language! Without using strings, vertical moment in BASIC is slow and jerky.

*I'd like to thank Sheldon Leemon for his excellent article on how to use strings for fast PMG animation: "Take Apart: Outer Space Attack," Softside Magazine, March 1982.

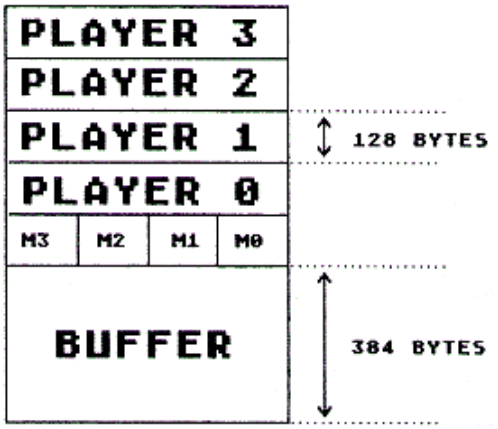
DIMENSIONING STRINGS FOR PMG

Using this approach, you dimension a separate string at the beginning of your program for each of your "players." Also, you dimension one string for the missile area. In dimensioning these strings, there are certain rules you must keep in mind.

THE BUFFER STRING

One rule is that the beginning of PM memory is not used to display graphic images directly. That is, data in this beginning area are not automatically displayed on the screen. In this book, we will call this beginning area of PM memory a buffer.

If you want, you can store player image data in the buffer and then move it to the display area whenever you want to. Here's a diagram that shows the PM memory area. See if you can distinguish between the buffer and display areas.



Note that this diagram shows the PM memory organization for what is called **double-line resolution**. Double-line resolution simply means that two lines are used to display each byte in PM memory. That is, if a bit is set to "1," then two pixels (one underneath the other) are lit up. We'll talk about double-line resolution in more detail later. Just remember that this diagram is for double-line resolution. Later we'll discuss single-line resolution (see Chapter 9).

- Study the diagram. Then answer these questions.

  1. How big is the buffer area at the beginning of the PM memory area?

  2. In double-line resolution, how many bytes are set aside for each player?

  3. Where does the PM display area start?

     a. 384 bytes past the beginning of the PM memory area.
     b. At the very beginning of PM memory area.
     c. Neither a nor b.

ANSWER

1. 384 **2.** 128 **3.** a

One important rule, then, is that when using double-line resolution, you must have a **384**-byte buffer at the beginning of the PM memory area. This buffer area is not used to turn on screen pixels. That is, if a bit is set to "1" in this area, there will be no direct effect on what is displayed on the screen. That's why we say the actual PM display area starts 384 bytes after the beginning of the PM memory area.

Another important rule is that when you are using double-line resolution, the PM memory area must start on what is called a "1K boundary." If you know what I mean by "1K boundary," you can skip to "Starting on a 1K Boundary." Otherwise, read on and I'll explain.

1K BOUNDARY

A 1K boundary is simply a location in memory that can be divided evenly by 1K (1K = 1024 bytes). For example, 2048 is on a 1K boundary because 1024 goes into 2048 evenly two times.

- Which of these are on a 1K boundary?

  a. 4096
  b. 3073
  c. 3072

ANSWER

a and c (1024 goes into 4096 evenly four times. 1024 goes into 3072 evenly three times).

STARTING ON A 1K BOUNDARY

To get your Player-Missile Base Register (PMBASE) to start on a 1K boundary, you need to do some calculations. That's because the first string you define won't automatically start at any given point in memory. There are different ways to find a 1K boundary. The method I recommend is to dimension some "filler strings" to take up space until the 1K boundary is reached. Here is some code you can use to define these "filler" strings. Take a moment to look it over.

```
DIM FILLER1$(1),FILLER2$((INT(ADR(FILLER1$)/1024)+1)*1024-ADR$(D$)-1)
```

Note: When typing this code into your computer, you would normally enter it as one continuous line. I've broken it into separate lines here to make it easier to read.

You don't really need to understand this filler code. All you need to know is that it takes up string memory until a 1K boundary is reached. But, if you like, I'll explain; otherwise, you can skip ahead to "Setting Up PM Memory."

UNDERSTANDING THE FILLER CODE

Since you're so interested in the filler code, I'll rewrite it for you in a slightly different way so it's easier to understand:

```
10 DIM FILLER1$(1)
20 BOUNDARY=INT(ADR(FILLER1$)/1024+1)*1024
30 LENGTHFILLER2=(BOUNDARY-ADR(D$)-1
40 DIM FILLER2$(LENGTHFILLER2)
```

Let's examine this code line by line, starting with line 10:

```
10 DIM FILLER1$(1)
```

In line 10 we define our first filler string, which we call FILLER1$. We do this so we have a reference point, to know where the beginning of string memory starts.

```
20 BOUNDARY=INT(ADR(FILLER1$)/1024+1)*1024
```

In line 20 we calculate the address of the next 1K boundary above the address of FILLER1$. We do this by dividing the address of FILLER1$ by 1024, adding 1 and throwing away the remainder. Then we multiply that answer by 1024. (INT "throws away the remainder." ADR gives us the address of FILLER1$.)

Let's apply this code to a simple example. Suppose the address of FILLER1$ is 2040. We divide 2040 by 1024 and get 1.992. We add 1 to 1.992 and get 2.992. We throw away the remainder (.992) and get 2. Then we multiply 2 by 1024 to get 2048, which is the 1K boundary above 2024.

```
30 LENGTHFILLER2=(BOUNDARY-ADR(D$)-1)
```

In line 30 we simply subtract the address of FILLER1$ from the 1K boundary and then subtract 1 from that. We subtract 1 because we want FILLER2$ to fill up space right up to (but not over) the 1K boundary.

We still need to find the length of our second string, namely FILLER2$. If we subtract the address of FILLER1$ from the next 1K boundary, we would get 8, since 2048 - 2040 = 8. But to get the proper length for FILLER2$, we need to subtract 1 from 8 (since we want to take up space just up to the 1K boundary). Look at this diagram:

- Suppose that the address of FILLER1$ were 2030. What would the proper string length be for FILLER2$?

ANSWER
17 (2048-2030-1=17)

This finishes our explanation of the filler code. Now let's go on to the next step.

SETTING UP PM MEMORY

After dimensioning the filler strings, the next step is to allocate space for the PM memory by dimensioning additional strings. Let's look again at the way PM memory is organized so you can see what PM strings will be needed:

- Based on this diagram, which string do you think we should define first?

  a. BUFFER$
  b. MISSILES$
  c. PLAYER0$

ANSWER
a. BUFFER$

It's important to determine the BUFFER$ string immediately after the filler strings. That way the buffer will start on a 1K boundary. For example:

```
10 DIM FILLER1$(1),FILLER2$((INT(ADR(A$)/1024+1)*1024-ADR(FILLER1$)-1)
20 DIM BUFFER$(384)
```

- Actually, line 20 isn't finished yet. Can you explain why?

ANSWER

It's not finished because we still need to dimension strings for the missiles and players.

In doing this, we use one string for all of the missiles and a separate string for each of the players.

- With this in mind, see if you can finish line 20:

  20 DIM BUFFER$(384)

ANSWER

20 DIM BUFFER$(384),MISSILES$(128),PLAYER1$(128),PLAYER2$(128),PLAYER3$(128)

Of course, you could use different variable names, such as M$ for MISSILES$, or P0$ for PLAYER1$, and so on. Also, notice that there is only one string defined for all four missiles.

PLAYER NUMBERING

Notice that the players are numbered from zero to three. Does that seem a little strange to you? (It does to me!) Actually, you can name the players any way you want to. This kind of naming convention is often used by assembly language programmers. It really means that PLAYER0 starts 0 bytes from the beginning of the player area. Similarly, the name "PLAYER1" means that the player is located 1*128 or 128 bytes from the beginning of the player area. Similarly, PLAYER2 is located 2*128 or 256 bytes from the beginning of the player area.

That's quite a mouthful I know. Let's see if I got the message across:

- Suppose you know that the beginning of the player memory area starts on byte 4480. Where would PLAYER1 start?

ANSWER

4608 (4480+1*128=4608)

- Again, assuming the player memory area starts on byte 4480, where would PLAYER3 start in memory?

ANSWER

4864 (4480+3*128=4864)

You now know how to allocate space for the PM memory area using double-line resolution. Later you'll learn how to do a similar setup for single-line resolution. That will be easy since you've already learned the fundamental concepts involved. Now that we've defined our player image and allocated space for PM memory, let's get a player on the screen!

Return to Table of Contents | Previous Chapter | Next Chapter

4

Getting a Player on the Screen

Now that you have designed a player and have allocated space for PMG memory, let's see what that player looks like.

- When you finish this chapter you will be able to:
- Write the code needed to get a player on the screen.
- Put the player anywhere you want by specifying horizontal and vertical coordinates.
- Make the player any desired color.
- Use double-line resolution.

To start with, let's go over the main programming tasks required to get a player on the screen.

MAIN PROGRAMMING TASKS

1. Define filler strings so that the PM strings area can start on a 1K boundary.
2. Allocate space for the PM memory area.
3. Set the PM memory area to zeros.
4. Dimension a separate string for holding our player image data.
5. Put our player data into that string.
6. Specify the color of our player.
7. Specify the location of PM memory.
8. Specify double-line resolution.
9. Turn on PMG.
10. Specify the desired horizontal location of the player.

11. Put player image into the desired vertical location.

You have already learned to write the code for steps 1 and 2. so let's go on to step 3.

Setting the PM Memory Area to Zeros

When you run a program, the data in the string area is not automatically cleared out. That's why we need to explicitly set the PM memory area to zeros. Let's start with BUFFER$. Here's a quick way to set it to zeros:

```
11020 BUFFER$=CHR$(0)
11030 BUFFER$(384)=CHR$(0)
11040 BUFFER$(2)=BUFFER$
```

This code may seem somewhat confusing, but rest assured--it works. Line 11020 sets the first byte of BUFFER$ to zero. Line 11030 sets the 384th byte to zero. Line 11040 says "start with the second byte of BUFFER$ and set all the remaining bytes of BUFFER$ to the first byte of BUFFER$." (I know, it's a mouthful.)

As an aside, if you wanted to set all of BUFFER$ to some other value besides zeros, all you need to do is to change line 11020. For example to set BUFFER$ so that it has nothing but "X's," change line 11020 to read:

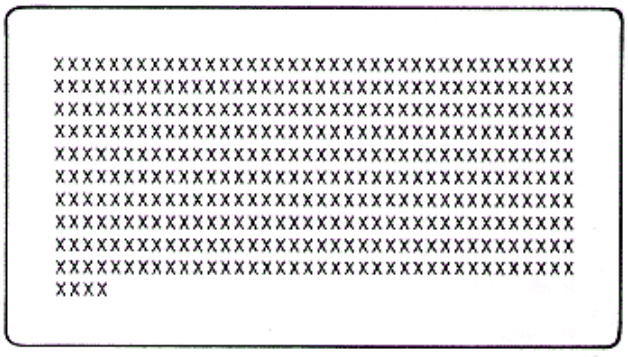
```
11020 BUFFER$="X"
```

If you'd like to take a break from reading, you may want to enter the above lines and experiment with changing line 11020 to see what affect it has on BUFFER$. If so. I suggest you start by entering these lines:

```
11010 DIM BUFFER$(384)
11020 BUFFER$="X"
11030 BUFFER$(384)=CHR$(0)
11040 BUFFER$(2)=BUFFER$
11050 ? BUFFER$
```

Note that in line 11020 we set the first byte of BUFFERS to "X." In line 11050 we print BUFFER$ just to see what it contains. (The question mark is short for "PRINT.")

After entering the lines, type "RUN." Since all of BUFFER$ has been set to "X," you will see this:



Now, change line 11020 so that it reads:

```
11020 BUFFER$=CHR$(0)
```

Run the program again. This time you will see a screen full of hearts. That's because the Atari code for a heart character is a zero.

- Think for a moment. Does the memory location of BUFFER$ contain hearts or zeros?

  a. Hearts
  b. Zeros

ANSWER

Zeros, of course! Remember, in memory, a bit always contains either a one or a zero.

To see the zeros in BUFFER$, try running this program:

```
11060 FOR I=1 to 384
11070 ? ASC(BUFFER$(I))
11080 NEXT I
```

Note: In line 11070 I am using "ASC" to return the actual number stored in each byte of BUFFER$ rather than displaying a character.

Now that we have set BUFFER$ to zeros, we can use it to set the rest of PM memory to zeros. For example, we can set the missile area to zeros like this:

```
MISSILES$=BUFFER$
```

- You try it. Assume that BUFFER$ has already been set to zeros. Then, write the statements needed to set the entire PM memory area to zeros. (Remember, besides BUFFER$, our PM memory area contains these strings: MISSILES$, PLAYER0$, PLAYER1$, PLAYER2$, PLAYER3$.)

ANSWER

```
MISSILES$=BUFFER$:PLAYER0$=BUFFER$:PLAYER1$=BUFFER$:
PLAYER2$=BUFFER$: PLAYER3$=BUFFER$
```

Player Image String

Besides dimensioning the PM memory area and setting it to zeros, we need to dimension a string to hold our player image data. We'll call it "IMAGE1$," meaning "image one for player one." As you will see later, we will want to have more than one image for each of our players so that we can do things like make their legs move.

Dimensioning the string is easy:

```
DIM IMAGE1$
```

Putting the proper data into a string is a bit harder. First, you need to know how to refer to specific bytes of a string. If you already know how to do that, you may wish to skip ahead to "Putting Data into IMAGE1$."

Referencing Specific Bytes

In Atari BASIC, you can refer to specific sections of a string variable by numbers in parentheses after the name of the string. The first number gives the starting byte of the section of the string you want to refer to. The second number gives the ending byte. For example, to refer to the first three bytes of a string called STORAGE$, we would write:

- How would you refer to bytes 5 through 9 of STORAGE$?

ANSWER

STORAGE$(5,9)

If you want to refer to a single byte, just list the number of that byte twice. For example, to refer to byte 5 of STORAGE$, you would write:

STORAGE$(5,5)

- How would you refer to byte 15 of STORAGE$?

ANSWER

STORAGE$(15,15)

Putting Data Into IMAGE1$

To get out player image data into the string we use a "FOR-NEXT" loop. Our first statement in this loop is:

FOR I=1 to 15

This sets up the loop so that I is set to 1 the first time through the loop. The next time through the loop. I will be set to 2 and so on until I is 15.

- Can you guess why we need to make 15 passes through the loop?

ANSWER

When we created our player, we had 15 rows in our matrix. We had a number for each row. These numbers will first be stored in data statements. Each time through the loop we will read a number and then put it into IMAGE1$. Since we will be adding two rows of zeros at the top of the matrix and two rows at the bottom, we have 15 numbers altogether.

Here is the complete loop for putting all 15 numbers into IMAGE1$:

FOR I=1 TO 15:READ A:IMAGE1$(I,I)=CHR$(A):NEXT I

Let's look at each statement separately. We've already discussed "FOR I=1 TO 15." The next statement is "READ A."

READ A This simply gets a number contained in a data statement and puts it in variable A.

IMAGE1$(I,I)=CHR$(A) This statement puts the number into successive bytes of IMAGE1$. Note the (I,I). The first time through the loop the variable I will be set to 1. So the statement will be equivalent to:

IMAGE1$(1,1)=CHR$(A)

This means set the first byte of IMAGE1$ to the contents of variable A. The second time through the loop, I will be set to 2. So the statement will be equivalent to:

IMAGE1$(2,2)=CHR$(A)

- What does this mean?

ANSWER

Set the **second** byte of IMAGE1$ to the contents of **A**.

Now let's talk about **CHR$(A)**. We need this because the variable **A** is numeric. You can't put numeric data into a string directly. It has to be converted to string data (also referred to as **character data**). That's what **CHR$** does. It

takes whatever number is in parentheses and gets the corresponding character, which can then be stored in the string.

- What do you think would happen if we simply wrote:

  IMAGE1$=A

ANSWER

Actually Atari's syntax checking feature won't even let us enter a statement like this. If we try to, the word "ERROR" appears, and the "A" will be displayed in inverse video showing that something else probably needs to come after the equal sign.

Now that we have set the PM memory area to zeros, our next step is to specify what color we would like our player to be.

Specifying Player Color

You can easily control the color of a player by putting a number into a memory location. Here are the locations for each of the players:

Player	Location
0	704
1	705
2	706
3	707

(For your convenience these locations are also summarized in Appendix A.)

If you use graphics modes 3, 5, or 7, then you can use the following numbers as a guide for designating your player's color:

Color	Number
Gray	0
Gold	16
Orange	32
Red	48
Pink	64
Violet	80
Purple	96
Blue	112
Blue	128
Light Blue	144
Turquoise	160
Blue-Green	176
Green	192
Yellow-Green	208
Orange-Green	224
Light Orange	240

These are starting numbers. To any given number you can add an even number from 0 to 14 to change the lightness of the color. For example, for a light green color you might add 14 to 192 and use 206 as the color number. For a darker green you might add, say, 4 to 192 and use 196.

- Would 48 give you a bright red or dark red color?

ANSWER

Dark

- What number would you use for a light red?

ANSWER

62(48+14)

You might also use 56, 58, or 60 for a light red, depending on how light a shade you want. Notice that when you go beyond 62, you start moving into pink.

Poking the Color Number

You can use the **Poke** command to specify the player's color. To specify a dark gold color for player 0, you might write:

POKE 704,16

- Write a command to set player 2's color to dark violet.

ANSWER

POKE 706,80 (You might also use 82, 84, or 86 for dark violet. The lower the number, the darker the color.)

Specifying Start of PM Memory

The Atari microprocessor called ANTIC automatically takes care of displaying PM images on the screen. But before ANTIC can do that we have to tell it where PM memory starts. To do that we simply POKE the proper address into location 54279. Since our PM memory area is string memory, we can use the ADR function. The ADR function gives the address of the string that follows it in parentheses. For example:

ADDRES$=ADR(STORAGE$)

After this statement is executed, the variable ADDRES$ will contain the starting address of the string area called STORAGE$.

- Recall the structure of the PM memory area. What is the first string in the PM memory area?

ANSWER

BUFFER$ (Remember this area? It's sort of a multipurpose PMG storage area. Data stored here does not display on the screen).

- How would you refer to the address in memory where BUFFER$ starts?

ANSWER

ADR(BUFFER$)

- From what you've learned, see if you can write a statement to inform ANTIC of the location of the PM memory area.

ANSWER

POKE 54279,ADR(BUFFER$) (We poke the address of BUFFER$ into 54279 because the address of BUFFER$ is the same as the address of the start of the PM memory area.)

PMBASE

Location 54279 is known as the **Player-Missile Base Register (PMBASE)**. If you initialize a variable called PMBASE to 54279 at the beginning of the program like this

PMBASE=54279

then we can tell ANTIC where PM memory starts by writing a statement like this:

POKE PMBASE,ADR(BUFFER$)

Did you notice how I use the word "register" above to refer to a memory location? (Player-Missile Base Register=location 54279.) Remember, in data processing, a register is nothing more than a memory location dedicated to a specific purpose. Usually we put data into a register to control the computer's operation.

Specifying Resolution

Another important register is the one used to specify resolution. We touched on resolution earlier, but now let's consider it in more detail. With PMG you have your choice of either double- or single-line resolution. Single-line resolution means that when ANTIC sees a "1" in PM memory it will light up a single pixel.

In double-line resolution, when ANTIC sees a "1" in PM memory it lights up two pixels. Both pixels will be right underneath each other on two separate **lines** (hence the name double-**line** resolution). Perhaps an illustration will help clarify this. Suppose you have this binary data in PM memory:

00011100
00011100
00001000
00011100
00111010
01011001
00011000
00111100
00100100
00100100
01100110

With single-line resolution, your player will look like this:

With double-line resolution, you will get a player that looks like this:

- Which kind of resolution gives you a more "blocky" looking character? Which one looks more "detailed"?

ANSWER

Double-line resolution results in a more "blocky" looking player. You get a more detailed looking figure with single-line resolution.

- So far we have been using double-line resolution. How many bytes per player does double-line resolution require? (Hint: look back at how we dimensioned each player.)

ANSWER

128

If we were to use single-line resolution we would need twice as many bytes for each player because each bit in memory only lights up one pixel.

- How many bytes would each player need in single resolution?

ANSWER

256

Keep in mind that if we dimension our players for 128 bytes, then we must use double-line resolution. We can't change to single-line resolution, unless we change the way we dimension PM memory. Also, the filler strings (remember them?) must be dimensioned differently because single-line resolution must start on a 2K boundary. (More on single-line resolution later.)

- We can ask for double-line resolution by poking location 559 with 46. How would you code that?

ANSWER

POKE 559,46

Location 559 is called the direct memory access control register (DMACTL). It is used for more than just specifying the type of resolution. We'll talk more about DMACTL later. For now just note that you need to poke it with 46 to ask for double-line resolution. As we did before, you could use DMACTL as a variable in place of "559." At the beginning of the program you could write:

DMACTL=*559*

- What is the other statement needed to ask for double-line resolution?

ANSWER

POKE DMACTL,46

Turning on PMG

Another important register is memory location 53277. called the **graphics control register** (GRACTL). You use GRACTL to "turn on" the PMG system. You have different options here. You can turn on just players, just missiles, or players and missiles. Here's how you do it:

If you want:	Then:
Missiles only	POKE 53277,1
Players only	POKE 53277,2
Players and missiles	POKE 53277,3

- Suppose you want to use players and missiles. How do you specify this with the GRACTL register?

ANSWER

POKE 53277,3

Using Register Abbreviations

If you want, you can also initialize the variable GRACTL to 53277 at the beginning of your program like this:

GRACTL=53277

Then, to turn on PMG with both players and missiles, you could write:

POKE GRACTL,3

- Can you think of a disadvantage in doing it this way? An advantage?

ANSWER

It takes a little more memory to initialize GRACTL to 53277. Also, it uses up a variable. In Atari BASIC you are limited to 128 variables. The advantage is that your code is somewhat easier to read.

Well, we're almost there. Once you have taken care of these steps you have completed the task of setting up PMG. Often you will need to carry out these steps only once, so it's a good idea to put the code needed to do this into a subroutine.

Assigning Line Numbers As a rule of thumb, I suggest you assign high numbers to subroutines that will be performed just once or when speed is not critical. That's because when Atari BASIC executes a subroutine, it has to search for it sequentially, starting with the lower line numbers.

- Because the PMG setup routine will usually be done just once at the beginning of the program, which of these line numbers do you think would be most appropriate for it?

  a. 100
  b. 500
  c. 11000

ANSWER

c. 11000 (That way you will save the lower line numbers for routines where speed is important).

Once the setup routine is complete, all you need to do is say where you want the player to appear on the screen horizontally and vertically. Let's take the horizontal specification first, since it's the easiest. We use a set of registers for that--the horizontal position registers for players (HPOSP0 to HPOSP3).

HPOSP0

HPOSP0 is the horizontal position register for player 0. It is memory location 53248.

Here are the values you poke for various horizontal screen locations:

Check your understanding:

- Write a statement to put player 0 at the:

  1. Left edge of the screen.

  2. Right edge of the screen.

  3. Center of the screen.

ANSWER

1. POKE 53248,50 or POKE HPOS0,50
2. POKE 53248,200 or POKE HPOS0,200
3. POKE 53248,120 or POKE HPOS0,120

- Where do you think the horizontal position register is for:

  1. player 1
  2. player 2
  3. player 3

ANSWER

1. HPOSP1=53249 (one more than 53248)
2. HPOSP2=53250
3. HPOSP3=53251

Often it's handy to use a variable to specify the value to be poked into the horizontal position register. For example, you might use H1 to designate the horizontal coordinate for player 1. Then you would write:

POKE 53248,H1 or POKE HPOS0,H1

More on why this is useful later. Let's now look at how you set the vertical position of a player.

Vertical Position

Atari BASIC doesn't have a vertical position register. So specifying vertical position can be a problem. But because we use string memory for PMG, it's easy to specify vertical position. First, set a variable like, say, "vert" to the desired value:

VERT=30

Then use a string command to set the player string to the data contained in the player image string, like so:

PLAYER0$(VERT)=IMAGE1$

Remember our discussion of how string commands work? The value in parentheses after the string specifies the starting byte. If VERT equals, say, 30, then the data in IMAGE1$ will be placed in PLAYER0$ **starting with the 30th byte**. This has the effect of displaying our player at vertical position 30.

- If we wanted to position our player at vertical position 80, we would poke the image data into the 80th byte of the player string. Altogether our player string has 128 bytes, so how many vertical positions do we have available?

ANSWER

128 (since each player string has 128 bytes).

- Write the two statements needed to put player 1 at vertical position 80. Use the data contained in IMAGE1$. Use the variable VERT to specify the 80th byte of PLAYER1$.

ANSWER

```
VERT=80
PLAYER1$(VERT)=IMAGE1$
```

TRYING IT OUT

Well that's it! Now let's put this all together and get that player on the screen. A complete program appears on the next page. Look it over briefly. Then read the comments that follow it. For your convenience, I have listed the program so that each line has no more than 38 characters. This way the printed listing will match up with your screen listing.

Caution: Be careful in typing X0 and Y0 at lines 13000 and 13010 (and elsewhere in the program). "X0" is "X zero," not "XO." Notice the difference? The zero is more oval shaped than the letter "0."

The zero represents player number 0. "X0" contains the horizontal position for player 0. "Y0" contains the vertical position for player 0. Of course, you can adopt any variable naming scheme that you want. Just be consistent. Don't type "Y0" one time and "YO" the next.

Comments on Program

This program will let you look at all the different colors that a player can be. We have already gone over the code in detail so I will just comment briefly on a few points of interest.

ONSCREEN.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Line 1: GOSUB 2000 As you can see I start the program by "calling" a subroutine at line 2000. The purpose of this subroutine is to take care of what are sometimes called "housekeeping routines." These are routines that you usually only need to do once at the beginning of the program. By using high line numbers for these setup routines, I leave space at the beginning of the program for the main routine. This is preferable because routines requiring transfer of control (GOTOs or GOSUBs) execute faster if they are placed at the beginning. This will become more important later when our main routine becomes more complex.

Notice that the subroutine at line 2000 calls these additional setup routines:

10000 Miscellaneous Initialization
11000 PMG Setup
12000 Drawing of Playfield
13000 Beginning Player Color and Screen Position

The "playfield" referred to here is simply the screen image that the player moves around on. In this case, I have used a few PLOT and DRAWTO statements to create a simple playfield. Notice that I set the graphics mode before doing the PMG setup. This is important because once the PMG setup is done, the graphics mode should not be changed. (Actually, you could change the graphics mode after setting up PMG, but then you need to call the PMG setup routine all over again.)

I won't go into the details of how to create playfields. For details on playfield creation, you may wish to refer to **Designs from Your Mind with Atari** by Tom Rowle (Reston, 1983).

Line 2: GOTO 200 Obviously this line transfers control to line 200. But why? Simply because we want to jump over line 3, which is really not part of the program.

Line 3: SAVE "D:ONSCREEN.SAV":STOP I use this line to save the program to disk after I've made changes to it. It's really quite handy. After I've finished editing the program, I just type G.3 and the program starts saving itself! (Atari understands that "G." is an abbreviation for "GOTO.") Notice the command "STOP" at the end of line 3. This keeps the program from continuing to execute after it saves itself.

Line 200 to 240 Here is where the main "activity" in the program occurs. Notice that I have created a loop in which successive even numbers are poked into location 704, which is the color control register for player 0.

AN ASSIGNMENT

I suggest you type the program into memory. Then save it by typing **G.3**. Next, try running it. If it doesn't work, proofread your work carefully. Watch out for typos! They are especially destructive when you are working with PMG. That's why it's important to save the program before running it.

Once the program is working, you may wish to try modifying it slightly. That's the best way to learn. For example, you might try changing the values following the SETCOLOR command at line 13000. If you do, notice how this affects the color of the playfield and the player. Or, try changing the horizontal and vertical position of the player. You can do that by changing the values assigned to X0 and Y0 in the subroutine starting at line 13000. Happy hacking!

Return to Table of Contents | Previous Chapter | Next Chapter

5

Animating Your Player

So far you have learned to design a player image and carry out the necessary PMG setup routines needed to display it on the screen in any desired color. After finishing this chapter you will be able to move your player around on the screen with or without joystick control.

Animation of a player without joystick control is much easier to program, so let's start with that first.

ANIMATION WITHOUT A JOYSTICK

Once you have a PMG setup routine working, it's quite easy to produce different results with minor program changes. In the program in the previous chapter we established the position of our player with the statements "X0=175" and "Y0=75." We used X0 to specify the horizontal position and Y0 to specify the vertical. In our new animation program, we will still use these statements to tell where we want the player first positioned. But after that we will set up a loop in which we will either increase or decrease the values in X0 and Y0. (We will use X0 for the horizontal position of player 0 since "X" in mathematics traditionally refers to the horizontal plane. Similarly, "Y" usually refers to the vertical plane. So I use Y0 to control the vertical position of the player.)

Main Program Loop

Here is the start of the main loop of our new program.

- As you examine this code, see if you can tell which statement puts the program into a never-ending loop.

```
1 GOSUB 200:GOTO 190
2 SAVE"D:MOVE.SAV":STOP
190 SP=1
200 POKE 53248,X0
220 PLAYER0$(Y0)=IMAGE1$
230 GOTO 230
```

ANSWER

The "GOTO" statement on line 230 puts the program into a never-ending loop.

Note that because the program is looping, the horizontal and vertical positions of the player are constantly being set. With every pass through the loop we have the opportunity to change the player's position. For example, to move the player **up** the screen, all we need do is **decrease** the value in Y0. We can do this with the statement Y0=Y0-1. A better way, though, is to use the statement Y0=Y0-SP, where SP is a variable set to "1." This way you have more flexibility. You can easily set SP to a different value and the player will move more or less rapidly.

- Now, take a look at the code above for a moment and decide where we need to put Y0=Y0-SP to make the player move up the screen.

ANSWER

To make the player move up the screen, we need to insert the statement somewhere between lines 200 and 230. Notice that the statement must be within the loop so that the value in Y0 decreases with each pass through the loop.

Try it. Load the program you typed in from the previous chapter. (You did type it in didn't you?) Then make these changes:

1. Delete lines 2,3,200,210,220,230,240, and 250

2. Change lines 1 and 13020 to read as follows:

1 GOSUB 2000:GOTO 190
13020 POKE 704,88
3. Add these lines:

2 SAVE"D:MOVE1.SAV":STOP
190 SP=1
200 POKE 53248,X0
220 PLAYER0$(Y0)=IMAGE1$
225 Y0=Y0-SP
230 GOTO 200

Important: again, watch out that you don't confuse X0 with XO. "XO" is "X zero." Similarly, "Y0" is "Y zero" not YO. Remember that a zero has a different shape than the letter "0."

Run the program. When you've got the player moving up and off the screen, continue reading below.

Error 5 What happened when the player disappeared off the screen? You should have gotten Error 5: "String Length Exceeded." I'll show you how to handle this problem later. For now, just know that this error is to be expected whenever your player moves too far up or down. It happens whenever Y0 is set to a number less than 1 or greater than 128. That's because PLAYER0$ is dimensioned to 128 bytes. On line 220 we are moving data into PLAYER0$ at the byte specified by Y0. Because there is no such thing as byte 0 for a string, a value of 0 in Y0 results in an error.

As soon as you get Error 5. try this: Print the value in Y0. (Type ? Y0 and press RETURN.) You will notice that Y0 contains "0." an illegal value!

Horizontal Movement

- By now you may already know how to move the player horizontally. Assuming that you take out line 225, what statement would you need to put into the loop to make the player move horizontally across the screen from right to left.?

ANSWER

X0=X0-SP

Try it. First insert "REM" at the beginning of line 225 so that line 225 will not be executed. Then insert X0=X0-SP, as line 227. When you run the program, the player should move from right to left across the screen. Again, don't worry about the error message that occurs when the player runs off the screen. This time you will get Error 3: "Bad Value." The bad value this time will be a -1 in X0. We'll handle this problem later.

Diagonal Movement

Now let's try diagonal movement. First, let's position the player at the top left corner of the screen. We do this by changing lines 13000 and 13010 so that they read:

13000 X0=52
13010 Y0=12

For the moment also change line 230 to "GOTO 230." When you run the program, the player will appear at the top left corner of the screen and just stay there.

- Now let's make him run diagonally. See if you can come up with the statements required to make that happen. The compare your code with mine.

ANSWER

To make the player move diagonally, change lines 225, 227, and 230 as follows:

225 Y0=Y0+SP
227 X0=X0+SP
230 GOTO 200

Try it!

- Now here's another one. Write the statements needed to move the player diagonally (as before). But when she reaches vertical position 45, move her horizontally from left to right until she reaches horizontal position 125, at which point just have her stop. Write the necessary code in the space provided. Then try it out and debug it as necessary. It may take a bit of experimenting to get it to work. Finally, compare your code with mine.

ANSWER

Here's one way of doing it:

```
200 POKE 53248,X0
220 PLAYER0$(Y0)=IMAGE1$
225 Y0=Y0+SP
227 X0=X0+SP
230 IF Y0>45 THEN Y0=45
235 IF X0>125 THEN X0=125
240 GOTO 200
```

Note: Lines 200-227 could be combined into a single line. This would make the player move slightly faster. But to make these programs easier to read, I'm putting each statement on a separate line.

Well, moving a player without a joystick was fairly simple. Moving her with a joystick is not quite so easy.

JOYSTICK CONTROL

To control the movement of a player with a joystick, you need to:

- Read the joystick.
- Figure out the position that the stick has been moved to.
- Change the value of X0 or Y0 accordingly.

Reading the stick is easy. For example, this statement will read joystick 0 (the one on the far left) and put a numerical value in "S:"

- **S=STICK(0)**

Note: The place where you plug in joystick 0 is labeled "Controller Jack 1." Similarly, "Controller Jack 2" is the label for the port for joystick 1, and so on.

This diagram shows the various numbers that will be read into "S" when the joystick is moved to each position.

- Suppose you move the joystick to the right. What will be the value of "S" after the statement S=STICK(0) is executed?

ANSWER
7

- What is the value of "S" when the joystick is not moved, but left in its normal resting position?

ANSWER
15

You can handle the values put into "S" in different ways. Here's one simple way:

```
S=STICK(0)
IF S=7 THEN X0=X0+SP
IF S=11 THEN X0=X0-SP
IF S=14 THEN Y0=Y0-SP
IF S=13 THEN Y0=Y0+SP
IF S=6 THEN Y0=Y0-SP:X0=X0+SP
IF S=10 THEN Y0=Y0-SP:X0=X0-SP
IF S=9 THEN Y0=Y0+SP:X0=X0-SP
IF S=5 THEN Y0=Y0+SP:X0=X0+SP
```

This works. But look at all those IF-statements! They slow down program execution and make it difficult to get smooth and lively animation. Actually, you don't need to use any IF-statements at all. You don't even need the STICK statement.*

For maximum speed, it's better to peek into memory location 632 to read joystick 0. like this:

```
PEEK(632)
```

For joystick 1 we would use "PEEK(633)," for joystick 2. "PEEK(634)" and so on.

Now, to handle the joystick value without "IF-statements," we will simply use a GOSUB command. A powerful feature of Atari BASIC is that you can use a PEEK command in combination with a GOSUB statement.

- Suppose you pull straight back on joystick 0. What value will be in memory location 632? (Look at the joystick drawing if you need to.)

ANSWER

13

Now suppose we write this statement:

GOSUB PEEK(632)

- To which subroutine will the program go if you pull back on the joystick?

ANSWER

It will go to the subroutine starting at line 13 (since location 632 will contain a 13 when the joystick is pulled backward).

*Thanks go to that wild and wacky game programmer, Robert Sombrio, for showing me the fast joystick reading routine presented in this chapter. I tested it against a machine language subroutine published in a major magazine. Robert's routine won hands down!

MODIFYING THE PROGRAM

Now let's modify our program again. At line 10050 let's initialize a variable called "JOYSTICK" to 632. And while we're at it, let's initialize HPOS0 to 53248, and SP to 1 also at line 10050. Then we can peek into "JOYSTICK" to determine which subroutine to execute. Line 10050 will then read:

10050 JOYSTICK=633:HPOSP0=53248:SP=1:RETURN

Our main loop will then look like this:

200 GOSUB PEEK(JOYSTICK):POKE HPOSP0,X0:PLAYER0$(Y0)=IMAGE1$:GOTO 200

Now we also need something at lines 5, 6, 7, 9, 10, 11, 13, 14, and 15. Here are lines 5, 6, 7, 9, and 10:

5 X0=X0+SP:Y0=Y0+SP:RETURN
6 Y0=Y0-SP:X0=X0+SP:RETURN
7 X0=X0+SP:RETURN
9 X0=X0-SP:Y0=Y0+SP:RETURN
10 X0=X0-SP:Y0=Y0-SP:RETURN

- To complete the routine, we need to add lines 11, 13, and 14. See if you can write line 11:

ANSWER

11 X0=X0-SP (If the joystick is moved left, location 632 will contain 11. To move our player to the left, we decrease X0 by one with "X0=X0-1").

- Now write the code for lines 13 and 14. (If location 632 contains 13. the joystick was moved down; if location 632 contains 14. the joystick was moved up.)

ANSWER

13 Y0=Y0+SP:RETURN
14 Y0=Y0-SP:RETURN

- How about line 15? If the joystick is not moved at all, location 632 will contain 15. In this case we don't want to change the X0 or Y0 values at all but simply **return** from our subroutine. Given that information see if you can write line 15.

ANSWER
15 RETURN

To clean up the program, be sure to delete lines, 2, 190, 220, 225, 227, 230, 235, and 240 since they are not needed for joystick control.

Also, after deleting line 190, be sure to change line 1 to:

1 GOSUB 2000:GOTO 200

On the next page you will find a complete listing of the program for moving a player under joystick control. Try it out. When you've got it running, continue reading and we'll make a few more changes.

SETTING DISPLAY PRIORITIES

Another nice feature of PMG is that you can make players hide **behind** certain playfield objects, yet come out in front of others. This is called setting display priorities (or often simply "picking a priority option"). When you are programming in BASIC, you'll use memory location 623 as the **priority register**.* (This register is also used for other purposes, but one thing at a time!)

*Location 623, technically, is the shadow of hardware register 53275. A shadow is a RAM register as opposed to a ROM hardware register in an Atari chip like GTIA. Thirty times a second the operating system takes whatever value is in the shadow register and sticks it into the corresponding hardware register.

You can set various display priorities by poking either 1, 2, 4, or 8 into location 623. **If you poke 1 into location 623**, players will show up in front of all playfields. Furthermore, player 0 will appear in front of player 1, player 1 will appear in front of player 2, and so on.

If you poke 2 into location 623, then players 2 and 3 will appear **behind** all playfield objects. Players 0 and 1 will still appear **in front** of all playfield objects.

If you poke 4 into location 623, then all playfields will appear in front of all players.

If you poke 8 into location 623, then players will go behind some playfields and in front of others. In our sample program players will go **behind** playfield objects drawn with color 1 or 2, but in front of objects drawn with color 3.

MOVE1.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

The best way to learn to use the priority register is to experiment with it.* Let's modify our program to do that by making these seven changes:

*I am deliberately avoiding, here, the usual discussion of "playfields 0 thru 4." That's because it's difficult to define how a specific playfield is created using SETCOLOR, COLOR, PLOT, and DRAWTO statements.

1. Change line 1 so that it reads:

GOSUB 4000:GOSUB 2000:GOTO 200

2. Add this subroutine starting at line 4000:

4000 ? "What number would you like to poke into the priority register?":INPUT PR
4020 RETURN

3. At line 10050 set the variable PRIOR to 623.
4. Delete the RETURN statement at line 11095.
5. Add line 11110 as follows:

POKE PRIOR,PR

6. Put a RETURN at line 11299.
7. Change line 2 to:

SAVE"D:MOVE2.SAV":STOP

After making these changes save the program and then run it. Try poking different numbers (1,2,4,8) into the priority register. Then move your player in front of the different colored playfield objects on the screen. Make a note of the results.

If you're going to take a break, now is a good time. When you come back, we'll discuss speed.

CHANGING SPEED

In some situations you may want to change the speed of a player. For example, in a horse race simulation, you might want to assign different speeds to the different horses. In a baseball game, you might want to assign various running speeds to different players. To do this all you need to do is set SP to different values.

Experiment with Different Speeds

To experiment with different speeds, try adding line 4010 as follows:

?"ENTER A NUMBER BETWEEN 1.0 AND 2.0 TO SET SPEED OF PLAYER.":INPUT SP:RETURN

Also, delete SP=1 from line 10050. Then run the program several times, setting SP to various numbers from 1.0 to 2.0. When you've got the player's speed under your control, continue reading.

Faster Speeds

Now try to set the player's speed even faster--to a number greater than 2.0.

- What happens when you move the player vertically? Horizontally?

ANSWER

Notice that when you move the player vertically, you leave a trail consisting of pieces of the player! But when you move him horizontally, this doesn't happen. That's because horizontal movement is controlled by poking a number into the horizontal position register. Atari then takes care of erasing the player and repositioning him horizontally. But for vertical movement there is no horizontal position register, yet.*

*I suspect that when Atari updates its PMG system, a vertical position register will be added. After all, the Commodore 64 has one! In the meantime, we have to manage with other means to move our players vertically.

To move the player vertically, we have to write the player image data into different bytes of the player image string. Consider for a moment the data that we put into IMAGE1$:

0,0,28,28,8,28,58,89,24,40,76,68,68,0,0

- Notice that the first two bytes of IMAGE1$ contain zeros. What do the last two bytes contain?

ANSWER

They contain zeros also.

These zeros serve the purpose of automatically erasing the player image as we move it around within PLAYER0$. Let's look at some specific examples. Suppose we position the player near the bottom of the screen at vertical position 80. To do this we might use these commands:

Y0=80:PLAYER0$(Y0)=IMAGE1$

The diagram below shows what will be in each byte of PLAYER0$:

Byte	Contents	
1	0	
2	0	
3	0	
"	"	
"	"	
(and so on)		
80	0	
81	0	
82	28	
83	28	
84	8	
85	28	
86	28	
87	89	PLAYER IMAGE DATA
88	24	
89	40	
90	76	
91	68	
92	68	
93	0	
94	0	

Notice that our player image data is located at bytes 80 thru 94. Now suppose we want to move our player upward by one byte. To do this we need to move all 15 bytes upward. For example, the value 76 in byte 90 will be moved up so that byte 89 will contain 76.

- Consider byte 92. Before we move the player, byte 92 contains 68. This is the data for the bottom of the player (his feet). When we move the player image data up one byte, what will be in byte 92?

ANSWER

A zero! This is important. As we move the player data up, the player erases himself.

Since we have two zeros at the beginning of IMAGE1$ and two at the end, we can move the data in one or two byte increments and the "trailing zeros" will automatically erase the player image.

- We can still move the player vertically in three byte increments. But we will need to change IMAGE1$. What changes would we need to make?

ANSWER

We would need to dimension IMAGE1$ for 17 bytes. Then we would need to make bytes 1-3 and bytes 15-17 zeros. IMAGE1$ would then contain this data:

0,0,0,28,28,8,28,58,89,24,40,76,68,68,0,0,0

To make this change, we would need to change line 11300, so the data matches that given above. We would also need to change line 11050 so that it reads:

```
11050 DIM IMAGE$(17)
```

And line 11060 should be:

```
11060 FOR 1=1 TO 17:READ A:IMAGE1$(I,I)=CHR$(A):NEXT I
```

Make those changes. Then try setting SP to 3. Your player will now erase himself completely as he moves vertically at a speed of 3 bytes per move!

LOOKING AT PM DATA

To get a better idea of how PM data is stored in PLAYER0$, try this. Move the player to some vertical position of interest. Then press the SYSTEM RESET key. Next type in and run the following routine. You can run the routine simply by typing "GOTO 600," and pressing RETURN. The routine will show you the contents of PLAYER0$ byte by byte. To temporarily halt the routine, hold down the CONTROL key and press "1." Do the same to restart it.

```
600 GR.0:TRAP 610:FOR I=1 to 128:? I;" ";ASC(PLAYER0$(I)):NEXT I:STOP
610 END
```

Well, we've come a long way. But there is still a lot to learn about PMG! For one thing, although our player moves around the screen at our command, she looks funny because her legs don't move! In the next chapter you'll learn how to fix that.

For your convenience at the end of this chapter is a complete listing of the program. It's set up so you can choose the value to poke into the priority register. You can also choose the value for SP, which determines the speed of the player.

MOVE2.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

6

Making Your Player Dance

Next you will learn how to add life to your player by making his or her legs move. You'll also be able to create other kinds of movements as desired. In addition, you'll learn how to take care of those troublesome errors that occur when you move a player too far off the screen.

Let's start by animating those legs.

DEFINING IMAGES

To make our player's legs move, we need to define three player images, which we will call "LEGS1$, LEGS2$, and LEGS3$."

Here's the first image for our player along with the data needed to create her. (Notice that we have omitted the leading and trailing zeros.)

And here's the second image with accompanying data.

- Now it's your turn. As a review, see if you can calculate the data for this third image:

ANSWER

28,28,8,28,58,89,24,56,72,132,130

INITIALIZING THE IMAGE STRINGS

Now that we've designed the images, we need to initialize our player image strings. Remember how we do it? To initialize our first image, we use a line like this:

11060 FOR I=1 TO 17:READ A:LEGS1$(I,I)=CHR$(A):NEXT I

- The data for this will appear online 11300 like this:

- Why did we do that?

ANSWER

So the player will erase herself when we move her up or down. By using three leading and trailing zeros we can bump the player up three bytes and still get a clean erase.

Now to initialize our second image we simply insert line 11065, which is almost identical to line 11060:

11065 FOR I=1 TO 17:READ A:LEGS2$(I,I)=CHR$(A):NEXT I

Of course we also need to add the data for LEGS2$. We do that on line 11310:

11310 DATA 0,0,0,28,28,8,28,58,89,24,40,76,68,68,0,0,0

- Your turn again. Write the lines needed to initialize the third player image. I'll supply the line numbers.

  11067

  11320

ANSWER
11067 FOR I=1 to 17:READ A:LEGS3$(I,I)=CHR$(A):NEXT I 11320 DATA 0,0,0,28,28,8,28,58,89,24,56,72,132,130,0,0,0

Oh yes, another thing. We need to dimension our new strings. We can easily do this on line 11050:

11050 DIM LEGS1$(17),LEGS2$(17),LEGS3$(17)

- That takes care of setting up the image strings. Next we need to set up a routine to move these images into the PM memory area. Specifically, we will want to move them into which string?

ANSWER
We'll want to move the data into Player0$, which is the PM memory area for PLAYER0.

AN EXPERIMENT

First, enter the last program from the previous chapter into memory. Then make these changes:

1. Delete lines 4000 through 4020 and remove GOSUB 4000 from line 1.

2. Change line 11050 so it reads:

11050 DIM LEGS1$(17),LEGS2$(17),LEGS3$ (17)

3. Add lines 11060, 11065, and 11067:

11060 FOR I=1 to 17:READ A:LEGS1$(I,I)=CHR$(A):NEXT I

11065 FOR I=1 to 17:READ A:LEGS2$(I,I)=CHR$(A):NEXT I

11067 FOR I=1 to 17:READ A:LEGS3$(I,I)=CHR$(A):NEXT I

4. Add these data lines:

11300 DATA 0,0,0,28,28,8,28,58,89,24,60,36,36,102,0,0,0

11310 DATA 0,0,0,28,28,8,28,58,89,24,40,76,68,68,0,0,0

11320 DATA 0,0,0,28,28,8,28,58,89,24,56,72,132,130,0,0,0

5. Change line 200 so it reads like this:

GOSUB PEEK(JOYSTICK):GOSUB MOVELEGS:GOTO 200

6. Insert at line 10050:

MOVELEGS=30:SP=1:PR=8

7. Add a subroutine at line 30 as follows:

30 POKE HPOSP0,X0:PLAYERO$(Y0)=LEGS1$:PLAYERO$(Y0)=LEGS2$:

31 PLAYER0$(Y0)=LEGS3$:RETURN

8. Change line 13040 so it reads:

PLAYER0$(Y0)=LEGS1$

Notice that we are now moving each of the three images, in turn, into the PM memory area.

- Before you run the revised program, write down a prediction as to how it will turn out. Then try it. What's the result?

ANSWER

(I trust you have tried the experiment by now. If you haven't you might want to do that now. You'll learn more that way, honest.) Well, the player's legs are moving all right, but they're moving too fast! We need to slow them down. First, delete lines 30 and 31 and I'll show you how.

One way is to set up a counter. We'll use the variable Z for this. To increment the counter we simply put the statement "Z=Z+1" at the beginning of line 30.

Remember that our animation routine is based on a loop. Each time we pass through the loop, variable Z will be increased by one. Also, our player will be moved to a new position based on the new horizontal and vertical coordinates. Suppose we want the first image to be displayed during the first three times through the animation loop. Then we might use an IF-statement at the end of line 30. Line 30 would then look like this:

Z=Z+1:POKE HPOSP0,X0:IF Z<4 THEN PLAYER0$(Y0)=LEGS1$:RETURN

Notice the "RETURN" statement that sends control back to the main loop at line 200.

- Suppose we want image two to be displayed during loop passes 4 through 6. Code the two statements needed to make that happen. Use line 31.

ANSWER

31 IF Z<7 THEN PLAYEROS(Y0)=LEGS2$:RETURN

Following the same pattern, we might want to display image three during loops 7 through 9. We don't need an IF-statement this time. If Z is 7 or greater, control will simply pass to line 32. At line 32 we simply set PLAYER0$ to the third image. However, when X=9. we will want to reset Z to 0 so that next time image one will be displayed.

- See if you can code the statements for line 32.

ANSWER

Here's how I did it:

32 PLAYER0$(Y0)=LEGS3$:IF Z=9 THEN Z=0:RETURN

- We need to add a statement online 33 before this routine will work. See if you can figure out what it is. Here's a hint. What happens if X=7? Control will pass to line 32, and the third image will be moved into PLAYER0$. But since Z is not equal to 9 . . . what will happen?

ANSWER

Since Z is not equal to 9, control will pass to the next statement, whatever it may happen to be, and we will never return from the subroutine. This will lead to problems! Consequently, line 33 needs to be "RETURN" so that control will return to the main loop.

Try it. Make sure lines 30 through 33 read like this:

30 Z=Z+1:POKE HPOSP0,X0:IF Z<4 THEN PLAYER0$(Y0)=LEGS1$:RETURN

31 IF Z<7 THEN PLAYER0$(Y0)=LEGS2$:RETURN

32 PLAYER0$(Y0)=LEGS3$:IF X=9 THEN X=0:RETURN

33 RETURN

- Try this out. What additional problem is there?

ANSWER

The player's legs move even when he is standing still! Fortunately, this is easy to fix. Just change line 15 to read:

15 POP:PLAYER0$(Y0)=LEGS1$:GOTO 200

Control is passed to line 15 when the joystick is not moved--when the player is stationary. So we can use this line to display a single image and then jump back to line 200. The POP command serves to cancel the GOSUB that passed control to line 15. By using "POP" we make it "legal" to use the "GOTO 200" statement. Without the POP statement, you would eventually end up with an error message as a result of "GOTO 200." That's because BASIC is set up to expect a RETURN statement at the end of a subroutine.

Try adding line 15. If everything goes well, you will find that the player's legs move only when you move the joystick! If you want, you can experiment with a different "standing still" image of the player. For example, you might code line 15 like this:

15 POP:PLAYER0$(Y0)=LEGS2$:GOTO 200

With this statement, image 2 will be displayed when you leave the joystick in the upright position. The program will now work fine. If yours doesn't, you may wish to compare it with the listing at the end of this chapter.

TRAPPING ERRORS

I promised you I'd explain how to handle those error messages that pop up when you move the player too far off the screen. Here's one simple way:

1. Change line 1 so it reads like this:

1 TRAP 3000:GOSUB 2000:GOTO 200

2. Add line 3000:

3000 PLAYER0$=BUFFER$:Y0=128:X0=80:Z=0:TRAP 3000:GOTO 200

The TRAP at line 1 says. "Trap any error you find. Don't print an error message. Instead, just GOTO line 3000.

- What do you think line 3000 says?

ANSWER

Erase the player image from the screen. (BUFFER$ is set to zeros, so setting PLAYER0$ equal to BUFFER$ is the same as setting PLAYER0 to zeros. That, in effect, erases the PLAYERS image.)

After that, the variables Y0 and X0 are set to some specific values. The value of 128 for Y0 hides the player off the bottom of the screen. The value 80 for X0 puts the PLAYER$ at the left of the center of the screen. The result is that if you move the player too far off the screen in any direction, she will be automatically repositioned off the bottom of the screen on the left side. To bring her back,just push forward on the joystick and she will scoot up onto the screen.

The idea here is that the error condition is caused by a bad value in either X0 or Y0. To fix that, we simply trap the error and reset X0 and Y0 to any desired "legal" values. In addition, we reset the leg movement counter (Z) to zero.

Also, notice that we repeat the TRAP command at line 3000, so that if another error occurs, we will again branch to line 3000. (A TRAP statement must be reset after an error occurs.)

If you haven't yet got your player's legs to move, now's the time. You may wish to refer to the following program, which is a complete listing of all the lines needed to make your player dance!

Once you have it working, you may wish to animate your player in other ways. For example, you might want to add some additional images to make the legs move more smoothly--or you may wish to have the player move her arms or other body parts. Have fun!

LEGS.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

7

Adding Sound

Now that you have your player dancing around the screen, let's
add some sound effects! They can help to make an exciting animation sequence even better.

When you finish this chapter you'll be able to create some interesting sound effects to go along with the movement of your players. In addition, you'll learn how to maximize the execution speed of these sound routines.

I'm going to begin with a brief discussion of the SOUND statement. If you're already an expert on this, you may wish to skip ahead to "Chords" in this chapter. Otherwise, read on.

THE SOUND STATEMENT

Most Atari BASIC programs use the SOUND statement. It's a powerful statement because it enables you to control the pitch, distortion, and volume for each of four "voices."

Yes, Atari has four voices. Before we get into the details of the SOUND statement, let's consider what is meant by the term "voice." Think of a four-voice choir. In such a choir there are four separate groups of singers. Each group (or "voice") usually sings a separate melody (series of notes). Here are the names and ranges of the different voices in a four-voice choir:

Name of Voice	Range
Soprano	(Usually sings highest notes)
Alto	(Next to the highest notes)
Tenor	(Next to the lowest notes)
Bass	(Lowest notes)

Now back to the SOUND statement. In Atari BASIC the names of the four voices are 0, 1, 2, 3. (Again, notice that we start counting with "0" rather than "1.") Here's how you specify the voice, pitch, distortion, and loudness with a sound statement:

SOUND VOICE, PITCH, DISTORTION, VOLUME

"SOUND" is a key word such as "GOTO." The other items ("VOICE," "PITCH," "DISTORTION," and "VOLUME") are called parameters. Let's call them "palms" for short. In the above example they are variables. Before executing the above statement, you would need to set these variables to appropriate values.

Alternatively, you could use fixed numerical values (constants) or arithmetic expressions in place of those variables. For example, you might write:

SOUND 0,121,10,8 or SOUND 0,5*20+1,10,8

Both statements would cause voice 0 to play the note called "Middle C" since a value of 121="Middle C." (See page 58 in your ATARI BASIC REFERENCE MANUAL for numbers corresponding to the various musical pitches.)

The "10" in the above statement (the third parm) creates a "pure tone"--one with no distortion. Instead of "10," you can use other even numbers between 0 and 14 for various sound effects.

The fourth parm tells Atari how loud you want the sound to be. This value can be between 1 and 15. The higher the number, the louder the sound.

Before we go any further, let's experiment with a few simple SOUND statements. Get your Atari up and running. Then type in this next command just like it is--without a line number:

SOUND 0,121,10,8

- When you press RETURN the command will immediately execute. What note will you hear?

ANSWER

Middle C. (121 is the value for Middle C, remember.)

Try it. Then enter this command:

STOP

- What happened?

ANSWER

Nothing really. The sound will continue. The STOP command has no effect on a SOUND command. To turn off the sound, simply enter the END command. Try it.

The END command can come in handy when you decide you want some peace and quiet for a change.

Experiment

Move the cursor back up to the SOUND command you entered earlier. (To move the cursor, hold down the CTRL key while pressing the arrow keys.) Try changing the distortion parameter (the third one). Enter various even numbers from 0 to 14. If you hear an effect you like, you may wish to make a note of the SOUND parms. In the same way, experiment with the volume parm by entering various numbers from 1 to 15.

CHORDS

A chord is a combination of musical pitches. It's easy to make chords with the Atari. You simple turn on more than one voice. Try this example. It will produce what musician's call a C-major chord:

SOUND 0,243,10,8
SOUND 1,121,10,8
SOUND 2,96,10,8
SOUND 3,81,10,8

One caution: if the total of all the volume parms exceeds 32, an unpleasant "clipped" tone will result.

- In the example above, did I observe this caution? What is the total of the volume parameters in the example?

ANSWER

Yes. The volume parms add up to exactly 32 (8+8+8+8=32).

POKING SOUND PARMS

Because of its ease of use, you may choose to use the SOUND command in your programs. But if you are coding an animation sequence for, say, a game or other simulation, then it may be better to poke sound parms directly into memory. SOUND statements execute more slowly than direct pokes.

Here's how you poke sound parms. To set the pitch of voice 0, simply poke a number into memory location 53760. To set the distortion and volume, first multiply the desired distortion number by 16 and add it to the value for volume. Then poke this one number into 53261.

For example, instead of writing SOUND 0,121,10,8, you would write:

POKE 53760,121
POKE 53761,168

(We poked 168 into 53761 because 16*10+8=168.)

- Now you try it. What POKE commands would be needed to replace this SOUND command?

  SOUND 0,96,5,5

ANSWER

POKE 53760,96
POKE 53761,85 (since 5*16+5=85)

(Try these out. Enter them directly without line numbers.)

SOUND REGISTERS

So far I've shown you only two sound control registers: 53760 and 53761. Here is a more complete list:

Location	Used to Control
53760	Pitch of voice 0
53761	Distortion and volume of voice 0
53762	Pitch of voice 1
53763	Distortion and volume of voice 1
53764	Pitch of voice 2
53765	Distortion and volume of voice 2
53766	Pitch of voice 3
53767	Distortion and volume of voice 3
53768	Pitch of all voices (More on this later)

- Look over these locations for a moment. What do the even numbered locations all control? The odd ones?

ANSWER

The even numbered locations all control pitch. The odd ones control distortion and volume.

MORE EXPERIMENTS

Now let's try some more experiments. Type in and try out this little sound demo program. Then read on.

GLODE.BAS
Download (Saved BASIC)

Download / View (Listed BASIC)

Notes on the Sound Demo

Line 60: POKE S0,P0 This is where I specify what pitch to make the sound. S0 is the pitch control register for voice 0. (I initialized S0 to 53760 in the subroutine at line 1000.) P0 will be some value between 1 and 255. That's because you are asked to enter such a number at line 20.

Line 70: POKE DV0,D0 Here I turn on the distortion and volume of voice 0. (I initialized DV0 to 53762 in line 1000. 53762 is the distortion and volume control register for voice 0.) The value contained in D0 was calculated on line 35.

Line 35 D0=16*DIST+VOL Here I set D0 to the value that is derived from the desired distortion and volume. As I mentioned earlier, we do this by first multiplying the desired distortion by 16 and then adding the desired volume. You'll notice that DIST is set at line 30 in response to the prompt "ENTER DISTORTION." I initialized VOL to 8 in line 1000. I could have set it to any value from 0 to 15.

50 FOR P0=1 TO 255 STEP ST This is the beginning of a FOR/NEXT loop. Note the part that says "STEP ST." This command tells the computer how much to increment P0 at each pass through the loop. ST is set at line 30 in response to the prompt "ENTER GLIDING SPEED." If ST is set to, say, 5 then P0 will be set to 1 the first time through the loop, but 6 the second time (since 1+5=6).

- Suppose we set ST to 20. What will P0 equal on the second pass through the loop?

ANSWER
21 (1+20)

So the higher the value of ST, the bigger change in P0 at each pass through the loop. When ST is set to, say, 1, there will be a slow, smooth gliding pitch. When ST is set to progressively higher values, the sound changes in pitch rapidly and has a shorter duration (since it takes fewer times through the loop to reach 255).

Line 90: POKE DV0,0 This simply turns off the volume for voice 0. I do this so that the sound stops when control returns to the initial prompt at line 20. I could also have turned off the sound by poking a 0 into D0.

Experiment

If you haven't already done so. I suggest you try experimenting with different values for distortion (DIST) and gliding speed (ST).

Also, you might want to modify the program to make it easier to hear the various sounds being produced. For example you might put a delay on the FOR/NEXT loop. You could do this by adding this line to the program:

```
75 FOR PAUSE=1 to 100:NEXT PAUSE
```

Another idea would be to print the value of P0 on the screen so you could observe the values being poked into S0.

USING A LOOP

Notice how I produced a gliding effect by putting the poke statements inside of a loop. In the same way you can put these poke statements within your animation loop. This way, you get a kind of gliding effect as your player or missile moves around the screen.

Assignment

Let's try this with the moving legs routine we developed in the last chapter. What we will do is create some interesting "traveling sounds" for our player. Whenever the player moves we'll play some sounds. When the player stops, the sound will stop.

Perhaps you have some ideas about how this might be done. If so, you might want to experiment on your own before continuing.

Traveling Sounds

Take a look at the program you entered in the previous chapter (which I called "LEGS.SAV"). One approach to making traveling sounds might be to add a "sound statement" to the "MOVELEGS" subroutine, which begins at line 30.

A simple way to do this would be to use the variable Z to set the pitch of the sound. As you will recall we used Z as a counter to control the display of various running images of our player. When Z was less than 4, we displayed "LEG1$". When Z was in the range of 4 through 6, we displayed "LEG2$," and so on. We added 1 to Z at the beginning of the MOVELEGS routine so that each time the routine was called, Z was greater than before. When Z reached 9, it was reset to 0.

Well, we can make Z do double work. Here's how we might do it with a sound statement:

```
SOUND 0,Z,10,8
```

- Based on what I've said so far, at what line might you insert this sound statement? What would the revised line look like?

ANSWER

You might insert the statement into line 30, like this:

```
30 Z=Z+1:SOUND 0,Z,10,8:
    IF Z<4 THEN PLAYER0$(Y0)=LEGS1$:RETURN
```

Before you continue reading, you might want to revise the "LEGS" program by changing line 30 as we've just discussed.

USING POKES

Suppose you were to use Poke statements to create the sound. First, you might initialize these variables:

S0=53760
DV0=53761

- What would S0 stand for? DV0?

ANSWER

S0 would stand for the pitch control register for voice 0. DV0 would stand for the distortion/volume control register for voice 0.

- What would line 30 look like if you used Poke statements to create sound?

ANSWER

```
30 Z=Z+1:POKE S0,Z:POKE DV0,168:
IF Z<4 THEN PLAYER0$(Y0)=LEGS1$:RETURN
```

(Recall that to find the proper value for a distortion of 10 and a volume of 8, we multiply 16 times the distortion. This gives us 160. We then add the volume (8) to 160 to arrive at the DV0 value of 168.)

ADDING VARIETY

To give more variety to the sound, you might consider multiplying Z by some number. For example:

POKE S0,Z*TONE:POKE DV0,Z*TONE

With this code, different sounds will be produced depending on the value of TONE. For example if TONE=20 then the first time through the loop Z*TONE will equal 20; the second, 40; the third 60; and so on. The bigger the value in TONE the bigger the jump in pitch that will occur on each pass through the MOVELEGS routine.

At the end of this chapter is a program that will let you experiment with different "traveling sounds" for your player. It is similar to the LEGS program from the previous chapter. I have circled lines that need to be added or changed.

As you will probably notice, the addition of sound to the program slows down the player somewhat. That's one of the trade-offs. The more features you add--like sound, missile firing, collision detection, and so on, the bigger your loop becomes and the slower the animation. That's why I've stressed techniques that will help you maximize execution speed--such as poking values into sound registers and placing frequently executed code early in the program. Of course, machine language is probably the best--although the hardest--way to obtain fast player/missile action.*

Well, speaking of missiles, in our next chapter let's take a look at what they are and how to use them.

*When you're ready to dirty your hands with machine language, look for my new book on **Player Missile Graphics in Assembly Language.**

SOUNDRUN.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

8

Missiles

As I mentioned earlier, Atari Player-Missile Graphics was originally designed to simplify the creation of arcade-style games. These games often include little creatures who fire missiles (or bullets) at each other. In this chapter you'll learn to create and animate these missiles. Keep in mind that missiles can also serve other purposes: they can highlight text in graphics mode 0, for example.

DEFINING THE MISSILE IMAGE

There are four missiles available in PMG. All four missiles are stored in a single byte in "missile memory."

- How many bits wide is a single missile? (Keep in mind that a byte is made up of eight bits.)

ANSWER

Each missile is two bits wide (in contrast to players, which are eight bits wide).

Let's look at a map of a single byte of the missile memory area:

Since this byte has nothing but zeros in it, no missiles would appear on the screen. Now if we wanted to make missile 1 appear on the screen, we might put this binary data into a byte of missile memory:

- What would be the decimal number needed to turn on missile 1? (Remember how we calculated those decimal numbers for players? We do the same thing with missiles. The bit on the far right is worth 1; the next bit to the left is worth 2; the next bit to the left, 4; and so on. Each successive bit to the left is worth twice as much as the previous bit.)

ANSWER

We need a decimal 12 to create the necessary binary data in missile memory since a binary 00001100 = a decimal 12.

- If we wanted a thinner missile 1, what binary data might we use?

ANSWER

We could use either 00000100 or 00001000.

So if we wanted a thin missile 1, we might use a decimal 4 or 8 to put the proper binary data into missile memory.

- Ok, try this one. What binary number would you use to turn on missile 0 so that it has full width? What would the corresponding decimal number be?
  Binary Number:

  Decimal Number:

ANSWER

The binary number would be 00000011; the decimal number. 3.

- Suppose you wanted to turn on both bits of missiles 0 and 1. What binary number would you need? Decimal number?
  Binary Number:

  Decimal Number:

ANSWER

You'd need a binary 00001111 or a decimal 15.

Ok, so let's say we want to turn on both bits of missile 0. We know that we need to put a 3 into missile memory to do this since a binary 00000011 = 3. But what specifically do we do to code this?

Image String

It's fairly simple. We just create an "image string" similar to the strings we defined earlier. For a missile, a single byte string may be enough. Remember, the more bytes in your image string, the **taller** the player or missile will be.

Once we have the data in the image string we can easily zap it into the missile memory area at machine language speed. This only works, of course, because we have carefully defined our PMG area with strings.

- Let's call our missile image string MIMAGE$. We need to do two things to set up MIMAGE$: dimension it and then plug in a decimal 3. See if you can write the code to do it. (It will go into your new program as line 11070.)

ANSWER

Here's one way:

11070 DIM MIMAGE0$(1):MIMAGE0$=CHR$(3)

(I suggest you enter SOUNDRUN.SAV--the program from the previous chapter--and add the new statements as you read along.)

HORIZONTAL POSITION REGISTER

Let's initialize another variable for the horizontal position register for missile 0. We'll call it HPOSM0. So at line 10060, we add this statement:

HPOMS0=53252

(Note that HPOSM0 ends with a zero, **not** the letter "0.")

Another minor change to make in SOUNDRUN is the automatic save routine at line 2. Change it to:

2 SAVE"D:MISSILE.SAV":STOP

That way you won't wipe out the previous program when you save this one.

REVISING THE MAIN LOOP

Since we will be using the joystick button to fire missiles, we need to delete line 200 from the SOUNDRUN program. After deleting line 200, renumber line 204 so that it now becomes 200. Line 200 will now read:

200 GOSUB PEEK(JOYSTICK):GOSUB MOVELEGS:GOTO 200

Also, let's delete these pokes from line 30:

POKE S0,X*TONE
POKE DV0,D0

since we won't be including player traveling sounds this time.

- In a moment we are going to add a routine to move our missile. We want line 200 to fall through to this new routine. So what change do we need to make to our new line 200 shown above?

ANSWER

We need to delete GOTO 200; otherwise control will not fall through to our new routine.

BUILDING A MISSILE MOVE ROUTINE

As you know, we have set up our player move routine so that our player can move in any one of the eight joystick directions (up, down, left, right, plus the four diagonal directions). Wouldn't it be nice if our player could also fire a missile in any one of those directions?

To make this work, let's make a rule that a player must be moving when she fires a missile. Furthermore, the missile will go in the same direction as the player. Now, how do we write the code to accomplish this?

First, we need to identify when the fire button is pressed while the stick is also being deflected from its upright position. If that happens, we will set a variable called MOVE0 to the stick value. For example, if the stick is moved to the left while the fire button is pressed, we will set MOVE0 to 11. (The stick value for left is 11.) Also we will increment an "indicator"--a variable that we will call FIRE0. This variable will help the routine remember that the joystick was pressed. Also, it will serve as a counter. It will count the number of passes that have been made through the missile move routine. We need this because we need to do different things on the first pass through the loop than on successive passes.

- Got all that? See if you can write the code for it.

ANSWER

Here's how I did it:

IF PEEK(BUTTON)=0 AND PEEK(JOYSTICK)<>15 THEN MOVE0=PEEK(JOYSTICK):FIRE0=FIRE0+1

Of course, Atari won't pay any attention to this fancy indentation. But it does make it easier to read. It's also easier for me to type on my ATARI TEXT WIZARD. This will be line 301. (Yes, there is a line 300. But it will come later. For now just put a REM at line 300.)

Next, we need to be able to jump out of this loop if the fire button was not pressed. We can't just peek at BUTTON0 to decide whether to leave the missile routine because BUTTON0 will return to zero as soon as the button is released.* If we relied on BUTTON0, then on the second pass through the loop our missile would stop.

So we look at FIRE0. If it equals 0 then we know we need to return to line 200. The code would then be:

305 IF FIRE0=0 THEN GOTO 200

*Technical note: there is a way to "latch" the fire button so that location 644 is not reset as soon as the button is released. I find it easier to use this method, however.

Positioning the Missile

The first time through the missile move routine, we need to plop the missile down on the screen "underneath" our player so that when the missile is moved on subsequent passes through the loop, the missile will appear to be coming from him.

- How does the routine detect the first pass?

ANSWER

With a statement such as IF FIRE0=1 THEN ....

On first pass through the missile move routine we will **also** want to:

- Set DIR0 to a number representing the direction we want the missile to move. (Again that will be the familiar joystick value such as 7 for right, 14 for up, and so on.)
- Add 1 to FIRE0 so that on the next pass through the loop FIRE0 will be greater than 1.
- Consider for a moment what the code might look like for doing that:

ANSWER

Here's one way:

315 IF FIRE0=1 THEN MX0=X0+3:MY0=Y0+7:DIR0=MOVEO:FIRE0=FIRE0+1

Note that with this code, DIR0 will be set to the stick deflection value only on the first pass through the loop. Also notice that I set MY0 to Y0+7. That's because I wanted to move the missile down a bit from the player image. Remember that the player image has zeros at the top so she erases herself as she moves down the screen.* I set MX0 to X0+3 because a value of X0 would put the missile at the far left edge of the player image--I want the missile to be hidden "underneath" the player.

*By now you have probably noticed that I sometimes refer to our player as male and sometimes as female. Working with a feminist writer taught me to do this as a way to avoid sexist writing. I hope it doesn't jar you too much. Of course, you might still complain that the player doesn't look at all female--oh well.

Putting the Missile on the Screen

- Now we're ready to put the missile on the screen! We simply erase the data in MISSILES$, POKE HPOSM0 with MX0, and set MISSILES$(MY0) equal to our missile image string. Can you write the code?

ANSWER

335 MISSILES$:BUFFER$:POKE HPOSM0,MX0:MISSILES$(MY0)=MIMAGE0$

Also, we'll need to make these changes:

Add this to line 10060:

HPOSM0=53252

Add line 11070:

11070 DIM MIMAGEO$(1):MIMAGEO$=CHR$(3)

Change lines 13000 and 13010 to:

13000 X0=120
13010 Y0+43

Change line 2 to:

SAVE"D:MISSILE.SAV":STOP

Change line 14000 to:

14000 ? "To fire missile, move joystick and push fire button.":RETURN

Change line 15 so it reads:

15 POP:PLAYER0$(Y0)=LEGS1$:GOTO 300

Delete lines 1000 through 1020.

Now (finally) we're about ready to take a preliminary test run of our missile routine. (Assuming you've typed in all of the preceding lines along the way.) But first, in line 315 change MX0=X0+3 to:

MX0=X0-10

Why? Because I want you to be able to see the missile so you can verify that your routine is working at this point. Otherwise, the missile will be positioned directionally on top of the player and you won't be able to see it. (Remember, the missile is normally the same color as the player it belongs to.)

Also, put a stop at line 340 just to temporarily freeze the action so you can see exactly where the missile appears when you press the fire button.

I've covered a lot, I know. So for your convenience, here is a listing of the new program with the changes circled,

MISSIL1.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

When your new program is working properly, the missile will appear to the left of the player. That's because we set the horizontal coordinate (MX0) to ten less than the player coordinate. Now that we've got the missile on the screen, let's see about moving it. But first, go back and change line 315 so it says MX0=X0+3.

Moving the Missile

Now let's get on with animating that missile. For now, delete the TRAP statement from line 1. Now, remember lines 5 through 15? Let's review. We used them to adjust the coordinates for the player's position. If the stick reading is 7 (right), then we GOSUB 7. Line 7 is a "minisubroutine" that increments X0 and immediately returns. We can use that same approach to move our missiles. To do that; we will write a similar routine that starts at an **offset** of 60 from the first one. That is, instead of beginning at line 5, our new missile subroutine will start at line 65 (5+60). Instead of adjusting X0 and Y0, we will adjust MX0 and MY0, the missile coordinates. Here is the complete missile coordinate adjustment routine:

```
65 MX0=MX0+MS0:MY0=MY0+MS0:RETURN
66 MY0=MY0-MS0:MX0=MX0+MS0:RETURN
67 MX0=MX0+MS0:RETURN
69 MX0=MX0-MS0:MY0=MY0+MS0:RETURN
70 MX0=MX0-MS0:MY0=MY0-MS0:RETURN
71 MX0=MX0-MS0:RETURN
73 MY0=MY0+NS0:RETURN
74 MY0=MY0-MS0:RETURN
```

Notice that we are now using MS0 to increment or decrement our missile coordinates. (MS0=Speed of Missile 0.)

In addition to the above lines, we will need these lines in our missile move routine:

```
320 IF FIRE0>1 THEN GOSUB DIR0+60
449 GOTO 200
```

- According to line 320, where will control pass if the joystick is pushed forward? (Hint: the number 14 is returned when you push the stick forward.)

ANSWER

Control will pass to line 74 (14+60)

What happens at line 74?

ANSWER

We decrement MY0, and the effect is that our missile will move up the screen. (The smaller MY0, the higher will be the position of the missile on the screen.)

Let's try it. In summary:

1. Add lines 65 through 75 above.
2. Add line 320.
3. In line 10060 initialize MS0 to 6 (MS0=6).
4. Delete line 340.
5. Add line 499.
6. Delete the TRAP statement from line 1.

Then, save and then run the program. If all went well, your player can now fire the missile in any one of six directions!

ERROR ROUTINE

Having played around a bit firing missiles, you are probably tired of seeing that error message pop up. You know, the one that comes up every time you fire a missile off the screen. That's fairly easy to fix. Let's write an error correction routine for it.

Once again, let's begin our program with a TRAP statement at line 1:

```
1 TRAP 3000:GOSUB 2000:GOTO 200
```

Whenever an error occurs, program control will branch to line 3000. I've used a high line number so that the trap routine is out of the way of our main loop. Now when an error occurs, control will pass to an error correction routine at line 3000. The error correction routine will check for various error conditions and then correct them. I know, we did this in a previous chapter, but this time the correction routine will be a bit more detailed.

First, let's handle the situation where the player moves off the screen too far to the right. Horizontal coordinate 210 is just off the screen to the right. So if the X0 coordinate is greater than 210 let's reset X0 to 210. The statement IF X0>210 THEN X0=210 will do that nicely.

In the same way we can reset the other X0 and Y0 values if they move too far off the screen. Like this:

```
3000 IF X0>210 THEN X0=210
3010 IF X0<39 THEN X0=39
3020 IF Y0<1 THEN Y0=1
3030 IF Y0>128 THEN Y0=128
```

In addition, we need to reset the leg movement counter Z. We can do that with the statement Z=0. Let's add it at line 3035:

```
3035 Z=0
```

That takes care of our player. Now she can hide off the screen--even move around off-screen and then come back.

Let's fix the missile, too. Here's a simple way to do it for now: reset the X0 and Y0 coordinates whenever an error occurs. Reset them to what? Well, let's simply reset them to the player's coordinates, like this:

```
3040 MX0=X0:MY0=Y0
```

Also, we need to turn off the missile sound when an error occurs. We can do that simply by setting D1 to zero:

```
3050 D1=0
```

Next, we need to turn off the missile routine, so it doesn't continue to adjust the MX0 and MY0 coordinates. We do that by setting FIRE0 to zero:

```
3060 FIRE0=0
```

Finally, we need to include another TRAP statement so that if another error occurs control will once again return to line 3000. Then we can jump back to line 300 in the main routine:

```
3080 TRAP 3000:GOTO 200
```

Add these lines. Then you'll be able to fire missiles from an ambush position--while the player is hiding off the screen!

Our missile move routine is now pretty much finished. But let's add a few embellishments. First, let's add sound to the firing of the missile.

Add line 300 as follows:

```
300 POKE SD1,P1:POKE DV1,D1
```

At line 10070 initialize P1, D1, SD1, and DV1 as follows:

P1=10 (Pitch for sound)
D1=0 (Distortion/Volume)
SD1=53762 (Pitch Register)
DV1=53763 (Distortion/Volume Register)

Also, remove the RETURN from line 10060 and put RETURN at line 10900. Now we can use variable D1 to set the volume of our missile sound. A good place to do this is on the first pass through the missile loop. If we set the volume to maximum on the first pass and then decrement it on each successive pass, we can create an explosive type sound that gradually fades away. We can set the volume to maximum by writing D1=15.

- Where would be a good place to insert that statement?

ANSWER

At line 315 where we test to see if we are on the first pass through the missile routine. So add D1=15 to the end of line 315.

Now we still need to decrement D1. Let's do that at line 330. We can decrement D1 easily enough with D1=D1-1. But that's not enough, because that way D1 will eventually become a minus value and we will get an error message.

- How can we avoid that? Write the code.

ANSWER

Here's how I did it:

330 D1=D1-1:IF D1<1 THEN D1=0

Add line 330. Save the program and run it. You will now hear an explosive sound whenever you fire a missile.

MISSILE SIZE REGISTER

Now let's play with the missile size register. That's a register that lets us instantly increase or decrease the width of a missile. We can set a missile's size to normal, double, or quadruple width. The missile size register, at location 53260, controls the size of all missiles.

Here's a chart showing the proper numbers to poke in for various missile sizes.

Missile Number	Normal	Double	Quadruple
0	0	1	3
1	0	4	12
2	0	16	48
3	0	64	192

Important: if you want to set the size of two or more missiles at once, then add the proper values for each missile depending on the desired size. Then poke the **total** into location 53260.

For example, to set missile 3 to quadruple size and missile 1 to double size, you would poke 196 into 53260. (192 gives quad size for missile 3, and 4 gives double size for missile 1. We get 196 by adding 192 and 4.)

- Suppose you wanted to set missile 0 to quad size and missile 1 to double size? What number would you poke into 53260?

ANSWER

7 (3+4=7)

- How would you set missiles 0 and 1 to double size?

ANSWER

POKE 53260,5

To make our program easier to read, let's set up a variable called SIZEM (size of missiles) at line 10070, like so: SIZEM=53260.

We can create a nice effect by poking a random number into SIZEM. First (also at line 10070) let's set up a variable called RANDOM: RANDOM=53770. Location 53770 is constantly being updated by Atari's operating system with a random number. So a quick way to get a random number is to "peek" into location 53770.

- Write a statement to poke a random number into the size register for missiles.

ANSWER

Here's one easy way:

POKE SIZEM,PEEK(RANDOM)

Try it. Be sure to initialize SIZEM and RANDOM at line 10070. Then add POKE SIZEM,PEEK(RANDOM) to line 300.

You'll now see the missile randomly expand to various widths as it sails across the screen. Notice the different effect produced by firing the missile vertically versus firing it horizontally.

COMPLETE LISTING

Here is a complete listing of the missile program that we have developed in this chapter. Again. I have circled the lines that are to be changed or added. Be sure to save it to disk or tape. You'll need it. Remember, each chapter will build on the previous program in some way.

In the next chapter, you learn how to create single-line resolution PMG!

MISSIL2.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

9

Single-line Resolution

So far we have been dealing strictly with double-line resolution.
Since double-line resolution is easier to handle and takes less memory. I'll continue to use it for most of the examples in this book. But our little player is so cute in single-line resolution! I just couldn't resist introducing him to you.

Be sure to save copies of the previous **double-line** resolution programs, since future programs will be based on them--not on the program in this chapter. In fact, I suggest you just read over this chapter without typing in any of the examples. Then after you've mastered double-line resolution, come back and try out the program in this chapter. (Whatever. You'll do it your way, anyway.)

At the end of this chapter is a complete listing of a program that does essentially the same thing as the program in the previous chapter--the only difference is that it is in single-line resolution. The program is called SLRES.SAV--short for "single-line resolution." Here is a summary of the changes you'll need to make to MISSILE.SAV to produce SLRES.SAV:

LINE 11000

This is the "filler" line that makes sure that PM memory starts on a 2K boundary. The only change is that while the number 1024 appeared in the previous program, 2048 appears in this program (1024=1K: 2048=2K).

LINE 11010

In the previous program I had a string called BUFFER$ dimensioned to 384 bytes. Well, in single-line resolution, this string needs to be twice as many bytes: 384*2 or 768.

Instead of dimensioning BUFFER$ to 768 bytes, I broke it down into three strings of 256 bytes each: ERASE$, FILLER3$, and FILLER4$. (Notice that it works out to be the same thing since 3*256=768.) Also, notice that each of the other strings are now 256 bytes long instead of 128--the way they were in double-line resolution. And notice that I omitted the dimensioning of PLAYER1$, PLAYER2$, and PLAYER3$.

LINES 11020 THROUGH 11040

Here I am setting ERASE$ to binary zeros. Notice the RETURN at the end of line 1040. I broke the PMG setup routine into two parts. (For some unknown reason BASIC likes it better this way.)

Lines 11045 through 11095 make up the next part of the PMG setup. I made these lines into another subroutine, which is called immediately after the subroutine starting at line 11000. (See the "executive" subroutine beginning

at line 2000, which calls the various setup routines.)

LINES 11045 AND 11047

Here I am initializing MISSILES$ and PLAYER0$ in a slightly different way. This is necessary because of an apparent bug in Atari BASIC. It seems that a statement such as MISSILES$=ERASE$ won't work if MISSILES$ is 256 bytes or more.

In the previous program, line 335 contained the statement MISSILES$=BUFFER$, which served to "erase" data from the missile memory area. (Actually it fills MISSILES$ with zeros.) I had to take this statement out when converting the program to single-line resolution. That's because of the apparent bug in Atari BASIC that I just mentioned.

So to accomplish vertical missile movement, I had to find another way to erase the missile. I did that by rewriting lines 11070 and 11330.

LINES 11070 AND 11330

At line 11070 I dimensioned the missile image string to 13, instead of 1. Why? Because I wanted to have the missile erase itself with trailing zeros during vertical movement.

Notice the zeros in the missile image data at line 11330. These zeros erase the missile as it moves vertically. Using this arrangement, I found that I didn't need the statement "MISSILES$=ERASE$" at line 335.

LINE 11085

This is important. Here I specify single resolution by adding 16 to the 46 that is to be poked into location 559. the direct memory access control register. I'll explain this more later. (See the "Odds and Ends" chapter if you can't wait.)

LINES 5 THROUGH 14

Here I use SY to adjust Y0, where previously I used SP. SY refers to "speed of Y (up/down) movement." In single-line resolution, remember, there are twice as many vertical locations. So to get approximately the same vertical speed, we have to move the player 2 bytes instead of just 1. Consequently, SY is initialized to 2 in line 10070 and SP is still initialized to 1. Horizontal movement is no different in single- versus double-line resolution.

A similar change could also be made for the missile coordinate adjustment routine at lines 60 through 75. When you get this program running, notice how the missiles don't go exactly as you might expect when you fire them in a diagonal direction. That's because in single-line resolution there are about 224 vertical positions, but only 150 horizontal ones.

LINE 10070

In line 10070 I initialized SY to 2 as I just mentioned.

LINE 315

To line 315 I added:

MY0S=MY0

MY0S is short for MY0SAVE. Here I am saving the vertical coordinate for the missile. Notice that I am doing this when the missile's vertical coordinate is first set. I use this saved value later in the error correction routine (which

begins at line 3000) to help me erase the missile from the screen.

LINES 3000 THROUGH 3080

The error correction routine at lines 3000 through 3080 needed quite an overhaul because of the differences in vertical coordinates in single-line resolution. Also the manner in which I erase a missile is different. I now need to reference specific bytes in MISSILES$ when erasing the missile. I can no longer merely code MISSILES$=BUFFER$ to set the missile memory area to zeros. Again, this is necessary because of an apparent bug in Atari BASIC.

That does it. The complete listing for firing missiles in single-line resolution follows. To make the required changes easy to see, I have circled them. Notice that in line 2, I made a comment to myself that I saved the program on disk 30. I suggest you revise this REM statement so it reminds you where **you** saved the program. In the next chapter, we'll put our player in a maze and add collision detection!

SLRES.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

10

Detecting Collisions

Suppose a player fires a missile at an alien attacking space craft.
Wouldn't it be nice to have a simple way to tell when there is a collision between the missile and alien ship? Or suppose you're making a maze game. Would you like to be able to detect when a player touches a maze wall?

COLLISION DETECTION REGISTERS

You can. Collision detection is easy with PMG! The Atari engineers wisely provided several collision detection registers for this purpose. Here they are:

Missile Collisions

Memory Location	Shows:
53248	Missile 0 to playfield collision
53249	Missile 1 to playfield collision
53250	Missile 2 to playfield collision
53251	Missile 3 to playfield collision
53256	Missile 0 to player collision
53257	Missile 1 to player collision
53258	Missile 2 to player collision
53259	Missile 3 to player collision

Player Collisions

Memory Location	Shows:
53260	Player 0 to player collisions
53261	Player 1 to player collisions
53262	Player 2 to player collisions
53263	Player 3 to player collisions
53252	Player 0 to playfield
53253	Player 1 to playfield
53254	Player 2 to playfield
53255	Player 3 to playfield

ASSIGNING VARIABLE NAMES

To simplify your programming task, I recommend that you assign a variable name to each collision register you need to use. It's a lot easier to keep track of variable names than all those memory locations.* Atari recommends

variable names such as these:

M0PF (missile 0 to playfield collision) M1PF (missile 1 to playfield collision)

In the demonstration program in this chapter we will be using two collision registers: M0PF and P0PF.

- What register do you think P0PF refers to?

ANSWER

1. 384 **2.** 128 **3.** a

Player 0 to playfield collision (53252) If you have the program from the previous chapter loaded into memory, you might want to initialize P0PF and M0PF right now by adding these statements to line 10070: P0PF=53252: M0PF=53248

*Later, though, you may wish to go back to using constants (such as 53260, 53261, etc.). That's because Atari BASIC only allows you 128 variable names. Also, statements with constants actually execute slightly faster than those using a corresponding variable! (This is not true with other computers such as the PET and Apple, where variables execute 10 to 40 times faster than constants.) I'd like to thank B. B. Garrett for clarifying this in his informative article "Atari Times" in the May, 1983 issue of Compute!

READING COLLISION REGISTERS

You can read a collision register with a peek command. For example:

COLLISION=PEEK(P0PF)

After this statement executes, COLLISION will contain either 0,1,2, or 4. In our sample program you will find that:

- If P0PF contains a zero, then there was no collision.
- If P0PF contains a 1, then PLAYER 0 collided with that part of the playfield drawn with COLOR 1.
- If P0PF contains a 2, then PLAYER 1 collided with that part of the playfield drawn with COLOR 2.
- If P0PF contains a 4, then PLAYER 1 collided with that part of the playfield drawn with COLOR 3.

Ok, try this one. Suppose you draw a line on the screen with these statements:

GRAPHICS 7:SETCOLOR 2,3,4:COLOR 3:PLOT 0,0:DRAWTO 159,0

Furthermore, suppose you read a collision register like this:

COLLISION=PEEK(P0PF)

- What value will be in COLLISION if player 0 is touching the line you drew?

ANSWER

4 (a 4 shows a collision with a playfield drawn with COLOR 3).

MULTIPLE COLLISIONS

Once a collision occurs, the collision register retains the number that was placed in it. If a second collision occurs, the next number is **added** to the number that already exists there.

- Suppose player 0 collides with a playfield drawn with COLOR 1 and then collides with a playfield drawn with COLOR 3. What value will be the collision register? (Careful now, this one is a bit tricky.)

ANSWER

5 (the 1 from the first collision will be added to the 4 from the second collision).

CLEARING COLLISION REGISTERS

Since the collision registers retain the values put in them when a collision occurs, it's important to reset them. This is almost as easy as taking candy from a baby. All you do is poke a 1 into the HIT CLEAR register at location 53278.

I suggest you use a variable for the HIT CLEAR register. Let's call it HITCLR. At line 10070 insert this statement:

HITCLR=53278

Often we think of clearing something by poking zeros in it. But this is different; here we are **turning** on the hit clear switch. That's why we poke it with 1 rather than 0.

- When we poke a 1 into HITCLR what do you think happens?

  a. All collision registers are cleared.

  b. Only selected registers are cleared.

ANSWER

You're right if you said "a. All collision registers are cleared."

USING COLLISION REGISTERS

Often programmers peek at the collision registers and then use a series of IF-statements to decide on what action to take. But there's a much better way. Remember, in Atari BASIC you can GOSUB a variable or even GOSUB a PEEKED value! We did this in our joystick routine, and we can also do it with collision registers.

- Suppose we want to execute a subroutine that starts at line 41 if player 0 collides with a COLOR 1 playfield.* Write a statement to make that happen. (GOSUB the value in P0PF plus an offset.)

ANSWER

GOSUB PEEK(P0PF)+40

- Now suppose player 0 hits a COLOR 3 playfield. At what line number must we have a subroutine to handle this possibility?

ANSWER

We need a subroutine at line 44. That's because P0PF will contain a 4 if player 0 hits a COLOR 3 playfield.

*In this book, a COLOR 1 playfield is simply a playfield drawn with COLOR 1. The term "playfield 1," as used in the Atari technical manual, is not synonymous with the term "COLOR 1 playfield."

DRAWING A PLAYFIELD

Now let's use some of these techniques in a PMG program. First, let's draw a playfield. We'll make it a maze so that in a later chapter we can expand the program into a full-fledged game.

We've already reserved lines 12000-13000 for drawing a playfield, so let's put our new playfield subroutine there. (Note that we need to delete the previous playfield, which was contained in lines 12000, 12010, 12020, and 12030.)

Here are the lines for the new playfield subroutine. I suggest you enter them now.

```
11999 REM DRAW PLAYFIELD
12000 SETCOLOR 2,3,4:COLOR 1
12010 PLOT 0,0:DRAWTO 159,0:DRAWTO 159,79:DRAWTO 0,79:DRAWTO 0,0
12015 COLOR 2
12020 PLOT 80,61:DRAWTO 80,79:PLOT 0,60:DRAWTO 30,60:DRAWTO 30,79
12025 COLOR 3
12030 PLOT 52,6O:DRAWTO 52,45:DRAWTO 110,45:DRAWTO 110,60:DRAWTO
      137,60:DRAWTO 137,45:DRAWTO 110,45:PLOT 159,32
12040 DRAWTO 110,32:PLOT 80,45:DRAWTO 80,20:PLOT 130,16:DRAWTO 130,14:
      DRAWTO 127,14:DRAWTO 127,16:DRAWTO 130,16
12050 PLOT 43,0:DRAWTO 43,30:PLOT 52,45:DRAWTO 22,45:DRAWTO 22,13:PLOT
      103,0:DRAWTO 103,17:POKE 559,34:RETURN
```

Also, let's revise lines 2010 and 2015 so that line 2010 becomes 2015 and 2015 becomes 2010. It seems that PMG works better when the playfield is drawn **before** the PMG setup is executed.

And at line 2005 change GRAPHICS 5 to GRAPHICS 7 since our new playfield requires that graphics mode.

REVISING THE MAIN LOOP

Now let's revise the main loop of our program so that it detects when our player touches the sides of the walls. First, change line 200 into line 201. Do this so that we can create some new collision detection routines at line 200.

Now at the beginning of line 200, let's simply print the contents of the collision register--just so we can see what they contain when various objects collide. This is easy to do, like so:

? "P0PF=";PEEK (P0PF),"M0PF=";PEEK(M0PF)

Next, also at line 200, let's call for the execution of the two collision detection subroutines, one for P0PF and one for M0PF. And let's arrange things so that if there is no collision, we immediately return from each subroutine. We'll let the P0PF subroutine start at line 40 and the M0PF subroutine start at line 50.

Here's the call to the M0PF subroutine:

GOSUB PEEK(M0PF)+40:

- Now you write the call to the P0PF subroutine:

ANSWER

GOSUB PEEK(P0PF)+50

Altogether then, line 200 will look like this:

200? "P0PF=";PEEK(P0PF),"M0PF=";PEEK(M0PF):GOSUB PEEK(M0PF)+40:GOSUB PEEK(P0PF)+50

PLAYER-PLAYFIELD COLLISIONS

Simple right? Now all we need to do is decide what we want to happen when a collision occurs. For now, let's fix things so that our player cannot walk through the maze walls. Here's how we'll do it. At the beginning of line 201 we'll save the X0 and Y0 coordinates by inserting the statement X0A=X0 and Y0A=Y0:

201 Y0A=Y0:X0A=X0:GOSUB PEEK(JOYSTICK):GOSUB MOVELEGS

Then when our player hits a maze wall, we'll reset X0 and Y0 back to what they were **before** the collision. Got it?

Suppose our player hits a COLOR 1 playfield. Then P0PF will contain a "1," and control will pass to line 41 (as a result of GOSUB PEEK(P0PF)+40). Then at line 41 all we need to do is:

1. Set the Y0 and X0 coordinates to what they were before the collision.

2. Clear the collision registers by poking a 1 into HITCLR.

- See if you can write the code for line 41:

ANSWER

41 X0=X0A:Y0=Y0A:POKE HITCLR,1:RETURN

We also need this same subroutine at line 42. That's because P0PF will contain a 2 if our player hits a COLOR 2 playfield.

If our player hits a COLOR 3 playfield, P0PF will contain a 4 (strange as this may seem).

- So what line will be executed if our player hits a COLOR 3 playfield?

ANSWER

44 (Since 40+4=44.)

- So what code do we need at line 44?

ANSWER

44 X0=X0A:Y0=Y0A:POKE HITCLR,1:RETURN

Let's look at the playfield detection statement more closely. It is: GOSUB PEEK(M0PF+40).

- If the player does **not** hit anything. P0PF will contain a zero. So where will control pass when the playfield detection statement is executed?

ANSWER

Control will pass to line 40 (0+40=40).

- What code would be appropriate for line 40? (Hint: we don't really need to do anything since no collision has occurred. All we need to do is get back to the main loop.)

ANSWER

All we need is a RETURN statement at line 40. This will return control to the main loop.

MISSILE COLLISIONS

The missile collision subroutines start at line 50 since the calling routine is GOSUB M0PF+50. If no missile-playfield collision has occurred, we won't need any specific action.

- What will the code be for line 50?

ANSWER

50 RETURN

If a missile hits a wall, we will want to do these things:

- Move the missile off the screen by poking zero into HPOSM0.
- Turn off the missile sound.
- Turn off the missile move indicator (FIRE0).
- Clear the collision registers.

- Since we will have to do this whenever a missile hits a wall, let's put it into a subroutine at line 190. See if you can write the code:

ANSWER

190 POKE HPOSM0,0:D1=0:FIRE0=0:POKE HITCLR,1:RETURN

- Now if a missile hits playfield 1, to which line will control pass?

ANSWER

51 (M0PF will contain 1.1+50=51)

- What additional line numbers will we need for playfields 2 and 3?

ANSWER

We'll need line 52 and line 54. (Remember, if a missile hits playfield 3 then the collision register will contain a 4.)

Line 190 contains the commands that we want executed if a missile hits a wall so at line 52 all we need is:

52 GOSUB 190:RETURN

Similarly, at lines 53 and 54 we'll use the same code:

53 GOSUB 190:RETURN
54 GOSUB 190:RETURN

TRY IT

Make all the changes I've discussed so far to MISSILE.SAV. Also, change line 2 to:

SAVE"D:MISSCOL.SAV":STOP

All the changes are summarized in the listing that follows:

MISSCOL.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Save the program and then run it. Note: I suggest you push the RESET button each time before you run the program; this will ensure that the PMG image appears correctly when the program first starts. As you move the player against the various walls of the maze, notice the values that appear in P0PF.

Next try firing the missile in various directions. Notice that sometimes the missile will hit a wall. When this happens, the missile sound stops and the missile disappears. The player can now immediately fire another missile.

Sometimes the missile will go right "through a wall." That's because the missile is moving in increments of 6. Actually, the missile is "hopping over the walls"--it really never touches the wall--hence there is no collision. This could be fixed by making the walls thicker or the missile bigger. Or the missile coordinates could be adjusted by 1 instead of 6. Of course, then the missile would probably move too slowly for most purposes.

Yet another approach would be to use an assembly language routine to move the missile.* In the programs in this book we'll simply let the missile pass through walls. In a game situation, this effect is good because you never know when one of your missiles will be "super-charged" and capable of passing through a wall.

*Watch for my next book on advanced PMG techniques. In it I'll show you how to use machine language subroutines to speed up your PMG programs.

SLOW PLAYER MOVEMENT

Notice that the player moves rather slowly. That's mainly because we are constantly peeking at and printing the values contained in the collision registers. You can speed up the player's movement considerably by deleting ?P0PF=";PEEK(P0PF) and ?M0PF=";PEEK(M0PF) in line 200.

Of course part of the reduction in the player's speed results because the main loop is becoming more complicated. Remember, we are doing quite a bit in this little BASIC program. We have created:

- Vertical, horizontal, and diagonal movement
- Leg animation
- Missile firing and moment in eight directions
- Random missile-size selection
- Explosive missile sound

- Player and missile collision detection.

But with player-missile graphics, once we have come this far, it's easy to add even more "features."

ADDING MORE FEATURES

Let's try something a little different. Let's pretend that the COLOR 1 playfield has an electrical charge that will zap our player if he touches it. To produce the "zapping effect," let's produce a strange sound and flash several colors through the player when he touches playfield 1. Furthermore, let's set our player to a random new color and then put him back to his starting location. The code to do this is relatively simple and won't slow down the main action. That's because the code will not be in the main loop but in subroutines that will execute only when a collision occurs.

Here's the new code to "zap" the player when he touches the COLOR 1 playfield. Change line 41 to:

```
41 X0=175:Y0=80:GOSUB 400:POKE 704,PEEK(RANDOM):
PLAYER0$=BUFFER$:POKE HPOSP0,X0:POKE HITCLR,1:RETURN
```

And add lines 400 and 410:

```
400 FOR I=0 to 100 STEP 2:POKE 704,I:SOUND 0,I,10,15:SOUND 1,I+50,10,15:NEXT I
410 SOUND:RETURN
```

Also change line 499 to line 390 so that line 390 reads "GOTO 200."

As you can see the subroutine at line 400 rapidly moves different values into player 0's color register. This causes him to "glow." In the same loop we also include a nice sound effect with the aid of a couple of SOUND statements. (I used SOUND statements, here, rather than poking audio control registers, because SOUND statements are easier to use. Remember, speed is not so important here since we are not in the main loop. When a player is zapped, it's natural for the action to stop. All attention is focused on the zapped player, anyway.)

There you have it. Collision detection complete with a fancy routine to zap a player and move him back to his starting location if he hits a specific kind of playfield.

In the next chapter we'll pull together everything you've learned so far and create "MAZEDUEL," a racing game in which two players compete for a dangerous but valuable "crystal."

Return to Table of Contents | Previous Chapter | Next Chapter

11

Programming a Game

In this chapter I'll show you how to use the techniques you've learned to write an arcade-style game. It doesn't have the speed of a machine language program, but thanks to Atari's PMG it contains a lot of action-packed features: animated players, sound effects, missiles, lasers, collision detection, and flashing colors. And best of all, since you've come this far, you'll be able to modify it to your liking. Before you know it, you'll be creating your own **original games!**

MAZEDUEL

Let's call the game **"Mazeduel"** since two players will be fighting it out as they chase each other around a maze. I'll go over the rules of the game first and then discuss the programming techniques.

Winning a round. The game will have rounds (sort of like a boxing match). In the upper right corner of the maze you'll see a magic crystal. If a player touches the crystal, an alarm will sound as the screen flashes different colors. The player who touches the crystal wins the round and is awarded five points. Both players will then be moved back to the starting box.

A player can also win a round by hitting the crystal with a missile. In this case, however, that player gets only one point.

Firing missiles. Players can fire missiles at each other. Missiles contain a strange substance that causes players to instantly expand to double size if they are hit. If a player expands to double size, he stays that way until the other player gets zapped.

If you are hit by a missile you will be zapped back to the starting box and enlarged to double size. You get one point for hitting another player with a missile.

Maze walls. You must exercise extreme caution when moving around the maze. If you touch a maze wall, you are also zapped back to the starting box and enlarged to double size. In addition, the other player will gain a point.

Beware of the crystal. The crystal is harmless--unless you try to win a round by touching it. When the crystal's defense system is activated, it may send out a storm of laser-type missiles. If you are quick enough though, you may be able to avoid the lasers. But if you don't, you will again be expanded to double size and zapped back to the starting box. Another penalty is that the other player will gain one point. The lasers destroy any part of the maze that they hit, but only stun players. As the maze starts to break up more and more, you will find it easier to sneak up on the crystal.

It's possible for a player to escape from the maze and sneak up on the crystal while "off screen." (I won't tell you how.) Be alert when hiding behind maze walls. Missiles fired by players sometimes penetrate walls! To fire a missile, move the player in the direction you want the missile to go and press the fire button. The first player to get ten or more points wins. You can start a new game by pressing your fire button.

Selecting colors. At the beginning of each game, you can select the color of the playfield by pressing the SELECT key. Just keep pressing it (or hold it down) until you get a color you like. When you're ready to start the game, press the START key.

To select colors for players, press the OPTION key.

As usual, the complete listing appears at the end of this chapter. You may wish to load the program from the previous chapter. MISSCOL.SAV, and then modify it to produce MAZEDUEL.SAV. I've made a lot of modifications, however, and it may be just as easy to type it in from the start.

For maximum execution speed, don't type in any of the REM statements. I included them to help you understand the program (and to help myself keep track of what I was doing).

Again, the subroutine beginning at line 2000 takes care of calling the various setup subroutines. Here are the setup subroutines along with their beginning line numbers:

10000 Initialize Constants
12000 Draw Playfield
11000 Set up PMG
13000 Specify player colors and initial position (off screen)
5000 Allow users to select colors

Note that line 10000 is no longer a REM statement. If it were, the GOSUB 10000 statement at line 2000 would produce an error if you were to delete the REM statements.

In the 13000 subroutine, at line 1345, I poke PLAYER1$ with the data in LEG1$. Remember, PLAYER1$ is the PM memory area for our second player. PLAYER0$ is the memory area for our first player. To simplify the programming, I made both players have exactly the same image. (They are easy to tell apart, because each has its own color.)

Notice that in the subroutine starting at line 5000. I move the players onto the screen and display a message in the text window. Location 53279 tells me which of the console keys have been pressed. When 53279 contains a 6, I know the user has pressed the **START** key and it's time to return from this subroutine.

Control then returns to the "executive" setup subroutine at line 2000. I call the subroutine at line 2000 an "executive" because it controls other subroutines rather than carrying out any direct action of its own. After the last subroutine within the executive setup subroutine, control returns to line 1. From there we jump to the main loop at line 200.

CHECKING FOR COLLISIONS

Lines 200 and 201 now check for the various collisions. Notice how easy this is. We simply call various subroutines. The subroutine called depends on the contents of the appropriate collision register. For example, look at the first statement in line 200:

GOSUB PEEK(P0PF)+100

- If the player 0 to playfield collision register (P0PF) contains a zero, meaning no collision, which subroutine will be executed?

ANSWER

The one at line 100 (0+100=100).

So at line 100 we put a RETURN statement. Now, the magic crystal was drawn using COLOR 2. So the magic crystal is considered a "COLOR 2 playfield."

- Suppose player 0 touches the crystal. Which subroutine will be executed?

ANSWER

The one at line 102 (2+100=102).

The other collision detection subroutine calls work the same way. Slick, huh? And much faster than IF-statements!*

*I'd like to thank my son. Dan Seyer, for suggesting this collision detection method.

MOVING LEGS

At line 204 we check to see if joystick 0 has been moved. J0 is now a variable initialized to 632, the memory location that contains the value for joystick 0. If J0 is not equal to 15 (upright joystick), then I add 1 to Z. Z will now be equal to 1. At this point, I GOSUB line 35 (34+Z will equal 35). Line 35 moves LEG1$ data into PLAYER0$. The next time through the main loop, line 204 will call the subroutine at line 36 (since Z will now be equal to 2).

This is similar to the earlier leg movement routine. I modified the earlier routine to make it easier to handle the movement of two separate players.

If the joystick is not moved, then control will pass to line 205. At line 205 I simply move the "standing still" image of player 0 into PLAYER0$. I do a similar thing for player 1 at lines 206 and 207.

CRYSTAL DEFENSE SYSTEM

Control now passes to the crystal defense system at line 210. Notice that I am using the random number generator at location 53770. (RANDOM is initialized to 53770.) If RANDOM contains a number greater than 220 then we check to see if either of the players has entered the crystal's attack zone. We know a player is in the crystal's attack zone if its X coordinate is greater than 142 and its Y coordinate is less than 34. During game development, you can easily discover the coordinates for a specific location by positioning a player where you want him, hitting the break key, and then printing the coordinates.

Why the check of the random number? Well, this way the crystal's defense system isn't perfect. Sometimes a player will be able to sneak past it and reach the crystal before it starts wildly firing lasers in all directions. By changing 220 to some other number, you can increase or decrease the probability that a player will be able to sneak in and touch the crystal. The higher the number (up to 255) the better the player's chances will be. (By the way, location 53770 generates a random number between 0 and 255.)

If all conditions are satisfied, then the laser firing routine is activated starting at line 450. This is a fairly simple routine, but the results are quite dramatic. In this routine I use DRAWTO statements to create the lasers. A nice effect is produced by drawing the various **random** vertical coordinates. Again I used location 53770 to set a variable called VT to various values. VT is then used in the DRAWTO statement as the vertical coordinate. Notice that I use COLOR 1 when drawing the laser. The collision detection routine is already set up to detect a collision with anything drawn with COLOR 1. So if the laser hits the player, zapo-whamo! After drawing a laser, I erase it by redrawing it using COLOR 0.

After the crystal defense check at line 210, control passes on to the missile move routine at 300.

MISSILE MOVE ROUTINE

This routine is quite similar to the one in the previous chapter, except that now we have to deal with two missiles. This can present a problem when two missiles are both fired at the same time. The binary image data for missile 0 is:

00000011 (or 3 in decimal)

For missile 1, the binary data is:

00001100 (or 12 in decimal)

Suppose both missiles happen to occupy the same byte in missile memory. (This will happen whenever MY0=MY1.)

- What if we move 00000011 into byte 22 of MISSILE$ and then immediately move 00001100 into that same byte of MISSILE$? What will be the effect on missile 0?

ANSWER

Missile 0 will disappear! That's because the zeros in the far right bit position of missile 1 will take the place of the 1's that were there before.

Look again at the binary data:

Missile 0: 00000011
Missile 1: 00001100

This can be solved with a machine language subroutine that addresses specific bits. That is, if we wanted to turn on missile 0, we would turn on only the two bits on the far right. But we would not move zeros into the other bits.

In BASIC we cannot address specific bits. That is, we cannot move data only into certain bits, but not others. In BASIC we must deal in bytes.

My solution to the problem was to "turn on" both missiles whenever both missiles happened to occupy the same byte in missile memory. I did this by initializing a variable called BMISS$ to 15. Then at line 335 I check to see if MX0=MX1. If MX0=MX1, I jump to line 400 where I move 15 into MISSILE$.

- Why does putting a decimal 15 into MISSILES$ turn on both missiles$?

ANSWER

Because in binary, 00001100 + 00000011 = 00001111 and 00001111 = a decimal 15.

Now if MY0 does **not** equal MY1, the missiles are not destined to occupy the same byte in missile memory. Consequently, I move the missile data for each missile with separate statements in line 340.

Yes, the IF-statements do slow things down. You're right if you're thinking that this is a good place for a machine language subroutine. (Look for it in my next book on PMG)

After the missile data is moved into the proper byte of MISSILE$, control returns to line 200. the first line of the main routine.

TYPING IN THE PROGRAM

The program listing appears at the end of this chapter. It may look long, but remember, a lot of it is a duplication of the code from the previous chapter.

As I mentioned earlier, it's probably a good idea to omit all REM statements when typing in the program. It will require less memory this way and run faster. The program will fit into a 16K cassette system or a disk drive system with 24K.

Since this program was designed for instructional purposes, more line numbers are used than actually needed. You can speed up the program somewhat by combining some lines. For example, line 330 could be combined with 335. But be careful in doing this. For example, don't try to combine two IF-statements. If you do, then the second IF-statement won't execute unless the first condition is true. For best results, I suggest you type in the program exactly as is. Then after you've saved a working copy, you can do your thing! Have fun. Here are some revision suggestions:

1. Make the crystal more interesting by creating him with PLAYER2$. You can put PLAYER2$ right on top of (or underneath) a playfield.

2. You could create a three-colored crystal by combining PLAYER2$ with PLAYER3$ (see the next chapter for details).

3. If you make the crystal with PMG, then you can make it glow by poking different values into the appropriate color registers. You could also easily animate it or change its size, with just a few statements. If you do this at line 450 (the laser firing routine), you won't slow down the main loop. That's because line 450 executes only when a laser is fired by the crystal.

4. Using the other players, you might want to have various alien creatures pop up in the maze and block the movement of the main two players.

5. You may wish to experiment with changing the players to different sizes at different times. Here are the size registers:

Player Number	Size Register
0	53252
1	53253
2	53254
3	53255

To set a player's size, poke the appropriate register with 0,1,2, or 3 as follows:

Player Size	Value to Poke into Register
Normal	0 or 2
Double	1
Quadruple	3

So for normal player size, poke the appropriate register with zero or 2. For double player size, poke it with 1. For quadruple player size, poke it with 3.

6. For faster action, you might want to try your hand at converting part of the program to machine language. MAZEDUEL is written entirely in BASIC. Again, look for my next book on PMG for hints on how to do this! In the meantime, I suggest you keep reading your favorite magazine for ideas. My favorites (in alphabetical order) are: **Analog, Antic, Byte, Compute, Creative Computing,** and **Micro**.

Happy hacking.

Congratulations. You've just about finished this book. By now, you've mastered most of the fundamentals of PMG--one of the most powerful and least understood features of the Atari computer.

In the next chapter, I'll wrap things up (at least for now), by discussing a few additional PMG odds and ends.

MAZEDUEL.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

12

Odds and Ends

Great! You've mastered the fundamentals of PMG and have seen how it can be used to develop a two-player, arcade-style game. In this chapter you'll learn some additional programming kinks.

FIVE PLAYERS!

So far I've said that you can create up to four players and that each player has a missile. Well, as you'll soon see, it's possible to have five players.

There is a trade-off, though; you have to give up all of your missiles. In other words, if you want, you can use the missile memory area as a fifth player. To do that you simply add 16 to the value you would normally poke into the priority register.

Using the Priority Register. Remember the priority register? It is location 623. You set various display priorities by poking either a 1, 2, 4, or 8 into location 623. (To review, see "Setting Display Priorities" in Chapter 5.)

Suppose you want all players to appear in front of all playfields. To set up this display priority, you would normally poke a 1 into location 623. But if you want to turn the missiles into a fifth player, you would poke 17 into 623 (1+16).

To help you keep track of what you're doing you might want to write the code like this:

```
PRIOR=623
POKE PRIOR,1+16
```

Suppose you want your normal player (players 0 through 3) to appear **behind** all playfields. The usual practice would be to poke 4 into location 623. But suppose you want a fifth player. Assuming PRIOR is initialized to 623, how would you code this?

ANSWER

POKE PRIOR,4+16 (or POKE PRIOR,20)

Setting the Fifth Player's Color. To specify the color you want for the fifth player, poke a number from 0 to 254 into **location 711**. (To review, see "Specifying Player Color" in Chapter 4.)

Moving the Fifth Player. Although some PMG programmers may have problems with vertical movement, it will be easy for you! Just use the same method you learned earlier. Here's an example:

```
MISSILES$(Y4)=LEGS1$
```

This statement will cause the image contained in LEGS1$ to appear on the screen at whatever vertical location is specified by Y4. (Remember Y0 goes with player 0. Y1 with player 1, Y2 with player 2, and so on. Remember, too, that the **fifth** player is called **Player 4**.)

Horizontal Movement. Horizontal movement of the fifth player is not so easy. Each missile still keeps its own horizontal position register. So to move the fifth player horizontally (in one piece) you have to poke all four horizontal position registers. Assuming that you have all missiles set to normal width, you might do it like this:

```
POKE HPOSM0,X4
POKE HPOSM1,X4+2
POKE HPOSM2,X4+4
POKE HPOSM3,X4+6
```

All those POKEs add up and slow down the animation loop. So here's another case where a machine language routine might come in handy, but that's the subject of my next book.

Even though these separate horizontal position registers pose a speed problem, you still might use them to good effect. For example, you might "blow up" the fifth player and scatter his pieces all over the screen. Or you might mysteriously create the fifth player by gradually moving his various pieces together.

Collision Detection. Another consideration is collision detection. Each missile still has its own collision detection register. That could be interesting. For example, you might specify that your player is vulnerable only if he is hit in the right arm. (He couldn't be hit by a missile, of course, but he might be hit by a "laser" created with a DRAWTO statement. Or he might be hit by another player.)

MULTICOLORED PLAYERS

So far each player has only one color. But if you want, you can overlap player 0 with player 1 (or player 2 with player 3). In the area of overlap, a third color will be produced. In effect, then, you can have a three-colored, animated object. To make this work, you must first add 32 to the value that you would otherwise poke into the priority register (location 623).

Let's try a problem to clarify that. Suppose you want your players to have priority over playfields. Normally, you'd poke 1 into 623. But let's say you want a fifth player and you also want to combine player 0 and player 1 to create a multicolored spaceship. How would you code the poke into location 623?

ANSWER

POKE 623,1+16+32 (or POKE 623,49)

GRAPHIC SHAPE REGISTERS

Now let's turn to another topic: the graphic shape registers. So far we haven't needed these registers. That's because we allocated a special area in memory to contain the PM shape data. And as you will recall, we told ANTIC where this PM memory area was by poking the address of BUFFER into location 54279. (To review, see "PMBASE" in Chapter 4.)

Also, we set up what is called "direct memory access" by poking location 559 with an appropriate value. In addition, we specified the type of resolution we wanted by poking location 53277.

Well, if you use the graphic shape registers, you can bypass ANTIC, and you don't have to poke anything into locations 54279. 559, or 53277. Consequently, your PMG setup is greatly simplified.

But the graphics shape registers are not so great for animation. That's because each can contain only one byte of data. They are useful, though, when you want to display a player image the entire length of the screen. You might want to do this to highlight textual material or to create a special boundary for a playfield.

Here are the graphics control registers for each of the players:

Player Number	Location
0	53261
1	53262
2	53263
3	53264

The graphics shape register for all of the missiles is at location 53265. Even though each graphics shape register holds only one byte of image data, that byte runs the entire length of the screen. You'll see an example of these registers in action in a moment, but first I'd like to expand on DMACTL, the direct memory access control register at location 559.

DMACTL

You can specify different options using this register. In the chart below, just add up the options you want. Then poke the total into location 559. But note that you must pick only one of the first four options:

Option	Value to Poke into 559	
No playfield	0	
Narrow playfield	1	(pick one)
Standard playfield	2	
Wide playfield	3	
Missiles	4	
Players	8	
Single-line resolution	16	
Double-line resolution	32	

Note: you won't be able to see the edges of the wide playfield unless you scroll off the usual screen.

SAMPLE PROGRAM

As usual I will end this chapter with a sample program. The program illustrates the use of:

PMG in graphics mode 0
The fifth player
Priority selection
Playfield size selection
Overlapping of players to get multicolored objects
A simplified PMG setup (using the graphics shape registers)

If you have written programs in graphics 0, you may want to consider spicing them up with some of the techniques demonstrated in this program.

Happy hacking!

GRAFP.BAS
Download (Saved BASIC)
Download / View (Listed BASIC)

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix

Player Color Location

Player	Location
0	704
1	705
2	706
3	707
4 (missiles)	711

A guide for designating your player's color:

Color	Number
Gray	0
Gold	16
Orange	32
Red	48
Pink	64
Violet	80
Purple	96
Blue	112
Blue	128
Light Blue	144
Turquoise	160
Blue-Green	176
Green	192
Yellow-Green	208
Orange-Green	224
Light Orange	240

These numbers are starting numbers. To any given number you can add an even number from 0 to 14 to change the lightness of the color.

Player Missile Memory

To tell ANTIC where the start of PM memory is we simply poke the proper address into location **54279**, the player-missile base register. Example:

POKE 54279,ADR(BUFFER$)

Location 559, DMACTL

Choose the options you want and add up their values. Poke the **total** into 559.

Option	Value
No playfield	0
Playfield size:	
Narrow	1
Standard	2
Wide	3
Turning on:	
Missiles only	4
Players only	8
Missiles and players	12
Single-line resolution	16
Double-line resolution	0
Turn on DMA	32

Location 53277, The Graphics Control Register (GRACTL)

Use this location along with 559 to "turn on" the PMG system.

If you want:	Then:
Missiles only	Poke 53227,1
Players only	Poke 53277,2
Players and Missiles	Poke 53277,3

Horizontal Position Registers

Location	Player
53248	0
53249	1
53250	2
53251	3

Location	Missile
53252	0
53253	1
53254	2
53255	3

Horizontal Coordinates

Joystick Reading

Peek into these locations to read joystick values.

Location	Joystick Number
632	0
633	1
634	2
635	3

Joystick Values

This diagram shows the values produced when you move the joystick in various directions. Note that if you do not move the joystick, a value of 15 is produced.

Sound Registers

Location	Used to Control
53760	Pitch of voice 0
53761	Distortion and volume of voice 0
53762	Pitch of voice 1
53763	Distortion and volume of voice 1
53764	Pitch of voice 2
53765	Distortion and volume of voice 2
53766	Pitch of voice 3
53767	Distortion and volume of voice 3
53768	Pitch of all voices

Note: before using pokes to create sound, initialize the sound system with:

POKE 53768,0 POKE 53775,3 (Or simply code: SOUND 0,0,0,0)

Location 53768 can be used to control the quality of sound. For example, you can filter out certain frequencies or combine two channels to improve pitch range and accuracy. See the book **De Re Atari** for details. (Available from Atari Program Exchange.)

Missile Collisions

Memory Location	Shows:
53248	Missile 0 to playfield collision
53249	Missile 1 to playfield collision
53250	Missile 2 to playfield collision
53251	Missile 3 to playfield collision
53256	Missile 0 to player collision
53257	Missile 1 to player collision
53258	Missile 2 to player collision
53259	Missile 3 to player collision

Player Collisions

Location	Collision Shown
53260	Player 0 to player

53261	Player 1 to player
53262	Player 2 to player
53263	Player 3 to player
53252	Player 0 to playfield
53253	Player 1 to playfield
53254	Player 2 to playfield
53255	Player 3 to playfield

Location 53278, Hit Clear Register

To clear all collision registers, poke a 1 into HITCLR (location 53278).

Location 623, Priority Register

Display Priority	Value to Poke into 623
All players in **front** of all playfields.	1
Players 2 and 3 **behind** playfields; players 0 and 1 in **front** of playfields.	2
All playfields in **front**.	4
Players in **front** of some playfields, but behind others.	8

Suggestion: Experiment! Move your players over various playfield objects to discover the effect of various priority settings.

Other Options for Location 623

Pick the desired options and add up the values. Poke the **total** into location 623.

Option	Value
Combine missiles to make a 5th player.	16
Overlap players to make a third color.	32

Graphics Shape Registers

Player Number	Location
0	53261
1	53262
2	53263
3	53264
All missiles	53265

Return to Table of Contents | Previous Chapter | Next Chapter

Atari Player-Missile Graphics in BASIC Software Archive

APMBASIC.ATR: All programs from the book (ATR file) Download

Return to Table of Contents