Player-Missile Graphics

Objectives of Module

Players are shapes which can move around on the screen quickly and smoothly without disturbing the rest of the screen display. In this module you will learn how to create your own players and how to animate them. You will also explore how to change the size, shape, and color of your players.

Overview (subtopics)

1. What is Player-Missile Graphics?
   Purpose of Player-Missile Graphics.
   What is a player? A missile?
   Features of players and missiles.

2. Player-Missile RAM.
   What is Player-Missile RAM?
   Designing and moving players.

3. Vertical Motion of Players.

4. Setting Up Player-Missile Graphics.
   How to turn on Player-Missile graphics.
   Changing resolution, width, and color.

5. Summary and Challenges.
   Setting priorities.
   Collision detection.

Prerequisite Understanding Necessary

1. You must know Atari BASIC.

Materials Needed

1. BASIC Cartridge.

2. Advanced Topics Diskette.

What is Player-Missile Graphics?

The Atari computer has a very useful feature built into the hardware called Player-Missile graphics. In this section you will learn what players and missiles are, why they are useful, and some of their special properties.

Many of the interesting things one can do with a computer involve moving objects on the screen. This is especially true of many games that are played on the computer. One of the most straight forward techniques for animating an object on the screen is to draw it, erase it, and then redraw it in a slightly shifted position. If you continue this process, the object appears to move across the screen.

Unfortunately it is rather cumbersome and slow to continually draw and erase graphics on the screen. To avoid this, the Atari computer was designed with a feature for moving objects on the screen more quickly and easily. There are some limitations to the size and number of objects that can be animated, but the limitations are small when compared with the ease with which one can move a player on the screen. Worksheet #1 introduces you to what Player-Missile graphics is really all about.

Player-Missile Worksheet #1

Most people are unaware of the true nature of players and missiles. Let's start by finding out what they really look like. RUN the INTRO.PM program on your Advanced Topics Diskette. Type:

```
RUN "D:INTRO.PM"
```

You will have to wait just a moment and then READY will appear on the screen.

This program will enable you to display and move players and missiles on the screen. For now, don't concern yourself with how the program works. Type in the following POKE statements without line numbers and pressing RETURN after each line. Be sure to type a zero after the "P", not the letter "o".

```
POKE P0,60
POKE P1,90      These are the four players.
POKE P2,140
POKE P3,180

POKE M0,100
POKE M1,120     These are the four missiles.
POKE M2,80
POKE M3,160
```

As you can see, players and missiles are nothing more than strips from top to bottom on the screen. Players are simply four times as wide as missiles. Try moving the players and missiles around a little by POKEing numbers into P0 through P3 and M0 through M3. Try something like this:

```
FOR I=0 TO 200:POKE P0,I:NEXT I
```

Or try this:

```
FOR I=0 TO 200:POKE P0,I:POKE M0,200-I:NEXT I
```

When you finish experimenting, make sure all of your players and missiles are visible on the screen.

Now let's try something different. Type the following:

```
FOR I=768 TO 2047:POKE PM*256+I,0:NEXT I
```

The players and missiles are still there, but now they are "blanked out." Later we'll explore what's happenning here. For now, try the following FOR ... NEXT loop below.

```
FOR I=768 TO 2047:POKE PM*256+I,170:NEXT I
```

Experiment with replacing the 170 with other numbers up to and including 255. Use the CTRL and the arrow keys to move the cursor up to edit the line so that you can make the changes without retyping the line.

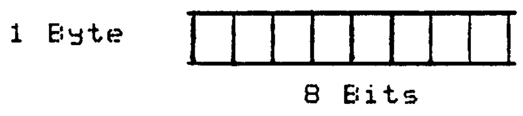
Player-Missile RAM

An important concept that you must understand to use Player-Missile graphics is how players and missiles are stored in computer RAM. In this section the relationship between players and missiles and computer memory are explained.

Before exploring Player-Missile RAM, you must have a complete understanding of bytes and bits, and the binary (base 2) number system.

If your computer has 48K RAM, that means there are 48K, or 48 × 1024, cells of memory. Each memory cell can hold one byte. A byte is simply a number from 0 to 255. If you are interested in exploring how the computer can do all the things you see it do with only the capability of storing numbers from 0 to 255 see the Internal Representation of Text and Graphics module.

<u>Diagram 1</u>



The bytes placed in computer memory are made up of 8 smaller units called "bits" (<u>b</u>inary dig<u>its</u>). Each bit is either a zero or a one -- zero and one are the only digits used in the binary number system.

In base ten there are 10 digits used: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. To represent numbers greater than nine, place values are used. The number 746 in base ten has a 6 in the ones place (6 × 1) and a 4 in the tens place (4 × 10) and a 7 in the hundreds place (7 × 100). The value of the number is based on the digits in each place value. Base two follows the same principles. See Diagram 2. Note that the exponents for the place values in both number systems are the same.

Diagram 2

Base 10

	6	3	0	9
Place Values	10^3	10^2	10^1	10^0
	1000	100	10	1

<--- Digits 0 - 9

Base 2

	1	0	1	1
Place Values	2^3	2^2	2^1	2^0
Decimal Equivalents	8	4	2	1

<--- Digits 0 - 1

All number systems follow a similar scheme. The value of a number is based on the sequence of the digits in the number. A binary number with a one in the two's column and a one in the eight's column (1010_2) is equivalent to ten in base ten. If you add up the decimal equivalents of all the place values which hold a one in a binary number, you will get the decimal equivalent to the number. Try filling in the blanks in the following problem which converts a binary number to its decimal equivalent. To learn more about the binary number system and conversion see the section on Number Systems in the Machine Architecture module.

Binary Number	1	0	0	1	0	1	1	0
Place Value	128	___	___	___	8	4	2	1

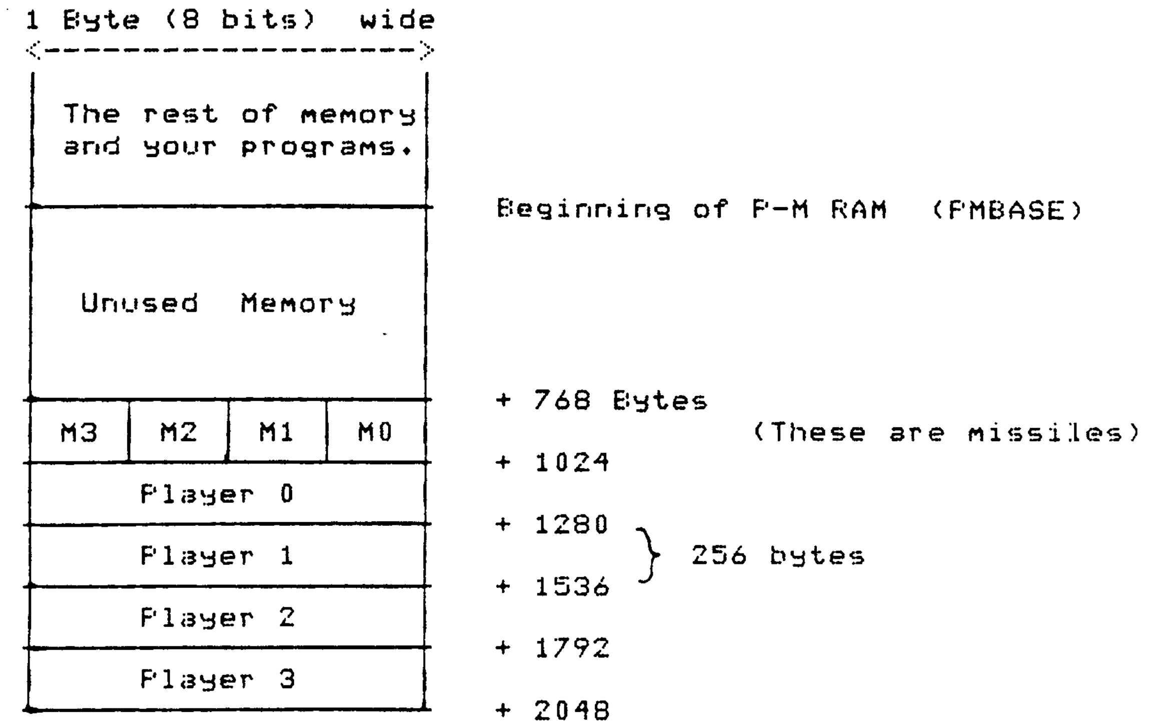
```
Decimal Equivalent = 1 x 128 +
                     0 x ___ +
                     0 x ___ +
                    __ x ___ +
                    __ x  8  +
                    __ x ___ +
                    __ x ___ +
                    __ x ___
                               = 150
```

Like all other memory locations, Player-Missile RAM is just a sequence of bytes used for a special purpose. Each byte is a value from 0 to 255. The reason these memory locations are called Player-Missile RAM is because the shapes and vertical position of each player and each missile are defined by the numbers that are stored in this portion of memory.

Player-Missile RAM is not restricted to a particular area in memory, but rather you can put it almost anywhere. The computer, however, always needs to know where to find the Player-Missile RAM so that it can use the information stored there to display the shapes on the screen.

Diagram 3 shows how Player-Missile RAM is organized for single resolution players. 2048 bytes of RAM (2K) are needed for four players and four missiles.

<u>Diagram 3</u>



Note that each player is 1 byte (8 bits) wide while each missile is only 2 bits wide. This is why missiles are four times narrower than players. Also, note that 256 bytes are reserved for each player. Another 256 bytes are reserved for the four missiles. This area will be utilized for displaying the shapes and moving them on the screen. Worksheet #2 explores how these 256 bytes are used to display a player on the screen.

Player-Missile Worksheet #2

Insert the Advanced Topics Diskette and RUN "D:INTRO.PM". Type in the following lines to create your own player.

```
PM=256*PM
```

PM was initially set the by the INTRO.PM program. This sets the variable PM to the address of the beginning of our Player-Missile RAM.

```
POKE P1,100
```

This moves Player 1 onto the screen. The variable P1 holds the player's horizontal position. The FOR loop below clears out all of Player 1's RAM which begins 1280 bytes after the beginning of P-M RAM. See Diagram 3. Type in the following:

```
FOR I=PM+1280 TO PM+1535:POKE I,0:NEXT I
```

Now let's try POKEing some different numbers into Player 1's RAM.

	Decimal	Binary
POKE PM+1320,24	24	00011000
POKE PM+1321,60	60	00111100
POKE PM+1322,60	60	00111100
POKE PM+1323,153	153	10011001
POKE PM+1324,255	255	11111111
POKE PM+1325,24	24	00011000
POKE PM+1326,24	24	00011000
POKE PM+1327,24	24	00011000
POKE PM+1328,102	102	01100110

```
FOR I=0 TO 200:POKE P1,I:NEXT I
```

Each player has a horizontal position determined by a number POKEd into a special location. The variable P1 contains the location used to determine Player 1's horizontal position. If you PRINT P1, you see Player 1's horizontal position register is location 53249. (See the chart at the back of the module for a summary of the horizontal position registers.) Experiment with POKEing different numbers into this location to see where the player ends up on the screen. Numbers less than about 42 put the player off the left edge, and numbers greater than 200 put the player off the right edge of the screen.

Try the following:

```
5 FOR I=46 TO 200:POKE P1,I:NEXT I:FOR I=200 TO 46 STEP -1:
POKE P1,I:NEXT I:GOTO 5

GOTO 5  (No line number)
```

In the grid below, create your own shape. List the binary numbers which correspond to each line of your shape in the column on the left. Calculate the decimal equivalent to each of the binary numbers and record them in the column labeled decimal.

Decimal	Binary
POKE PM+1320,____	______________
POKE PM+1321,____	______________
POKE PM+1322,____	______________
POKE PM+1323,____	______________
POKE PM+1324,____	______________
POKE PM+1325,____	______________
POKE PM+1326,____	______________
POKE PM+1327,____	______________
POKE PM+1328,____	______________

Now POKE the decimal values which correspond to your shape into Player-Missile RAM.

Move your player back and forth across the screen.

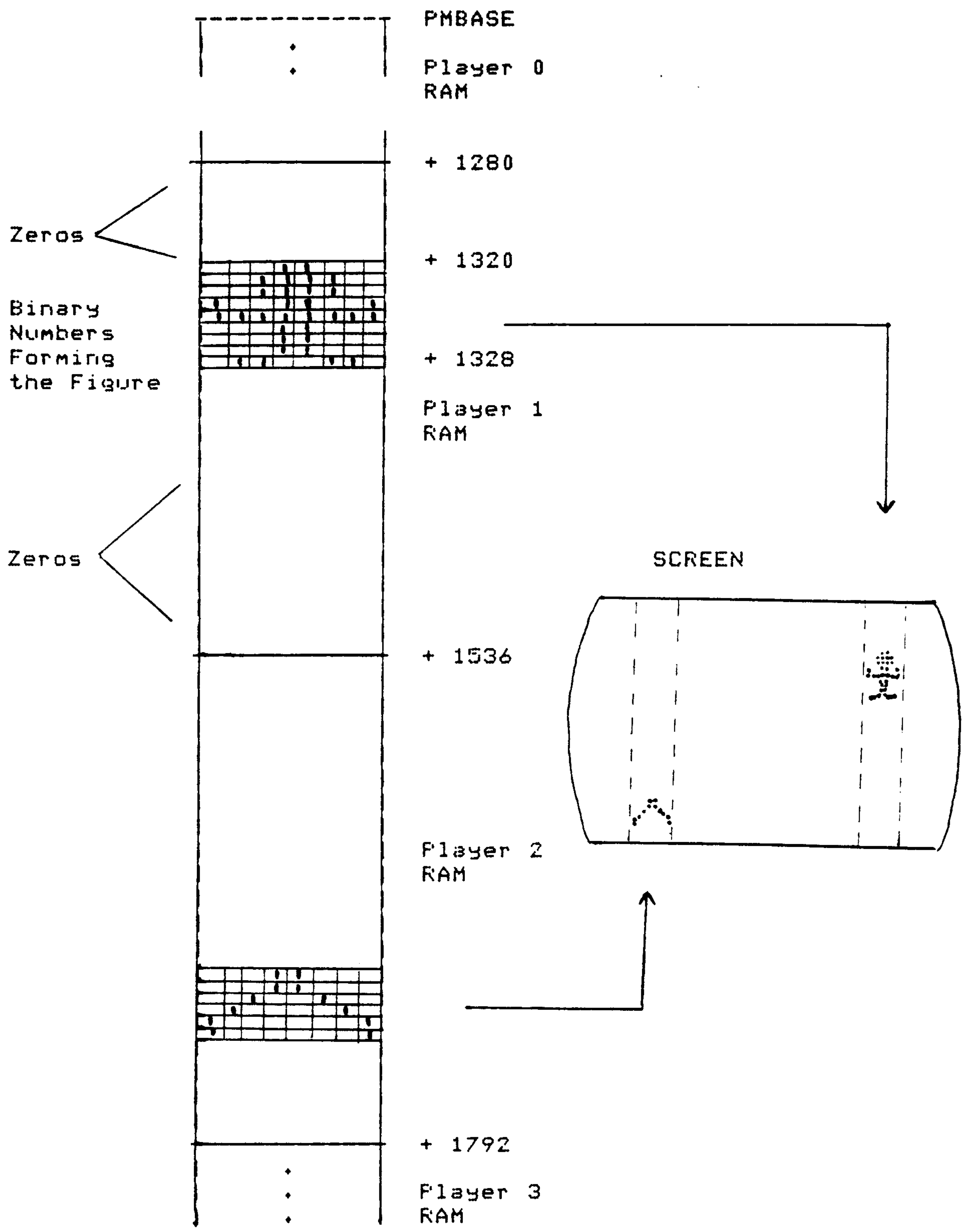
Vertical Motion of Players

Moving a player vertically up and down on the screen is relatively difficult and can be quite slow. Assembly language is needed to obtain smooth motion up and down, but in this section we'll explore how to accomplish reasonable vertical motion using only BASIC.

In worksheet #2 you discovered that you can display an object on the screen by putting zeros into most of a player's RAM (blanking it out) and then turning on certain bits by POKEing numbers into locations in the player's RAM. Diagram 4 below illustrates how bits which are ones in player-missile RAM are displayed on the screen.

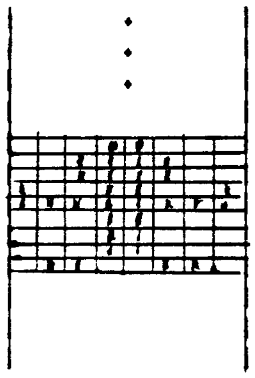
There is a correspondence between the 256 bytes of RAM allocated for the player and the screen display. If the bits that are turned on (the ones) to represent the player are near the beginning of the player's RAM, then the figure will show up near the top of the screen. Similarly, if the bits that are turned on to display the player are placed closer to the end of the player's RAM then the figure will be lower on the screen. To move the figure up or down on the screen, one simply has to move the bytes representing the player up or down in the player's RAM. Worksheet #3 shows you how to do this.

Diagram 4


PMBASE
Player 0 RAM
+ 1280
Zeros
+ 1320
Binary Numbers Forming the Figure
+ 1328
Player 1 RAM
Zeros
SCREEN
+ 1536
Player 2 RAM
+ 1792
Player 3 RAM

Player-Missile Worksheet #3

In worksheet #2 you created a little man on the screen and moved him horizontally. In this worksheet you'll bring the man back on the screen and learn how to move him vertically. To get the man (Player 1) back on the screen, insert the Advanced Topics Diskette and type: RUN "D:MAN.PM". It will be a moment before the little man appears on the screen. If you get funny vertical lines on the screen, press SYSTEM RESET and RUN the program again.



The man is stored in memory (Player 1 RAM) beginning at location PM+1320 and ending at location PM+1328.

Let's write a subroutine which copies all of the bytes down one notch in memory. The variable START will hold the location in memory that the man starts at. The address of the first byte of the man is location PM+1320. Thus, our man is stored in the bytes from START to START+8. To move him down, it is necesssary to copy these bytes to locations START+1 to START+9. The following subroutine does this. Read through the routine carefully and then type it in.

```
500 REM MOVE DOWN
510 FOR BYTE=START+8 TO START STEP -1
520 POKE BYTE+1,PEEK(BYTE)
530 NEXT BYTE
540 REM Erase the starting byte so as not to leave a trail.
550 POKE START,0
560 REM Change START to the new starting byte.
570 START=START+1
580 RETURN
```

Now all that needs to be done is for START to be set to the correct beginning value and then we can use the subroutine. Try the following:

```
START=PM+1320
FOR I=1 TO 100:GOSUB 500:NEXT I
```

Before continuing, write a subroutine starting at line number 600 that moves the man up one byte. It will resemble the subroutine that moves him down, but it will copy bytes starting at the top of the player rather than the bottom. Be careful -- there are some other minor differences!

See if you can write the necessary BASIC to enable you to move the man around the screen with a joystick. Remember that the variable P1 (53249) is the location to POKE to change the man's horizontal position. Unfortunately, you can not PEEK in this location so you have to be a little tricky to keep track of the man's horizontal position (Hint: use another variable just for this purpose). You can see an example of this by running a program called MOVEMAN.PM on the Advanced Topics Diskette.

Setting Up Player-Missile Graphics

In this section you will learn how to turn on the Player-Missile Graphics feature of the Atari. You will also learn how to create different resolution players and how to change their width and color.

You have seen that Player-Missile graphics requires that you set aside 2K of RAM which is called Player-Missile RAM (see Diagram 3). It is common to reserve this RAM at the end of memory by making the computer think that it has less memory than it really does. That enables you to reserve the extra memory for your own use.

Memory location 106 tells the computer how many "pages" of RAM it has. One page of memory is 256 bytes. Four pages of memory equals 1K or 1,024 bytes. Thus, if you subtract 8 pages from the total indicated in location 106, you will reserve 2K of RAM for Player-Missile graphics. The following subtracts 8 pages from the total number of pages stored in 106 in order to reserve the top of memory for Player-Missile RAM. Type it in.

```
PM=PEEK(106)-8:POKE 106,PM:GRAPHICS 0
```

Note: The GRAPHICS 0 statement is necessary so that the computer relocates some things that it had already placed in the 8 pages that you are stealing -- in particular, screen RAM and the display list.

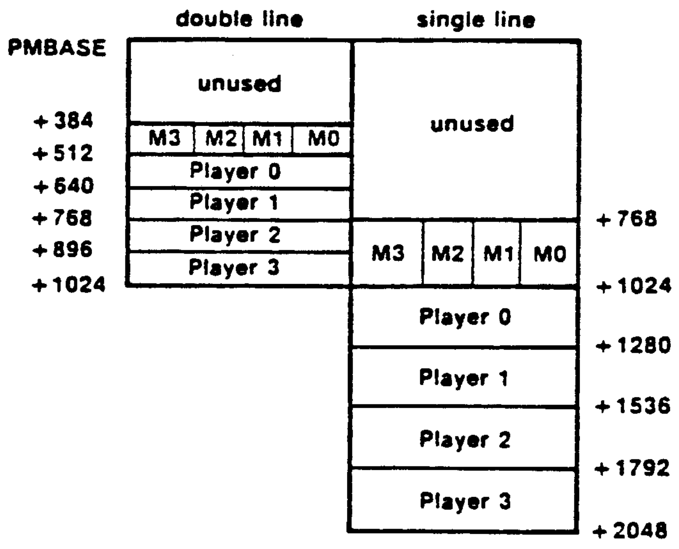
To learn more about how memory is organized and how to read memory maps see the section on MEMORY in the Machine Architecture module.

Continue now by completing worksheet #4.

Player-Missile Worksheet #4

Players and missiles can be either single or double resolution. Double resolution players use up half as much memory (1K) as single resolution players, but their design is less precise as you will see below. Diagram 5 shows you the layout of Player-Missile RAM for both resolutions.

Diagram 5



Type in the following program in order to see the difference between single and double resolution players. Read the comments accompanying each line carefully.

```
10 PMBASE=54279
```

PMBASE stands for Player-Missile RAM base address. This is the computer location where the beginning address of P-M RAM must be stored.

```
20 PM=PEEK(106)-8:POKE 106,PM:GRAPHICS 0
30 POKE PMBASE,PM
```

POKE in the address -- actually the page number -- of where Player-Missile RAM will begin by reseting the total number of memory pages.

```
40 PM=256*PM
```

Store the actual address, rather than the number of pages, in PM for later use.

```
50 FOR I=0 TO 2047:POKE PM+I,0:NEXT I
```

Clear out all of P-M RAM by storing zeros in every location.

```
60 P1=53249:POKE P1,150
```

This is the horizontal position register of Player 1. POKEing a number in this location changes the horizontal position of the player.

```
70 POKE 53277,3
```

This turns on Player-Missile graphics. See also lines 180 and 190.

```
80 FOR I=0 TO 8
90 READ BYTE:POKE PM+1400+I,BYTE
100 NEXT I
110 DATA 24,60,60,153,255,24,24,24,102
```

This puts player data into Player 1's single resolution RAM. The following puts the same data into the RAM area for double resolution players. See Diagram 5.

```
120 RESTORE
130 FOR I=0 TO 8
140 READ BYTE:POKE PM+700+I,BYTE
150 NEXT I
```

Finally, let's ask which resolution player the user wants to see.

```
160 PRINT "Resolution 1 or 2";
170 INPUT RES
180 IF RES=1 THEN POKE 559,62
```

This POKE turns on single resolution P-M graphics.

```
190 IF RES=2 THEN POKE 559,46
```

This POKE turns on double resolution P-M graphics.

```
200 GOTO 160
```

Now try running the program. Wait just a few moments for the user prompt. Try writing some code that moves the player around the screen in both resolutions. It's easy to move it horizontally in either resolution -- just POKE a new number into P1. But moving it vertically in both resolutions simultaneously requires that you shift bytes in two places in memory -- single resolution player RAM and double resolution player RAM.

Let's experiment with some other properties of players. First type: RUN "D:PLAYER.PM". Then try the following POKES.

```
POKE 559,46          (Changes to double resolution.)
POKE 559,62          (Changes back to single resolution.)
WIDTH=53257          (Location determining Player 1's width.)
POKE WIDTH,1         (Double size player.)
POKE WIDTH,3         (Quadruple size player.)
POKE WIDTH,0         (Back to normal size.)

POKE 559,46
POKE WIDTH,1

COL=705              (Location of Player 1's color.)
POKE COL,14
```

Try POKEing other numbers between 0 and 255 into COL. Play with changing the resolution, width, color, and position of your player. Lists of the horizontal position, width, and color registers are presented in the summary of this module.

Note: Much of the work involved in designing and moving players can be avoided by using one of the many software utility programs available for creating players. Experiment with some of these Player-Missile Graphics development tools. For example, try PLAYER MAKER by Wayne Harvey. You will find copies in the camp library.

Summary and Challenges

The Atari computer has four color registers, in addition to the background color, that are used for graphics in the different graphics modes. These color registers are sometimes referred to as "playfields" and are labeled Playfield 0 (PF0), PF1, PF2, and PF3. In GRAPHICS 0 and 8, most of the screen (all of the blue) comes from PF2.

When you use players, sometimes you want them to move in front of or on top of other things you have drawn on the screen, and sometimes you want them to move behind the playfields. You do have some control over these priorities.

If you want all of the players to always move in front of the rest of the screen display, then POKE location 623 with a 1. If you always want the players beneath other things drawn on the screen then POKE 623,4. Look over the chart below. Note that if you put the players beneath the playfields, then in GRAPHICS 0 and 8 you wouldn't even see them because the entire screen is Playfield 2.

<u>Priorities</u>

P0-P3, PF0-PF3, Background	POKE 623,1
P0-P1, PF0-PF3, P2-P3, Background	POKE 623,2
PF0-PF3, P0-P3, Background	POKE 623,4
PF0-PF1, P0-P3, PF2-PF3, Background	POKE 623,8

You also have the capability of detecting if a player or missile collided with some other player or with a playfield, meaning something drawn on the screen with some color register. The chart below shows the locations to PEEK in. If the value read is a 1, 2, 4, or 8, then the collision occurred with player or playfield 0, 1, 2, or 3. See the examples following the chart below.

53248	Missile 0 to playfield collision
53249	Missile 1 to playfield collision
53250	Missile 2 to playfield collision
53251	Missile 3 to playfield collision
53252	Player 0 to playfield collision
53253	Player 1 to playfield collision
53254	Player 2 to playfield collision
53255	Player 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
53260	Player 0 to player collision
53261	Player 1 to player collision
53262	Player 2 to player collision
53263	Player 3 to player collision

Examples:

```
IF PEEK(53253)=4 THEN  Player 1 collided with Playfield 2.
IF PEEK(53258)=1 THEN  Missile 2 collided with Player 0.
IF PEEK(53262)=8 THEN Player 2 collided with Player 3.
```

Note: After a collision occurs, you can only clear the register by doing a: POKE 53278,1.

To detect a collision, first determine what two things which may have collided that you want to check for. Use the PEEK command in an IF .. THEN statement to check for a collision in a specific collision register. For example, if you want to check for a collision between Player 1 and Playfield 2, you PEEK in location 53253. The number stored in the collision register will be a 1, 2, 4, or 8. If it is a 0, then you collided with yourself, which is meaningless. If it is a 1, then you collided with Playfield 0, Player 0, or Missile 0, depending on the register you are using. If it is a 2, then a collision has occurred with Playfield 1, Player 1, or Missile 1. If you find a 4, then a collision has occurred with Playfield 2, Player 2, or Missile 2. If you encounter an 8 in the collision register, then the collision has occurred with Playfield 3, Player 3, or Missile 3. Once a collision has been detected be sure to clear register 53278 (POKE 53278,1) in order to detect subsequent collisions. To read more about collision detection see the section on Animating With Player-Missile Graphics in Compute's First Book of Atari Graphics. Tricky Tutorials also has an example of player-missile animation and collision detection.

To set the horizontal position of players or missiles, POKE the appropriate register with a number from 0 to 255. Only numbers from about 44 to 200 will put players or missiles on the visible screen. Thus, by POKEing a zero into the horizontal position register, you can move players or missiles off the screen. That's the easiest way to "clear" them from the screen.

53248	Horizontal position of Player 0
53249	Horizontal position of Player 1
53250	Horizontal position of Player 2
53251	Horizontal position of Player 3
53252	Horizontal position of Missile 0
53253	Horizontal position of Missile 1
53254	Horizontal position of Missile 2
53255	Horizontal position of Missile 3

The following locations can be used to set the width of players. POKEing a zero into these registers gives normal width, a 1 is double width, and a 3 is quadruple width.

53256	Width of Player 0
53257	Width of Player 1
53258	Width of Player 2
53259	Width of Player 3

The following locations are used to change colors.

704	Color of Player/Missile 0
705	Color of Player/Missile 1
706	Color of Player/Missile 2
707	Color of Player/Missile 3

Finally, the following code is used to setup Player-Missile Graphics.

```
100 PM=PEEK(106)-8:POKE 106,PM:GRAPHICS 0
110 POKE 54279,PM:PM=256*PM
120 POKE 53277,3:POKE 559,62
```

Lines 100 and 110 setup the starting location for the Player-Missile RAM at the top of memory. Line 120 turns on Player-Missile graphics and turns on single resolution graphics.

Note: On line 120: POKE 559,46 for double resolution.

Challenges:

1. Draw your own player on the screen.

2. Change your player's color and width.

3. Move your player in all directions, including diagonally, with a joystick.

4. Detect a collision with a playfield. Make a sound when a collision has occurred. Be sure to POKE 53278,1 in order to detect the next collision.

5. Design and move two players.