ADVANCED PROGRAMMING TECHNIQUES FOR YOUR ATARI®

INCLUDING GRAPHICS & VOICE PROGRAMS

BY LINDA M. SCHREIBER

ADVANCED PROGRAMMING TECHNIQUES FOR YOUR ATARI®

INCLUDING GRAPHICS & VOICE PROGRAMS

Also by Linda Schreiber from TAB BOOKS:

No. 1485 *ATARI Programming . . . with 55 programs*

ADVANCED PROGRAMMING TECHNIQUES FOR YOUR ATARI®

INCLUDING GRAPHICS & VOICE PROGRAMS

BY LINDA M. SCHREIBER

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

FIRST PRINTING

Library of Congress Cataloging in Publication Data

Schreiber, Linda M.
Advanced programming techniques for your ATARI including graphics and voice programs.

Includes index.
1. Atari computer—Programming. 2. Computer graphics.
3. Speech synthesis. I. Title.
QA76.8.A82S367 1983 001.64'2 82-19340
ISBN 0-8306-0145-7
ISBN 0-8306-1545-8 (pbk.)

To my family for their continuing support.

Contents

Program Listings

Introduction

This book was written for the person who wants to get the most out of his or her ATARI computer. If you have a good understanding of BASIC but want to know how to get more special effects, more sounds, and more graphics from your computer, this book is for you. Every program is explained in detail so that after you enter the program and understand how it works, you can use this knowledge in writing your own programs.

All the programs will run on any ATARI personal computer. The ATARI BASIC cartridge CXL4002 and ATARI DOS (for disk programs) were used in creating them.

Memory locations that are used by the Operating System are presented. Explanations are given on how to change their values for different programming effects. The chapters on the disk explain the file structure to give you control over the drive. Everything is included here to make you an Advanced Programmer!

There is a chapter on creating your own character sets, mixing graphic modes, using the player/missile graphics, and flipping screens. You will even learn how to enter a machine-language subroutine to play music while a BASIC program is running!

Note: Because many of the listings in this book use graphics characters and/or reverse video, the following codes have been used in the listings:

- ⟩clear⟩ Press the escape key and the shift/clear key. This clears the screen.
- ⟩characters or letters⟩ Press the control key and the letter indicated between the brackets. *All* characters or letters between the brackets ⟩⟩ are graphic characters.
- Underlined characters or letters are in reverse video.

Chapter 1
Working with Numbers

Ever since man had the need to know how many items he had in his possession, how much grain he needed, or how many days since the last rain, he has had to devise counting systems. It is believed that some ancient tribes used the base two, or three for counting. There is some evidence that base twenty was used by a few early tribes, since their handiest counting device was their fingers and toes.

With numbers came the need to do simple calculations. Soon the problems were no longer simple, and man quickly learned that if he marked the numbers in the dirt or on a tablet he could compute much faster. Stones were probably used much the way we use poker chips today with each type of stone representing a different group of numbers—ones, fives, tens, etc. The figures themselves evolved from crude lines and shapes to the forms we are familiar with today.

The abacus is the oldest, and yet the simplest, adding machine invented. The principle of moving the beads on rods has survived the test of time. Many people consider the abacus to be the first type of computer.

BINARY SYSTEM

As with the abacus, the computer uses its own number system—binary. If you think of a light bulb, a candle, a lock, or a trap, each item has only two states. It can be either on or off, open or closed, set or sprung. The computer operates in the same manner. Each memory location in the computer can be either on or off.

The memory in your computer can hold a charge. This is represented by the number one. When a location has no charge, it is represented by a zero. The computer, then, uses binary or base 2 as its number system.

In our decimal system, each number position is a multiple of ten. The position to the left of the decimal is the unit position. In the binary system, each position is a multiple of 2 with the position to the left of the decimal the unit position. In the decimal system, there are ten numerals, 0 through 9. In the binary system, only the numerals 0 and 1 are used. The binary number 10110 is 22 in decimal. To convert a binary number to decimal, we add the places that contain a 1 and ignore the place values where there is a zero.

```
1
2 6 3 1
8 4 2 6 8 4 2 1
0 0 0 1 0 1 1 0
```

In our example, 10110, there is a 1 in the 16's column, a 1 in the 4's column and a 1 in the 2's column. If we add 16+4+2, we arrive at 22, the decimal equivalent of 10110. Most computers have 8 positions in each memory location. This means that each location can contain a number from 0 to 255.

The number that is stored in each memory location is called a *byte.* Each one or zero in the byte is referred to as a *bit.* The ATARI computer is an 8-bit computer. There are some 4-bit and 16-bit computers also. Each byte can also be divided into two 4-bit *nybbles.*

Although it seems confusing at first, using the binary system in computers conserves on space and increases speed. If a switch with ten different settings were used, the computer would first have to determine whether or not the switch was set, and then determine which setting it was pointing to. In binary, there are only two possibilities, a 1 or a 0. It takes only 8 bits or switches to count to 255. By adding 8 more, any number up to 65535 can be displayed. Work the following examples to practice converting binary numbers to decimal.

1. 01100001
2. 10110111
3. 11001000
4. 00111001
5. 01110010
6. 00111100
7. 00011110
8. 11011000
9. 01111010
10. 11110001

The decimal equivalents are: 1-97; 2-183; 3-200; 4-57; 5-114; 6-58; 7-30; 8-216; 9-122; 10-241

UNDERSTANDING HEX

Although the binary system increases the computer's speed, most of us cannot readily convert a string of 1s and 0s into a number that we can understand. To help us, most programmers and manuals reference the memory locations and the numbers stored in them in hex. The hexadecimal system uses the base 16. The numbers after 9 are represented as the letters A-F. To convert a binary number to hex, we first divide the byte into two nybbles. If, for example, we needed to convert 11001101 into hex, we would divide it into two nybbles: 1100 and 1101. Each nybble consists of four bits. Now we treat each nybble as a separate number. By adding the place values of the first nybble, 8+4, we get 12. Twelve is not a one digit number, so we use the letter C. The next nybble is 8+4+1, or 13. One number higher than C is D. Our hex number for 11001101 is CD.

Let's try that again with another binary number: 10010111. Divide this 8-bit number into 2 nybbles: 1001 and 0111. The first nybble is 8+1 or 9, the second is 4+2+1 or 7. The hex number for 10010111 is '97'.

There are times when you will want the decimal equivalent to a hex number. When you are

working in BASIC and want to poke a location with a number, both the location and the number that you are poking must be in decimal. Often the manual you are using will provide only the hex addresses to be poked or the hex values that should be entered. To convert a hex number to decimal is fairly easy. Since each number/letter represents a value from one to 15, each place value in hex is a multiple of 16. If the hex number has only two place, for example, B3, you should multiply the number in the second position from the end by 16 and add the value in the rightmost position. B is equal to 11 decimal, 11×16 is 176. Add 3 and the decimal value of hex B3 is 179. Since the computer can access over 64000 memory locations, the hex number will often contain four places. To convert C253 hex to decimal we would multiply the C (decimal 12) by 4096, the 2 by 256, the 5 by 16 and add 3.

$$(12\times4096)+(2\times256)+(5\times16)+3=49747$$

To convert a decimal number to hex, you should divide the number by the largest place value feasible; the quotient is the value for that place. Then divide the remainder by the next place value, and continue until there is a remainder less than 16. That number is the last number of the hex number. If, for example, the decimal number is 21013, we would divide the number by 4096. The first or leftmost value of the hex number then is a 5. The remainder is 533. When this number is divided by 256, the next quotient is a 2 with a remainder of 21. 21 divided by 16 is 1 with a remainder of 5.

Therefore, the hex equivalent of 21013 is 5215.

```
------5          ---2     --1  5
4096) 21013    256)533   16)21
-20480           -512     -16
------           ----     ---
  533              21        5
```

The following program will convert a decimal number to hex or binary, and a binary or hex number to decimal.

Listing 1-1. Conversions

```
10 REM LISTING 1.1
20 REM BY L.M.SCHREIBER FOR TAB BOOKS
30 REM CONVERSIONS
40 DIM A$(8):REM MOST POSITIONS IN A B
INARY NUMBER
50 GRAPHICS 0:POKE 752,1:? "}CLEAR}":P
OKE 710,100:POKE 712,100:COLOR 18:PLOT
 7,2:DRAWTO 33,2:PLOT 7,4:DRAWTO 34,4
60 REM PRINT cntrl-R to make the top a
nd bottom of box.
70 POSITION 6,2:? "¦":POSITION 3
4,2:? "¦":REM cntrl-R,esc-cntrl-
downarrow,esc-cntrl-backarrow,shift=
80 REM cntrl-R,shift=,cntrl-Z & cntrk-
E,shift=,cntrl-c with down-arrow & bac
```

Listing 1-1. Conversions (continued from page 3).

```
karrow to make left & right sides
90 POSITION 8,3:? "Please enter a sele
ction:"
100 ? :? :? "    1. Decimal to Hex":RE
M escTAB
110 ? :? "    2. Decimal to Binary":RE
M escTAB
120 ? :? "    3. Hex to Decimal":REM e
scTAB
130 ? :? "    4. Binary to Decimal":PO
KE 752,0:REM escTAB
140 TRAP 140:POSITION 19,14:? "  ";:
INPUT N:REM TWO SPACES - 2 esc-cntrl b
Ack arrows - erase a previous answer
150 IF N<1 OR N>4 THEN 140:REM CHECK F
OR CORRECT INPUT
160 ON N GOSUB 200,500,700,900
170 GOTO 50
190 REM ROUTINE TO CONVERT DECIMAL NUM
BERS TO HEX
200 ? "}CLEAR}PLEASE ENTER THE DECIM
AL NUMBER TO BE CONVERTED TO HEX.":? "
 NUMBER CANNOT EXCEED 65534."
205 REM CLEAR SCREEN, 2 esc-cntrl down
s
210 ? :? "TO EXIT THIS ROUTINE, SIMPLY
 PRESS    THE RETURN KEY.":? :?
220 TRAP 50
230 INPUT N
250 IF N>65535 OR N<0 THEN 200
260 N1=N:REM STORE THE NUMBER ENTERED
FOR THE CONVERSION ROUTINE
270 P=1:D=4096:REM P IS THE HEX POSITI
ON - D IS EQUAL TO THE VALUE OF THE HE
X POSITION
280 GOSUB 390:REM USE THE SUBROUTINE T
HAT GETS THE FIRST POSITION
290 P=2:D=256:REM P IS THE NEXT POSITI
ON - D IS THE VALUE OF THAT POSITION
300 GOSUB 390
310 P=3:D=16:REM P IS THE THIRD POSITI
ON - D IS THE VALUE OF THAT POSITION
320 GOSUB 390
330 P=4:D=1
```

```
340 GOSUB 400
350 ? :? "THE HEX EQUIVALENT OF "¡N¡"
IS "¡A$
360 GOTO 210
380 REM THIS ROUTINE DOES THE ACTUAL C
ONVERSION
390 IF N1<D THEN A$(P,P)=STR$(0):RETUR
N
400 H=INT(N1/D):REM DIVIDE THE NUMBER
BY THE VALUE OF THE POSITION
410 N1=N1-H*D:REM STORE THE REMAINDER
FOR THE NEXT CONVERSION
420 IF H>9 THEN A$(P,P)=CHR$(H+55):RET
URN
430 A$(P,P)=STR$(H):RETURN
490 REM CONVERT A DECIMAL NUMBER INTO
A BINARY NUMBER - DIVIDE THE NUMBER BY
 EACH PLACE VALUE
500 ? "}CLEAR}PLEASE ENTER THE DECIM
AL NUMBER TO BE CONVERTED TO BINARY.":
? " NUMBER CANNOT EXCEED 255."
505 REM CLEAR SCREEN, 2 esc-cntrl down
s
510 ? :? "TO EXIT THIS ROUTINE, SIMPLY
 PRESS    THE RETURN KEY.":? :?
520 TRAP 50
530 INPUT N
550 IF N>255 OR N<0 THEN 500
560 N1=N:REM STORE THE NUMBER ENTERED
FOR THE CONVERSION ROUTINE
570 P=1:D=128:REM P IS THE BINARY POSI
TION - D IS EQUAL TO THE VALUE OF THE
BINARY POSITION
580 GOSUB 390:REM USE THE SUBROUTINE T
HAT GETS THE FIRST POSITION
590 P=2:D=64:REM P IS THE NEXT POSITIO
N - D IS THE VALUE OF THAT POSITION
600 GOSUB 390
610 P=3:D=32:GOSUB 390
620 P=4:D=16:GOSUB 390
630 P=5:D=8:GOSUB 390
640 P=6:D=4:GOSUB 390
650 P=7:D=2:GOSUB 390
660 P=8:D=1:GOSUB 400
```

Listing 1-1. Conversions (continued from page 5).

```
670 ? :? "THE BINARY EQIVALENT OF "$N:
? "IS "$A$(1,4)$" "$A$(5,8)
680 GOTO 510
700 ? "}CLEAR}PLEASE ENTER THE HEX N
UMBER TO BE CONVERTED TO DECIMAL."::? "
 NUMBER CANNOT EXCEED FFFF."
710 ? :? "TO EXIT THIS ROUTINE, SIMPLY
 PRESS    THE RETURN KEY."::? :?
720 N=0:INPUT A$
730 IF A$="" THEN 50
740 P=LEN(A$):REM FIND OUT HOW MANY PO
SITIONS
750 IF P>4 THEN 700
760 FOR D=1 TO P:C=ASC(A$(D,D)):IF C>7
0 THEN 700:REM INVALID LETTER/CHARACTE
R
770 C=C-55:IF C<10 THEN C=VAL(A$(D,D))
:REM IF IT'S NOT A LETTER THEN GET THE
 VALUE
780 ON P-D+1 GOTO 820,810,800,790
790 N=C*4096:NEXT D
800 N=N+C*256:NEXT D
810 N=N+C*16:NEXT D
820 N=N+C
830 ? :? "THE DECIMAL EQUIVALENT OF "$
A$$:? " IS "$N
840 GOTO 710
900 ? "}CLEAR}PLEASE ENTER THE BINAR
Y NUMBER TO BE CONVERTED TO DECIMAL."::
? " NUMBER CANNOT EXCEED 11111111."
910 ? :? "TO EXIT THIS ROUTINE, SIMPLY
 PRESS    THE RETURN KEY."::? :?
920 N=0:INPUT A$
930 IF A$="" THEN 50
940 P=LEN(A$):REM FIND OUT HOW MANY PO
SITIONS
950 IF P>8 THEN 700
960 TRAP 900:FOR D=1 TO P:C=VAL(A$(D,D
))
970 IF C>1 THEN 900:REM INCORRECT ENTR
Y
980 IF C=1 THEN ON P-D+1 GOTO 1060,105
0,1040,1030,1020,1010,1000,990
985 NEXT D
```

```
990 N=C*128:NEXT D
1000 N=N+C*64:NEXT D
1010 N=N+C*32:NEXT D
1020 N=N+C*16:NEXT D
1030 N=N+C*8:NEXT D
1040 N=N+C*4:NEXT D
1050 N=N+C*2:NEXT D
1060 N=N+C
1070 ? :? "THE DECIMAL EQUIVALENT OF "
;A$:? "IS ";N
1080 GOTO 910
1090 END
```

Line 40 sets the string space used for the binary or hex numbers.

Line 50 removes the cursor, clears the screen, changes the color of the background and border to violet, and draws the top and bottom of the box. To use the plot and DRAWTO commands, you must specify the graphics mode. The color 18 is a control R.

Line 70 draws the right and left side of the box for the on-screen display. Use a control Q, shift =, and a control Z for the left side of the box; and a control E, shift =, and a control C for the right side. Use a down arrow and a backspace between each character.

Lines 90-130 place the menu on the screen. Keep it neat by using an extra print and a tab in each line.

Line 140 uses the trap command. If a letter or character is entered instead of a number, the program will not crash. The two spaces and backspaces will clear any input that was incorrect.

Line 150 checks the input. If the number entered is incorrect, the program goes back to the previous line and waits for the correct entry.

Line 160 directs the program to the correct routine. The program will return to the menu in line 170.

Lines 200-210 clear the screen and place the directions on the screen.

Line 220 is another trap. If you enter a letter or simply press the return key, you will return to the main menu. When writing a menu driven program, it is a good idea to give the user a way out in case the wrong selection was made.

Lines 230-250 get the number to be converted and check to make sure that the number is within the specified range. If it is not, the program will go back to the beginning of this routine.

Line 260 stores the number entered in another variable, N1. The number in this variable will be converted to a hex number.

Lines 270-340 convert the number into a hex number. The variable P is the position that is being converted. When converting numbers from decimal to hex or binary, we start with the leftmost position and work to the right. The value of the first hex position in a four digit hex number is 4096. This number is stored in the variable D. The program then uses the subroutine that begins with line 390 to convert the number. Each time we return from this subroutine, the value in P is increased by one to reflect the next place in the number and the value of D changes to the value of the place. In line 340 we GOSUB to line 400 since this is the last or one's position and the value here will most likely be larger than the value of D.

Lines 350-360 display the number entered and its hex equivalent. The program goes back to line 210 and waits for another number or a return.

Lines 390-430 do the actual conversion of a decimal number to hex or binary. First the number is compared to the place value. If the number is less than that value, a zero is stored there. Next the number is divided by the place value. The integer or whole number is stored in the variable H. To get the remainder, we multiply the place value by the whole number and subtract it from the number. The new number or remainder is now stored in N1 and will be used when we continue to convert the number. Line 420 is used for decimal to hex conversions. The number is checked to see if it is greater than a 9. If it is, it must be converted into a letter (A-F). This is done by adding 55 to the number. If the number is less than 10, that number is placed into the string. In either case, the routine returns to continue the conversion.

Lines 500-680 convert a decimal number to binary. Again, we trap the input so that pressing the return key will return you to the main menu. The number is tested and stored in N1. Lines 570-660 keep track of which position is being converted and the value of that position. The same subroutine that was used to convert the decimal number to hex is used to convert the decimal number to binary.

Lines 670-680 display the results of the conversion and go back to the beginning of this subroutine.

Lines 700-840 convert a hex number to decimal. This time we are using a string for the input. If the string is empty, the program will go back to the main menu. The program checks the length of A$. If more than 4 letters or numbers have been entered, the program will go back to the beginning of this subroutine. The ASCII value for each number/letter is stored in C. If the value of C is greater than 70, the letter entered is invalid and the program goes back to the beginning of this subroutine. If it is a valid letter/number, 55 is subtracted from it. If C is less than 10, that position contains a number and its value is placed in C. If the position contains a letter, the value of that letter is obtained by subtracting 55 from its ASCII value. The value in C is multiplied by the place value of its position and added to any previous conversion value. The new conversion value is stored in N. Once the hex number has been converted into its decimal equivalent, both numbers are displayed on the screen. This subroutine continues until the return key is pressed.

Lines 900-1080 use the same principle to convert a binary number to decimal. Since the program is expecting only 1s and 0s, a trap is placed in line 960 for any letter/character. The value of each position is stored in C, one at a time. If this value is greater than 1, the program returns to the beginning of this module. If C is 1, the value of that position is added to the value in N. This number will be the decimal equivalent of the binary number.

Chapter 2

Working with the Display List

All microcomputers have one thing in common—a CPU or *central processing unit.* It is the brains or workhorse of the system depending on how you look at it. Whether it is a 6502, 8080, Z-80, or some other processor, it is what keeps the machine going. The ATARI, however, has three special-purpose LSI (large-scale integrated) chips to take some of the burden off the 6502. They are called ANTIC, CTIA, and POKEY. This chapter will discuss the purpose of ANTIC and how to make use of its capabilities.

THE PURPOSE OF ANTIC

ANTIC is a microprocessor that is dedicated to updating the screen display. It is a true microprocessor because it has its own instruction set, a program, and data. It operates simultaneously and in conjunction with the 6502. ANTIC uses the same bus and memory locations that the 6502 does. In order to operate, ANTIC must stop or halt the 6502, get its information, and then let the 6502 continue its work. Both must operate without missing a step.

ANTIC's job is to keep the screen updated with the current information. A television screen is normally updated 60 times a second. Since ANTIC must stop the 6502 each time it does its job, the higher the resolution that is displayed on the screen, the more often the 6502 will be interrupted. If you turn off the ANTIC chip completely, the 6502 will operate at maximum efficiency.

The ATARI computer and many other personal computers like the Apple and the TRS-80 have memory mapped screen displays. This means that the information that you see on the screen is also stored in a specific portion of the computer's memory. The higher the graphics resolution, the more memory is needed to store this information. The upper left corner of the screen is the lowest memory address, and the lower right corner is the highest memory address. The address of each location in between follows the first address sequentially. In most other computers, you have a choice of one or two graphic modes, or text. It is not always possible to mix the types of graphics modes. The ATARI offers 14 different modes with its CTIA graphics chip. Each mode can be displayed individually or mixed with the others. This is the reason for the ANTIC chip. It has its own program just above the memory set aside for the screen. This program is really a list of the graphics modes that we are using in our program. ANTIC checks this list for the mode of the line, and sends this information along with the data that will be displayed on this line to the CTIA

(or GTIA) chip. This chip is the television interface chip. It in turn converts the information that it receives into the signal that we see as a picture or text on the screen. If each line is a different mode, ANTIC tells this to the CTIA (GTIA) chip, and it translates the signal accordingly.

FINDING THE DISPLAY LIST

Just before the memory set aside for the screen display is the display list that ANTIC uses. The address of the display list is stored at memory locations 560 and 561. To obtain the address of the display list enter this command.

```
PRINT PEEK(560)+PEEK(561)*256
```

The number that appears on the screen is the starting address of the display list. In graphics mode 0, the display list is 32 bytes long. To see the list type:

```
10 DLIST=PEEK(560)+PEEK(561)*256
20 FOR X=DLIST TO DLIST+31:? PEEK(X):NEXT X
```

Your list should look like:

```
 112
 112
 112
  66
  64  may
 156  differ
   2 —
   2   —
   2  —  (23 twos)
   2  —
   2 —
  65
  32  may
 156  differ
```

Before interpretting the display list, you must understand your screen. If you look very closely at the screen, you will see that each character is made up of tiny dots. The picture that you see on the screen is actually many rows of dots stacked from the top of the screen to the bottom of the screen. You do not actually see the complete picture on the screen. The parts of the picture that are above, below, or to the sides of the screen are called *overscan.* The picture that is transmitted from the networks contains information on the lines above the actual picture. You never see this information because it is in the overscan area.

Now look at the display list again. The first three numbers tell ANTIC to display three blank lines that are 8 rows high. This makes sure that the text or picture will be on the screen and not in the overscan area.

The next number, 66, is a two part instruction. A 64 tells ANTIC that the following two bytes contain the address of the beginning of the screen display area. The 2 added to the 64 is the ANTIC mode of the first line of the screen. Any number from 2 to 15 can be added to the 64. The number added is always ANTIC's value for the graphics mode. We will discuss the fourteen

graphics modes and their ANTIC values later in this chapter. For now, your screen is in graphics mode 0, which is ANTIC mode 2.

If you multiply the sixth number in the display list by 256 and add the fifth number of the display list, you would know the exact memory location of the first location in the first line on the screen, or position 0, 0. If you poked this location or any location after this one with a value, you would see characters on your screen. This is a memory mapped screen display. The contents of the memory locations set aside for the screen are visible on the screen. When you use the locate command in BASIC and you tell the computer LOCATE 5,6,B, the computer calculates the memory location of the fifth column and the sixth row based on the mode that we are using and sets the variable B to the value in that memory location. If your screen display begins at memory location 40000 (40K system), and you are in graphics 0, you can use the locate command to find out what is stored on the screen at any location. The command in this example is LOCATE 5,6,B. The computer multiplies the row number by the number of characters in the row, in this case 40, and adds the column number: 6 times 40 plus 5 equals 245. This number is added to the first memory location for the screen. The computer examines that memory location and changes the value found there to ATASCII. This value is stored in B, and the computer can tell you the value of location. Now that you know how it's done, you can use either the locate command, or the peek function for yourself.

Going back to the list of numbers that make up the display list, you see that there is a string of 23 twos before you come to any other numbers. In graphics mode 0, there are 24 rows on the screen. ANTIC knows from the display list that the first row is in graphics mode 0. It also knows where the beginning of the screen display is. The next 23 numbers tell ANTIC that the next 23 rows will all be in graphics mode 0 or ANTIC mode 2. The next number after all the twos is a 65. This again is a two part instruction. The 64 tells ANTIC that there is an address in the next two bytes that it should jump to, and the 1 tells ANTIC to wait for a vertical blank before jumping.

When the last number in this list is multiplied by 256 and then the number just before it is added to this product, the total is equal to the beginning address for this display list. Thus ANTIC will repeat these same instructions over and over again until they are changed. It will start at the beginning of the display list, look at the mode the first line is in, take the information from the screen display area, and transfer this information to the CTIA chip for the actual screen display. It goes to the next line, looks at the mode, gets the next lines information, and transfers it. The process is repeated until it reaches the 66, where it waits for a signal called a *vertical blank* and goes back to the beginning of the display list.

What's a vertical blank? The picture that you see on your television screen, whether it's generated by a computer or comes from the main television station, is "painted" on your screen line by line. A beam called a *raster scan* starts at the left side of the screen (as you look at it), and paints a picture by turning on the correct color beams in that row. When it reaches the end of the row, the beam shuts off, retraces the line, steps down a row, and turns itself back on to paint another line on the screen. The line that has been painted is called a horizontal scan line. The time that the beam is shut off to retrace the line is called a *horizontal blank.* When the beam reaches the last row on the screen, it not only has to come back to the left side of the screen, but also has to go back to the top of the screen. This period of time, while the beam is shut off and going back to the top side of the screen, is called the vertical blank. If the computer was not synchronized with the television, the display would appear jittery and distorted. By waiting for the vertical blank before beginning the display list again, ANTIC is sure that the display will appear correctly on the screen.

ANTIC'S INSTRUCTION SET

Looking at the display list again, you see that ANTIC does have a set of instructions that it follows. A blank line can be inserted anywhere in the display list. This blank line can be from one to eight pixels tall. In the display list, the first three numbers were 112, 112, 112. In hex this translates to 70, 70, 70. When you want to tell ANTIC to send a blank line, you use an instruction that only uses the left nybble of the byte. Table 2-1 shows the values used to issue a blank line from one to eight pixels high.

ANTIC is also capable of jumping. At the end of the display list, it has the instruction 65. In a jump instruction, the second nybble of the byte is always set to 1. The first nybble can be either 0 or 4 hex. If the instruction is 1, ANTIC will jump to the address indicated in the next two bytes, and continue there. If the instruction is 65 or 41 hex, ANTIC will wait for the vertical blank before jumping. The jump instruction, 1, is used when the display list must cross a 1K boundary in memory. ANTIC cannot calculate past 255, so it must be told to jump to the next location. This jump should be placed just before the boundary. The two bytes after the jump instruction should contain the low order and then the high order address of the location that ANTIC should jump to.

ANTIC also has display instructions. This time the instruction byte is divided into its individual bits. The last four bits of this byte tell ANTIC which display mode to use. ANTIC's display modes are 2 - 15. If bit 4 is set to 1, then ANTIC can do a horizontal scroll. If it is 0, it cannot. Bit 5 is set to enable the vertical scroll and when bit 6 is set, ANTIC will use the next two bytes as the beginning of the screen memory. See Fig. 2-1.

COMBINING GRAPHICS MODES

Looking at the ATARI manual, you see that there are eight graphics modes, one for normal text, two for color text, and five for graphics. Yet, looking at ANTIC's instruction set, you can see that there are fourteen different modes. Six of these modes cannot be accessed by BASIC with the graphics command. They can, however, be used with BASIC when you create your own display lists.

Combining the different graphics modes and creating your own display list takes a little imagination and the ability to add to 192. Why 192? Think back to the television screen and the horizontal scan. The beam that paints a picture on the screen does so row by row. Every row is the same height. Therefore, every picture has the same number of scan lines. When you count the number of pixels used in a character in graphics mode 0, you see that the character stands 8 pixels or rows high. There are 24 rows in mode 0. $24 \times 8 = 192$. Multiply the number of rows for any other mode time the number of pixels used in one row. The answer will always be 192. Therefore,

Table 2-1. Values to Issue Blank Lines.

blank lines	hex value	decimal value
1	00	0
2	10	16
3	20	32
4	30	48
5	40	64
6	50	80
7	60	96
8	70	112

```
        --- --- --- --- --- --- --- ---
bit #   7   6   5   4   3   2   1   0
```

set bit	instruction
7	display list instruction interrupt
6	load memory scan with next two bytes
5	vertical scroll
4	horizontal scroll
3-0	display mode

Fig. 2-1. Set bits for display list.

when you create a new display list, you want the number of rows used in each of the modes to total 192.

To create a multi-mode screen, you must first decide what you want your screen or display to look like; which graphics modes are best suited for the display; and how you want to organize the general layout of the screen. Once you decide how the screen should look, you are ready to create a new display list. As an example you are writing a program that involves some text, some prompting from the program, and a score that is kept on the screen. The main part of the program will be done in graphics mode 7. The first thing that you have to decide is how many lines you need for the text. This program will only need two lines for text: one line at the top and one line near the bottom. This leaves the rest of the screen for the graphics. In a full screen (no text window), graphics mode 7 has 96 rows, each 2 pixels high (96×2=192). For this program you will use graphics mode 2 at the top and graphics mode 1 near the bottom. To calculate the number of rows that will be in graphics mode 7, you must first determine the number of rows of pixels the other two modes will use. The characters in graphics mode 2 are 16 rows high. The characters in graphics mode 1 are only 8 rows high. This means that 24 rows of the screen will be used by these two modes (8+16=24). By subtracting this number from 192, you can calculate that there are 168 rows left for graphics mode 7 (192−24=168). Since each row of graphics mode 7 uses two rows of pixels, there will be 84 rows of graphics mode 7 on the screen (168/2=84). The following program, which draws a baseball diamond, creates a new display list for your program.

Listing 2-1. Mixed Modes Program

```
10 REM LISTING 2.1
20 REM MIXED MODES
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
1982
40 DIM M$(20):GRAPHICS 23:REM HAVE COM
PUTER SET DISPLAY LIST FOR GRAPHICS 7
WITH NO TEXT WINDOW
50 DLIST=PEEK(560)+PEEK(561)*256
60 ANTIC=PEEK(559):REM GET THE VALUE I
N THE SHADOW FOR ANTICS'S STATE
70 POKE 559,0:REM SHUT OFF ANTIC FOR D
ISPLAY LIST CHANGES
80 POKE DLIST+3,71:REM MAKE THE FIRST
```

Listing 2-1. Mixed Modes Program (continued from page 13).

```
LINE GRAPHICS 2
90 POKE DLIST+89,6:REM MAKE THE BOTTOM
 ROW GRAPHICS 1
100 POKE DLIST+90,65:REM MOVE THE JUMP

110 POKE DLIST+91,PEEK(560)
120 POKE DLIST+92,PEEK(561)
130 POKE 559,ANTIC:REM TURN ANTIC BACK
 ON TO IT'S PREVIOUS STATE
140 MEMORY=PEEK(DLIST+4)+PEEK(DLIST+5)
*256
150 M$="02%33@(@%/2@()4":REM INVERSE O
PEN PARENTHESIS - FIRST ONLY
160 BOTTOM=MEMORY+2
170 FOR X=BOTTOM TO BOTTOM+LEN(M$)-1:P
OKE X,ASC(M$((X-BOTTOM+1),(X-BOTTOM+1)
)):NEXT X
180 M$="25.3@@@@@@25.3":REM LAST FOUR
CHARACTERS ARE INVERSE
190 BOTTOM=83*40+MEMORY+22
200 FOR X=BOTTOM TO BOTTOM+LEN(M$)-1:P
OKE X,ASC(M$((X-BOTTOM+1),(X-BOTTOM+1)
)):NEXT X
210 COLOR 1:PLOT 0,1:DRAWTO 60,40:DRAW
TO 0,80:PLOT 159,0:DRAWTO 99,39:DRAWTO
 159,79
250 GOTO 250
```

Line 40 sets M$ to 20. This string will be used to store the characters for the message that will appear, on the screen. The graphics mode is set to 23, which is graphics mode 7 without the text window. Of the three modes that will be used, graphics 7 mode uses the most memory, so it is advantageous to set the mode to 7. The computer will calculate the amount of screen memory needed and move the display list above it. This leaves you with less to figure out.

Line 50 finds the address for the beginning of the display list.

Line 60 stores ANTIC's state in the variable ANTIC. The value in this memory location varies depending on what control the program will have. It determines the width of the *playfield* and the resolution of *players* and *missiles,* and enables the use of *player/missile* graphics.

Line 70 turns off ANTIC while the display list is changed. If ANTIC was left on, it would be using the values in the display list while it was being changed. This could cause the program to crash.

Line 80 changes the first row on the screen from graphics mode 7 to graphics mode 2. Remember, you must add the display instruction to the graphics mode in this location.

Line 90 changes the last row on the screen to graphics mode 1. You must add 89 to the display

list to arrive at the position of the last row of the screen in the display list 89 is obtained in the following manner:

1. The total number of rows on the screen, calculated in terms of graphics mode 7, are tabulated. This number is 96. (Each row is two pixels high.)
2. The total number of rows, calculated in terms of graphics mode 7, done in modes 1 and 2 are calculated. Row 1, which is in mode 2, is 16 pixels or 8 mode-7 rows high. The last row, which is in mode 1, is 8 pixels or 4 mode-7 rows high. The total (8+4) is 12.
3. The 12 from item number two is subtracted from the 96 obtained in item one: 96–12=84. These are rows 0-83 in terms of graphics mode 7. We know that the first three bytes of the display list (0-2) are for the overscan. The fourth byte (3) sets the first row on the screen, and the next two bytes (4 and 5) indicate the memory location of the screen. 83+5=88—therefore the 89th row in the display list is set for graphics mode 1.

Lines 100-120 move the jump and the address of the beginning of the display list into the correct position.

Line 130 turns ANTIC back on to its previous state. Now it can execute the new display list.

Line 140 finds the location of the screen memory in the display list and stores it in the variable MEMORY. We will use this value when we display the message on the screen.

Line 150 places the message in M$. The first open parenthesis is in inverse video. If you have been replacing the character set in your ATARI with a new character set in RAM, you have probably discovered that the character set does not follow its ATASCII or decimal values. Because we have set the graphics mode to 7, the computer will not print a message as such on the screen. Graphics mode 7 means: "use only two bits from every byte; add four 2-bit combinations together and store them in a memory location on the screen." If we told the computer to PRINT"PRESS" on the screen, it would use only the last two bits of each letter between the quotation marks. The PRES would be combined for one character, the S would be the second. The character on the screen would be the character that was in that location of the ATARI character set. In the next chapter we will redesign a character set.

Line 160 stores the value of MEMORY + 2 in the temporary variable BOTTOM.

Line 170 is a for . . . next loop that pokes each character of M$ into the memory that is reserved for the screen. Now the ATASCII value of each character of M$ is stored directly on the screen.

Line 180 stores a new message in M$. This is the message that will appear on the bottom of the screen. Once again, the message that appears on the screen is not the same as the characters between the quotes.

Line 190 calculates the memory location of the last row on the screen.

Line 200 pokes this message directly into the screen memory.

Line 210 draws the diamond on the screen. This can be done with the simple plot and DRAWTO commands. Notice that the position PLOT 0,1 is actually the center of the screen rather than the left side. All the positions are shifted over by 80. This often happens when different modes are mixed. Our first row on the screen is in graphics mode 2. ANTIC took the first 80 bytes for this row because it was set for graphics mode 7 which uses 4 bytes for every location on the screen. It knew, though, that graphics mode 2 only needed 20 locations on the screen (20×4=80). Graphics mode 7 uses 160 bytes for every line. The next 160 bytes are displayed on the next row on the screen. The computer thinks that the first 80 bytes belong to the previous row. When the computer calculates the memory position for PLOT 0,1, though, it does not look at the display list to see if any of the rows have more or less bytes in them than the present mode

calls for. It simply multiplies the row number by the number of bytes in a row. Because this is a graphics mode, before it adds the column, it divides by the number of bytes used to display one byte (in this mode, 4) and then adds the column divided by 4. Every row on the screen will be off center because of this method of calculation.

Line 250 loops back on itself. You can stop this program by pressing the system reset key.

THE FOURTEEN MODES

One of the features of the ATARI that has not been widely publicized is that the number of graphics modes is not 9 but 14 with the CTIA chip. When we are working with BASIC, we have 9 choices, Graphics modes 0-8. Graphics mode 9-11 are reserved for the GTIA chip. Any other number will give us an error message. However, when we work directly with the display list, we learn that ANTIC will set the screen for any graphics mode from ANTIC 2 to ANTIC 15. Table 2-2 shows us the different modes and the unique features of each mode. ANTIC modes 2-7 are all text modes. ANTIC modes 8-F are graphics modes. ANTIC gives you three additional text modes and two additional graphic modes.

TEXT MODES

There are two 2-color text modes available: the standard text mode referred to as Graphics mode 0, and ANTIC mode 3. ANTIC mode 3 has ten scan lines or rows for each character. This feature allows for true descenders on the letters q, y, p, j and for subscripts and superscripts. Because each line in this mode uses ten scan lines or rows on the screen, the display will be two rows shorter than a normal display (10×19=190) or eight rows longer (10×20=200).

When you redefine a character set for use in this mode, you do not need to add two blank bytes either before the character or after it. When the computer reads the bytes for the characters from the character set, it will add the two blank bytes automatically to the character. For all the characters, numbers, and uppercase letters, the two blank bytes will be added to the bottom of the letter. Lowercase letters are treated differently. The computer takes the first and second byte of the character and places it in the ninth and tenth row of the character on the screen. It places 2 blank lines in the first and second row and then places the rest of the bytes in the correct position. This can cause some problems if you are using this mode with the character set in ROM.

In the following program, the display list will be changed to use ANTIC 3. A message will be printed on the screen using the character set from ROM. Notice the distortion in the letters l, j, and k. There is no distortion in any of the capital letters, numbers, or graphics because they are not the last 32 characters/letters of the character set.

value in two bits of graphic byte	color register
00	0 (background)
01	1
10	2
11	3

All four color registers are used in GRAPHICS 3, 5, 7.
Color registers 0 and 1 are used in GRAPHICS 4 and 6.

Fig. 2-2. Color register codes.

Listing 2-2. ANTIC 3

```
10 REM LISTING 2.2
20 REM ANTIC 3
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 ? ")Aa Bb Cc Dd Ee Ff Gg Hh Ii JJ
 Kk Ll Mm Nn Oo Pp Qq Rr Ss Tt Uu Vv W
w Xx Yy Zz ,;!#$%&'8@()90"
50 DLIST=PEEK(560)+PEEK(561)*256:ANTIC
=PEEK(559):REM GET THE VALUE IN THE SH
ADOW FOR ANTICS'S STATE
60 POKE 559,0:REM SHUT OFF ANTIC FOR D
ISPLAY LIST CHANGES
70 POKE DLIST+3,67:REM MAKE THE FIRST
LINE ANTIC 3
80 FOR X=6 TO 24:POKE DLIST+X,3:NEXT X
:REM CHANGE LIST FOR ANTIC 3
90 POKE DLIST+25,65:REM MOVE THE JUMP
100 POKE DLIST+26,PEEK(560)
110 POKE DLIST+27,PEEK(561)
120 POKE 559,ANTIC:REM TURN ANTIC BACK
 ON TO IT'S PREVIOUS STATE
130 A=PEEK(106)-8:NB=A*256:REM PLACE C
HARACTER SET 2K BEFORE END OF MEMORY -
 STORE VALUE IN NB
140 CB=PEEK(756)*256:TH=CB:X=TH:REM CH
ARACTER SET IN ROM
150 FOR LC=0 TO 11:POKE NB,PEEK(X+7):N
B=NB+1:FOR X=TH TO TH+6:POKE NB,PEEK(X
):NB=NB+1:NEXT X:TH=TH+8:X=TH:NEXT LC
160 FOR X=TH TO TH+7:POKE NB,PEEK(X):N
B=NB+1:NEXT X:TH=X:REM SAME
170 FOR LC=13 TO 26:POKE NB,PEEK(X+7):
NB=NB+1:FOR X=TH TO TH+6:POKE NB,PEEK(
X):NB=NB+1:NEXT X:TH=TH+8:X=TH:NEXT LC
180 FOR X=TH TO TH+7:POKE NB,PEEK(X):N
B=NB+1:NEXT X:TH=X:REM SAME
190 FOR LC=28 TO 64:POKE NB,PEEK(X+7):
NB=NB+1:FOR X=TH TO TH+6:POKE NB,PEEK(
X):NB=NB+1:NEXT X:TH=TH+8:X=TH:NEXT LC
200 FOR LC=65 TO 79:FOR X=TH TO TH+7:P
OKE NB,PEEK(X):NB=NB+1:NEXT X:TH=X:NEX
T LC:REM SAME
210 POKE NB,PEEK(X+7):NB=NB+1:FOR X=TH
 TO TH+6:POKE NB,PEEK(X):NB=NB+1:NEXT
X:TH=TH+8:X=TH:REM F
```

Listing 2-2. ANTIC 3 (continued from page 17)

```
220 FOR LC=81 TO 90:FOR X=TH TO TH+7:P
OKE NB,PEEK(X):NB=NB+1:NEXT X:TH=X:NEX
T LC:REM SAME
230 FOR LC=91 TO 123:POKE NB,PEEK(X+7)
:NB=NB+1:FOR X=TH TO TH+6:POKE NB,PEEK
(X):NB=NB+1:NEXT X:TH=TH+8:X=TH
240 NEXT LC
250 FOR X=TH TO TH+7:POKE NB,PEEK(X):N
B=NB+1:NEXT X:TH=X:REM SAME
260 POKE NB,PEEK(X+7):NB=NB+1:FOR X=TH
 TO TH+6:POKE NB,PEEK(X):NB=NB+1:NEXT
X:TH=TH+8:X=TH:REM F
270 CB=A*256+103*8:POKE CB+1,PEEK(CB):
REM MOVE BYTE DOWN ONE
280 POKE CB,PEEK(CB+7):REM REPEAT LAST
 BYTE
290 CB=A*256+106*8:POKE CB+1,PEEK(CB):
REM MOVE BYTE DOWN ONE
300 POKE CB,PEEK(CB+7):REM REPEAT LAST
 BYTE
310 CB=A*256+112*8:POKE CB+1,PEEK(CB):
REM MOVE BYTE DOWN ONE
320 POKE CB,PEEK(CB+7):REM REPEAT LAST
 BYTE
330 CB=A*256+113*8:POKE CB+1,PEEK(CB):
REM MOVE BYTE DOWN ONE
340 POKE CB,PEEK(CB+7):REM REPEAT LAST
 BYTE
350 CB=A*256+121*8:POKE CB+1,PEEK(CB):
REM MOVE BYTE DOWN ONE
360 POKE CB,PEEK(CB+7):REM REPEAT LAST
 BYTE
370 POKE 756,A
```

Line 40 is the line that will be printed on the screen. It clears the screen and prints each letter of the alphabet in both upper and lowercase. It also prints some numbers, symbols, and graphics on the screen.

Line 50 stores the beginning location of the display list in DLIST and the display status in ANTIC.

Line 60 shuts off ANTIC so that we can change the display list.

Lines 70-110 change the display list from ANTIC mode 2 to ANTIC mode 3. The jump command is moved up since there will be fewer display lines in this mode. The beginning address of this display list is also moved up.

Line 120 turns ANTIC back on with the same value or status that it had before we changed the display list.

Line 130 subtracts 8 from the amount of memory available in our computer. This value is 2K less than the amount of memory in our computer. We will leave the top 1K for the screen display and the display list. The second 1K will be used for the character set. We will store the actual decimal address in NB.

Line 140 stores the address of the character base in ROM in the variable CB. We also want to store this value in two more variables. Now we can manipulate the address without losing it.

Lines 150-260 move the character set from ROM into RAM. If you look at your screen while the character set is being moved, you will see the distortion in several of the lower case letters. You will also see that the letters with descenders are no different in this mode than they are in mode 0. Line 150 transfers the characters from the space to the plus sign from ROM into RAM. When we transfer these characters, we want to take the last byte of the character and place it into the first byte for the character in RAM. The rest of the character will remain the same. Now the character will appear one scan line lower on the screen. The next character is the comma. We do not want to move its last byte first because the comma uses its last byte for data. The other characters that we just transferred do not. If we moved the comma the same way that we moved the other characters, the character would be distorted on the screen when it was printed because the data that it needs for the last row would be in the first row. The 13th through 26th characters will move down one scan line when they are moved. What we are doing in these lines is lowering all the characters one scan line so that the lower case characters that use the second byte for data will not be distorted on the screen (l,j,k,d,b,f,i,t). The characters that use the 8th byte for data, the graphic characters, the comma, and the semicolon are moved into RAM exactly the way they appear in ROM.

Lines 270-330 move the data that is in the first byte for these letters into the second byte and repeat the data from the eighth byte into the first byte. These are the letters that have descenders. By adding one more byte of data to these letters, they will appear on the screen with true descenders.

When the program is finished, move the cursor over some of the letters on the screen. You will see that the uppercase letters are centered and the letters with descenders are aligned with the last two rows of the cursor.

The only letters or characters that place the first two bytes in the last two rows on the screen are the characters from the heart to the end of the character set. By replacing some of the lowercase letters/characters with numbers, you could create superscripts and subscripts. If you are not using any of the lowercase letters, you do not have to transfer the character set the way we did in this program. As long as you are using uppercase letters, numbers, and graphics, you can do a direct transfer of the characters, then create your own characters that would occupy the area normally used for the lower case letters.

COLOR TEXT MODES

As you can see in Table 2-2, ANTIC modes 6 and 7 correspond to graphics modes 1 and 2. These modes are capable of producing text in four different colors, plus a background color. The difference between these two modes is the size of the characters. Graphics mode 1 characters are just as high as graphics mode 0 characters, but they are twice as wide. Graphics mode 2 characters are both twice as high and twice as wide as graphics mode 0 characters. If you are writing a program where you need colorful letters, numbers, or characters of your own creation, it is much wiser, in terms of memory use to use one of these modes with a redefined character set than to use a higher resolution mode that uses more memory.

Table 2-2. All Display Modes Available with CTIA.

ANTIC MODE	BASIC MODE	COLORS	CHARACTER POINT SIZE
2	0	2	8x8
3	–	2	8x10
4	–	4	8x8
5	–	4	8x16
6	1	5	16x8
7	2	5	16x16
8	3	4	8x8
9	4	2	4x4
A	5	4	4x4
B	6	2	2x2
C	–	2	2x1
D	7	4	2x2
E	–	4	2x1
F	8	2	1x1

There are two more text modes between graphics mode 0 and graphics mode 1. These are ANTIC 4 and ANTIC 5. Both of these modes are unique in that they support multicolored text. They can also be used when you want to produce redefined characters that give the illusion of high resolution graphics using only a fraction of the memory.

Normal text cannot be used in either of these modes because unlike graphics modes 1 and 2, these modes produce both the color and the shape of the character from the data in the character set. Normal text is illegible in these modes. The following program changes the display list to illustrate these ANTIC modes.

Listing 2-3. ANTIC 4 and 5

```
10 REM LISTING 2.3
20 REM ANTIC 4 & 5
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 ? "}CLEAR}":REM CLEAR THE SCREEN
50 DLIST=PEEK(560)+PEEK(561)*256:REM F
IND THE BEGINNING OF THE DISPLAY LIST
60 X=PEEK(DLIST+3):POKE DLIST+3,X+2:RE
M CHANGE THE FIST ROW TO ANTIC 4
70 FOR X=DLIST+6 TO DLIST+28:POKE X,4:
NEXT X:REM CHANGE ALL THE ROWS EXCEPT
THE FIRST ONE TO ANTIC 4
80 LIST :REM LIST THIS PROGRAM ON THE
SCREEN TO SHOW THE MULTICOLOR MODE
90 FOR X=1 TO 500:NEXT X:REM WASTE TIM
E
100 X=PEEK(DLIST+3):POKE DLIST+3,X+1:R
EM CHANGE THE FIST ROW TO ANTIC 4
110 FOR X=DLIST+6 TO DLIST+28:POKE X,5
:NEXT X:REM CHANGE ALL THE ROWS EXCEPT
 THE FIRST ONE TO ANTIC 5
120 LIST :REM LIST THIS PROGRAM ON THE
 SCREEN TO SHOW THE MULTICOLOR MODE
```

When this program is run, the listing is shown in vivid yellow and blue. It is nearly impossible to read because the letters blur into each other. ANTIC 5 is nearly the same as ANTIC 4. The only difference is the letters/characters are larger.

If you look again at Table 2-2 you will see that there are four different modes that support four colors, three character colors plus the background color. The fourth mode is sandwiched between graphics modes 7 and 8. With this mode, ANTIC E, each point on the screen is only one scan line high, but two pixels wide. It's height is half that of a point in graphics mode 7, but it is just as wide. To use this mode, you would set the screen for graphics mode 8, then change every row in the display list from 15 to 14. This is the highest graphics resolution that is available with color.

In this table, you will also see an additional mode for two color graphics. It is located between graphics modes 6 and 7. This ANTIC mode is identical to ANTIC E except for the number of colors that it supports. With this mode, you can only set one color plus the background color.

Graphics mode 8 is listed as a two color mode. In actuality, it is like graphics mode 0. The second color depends on luminance or brightness rather than being a true second color.

Each of the modes from ANTIC 8-F uses more memory than the text modes. With the text modes, each memory location is one character. If our screen display is 40 characters wide by 24 rows, we are using 960 bytes for the screen display. The size of each character is 8×8 pixels. Which pixels are on and which are off is determined by the information in the character set. Each character uses 8 bytes of information stored in the character set. If the ATASCII value of the character is less that 128, the character will appear in normal video. If the ATASCII value is greater than 127, that is, if the high order bit is set, the character will appear in inverse video on the screen. The same holds true for ANTIC mode 3.

In graphics modes 1 and 2, which are both text modes, the character on the screen will be either 8×16, or 16×16. Each row on the screen is *20 bytes* long. Graphics mode 2 with no text window uses 480 bytes of memory (24×20). Graphics mode uses 240 bytes of memory (12×20) for the screen display. Both of these modes use only the last 6 bits of the internal value of the character on the screen. The first two bits determine which color is used. This is why only half of the character set can be displayed at one time. (If you have redefined characters, you will know that the order that the characters appear in the ROM character set is not according to their ATASCII value.) By using the first two bits of each byte, the computer can choose one of four colors for the character that will be displayed on the screen.

00 (character) = color 0
01 (character) = color 1
10 (character) = color 2
11 (character) = color 3

The background color is set with color 4.

The nontext modes do not follow this pattern. Graphics modes 3, 5, 7, and ANTIC mode E are four color modes. The character blocks on the screen can be one of three colors, the background is the fourth color. Each point that appears on the screen is not a byte. It is, rather, part of a byte. All the modes that are not text modes use only one or two bits per byte to determine the color that will appear on the screen.

Graphics mode 3 displays 10 bytes in each row. With no text window there are 24 rows. 240 (10×24) bytes of memory are used for screen display in this mode. Let's say that we tell the computer to plot a point on the screen in position 0,0. We want the point to appear orange, so we will use COLOR1. The computer will determine where in memory the first byte of the screen is located. This address is found in the display list. When the computer looks at the value of this byte, it should see 0000 0000 because there is nothing on the screen in this location or the next three locations. After the orange point is on the screen, the value of this byte of memory will be 64. The binary value of this memory byte is 0100 0000. The first two bits now tell the computer to use the color in color register 1. The next three positions are left empty. If we wanted to use green, the color in the second color register, the value of this memory byte would be 128 decimal, or 1000 0000 binary. When a color is plotted in the next position, the computer changes the values in the next two bits of the memory byte. Refer to Fig. 2-2 for the color values and to Fig. 2-3 for a sample point.

Graphics modes 5 and 7 are nearly the same. Graphics mode 5 uses 20 bytes in each row, and graphics mode 7 uses 40. Because of the way that the ATARI stores the color information, only three colors plus a background color can be displayed in these modes. If the computer used 4 bits per byte to display color, we could have 16 colors on the screen. Of course, the computer would need twice as much memory for the screen display.

The number of pixel rows per graphic row will depend on which mode we have chosen. Graphics mode 3 is 8 pixels high, Graphics mode 5 is 4, and graphics mode 7 is 2. ANTIC E also displays three colors plus the background color. It uses one row of pixels per row.

Graphics modes 4 and 6 can display only one color in addition to the background color. Each bit in the byte of memory on the screen determines whether or not that point will be on. If there is a 1, the color in color register 0 will appear on the screen. If there is a 0, that point will appear blank. Mode 4 is the same resolution as mode 5, but because each memory byte in mode 4

Fig. 2-3. Screen byte.

* The four points on the screen are one byte. The color of each point is determined by two bits. In graphics mode 3, if the byte is 00 01 00 11, then the four points would be: off (or background), orange, off (or background), blue.

represents one point on the screen, it uses half as much memory for screen display as mode 5. Graphics mode 4 displays 10 bytes in every row on the screen. Each point is 4 pixels high. Graphics mode 6 uses 20 bytes in each row. Each point is 2 pixels high. The ANTIC mode C also displays one color plus the background color. It displays 20 bytes in each row. Each row is one pixel high.

Each of the graphics modes described above is as many pixels wide as it is high. Each point appears as a square on the screen. The two ANTIC modes are both two pixels wide, but only one pixel or scan line high.

Graphics mode 8 is the only mode where each pixel can be individually controlled. Each row on the screen is 40 bytes. Each bit of each byte controls one pixel on the screen. If the bit is a 1, the pixel will be on. If it is a 0, it will be off. Unfortunately, the color of the on pixel is set by the luminance or brightness of the color in color register 1, but the actual color is the same as the background color. Graphics 8 uses the same amount of memory as ANTIC E.

Remember the first program in this chapter where we poked values onto the screen? The computer thought that we were in a graphics mode. Each character that it was given to print on the screen was treated as a graphics character. That is, it took the last two bits of the character and stored them in a byte for the screen display. It took the last two bits of the next character and stored them in the next two bits of the memory byte. By poking the data directly into the screen memory, we were able to trick the computer into printing the characters that we wanted on the screen. The characters also had to be the hardware value for the characters that we wanted displayed rather than the ATASCII value. We will go into the differences between the hardware value of a character and its ATASCII values in Chapter 12.

Listing 2-4. Color Artifacting

```
10 REM LISTING 2.4
20 REM COLOR ARTIFACTING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 8
50 DLIST=PEEK(560)+PEEK(561)*256
60 C=PEEK(DLIST+3):POKE DLIST+3,C-1
```

Listing 2-4. Color Artifacting (continued from page 23).

```
70 X=6
80 IF PEEK(DLIST+X)=15 THEN POKE DLIST
+X,14
90 IF PEEK(DLIST+X)=C THEN POKE DLIST+
X,C-1
100 X=X+1:IF X>200 THEN 120
110 GOTO 80
120 COLOR 1:FOR R=0 TO 9:FOR C=0 TO 15
9:PLOT C,R:NEXT C:NEXT R
130 COLOR 1:FOR R=10 TO 19:FOR C=1 TO
159 STEP 2:PLOT C,R:NEXT C:NEXT R
140 FOR R=20 TO 29:FOR C=0 TO 158 STEP
 2:PLOT C,R:NEXT C:NEXT R
150 STOP
```

This program changes the display list to ANTIC E. Using only one color register, it paints different colors on the screen.

Chapter 3

Graphics

Once we have decided on which mode or modes we will be using for our program, we can decide whether or not the standard character set will be suitable for our program. If it is, we can start working on the program. More than likely, if we have a particular design to our program, we will want to construct our own character set. We may want to redesign all the letters for special effects, or we may just want to recreate a few of the graphics characters for our particular application. Before we can redesign the characters, we must understand the size and structure of the character set.

CHARACTER SET MAKE-UP

As we determined in the previous chapter, each character in graphics 0 is 8 pixels or one byte wide and 8 rows or bytes high. The information to create one character on the screen occupies 8 bytes in the character set. The entire character set uses 1K or 1024 bytes of memory. This information is stored in ROM beginning with location 57344.

The characters in the character set are not in ATASCII order. In order to implement the color text modes (graphics modes 2 and 3), ATARI chose to change the order of the characters. On any computer, the ASCII value of A is 65. This is true on the ATARI computer. The change occurs internally. When the character set is stored in ROM, the first character is the space. It is followed by the exclamation point, quotation marks, etc. These hold the ATASCII values 32, 33, 34. What has happened is that the characters from 32 - 95 have been shifted up. These are the numbers, symbols, and uppercase letters. The graphic characters which appear to occupy the first 32 positions in the character set are moved to the area just past the uppercase letters and just before the lowercase letters as shown in Table 3-1. Now when you work in graphics modes 1 or 2, you can select either uppercase letters with numbers and symbols, or lowercase letters with graphics.

Each character in the character set uses 8 bytes. To find the set of bytes for a particular character, we multiply its place in the character set by 8 and add the product to 57344. Let's look at the construction of the exclamation point. It is the second character in the character set, but since the first character is in the zero position, the exclamation point's place is 1. 1×8+57344=57352. The data to place an exclamation point on the screen begins in this memory location. Use the following program to PEEK at this information.

Listing 3-1. Data for Exclamation Point

```
10 REM LISTING 3.1
20 REM DATA FOR EXCLAMATION POINT
30 REM BY LINDA M. SCHREIBER FOR TAB B
OOKS
40 CB=57344:REM BEGINNING ADDRESS FOR
CHARACTER SET IN ROM
50 EP=1*8+CB:REM MULTIPLY THE LOCATION
 OF THE CHARACTER BY 8 AND ADD THE BEG
INNING ADDRESS OF CHARACTER SET
60 ? "}CLEAR}":REM CLEAR THE SCREEN
70 FOR X=EP TO EP+7:REM GET THE INFORM
ATION
80 ? PEEK(X):REM PRINT THE INFORMATION
90 NEXT X
```

The following number should appear on your screen:

```
 0
24
24
24
24
 0
24
 0
```

This information determines which pixels will be turned on to form the exclamation point as shown in Fig. 3-1.

Because the character set is located in ROM, it is fixed. This character set cannot be changed. But . . . there is a pointer in RAM that tells the computer where the character set begins (location 756 decimal). By changing the value of this location, we can point to a new character set—one that we have created and stored in RAM. The only limitation we have is that the character set must begin on an even 1K boundary if we are using 1024 bytes or the entire character set. It must begin on an even ½K boundary if we are using the color text modes. The color text modes display only half of the character set at a time, so we do not need the entire character set in RAM. The character set developed and stored in RAM can be placed in any convenient location. Of course, it should not be in the way of the screen display, display list, or BASIC program. The best place for it is just before the display list. This way it is high enough in memory to be out of the way of our program, but it will not interfere with the screen in any way.

RESTRUCTURING THE SET

There are several programs on the market today to help you restructure or edit your character set. What these programs essentially do is move the character set from ROM into RAM, then display the character that you want to edit in a large form on the screen. You turn on or

decimal code	character
32	space
33	!
34	
35	#
.	.
.	.
.	.
.	.
65	A
66	B
67	C
.	.
.	.
.	.
.	.
.	.
0	(cntl ,)
1	(contl A)
2	(cntl B)
.	.
.	.
.	.
.	.
97	a
98	b
99	c

Table 3-1. Position of Characters in Character Set.

off the pixels that make up the character. When you are satisfied with the character, you can store it in the character set. When you are finished creating new characters, you can store the new character set on disk or cassette.

Another way to create a new character set is to move the character set into RAM, design your new characters on graph paper, set the decimal value of each of the eight rows that make up the character, and then poke these values over that character that you intend to replace.

Obviously, it is much easier to create new characters with the aid of a program than it is with paper and pencil. If you are redesigning most of the characters in the set, you would want to use the first method. To redesign a few characters, you could use the pencil method. The following program will allow you to redesign the character set and save the new character set to disk or cassette. This set can then be used by any BASIC program.

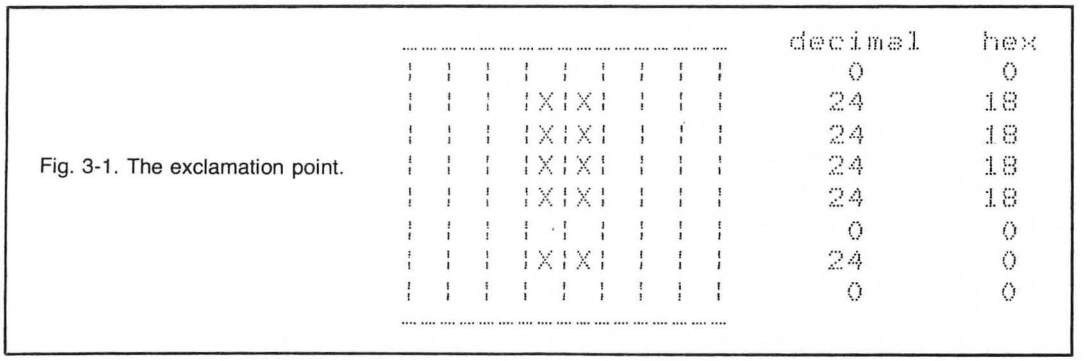


Fig. 3-1. The exclamation point.

Listing 3-2. Character Set Editor

```
10 REM LISTING 3.2
20 REM CHARACTER SET EDITOR
30 REM BY L.M. SCHREIBER FOR TAB BOOKS

40 DIM B$(10),CV(7,7),NAME$(8),NAM$(14
),P1$(20)
50 DATA !, , ! !, , EDIT,LETTER
55 REM DATA !, CTRL-QRE, ! !, CTRL-ZRC
, EDIT,LETTER - CONTROL CHARACTERS FOR
M BOX
60 GRAPHICS 0:POKE 752,1:? "CTRL E -
TO EDIT A CHARACTER":? "CTRL S - TO S
TOP EDIT"
70 ? "CTRL D - SAVE SET TO DISK":? "
CTRL L - LOAD SET FROM DISK":? "CTRL
Q - QUIT"
80 FOR X=16 TO 23:FOR C=6 TO 13:POSITI
ON X,C:? "":NEXT C:NEXT X:REM MAKE AN
 8x8 DISPLAY WITH CTRL-T
90 READ B$:FOR X=9 TO 13:READ B$:POSIT
ION 9,X:? B$:NEXT X
100 A=PEEK(106)-8:NCB=A*256:REM PLACE
NEW CHARACTER SET 2K BEFORE END OF MEM
ORY
110 POKE 204,A:POKE 206,224:REM STORE
THE NEW CHARACTER SET ADDRESS AND THE
ROM ADDRESS
120 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$
(B):NEXT X:REM MACHINE LANGUAGE SUBROU
TINE TO MOVE CHARACTER SET
125 DATA 104,162,4,160,0,177,205,145,2
03,200,208,249,230,206,230,204,202,208
,242,96,
130 Q=USR(ADR(P1$)):REM USE MACHINE LA
NGUAGE PROGRAM WITH THE USR FUNCTION
140 POKE 756,A:REM NOW WE CAN USE THE
SET IN RAM
150 POSITION 3,18:FOR X=0 TO 26:? CHR$
(X);:NEXT X:FOR X=27 TO 31:? CHR$(27);
CHR$(X);:NEXT X
160 POSITION 3,19:FOR X=32 TO 63:? CHR
$(X);:NEXT X
170 POSITION 3,20:FOR X=64 TO 95:? CHR
$(X);:NEXT X
180 POSITION 3,21:FOR X=96 TO 124:? CH
```

```
R$(X);:NEXT X:FOR X=125 TO 127:? CHR$(
27);CHR$(X);:NEXT X
190 OPEN #4,4,0,"K:":REM OPEN KEYBOARD
 FOR A READ
200 POSITION 3,16:? "READY FOR EDIT
            ":E=0:X=16:C=6:REM POSITIO
N OF FIRST CHARACTER EDIT LETTER .
210 GET #4,B:IF B=5 AND E=0 THEN E=1:G
OTO 400:REM EDIT CHARACTER
220 IF B=19 AND E=1 THEN E=0:GOSUB 800
:GOTO 500:REM DONE WITH THAT CHARACTER
230 IF B=4 AND E=0 THEN 1640:REM SAVE
CHARACTER SET
240 IF B=17 AND E=0 THEN POKE 756,224:
POKE 752,0:? "}":END :REM DONE WITH CH
ARACTER SET
250 IF B=12 AND E=0 THEN 1690:REM GET
NEW CHARACTER SET
260 REM USE THE NEXT 5 FOR TO EDIT A C
HARACTER
270 IF E=1 AND B=42 THEN 650:REM GO RI
GHT
280 IF E=1 AND B=43 THEN 670:REM GOT L
EFT
290 IF E=1 AND B=45 THEN 690:REM GO UP

300 IF E=1 AND B=61 THEN 710:REM GO DO
WN
310 IF B=32 AND E=1 THEN 750:REM CHANG
E BIT
320 GOTO 210
400 POSITION 3,16:? "PRESS LETTER TO E
DIT":GET #4,B
410 IF (B>26 AND B<32) OR (B>124 AND B
<128) THEN POSITION 11,10:? CHR$(27);C
HR$(B):GOTO 460
420 IF B>127 THEN 450
440 POSITION 11,10:? CHR$(B):GOTO 460
450 POSITION 5,17:? "CANNOT EDIT THAT
CHARACTER":E=0:FOR TL=1 TO 100:NEXT TL
460 POSITION 3,16:? "
    ":POSITION 5,17:? "
            ":IF E=0 THEN 210
470 AS=B:IF B>31 AND B<96 THEN AS=B-32
```

Listing 3-2. Character Set Editor (continued from page 29)

```
:GOTO 490
480 IF B<32 THEN A$=B+64
490 CP=NCB+A$*8:REM POSITION OF CHARAC
TER IN CHARACTER SET
500 FOR Q=0 TO 7:CV=PEEK(Q+CP):REM GET
 THE VALUES FOR THE CHARACTER
510 CV(0,Q)=0:IF CV>127 THEN CV(0,Q)=1
:CV=CV-128
520 CV(1,Q)=0:IF CV>63 THEN CV(1,Q)=1:
CV=CV-64
530 CV(2,Q)=0:IF CV>31 THEN CV(2,Q)=1:
CV=CV-32
540 CV(3,Q)=0:IF CV>15 THEN CV(3,Q)=1:
CV=CV-16
550 CV(4,Q)=0:IF CV>7 THEN CV(4,Q)=1:C
V=CV-8
560 CV(5,Q)=0:IF CV>3 THEN CV(5,Q)=1:C
V=CV-4
570 CV(6,Q)=0:IF CV>1 THEN CV(6,Q)=1:C
V=CV-2
580 CV(7,Q)=CV
590 NEXT Q
600 X=16:C=6:FOR Q=0 TO 7:FOR Q1=0 TO
7:POSITION X+Q1,C+Q:? "":IF CV(Q1,Q)=
1 THEN POSITION X+Q1,C+Q:? " "
610 NEXT Q1:NEXT Q:LOCATE X,C,CH:POSIT
ION X,C:IF CH<128 THEN ? "*":GOTO 630
620 ? "*"
630 IF E=1 THEN POSITION 3,16:? "USE A
RROW KEYS TO EDIT"
640 GOTO 210
650 POSITION X,C:? CHR$(CH):X=X+1:IF X
>23 THEN X=16
660 GOTO 720
670 POSITION X,C:? CHR$(CH):X=X-1:IF X
<16 THEN X=23
680 GOTO 720
690 POSITION X,C:? CHR$(CH):C=C-1:IF C
<6 THEN C=13
700 GOTO 720
710 POSITION X,C:? CHR$(CH):C=C+1:IF C
>13 THEN C=6
720 LOCATE X,C,CH:POSITION X,C:IF CH>1
27 THEN ? "*":GOTO 740
```

```
730 ? "*"
740 GOTO 210
750 LOCATE X,C,C2:IF C2>128 THEN CH=20
:C2=C2-128:CV(X-16,C-6)=0:GOTO 770
760 CH=160:C2=C2+128:CV(X-16,C-6)=1
770 POSITION X,C:? CHR$(C2):GOTO 210
800 POSITION 3,16:? "EDIT FINISHED
     ":REM CHANGE MESSAGE
810 CV=0:B=128:FOR Q=0 TO 7:FOR Q1=0 T
O 7:REM CONVERT THE 1 & 0'S TO DECIMAL

820 IF CV(Q1,Q)=1 THEN CV=CV+B
830 B=B/2:NEXT Q1:REM REDUCE B FOR EAC
H POSITION
840 POKE CP+Q,CV:REM CHANGE THE BYTE I
N THE CHARACTER SET
850 CV=0:B=128:REM RESET THE VARIABLES
 FOR NEXT BYTE
860 NEXT Q:REM FINISH THE CHARATER
870 CP=NCB:POSITION 11,10:? " ":RETURN
 :REM RESET TO CLEAR THE CHARACTER FRO
M SCREEN
1640 NAM$="                    ":POSITION 3,
16:? "ENTER NAME FOR FILE";:INPUT NAME
$:IF NAME$="" THEN 200
1650 NAM$(1,2)="D:":NAM$(3,10)=NAME$:N
AM$(11,14)=".CHB":REM CODE FILE FOR CH
ARACTER BASE
1660 TRAP 1740:OPEN #2,8,0,NAM$:FOR Q=
NCB TO NCB+1023:CV=PEEK(Q):REM GET THE
 VALUES FOR THE CHARACTER SET
1670 PUT #2,CV:NEXT Q:REM PUT THE VALU
ES ONTO DISK
1680 CLOSE #2:GOTO 200
1690 NAM$="                    ":POSITION 3,
16:? "ENTER NAME FOR FILE";:INPUT NAME
$:IF NAME$="" THEN 200
1700 NAM$(1,2)="D:":NAM$(3)=NAME$:NAM$
(LEN(NAME$)+3)=".CHB":REM CODE FILE FO
R CHARACTER BASE
1710 TRAP 1740:OPEN #2,4,0,NAM$:FOR Q=
NCB TO NCB+1023:GET #2,CV:REM GET THE
VALUES FOR THE CHARACTER SET
1720 POKE Q,CV:NEXT Q:REM PUT THE VALU
```

Listing 3-2. Character Set Editor (continued from page 31)

```
ES ONTO DISK
1730 CLOSE #2:GOTO 200
1740 ER=PEEK(195):CLOSE #2:REM GET THE
 ERROR NUMBER & CLOSE FILE
1750 IF ER=170 THEN POSITION 3,16:? "F

ILE NOT FOUND                    ":GOTO 18
00:REM GIVE ERROR MESSAGE
1760 IF ER=162 THEN POSITION 3,16:? "D
ISK FULL                         ":GOTO 18
00
1770 IF ER=169 THEN POSITION 3,16:? "D
IRECTORY FULL - GET NEW DISK ":GOTO 18
00
1780 POSITION 3,16:? "WE'VE GOT A PROB
LEM                              "
1800 FOR X=1 TO 100:NEXT X:GOTO 200
```

Line 50 is the data needed to draw the box on the screen to show what letter/character is being edited. Be sure that this data line is entered exactly as follows: an exclamation point, a space cntrl-Q cntrl-R cntrl-C, space shift-equals space shift-equals, space cntrl-Z cntrl-R cntrl-Z, space EDIT, LETTER.

Lines 60-70 set the graphics mode to 0, erase the cursor, and print the control codes on the screen.

Lines 80-90 print the grid and box on the screen. Use the cntrl-T to make the 8×8 grid.

Line 100 finds the end of memory and calculates the address that would be 2K before the end of memory. This leaves 1K for the character set and 1K for the screen display and display list.

Line 110 pokes the new character set address and the old character set address into RAM. These two values will be used in the machine language subroutine that moves the character set from ROM into RAM.

Line 120 contains the machine language subroutine to move the character set from ROM into RAM. P1$ must be exact if the routine is to work correctly. The data for the subroutine is in line 125. Be sure that these numbers are entered correctly. If they are not, the program will crash.

Line 130 uses the USR function to call the machine language subroutine. The Q is a dummy variable.

Line 140 changes the address of location 756 from the character set in ROM to the character set in RAM.

Lines 150-180 print the entire character set on the screen. In order to print some codes, the escape key must be entered first. When we are printing characters using the CHR$ command, we can issue an escape code by printing CHR$(27) just before the character. In this way we can display all the characters in the character set.

Line 190 opens the keyboard for reading. When editing the character set, we will use only one key stroke commands.

Line 200 prints the prompt under the square and grid. There are 15 spaces after the T to clear out any previous message. We will be returning to this line after several of the subroutines in this program. The variable E will be our flag to let the computer know whether or not we are editing a character. When it is set to 0, we can use the control codes to begin an edit, load a character set, save a character set, or quit. When the E is set to 1, we can stop an edit or edit a character using the arrow keys. The variable X is the row that the cursor is in on the screen, and C is the column.

Line 210 gets the keystroke from the keyboard and checks it for a control-E. If it is a control-E and we are not in the edit mode, the variable E will change to 1 and the program will go to line 400.

Line 220 checks the variable B for a control-S. If the program is in the edit mode, and the control-S is pressed, the program will leave the edit mode and go to the routines that will restore the grid and erase the character in the box. The variable E is also reset to 0.

Line 230 checks for a control-D. When a control-D is pressed and the program is not in the edit mode, the program will save the character set displayed on the screen. This program stores the character set to disk. It can be changed to store the characters to cassette by opening the cassette instead of the disk. You do not need a name for the file if you are using a cassette.

Line 240 will end the program when a control-Q is pressed. Before ending, the program will restore the ROM character set, restore the cursor, and clear the screen. Whenever an alternate character set is used in a program, it is good programming practice to reset the pointer to the ROM character set. Loading a new program with an alternate character set could confuse the next user.

Line 250 will direct the program to the routine that loads a new character set from disk or cassette when a control-L is pressed.

Lines 270-310 will direct the program to the lines that alter the character set when the variable E is set to 1 and an arrow key or space bar is pressed.

Line 320 will loop back to line 210 if the key pressed is not one of the control keys used in this program.

Line 400 begins the edit mode. The prompt under the grid is changed and the program waits for a key to be pressed.

Lines 410-460 check the key that has been pressed. If it is one of the screen function keys, clear screen, line up or down, etc., the program will not allow it to be edited, and will print a message to that effect on the screen. If it is a character that can be edited, it will be printed in the box on the screen.

Lines 470-480 check the value of B and store the actual location in the character set in variable AS. B is stored in AS before it is checked. There is no else command in ATARI BASIC, so we will store the actual value of B in AS. If it is not a character whose location needs to be changed, AS will be set correctly. If the ATASCII value of the key pressed is greater than 31 and less than 96, that is any key other than a graphics character or lowercase letter, the program will subtract 32 from B and store it in AS. These are the characters that have been moved up in the actual character set. If the value of B is less than 32, a graphics character has been entered and the program adds 64 to the value of B and stores it in AS. Remember that the graphics characters are located between the uppercase letters and the lowercase letters in the actual character set.

Line 490 calculates the position of the first byte of the character in the character set. The position of the character that is stored in AS is multiplied by 8 (each character uses 8 bytes) and this value is added to the location of the character set. The variable CP contains the location of the first byte of the character that will be edited.

Lines 500-590 convert the decimal value of each byte into a binary value. Each bit is stored in the array CV. This routine is similar to the routine used in Chapter 1. It takes the decimal value of the byte and compares it to 127. If the number is greater than 127, then the first bit is a one. The program subtracts 128 from the value in CV. The next line checks the remaining value to see if the next bit should be set. Each line continues to check the value of CV against the value of that bit less one. We use one less than the actual bit value because if that bit were set, and we subtracted the bit value, the remainder would be zero. There would be no indication that we should get that bit unless the decimal number was larger than the bit value. By using the bit value less one, we will get a remainder of one if the decimal value is equal to or larger than that place value. Every time the program sets a bit to one, it subtracts the value of that place or bit from the decimal value of the byte. This routine is repeated 8 times; once for each byte that makes up the character set.

Lines 600-620 reset the row and column values for the 8×8 grid and using the values in the array CV, draws the character onto the grid. A control-T is printed. Then the value of CV is checked for a 1. If that element of the array does contain a 1, an inverse-video cursor will be printed. Once the entire character has been drawn, the program will use the locate command to find out what has been printed in the upper left corner of the grid. The ATASCII value of this character will be stored in the variable CH. The asterisk will be our cursor while editing. If the character in the upper left corner is in inverse-video, an inverse-video asterisk will be printed there.

Line 630 checks E to see if we are, in fact, in the editing mode. If we are, it prints the prompt on the screen. This routine then goes back to line 210.

Lines 650-740 move the asterisk cursor in the grid. The character that is stored in CH is printed in the grid where the asterisk is. If we are moving the asterisk to the right (line 650), the variable X is incremented. If we are moving it to the left (line 670), X is decremented. Moving the asterisk up (line 690) decrements C and moving it down (line 710) increments C. After the variable X or C is changed, the program checks it to make sure that it is not beyond the grid area. If it is, the variable is reset to the other side of the grid, giving it a wrap-around feature. Once X or C are correctly set, the program gets the ATASCII value of the character that the asterisk will be replacing and stores it in CH. Once again, if the character was in normal text, an asterisk will be printed. If the character was an inverse-video cursor, an inverse-video asterisk will be printed.

Lines 750-770 change the character in the grid from a ball (cntrl-T) to an inverse-video cursor and back again. When the space bar is pressed, and the program is in the edit mode, it will be directed to this routine. The locate command gets the ATASCII value of the cursor. If it is in inverse-video, then the value of the character that was there will be changed to 20 (cntrl-T), and the asterisk will be reprinted in normal video. Otherwise, the character that was there will be changed to the inverse-video cursor value, and the asterisk will be printed in inverse-video. When the value of CH is changed to 160, indicating that we are setting that bit, the value in the array CV is changed to 1 for the corresponding bit. When we erase a bit from the screen, the value of the corresponding bit is changed back to 0.

Lines 800-870 are used when we are satisfied with the new character that we have just created. The new prompt is printed on the screen, and the values stored in the array CV are converted into decimal. This procedure is simpler than the convert from decimal to binary routine. The variable CV is cleared. This variable contains the decimal value of the character. The variable B is set to 128 - the bit value of the most significant bit in the byte. If the leftmost bit is set to one, this value will be added to CV. Since each bit is half the value of the preceding bit in a byte, we divide B by 2 each time we check the next element of the array. Each time an element

contains a 1, we add the value of B to CV. After we have checked each of the 8 elements of the array that represent the byte, we will have the decimal value of that byte. That value will be poked into the position of that byte in the character set in RAM. The variables CV and B are reset after each byte. Once the entire character has been moved into the character set, it is removed from the box on the screen and the routine returns to the main part of the program. Since the entire character set is displayed on the screen and it is being used for the prompts as well, it would not be wise to change the uppercase letters or the characters that are being used in the editing modes, especially the control-T, unless absolutely necessary.

Lines 1640-1680 save the character set that we edited to disk. The string variable NAM$ is cleared of the last entry, or garbage that the string contains from the previous program. The name that you want to call this set is placed in NAME$. If you press the return key without entering a name, the program will return to the menu. All good programs have an abort code that returns you to the menu should you enter a routine by mistake. When you enter the name of the character set, you do not have to enter the D: before the name.

Line 1650 takes the name that you enter and adds the D: before the name. It also appends the name with .CHB. This will separate the character sets from any other programs or files on the disk. The program then opens the file and, using the peek command gets every byte of the character set, and puts in on the disk. When the entire set has been stored on disk, the file is closed. There is a trap set before the file is opened. If the disk or the directory is full, it will be reported on the screen.

Lines 1690-1730 get the character sets that we previously stored on the disk. The string variable NAM$ is cleared, and the program asks for the name of the character set that you would like to bring in. If you press the return key without entering a name, the program will return to the main menu. Once again, the program will add the D: to the beginning of the character set name, and .CHB to the end of the name. The trap is set, and the program will bring in the character set. This routine can be used in any program to bring in a character set that is stored on disk. Instead of having the program ask for the name of the character set, you could specify it in the program lines. This way, any character set that is designed with this program can be used with any other program. Once the character set has been read in, the program will close the file and return to the menu.

Lines 1740-1800 trap the disk errors. The error number is stored in decimal location 195. By peeking at this location, we can get the number of the error. If the file was not found, the disk is full, or the directory is full, this message will appear on the screen. If any other error caused the program to go to this routine, We've got a problem will be printed on the screen. Normally, every possible error is tested for, but in this case, the error could be 144, which could be the result of anything from a bad disk to the disk door being left open. In this program, we'll let the user know that something has gone wrong, and then return to the menu.

The character sets created with this program can be used in any program that requires a different character set. It can be used in text mode or the colored text modes. If you have a cassette recorder, change the following lines.

70 change the word disk to cassette
1640 delete the INPUTs
1650 delete
1660 change the OPEN command to OPEN #7,8,0,"C:"

1680 CLOSE #7
1690 delete the INPUT
1700 delete
1710 change the OPEN command to OPEN #7,4,0,"C:"
1730 CLOSE #7

Lines 1740-1800 can be deleted or changed for cassette errors: error 143 and error 138.

When using the new character set in another program, calculate the location of the new character set. This will be the first location that the first byte of the character set will be poked in. Always begin the new character set at least 1K before the display list and the screen display.

THE INVISIBLE MODES

In the last chapter, we looked at all the modes that are available on the ATARI. Some of these modes are available in BASIC, others can only be accessed by changing the display list. Using the character editor in this chapter, we could reconstruct the lowercase letters for ANTIC 3, save them to disk, then read them in for the program. We would not have to move the character set from ROM. We would have our new set on disk.

Two other modes between graphics mode 0 and graphics mode 1 are ANTIC 4 and ANTIC 5. Both of these modes are text modes, but they support multicolored characters. Each character in these modes is eight pixels wide, but the pixels are turned on or off in pairs. The net effect is that the character is four bits wide.

The color of the character is determined by the bit combination of every pair of bits in the byte. Look at Fig. 3-2. The first bit pair is 11. The pixels that would be turned on for this part of the character would be in the color set by color register 3 (peek 711). The second pair of bits, 01, will be the color of color register 1 (peek 709). The third bit combination, 10, will be the color of color register 2 (peek 710). The last bit combination is 00. This is the background color, the color of register 0 (peek 708).

The characters designed using these modes can be one color, or several colors. In the following program, we will design a screen that uses ANTIC mode 4. It gives the illusion of being graphics mode 7 without the memory consumption.

This type of program is called a simulation. It simulates a simple circuit. When the circuit is complete, the light will light; when it is broken, the light will go out. The bottom of the light and the wire is the same character. The top of the battery and the wire is also the same character. By using two different bit patterns in each of these characters, we can create a two-color character. By using three different bit patterns, we could create a three-color character.

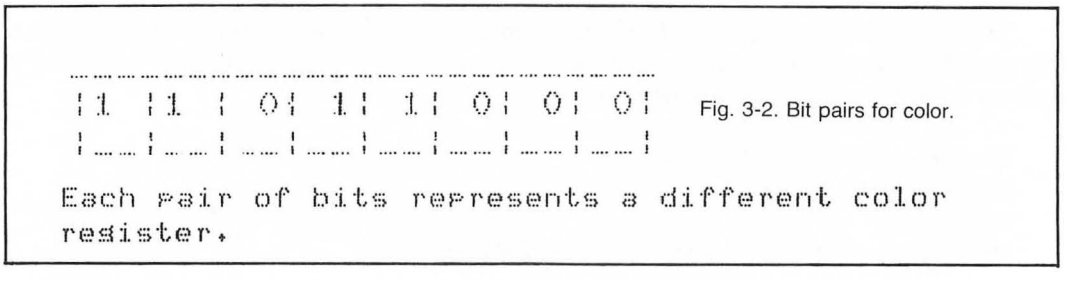


Fig. 3-2. Bit pairs for color.

Listing 3-3. Multicolor Characters

```
10 REM LISTING 3.3
20 REM ANTIC 4 - MULTICOLOR CHARACTERS
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM P1$(20)
50 A=PEEK(106)-8:NCB=A*256:REM PLACE N
EW CHARACTER SET 2K BEFORE END OF MEMO
RY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):? "}CLEAR}":REM USE
 MACHINE LANGUAGE PROGRAM WITH THE USR
 FUNCTION
90 DLIST=PEEK(560)+PEEK(561)*256:REM G
ET THE LOCATION OF THE DISPLAY LIST
100 POKE DLIST+3,68:REM CHANGE THE FIR
ST LINE TO ANTIC 4
110 FOR X=DLIST+6 TO DLIST+28:REM IN G
RAPHICS 0 THE DLIST IS 32 BYTES LONG
120 POKE X,4:REM CHANGE ALL THEN LINES
 TO ANTIC 4
130 NEXT X
140 REM CONTROL A - UPPER LEFT CORNER
150 DATA 85,64,64,64,64,64,64,64
160 REM CONTROL B - TOP LINE
170 DATA 85,0,0,0,0,0,0,0
180 REM CONTROL C - TOP OF BATTERY
190 DATA 81,17,17,17,170,170,170,170
200 REM CONTROL D - UPPER RIGHT CORNER
210 DATA 85,1,1,1,1,1,1,1
220 REM CONTROL E - LOWER RIGHT CORNER
230 DATA 1,1,1,1,1,1,1,85
240 REM CONTROL F - BOTTOM LINE
250 DATA 0,0,0,0,0,0,0,85
260 REM CONTROL G - BOTTOM OF BULB
270 DATA 51,51,51,51,12,12,12,85
280 REM CONTROL H - BOTTOM LEFT CORNER
```

Listing 3-3. Multicolor Characters (continued from page 37)

```
290 DATA 64,64,64,64,64,64,64,85
300 REM CONTROL I - LEVER UP
310 DATA 0,0,0,2,8,32,128,0
320 REM CONTROL J - LEVER PARTWAY DOWN
330 DATA 0,0,0,0,10,32,128,0
340 REM CONTROL K - LEVER NEARLY DOWN
360 DATA 0,0,0,0,0,10,160,0
370 REM CONTROL L - RIGHT SIDE
380 DATA 64,64,64,64,64,64,64,64
390 REM CONTROL M - LEFT SIDE
400 DATA 1,1,1,1,1,1,1,1
410 REM CONTROL N - LEVER DOWN
420 DATA 0,0,0,0,0,0,0,170
430 REM CONTROL O - BATTERY BOTTOM
440 DATA 170,170,170,170,170,170,170,1
70
450 REM CONTROL P - BULB TOP
460 DATA 0,0,0,0,12,51,51,51
470 FOR X=NCB+65*8 TO NCB+81*8-1:REM F
IRST BYTE OF CONTROL A
480 READ B:POKE X,B:NEXT X:REM REDESIG
N CONTROL CHARACTERS
490 POKE 756,A:REM USE THE NEW CHARACT
ER SET
500 POSITION 15,10:? "}ABBBCBBD}":POSI
TION 15,11:? "}L}   }O}  }M}":POSITION
 15,12:? "}L}    }P} }M}"
510 POSITION 15,13:? "}HFIFFGFE}"
520 TL=30:GOSUB 600:TL=5
530 POSITION 17,13:? "}J}":GOSUB 610:R
EM BRING THE SWITCH DOWN
540 POSITION 17,13:? "}K}":GOSUB 610
550 POSITION 17,13:? "}N}":C=PEEK(710)
:POKE 710,15
560 TL=30:GOSUB 600:TL=5
570 POSITION 17,13:? "}K}":POKE 710,C:
GOSUB 610
580 POSITION 17,13:? "}J}":GOSUB 610
590 POSITION 17,13:? "}I}":GOTO 520
600 IF PEEK(53279)<>6 THEN 600
610 POKE 77,0:FOR T=1 TO TL:NEXT T:RET
URN
```

Line 40 sets P1$ for 20 characters. The machine language subroutine to move the character set from ROM into RAM will be placed in this string.

Table 3-2. Machine Language Listing to Move Character Set From ROM to RAM.

Decimal Code	Assembly Language Listing	
104	PLA	;Pull the accumulator off the stack.
162 4	LDX #4	;Load the index X with 4.
160 0	LDY #0	;Load the index Y with 0.
177 205	LDA (205),Y	;Load the accumulator with the contents of the memory location that is arrived at by addins the contents of index Y to the memory location contents in location 205-206.
145 203	STA (203),Y	;Store the number in the accumulator in the address arrived at by addins the contents of the index Y with the contents of locations 203-204.

Table 3-2. Machine Language Listing to Move Character Set From ROM to RAM (continued from page 39).

```
200         INY             ;Increment the index Y.
208         BNE             ;Branch if the index Y is
249                         not 0 backwards 6 bytes.
230         INC 206         ;Add one to the number in
206                         location 206.
230         INC 204         ;Add one to the number in
204                         location 204.
202         DEX             ;Decrement the index X.
208         BNE             ;Branch if the index X is
242                         not 0 backwards 13 bytes.
 96         RTS             ;Return to BASIC.
```

Line 50 subtracts 8 from the number of pages of RAM in your system. This is 2K of memory. 1K is set aside for the screen and display list. The other 1K is for the character set. We will move it just before the display list. The variable NCB will contain the decimal location of the new character base.

Line 60 stores the location of the new character base and the ROM character base in two temporary locations. These locations will be used by the machine language subroutine.

Line 70 contains the machine language subroutine. The data for the machine language subroutine is in line 75. Be sure that all the numbers are entered correctly.

Line 80 uses the USR command to execute the machine language subroutine. See Table 3-2 for the assembly language listing of this routine.

Line 90 calculates the beginning address of the display list by multiplying the value of decimal location 561 by 256 and adding it to 560. The address of the display list is always a 2-byte figure.

Lines 100-130 change the display list from graphics mode 0 (ANTIC 2) to ANTIC 4. The first line of the screen is combined with a command that tells the CTIA where the first memory location of screen data is. To change this line to ANTIC 4, you must add 3 to the first address of the display list and poke this memory location with 68. Since ANTIC 4 uses the same number of lines on the screen as graphics mode 0, we know that there are 23 more locations to change from 2 to 4. By adding 6 to 28 to the address of the display list, we can change the entire screen from graphics mode 0 to ANTIC 4.

Lines 140-460 contain the data to change the standard graphics characters to the ones that

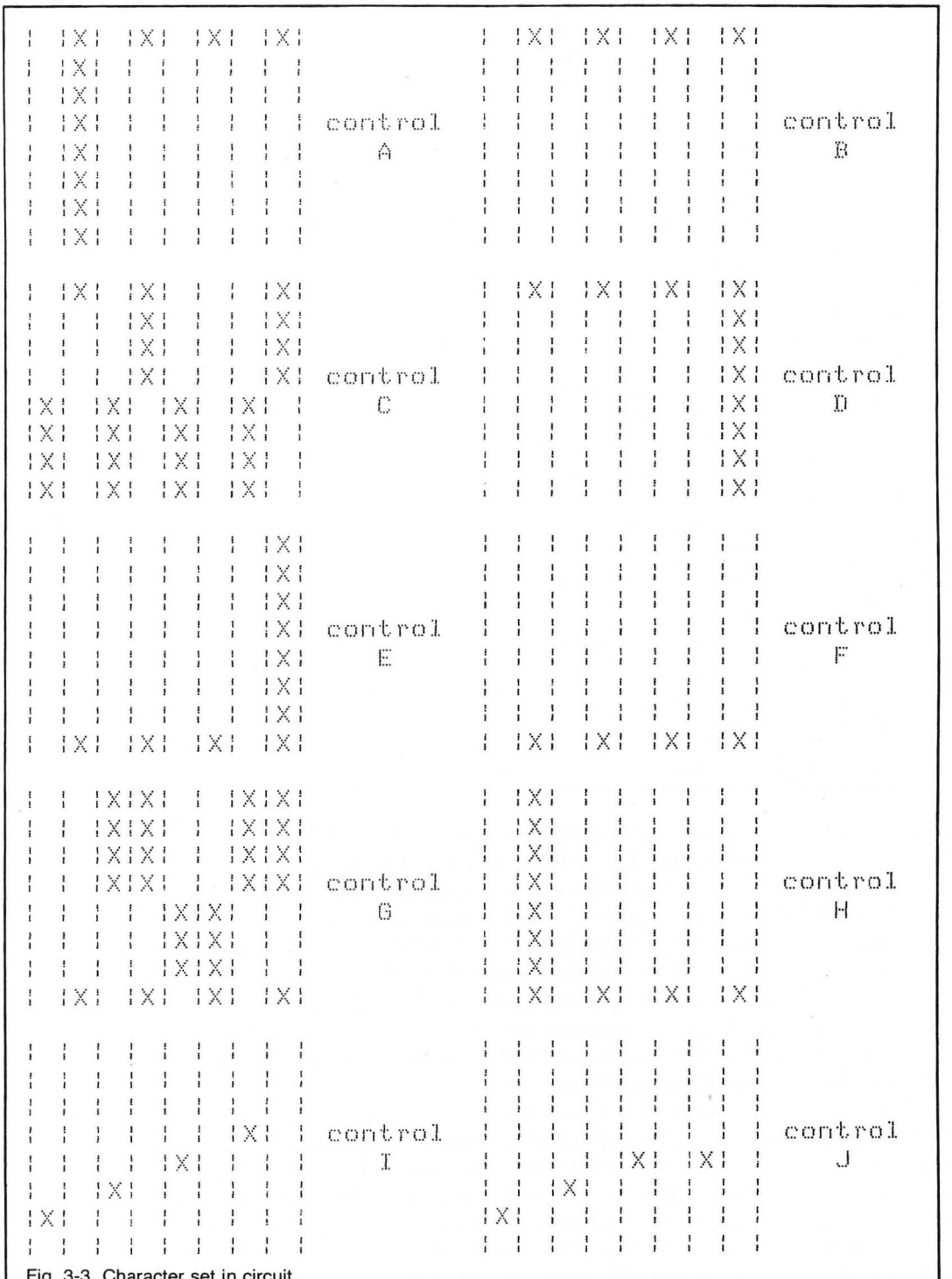


Fig. 3-3. Character set in circuit.

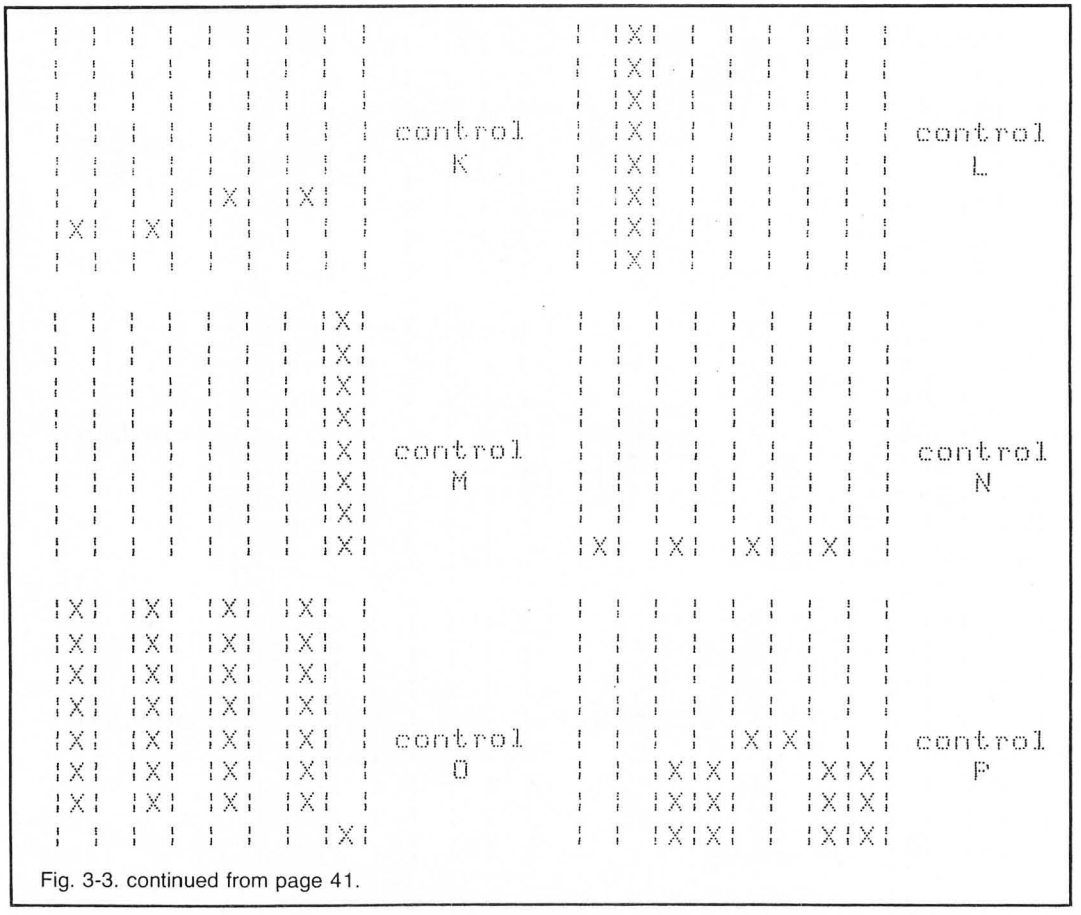


Fig. 3-3. continued from page 41.

will be used in this program. Each character can be one, two, or three colors depending on which bits are set in each byte. See Fig. 3-3 for a detailed description of each character.

Lines 470-480 read the bytes for each character and poke them into the correct locations.

Line 490 changes the character set used from the one in ROM to the modified one in RAM.

Lines 500-510 print the diagram on the screen. Each letter in the quotation marks should be entered as a graphics character. Use the control key to enter these characters into the program.

Lines 520-610 make up the body of the program. The variable TL is set to 30. This value will be used in the timing loop. The program goes to the subroutine in line 600. Here it waits for the start key to be pressed. The program will continue to loop until the start key is pressed. Once it is pressed, the program will reset the attract mode, and enter the timing loop in line 610. This timing loop is needed to smooth the program. Without it, the computer would read that the start key was pressed before you had a chance to take your finger off the key. After the timing loop, the program returns to the line that it came from. The variable TL is reset to 5, and the program will lower or raise the lever. Again, be sure that each letter in the quotation marks is entered as a graphics character. We will use the timing loop after each time the lever is drawn. If the lever is being lowered, the light will glow once the connection has been made. If the lever is being raised, the light will go out as soon as the connection is broken.

ANTIC 5 could use the character set that we created in this program. The characters, however, would be 16 scan lines tall instead of the 8 in ANTIC 4. The display list would be shortened by 12 bytes. The characters drawn with ANTIC 5 would appear to be in graphics mode 5. Again, we can obtain higher resolution graphics without using large amounts of computer memory.

Chapter 4
Principles of Animation

Good use of graphics will enhance any program. But graphics alone are like cake without icing. We all prefer movies to snapshots. Movies, cartoons, television, and most other forms of entertainment rely heavily on movement along with color, sound, and pictures.

In real life, people, animals, and objects move with a particular rhythm. Any movement that differs from the norm appears unnatural and artificial. Artists try to imitate the natural movements of their characters when they create cartoons. Hundreds of drawings make up one feature length film. Each drawing is slightly different than the last, so that when they are run together, the characters move smoothly across the screen.

Using animation in programs is not difficult, but it does take extra planning to create believable characters that move gracefully on the screen.

CHARACTERS WITH A PURPOSE

If we had a computer with block graphics, and no way to redefine the character set, we could make a few stick figures and leave the rest to the user's imagination. But, we don't. We can change the characters to create believable figures and characters. We can alter parts of the character so that when it is printed on the screen, we have true animation.

Good graphics and good animation does not come easily. Each character that is drawn on the screen must be carefully thought out. We have several different graphic modes available to us. Before designing the character set, we must decide what type of program we are designing, how much animation or movement will be involved, and how the screen will be laid out for good color and movement.

Once we have decided on the type of program, we can begin to design the characters. By using graph paper, we can set a good idea of what the character will look like on the screen.

The first program that follows uses a very simple form of animation. As the keys 1-8 are pressed, a note floats up from the corresponding pipe and a tone sounds. The note does not appear all at once. Figure 4-1 shows the parts of the note that appear on the screen. Once the entire note is on the screen, it floats to the top of the screen. Four more characters are used to give the illusion of movement on the screen. When the note reaches the top, it does not disappear from the screen all at once. One row of the note is removed at a time until the entire note is gone. As you can see from the drawings, several characters had to be redefined to give the notes the different

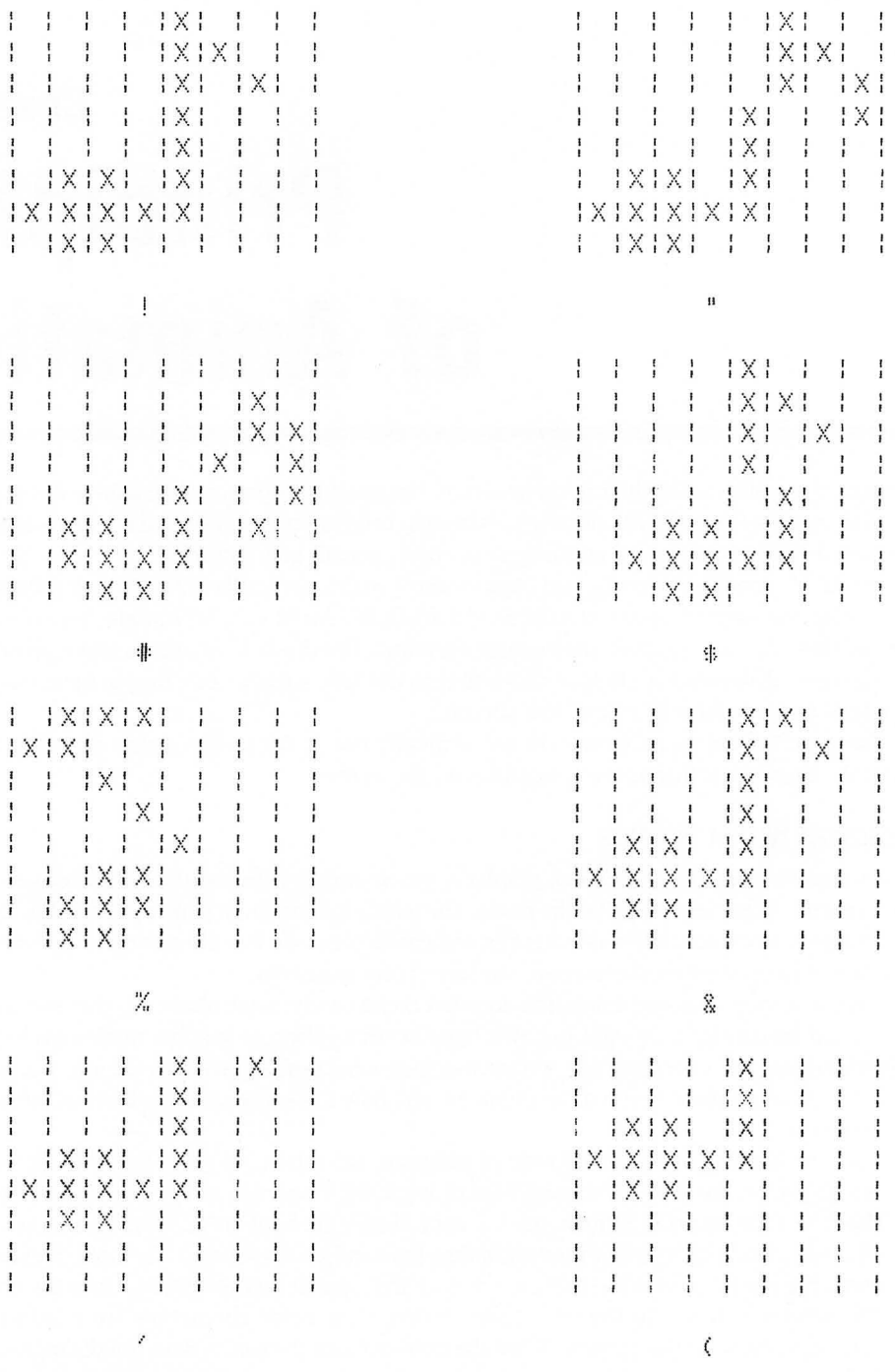
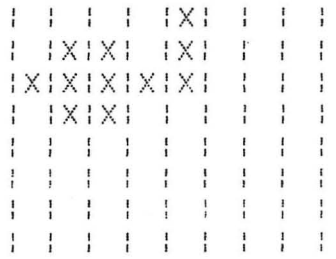
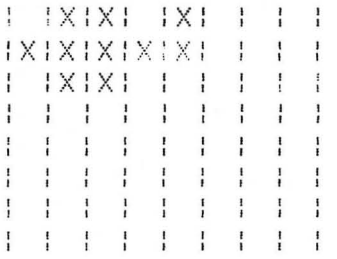


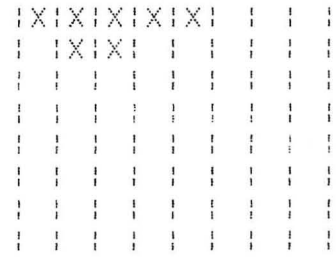
Fig. 4-1. Character set for notes.



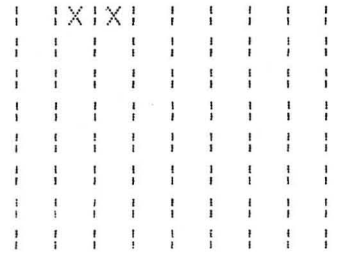
)



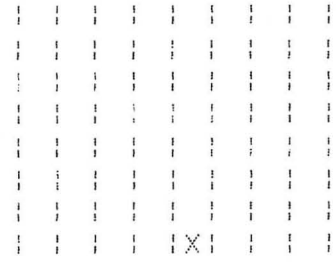
*



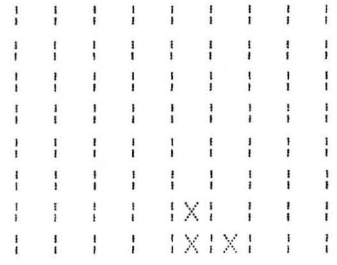
+



,

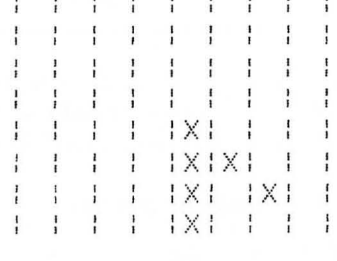


-



.

/



0

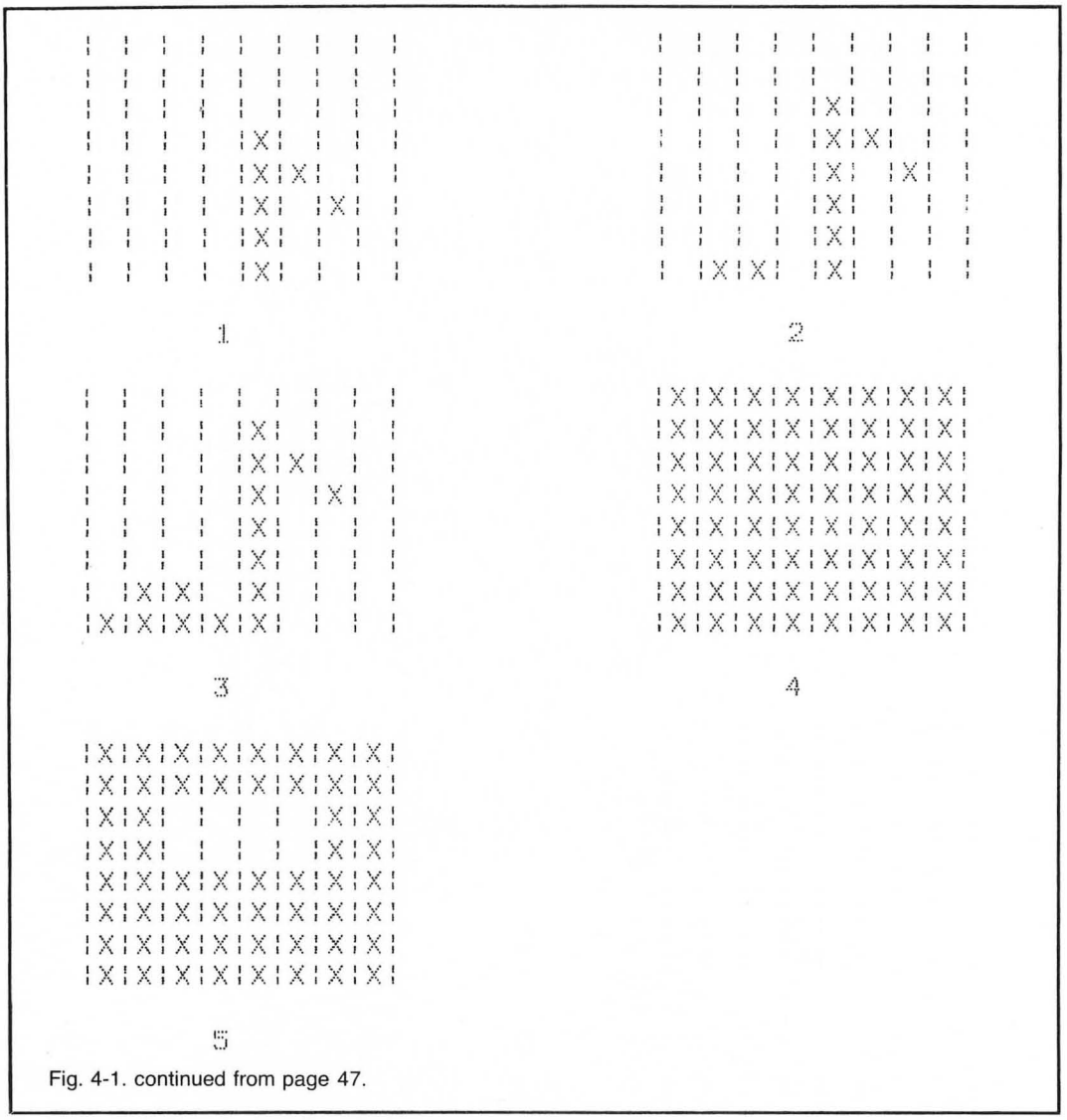


Fig. 4-1. continued from page 47.

forms that they need. Each note is just slightly different than the others. This keeps the movement of the notes smooth as they appear and disappear. If the notes were drastically different, their movement would be choppy and artificial looking.

Listing 4-1. Simple Animation

```
10 REM LISTING 4.1
20 REM SIMPLE ANIMATION
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
```

```
40 DIM NTE$(8),P1$(20)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=8 TO 175:READ C:POKE CB+X,C:N
EXT X
100 DATA 8,12,10,8,8,104,248,96
110 DATA 4,6,5,9,8,104,248,96
120 DATA 0,2,3,5,9,106,124,48
130 DATA 8,12,10,8,4,52,124,48
140 DATA 112,192,32,16,8,48,112,96
150 DATA 12,10,8,8,104,248,96,0
160 DATA 10,8,8,104,248,96,0,0
170 DATA 8,8,104,248,96,0,0,0
180 DATA 8,104,248,96,0,0,0,0
190 DATA 104,248,96,0,0,0,0,0
200 DATA 248,96,0,0,0,0,0,0
210 DATA 96,0,0,0,0,0,0,0
220 DATA 0,0,0,0,0,0,0,8
230 DATA 0,0,0,0,0,0,8,12
240 DATA 0,0,0,0,0,8,12,10
250 DATA 0,0,0,0,8,12,10,8
260 DATA 0,0,0,8,12,10,8,8
270 DATA 0,0,8,12,10,8,8,104
280 DATA 0,8,12,10,8,8,104,248
290 DATA 255,255,255,255,255,255,255,2
55
300 DATA 255,255,195,195,255,255,255,2
55
340 GRAPHICS 17:REM USE LARGE COLOR TE
XT MODE
350 POKE 756,A:REM NOW WE CAN USE THE
SET IN RAM
360 R2=9:FOR R=10 TO 20:FOR X=2 TO 16
```

Listing 4-1. Simple Animation (continued from page 49)

```
STEP 2:POSITION X,R:REM MAKE THE PIPES
370 IF R-X<>R2 THEN ? #6;"4":GOTO 390:
REM MAKE THE PIPE SOLID
380 ? #6;"5":R2=R2-1:REM MAKE THE AIR
HOLE
390 NEXT X:NEXT R
400 NTE$="y1`[QH@<":REM CHARACTER OF T
HE NOTES-y1 diamond left-bracket QH@<
410 OPEN #4,4,0,"K:":REM OPEN THE KEYB
OARD FOR A READ
420 POKE 764,255:GET #4,C:IF C>127 THE
N C=C-128:POKE 694,0:REM GET THE KEY P
RESSED
430 IF C<49 OR C>56 THEN 420:REM NOT A
 NUMBER
440 CLOSE #4:REM GOT THE NOTE
450 C=C-48:REM PLACE OF NOTE
460 X=C*2:R=9:REM COLUMN FOR THE NOTE
ON THE SCREEN
470 SOUND 0,ASC(NTE$(C,C)),10,10:REM S
OUND OF THE NOTE
480 FOR R1=1 TO 7:POSITION X,R:? #6;CH
R$(44+R1):REM PRINT PART OF THE NOTE
490 GOSUB 800:REM TIMER ROUTINE
500 NEXT R1:REM GET THE WHOLE NOTE OUT
510 POSITION X,R:? #6;"!":GOSUB 800:X2
=X:R2=R:R1=R:REM LAST POSITION OF NOTE
520 FOR R=R1 TO 0 STEP -1:REM MOVE UP
THE SCREEN
530 X1=INT(RND(0)*2):X1=X+X1:REM USE O
NE OF TWO POSITIONS
540 N=INT(RND(0)*5):N=33+N:REM GET A N
OTE DISPLAY
550 POSITION X1,R:? #6;CHR$(N):REM PRI
NT THE NOTE
560 POSITION X2,R2:? #6;" ":X2=X1:R2=R
:GOSUB 800:REM ERASE THE LAST NOTE & R
EMEMBER THIS ONE
570 NEXT R:REM ALL THE WAY UP THE SCRE
EN
580 POSITION X1,0:? #6;"!":REM RIGHT T
HE NOTE
590 FOR R=0 TO 6:POSITION X1,0:? #6;CH
```

```
R$(38+R):GOSUB 800:REM MAKE THE NOTE D
ISAPPEAR
600 NEXT R
610 POSITION X1,0:? #6;" ":REM ERASE I
T
620 SOUND 0,0,0,0:REM TURN OFF THE SOU
ND
630 GOTO 410
640 END
800 FOR T=1 TO 10:NEXT T:RETURN
```

Line 40 dimensions two strings. NTE$ will contain the characters whose ATASCII values represent the tones C-C. P1$ will contain the machine language subroutine to move the character set from ROM to RAM.

Line 50 subtracts 2K from the amount of RAM in the computer. The decimal address of the first location of the character set in RAM is stored in CB.

Line 60 pokes the high order address of the memory location of the RAM character set into 204 and the location of the ROM character set in 206. These addresses will be used in the machine language subroutine that moves the character set from ROM into RAM.

Line 70 places the machine language subroutine into P1$. The data is read from line 75 and placed in P1$. Be sure that the numbers in the data line are correct.

Line 80 calls the machine language subroutine.

Line 90 reads the data from lines 100-300 and replaces the characters in the character set from the exclamation point to number five. The variable X is first set to 8. The first 8 locations in the character set (0-7) contain the data for the space. If the information is changed, the screen will not be clear. By adding the value of X to CB, we will change the next 21 characters after the space.

Line 340 begins the program. The graphics mode is set to 17—large color print with no text window.

Line 350 pokes 756 with the value of A. The variable A contains the high order address of the RAM based character set. By poking this location, you can change the character set.

Lines 360-390 draw the pipes on the screen. To place the air hole in the correct position in each pipe, R2 is set to 9. When the difference between the pipe's row and its column is equal to R2, the air hole will be drawn instead of the solid pipe. After the air hole is drawn, R2 is decreased by 1. Each air hole will therefore be slightly lower than the previous one.

Line 400 places one character for each note into NTE$. Be sure that this line is entered correctly, or you will hear some strange tones. It is—y1 control period left bracket QH@<

Line 410 opens the keyboard with the read command.

Line 420 clears location 764. By setting it to 255, you guarantee that the next key pressed will be the tone that you want. If you didn't clear it, the last key pressed would be stored in that location. It could be a key that you pressed by accident. Now the program will wait until a key is pressed. The ATASCII value of the key will be stored in the variable C. If the value of C is greater than 127, it means that the inverse key was accidentally pressed. The program subtracts 128 from the value of C to get the correct value of the key pressed. By poking location 694 with a 0, the inverse flag is reset. The next key pressed will have a value less than 128.

Line 430 checks the value of C to make sure that it is a number between 1 and 8. If it is not, the program goes back to line 420 to wait for another key value.

Line 440 closes the keyboard.

Line 450 calculates the key that was pressed. Since the ATASCII value of 1 is 49, we can set the correct value of the number keys by subtracting 48 from C.

Line 460 calculates the column of the pipe that the note will appear above. The pipes are all 2 columns apart. The value of C is multiplied by 2. R is the row that the note will be printed in.

Line 470 turns on the sound. We use the ASCII of the character located in position C of NTE$. Remember, C is the key pressed. If a 2 was pressed, the tone would be the ASCII value of a lowercase 1. This value corresponds to the tone of D.

Lines 480-500 print the note above the correct pipe. Each time the note is printed, it is the next note in the sequence of characters that make up the notes. First the tip of the note shows, then a little of the stem, etc., until the entire note appears on the screen. The characters used for the notes come one after another in the character set. By adding the value of R1 to 44 (the character just before the first note), we can print the entire sequence of notes easily.

Line 510 prints the entire note above the correct pipe. The position of this note is stored in the variables X2, R2, and R1. These locations will be used in the next routine.

Lines 520-570 give the illusion that the note is floating to the top of the screen. Each note is printed either in the same column as the pipe, or one column to the right of the pipe. Line 530 chooses either a 0 or a 1. Add this value to the value of the pipe column (X) in order to calculate the column for the next image of the note. Line 540 chooses one of the five variations that the note can have. The value of N is added to 33 to arrive at the note that will be printed. Line 550 prints the chosen note in the correct column and row. The last note is erased, and the values of the new note are stored in X2 and R2.

Line 580 prints the note one last time before it begins to disappear off the top of the screen.

Lines 590-600 erase the note slowly from the screen. This time the characters that make the note disappear one row at a time will be used. These characters are also placed sequentially in the character set. By adding the value of R to 38, each character for the note will be printed.

Line 610 erases the last row of the note.

Line 620 turns off the sound.

Line 630 sends the program back to 410 to open the keyboard and wait for another key to be pressed.

Line 800 is the timing routine. The program uses it everytime a note is printed on the screen. Without it, the characters would be printed too fast on the screen.

In the next two programs, the character set for an airplane is redesigned. The plane will travel from right to left across the screen. Listing 4-2 uses graphics mode 2 without the text window. Listing 4-3 uses the text mode. In both programs, the plane shows some hesitation. The movement is not as smooth as in the previous program. This is because more than one character is moving on the screen at the same time. In the previous program, only one character was moving at any particular time. BASIC was fast enough to erase one character and replace it with a new one. In these programs, the five characters that create the plane are moving at the same time. There are three characters that must be erased while the plane is being drawn in the new position. BASIC is too slow to erase and draw this many characters at one time.

Listing 4-2. Simple Animation—Second Method

```
10 REM LISTING 4.2
20 REM SIMPLE ANIMATION
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM PLN$(4),PN1$(3),PN2$(3),P1$(20)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=8 TO 63:READ C:POKE CB+X,C:NE
XT X
100 DATA 0,8,8,11,7,11,8,8
110 DATA 0,0,8,11,7,11,8,0
120 DATA 0,0,0,11,7,11,0,0
130 DATA 0,0,0,0,12,30,30,30
140 DATA 30,30,30,255,255,255,30,30
150 DATA 30,30,30,30,12,0,0,0
160 DATA 0,6,7,254,255,254,7,6
170 GRAPHICS 17:REM LARGE COLOR TEXT
180 POKE 756,A
190 PLN$="!%' ":PN1$=" $ ":PN2$=" & "
200 FOR X=15 TO 0 STEP -1:REM FROM RIG
HT TO LEFT ACROSS THE SCREEN
210 IF X/4=INT(X/4) THEN PLN$(1,1)="!"
:GOTO 230:REM USE LONGEST PROP ONCE OU
T OF 4
220 PLN$(1,1)=CHR$(34):IF X/4-INT(X/4)
=0.5 THEN PLN$(1,1)="#":REM USE SHORTE
ST PROP ONCE OUT OF 4
230 POSITION X,10:? #6;PN1$:POSITION X
,11:? #6;PLN$:POSITION X,12:? #6;PN2$
240 FOR T=1 TO 25:NEXT T
250 NEXT X:REM GO ALL THE WAY ACROSS
260 GOTO 170
```

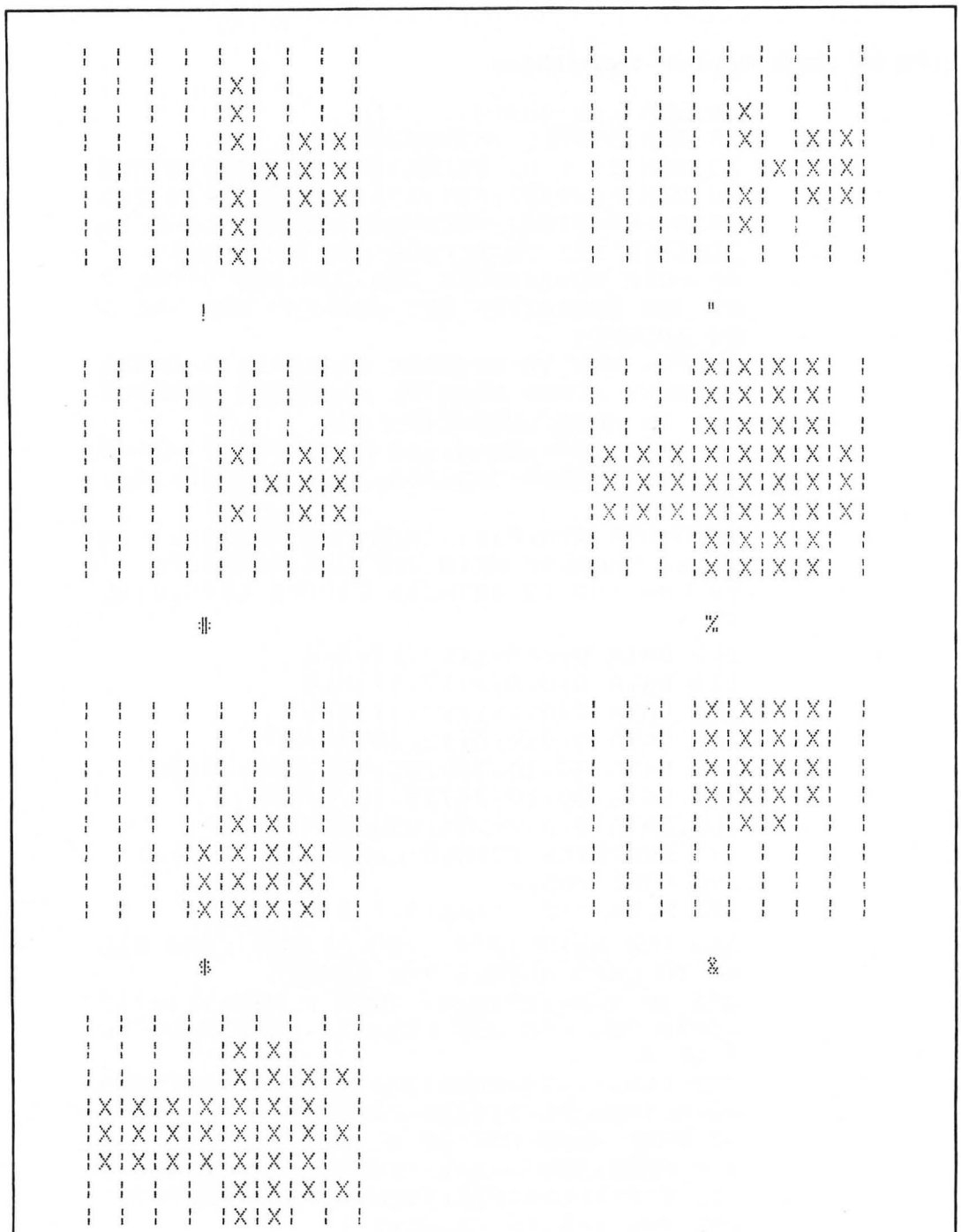


Fig. 4-2. Character set for airplane.

Line 40 dimensions 4 strings. P1$ will be used for the machine language subroutine that moves the character set from ROM into RAM. The other three strings will store the characters for the airplane.

Line 50 subtracts 2K from the amount of RAM in the system. The variable CB will hold the decimal address of the first memory location of the new character set.

Line 60 pokes the address of the new character set into location 204 and the address of the ROM character set into location 206. These two locations are used by the machine language subroutine.

Line 70 places the machine language subroutine in P1$. Be sure that the data line is entered correctly.

Line 80 sends the program to the machine language subroutine. When the program returns to BASIC, the ROM character set will be in RAM.

Line 90 reads the data in lines 100-160 to change seven characters in the RAM character set. See Fig. 4-2 for the characters that will be replaced.

Line 170 changes the graphics mode to 17—large color letters with no text window.

Line 180 tells the computer to use the new RAM character set.

Line 190 sets each string so that it will form a particular part of the airplane. The plane uses three rows on the screen. The main body of the plane is stored in PLN$. This is the propeller, the body of the plane and the tail. PN1$ is the top wing; PN2$ the bottom. There is one space after the characters in each string. This space is needed to erase the character that was drawn in that position previously.

Lines 200-250 move the airplane across the screen. The variable X is the column that the plane will be drawn in. By starting with 15 and counting backwards to 0 (step −1), we will be moving the plane from right to left. Lines 210-220 will determine which propeller to use. One out of every four positions will use the longest propeller. When the variable X divided by 4 is equal to the integer of X divided by 4, the computer will use the longest propeller, which is a redefined exclamation point. If this propeller is used, the program will be directed to line 230. In line 220, the middle size propeller is placed into the string. It is the redefined quotation mark. Since BASIC will not allow a quotation mark within quotation marks, the ATASCII value must be used for this character. This propeller size will be used twice in every spin. Now the variable X is checked to see the shortest propeller. The longest propeller is used when X can be evenly divided by 4. The shortest propeller is used when the remainder of X divided by 4 is .5, or half-way between numbers. When the remainder is .5, the propeller will be changed to the character that replaces the pound sign. Otherwise, the propeller is already set to the correct length. Line 230 prints the plane in the three rows on the screen. The column is set by the value of X. Line 240 is a short timing loop. Without it, the plane would move too fast across the screen.

Line 260 sends the program back to line 170 to repeat it again and again. To stop this program, press the system reset key.

Listing 4-3. Animation in the Text Mode

```
10 REM LISTING 4.3
20 REM ANIMATION IN TEXT MODE
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM PLN$(20),PN1$(3),PN2$(3),P1$(20
)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
```

Listing 4-3. Animation in the Text Mode (continued from page 55)

```
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=8 TO 63:READ C:POKE CB+X,C:NE
XT X
100 DATA 0,8,8,11,7,11,8,8
110 DATA 0,0,8,11,7,11,8,0
120 DATA 0,0,0,11,7,11,0,0
130 DATA 0,0,0,0,12,30,30,30
140 DATA 30,30,30,255,255,255,30,30
150 DATA 30,30,30,30,12,0,0,0
160 DATA 0,6,7,254,255,254,7,6
170 ? "}":POKE 752,1:REM ERASE CURSOR
180 POKE 756,A
190 PLN$="!%$ &  '  "
200 FOR X=35 TO 1 STEP -1:REM FROM RIG
HT TO LEFT ACROSS THE SCREEN
210 IF X/4=INT(X/4) THEN PLN$(1,1)="!"
:GOTO 230:REM USE LONGEST PROP ONCE OU
T OF 4
220 PLN$(1,1)=CHR$(34):IF X/4-INT(X/4)
=0.5 THEN PLN$(1,1)="#":REM USE SHORTE
ST PROP ONCE OUT OF 4
230 POSITION X,10:? PLN$;:REM USING A
SEMICOLON REDUCES FLICKERING
240 FOR T=1 TO 20:NEXT T
250 NEXT X:REM GO ALL THE WAY ACROSS
260 ? "}":GOTO 200
```

This program is essentially the same as the previous one. It is, however, done in the text mode. The two lines that are different are 40 and 190.

Line 40 dimensions two strings. This time PLN$ is dimensioned to 20 to accommodate the entire plane.

Line 190 stores the entire plane in PLN$. This string should read—exclamation point, percent sign, escape up-arrow, escape back-arrow, dollar sign, space, escape down-arrow, escape down-arrow, escape back-arrow, escape back-arrow, and sign, space, escape up-arrow, escape

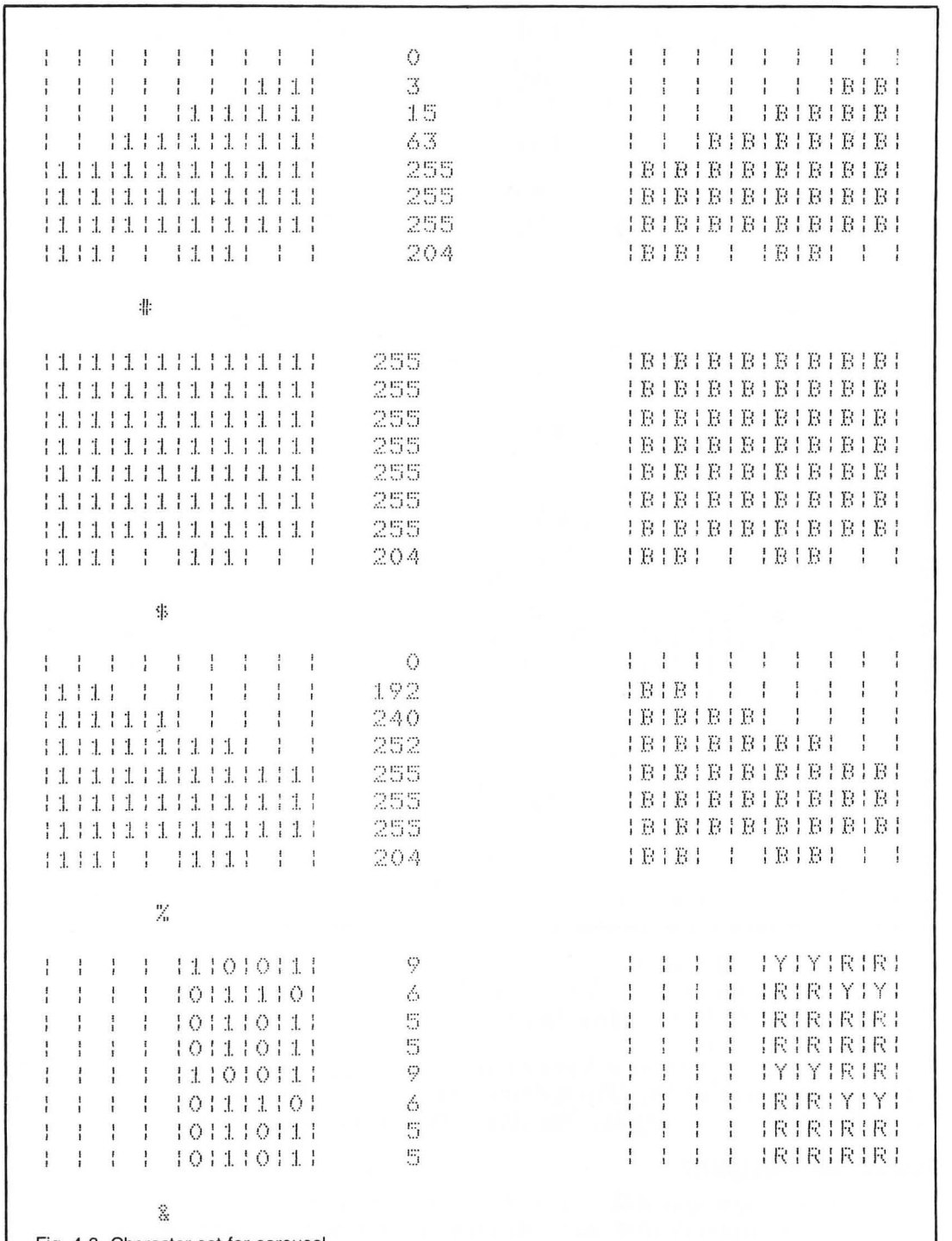


Fig. 4-3. Character set for carousel.

```
!1!0!0!1!  !  !  !  !    144          !Y!Y!R!R!  !  !  !  !
!0!1!1!0!  !  !  !  !     96          !R!R!Y!Y!  !  !  !  !
!0!1!1!0!1!  !  !  !     80          !R!R!R!R!  !  !  !  !
!0!1!0!1!  !  !  !  !     80          !R!R!R!R!  !  !  !  !
!1!0!0!1!  !  !  !  !    144          !Y!Y!R!R!  !  !  !  !
!0!1!1!0!  !  !  !  !     96          !R!R!Y!Y!  !  !  !  !
!0!1!0!1!  !  !  !  !     80          !R!R!R!R!  !  !  !  !
!0!1!0!1!  !  !  !  !     80          !R!R!R!R!  !  !  !  !

        ,

!1!0!0!1!  !  !  !  !    144          !Y!Y!R!R!  !  !  !  !
!0!1!1!0!  !  !  !  !     96          !R!R!Y!Y!  !  !  !  !
!0!1!0!1!  !  !  !  !     80          !R!R!R!R!  !  !  !  !
!0!1!0!1!  !  !  !  !     80          !R!R!R!R!  !  !  !  !
!1!0!0!1!  !  !  !  !    144          !Y!Y!R!R!  !  !  !  !
!0!1!1!0!  !  !  !  !     96          !R!R!Y!Y!  !  !  !  !
!1!1!1!1!1!1!1!1!1!1!    255          !B!B!B!B!B!B!B!B!B!B!
!1!1!1!1!1!1!1!1!1!1!    255          !B!B!B!B!B!B!B!B!B!B!

        (

!  !  !  !  !1!0!0!1!      9          !  !  !  !  !Y!Y!R!R!
!  !  !  !  !0!1!1!0!      6          !  !  !  !  !R!R!Y!Y!
!  !  !  !  !0!1!0!1!      5          !  !  !  !  !R!R!R!R!
!  !  !  !  !0!1!0!1!      5          !  !  !  !  !R!R!R!R!
!  !  !  !  !1!0!0!1!      9          !  !  !  !  !Y!Y!R!R!
!  !  !  !  !0!1!1!0!      6          !  !  !  !  !R!R!Y!Y!
!1!1!1!1!1!1!1!1!1!1!    255          !B!B!B!B!B!B!B!B!B!B!
!1!1!1!1!1!1!1!1!1!1!    255          !B!B!B!B!B!B!B!B!B!B!

        )
```

Fig. 4-3. continued from page 57.

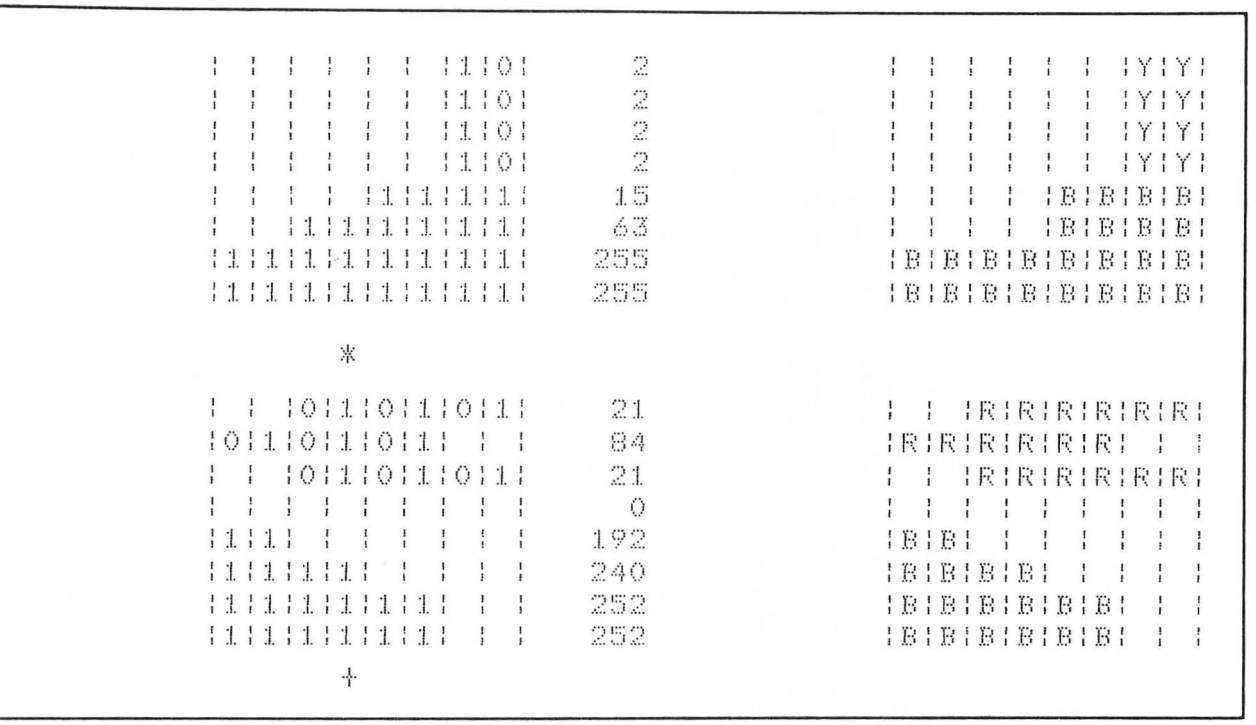
back-arrow, apostrophe, space.

The new spaces must follow the upper and lower wings and the tail to erase the characters that were previously drawn in those locations.

Both programs display some animation. If the for . . . next loops are left in the program, but the position of the plane in line 230 is made constant, you can see the propeller spin on the plane. It is very difficult to see it spin when the plane is moving across the screen.

SCENES AND MOVEMENT

Up to now, the programs that we created made very limited use of the character sets. The characters were simply created and the animation or movement was there, but there was very little in the way of a scene.



Scenes need two parts, the background or picture, and the characters that will move. The ATARI uses the term background to indicate the color of the screen. This color will show through any character, letter, or number that has any bits that are not turned on. If you print the letter O on the screen in graphics 2, the O would appear orange. The center of the O would have the background color showing through. Try this in the direct mode:

```
GR. 2:POSITION 2,2:? #6;"O"
```

The background color in all modes except the text mode is stored in decimal location 712.

Playfield characters are either the characters that can be printed on the screen in graphics modes 0-2, or the lines that are drawn on the screen with the plot and drawto commands in the other graphics modes. In the last three programs, the playfield characters were notes, pipes, and pieces of an airplane.

The playfield characters can be numbers, letters or graphic characters. The next program will draw a carousel on the screen using redefined characters. ANTIC mode 5, which will produce large multicolored characters, will be used in this program.

In ANTIC modes 4 and 5, characters are colored by setting one or two bits in every two bit set to indicate the desired color. Both pixels are turned on to that color. The characters appear to be 4×8. In contrast, in the text mode or in graphics modes 1 and 2, each bit that was set in the character turned on the corresponding pixel. An 8×8 grid contained one character.

In Fig. 4-3, the characters that replace the ROM characters are drawn. Instead of an X in the location of a pixel that is turned on, a 1 or a 0 is in that bit position. Next to it is the decimal code that will be used in the program. Next to that is a drawing of how the character will look on the screen. The B pixels will appear blue on the screen, the R will be red and the Y yellow.

Listing 4-4. Carousel

```
10 REM LISTING 4.4
20 REM CAROUSEL
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM P1$(20)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=24 TO 95:READ C:POKE CB+X,C:N
EXT X
100 DATA 0,3,15,63,255,255,255,204
110 DATA 255,255,255,255,255,255,255,2
04
120 DATA 0,192,240,252,255,255,255,204
130 DATA 9,6,5,5,9,6,5,5
140 DATA 144,96,80,80,144,96,80,80
150 DATA 144,96,80,80,144,96,255,255
160 DATA 9,6,5,5,9,6,255,255
180 DATA 2,2,2,2,15,63,255,255
190 DATA 21,84,21,0,192,240,252,252
200 GRAPHICS 17
210 DLIST=PEEK(560)+PEEK(561)*256:REM
FIND THE BEGINNING OF THE DISPLAY LIST
220 POKE DLIST+3,69
230 FOR X=6 TO 28:POKE DLIST+X,5:NEXT
X:REM CHANGE ENTIRE DISPLAY LIST TO AN
TIC 5
240 POKE 756,A
250 POSITION 17,2:? #6;"*+":REM FLAG O
N TOP
260 POSITION 7,4:? #6;"#$$$$$$$$$$$$$$
$$$$$$%"
270 FOR X=6 TO 18 STEP 2:POSITION 7,X:
? #6;" &      &      '      '":NEXT X:RE
M PRINT THE POLES
```

```
280 POSITION 7,X:? #6;"}N}}}NNNNN}}}NN
NNNN}{}NNNNN}{}N}"
290 GOTO 290
```

Lines 40-190 are the same as in the last program. In this program only one string is dimensioned. Otherwise, the machine language subroutine to move the character set from ROM to RAM is the same. The data lines replace the characters from the pound sign to the plus sign. These characters will draw the carousel on the screen.

Line 210 calculates the beginning of the display list. As mentioned before, this program will use ANTIC 5, which will produce multicolored characters.

Line 220 changes the first row on the screen from graphics mode 0 to ANTIC 5.

Line 230 changes the rest of the display list from graphics mode 0 to ANTIC 5.

Line 240 tells the computer to use the RAM character set.

Lines 250-280 print the new characters on the screen. The *+ is the flag on the top. The roof has 20 dollar signs ($) between the pound sign (#) and the percent sign (%). Line 270 draws the poles. There is one space, the and sign (&), five spaces, the and sign (&), six spaces, the apostrophe ('), five spaces, and another apostrophe ('). The last line prints the bottom. It uses the same spacing as the poles, CTRL N, close parentheses {)}, five CTRL Ns, close parenthesis {)}, six CTRL Ns, open parenthesis {(}, five more CTRL Ns, one last open parenthesis {(}, and a final CTRL N.

Line 290 loops back to itself.

To add animation to this scene, we will use the *player/missile graphics. Players* and *missiles* are terms used by ATARI for special characters that can be created and stored in memory. They are unlike the character sets because each player is only 8 bits or 1 byte wide. The player is, however, as tall as the display screen—using 255 bytes in the single line resolution or 128 bytes in double line resolution. Each player can be thought of as a band that extends from the top of the screen to the bottom. The character is drawn on this band. It can be moved from side to side, and up or down.

Because the players are stored in an area of memory other than the memory used for the screen display, their image seems to be superimposed onto the background and playfield characters.

In this program, we will create a horse for the carousel. To make it look realistic, we will use two players side-by-side. The horse can move up and down while it is going around on the carousel. Figure 4-4 shows how the horse is created.

Figure 4-5 shows the amount of memory needed for player/missile graphics. There are two different modes for player/missile graphics: single resolution and double resolution. In the single resolution mode, each byte is one row or pixel high on the screen. Each player and missile has 256 bytes of memory set aside for it. The area of memory set aside for the players and missiles must begin on an even 2K boundary. This means that the first byte of the memory must be evenly divisible by 2048. An easy way to find the boundary is to subtract 4 from the end of memory for every 1K. Memory location 106 contains the amount of memory available in the computer. In a 40K system, the amount stored in this memory location is 160. Multiply 160 by 256 and you get 40960—the amount of actual RAM available. Each time you subtract 1 from the number, you are

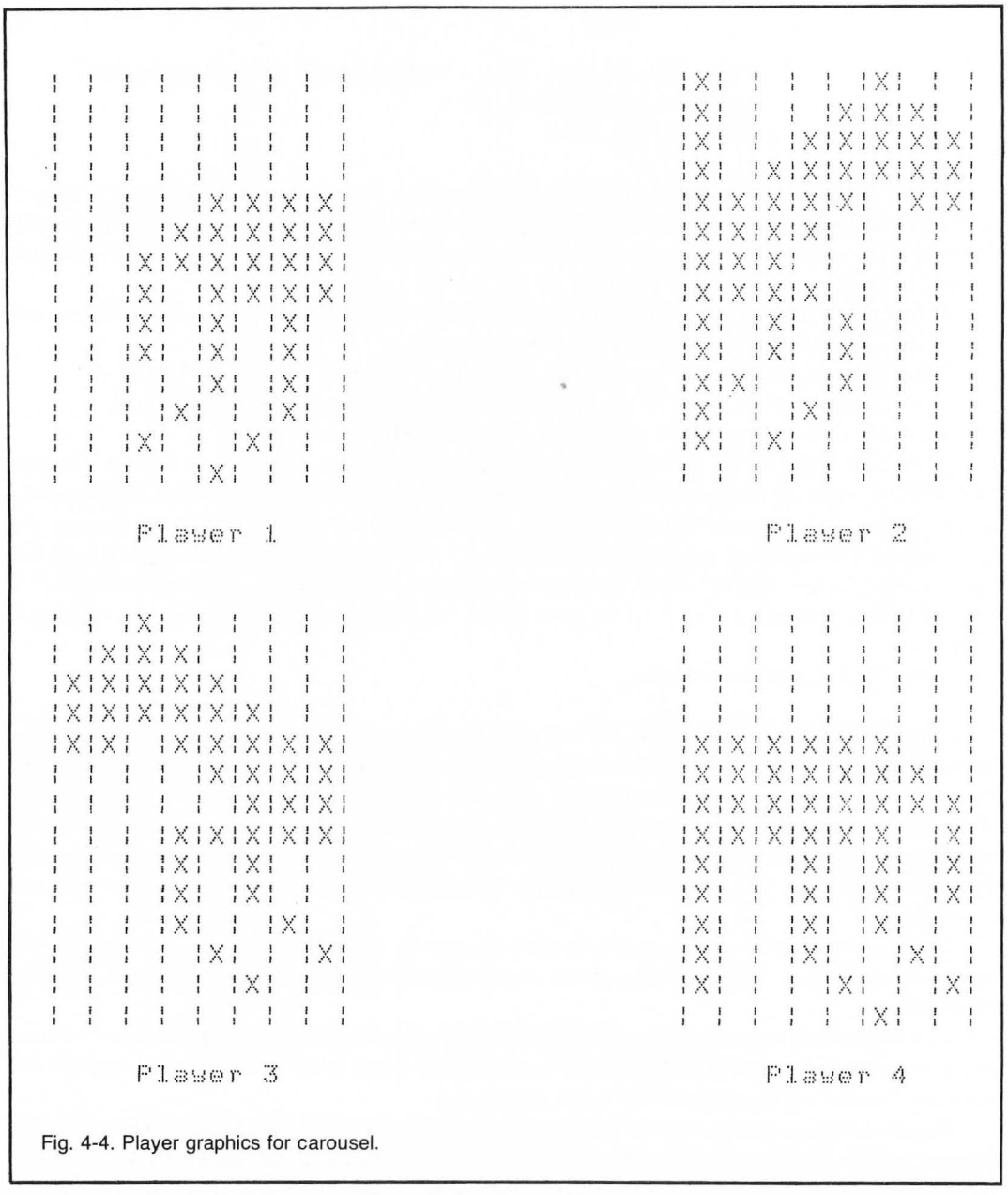


Fig. 4-4. Player graphics for carousel.

actually subtracting 256 bytes. By subtracting 4, you will subtract 1K from the amount of available RAM. Since the operating system uses 1K of RAM for the screen display and display list in mode 0, you cannot use the last 1K of RAM for the player/missile graphics. If you subtracted 8 from the amount in memory location 106, you would have an even 2K boundary, but you would be using the same memory that the screen and display list were using. When you are using single line resolution for our player/missile graphics, we subtract 16 from the amount of RAM available.

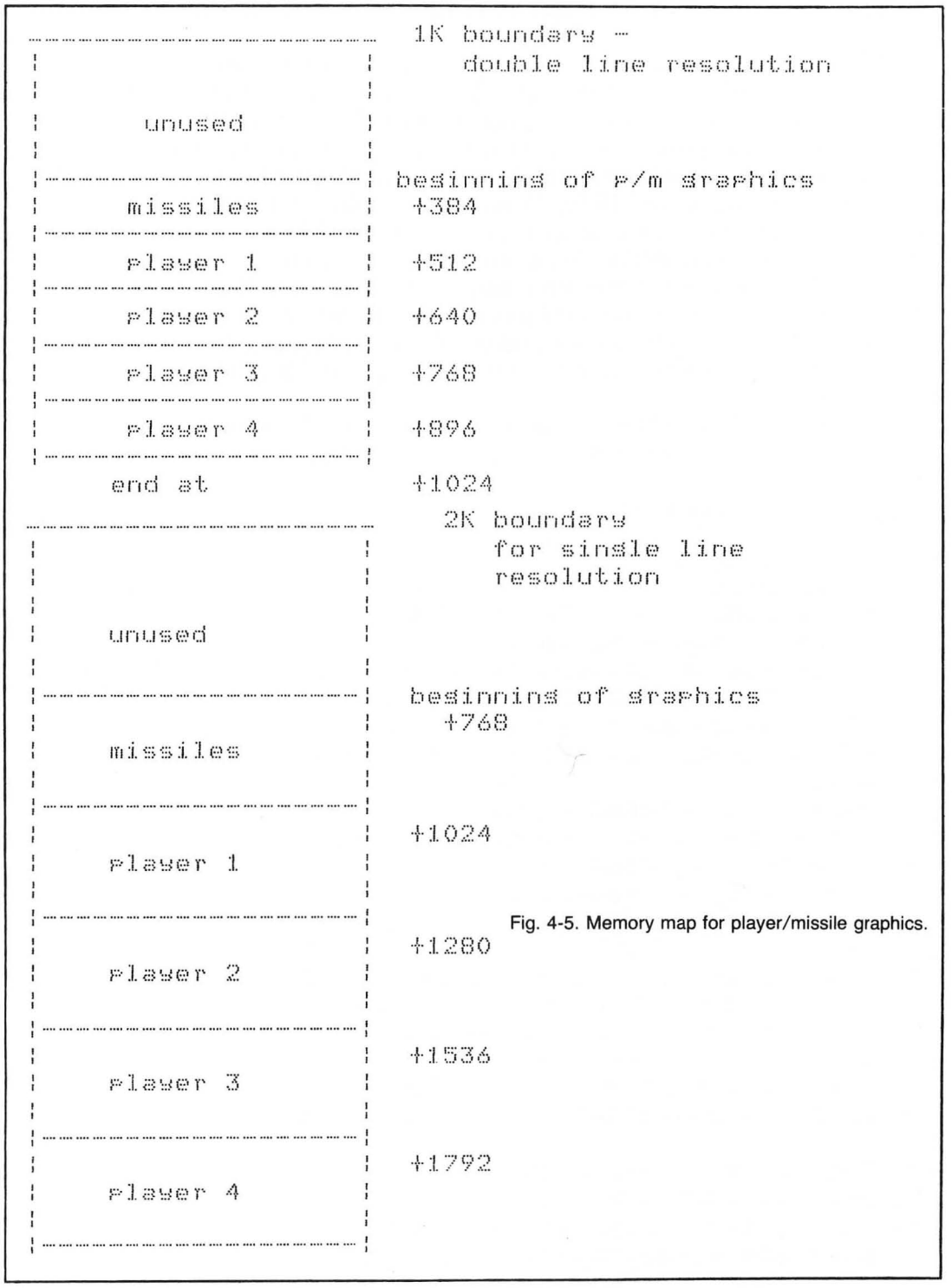


Fig. 4-5. Memory map for player/missile graphics.

This leaves 2K for the screen and display list and gives us an even 2K boundary to begin our graphics.

The second mode for player/missile graphics is the double resolution mode. This mode draws each byte on the screen twice as high, or uses two rows for each byte. Since it draws each byte twice, it uses only 128 bytes for each player or missile. The entire mode uses 1K of memory. To set aside memory for player/missile graphics that are double resolution, we need to start on an even 1K boundary. We can subtract 8 from the amount of RAM available. This will give us 1K for the player/missile graphics and 1K for the screen and display list in the text mode.

In addition to the two modes, the players and missiles can be in any of three sizes. When we set the mode to single or double resolution, we do it for all the players and missiles. Each player and/or missile can be in one of three sizes independent of each other. If the size is not set, the players and missiles will default to normal size. Each bit in the byte will be one pixel wide. Double size makes each bit two pixels wide and quadruple size makes the figure four times as wide as a normal one. Note: These figures refer to the width of the player. The resolution determines the height of the character.

By adding a few lines to the previous program, we can add a horse to the carousel. The following program uses double width and double line resolution.

Listing 4-5. Carousel—Animated

```
10 REM LISTING 4.5
20 REM CAROUSEL
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM P1$(20),UP$(20),DOWN$(20)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SSUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=24 TO 95:READ C:POKE CB+X,C:N
EXT X
100 DATA 0,3,15,63,255,255,255,204
110 DATA 255,255,255,255,255,255,255,2
04
120 DATA 0,192,240,252,255,255,255,204
130 DATA 9,6,5,5,9,6,5,5
140 DATA 144,96,80,80,144,96,80,80
150 DATA 144,96,80,80,144,96,255,255
```

```
160 DATA 9,6,5,5,9,6,255,255
180 DATA 2,2,2,2,15,63,255,255
190 DATA 21,84,21,0,192,240,252,252
200 REM DRAW CAROUSEL
210 ? "}CLEAR}":DLIST=PEEK(560)+PEEK(5
61)*256:REM FIND THE BEGINNING OF THE
DISPLAY LIST
220 POKE DLIST+3,69
230 FOR X=6 TO 28:POKE DLIST+X,5:NEXT
X:REM CHANGE ENTIRE DISPLAY LIST TO AN
TIC 5
240 POKE 756,A
250 POSITION 17,1:? "*+":REM FLAG ON T
OP
260 POSITION 7,2:? "$$$$$$$$$$$$$$$$$$
$$$%"
270 FOR X=3 TO 9:POSITION 7,X:? " &
  &       '       '":NEXT X:REM PRINT THE
 POLES
280 POSITION 7,X:? "}N}}}NNNNN}}}NNNNN
N}{}NNNNN}{}N}"
290 REM DRAW THE HORSE USING PLAYER/MI
SSILE GRAPHICS
300 A1=A-4:REM PLAYER/MISSILE GRAPHICS
 ARE 1K ABOVE THE CHARACTER SET
310 P1=A1*256+512:P2=P1+128:P3=P2+128:
P4=P3+128:REM FIRST MEMORY LOCATION OF
 PLAYERS
320 FOR X=P1 TO A*256-1:POKE X,0:NEXT
X:REM CLEAR OUT THE MEMORY
330 FOR X=P1+70 TO P1+79:READ C:POKE X
,C:NEXT X:REM GET DATA FOR THE HORSE
340 FOR X=P2+39 TO P2+65:POKE X,128:NE
XT X:FOR X=P2+66 TO P2+78:READ C:POKE
X,C:NEXT X:REM OTHER HALF OF HORSE
350 FOR X=P2+79 TO P2+102:POKE X,128:N
EXT X:REM REST OF THE POLE
360 FOR X=P3+58 TO P3+70:READ C:POKE X
,C:NEXT X
370 FOR X=P4+31 TO P4+61:POKE X,128:NE
XT X:FOR X=P4+62 TO P4+71:READ C:POKE
X,C:NEXT X
380 FOR X=P4+71 TO P4+94:POKE X,128:NE
XT X:REM REST OF THE POLE
390 POKE 559,46:REM DOUBLE LINE RESOLU
TION FOR P/M GRAPHICS
```

Listing 4-5. Carousel—Animated (continued from page 65)

```
400 POKE 53277,3:POKE 54279,A1:REM ENA
BLE P/M GRAPHICS
410 POKE 623,4:REM SET PLAYFIELD CHARA
CTERS PRIORITY OVER P/M GRAPHICS  - EN
ABLE FIFTH PLAYER
420 POKE 704,118:POKE 705,118:POKE 706
,116:POKE 707,116:REM HORSE OF A DIFFE
RENT COLOR
430 POKE 53256,1:POKE 53257,1:POKE 532
58,1:POKE 53259,1:REM MAKE HORSE LARGE
R
440 H1=105:H2=121:POKE 53248,H1:POKE 5
3249,H2:REM PUT HORSE ON THE SCREEN
450 DATA 15,31,63,47,42,42,10,18,36,8
460 DATA 132,142,159,191,251,240,224,2
40,168,168,200,144,160
470 DATA 32,112,248,252,223,15,7,31,20
,20,18,9,4
480 DATA 252,254,255,253,149,149,148,1
46,137,4
500 FOR X=1 TO 13:READ B:UP$(X,X)=CHR$
(B):NEXT X:REM MACHINE LANGUAGE SUBROU
TINE FOR UP
510 DATA 104,160,0,200,177,205,136,145
,205,200,208,247,96
520 FOR X=1 TO 13:READ B:DOWN$(X,X)=CH
R$(B):NEXT X:REM MACHINE LANGUAGE SUBR
OUTINE FOR DOWN
530 DATA 104,160,255,136,177,205,200,1
45,205,136,208,247,96
540 P01=INT(P1/256):P02=(P1-INT(P1/256
)*256):P03=INT(P3/256):P04=(P3-INT(P3/
256)*256):REM HI/LO ADDRESS OF HORSES
550 TRAP 660:RESTORE 640:DIR=2:HP1=532
48:HP2=53249:POKE 206,P01:POKE 205,P02
:REM MOVE TO THE RIGHT - PLAYERS USED
560 FOR X=1 TO 9:M=USR(ADR(UP$)):GOSUB
 670:NEXT X:GOSUB 600
570 POKE 206,P01:POKE 205,P02:IF DIR=-
2 THEN POKE 206,P03:POKE 205,P04
580 FOR X=1 TO 9:M=USR(ADR(DOWN$)):GOS
UB 670:NEXT X:GOSUB 600:GOTO 560
590 REM HORSE MOVING ROUTINE - DIR IS
POSITIVE - HORSE MOVES TO RIGHT - DIR
```

```
IS NEGATIVE HORSE MOVES LEFT
600 H1=H1+DIR:H2=H2+DIR
610 IF (DIR=2 AND H2>146) OR (DIR=-2 A
ND H1<72) THEN POKE HP1,0:POKE HP2,0:H
P1=HP1+DIR:HP2=HP2+DIR:DIR=DIR*-1
620 POKE HP1,H1:POKE HP2,H2:RETURN
630 REM MUSIC FOR THE CAROUSEL
640 DATA 121,121,108,108,96,81,96,121,
162,121,121,108,108,96,121,162,121
650 DATA 121,108,108,96,81,96,121,72,0
,108,91,96,121,0,0
660 TRAP 660:RESTORE 640:REM RESTORE M
USIC ON OUT-OF-DATA ERROR
670 IF (X+2)/3=INT((X+2)/3) THEN READ
S:SOUND 0,S,10,10
680 IF X/3=INT(X/3) THEN SOUND 0,0,0,0
690 RETURN
```

Line 300 subtracts 1K from the beginning of the character set. The character set begins on an even boundary, so subtracting 4 (or 1K) from its beginning yields an even boundary for our players.

Line 310 calculates the beginning address for each player. The first player (P1) begins 512 bytes after the beginning address for the player/missile graphics. The next player and each subsequent player begin 128 bytes after the previous player. We have already decided that these players will use double line resolution, so we know how much memory should be set aside for each player.

Line 320 clears the memory that will be used for the players. When the computer is turned on, or after a program has been run, there can be garbage in the memory area that we will be using. This line removes any data that may have been left there.

Lines 330-380 draw the horse in the player/missile area of memory. Player P1 is the back portion of the horse going to the right. Player P2 is the front of the horse and the pole. Player P3 is the front portion of the horse going to the left and player P4 is the back and the pole. The data to draw the horse is read from lines 450-480.

Line 390 pokes memory location 559 with 46. This sets the player/missile graphics to double line resolution.

Line 400 enables the player/missiles by poking 53277 with a 3 and tells the computer where the player/missiles begin by poking 54279 with the value stored in A1. Now the computer knows where the graphics are stored, and what the resolution of the graphics should be. If location 53277 is not poked with a 3, the player/missile graphics will not be enabled. Using player/missile graphics in a program requires both location 559 to be set and 53277 enabled.

Line 410 sets the priority levels of the players and characters on the screen. In this program, the characters that are printed on the screen will have higher priority than the players. This will make the horse appear to go behind the poles of the carousel.

Line 420 sets the colors used in the four players. The first two locations are for the first two

players. This is the horse as it is going from left to right. The next two locations are for the third and fourth players. This color is a little darker than the first color. The horse will be going from right to left. The darker color will give it the illusion of being further away.

Line 430 sets the size of the horses. By poking each of these locations with a 1, we will make each of the four players double the normal size.

Line 440 places the horse that is facing the right on the screen. Locations 53248 and 53249 set the first two players on the screen. To remove them from the screen, poke these locations with a 0.

Line 500 is the machine language subroutine that moves the horse up. Be sure that the data in line 510 is entered correctly. If it isn't, the horse will not move up correctly.

Line 520 is the machine language subroutine that moves the horse down. The instructions for this machine language subroutine are in line 530.

Line 540 calculates the beginning address of the first and third player. The machine language subroutines move 256 bytes up or down. The first horse uses the first two players, which add up to 256 bytes. The second horse uses the third and fourth players or the next 256 bytes. This line stores the high and low order address of the players in the variables P01, P02, P03, and P04.

Line 550 uses the trap command to test for the end of data.

Lines 640-650 contain the melody that the computer will be playing while the horse is going around. This melody will be played over and over. When the computer runs out of data it will come up with an error. The trap will direct the computer to Line 660. This line will reset the trap and restore the data. The DIR variable is the amount that will be added or subtracted from the position of the horse on the screen. When DIR is positive, the horse will move from left to right. When DIR is negative, the horse will move from right to left. HP1 and HP2 are the registers that are poked with the position of the horse on the screen. Memory locations 205 and 206 are poked with the memory location of the first player. Two bytes are needed for this location because the memory address is greater than 255.

Line 560 moves the horse up nine rows. The music subroutine is accessed every time the horse moves up one row. After the horse is moved up, the subroutine that moves it to the right or left is accessed.

Line 570 reinitializes the memory locations that are used in the machine language subroutines to the player that is being moved. If DIR is positive, the location of the first player will be stored in locations 205 and 206. If DIR is negative, the positions of the third player will be stored in these locations.

Line 580 uses the machine language subroutines to move the horse down. Again, the music subroutine will be accessed each time the horse is moved down one row. The subroutine to move the horse to the right or left will be used after the horse is moved down nine rows. These three lines will be repeated over and over again until the system reset key is pressed.

Line 600-620 contain the subroutine that moves the horse to the left or right. The value of DIR is added to the position stored in H1 and H2. If DIR is positive, two will be added to this value. If DIR is negative, two will be subtracted from this value. (Adding a negative number is the same as subtracting a positive number.) This line also checks the position of the horse on the screen. If the horse is at the end of the carousel, the value of DIR is reversed, the horse that is on the screen is removed, and the registers that control the position of the other horse are placed in variable HP1 and HP2. The other horse is then placed on the screen and the program returns.

Line 660 reinitializes the music routine. The program goes to this line when it runs out of

```
    PRIORITY SEQUENCE          POKE

       player 0
       player 1
       player 2
       player 3
  playfield 0                  1
  playfield 1
  playfield 2
  playfield 3 and player 5
  background

       player 0
       player 1
  playfield 0
  playfield 1
  playfield 2                  2
  playfield 3 and player 5
       player 2
       player 3
  background

  playfield 0
  playfield 1
  playfield 2
  playfield 3 and player 5
       player 0                4
       player 1
       player 2
       player 3
  background

  playfield 0
  playfield 1
       player 0
       player 1
       player 2                8
       player 3
  playfield 2
  playfield 3 and player 5
  background
```

Fig. 4-6. Player/missile priority order.

data. The trap is reset, the pointer for the data is restored to line 640. The program continues with the next line.

Line 670 reads a note every third time that this routine is accessed. We can calculate every third time beginning with the first time by adding 2 to the value of X and dividing it by 3. If it is a whole number (a number without a remainder) a new note will be read and played.

Line 680 turns the note off every third time that this routine is accessed. This time we simply divide X by 3. If it is a whole number, then the note will be turned off. The program will return to the main program.

SETTING THE PRIORITIES

In the last program, the horse seems to move behind the poles. The pole that the horse is on moves up behind the roof of the carousel and appears above it.

When the player/missile graphics are initialized, we can set priorities for the players, missiles, and the characters. The players can appear to move in front of the characters, behind the characters, or in front of some and behind others. Figure 4-6 illustrates the order of priority and the value that must be poked into decimal location 623. The playfield refers to the playfield characters, the characters that are printed on the screen or drawn on the screen with the plot and drawto commands. The *players* are the characters formed by the player/missile graphics. Player 5 is the fifth player or the missiles as a group. Note: The missiles can be used as four characters, each two bits wide, or as a fifth character eight bits wide. To enable the fifth player, poke 623 with 16 + the priority code.

As shown in Fig. 4-6, the top player or playfield has the highest priority. This character will appear on the screen in front of any other. Player 0 always has the highest priority followed by players 1, 2, and 3.

In the next program, which is a simple bird and fish game, we will set the priority code to 8. The players will move behind the playfield characters 0 and 1, but in front of playfield characters 2 and 3.

Some of the clouds are drawn using the color in playfield character 0, others use the color in playfield character 2. The water is made up of three different waves. Depending on the playfield color used, the fish will or will not be seen.

Listing 4-6. The Birds

```
10 REM LISTING 4.6
20 REM THE BIRD
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM P1$(20),UP$(20),DOWN$(20)
50 A=PEEK(106)-8:CB=A*256:REM PLACE CH
ARACTER SET 2K BEFORE END OF MEMORY
60 POKE 204,A:POKE 206,224:REM STORE T
HE NEW CHARACTER SET ADDRESS AND THE R
OM ADDRESS
70 FOR X=1 TO 20:READ B:P1$(X,X)=CHR$(
B):NEXT X:REM MACHINE LANGUAGE SUBROUT
INE TO MOVE CHARACTER SET
75 DATA 104,162,4,160,0,177,205,145,20
```

```
3,200,208,249,230,206,230,204,202,208,
242,96
80 Q=USR(ADR(P1$)):REM USE MACHINE LAN
GUAGE PROGRAM WITH THE USR FUNCTION
90 FOR X=24 TO 103:READ C:POKE CB+X,C:
NEXT X
100 DATA 63,127,255,255,127,63,7,0
110 DATA 0,192,240,248,240,240,192,0
120 DATA 0,60,126,255,255,127,63,12
130 DATA 48,56,124,254,255,127,255,78
140 DATA 5,79,127,63,127,255,99,0
150 DATA 224,246,255,255,252,222,140,4
160 DATA 34,68,85,119,255,255,255,255
170 DATA 0,68,85,255,255,255,255,255
180 DATA 68,34,170,238,255,255,255,255
190 DATA 255,255,255,255,255,255,255,2
55
200 GRAPHICS 17:POKE 756,A:REM COLOR T
EXT MODE & USE NEW CHARACTERS
210 POKE 708,14:POKE 709,180:POKE 710,
 14:POKE 711,180:POKE 712,150
220 FOR X=1 TO 10:C=INT(RND(1)*19):R=I
NT(RND(1)*10):POSITION C,R:C=INT(RND(1
)*6):IF X/2=INT(X/2) THEN C=C+128
230 ? #6;CHR$(C+35);:IF C=0 OR C=4 THE
N ? #6;CHR$(C+36)
240 NEXT X
250 FOR X=0 TO 19:C=INT(RND(1)*3):POSI
TION X,15:IF X/3=INT(X/3) THEN C=C+128
260 ? #6;CHR$(9+C);:NEXT X
270 FOR X=1 TO 8:? #6;"}LLLLLLLLLLLLLL
LLLLLL}";:NEXT X
280 A1=A-8:P1=A1*256+1024:P2=P1+256:P3
=P2+256:REM FIRST MEMORY LOCATION OF P
LAYERS-2K ABOVE CHARACTER SET
290 FOR X=P1 TO P1+1023:POKE X,0:NEXT
X:REM CLEAR OUT THE MEMORY
300 REM SET UP PLAYERS - BIRD & FISH
310 POKE 559,62:REM SINGLE LINE RESOLU
TION FOR P/M GRAPHICS
320 POKE 53277,3:POKE 54279,A1:REM ENA
BLE P/M GRAPHICS
330 POKE 623,8:REM SET PLAYFIELD CHARA
CTERS PRIORITY OVER P/M GRAPHICS - EN
ABLE FIFTH PLAYER
```

Listing 4-6. The Birds (continued from page 71)

```
340 POKE 704,2:POKE 705,200:POKE 706,2
00:REM COLOR BIRD & FISH
360 RESTORE 380:FOR X=P2+152 TO P2+159
:READ C:POKE X,C:NEXT X:REM DRAW FISH
SWIMMING
370 FOR X=P3+152 TO P3+159:READ C:POKE
 X,C:NEXT X:REM DRAW FISH SWIMMING
380 DATA 0,48,121,255,255,121,48,0
390 DATA 60,24,24,60,126,126,60,24
400 C=INT(RND(1)*3):RESTORE 401+C:FOR
X=P1+60 TO P1+65:READ C:POKE X,C:NEXT
X:REM GET DATA FOR THE BIRD
401 DATA 0,0,66,165,24,0
402 DATA 0,129,66,36,24,0
403 DATA 0,0,0,36,90,129
460 FOR X=1 TO 13:READ B:UP$(X,X)=CHR$
(B):NEXT X:REM MACHINE LANGUAGE SUBROU
TINE FOR UP
470 DATA 104,160,0,200,177,205,136,145
,205,200,208,247,96
480 FOR X=1 TO 13:READ B:DOWN$(X,X)=CH
R$(B):NEXT X:REM MACHINE LANGUAGE SUBR
OUTINE FOR UP
490 DATA 104,160,255,136,177,205,200,1
45,205,136,208,247,96
500 PO1=INT(P1/256):PO2=(P1-INT(P1/256
)*256):PO3=INT(P3/256):PO4=(P3-INT(P3/
256)*256):REM HI/LO ADDRESS OF PLAYERS

510 V=INT(RND(1)*100)+50:V1=200:U=60:P
OKE 206,PO1:POKE 205,PO2:GOSUB 720:REM
 SET VARIABLES FOR PLAYERS
520 POKE 53278,0:POKE 53249,V1:V1=V1-1
:IF V1=20 THEN V1=200
530 GOSUB 630
540 POKE 53248,V:C=INT(RND(1)*3):RESTO
RE 400+C:FOR X=P1+U TO P1+U+5:READ C:P
OKE X,C:NEXT X
560 IF PEEK(53260)<>2 AND (V1=200 OR U
=149) THEN IF V1/4=INT(V1/4) THEN FISH
=FISH+1:GOSUB 720:GOTO 520
570 IF PEEK(53260)<>2 THEN 520
580 POKE 53249,0:POKE 53250,V:BIRD=BIR
D+1
```

```
600 POKE 206,PO3:POKE 205,PO4:FOR X=1
TO 10:Q=USR(ADR(UP$)):Q=USR(ADR(DOWN$)
):NEXT X:REM BIRD EATS FISH
610 POKE 53250,0:FOR X=P1+U TO P1+U+5:
POKE X,0:NEXT X:REM ERASE BIRD
620 GOSUB 720:GOTO 500
630 IF STICK(0)=14 AND U>40 THEN Q=USR
(ADR(UP$)):U=U-1:RETURN :REM MOVE BIRD
 UP
640 IF STICK(0)=13 AND U<149 THEN Q=US
R(ADR(DOWN$)):U=U+1:RETURN :REM MOVE B
IRD DOWN
650 IF STICK(0)=7 AND V<190 THEN V=V+1
660 IF STICK(0)=11 AND V>53 THEN V=V-1
670 RETURN
710 REM ROUTINE PUTS SCORE ON SCREEN
720 POSITION 2,22:? #6;"BIRD":POSITION
 14,22:? #6;"FISH":POSITION 3,23:? #6;
BIRD;:POSITION 15,23:? #6;FISH;
730 IF FISH<100 AND BIRD<100 THEN RETU
RN
740 POSITION 2,10:? #6;"GAME OVER":POS
ITION 1,12:? #6;"PRESS START TO PLAY"
750 IF PEEK(53279)<>6 THEN 740
760 FISH=0:BIRD=0:POP :POKE 53248,0:PO
KE 53249,0:POKE 53250,0:POP :GOTO 200:
REM CLEAR VARIABLES
```

Lines 40-190 are the same as the past few programs. In this program they will dimension the strings used for the three machine language subroutines. The machine language subroutine to move the character set from ROM to RAM is the same. The data lines replace the characters from the pound sign to the plus sign. These characters will draw the clouds and the water on the screen. See Fig. 4-7 for these characters.

Line 200 sets the mode that will be used: large color characters with no text window. Poking 756 with the value of A changes the character set.

Line 210 changes the colors of the characters. There will be two colors used for all the characters: white and green. The background will be changed to light blue.

Lines 220-240 place the clouds on the screen. The position for the clouds is chosen randomly. No two screens will look the same. If the first or fifth cloud is chosen, the program will add the second half of the cloud. Half of the clouds on the screen will be drawn with the characters in the normal character set. The other half of the characters will be drawn as if they were printed in inverse video. The color will be the same, white, but the priority will be different.

Lines 250-260 print the top of the water on the screen. This time every third will be printed as a inverse video character.

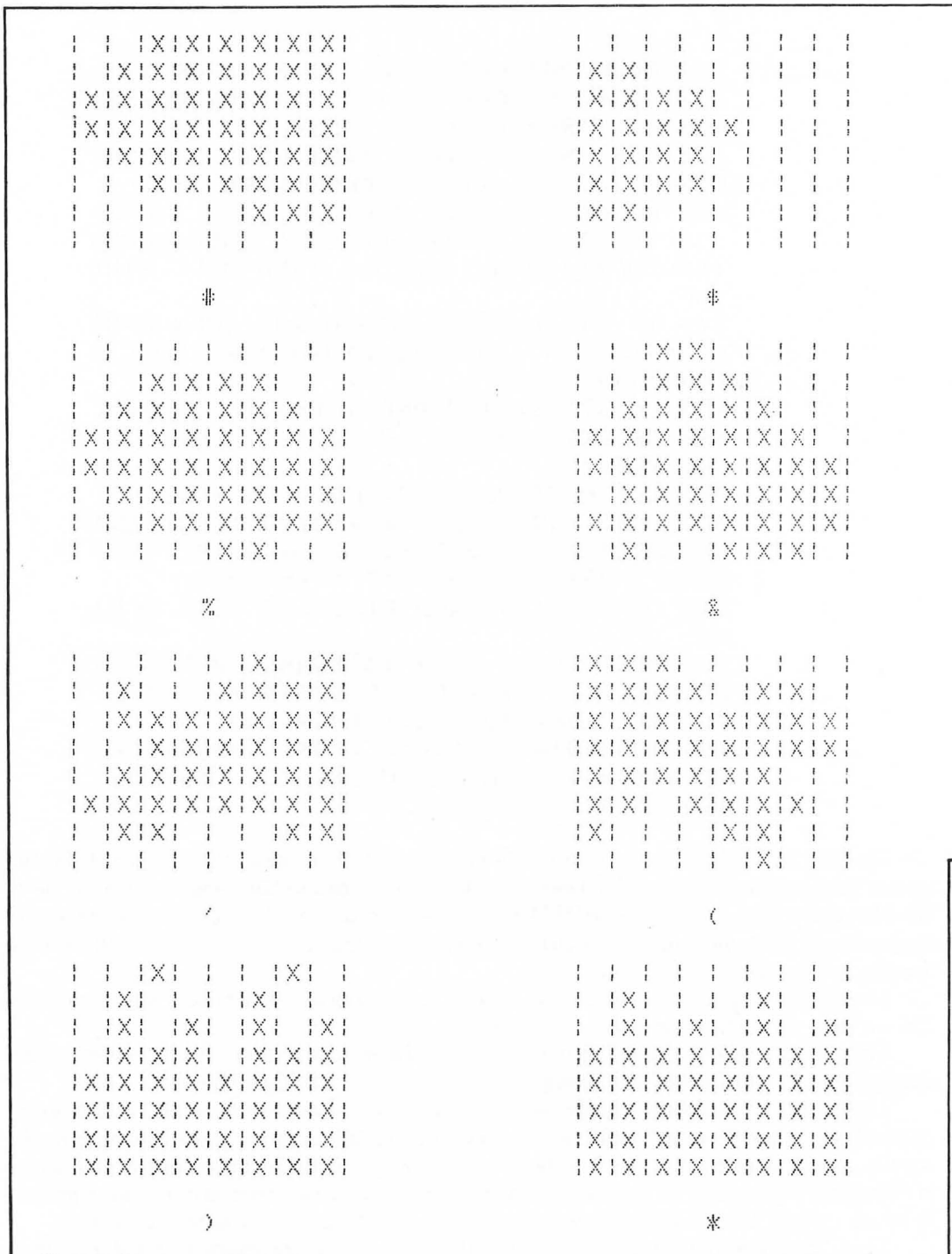
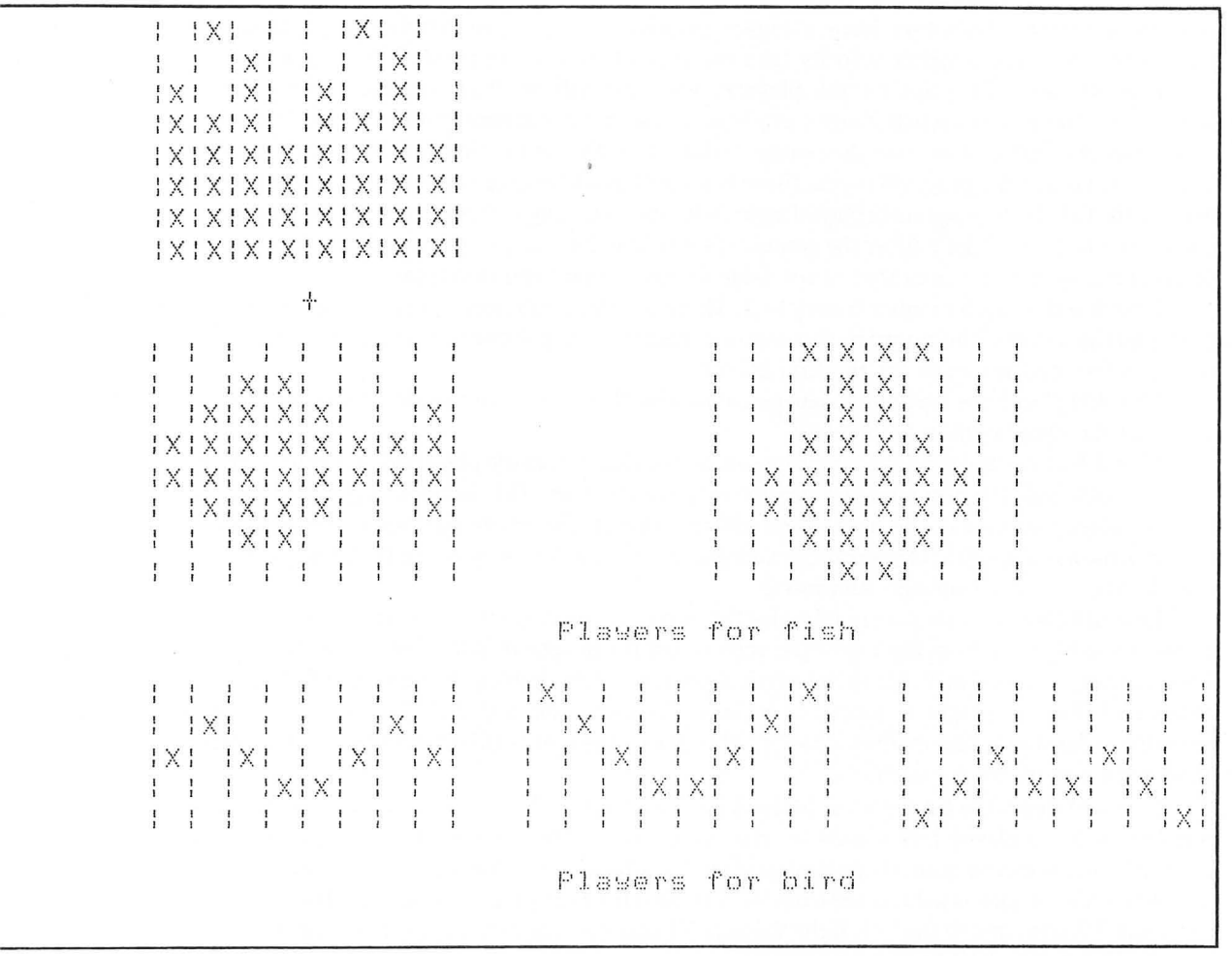


Fig. 4-7. Character set for birds.



Line 270 colors the bottom of the screen with water.

Line 280 calculates the beginning of the player/missile graphics area. In this program we are using the single resolution character set so the player/missile graphics must begin on an even 2K boundary. By subtracting 8 (2K) from the beginning of the character set, we know where to begin the player/missile graphics. The first player begins 1024 bytes from the beginning of the memory set aside. The second and third players are 256 bytes apart.

Line 290 clears the area of memory that will be used for player/missile graphics. This memory could contain data from a previous program or garbage. This would show up on the screen in the player/missile area.

Line 310 pokes 559 with 62. This tells the computer that we are using single resolution for these players.

Line 320 enables the player/missile graphics. If a 3 were not poked into 53277, the player/missile graphics would not be enabled. Poking 54279 with the value of A1 tells the computer where the player/missiles begin.

Line 330 sets the priorities of the players and characters. The characters that would be

printed as normal characters have a higher priority than the player/missile graphics. The player/missiles have a higher priority than the characters that are printed in inverse video.

Line 340 sets the color for the players. The bird will be black and the fish are green. Remember, there are two fish images as players, one that is swimming and one that is caught.

Lines 360-390 first restore the pointer to line 380. This is the first line of data for the fish. The first time that the program is run, there is no problem having the computer read the correct data for the fish. If the program is played again, without rerunning it, the pointer will be pointing to one of the lines of bird data. After the pointer is set to line 380, the program reads the data into the player/missile graphics area that is set aside for the second and third players.

Line 400 chooses a number from 0 to 2. There are three different ways that the bird can be drawn on the screen. The program chooses one, restores the pointer to that line, then reads the data into the area set aside for the first player.

Line 460 places the machine language subroutine that will move the player up into UP$. Be sure that the data is entered correctly.

Line 480 places the machine language subroutine that moves the players down into DOWN$.

Line 500 finds the first memory location of the first player. This location is greater than 255, so it occupies two bytes. The high order address, that is, the whole number of the address, is stored in the variable P01. The low order address, the remainder, is stored in P02. This address is used by the machine language subroutine.

Line 510 chooses a random number for the vertical position of the bird. The bird will always be the same distance from the top of the screen, but it can appear in any column on the screen. This number is stored in V. V1 is the vertical position of the fish on the screen. The beginning address of the first player is placed in memory locations 205 and 206. The program is then directed to the subroutine in lines 720-730. This places the words BIRD and FISH on the screen along with a score (0) for each.

Line 520 clears the *hit* register. In the ATARI computer, there is one memory location that registers when a player hits a missile, characters, or another player. This register must be cleared before it can be read. By poking location 53249 with the value stored in V1, we place the fish on the screen. One is subtracted from V1. The next time the program executes this line, it will move the fish over one to the left. If the value in V1 becomes equal to 20, the fish is nearly off the screen. The value of V1 is reset to 200.

Line 530 sends the computer to line 630. The computer will check the joystick to see if it has been moved. When the computer returns to this part of the program, the bird may have moved up or down, or the value in variable V may have changed indicating that the bird has moved either to the right or the left.

Line 540 moves the bird to the right or left by poking the value of V into register 53248. The computer then chooses a random number to change the wings on the bird. The number is added to 400 and the pointer for the data is restored to this line. The computer reads the new data for the bird and pokes it into the area of memory in the player/missile graphics area that the bird occupies. The variable P1 is the beginning of the first player's area of memory. The variable U is the offset for the first byte of the bird. It increases and decreases as we move the bird up and down on the screen.

Line 560 checks the register that will record whether or not the bird (player 1) has hit the fish (player 2). If the bird has not hit the fish (PEEK(53260)< >2) and the fish has been placed back on the right side of the screen, (V1=200) or if the bird is resting on the water (U=149) and the fish has moved four places, the fish will be given a point. This keeps the player from landing the bird on

the water and waiting for the fish to come by. The subroutine in line 720 updates the score and the computer continues the program with line 520.

Line 570 sends the computer back to line 520 if the bird has not hit the fish.

Line 580 removes the swimming fish from the screen. The computer will execute this line if the value in register 53260 is 2. The swimming fish is replaced by the hanging fish. The score for the bird is increased by one.

Line 600 pokes the beginning address for the third player, the hanging fish, into memory locations 206 and 205. The machine language subroutines to move the fish up and down are executed ten times. The fish never really moves up and down. This gives the effect of the fish wiggling while the bird tries to eat it.

Table 4-1. Machine Language Listing to Move Players Up/Down.

```
Decimal                          Assembly Language Listing
 Code

  104       PLA                  ;Pull the accumulator off the stack
  160       LDY #0               ;Load the index Y with zero
    0
  200       INY                  ;Increment the index Y
  177       LDA (205),Y          ;Load the accumulator with the value
  205                             of the address in location 205-206
                                  offset by Y
  136       DEY                  ;Decrement the index Y
  145       STA (205),Y          ;Store the value in the accumulator
  205                             in the memory locations pointed to
                                  by 205-206 offset by Y
  200       INY                  ;Increment Y
  208       BNE                  ;If the index Y is not zero, go
  247                             back 8 bytes
   96       RTS                  ;Return to BASIC

                  Routine to move character up

  104       PLA                  ;Pull the accumulator off the stack
  160       LDY #255             ;Load the index Y with 255
  255
  136       DEY                  ;Decrement index Y
  177       LDA (205),Y          ;Load the accumulator with the
  205                             value of the address in location
                                  205-206 offset by Y
  200       INY                  ;Increment Y
  145       STA (205),Y          ;Store the value in the accumulator
  205                             in the memory location pointed to
                                  by 205-206 offset by Y
  136       DEY                  ;Decrement Y
  208       BNE                  ;If the index Y is not zero, go
  247                             back 8 bytes
   96       RTS                  ;Return to BASIC

                Routine to move character down
```

Line 610 erases the bird. The bird will be redrawn in the player area of memory. If we don't erase the bird, we will have two birds in that area when we only want one.

Line 620 updates the score and sends the computer to line 500 where the bird is repositioned on the screen and you are given another chance to try to catch a fish.

Lines 630-670 check the joystick to see if it has moved. If it has been moved up or down, the computer will use the correct machine language subroutine to move the bird. The variable U1 will be changed to reflect the new position of the bird on the screen. If the joystick has been moved to the left or right, the variable V will change.

Line 720 updates the score on the bottom of the screen.

Line 730 checks the score to see if either the bird or the fish has over 100 points. If neither does, the computer will return to the same.

Line 740-760 ends the game when either the fish or the bird passes 100. The computer checks location 53279 to see if the start key has been pressed. When the value of this location is 6, the start key has been pressed, and the program can continue. The scores are cleared, the players are removed from the stage and the program goes to line 200 to begin the game again. Since this was entered as a subroutine, the return address is popped off the stack. If the return address for a subroutine is not popped off the stack, it will stay there. If more and more addresses are placed on the stack and never removed, the stack could run out of space, causing the program to crash.

Table 4-1 contains explanations of the machine language subroutines that are used to move the players up and down. Each machine language subroutine that is to return to BASIC must pull the last byte off the stack. If it doesn't, the subroutine will not return.

Chapter 5
Looking at BASIC

By now, writing programs in BASIC is almost as natural as writing letters in English. But, did you ever wonder how the computer interprets the commands that you type in? Or how it knows that the line that was just entered contains an error? Just what does the computer do with a program?

THE TOKEN COMMANDS

Each time that you see a variable in a line or command, BASIC looks it up in its *Variable Name Table*. Each variable is assigned a number in the order that it was entered. This number is a token that represents the variable name in the line or command. If the variable appears in the table, BASIC assigns its token for the variable. If it doesn't appear in this table, it is added to the table. Up to 128 variables can be used in one BASIC program.

The BASIC commands are converted into token commands. A number or token represents every command that BASIC knows. A one number token uses less memory than a four or five character word. As you enter a program line, BASIC converts the line into a string of numbers or tokens. This makes it easy for BASIC to execute the program.

The line numbers that you enter are converted into two byte numbers and stored in the area that BASIC sets aside for the program. Why two bytes? BASIC will accept line numbers up to 32767. When this number is converted to hex, it becomes 7FFF, the largest positive sign number possible. Any number in hex 8000 or larger is considered a negative number, BASIC does not allow negative line numbers.

By changing the line number that we entered into a two byte number, every line in BASIC will have two bytes set aside for line numbers. This makes it easy for BASIC to manipulate the lines.

Once the line number has been converted into a two byte number, a dummy number will be placed into the line. This number will contain the offset, or the number of bytes in this line. Right now, BASIC does not know how many bytes are needed for this line. The next number is how many bytes are in this statement. Since there can be more than one statement on each line, BASIC must keep track of both the line length and the statement length. This number will also be a dummy number until the entire line is checked.

Now that BASIC knows that this is a program line, it looks for a command. The entire list of

Table 5-1. BASIC's for Tokens Commands.

TOKEN	COMMAND
0	REM
1	DATA
2	INPUT
3	COLOR
4	LIST
5	ENTER
6	LET
7	IF
8	FOR
9	NEXT
10	GOTO
11	GO TO
12	GOSUB
13	TRAP
14	BYE
15	CONT
16	COM
17	CLOSE
18	CLR
19	DEG
20	DIM
21	END
22	NEW
23	OPEN
24	LOAD
25	SAVE
26	STATUS

possible commands is in ROM. If the first command following the line number is not in this list, an error message will appear on the screen.

If the command is found in the table, it will be converted into a code or token value. Depending on the command, BASIC will check the next part of the statement to make sure that it is accurate. For example, the print command must be followed by double quotes, a variable, or a string variable, FOR must be followed by a variable that is equal to a number, the next part of the command, TO, and another number, TRAP must be followed by a line number or a variable, etc. If any element of the statement is missing or otherwise incorrect, BASIC will stop checking the line, reprint it on the screen, with the word ERROR, and highlight the possible problem area. Once BASIC has determined that the statement is correct, it will replace the dummy numbers with the correct figures and wait for the next statement or line to be entered.

A complete list of commands and their BASIC tokens are listed in Table 5-1. Each command has its own numerical token. When you enter a BASIC statement into the computer and you do not type the entire word out; for example, GR. 1 instead of GRAPHICS 1, and you list the program, BASIC will expand the command and print it correctly. When entering a BASIC program, you only

27	NOTE
28	POINT
29	XIO
30	ON
31	POKE
32	PRINT
33	RAD
34	READ
35	RESTORE
36	RETURN
37	RUN
38	STOP
39	POP
40	? (PRINT)
41	GET
42	PUT
43	GRAPHICS
44	PLOT
45	POSITION
46	DOS
47	DRAWTO
48	SETCOLOR
49	LOCATE
50	SOUND
51	LPRINT
52	CSAVE
53	CLOAD
54	LET (IMPLIED)
55	ERROR

save keystrokes, not bytes, when you use the abbreviated forms of the commands. The command will use the same amount of memory no matter which way it was entered. The same is true about spaces. On certain other computers, you can save memory by eliminating the spaces in the statements. ATARI BASIC will automatically place spaces between the commands when it lists the program on the screen.

FILE STRUCTURES

When we store a value in a variable, string, or array, BASIC must be able to reference the variable and to store or retrieve the information. First, it must be able to identify the type of variable. Then, it must have memory set aside for it.

Each time we use a new variable in our program, we use eight bytes of memory. A string and an array uses the eight bytes plus the size of the string or the array. The names of the variables are stored in a table. A second table stores the value of the variable or the location in memory that stores the string or array information. Because of the amount of memory that is used by variables, strings, and arrays, we try to reuse variable names whenever possible.

However, it is better to use variables than numbers in a program if you will be using the number often. For example, if you will be going to a line for a timing routine from different parts of the program, it saves memory to make the line number for timing routine equal to a variable, and then GOSUB the variable.

BASIC TABLES

The first table that BASIC uses is called the Variable Name Table. Every variable used in a program is assigned a number from 0 to 127. If you try to use more than 128 variables in a program you will set an error message.

Listing 5-1. BASIC Tables—Variable Name Table

```
10 REM LISTING 5-1
20 REM BASIC TABLES
30 REM BY L.M. SCHREIBER FOR TAB BOOKS

40 DIM A(10,2),STRING$(10)
50 TABLE=PEEK(130)+PEEK(131)*256
60 FOR X=TABLE TO TABLE+20:? CHR$(PEEK
(X));:NEXT X
```

If you enter the program without any errors, your screen should display -

A(STRING$TABLEX

and some other characters. The underlined characters appear in inverse video on the screen.

Look at the listing. Each variable appears in the order that it was entered into the program. If an error was made, for example, PRRK was typed in line 50 instead of PEEK, BASIC would have treated PRRK as a variable and placed it in the table even if you corrected it before the program was run!

Each type of variable is stored differently in the table so that BASIC can tell which are variables, arrays, and strings. If it is a variable, its last character is stored with the most significant bit set. This makes the character appear in inverse on the screen. The E in the variable table is in inverse. The X is only one character long, so it is in inverse video.

If the variable is an array, the character after the variable is an open parenthesis with the most significant bit set. The first variable in our table is an array.

If the variable is a string variable, the first character after the variable will be the $ with the most significant bit set. STRING$ has the most significant bit of the $ set.

As new lines are typed in for a program, BASIC checks this table to see if each variable has been used before. If it has, BASIC assigns its token for the variable. If it doesn't appear in this table, it is added to the table and its token is used in that line.

Because each variable is stored in this table, shorter variable names will, of course, use less memory. But, because each variable has its own token in a BASIC line, you only save memory in the variable table. The program will use the same number of bytes whether the variable is one character long or 10 characters long.

Once the variables have been stored in the Variable Name Table, BASIC has the token number for that variable. The first variable is 0, the second 1, etc. Information for each variable is

stored in the Variable Value Table. Each variable is listed in this table in the order that it is listed in the Variable Name Table. Let's add these lines to our program.

Listing 5-1A. BASIC Tables—Variable Value Table

```
10 REM LISTING 5-1A
20 REM BASIC TABLES
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM A(10,2),STRING$(10)
50 TABLE=PEEK(130)+PEEK(131)*256
60 FOR X=TABLE TO TABLE+20:? CHR$(PEEK
(X));:NEXT X:? :REM SHOW THE VARIABLES
 IN THE TABLE
70 VLUE=PEEK(134)+PEEK(135)*256:REM AD
DRESS OF THE VARIABLE VALUE TABLE
80 P=0:FOR X=VLUE TO VLUE+31:REM SHOW
INFORMATION FOR FIRST 4 VARIABLES
90 PRINT PEEK(X),:P=P+1:IF P=4 OR P=8
THEN PRINT :IF P=8 THEN PRINT :P=0
100 NEXT X
```

The numbers that appear on your screen tell the computer what kind of variable each one is and supplies information on its contents. Your screen should look like this:

```
65      0       0       0
11      0       3       0

129     1       198     0
0       0       10      0

0       2       65      118
118     0       0       0

0       3       65      119
37      0       0       0
```

The first eight bytes (numbers) contain information for the first variable A. The 65 indicates that it is a dimensioned array. The 0 is the variable number. A is the first variable entered in this program. The next two bytes are added to the beginning of the string/array area to find the beginning of the data for the array. In this case, A is the first variable dimensioned, so its data will not be offset from the beginning of the string/array area. It will be contained in the first 198 bytes. The next two numbers, the 11 and the 0 are a two byte value for the first dimension of the array. Our array is dimensioned to 10,2; the first dimension is 10. The first dimension of the array is always one greater than the value in the dimension statement. The last two numbers, the 3 and 0 is the second value of the array. Again, this value is one greater than the value in the DIM statement.

Table 5-2. Variable Value Table.

byte	1	2	3	4	5	6	7	8
numeric variable	0	v#	6 byte Binary Coded Decimal					
array	65	v#	offset		1st DIM+1		2nd DIM+1	
string	129	v#	offset		length		DIM	

The second group of numbers is the information for our string, STRING$. The first byte is 129. This means that this variable is a dimensioned string. The next byte is the variable number. STRING$ is the second variable in the table. The next two bytes contain the offset that is added to the beginning of the string/array area to find out where the data for the string is stored. The contents of the first byte is added to the contents of the second byte after it is multiplied by 256. In this case, the second byte is zero. So we know that the information stored in STRING$ begins 198 bytes after the beginning of the string/array area. The next two bytes contain the length of STRING$. We have not stored anything in this string, so its length at this time is zero. The last two bytes in this group tell the computer how many bytes to set aside for this string. We dimensioned STRING$ to 10, so the first of the two bytes is 10, the other is zero.

The next group of bytes contain the information for the variable TABLE. This variable was not dimensioned. It was the next variable that was entered into the computer. The first byte for this variable is a zero. This means that it is a numeric variable. Only one number can be stored in this variable. The next byte is the variable number. This is the third variable in this program, so its number is 2. The next six bytes contain the value of TABLE. Since only one number can be stored in a numeric variable at a time, the computer stores this information right in this table. It uses six bytes because it stores the number as a Binary Coded Decimal. This format differs from the format used when the computer stores a number using two bytes. At this point it is not necessary to understand how or why the computer uses this format, just that it does.

The last group of bytes contain the information for the variable X. Again, this is a numeric variable as indicated by the zero. It is the fourth variable used in this program so its number is 4. The value stored in X is represented in the next 6 bytes. Table 5-2 is a chart showing the different ways the variables can be represented in the Variable Value Table.

The area set aside for the strings and arrays is called the String/Array area. The address for the beginning of this area is stored in memory locations 140 and 141. Let's add the following lines to our program.

Listing 5-1B. BASIC Tables—String-Array Area

```
10 REM LISTING 5.1B
20 REM BASIC TABLES
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM A(10,2),STRING$(10)
50 TABLE=PEEK(130)+PEEK(131)*256
60 FOR X=TABLE TO TABLE+20:? CHR$(PEEK
```

```
(X));:NEXT X:? :? :REM SHOW THE VARIAB
LES IN THE TABLE
70 VLUE=PEEK(134)+PEEK(135)*256:REM AD
DRESS OF THE VARIABLE VALUE TABLE
80 P=0:FOR X=VLUE TO VLUE+31:REM SHOW
INFORMATION FOR FIRST 4 VARIABLES
90 PRINT PEEK(X),:P=P+1:IF P=4 OR P=8
THEN ? "":IF P=8 THEN PRINT :P=0:REM
PRINT ESC-CTRL-UPARROW IF 4 OR 8
100 NEXT X
110 STRING$="HI THERE"
120 FOR X=VLUE+8 TO VLUE+15:? PEEK(X),
:NEXT X:? :REM SHOW THE CHANGE IN THE
VARIABLE VALUE TABLE
130 STAREA=PEEK(140)+PEEK(141)*256:REM
 FIND THE BEGINNING OF THE STRING/ARRA
Y AREA
140 MESSAGE=STAREA+198:REM FIND THE ST
RING
150 FOR X=MESSAGE TO MESSAGE+LEN(STRIN
G$)-1:REM START TO END OF STRING$
160 ? CHR$(PEEK(X));:REM PRINT THE CHA
RACTER OF THE VALUE IN THIS AREA
170 NEXT X
180 ? :? "STRING=";STRING$
```

The fifth group of bytes is nearly identical to the second group. This is the information for STRING$. The only byte that is different is the fifth byte. It contains an eight because the message in STRING$ is eight characters long. The next line prints the contents of STRING$ by peeking at the area in memory where STRING$ is stored. Finally, STRING$ is printed to show that the message is the same.

By knowing this information, it is possible to trick the computer into looking at memory that was not originally set aside as a string by the computer. In the next chapter you will learn how to manipulate this information.

Another area of memory that BASIC uses is the Output Buffer. When a BASIC line is entered into the computer, the entry must be stored somewhere while it is being tokenized and checked for the proper structure. The area set aside for this is stored in memory locations 128 and 129. This area or buffer is 256 bytes long.

Listing 5-1C. BASIC Tables—Buffer

```
10 REM LISTING 5.1C
20 REM BASIC TABLES
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
40 DIM A(10,2),STRING$(10)
50 TABLE=PEEK(130)+PEEK(131)*256
60 FOR X=TABLE TO TABLE+20:? CHR$(PEEK
```

Listing 5-1C. BASIC Tables—Buffer (continued from page 85)

```
(X));:NEXT X:? :? :REM SHOW THE VARIAB
LES IN THE TABLE
70 VLUE=PEEK(134)+PEEK(135)*256:REM AD
DRESS OF THE VARIABLE VALUE TABLE
80 P=0:FOR X=VLUE TO VLUE+31:REM SHOW
INFORMATION FOR FIRST 4 VARIABLES
90 PRINT PEEK(X),:P=P+1:IF P=4 OR P=8
THEN ? "":IF P=8 THEN PRINT :P=0:REM
PRINT ESC-CTRL-UPARROW IF 4 OR 8
100 NEXT X
110 STRING$="HI THERE"
120 FOR X=VLUE+8 TO VLUE+15:? PEEK(X),
:NEXT X:? :REM SHOW THE CHANGE IN THE
VARIABLE VALUE TABLE
130 STAREA=PEEK(140)+PEEK(141)*256:REM
 FIND THE BEGINNING OF THE STRING/ARRA
Y AREA
140 MESSAGE=STAREA+198:REM FIND THE ST
RING
150 FOR X=MESSAGE TO MESSAGE+LEN(STRIN
G$)-1:REM START TO END OF STRING$
160 ? CHR$(PEEK(X));:REM PRINT THE CHA
RACTER OF THE VALUE IN THIS AREA
170 NEXT X
180 ? :? "STRING=";STRING$
190 BUFFER=PEEK(128)+PEEK(129)*256:REM
 BEGINNING OF THE BUFFER
200 FOR X=BUFFER TO BUFFER+150:? CHR$(
PEEK(X));:REM PRINT THE CONTENTS OF TH
E BUFFER
210 NEXT X
```

The contents of this buffer will vary from program to program depending on what has been entered into the computer.

Once the lines of the program have been tokenized and placed in the program, BASIC has to know where in memory the program begins. The address of the beginning of the BASIC program is stored in the *Statement Table* in memory locations 136 and 137. By adding a few more lines to the program you can see the tokenized BASIC program.

Listing 5-1D. BASIC Tables—Statement Table

```
10 REM LISTING 5.1D
20 REM BASIC TABLES
30 REM BY L.M. SCHREIBER FOR TAB BOOKS
```

```
40 DIM A(10,2),STRING$(10)
50 TABLE=PEEK(130)+PEEK(131)*256
60 FOR X=TABLE TO TABLE+20:? CHR$(PEEK
(X));:NEXT X:? :? :REM SHOW THE VARIAB
LES IN THE TABLE
70 VLUE=PEEK(134)+PEEK(135)*256:REM AD
DRESS OF THE VARIABLE VALUE TABLE
80 P=0:FOR X=VLUE TO VLUE+31:REM SHOW
INFORMATION FOR FIRST 4 VARIABLES
90 PRINT PEEK(X),:P=P+1:IF P=4 OR P=8
THEN ? "":IF P=8 THEN PRINT :P=0:REM
PRINT ESC-CTRL-UPARROW IF 4 OR 8
100 NEXT X
110 STRING$="HI THERE"
120 FOR X=VLUE+8 TO VLUE+15:? PEEK(X),
:NEXT X:? :REM SHOW THE CHANGE IN THE
VARIABLE VALUE TABLE
130 STAREA=PEEK(140)+PEEK(141)*256:REM
 FIND THE BEGINNING OF THE STRING/ARRA
Y AREA
140 MESSAGE=STAREA+198:REM FIND THE ST
RING
150 FOR X=MESSAGE TO MESSAGE+LEN(STRIN
G$)-1:REM START TO END OF STRING$
160 ? CHR$(PEEK(X));:REM PRINT THE CHA
RACTER OF THE VALUE IN THIS AREA
170 NEXT X
180 ? :? "STRING=";STRING$
190 BUFFER=PEEK(128)+PEEK(129)*256:REM
 BEGINNING OF THE BUFFER
200 FOR X=BUFFER TO BUFFER+150:? CHR$(
PEEK(X));:REM PRINT THE CONTENTS OF TH
E BUFFER
210 NEXT X
220 PROGRAM=PEEK(136)+PEEK(137)*256:RE
M THE PROGRAM
230 FOR X=PROGRAM TO PROGRAM+1100:? CH
R$(PEEK(X));:NEXT X:REM PRINT THE TOKE
NIZED PROGRAM
```

The remarks are perfectly readable. The rest of the listing should look like code. It is. Every command is converted to one of the token values (Table 5-1). The variables have their own tokens. It would be very hard for you to try to read this listing.

In order for BASIC to keep track of where it is when executing the program, it sets aside two bytes of memory to use as a pointer. Memory locations 138 and 139 contain the address of the current statement. When BASIC is not executing a program, this buffer points to the beginning of

the immediate line mode, which is the area that the next command will be placed if it is not a BASIC line, but an immediate command.

When BASIC is executing a program, it also needs an area set aside for return statements and for . . . next loops. This is called the Run Time Stack. When BASIC executes a GOSUB, it must know where to return to. Four bytes are used for every GOSUB. One byte indicates that the next address is a return address for a GOSUB. The next two bytes contain the line number to return to. The fourth byte is the offset in the line, so that BASIC will continue with the next statement on that line.

A for . . . next loop uses 16 bytes of memory in the stack: the last number that the variable can count to (6 bytes), the step of the for . . . next loop (6 bytes), the variable name of the variable that is counting, the line number (2 bytes), and the offset of the for statement. The first two numbers use the Binary Coded Decimal format.

Incorrect use of the for . . . next loop or GOSUBs without RETURNs and/or POPs can cause a program to crash. Table 5-3 is a chart that shows the addresses that BASIC uses to store this information.

SPEEDING UP A PROGRAM

Now that we have an understanding of how BASIC stores a program and the pointers that it uses to keep track of the program as it runs it, we can use different techniques to speed up a program and to use memory effectively. A well written program should run smoothly and use only as much memory as necessary. Having a computer with 48K in it does not mean that you should not try to conserve memory. If your short programs are written loosely, and you do not get into the habit of trying to write the program as tightly as possible, you will run out of memory very quickly when you try to write a large program.

Sometimes the only way to shorten a program is to recode it. If the program was not flow charted or modifications were added to the program after it was written, you may find that the

Table 5-3. Table of Addresses for BASIC Tables.

decimal address	table name
128,129	Output buffer
130,131	Variable Name Table
134,135	Variable Value Table
136,137	Statement Table
138,139	Current Statement
140,141	String/array area
142,143	Run time stack

program has several routines that are the same. These routines can be made into subroutines, and several lines of code can be removed from the program. You may also find a subroutine that is only called once. Move that routine to the main program. GOSUBs and RETURNs waste bytes if the routine is only used once.

Use a variable instead of a number if the number is used more than once. Each time a number is placed in a BASIC line, it uses 7 bytes. If the same number is used twice, that's 14 bytes. Assigning a number to a variable uses 10 bytes. Each time the variable is used in a line, the variable's token is placed in the line. This is one byte. Assigning a number to a variable, then using that variable twice uses only 12 bytes. Obviously, the more frequently the number will be used in the program, the more bytes will be saved.

Place the most frequently called subroutines at the beginning of the program. BASIC begins at the beginning of the program and works its way down looking for lines. If the line is at the beginning, BASIC doesn't have to look very far.

If one subroutine calls another subroutine and then returns to the main program, have the first subroutine GOTO the second subroutine so that it will return to the main program from the second subroutine.

Example 1

```
100  GOSUB 500
110  . . .
120  . . .
490  . . .
500  TL=10:PRINT"Let's try that again":GOSUB 600
510  RETURN:REM this return is unnecessary -
```

Example 2

```
100  GOSUB 500:REM the correct way
110  . . .
120  . . .
490  . . .
500  TL=10:PRINT"Let's try it again.":GOTO 600
```

In the first example, line 500 contains a GOSUB. BASIC will go to that subroutine, then return to line 510. Line 510 contains a return. So, in effect, the computer is returning to a return. In the second example, the GOSUB is replaced with a GOTO. Line 600 is still a subroutine, it still has a RETURN at the end of it, but because the program went to the subroutine as a GOTO, when it comes to the RETURN, the address on the stack will be the next line in the main program. This method saves both time and memory.

Use POKEs instead of SETCOLOR. This will save about 8 bytes. POKEs can also be used for the sound command.

Use logic instead of comparison if possible. One way is to set a variable to 0 if the condition is false and to 1 if it is true. Then instead of an IF X= statement, you can use an IF X THEN. For example, you may have a program that has a printer option. The prompt DO YOU WANT A HARD COPY? appears. If the user answers yes, a variable is set to 1 (PRTR=1). If the user answers no, the variable is set to 0 (PRTR=0). Now, when you get to the part of the program that will print either to the screen or the printer, instead of a line that reads:

```
1200  IF PRTR$="YES" THEN LPRINT "REPORTS"
```

your line will read:

```
1200  IF PRTR THEN LPRINT "REPORTS"
```

If PRTR is one, the statement is true and the word REPORTS will be printed on the printer. If the variable PRTR is 0, the statement is false and the computer will go on to the next line.

Use assembly language subroutines when possible. The assembly language subroutine to move the character set from ROM to RAM is shorter and faster than the BASIC routine.

Place short lines together on one line. Each new BASIC line uses three more bytes than the same statement placed in an existing line. Be careful here with GOSUBs, and if . . . next statements.

Many of the programming techniques used in commercial programs are not programming tricks, but good programming practices.

Chapter 6

Tricks with Strings

In the past few chapters, we have created animated scenes with various graphics modes and the player/missile graphics. In most of the programs, we printed the characters on the screen. With the player/missile graphics, we were able to move the character to the left or right by poking a register; we used a machine language subroutine to move the character up or down. Sometimes the movement was smooth, at other times it wasn't.

By placing the characters that form the graphics into strings, the animation that we are trying to create on the screen can often be simplified. The following program uses strings to move the graphics on the screen. It is an animated version of the classic puzzle of the farmer with a fox, a bag of wheat, and a duck. He must take all three across the river, but his boat can only carry one item in addition to himself at a time. If he leaves the duck with the fox, the fox will eat the duck. If the duck is left with the wheat, the duck will eat the wheat. To place the fox in the boat, press the f; press the w for the wheat, and the d for the duck. If you want the farmer to row alone, just press the space bar.

Listing 6-1. The Farmer and the Duck, Fox, and Grain Puzzle

```
10 REM LISTING 6.1
20 REM FARMER AND THE DUCK, FOX, AND GR
AIN PUZZLE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM W$(28),W1$(28),C$(20),TEMP$(28)
50 A=PEEK(106)-8:POKE 204,A:POKE 206,2
24:REM STORE THE BEGINNING OF THE NEW
& OLD CHARACTER SETS
60 FOR X=1 TO 20:READ B:C$(X,X)=CHR$(B
):NEXT X:REM MACHINE LANGUAGE SUBROUTI
NE TO MOVE THE CHARACTER SET TO RAM
70 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
```

Listing 6-1. The Farmer and the Duck, Fox, and Grain Puzzle (continued from page 91)

```
90 Q=USR(ADR(C$)):REM MOVE THE CHARACT
ER SET
100 CB=A*256:FOR Q=24 TO 175:READ C:PO
KE Q+CB,C:NEXT Q:REM MOVE THE NEW CHAR
ACTERS INTO RAM
110 DATA 255,255,255,255,255,255,255,2
55
111 DATA 0,0,192,240,252,255,255,255
112 DATA 0,0,0,0,0,192,240,170
113 DATA 0,0,0,0,0,8,130,170
114 DATA 0,0,0,0,255,63,15,170
115 DATA 0,0,0,0,255,255,245,215
116 DATA 20,20,16,20,223,127,255,170
117 DATA 0,0,0,0,255,255,255,170
118 DATA 0,0,0,0,255,252,240,170
119 DATA 0,0,0,0,0,3,15,255
120 DATA 0,0,63,63,255,255,255,255
121 DATA 4,84,5,5,255,255,255,170
122 DATA 0,20,20,20,255,255,255,170
123 DATA 0,1,5,1,255,255,255,170
124 DATA 80,80,64,64,127,223,245,165
125 DATA 170,170,170,170,170,170,170,1
70
126 DATA 0,0,0,0,20,20,20,20
127 DATA 0,0,0,0,0,1,5,1
128 DATA 0,0,0,0,4,84,5,5
130 DLIST=PEEK(560)+PEEK(561)*256
140 POKE DLIST+3,69:REM ANTIC MODE 5
150 FOR Q=7 TO 28:POKE DLIST+Q,5:NEXT
Q:POKE DLIST+6,6:REM CHANGE THE DISPLA
Y LIST TO ANTIC MODE 5
160 POKE 709,152:POKE 710,198:POKE 711
,20:POKE 82,0
170 POKE 756,A:? "❯":POSITION 20,8:? "
###$%&&&&&&&&&&&&&&&&&&&&&&&&&&&&,-##
##"
180 FOR Q=9 TO 12:POSITION 20,Q:? "222
22222222222222222222222222222222222"
:NEXT Q
190 W$="&&&&&&&&&&&&&&&&&&&&&'()*+&":
REM FARMER UPRIGHT
200 W1$="&&&&&&&&&&&&&&&&&&&&&'*1*+&&"
:REM LEANING FORWARD
210 POSITION 0,1:? "TAKE dUCK fOX wHEA
```

```
T":REM SHOW OPTIONS
220 C$="   534":F=1:F1=0:W=1:W1=0:D=1:
D1=0:POSITION 26,8:? W$:REM PRINT FARM
ER-SET VARIABLES
230 POSITION 21,7:? C$(1,3):POSITION 1
6,8:? C$(4,6):OPEN #1,4,0,"K:":REM PRI
NT POSITIONS OF DUCK-WHEAT-FOX
240 GET #1,B:IF B>127 THEN B=B-128:POK
E 694,0:REM RESTORE FROM INVERSE
250 IF D AND (B=68 OR B=100) THEN C$(6
,6)=" ":W$(26,26)="O":C$(3,3)="4":D1=1
:D=0:GOTO 290:REM DUCK
260 IF F AND (B=70 OR B=102) THEN C$(4
,4)=" ":W$(26,26)=",":F=0:F1=1:C$(1,1)
="5":GOTO 290:REM FOX
270 IF W AND (B=87 OR B=119) THEN C$(5
,5)=" ":W$(26,26)="/":W=0:W1=1:C$(2,2)
="3":GOTO 290
275 IF B=32 THEN 290
280 GOTO 240
290 POSITION 16,8:? C$(4,6):CLOSE #1:W
1$(25,25)=W$(26,26):REM PUT IN OTHER S
TRING
300 FOR Q=1 TO 11:POSITION 26,8:? W1$:
REM MOVE THE BOAT
310 TEMP$=W$(1,2):W$=W$(3,28):W$(27,28
)=TEMP$:REM MOVE THE BOAT TO THE LEFT
320 POSITION 26,8:? W$
330 TEMP$=W1$(1,2):W1$=W1$(3,28):W1$(2
7,28)=TEMP$
340 NEXT Q:IF F1 AND D1 AND W1 THEN PO
SITION 0,1:? "  YOU GOT IT !!     ":GO
TO 540
350 IF F AND D THEN POSITION 0,1:? "
FOX EATS DUCK!!  ":GOTO 540
360 IF D AND W THEN POSITION 0,1:? "
DUCK EATS WHEAT  ":GOTO 540
370 W$(4,4)="*":W1$(3,3)="*":POSITION
26,8:? W$:POSITION 21,7:? C$(1,3):OPEN
 #1,4,0,"K:"
380 GET #1,B:IF B>127 THEN B=B-128:POK
E 694,0:REM RESTORE FROM INVERSE
390 IF D1 AND (B=68 OR B=100) THEN C$(
3,3)=" ":W$(4,4)="O":C$(6,6)="4":D1=0:
```

Listing 6-1. The Farmer and the Duck, Fox, and Grain Puzzle (continued from page 93)

```
D=1:GOTO 430:REM DUCK
400 IF F1 AND (B=70 OR B=102) THEN C$(
1,1)="  ":C$(4,4)="5":W$(4,4)=".":F1=0:
F=1:GOTO 430:REM IT'S THE FOX
410 IF W1 AND (B=87 OR B=119) THEN C$(
2,2)="  ":C$(5,5)="3":W$(4,4)="/":W1=0:
W=1:GOTO 430
415 IF B=32 THEN 430
420 GOTO 380
430 POSITION 21,7:? C$(1,3):CLOSE #1:W
1$(3,3)=W$(4,4):REM PUT IN OTHER STRIN
G
440 FOR Q=1 TO 11:POSITION 26,8:? W$:R
EM MOVE THE BOAT
450 TEMP$(1,2)=W1$(27,28):TEMP$(3,28)=
W1$(1,26):W1$=TEMP$:REM MOVE BOAT TO R
IGHT
460 POSITION 26,8:? W1$
470 TEMP$(1,2)=W$(27,28):TEMP$(3,28)=W
$(1,26):W$=TEMP$
480 NEXT Q
490 IF F1 AND D1 THEN POSITION 0,1:? "
  FOX EATS DUCK!!   ":GOTO 540
500 IF D1 AND W1 THEN POSITION 0,1:? "
  DUCK EATS WHEAT   ":GOTO 540
510 POSITION 16,8:? C$(4,6):W$(26,26)=
"*":W1$(25,25)="*":POSITION 26,8:? W$
530 GOTO 230
540 FOR Q=1 TO 1000:NEXT Q:GOTO 170
```

Line 40 dimensions four strings. The W$s will be the boat in the water. C$ is the machine language subroutine to move the character set from ROM to RAM. TEMP$ contains temporary information for the strings.

Line 50 finds the top of memory available on your system and subtracts 2K from it. This is where the RAM character set will begin. This location is poked into memory location 204. The beginning of the ROM character set is poked into location 206. These two memory locations will be in the machine language subroutine.

Line 60 is the machine language subroutine that moves the character set from ROM to RAM. Be sure that the data in line 70 is entered correctly. It must be typed in exactly or it won't work.

Line 90 uses the machine language subroutine to move the character set from ROM to RAM.

Line 100 reads new characters into the character set. We will be replacing the characters from the pound sign (#) to number 5. See Fig. 6-1 for the new characters. To get the decimal location of the first byte in the RAM character set, we multiply the value of A by 256. Since we want to begin the new characters with the fourth character, we multiply 8 (bytes per character) by

3 (characters to skip). We will begin replacing the characters with byte 24.

Lines 110-128 contain the data used to change the characters. Each line is a different character.

Line 130 finds the beginning of the display list. We will be changing the display list to work in ANTIC 5.

Line 140 changes the fourth byte of the display list to 69. This is the instruction that tells the computer that the next two bytes indicate where the screen memory begins, and the mode of the first line on the screen.

Line 150 changes the rest of the lines in the display list to ANTIC mode 5. The seventh line of the display list is changed to ANTIC 6 (graphics mode 1). This line will contain text.

Line 160 changes the colors that will be used in the program. Location 82 is poked with a zero to change the left margin on the screen. All the lines on the screen are 40 characters wide except the second line. This line is 20 characters wide. This places the margin in the center of the screen. If we don't change the left margin to zero, the computer will skip the screen area that would normally be the left margin. Because it is now in the middle of the screen, there would be a strange empty line down the middle of the screen.

Line 170 pokes location 756 with the new character set location. The screen is cleared and the shore lines and water are printed on the screen. Notice that the shore begins on the left side of the screen and extends to the right margin, but the program tells the computer to begin the print with position 20. All the lines after the second one are 20 characters off.

Line 180 fills the bottom of the screen with blue.

Lines 190-200 place the waves and the farmer in the boat into the two strings. In W$, the farmer will be upright; in W1$, the farmer will appear to be leaning forward.

Line 210 prints the options on the screen. The D in duck, the F in fox and the W in wheat are different colors; these are the keys that will be pressed to place that object into the boat and row it across.

Line 220 places three spaces and the numbers 534 into C$. C$ will not be used any more in the program, so rather than use a different variable, we are reusing C$. The 5, 3, and 4 are the new characters for the fox, the wheat, and the duck. The next six variables will indicate where the fox, the duck, and the wheat are. F, W, and D will be set to 1 when the fox, the wheat, and the duck are on the right side of the screen. The F1, W1, and D1 will be set to one when the fox, the wheat, and the duck are on the left side of the screen. The farmer in the boat is printed on the screen.

Line 230 prints the items on both sides of the screen. When the program begins, the first three elements of C$ will be empty. The fourth through sixth will contain the fox, the wheat, and the duck. The keyboard is opened for input.

Line 240 gets a value from the keyboard. This value will be an ATASCII value for the key pressed. If the value of B is greater than 127, the inverse key has been accidentally pressed. By subtracting 128 from this value, it will be restored to the correct value. By poking location 694 with a zero, the flag is reset for normal characters.

Lines 250-270 check the key that has been entered for one of the three characters. If the variable for that character is a one, the first part of the if . . . then statement is true. We do not have to enter IF D=1, IF D serves the same purpose. If the D is one, the computer will go on the second part of the statement and check to see if the D or d key has been pressed. If it has, the duck is placed into the boat. The redefined character for 0 is the duck. When the boat is on the right side of the screen, that part of the boat is the 26th position in the string. The third character in C$ will be the duck. This element is replaced with the 4, the duck on the ground. The variable for the left side

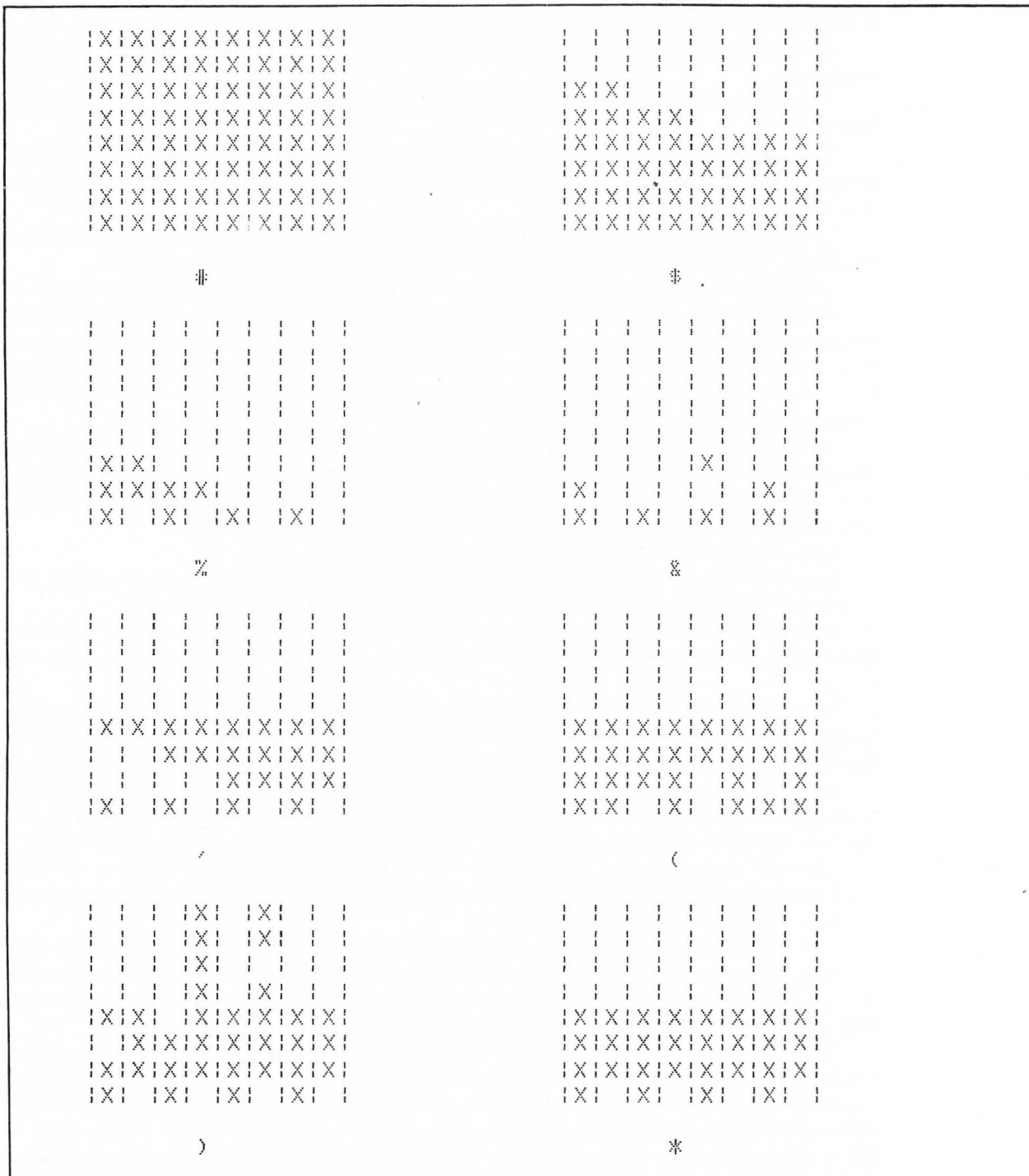
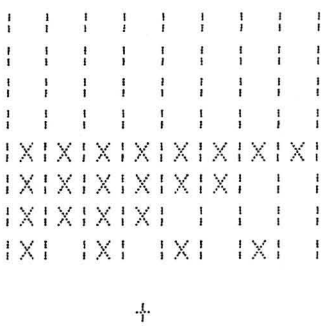
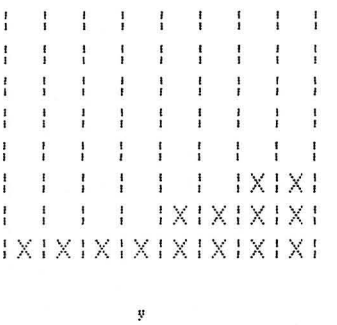


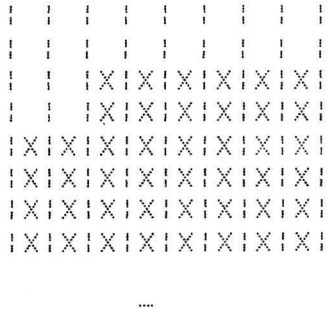
Fig. 6-1. Character set for farmer, duck, fox and grain.



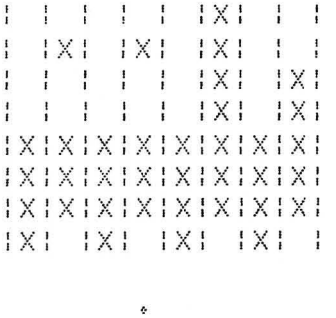
+



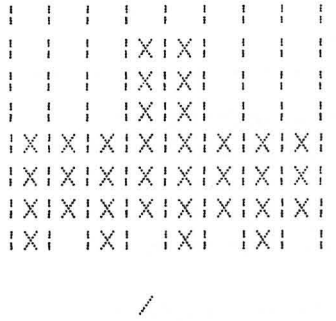
"



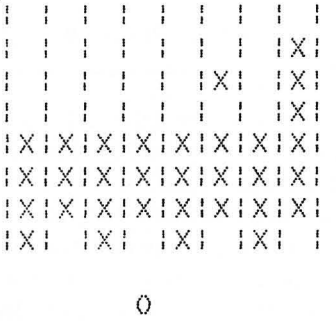
—



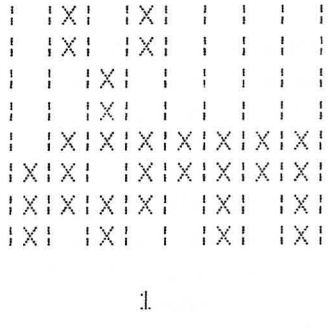
.



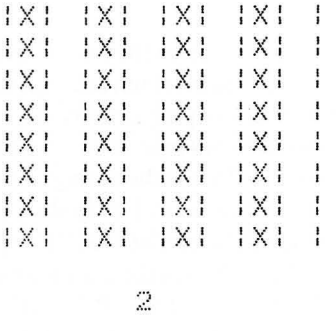
/



0



1



2

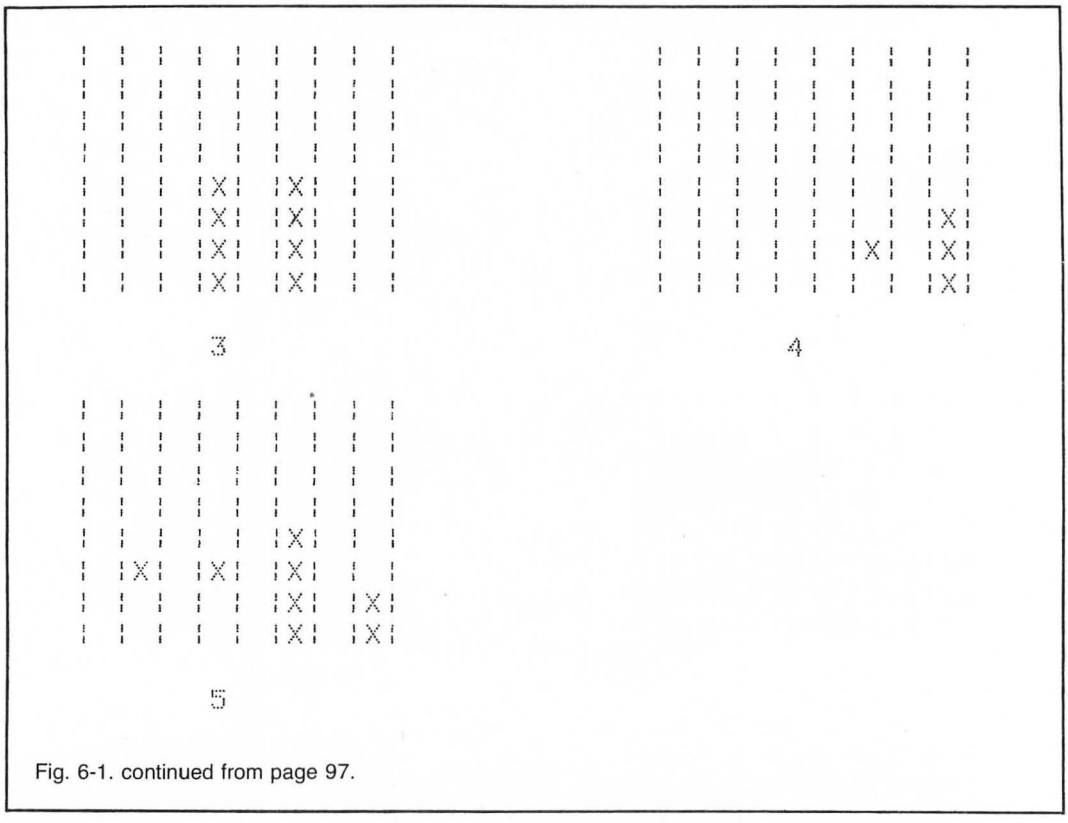


Fig. 6-1. continued from page 97.

of the screen is changed to a 1 and the variable for the right side of the screen is set to zero. The program goes on to line 290. The other two lines are the same. They use the redefined characters for the fox and wheat.

Line 275 checks for the space bar. If the space bar is pressed, the program will go on to the next part of the routine and row the boat across the screen. Only press the space bar when you want the farmer to row the boat with nothing in it.

Line 280 sends the computer back to line 240. The key that has been pressed is not one of the four correct keys, or the character that you are trying to take across the river is already on the other side.

Line 290 reprints the fourth through sixth characters of C$ on the screen. The character that is in the boat will be removed from the shore. The keyboard is closed and the character that is in the boat in W$ is placed in W1$. The boat is one character to the left in W1$.

Lines 300-340 form a for...next loop to move the boat from the right to the left. W1$ is printed on the screen. This is the string with the farmer leaning forward in his boat. The oar is in a slightly different position also. The first two characters of W$ is stored in TEMP$. These two characters will end up at the end of W$. The rest of W$, from the third position to the end is placed in W$. This, in effect, moves the characters up two places. The characters stored in TEMP$ is placed in the last two places of W$. W$ is printed on the screen. The same procedure is repeated with W1, except it is not printed on the screen. W1$ is printed when the routine is repeated. When the loop

ends, W$ will be on the screen. The computer will check the values of F1, D1, and W1. If they are all 1, all three characters are on the left and the game is over. The winning message is printed on the screen.

Line 350 checks the variables F and D. If both of these are a 1, then the fox and duck are together. Since they cannot be together, the message FOX EATS DUCK appears on the screen.

Line 360 checks the values of D and W. If both of these are a 1, the duck and the wheat are together. The message DUCK EATS WHEAT appears on the screen.

Line 370 restores the boat to normal in both strings, prints the empty boat on the screen, and prints the first three characters of C$ on the shore. The character that has been brought over on the boat will now appear on the land.

Line 380 gets a new key from the keyboard. Again, the value is checked to see if the inverse key has been set. If it has, 128 will be subtracted and the flag reset.

Lines 390-410 check the key that has been pressed against the values in the variables. If the correct key has been pressed, but the corresponding variable is a zero, then that character is on the other side of the river and the boat must be rowed back before the character can be placed in it.

Line 415 checks for the space bar. If it has been pressed, the program will go on the routine that moves the boat back across the water.

Line 420 sends the computer back to line 380 for a new input. The key pressed was not an F, D, or W or the space bar.

Line 430 prints C$ on the shore. The character placed in the boat will be removed. The character is placed into W1$. Again, because W1$ is offset by one character, the character that was placed in the boat is placed one position to the left in W1.

Lines 440-480 move the boat back across the screen. The lines are similar to the ones that moved the boat across the screen the first time. This time the last two characters are placed in the first two positions of TEMP$. The first through 26th characters of W1$ are placed in the 3rd through 28th elements of TEMP$. Then the contents of TEMP$ are placed in W1$. The same procedure is used to move the boat in W$.

Lines 490-500 check to see if the fox is left with the duck, or the duck is left with the wheat.

Line 510 prints the fourth through sixth characters of C$ onto the shore. The new character will appear here now. The boat will be restored, and the empty boat will be printed on the screen. The program continues until the characters are all on the left side of the screen.

Line 540 is a timing subroutine. It is used to give you a chance to read the message on the screen.

To restore the screen, press the system reset key.

MACHINE LANGUAGE SUBROUTINES

Until now, you have placed machine language subroutines into a string by setting a position of the string equal to the character string of a number. Every program that moves the character set to RAM, or moves a player/missile up or down used the same machine language subroutine.

The following two programs place a machine language subroutine in a string. The program then prints the string on the screen with a line number in front of it. By moving the cursor over the line and pressing return, we place the new line with the subroutine into the program. This line can then be stored on disk for future use. When you want this subroutine in a program, you do not have to use a for . . . next loop with the data lines. The line with the characters already in the string can be used.

Listing 6-2. Move Character Set

```
10 REM LISTING 6.2
20 REM MOVE CHARACTER SET
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM C$(20)
50 FOR X=1 TO 20:READ B:C$(X,X)=CHR$(B
):NEXT X
60 DATA 104,162,4,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
70 PRINT "}";50;" C$=";CHR$(34);C$;CHR
$(34):? 60:? 70
```

Line 40 sets aside 20 bytes for the machine language subroutine.

Line 50 reads the data into the string. When it finishes, each byte of the string will be an instruction of the subroutine.

Line 70 clears the screen. The number 50 and the string will be printed on the screen. The numbers 60 and 70 will also be printed on the screen.

Now move the cursor to the top of the screen and press the return key three times. Line 50 will be replaced with the string that contains the subroutine. Lines 60 and 70 will be erased. To use this subroutine, subtract 2K from the amount of memory available in your system (A=PEEK(106)−8). Store the value of A in location 204 (POKE 204,A). Store the beginning address of the ROM character set in location 206 (POKE 206,224). To execute the subroutine use the USR command—Q = USR(ADR(C$)). This subroutine will move the ROM character set into RAM.

Listing 6-3. Move Player/Missile Up/Down

```
10 REM LISTING 6.3
20 REM MOVE PLAYER/MISSILE UP/DOWN
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM UP$(13),DOWN$(13)
50 FOR X=1 TO 13:READ B:UP$(X,X)=CHR$(
B):NEXT X
60 DATA 104,160,0,200,177,205,136,145,
205,200,208,247,96
70 FOR X=1 TO 13:READ B:DOWN$(X,X)=CHR
$(B):NEXT X
80 DATA 104,160,255,136,177,205,200,14
5,205,136,208,247,96
90 ? "}CLEAR}";50;" UP$=";CHR$(34);:FO
R X=1 TO 13:? CHR$(ASC(UP$(X,X)));:NEX
T X:? CHR$(34)
100 ? 60;" DOWN$=";CHR$(34);:FOR X=1 T
O 13:IF ASC(DOWN$(X,X))>252 THEN ? CHR
$(27);
```

```
110 ? CHR$(ASC(DOWN$(X,X)));:NEXT X:?
CHR$(34);? 70:? 80:? 90:? 100:? 110
```

Line 40 sets aside 13 bytes each for the two machine language subroutines.

Line 50 reads the data from line 60 into UP$. UP$ will contain the machine language subroutine to move the players up.

Line 70 reads the data from line 80 into DOWN$. This is the machine language subroutine to move the players down.

Line 90 clears the screen and prints the two strings on the screen with line numbers before them. The numbers 70, 80, 90, 100, and 110 also appear on the screen. Move the cursor to the top of the screen and press the return key 7 times. Lines 50 and 60 will be replaced with the new lines that contain the machine language subroutines. Lines 70, 80, 90, 100, and 110 will be erased from the program.

To use these subroutines, store the high order address of the player to be moved in location 206 and the low order in location 205. Executing either subroutine with the USR command will move the player up or down.

When storing a machine language subroutine in a string, make sure that the subroutine contains relative branches. That is jumps that depend on a byte count instead of a firm address. The string can be placed anywhere in memory. The location of the string will vary from program to program. If the machine language subroutine contained a set jump, for example, jump to address 2048, the program would always jump to that address. If the correct instruction was there, the routine would work properly. If it wasn't, the program would crash. By using a relative jump, for example, Branch Not Equal, the program will be directed to a location by bytes (forward 8 bytes, backwards 3 bytes, etc.). This routine could be located anywhere since the same instruction will always appear 8 bytes after or 3 bytes before the current instruction.

RELOCATING THE STRINGS

In the last chapter, tables for the variables, their names, and the string/array area were listed. Knowing where the computer looks to find out where a string is located enables us to be able to relocate strings to areas that are more suitable for our program needs.

The two tables that we will be using in the next program are the variable value table and the string/array area. By changing the offset values in the variable value table, we can make the computer "think" that the string is located elsewhere in memory. By changing the values in the fifth and sixth locations in the variable value table, we can change the length of the string.

In effect, we can make the screen one string, and manipulate it as such. We could even store a program in a string if we had reason to. One example of relocating the strings is in the next program.

STRINGS AND PLAYER/MISSILE GRAPHICS

The player/missile graphics have provisions for moving the player/missiles to the right or left with a simple poke command. But, in order to move the players up or down, we have to rely on machine language subroutines. If the player was extremely long (tall), the movement could look a little jumpy. In the next program, which simulates a slot machine, we will move the players up by manipulating a string.

Listing 6-4. Player/Missile Strings

```
10 REM LISTING 6.4
20 REM PLAYER/MISSILE STRINGS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM PL1$(1),PL2$(1),PL3$(1),TEMP$(1
),JCKPT$(8)
50 GRAPHICS 4:DLIST=PEEK(560)+PEEK(561
)*256:REM GET THE DISPLAY LIST
60 POKE DLIST+45,70:POKE DLIST+48,6:PO
KE DLIST+49,6:POKE DLIST+50,6:REM CHAN
GE TEXT WINDOW TO GR. 1
70 COLOR 1:PLOT 50,37:DRAWTO 50,5:DRAW
TO 24,5:POSITION 24,37:POKE 765,1:XIO
18,#6,0,0,"S:":REM DRAW SLOT MACHINE
80 COLOR 0:FOR X=10 TO 13:PLOT 26,X:DR
AWTO 32,X:PLOT 34,X:DRAWTO 40,X:PLOT 4
2,X:DRAWTO 48,X:NEXT X:REM MAKE WINDOW
90 VART=PEEK(134)+PEEK(135)*256:STTAB=
PEEK(140)+PEEK(141)*256:REM GET THE VA
RIABLE & STRING TABLES
100 POKE 559,46:REM SET FOR DOUBLE LIN
E P/M RESOLUTION
110 PMBASE=PEEK(106)-8:POKE 54279,PMBA
SE:REM SET P/M GRAPHICS 2K ABOVE END O
F MEMORY
120 PMBASE=PMBASE*256:REM FULL ADDRESS
 OF P/M GRAPHICS
130 X=0:FOR Z=512 TO 768 STEP 128:REM
BEGINNING OF EACH PLAYER
140 STROFFSET=PMBASE+Z-STTAB:REM CHANG
E THE OFFSET VALUE FOR THE STRINGS
150 STRVAL3=INT(STROFFSET/256):REM HIG
H ORDER ADDRESS OF OFFSET
160 STRVAL4=STROFFSET-256*STRVAL3:REM
LOW ORDER ADDRESS OF OFFSET
170 POKE VART+2+X,STRVAL4:REM THIRD BY
TE FOR STRING VARIABLE IN VARIABLE TAB
LE
180 POKE VART+3+X,STRVAL3:REM FOURTH B
YTE FOR STRING VARIABLE IN VARIABLE TA
BLE
190 POKE VART+4+X,128:REM MAKE STRING
128 BYTES LONG
200 POKE VART+5+X,0
210 POKE VART+6+X,128:REM SET LENGTH T
```

```
0 128
220 POKE VART+7+X,0:X=X+8:NEXT Z:REM I
NCREMENT X FOR NEXT STRING
230 PL1$(1)="}},}":PL1$(128)="}},}":PL1$
(2)=PL1$:REM CLEAR THE STRING
240 X=36:FOR Z=1 TO 54:READ B:PL1$(X,X
)=CHR$(B):X=X+1:NEXT Z:JCKPT$=PL1$(73,
80):REM GET SYMBOLS
250 DATA 255,0,32,16,16,40,108,108,0
260 DATA 255,0,0,24,60,126,60,24,0
270 DATA 255,0,0,0,60,126,60,0,0
280 DATA 255,0,48,8,24,60,126,60,0
290 DATA 255,0,0,0,126,126,0,0,0
300 DATA 255,0,16,16,56,124,124,124,0
310 PL2$=PL1$:PL3$=PL1$
320 POKE 53277,3:POKE 623,4:REM ENABLE
 P/M GRAPHICS - ESTABLISH PRIORITIES
330 POKE 704,90:POKE 705,120:POKE 706,
30:POKE 708,150:REM SET COLORS
340 MONEY=100:POKE 53248,104:POKE 5324
9,119:POKE 53250,135:REM FILL THE WIND
OWS - MOVE THE PLAYERS ON THE SCREEN
350 ? "}CLEAR}YOU HAVE $";MONEY:? "PRE
SS any KEY";:REM PRINT THE MESSAGE
360 IF PEEK(764)=255 THEN 360:REM WAIT
 FOR A KEY TO BE PRESSED
370 MONEY=MONEY-1:POKE 764,255:T=INT(R
ND(1)*15)+5:REM PICK NUMBER OF SPINS F
ROM 5 - 14
380 FOR X=1 TO T*9:TEMP$=PL3$(36):PL3$
(36,88)=PL3$(37,89):PL3$(89)=TEMP$:REM
 ROTATE THE STRINGS
390 FOR Z=1 TO 2:TEMP$=PL2$(36):PL2$(3
6,88)=PL2$(37,89):PL2$(89)=TEMP$:NEXT
Z
400 FOR Z=1 TO 3:TEMP$=PL1$(36):PL1$(3
6,88)=PL1$(37,89):PL1$(89)=TEMP$:NEXT
Z:NEXT X
410 T=INT(RND(1)*6)+6:REM PICK NUMBER
OF TURNS FROM 1 - 6
420 FOR X=1 TO T*9:TEMP$=PL2$(36):PL2$
(36,88)=PL2$(37,89):PL2$(89)=TEMP$:REM
 ROTATE THE FIRST TWO STRINGS
430 FOR Z=1 TO 2:TEMP$=PL1$(36):PL1$(3
```

Listing 6-4. Player/Missile Strings (continued from page 103)

```
550 ? "PRESS ANY KEY TO PLAY AGAIN";:R
EM NEXT PLAYER
560 IF PEEK(764)=255 THEN 560:REM WAIT
 FOR A KEY
570 GOTO 340:REM PRESS SYSTEM RESET TO
 END GAME
6,88)=PL1$(37,89):PL1$(89)=TEMP$:NEXT
Z:NEXT X
440 T=INT(RND(1)*6)+6:REM PICK NUMBER
OF TURNS FROM 1 - 6
450 FOR X=1 TO T*9:TEMP$=PL1$(36):PL1$
(36,88)=PL1$(37,89):PL1$(89)=TEMP$:NEX
T X:REM LAST STRING
455 REM CHECK FOR PAYOFF
460 IF PL3$(37,44)=PL2$(37,44) AND PL3
$(37,44)=PL1$(37,44) AND PL3$=JCKPT$ T
HEN MONEY=MONEY+500:S=75:GOTO 510
470 IF PL1$(37,44)=PL2$(37,44) AND PL3
$(37,44)=PL1$(37,44) THEN MONEY=MONEY+
100:S=100:GOTO 510
480 IF PL1$(37,44)=PL2$(37,44) THEN MO
NEY=MONEY+50:S=125:GOTO 510
490 IF PL1$(37,44)=PL3$(37,44) THEN MO
NEY=MONEY+5:S=150:GOTO 510
500 S=200:REM NO WIN
510 SOUND 0,S,10,10:FOR X=1 TO MONEY:N
EXT X:SOUND 0,0,0,0:REM MAKE SOUND TO
INDICATE WINNING
520 IF MONEY>0 AND MONEY<1000 THEN 350
:REM CHECK FOR END OF GAME
530 ? "}CLEAR}":IF MONEY=0 THEN ? "you
 lose":GOTO 550:REM LOST YOUR SHIRT
540 IF MONEY=1000 THEN ? "YOU WIN":REM
 BROKE THE BANK
550 ? "PRESS ANY KEY TO PLAY AGAIN";:R
EM NEXT PLAYER
560 IF PEEK(764)=255 THEN 560:REM WAIT
 FOR A KEY
570 GOTO 340:REM PRESS SYSTEM RESET TO
 END GAME
```

Line 40 sets aside one byte for the strings that will be used for the players. TEMP$ will hold one byte during the string manipulation. JCKPT$ is the jackpot character. In order for this procedure to work correctly, PL1$, PL2$, and PL3$ must be the first three variables in the

variable name table. Before typing this program in, turn off the computer; then turn it back on. Begin typing in this program. If the computer is given a variable before the PL strings, it will store that variable in the table. This will set the PL strings off and the program will not work correctly.

Line 50 sets the graphics to 4. We will only need two colors for the slot machine, the background color and the machine's color. The beginning of the display list is stored in DLIST.

Line 60 changes the text window from graphics mode 0 to graphics mode 1. When we are using a graphics mode with a text window, the display list will have another load command after the proper number of mode commands. This command tells the computer to start a new screen display in graphics mode 0. We will change that to graphics mode 1 by poking it with a 70. The three lines that would display graphics mode 0 are changed to graphics mode 1 by poking three more locations in the display list with a 6. (Graphics mode 1 is ANTIC 6.) Now the text window in the program will have large print.

Line 70 uses the XIO command to draw a slot machine on the screen.

Line 80 uses color 0 (black) to erase part of the slot machine for the windows.

Line 90 stores the location of the variable table in VART and the address of the string/array are in STTAB. These two values will be used to relocate the strings.

Line 100 sets the player/missile for double line resolution.

Line 110 sets the player/missile graphics 2K before the end of memory. The top of memory is stored in location 106. By subtracting 8 from it, we are subtracting 2K of memory. The beginning of the player/missile area is stored in 54279.

Line 120 multiplies the beginning address by 256 to arrive at the decimal location of the player/missiles. The number that is stored in PMBASE before it is multiplied is the high order address.

Lines 130-220 relocate the three strings that will be used for the player/missile graphics. The variable X is set to 0 (zero). This is the variable number or location in the variable table. The first player is 512 bytes past the address of the beginning of the player/missile area. Each player is 128 bytes apart, so the for . . . next loop uses a step 128. The string offset is stored in the variable STROFFSET. This is the value that the computer will add to the address of the string/array area to arrive at the location of the string. When the strings are not being relocated, the first string will have zeros as the offset since the first string will begin at the first byte of the string/array area. The second string's offset will be one more than the length of the first string. The string should begin with the first byte of the player. The value of Z will be added to PMBASE. This is where the player begins in the player/missile graphics area. The value of STTAB must be subtracted because the computer will "add" this value back when it is looking at the string. The value that you arrive at is the new offset for the string. This value is greater than 255, so it will have to be divided into two bytes. The high order address is the integer (whole number) of the offset divided by 256. The low order address is arrived at by subtracting the high order integer times 256 from the address. These two bytes are stored in the third and fourth bytes of the variable table for that string. The next two bytes in the variable table indicate the amount of memory set aside for the string. We want 128 bytes for each string, so the low order is poked with 128 and the high order with a 0 (zero). The last two bytes set the length of the string. We want the string to be 128 bytes long, so these two locations will be poked with a 18 and 0 (zero). The variable X is incremented by 8 because the information for the next string will begin 8 bytes down from the beginning of the information for this string. This routine is repeated two more times. When it is finished, the first three strings in the variable table will be 128 bytes long and their location will be the player/missile graphics area.

Line 230 clears the first string.

Line 240 reads the data in the next 6 lines and stores it in the string. This data will draw the cherry, orange, lemon, apple, gold bar, and bell into the player/missile graphics are. The string JCKPT is set to the 5th character in the string.

Line 310 sets the other two strings.

Line 320 pokes location 53277 with a 3 to enable the players, and location 623 is poked with a 4. This establishes the priorities of the players and the graphics on the screen. If the priorities were not set, some of the players would appear over the machine instead of inside it.

Line 330 sets the colors of the slot machine and the three wheels.

Line 340 sets the variable MONEY to 100. This is the amount of money you have to start the game. The three players are moved on the screen by poking their locations into the correct registers.

Line 350 prints a message in the text window.

Line 360 checks location 764 for a value other than 255. When the value of this location changes, then a key has been pressed.

Line 370 subtracts one dollar from the amount of money left. The key location is cleared, and a random number from 5 to 19 is chosen. This number determines the number of times the rightmost wheel will spin.

Lines 380-400 make the wheels spin. This works on the same principle as the boat in the first program of this chapter. The top character of the string (in this case it's in the 36th location) is removed from the string and stored in a temporary location. The rest of the string is moved up one byte. The character that is being stored in the temporary location is moved to the end of the string. Each of these strings is a different player. The rightmost wheel is rotated once, the middle one twice and the leftmost three times each time the program executes the loop. This gives the illusion of the wheels spinning at different speeds.

Lines 410-450 spin the middle and leftmost wheel after the right wheel stops. Each wheel is spun a few more times after one wheel stops.

Lines 460-500 check for a payoff. If all three characters shown on the machine are the same and they are the bars, the jackpot is won. 500 is added to the amount stored in MONEY. If the three are the same, but not the jackpot, 100 is added to the amount stored in MONEY. If the first two characters are a match, 5 is added. If there are no matches, nothing is added to the amount in MONEY.

Line 510 makes a sound to indicate the win. The variable S is set to a different value depending on the amount of MONEY won. This line generates the sound.

Line 520 checks the variable MONEY to see if you can still play.If there is no MONEY, or the value of MONEY is greater than 1000, the game ends.

Line 530 clears the screen and prints you lose if you have no money left.

Line 540 prints you win if the amount in MONEY is greater than 1000.

Line 550 prints the message on the screen.

Line 560 waits for a key to be pressed. The program will loop here until a key is pressed, if the system reset key is pressed, the program will end.

Line 570 sends the program back to line 340 to play again.

Chapter 7
Display List Interrupts

An interrupt is a subtle way to get the computer to do a task while it appears to be doing something else. When you are working on a program with your ATARI, the computer looks like it is just sitting there waiting for you to enter your new line or command. What is actually happening is the computer is very busy maintaining several areas. The ANTIC chip keeps the information on the screen; the clock keeps time; and the computer keeps checking to see if it's time to start the active mode. When you press a key, you are interrupting the 6502. It now checks which key was pressed, and sends that information over to the ANTIC chip so that it can be displayed on the screen. Because of the speed at which the computer works, this appears to be done instantaneously. The computer is capable of handling more functions if it is interrupted at the proper times.

HANDLING AN INTERRUPT

One of the best times to interrupt the computer is when it is drawing on the screen. To draw an image on the screen, the computer must be synchronized with the raster scan of the television set. This happens within the computer and we do not have to be concerned with it. When the raster scan begins to draw a picture on the screen, it begins with the upper left corner and goes across the screen to the upper right corner. When it is finished with the line, it shuts itself off, retraces its line, then drops down one line and turns itself back on. It continues to draw lines, turn itself off, retrace, drop down, and turn itself back on until it reaches the bottom of the screen. There it will shut itself off and return to the top of the screen, where it will begin to draw all over again.

The period of time during which the raster is turned off and is retracing the line is called a horizontal blank. The period of time needed for the raster to go back to the top of the screen is called the vertical blank. During this time we can interrupt the computer and have it do something that we want it to do: something that may not be possible in BASIC or machine language alone.

There are certain registers or memory locations in your ATARI that seem to have a double in ROM. The color registers are one set of these registers. There are 9 different locations to store the color values in RAM. There are also 9 color locations in the ROM or Operating System. If you poked a value into the RAM locations, you would change the color of the character on the screen. If you poked a value into the corresponding ROM location, nothing would happen.

ANTIC draws characters on the screen based on the colors in the ROM locations (these

Table 7-1. Hardware Registers and Shadows.

Register description	hardware address	shadow address
Character mode	54273	755
Character base	54281	756
Color background	53274	712
Color register 0	53270	708
Color register 1	53271	709
Color register 2	53272	710
Color register 3	53273	711
Color reg/player 0	53266	704
Color reg/player 1	53267	705
Color reg/player 2	53268	706
Color reg/player 3	53269	707
Display list high	54275	561
Display list low	54274	560

locations are also called the hardware registers). Each time the vertical blank occurs, 60 times every second, the computer looks at the colors in the RAM locations and stores them in the hardware registers. If you poked a color value into a hardware register, it will be replaced within a 60th of a second! Table 7-1 shows a list of hardware registers. The second location is called the shadow register. This is the register that contains the value that will be placed in the hardware register by the computer during the vertical blank.

There is a way to change a value in the hardware register without interference from the shadow register: interrupt the display list and change the value during a horizontal blank.

WRITING SERVICE ROUTINES

During the period of time that the raster scan is retracing itself, the computer is free to do other things, like change colors, character sets, or other images on the screen. To do this, however, the computer must be told to watch for an interrupt. In the following program, the

computer is told that there will be a display list interrupt. The correct bit is set for an interrupt at the end of a line, and the colors in color registers are changed.

The only problem with using a display list interrupt to change the color value is that the character will always be one color above the line and the other color below the line. If the character travels above the line it will be changed to the original color.

Listing 7-1. Color Service Routine

```
10 REM LISTING 7.1
20 REM COLOR SERVICE ROUTINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
50 GRAPHICS 18
60 DL=PEEK(560)+PEEK(561)*256:DL=DL+10
:REM THE 5TH LINE ON THE SCEEN
70 POKE DL,PEEK(DL)+128:REM set the 8t
h bit to 1 by addins 128 to the peek o
f that location
80 FOR ML=1536 TO 1564:READ Q:POKE ML,
Q:NEXT ML:REM POKE THE MACHINE LANGUAG
E SUBROUTINE INTO FREE RAM
90 POKE 512,0:POKE 513,6:POKE 54286,19
2:REM ADDRESS OF MACHINE LANGUAGE SUBR
OUTINE - ACTIVATE INTERRUPT
100 DATA 72,138,72,152,72,162,100,160,
8,169,88,141,10,212,141,24,208,142,23,
208,140,22,208,104,168,104,170,104,64
120 POSITION 2,1:? #6;"BLUE","green"
130 POSITION 2,3:? #6;"red","YELLOW"
160 POSITION 2,7:? #6;"PINK","purple":
POSITION 7,9:? #6;"GREY"
170 GOTO 170
```

Line 50 changes the graphics mode to 18—mode 2 without the text window.

Line 60 stores the beginning of the display list in variable DL. The address of the beginning of the display list is stored in locations 560 and 561. To arrive at this address, the contents of memory location 561 must be multiplied by 256 and the contents of memory location 560 must be added to this product. Add 10 to this address to point to the middle of the display list.

Line 70 changes the display list. By setting the high order bit of a display list value to 1, we tell ANTIC that after it draws or writes this line onto the screen, there will be an interrupt. Since we may not be sure what the value might be at this particular location, the simplest way to set the bit is to add 128 to the value of this location.

Line 80 reads the data from line 100 and pokes it into the free RAM beginning at location 1536. See Table 7-2 for the assembly language listing of the subroutine.

Line 90 stores the address of the beginning of the interrupt routine at memory locations 512-513. The address, in this case 1536 is divided by 256. The high order address is the integer part of the quotient. 1536/256=6. There is no remainder. The 6 is stored in memory location 513.

Table 7-2. Service Routine to Change Colors.

```
decimal code          assembly language listing
     72          PHA        ;Push accumulator on the stack.
    138          TXA        ;Transfer index X to accumulator.
     72          PHA        ;Push accumulator on stack
                            (contents transferred from index X)
    152          TYA        ;Transfer index Y to accumulator.
     72          PHA        ;Push accumulator on stack
                            (contents transferred from index Y)
    162          LDX #100   ;Load index X with 100.
    100
    160          LDY #8     ;Load index Y with 8.
      8
    169          LDA #88    ;Load accumulator with 88.
     88
    141          STA 54282  ;Store the contents of the
     10                     accumulator at this address and
```

If there was a remainder, it would be stored in location 512. Since there is none, a zero is stored there. Memory location 54286 is poked with 192. This is a hardware address that can enable the interrupt for the display list interrupt and the vertical blank interrupt. If it is not enabled, ANTIC would ignore the code in the display list that tells it that this is the place where it should execute the interrupt routine.

Lines 120-160 print on the screen. The first four words—blue, green, red, and yellow use the colors that are preset by the operating system. Be sure that BLUE is in uppercase and inverse, and red is lowercase inverse. Each word will appear in its color. The three words on the bottom will appear in new colors. PINK is uppercase inverse, and appears pink on the screen. Purple and GREY were previously green and YELLOW. The interrupt routine changed the color code in the

```
212                 wait for the horizontal sync.
141         STA 53272 ;Store in color resister.
 24
208
142         STX 53271 ;Store index X in color resister.
 23
208
140         STY 53270 ;Store index Y in color resister.
 22
208
104         PLA       ;Remove value from stack.
168         TAY       ;Transfer it to index Y.
104         PLA       ;Remove another value from stack.
170         TAX       ;Transfer it to index X.
104         PLA       ;Remove value from stack.
 64         RTI       ;Return from interrupt.
```

hardware register during a vertical blank. The operating system replaced the color with the color code from the RAM shadow address during the horizontal blank. Everytime something is printed on the top half of the screen, it will appear in the color set by the operating system, or the color that was placed in the shadow registers under program control. The colors on the bottom half of the screen will be the colors forced into the hardware registers during the interrupt.

End this program by pressing the system reset key.

Changing the colors in a program is only one possible use of interrupts. Any feature that has both a shadow or RAM address and a hardware address can have multiple uses with the interrupt. In the next program we will use two character sets in the same program. The standard character set will be displayed at the top of the screen, and the new characters will be used at the bottom.

Listing 7-2. Double Character Sets

```
10 REM LISTING 7.2
20 REM DOUBLE CHARACTER SETS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0
50 CB=PEEK(106)-8:POKE 204,CB:POKE 206
,224:REM PLACE NEW CHARACTERS 2K ABOVE
 END OF MEMORY
60 FOR ML=1536 TO 1555:READ Q:POKE ML,
Q:NEXT ML:REM MOVE THE MACHINE CODE IN
TO RAM
70 Q=USR(1536):REM NOW RUN IT
80 DATA 104,162,2,160,0,177,205,145,20
3,200,208,249,230,206,230,204,202,208,
242,96
90 C1=CB*256+263:FOR X=C1 TO C1+207:RE
AD Q:POKE X,Q:NEXT X
100 DATA 0,126,102,102,126,102,102,102
101 DATA 0,124,102,102,124,102,102,124
102 DATA 0,124,100,96,96,100,100,124
103 DATA 0,124,102,102,102,102,102,124
104 DATA 0,124,100,96,120,96,100,124
105 DATA 0,124,100,96,120,96,96,112
106 DATA 0,124,100,96,108,100,100,124
107 DATA 0,102,102,102,126,102,102,102
108 DATA 0,60,24,24,24,24,24,60
109 DATA 0,30,12,12,12,76,76,124
110 DATA 0,102,100,104,124,102,102,102
111 DATA 0,120,48,48,48,50,50,126
112 DATA 0,55,90,90,90,90,90,90
113 DATA 0,94,102,102,102,102,102,102
114 DATA 0,60,102,102,102,102,102,60
115 DATA 0,124,102,102,124,96,96,96
116 DATA 0,60,102,102,102,118,110,102
117 DATA 0,124,102,102,124,102,102,102
118 DATA 0,124,100,96,124,12,76,124
119 DATA 0,124,90,90,24,24,24,60
120 DATA 0,102,102,102,102,102,102,126
121 DATA 0,102,102,102,102,100,104,112
122 DATA 0,90,90,90,90,90,90,108
123 DATA 0,122,90,92,24,58,90,94
124 DATA 0,102,102,102,102,124,6,124
125 DATA 0,124,70,76,24,50,98,124
160 DL=PEEK(560)+PEEK(561)*256:DL=DL+1
```

```
5:REM ADD SERVICE ROUTINE IN MIDDLE OF
 DISPLAY LIST
170 POKE DL,PEEK(DL)+128:REM SET BIT F
OR INTERRUPT
180 FOR ML=1536 TO 1546:READ Q:POKE ML
,Q:NEXT ML:POKE 1538,CB:REM POKE THE M
ACHINE LANGUAGE SUBROUTINE INTO RAM
190 POKE 512,0:POKE 513,6:POKE 54286,1
92:REM ADDRESS OF MACHINE LANGUAGE SUB
ROUTINE - ACTIVATE INTERRUPT
200 DATA 72,169,152,141,10,212,141,9,2
12,104,64
210 POSITION 9,2:? "STANDARD CHARACTER
S"
220 POSITION 13,15:? "BOLD FACE"
300 GOTO 300
```

One of the first tricks that everyone learns is how to turn the screen upside-down by poking 755 with a 4. (If you've never tried it, do it now with the direct command:

POKE 755,4

(If you have a program listed on the screen, it will look most impressive!)

This is fine if you want everything to be upside down. But, maybe, you want just one or two lines in the middle of the screen turned for a mirror effect. Again, the interrupt routines can be used to do it right on cue!

Listing 7-3. Mirror Images Routine

```
10 REM LISTING 7.3
20 REM MIRROR IMAGES ROUTINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0
50 DL=PEEK(560)+PEEK(561)*256:DL=DL+15
:REM THE 5TH LINE ON THE SCEEN
70 POKE DL+X,PEEK(DL)+128:REM set the
8th bit to 1 by addins 128
80 FOR ML=1536 TO 1546:READ Q:POKE ML,
Q:NEXT ML:REM POKE THE MACHINE LANGUAG
E SUBROUTINE INTO FREE RAM
90 POKE 512,0:POKE 513,6:POKE 54286,19
2:REM ADDRESS OF MACHINE LANGUAGE SUBR
OUTINE - ACTIVATE INTERRUPT
100 DATA 72,169,4,141,10,212,141,1,212
,104,64
```

Line 40 sets the graphics mode to 0. This command will make the computer reset the entire display list. By having the display list reset, we can run this program several times. If the display list was not reset, line 70 would produce an error message.

Line 50 finds the beginning of the display list and stores this address in the variable DL. We add 15 to this address so that we will be working with the middle of the display list.

Line 70 sets the eighth bit of the byte in the display list. By adding 128 to the value found at this location, we set only one bit without disturbing the original value.

Line 80 reads the machine language subroutine that inverts the letters.

Line 90 pokes the address of the machine language subroutine into memory locations 512 and 513. 192 is poked into location 54286 to tell the computer that an interrupt will be generated.

Run the program. Now list the program. The listing on the top half of the screen will be correct. The listing on the bottom half of the screen will be inverted. Press the system reset key to turn the screen back to normal.

PRECISE TIMING

Another use for a service routine is for precise timing. The routing will be executed every time the computer comes to that line in the display list. Since the display list is synchronized with the raster scan on the television, the routine will be executed every 1/60th of a second. By placing a counter in the routine, a character can be moved, a sound can be made, or a color changed precisely on schedule.

In the following program, the neon sign is kept flashing with an interrupt service routine.

Listing 7-4. Precise Timing

```
10 REM LISTING 7.4
20 REM PRECISE TIMING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1
50 DL=PEEK(560)+PEEK(561)*256:DL=DL+6:
REM THE 2ND LINE ON THE SCEEN
70 POKE DL,PEEK(DL)+128:REM set the 8t
h bit to 1 by addins 128
80 FOR ML=1536 TO 1558:READ Q:POKE ML,
Q:NEXT ML:REM POKE THE MACHINE LANGUAG
E SUBROUTINE INTO FREE RAM
90 POKE 206,0:REM SET COUNTER AND DATA
 TO 0
100 POSITION 0,0:? " ":COLOR 42:PLOT 3
3,5:DRAWTO 33,18:DRAWTO 5,18:DRAWTO 5,
5:DRAWTO 33,5
110 COLOR 170:PLOT 34,4:DRAWTO 34,19:D
RAWTO 4,19:DRAWTO 4,4:DRAWTO 34,4
120 COLOR 42:PLOT 35,3:DRAWTO 35,20:DR
AWTO 3,20:DRAWTO 3,3:DRAWTO 36,3
130 POSITION 13,10:? "BEDDY-BY MOTEL"
```

```
140 POSITION 10,12:? "COLOR TV - WATER
 BED"
150 POSITION 15,14:? "LOW PRICE"
190 POKE 512,0:POKE 513,6:POKE 54286,1
92:REM ADDRESS OF MACHINE LANGUAGE SUB
ROUTINE - ACTIVATE INTERRUPT
200 DATA 72,165,19,240,16,169,255,133,
19,165,206,9,1,133,206,141,10,212,141,
1,212,104,64
210 GOTO 210
```

Line 40 sets the grapics mode to 0. This must be done at the beginning of the program because you will be adding a value to an existing value in the display list. By setting the graphics mode at the beginning of the program, the computer reinstates the original value into the display list. If this wasn't done, and the program was rerun, the computer would be adding a value to the value that was changed in the previous run. Poking 752 with a 1 removes the cursor from the screen.

Line 50 finds the beginning of the display list and sets the variable DL to the 2nd line or the 7th value in the display list.

Line 70 gets the value from the display list and adds 128 to it. The new value is poked back into the display list. This is the line that the graphics were set back to zero for.

Line 80 reads the machine language subroutine into memory. This is the routine that the computer will execute during the interrupt.

Line 90 pokes the memory location 206 with a zero. We will be using this location as a counter for this program.

Line 100 and 120 use the position command to erase the cursor that was left there by the graphic 0 command. The plot and DRAWTO commands draw a rectangle on the screen. The 42 after COLOR is the character that will be drawn. In this case, it will be an asterisk (*).

Line 110 draws another rectangle between the first and second rectangle.

Lines 130-150 print the message inside the sign. The words COLOR, WATER, and LOW PRICE are in inverse video.

Line 190 tells the computer where in memory the service routine is located and tells the computer that there will be an interrupt.

Line 200 contains the data for the machine language subroutine. This subroutine uses the clock to find out when the words should be turned on and off. See Table 7-3 for the assembly language listing for this program.

To end this program, press the system reset key.

Listing 7-4A. Precise Timing—Second Method

```
10 REM LISTING 7.4A
20 REM PRECISE TIMING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1
50 DL=PEEK(560)+PEEK(561)*256:DL=DL+6:
REM THE 2ND LINE ON THE SCEEN
70 POKE DL,PEEK(DL)+128:REM set the 8t
```

Table 7-3. Service Interrupt to Flash Inverse Characters.

```
decimal code        assembly language listing
     72         PHA        ;Push contents of accumulator on
                           stack.
    165         LDA 19     ;Load the accumulator with the
     19                    contents of memory location 19.
    240         BEQ 16     ;Branch if it is zero ahead 16
     16                    bytes.
    169         LDA #255   ;Load the accumulator with 255.
    255
    133         STA 19     ;Store it at location 19.
     19
    165         LDA 206    ;Load the accumulator with the
    206                    contents of location 206.
      9         ORA #1     ;OR the contents of the accumulator
      1                    with 1.
    133         STA 206    ;Store the accumulator at location
    206                    206.
    141         STA 54282  ;Store the accumulator at this
     10                    location and wait for horizontal
    212                    sync.
    141         STA 54273  ;Store is at this location -
      1                    hardware address for character
    212                    mode.
    104         PLA        ;Pull the value for the accumulator
                           off the stack.
     64         RTI        ;Return from interrupt.
```

Listing 7-4A. Precise Timing—Second Method (continued from page 115)

```
h bit to 1 by adding 128
80 FOR ML=1536 TO 1558:READ Q:POKE ML,
Q:NEXT ML:REM POKE THE MACHINE LANGUAG
E SUBROUTINE INTO FREE RAM
90 POKE 206,0:REM SET COUNTER AND DATA
 TO 0
100 POSITION 0,0:? " ":COLOR 42:PLOT 3
3,5:DRAWTO 33,18:DRAWTO 5,18:DRAWTO 5,
5:DRAWTO 33,5
110 COLOR 170:PLOT 34,4:DRAWTO 34,19:D
RAWTO 4,19:DRAWTO 4,4:DRAWTO 34,4
120 COLOR 42:PLOT 35,3:DRAWTO 35,20:DR
AWTO 3,20:DRAWTO 3,3:DRAWTO 36,3
130 POSITION 13,10:? "BEDDY-BY MOTEL"
140 POSITION 10,12:? "COLOR TV - WATER
 BED"
150 POSITION 15,14:? "LOW PRICE"
190 POKE 512,0:POKE 513,6:POKE 54286,1
92:REM ADDRESS OF MACHINE LANGUAGE SUB
ROUTINE - ACTIVATE INTERRUPT
200 DATA 72,165,19,240,16,169,255,133,
19,165,206,9,0,133,206,141,10,212,141,
1,212,104,64
210 GOTO 210
```

This listing is nearly identical to the previous one. The difference is in the data line. In the first program, the value in location 206 was "or'd" with 1. The value was stored in 54273. By alternating this value between a 1 and a 0, we turn on the characters that were printed in inverse video. In the second program a zero is stored in this location. Now the computer will print the same characters in inverse video, then normal video.

PLAYER/MISSILE ENHANCEMENTS

In a previous chapter, the player/missile graphics were described as a band that fit over the screen. This band could be moved to the right or left by poking a location with a value. In the next program we will use the service routine to continually move a player on the screen.

Listing 7-5. Moving Players

```
10 REM LISTING 7.5
20 REM MOVING PLAYERS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
50 GRAPHICS 0
60 DL=PEEK(560)+PEEK(561)*256:DL=DL+6:
REM THE 2ND LINE ON THE SCEEN
70 POKE DL,PEEK(DL)+128:REM set the 8t
```

Listing Listing 7-5. Moving Players (continued from page 117)

```
h bit to 1 by adding 128 to the peek o
f that location
90 A=PEEK(106)-16:REM 2K ABOVE DISPLAY
 LIST
100 POKE 54279,A:PMBASE=A*256:REM TELL
 ANTIC WHERE PLAYERS WILL BEGIN
110 POKE 559,62:POKE 53277,3:REM SINGL
E LINE RESOLUTION
120 FOR X=PMBASE+1024 TO PMBASE+1280:P
OKE X,0:NEXT X:REM CLEAR OUT THE GARBA
GE
130 RESTORE 140:FOR X=0 TO 10:READ Q:P
OKE PMBASE+1104+X,Q:NEXT X
140 DATA 64,120,94,95,94,120,64,64,255
,126,60
150 POKE 704,60:POKE 53248,0:POKE 205,
0:POKE 710,0
170 FOR ML=1536 TO 1548:READ Q:POKE ML
,Q:NEXT ML:REM POKE THE MACHINE LANGUA
GE SUBROUTINE INTO FREE RAM
190 POKE 512,0:POKE 513,6:POKE 54286,1
92:REM ADDRESS OF MACHINE LANGUAGE SUB
ROUTINE - ACTIVATE INTERRUPT
200 DATA 72,230,205,165,205,141,10,212
,141,0,208,104,64
210 GOTO 210
```

Line 50 sets the graphics mode to 0. This is done at the beginning of the program because a value will be added to an existing value in the display list. By setting the graphics mode at the beginning of the program, the computer reinstates the original value into the display list. If this wasn't done and the program was rerun, the computer would be adding a value to the value that was changed in the previous run.

Line 60 finds the beginning of the display list and sets the variable DL to the 2nd line or the 7th value in the display list.

Line 70 gets the value from the display list and adds 128 to it. The new value is poked back into the display list. This is the line that the graphics were set back to zero for.

Line 90 finds out how much memory is in the system and subtracts 2K from it. This places the player/missile graphics before the display list.

Line 100 pokes the beginning location of the player/missile area into 54279. That amount is multiplied by 256 to get the actual location of the beginning of the player/missile area.

Line 110 sets the resolution to single line resolution and tells the computer to enable the players/missiles.

Line 130 clears out the memory area that will be used by the player. If this memory is not

cleared, the player could contain random bits or the data from a previous program. This clutter would appear on the screen when the player is moved onto the screen.

Line 130 reads the data that makes up the boat into the player/missile graphics area.

Line 140 is the data for the boat.

Line 150 sets the color of the boat, the background color of the screen, and sets the location of the boat on the screen to zero. Memory location 205 will be used as a shadow location for 53248. The value in 205 will be changed by one each time the computer executes this subroutine. This value will then be stored in location 53248 by the service routine. Every time this value changes, the boat will move on the screen.

Line 170 reads the values for the machine language routine and stores them in RAM.

Line 190 tells the computer where in memory the service routine is located and tells the computer that it will have an interrupt.

Line 200 is the data for the machine language subroutine.

To end this program press SYSTEM RESET.

When you use any service routine in program, keep in mind that the computer will execute the routine continually until the system reset key is pressed. If it is used with a BASIC program, the program can be processing information, waiting for an input, or drawing on the screen while the service routine performs its functions. In effect, you are running two programs at the same time.

Chapter 8

Scrolling

Scrolling is a technique that moves the contents of a screen up, down, left, or right. When the screen appears to move up or down, the movement is referred to as *vertical scrolling*. When it moves left or right, the movement is referred to as horizontal scrolling. Either the entire screen or selected lines can move.

Text adventure games usually scroll the bottom part of the screen, while the top part remains the same. The text appears to disappear just under the dotted line. Space games often use both horizontal and vertical scrolling. The entire galaxy moves to the left or right, up or down, depending on the direction of the joystick. This gives the illusion of a larger playfield and adds realism to the game.

There are two methods of scrolling on the ATARI computer. The course vertical scroll moves information up or down one line at a time. The entire screen moves up or down. If there is a lot of information on the screen, the movement could appear jumpy. The course horizontal scroll moves every line to the left or right one byte. Again, moving the entire screen could make it appear to roll. It does not look smooth.

Fine vertical scrolling moves the line up or down by one pixel. As each line moves up, the line beneath it also moves up. The movement is much smoother than the course scroll. The fine horizontal scroll moves the line to the left or right by one pixel.

COURSE SCROLLING

In a previous chapter we looked at the display list. The fifth and sixth bytes of the display list tell the computer where the screen memory begins. If this value were changed, the screen display would be different. In the next program, you will print a message on the screen. The message is too long to be displayed on the screen all at one time. You could print part of it, have a timing loop in the program, and then print the rest of it, but in this program the entire message is printed on the screen slowly. The message scrolls from the bottom of the screen to the top. It is not difficult to read, although it is a bit jumpy.

Listing 8-1. Course Vertical Scroll

```
10 REM LISTING 8.1
20 REM COURSE VERTICAL SCROLL
```

Listing 8-1. Course Vertical Scroll (continued from page 121)

```
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0
50 DLIST=PEEK(560)+PEEK(561)*256:REM G
ET THE BEGINNING OF THE DISPLAY LIST
60 POKE DLIST+3,PEEK(DLIST+3)+5
70 FOR X=6 TO 28:POKE DLIST+X,7:NEXT X

80 ? "THIS MESSAGE WILL"
90 ? "SCROLL ON THE"
100 ? "SCREEN.  IT IS"
110 ? "MUCH TOO LONG TO"
120 ? "BE SHOWN ALL AT"
130 ? "ONCE -- SO YOU"
140 ? "ONLY SEE PART OF"
150 ? "THE MESSAGE AT A"
160 ? "TIME."
170 ? "WHAT YOU DON'T"
180 ? "SEE IS IN THE PART"
190 ? "OF MEMORY THAT IS"
200 ? "NOT DISPLAYED ON"
210 ? "THE SCREEN."
220 ? "THE COURSE SCROLL"
230 ? "MAKES THE ENTIRE"
240 ? "MESSAGE SHOW!"
300 SCRLOW=PEEK(DLIST+4):SCRHI=PEEK(DL
IST+5)
310 FOR X=1 TO 11:SCRLOW=SCRLOW+40:REM
 ADD TO THE VALUE TO MOVE TWO LINES UP
 ON THE SCREEN
320 FOR Y=1 TO 500:NEXT Y:REM TIMING L
OOP FOR READABILITY
330 IF SCRLOW>256 THEN SCRLOW=SCRLOW-2
56:SCRHI=SCRHI+1:REM GO TO THE NEXT PA
GE OF MESSAGE
340 POKE DLIST+4,SCRLOW:POKE DLIST+5,S
CRHI:REM DISPLAY NEW SCREEN
350 NEXT X:REM SHOW ENTIRE MESSAGE
```

Line 40 sets the graphics mode to 0. This is done because you will be poking a value based on the peek of a location in the display list. If the value has already been changed and the program is run again, the results on the second run may not be the same as on the first run.

Line 50 finds the beginning of the display list.

Line 60 changes the value of the fourth byte of the display list (the first byte is DLIST+0). It will now be graphics mode 2.

Line 70 changes the rest of the display list to graphics mode 2. This mode uses 16-pixel rows

instead of 8-pixel rows like modes 0 and 1. Since only 12 rows will be displayed on the screen at one time, half of the display list will not be seen.

Lines 80 - 240 print the message on the screen. There are 17 rows of text. Since we changed the mode by changing the display list, the computer will try to place 40 characters on a line. Each line that we are printing is less than 20 characters. The text will appear double spaced on the screen. All 17 rows of text will be placed in the area of memory set aside for the screen display.

Line 300 finds the beginning of the memory set aside for the screen. The two bytes are stored in variables SCRLOW and SCRHI

Lines 310 - 350 scroll the message on the screen. Line 310 begins the for . . . next loop that will move the beginning byte of the screen display area 11 times. 40 is added to the value in SCRLOW. This is the low order byte of the beginning of the screen display area. We add 40 so that we can move 2 screen rows at the same time. Line 320 contains a timing loop. If we did not slow down this routine, the lines would be printed so fast that we could not read them! Line 330 checks the value of SCRLOW. If it is greater than 255, it will be reduced by that amount, and SCRHI will be increased by 1. If we didn't increase SCRHI by 1 every time SCRLOW reached 255, the computer would stay on the same page of memory. Line 340 changes the beginning of the screen memory by changing the contents of the memory locations that ANTIC looks at to start displaying information on the screen. Line 350 continues the for . . . next loop.

To return to the text mode, press the system reset key.

The next program prints every character that the computer is capable of printing in the color text mode. Every line contains all 256 characters, but only 20 characters can be displayed on the screen at one time. How can every line contain 256 characters? Remember that the fourth instruction in the display list tells ANTIC the mode of the top line and that the next two bytes contain the beginning of the screen memory. If we change the display list so that every line starts its own screen memory, we can make the lines as long or as short as we need. Each line can be in a different mode, or they can all be in the same mode. The next listing prints all the characters in the same mode. The one following it uses mixed modes.

Listing 8-2. Course Horizontal Scroll

```
10 REM LISTING 8.2
20 REM COURSE HORIZONTAL SCROLL
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 POKE 559,0:REM TURN OFF ANTIC
50 SCREEN=PEEK(106)-12:REM THE SCREEN
WILL BEGIN 12 PAGES FROM THE END OF ME
MORY
60 DLIST=SCREEN*256-50:REM START THE D
ISPLAY LIST 50 BYTES BEFORE THE SCREEN
70 FOR X=0 TO 2:POKE DLIST+X,112:NEXT
X:REM FIRST THREE BYTES OF DISPLAY LIS
T
80 FOR X=1 TO 12:REM TWELVE LINES DISP
LAYED ON THE SCREEN
90 POKE DLIST+3*X,71:REM MODE 2
```

Listing 8-2. Course Horizontal Scroll (continued from page 123)

```
100 POKE DLIST+3*X+1,0:REM LOW ORDER A
DDRESS OF SCREEN LINE
110 POKE DLIST+3*X+2,SCREEN+(X-1):REM
HIGH ORDER ADDRESS OF SCREEN LINE
120 NEXT X
130 POKE DLIST+X*3,65:REM JUMP INSTRUC
TION TO BEGINNING OF DISPLAY LIST
140 DLHI=INT(DLIST/256):DLLO=DLIST-(DL
HI*256)
150 POKE DLIST+X*3+1,DLLO
160 POKE DLIST+X*3+2,DLHI
170 POKE 560,DLLO:POKE 561,DLHI:REM PU
T THE NEW DISPLAY LIST ADDRESS INTO IT
S REGISTERS
180 POKE 88,0:POKE 89,SCREEN:REM TELL
THE OPERATING SYSTEM WHERE THE SCREEN
BEGINS
190 POKE 559,34:REM TURN ANTIC BACK ON
200 POKE 82,0:REM START AT THE EDGE OF
 THE SCREEN
210 ? "}"
220 DIM A$(256)
230 FOR X=1 TO 256:A$(X,X)=CHR$(X-1):N
EXT X
240 FOR Z=0 TO 11:DISPLAY=(SCREEN+Z)*2
56:FOR X=0 TO 255:POKE DISPLAY+X,ASC(A
$(X+1,X+1)):NEXT X:NEXT Z
250 FOR Z=0 TO 235:FOR X=1 TO 12:REM L
EAVE 20 CHARACTERS ON THE SCREEN - 12
LINES ON SCREEN
260 POKE DLIST+X*3+1,Z:REM POKE THE LE
FT MARGIN INTO THE LOW ORDER SCREEN AD
DRESS FOR EACH ROW
270 NEXT X:NEXT Z:REM DO ENTIRE SCREEN
280 FOR Z=235 TO 0 STEP -1:FOR X=1 TO
12:REM DO IT BACKWARDS
290 POKE DLIST+X*3+1,Z:REM POKE THE LE
FT MARGIN INTO THE LOW ORDER SCREEN AD
DRESS FOR EACH ROW
300 NEXT X:NEXT Z:REM DO ENTIRE SCREEN
310 GOTO 250:REM KEEP DOING IT
```

Line 40 turns off ANTIC. Since we are making major changes to the display list, it is a good idea to turn off ANTIC. Otherwise we could confuse or lock up the computer.

Line 50 finds the amount of memory available in the system and subtracts 12 from it. This will give us 12 pages or 3K of memory for the screen display.

Line 60 subtracts 50 bytes from the beginning of the screen area. That should be enough for the display list.

Lines 70 - 160 set up the new display list. Line 70 pokes in the values for the first three bytes of the display list. Lines 80 - 120 set up the main body of the display list. There will be 12 lines on the screen. Instead of poking in the value for that line of the display list, we need to do more. First, we will offset the value of X by multiplying it by 3. Three is used because there are 3 values that must be poked in for every line. The first value is 71; this is the same value that would normally be placed in a display list for mode 2. It means that the line is in mode 2. The next 2 bytes indicate the beginning of the screen display area. The next value to be poked in is a zero. Every line will begin on an even page of memory. The next value is the high order byte for the screen display area. The line number minus one is added to the value of screen. The jump instruction for ANTIC is poked into the end of the display list along with the two byte address for the beginning of the display list. Now ANTIC will know where to jump when it gets to the end of the display list. The new values for the location of the display list are placed in memory locations 560 and 561. ANTIC is turned back on and the left margin of the screen is changed to 0.

Line 210 clears the screen.

Line 220 sets aside 256 bytes for A$. This string will be displayed in every line on the screen.

Line 230 places the character for every value from 0 to 255 into A$.

Line 240 calculates the beginning of the screen display area for the line by multiplying the value of SCREEN plus Z by 256. The ASC (ATASCII) value of every character in A$ is poked into this memory area.

Lines 250 - 270 change the value in the low order byte of the screen display area for every line. This makes the rows on the screen appear to move to the left.

Lines 280 - 300 reverse the process. By subtracting one from the low order byte of every row displayed, the screen will shift to the right.

FINE VERTICAL SCROLL

The fine vertical scroll is achieved by moving the characters on a line up or down one pixel at a time. After the first line has been scrolled as far as possible, the scrolling to the next line is done the same way it was done in the course scroll program.

Once again, the display list plays an important part in the fine scroll. The sixth bit of the byte that tells ANTIC what mode the line is must be set in order to scroll that line. An easier way is to

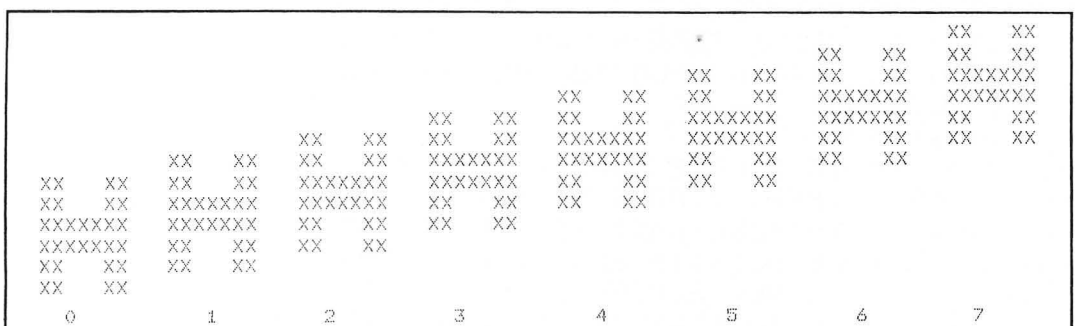


Fig. 8-1. Scroll values.

add 32 to the mode. If the screen is mode 0, the display list has 2 for the mode. Change the 2 to 34 and that line can scroll vertically.

Of course, the line will not scroll all by itself. The computer must be told when to scroll that line and how far to scroll it. 54277 is poked with a value from 0 to 7. A zero holds the character in the position that it would be in if there was no scroll. A seven moves the character up seven pixels. It is almost in line with the correct position for the line just above it. Figure 8-1 shows the character position from 0 to 7.

The following program uses the fine scroll with the course scroll to move a message up on the screen.

Listing 8-3. Fine Vertical Scroll

```
10 REM LISTING 8.3
20 REM FINE VERTICAL SCROLL
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0
50 DLIST=PEEK(560)+PEEK(561)*256:REM G
ET THE BEGINNING OF THE DISPLAY LIST
60 POKE DLIST+3,102
70 FOR X=6 TO 28:POKE DLIST+X,38:NEXT
X
80 POSITION 2,8
170 ? "THIS SCROLLING"
180 ? "IS EASIER TO READ"
190 ? "THAN THE COARSE"
200 ? "METHOD."
210 ? "YOU CAN MAKE"
220 ? "IT SCROLL  FAST"
230 ? "OR SLOW WITH"
240 ? "A TIMING LOOP."
290 SCRLOW=PEEK(DLIST+4):SCRHI=PEEK(DL
IST+5)
300 FOR X=1 TO 8:SCRLOW=SCRLOW+20:REM
ADD TO THE VALUE TO MOVE TWO LINES UP
ON THE SCREEN
310 IF SCRLOW>256 THEN SCRLOW=SCRLOW-2
56:SCRHI=SCRHI+1:REM GO TO THE NEXT PA
GE OF MESSAGE
320 LN=LN+10
330 FOR Z=0 TO 7:POKE 54277,Z:FOR Y=1
TO 25:NEXT Y:NEXT Z:POKE 54277,0
340 POKE 559,0:POKE 54277,0:POKE DLIST
+4,SCRLOW:POKE DLIST+5,SCRHI:POKE 559,
34:REM DISPLAY NEW SCREEN
350 FOR Y=1 TO 10:NEXT Y:NEXT X:REM SH
OW ENTIRE MESSAGE
```

Line 40 resets the display list every time the program is run.

Line 50 finds the beginning of the display list. Every byte that tells ANTIC the mode of the line must be changed. The 6th bit of the byte must be set to 1.

Line 60 sets the top line of the screen for the scroll. This byte also tells ANTIC that the next two bytes are where the screen memory begins.

Line 70 sets the rest of the bytes in the display list for the vertical scroll.

Line 80 sets the position on the screen where the message will begin to be printed.

Lines 170 - 240 print the message on the screen. The entire message will not be printed on the screen since this is graphics mode 2. The display list is as long as it would be in mode 0, so part of the message will be printed out of the screen area.

Line 290 sets the variable SCRLOW to the low order address of the screen and SCRHI to the high order address of the screen.

Lines 300 - 350 scroll the message on the screen. Line 300 begins the for . . . next loop. 20 is added to the value in SCRLOW because there are only 20 characters displayed in a line on the screen. If the value of SCRLOW exceeds 255, the value is reset by subtracting 255 from it, and 1 is added to the value of SCRHI. This will give ANTIC a new page to display on the screen. If SCRHI was not incremented, ANTIC would only display 1 page of screen memory over and over again. Line 330 begins the fine vertical scroll. A value from 0 to 7 is poked in memory location 54277. After each poke, there is a timing loop. This will slow down the process of scrolling and make the text more readable. Once the lines are scrolled up as high as they can go, we shut off ANTIC, poke 54277 with 0 to reset the scroll or put the line back to normal, and poke the display list with the new screen values. This is the course scroll. Because ANTIC is shut off, it will be a smooth scroll. ANTIC is turned back on and after another timing loop, the program continues with the original for....next loop.

When the entire message has scrolled up the screen, the program will end. Press the system reset key to return to the text mode.

When you want a message or characters to move down on the screen, the process must be reversed. The characters must be brought down one line by the course scroll, scrolled all the way up to the top of that line, then slowly scrolled down on the screen. A downward scroll is not always as smooth as an upward scroll.

Listing 8-4. Fine Vertical Scroll: Down

```
10 REM LISTING 8.4
20 REM FINE VERTICAL SCROLL - DOWN
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0
50 DLIST=PEEK(560)+PEEK(561)*256:REM G
ET THE BEGINNING OF THE DISPLAY LIST
60 POKE DLIST+3,102
70 FOR X=6 TO 28:POKE DLIST+X,38:NEXT
X
80 POSITION 2,8
90 SCRLOW=244:SCRHI=PEEK(DLIST+5):POKE
 DLIST+4,SCRLOW
```

Listing 8-4. Fine Vertical Scroll: Down (continued from page 127)

```
170 ? "THIS SCROLLING"
180 ? "IS EASIER TO READ"
190 ? "THAN THE COARSE"
200 ? "METHOD."
210 ? "YOU CAN MAKE"
220 ? "IT SCROLL  FAST"
230 ? "OR SLOW WITH"
240 ? "A TIMING LOOP."
300 FOR X=1 TO 8:SCRLOW=SCRLOW-20:REM
SUBTRACT TO MOVE TWO LINES DOWN ON THE
 SCREEN
330 POKE 559,0:POKE DLIST+4,SCRLOW:POK
E DLIST+5,SCRHI:POKE 559,34:REM DISPLA
Y NEW SCREEN
340 FOR Z=7 TO 0 STEP -1:POKE 54277,Z:
FOR Y=1 TO 20:NEXT Y:NEXT Z
350 FOR Y=1 TO 20:NEXT Y:NEXT X:REM SH
OW ENTIRE MESSAGE
```

Line 40 resets the graphics to mode 0. This restores the display list to its original contents.

Line 50 finds the beginning of the display list.

Line 60 sets the fourth byte of the display list for the vertical scroll and graphics mode 1.

Line 70 sets the rest of the lines in the display list for graphics mode 1 with the bit set for the vertical scroll.

Line 80 begins printing the message on this line.

Line 90 sets the variable SCRLOW to 244. This figure was calculated by printing the value of SCRLOW at the end of the last program. The variable SCRHI should remain the same value as the high byte address in the display list. Change the display list low order byte for the screen address.

Lines 170 - 240 print the message on the screen. This time the entire message will be visible on the screen.

Lines 300 - 350 scroll the message down on the screen. This time 20 is subtracted from the beginning of the screen memory. By subtracting 20, ANTIC starts to display the screen area 20 bytes before the point where the message begins. This pushes the message down one line on the screen.

Line 330 shuts off ANTIC while the new screen memory values are poked into the display list. ANTIC is turned on and the screen displays the message one line lower. By poking memory location 54277 with 7 to 0, we start the scroll with the lines scrolled up as high as possible. They are then slowly lowered on the screen. The timing loops keep the scrolling smooth. Every time the line is in the correct position, ANTIC is shut off, and the line is moved down one with a course scroll, but it is immediately scrolled up after ANTIC is turned on. A scroll down is not always as smooth as a scroll up.

PLAYFIELD WIDTHS

The width of the playfield is the number of columns that you can place information in. Up until

now, no matter which mode we worked in, the width of the screen was always the same - 40 columns or 320 pixels. The width of the playfield does not have to remain constant. There are three different playfield widths that you can choose from. Try this in the direct mode:

```
POKE 559,33
```

Look at the screen! It is much smaller now. It is only 32 columns, or 256 pixels wide. Everything that is on the screen looks like it is in the wrong place. ANTIC doesn't look to see how wide the screen is. It will still try to put 40 characters in a line. If the line is too short, it will put the rest in the next line.

Now try this - POKE 559,35

There is no margin on the left and right sides of the screen. Again, the information on the screen is not in the right place. This is the wide playfield. It is 48 characters or 384 pixels wide. The screen information will not be in the right place, because the information appears on the screen sequentially. If the line is longer than 40 characters, the information from the next line is placed on this line. To get back to the normal width, poke 559,34 or press the system reset key.

These playfields can be used to create special effects. The wide playfield is used with the fine horizontal scroll.

FINE HORIZONTAL SCROLL

The fine horizontal scroll is very similar to the fine vertical scroll. Every character in the line that is to be scrolled is moved to the right approximately 2 pixels. Clear the screen and try this in the direct mode:

```
X = PEEK(560) + PEEK(561) * 256 + 7:POKE X,18
```

This sets one line of the display list for the horizontal scroll. When you press the return key, the 'X=' will move to the left and seem to disappear off the screen. READY will appear under the 8 and the left margin will seem to be on the right side of the screen. Now enter this:

```
FOR Y=0 TO 15:POKE 54276,Y:NEXT Y
```

The third line moves to the right and the entire line can be read. If you reverse the command:

```
FOR Y=15 TO 0 STEP -1:POKE 54276,Y:NEXT Y
```

the line will move back to the left. The rest of the screen is offset because this line is 8 characters or bytes longer than the other lines on the screen. READY is on the next line because the first 8 bytes of the fifth line are now on the fourth line. Every line after that is offset by 8 bytes. If enough lines are set for the horizontal scroll, the offset will work itself back to the left margin.

The next two programs are variations on a ticker tape routine. The first program performs the horizontal scroll using the same technique as was used with the vertical scroll. Notice that in addition to scrolling to the left, the message is also scrolling up the screen! Press the system reset key before the message goes too far. The second program shows a solution to this program.

Listing 8-5. Fine Horizontal Scroll

```
10 REM LISTING 8.5
20 REM FINE HORIZONTAL SCROLL
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM A$(48)
50 GRAPHICS 0:POKE 82,0:POKE 752,1:REM
 SET LEFT MARGIN
```

Listing 8-5. Fine Horizontal Scroll (continued from page 129)

```
60 DLIST=PEEK(560)+PEEK(561)*256:SCRLO
W=PEEK(DLIST+4):SCRHI=PEEK(DLIST+5):RE
M DISPLAY LIST AND SCREEN MEMORY
70 POKE DLIST+10,PEEK(DLIST+10)+16:REM
 SET IT UP FOR HORIZONTAL SCROLL
80 A$="STAY TUNE FOR AN IMPORTANT MESS
AGE ON COMPUTERS "
90 POKE 54276,15:REM SET SCROLL TO THE
 RIGHT
100 POSITION 0,5:? A$
110 FOR X=15 TO 0 STEP -1:POKE 54276,X
:FOR Y=1 TO 40:NEXT Y:NEXT X:REM SCROL
L TO THE LEFT
120 SCRLOW=SCRLOW+4:IF SCRLOW>255 THEN
 SCRHI=SCRHI+1:SCRLOW=SCRLOW-255
130 POKE DLIST+4,SCRLOW:POKE DLIST+5,S
CRHI
140 FOR Y=1 TO 10:NEXT Y:GOTO 110
```

Line 40 sets aside 48 bytes for the message. This is the number of characters that will fit on one line when the screen is set for the wide playfield.

Line 50 clears the screen, sets the left margin to 0, and removes the cursor from the screen. If we did not set the left margin to 0, the computer would leave two bytes blank on the screen in the line that the message is printed in. These two bytes are where the left margin is for a normal playfield width.

Line 60 finds the beginning of the display list and places the screen memory address in variables SCRLOW and SCRHI.

Line 70 sets the sixth row of the screen for the fine horizontal scroll. The fine horizontal scroll is set by adding 16 to the value in the display list.

Line 80 places the message into A$. Be sure to add the space after the last word, computers.

Line 90 pokes 15 into memory location 54276. This will scroll the line all the way to the right.

Line 100 prints the message on the sixth line of the screen. (The first line is 0.)

Line 110 begins the horizontal scroll. The for...next loop decreases from 15 to 0, moving the message from the right to the left.

Line 120 adds 4 to the low byte of the screen memory area. Scrolling from 15 to 0 moves 4 characters off the screen.

Line 130 pokes the new values into the display list and completes the scroll.

Notice that the message spirals on the screen. Four characters at a time appear on the line above the scrolling line. This line is not set to scroll, so the characters stay there until the next course scroll. We are moving the screen display area, so even though we are moving the characters across on the line, we are also moving them up in the screen memory.

The next program tries to correct this situation. When the words are scrolled to the left, the letters are removed and placed at the end of the line. The screen is still spiralling up the display,

but because the letters are constantly being removed from the beginning of the line and added to the end, it gives the illusion of remaining on the same line.

Listing 8-5A. Fine Horizontal Scroll—Second Method

```
10 REM LISTING 8.5A
20 REM FINE HORIZONTAL SCROLL
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM A$(48)
50 GRAPHICS 0:POKE 82,0:POKE 752,1:REM
 SET LEFT MARGIN
60 DLIST=PEEK(560)+PEEK(561)*256:SCRLO
W=PEEK(DLIST+4):SCRHI=PEEK(DLIST+5)
70 POKE DLIST+10,PEEK(DLIST+10)+16:REM
 SET IT UP FOR HORIZONTAL SCROLL
80 A$="STAY TUNE FOR AN IMPORTANT MESS
AGE ON COMPUTERS "
90 POKE 54276,15:POSITION 0,5:? A$
100 LINE5=SCRHI*256+SCRLOW+199:REM FIR
ST BYTE OF THE FIFTH LINE
110 POKE LINE5+47,PEEK(LINE5):POKE LIN
E5,0:LINE5=LINE5+1
120 SCRLOW=SCRLOW+1:IF SCRLOW>255 THEN
 SCRLOW=SCRLOW-255:SCRHI=SCRHI+1
130 FOR X=15 TO 12 STEP -1:POKE 54276,
X:FOR Y=1 TO 30:NEXT Y:NEXT X
140 POKE 559,0:POKE DLIST+4,SCRLOW:POK
E DLIST+5,SCRHI:POKE 559,34:GOTO 110
```

PLAYER/MISSILE GRAPHICS

Many applications for using horizontal and vertical scrolling come to mind when you consider adding player/missile graphics. A map of the United States or the world could be drawn on the screen. To keep it at a reasonable size, you would not have to fit the entire map on the screen. A player could be used as a cursor. Moving the joystick would move the cursor up, down, left, or right on the map. When the red button on the joystick is pressed, the screen would scroll in the direction indicated by the joystick, exposing other parts of the map.

Chapter 9

Page Flipping

In the display list there are two bytes set aside to tell ANTIC where the beginning of the screen memory is. If we were to change these bytes, we would get some strange displays on the screen. It could be garbage, or it could be a clear screen. It all depends on what is in the memory area that ANTIC is trying to display. Try this in the direct mode:

```
DLIST=PEEK(560)+PEEK(561)*256:POKE DLIST+5,6
```

What do you see on your screen? This is the area of memory that we usually store machine language subroutines in because the operating system doesn't use it.

DISPLAYING TWO SCREENS

By adjusting the memory that will be used by the screen, we can set aside memory for more than one screen display. We can then flip back and forth between the two screens. The new picture or message will immediately appear on the screen. If speed is critical, or you do not want the user to see the image being drawn, screen flipping is the answer. It does, however, use up a lot of memory. In the text mode, figure another 1K of memory for each additional screen that you want displayed. If you are using a high resolution graphics mode, the memory cost is even more!

The following program shows two messages that are displayed on two different screens.

Listing 9-1. Screen Flipping

```
10 REM LISTING 9.1
20 REM SIMPLE PAGE FLIPPING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1:? "}CLEAR}"
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM RESET THE DISPLAY LI
```

Listing 9-1. Screen Flipping (continued from page 133)

```
ST
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:? "THIS MESSAGE IS ON
PAGE 2":? "WHILE YOU ARE READING IT":?
 "I'M WRITING ON SCREEN 2"
100 POKE 88,SCR1:POKE 89,SCR2:REM LET
THE COMPUTER WRITE ON THE OTHER SCREEN
110 ? "     GOOD WORD!":REM 2 ESC-DO
WN ARROWS - FIVE SPACES
120 POKE 88,SC1:POKE 89,SC2:C=0:? "
  I'm ready press any key!":REM 2 DOWN
ARROWS - 3 SPACE
130 IF PEEK(764)=255 THEN 130
140 POKE 764,255:REM CLEAR THE REGISTE
R
150 POKE DL2+4,SCR1:POKE DL2+5,SCR2:RE
M SHOW SCREEN 1
160 FOR Y=1 TO 200:NEXT Y:REM TIMING L
OOP
165 ? "}CLEAR}":IF C THEN 180
170 ? "This was printed while you
were       watching the other screen."
:C=1:REM 4 DOWNARROWS
180 ? "press any key":REM DOWNARROW
& TAB
190 POKE DL2+4,SC1:POKE DL2+5,SC2:REM
RESET THE SCREEN
200 GOTO 130
```

Line 40 sets the graphics mode to 0. This line also removes the cursor and clears the screen. This will be considered page 1. It is ready for something to be written on it.

Line 50 finds the beginning of the display list. The location of the screen memory is stored in the fifth and sixth bytes of the display list. These values are stored in the variables SC1 and SC2. The first byte of the screen is stored in the variable SCR.

Line 60 subtracts 4 from the value in memory location 106. This memory location tells the computer how much memory (RAM) is available for it to use. It will count backwards from this location when setting aside screen space and the display list. By poking 4 less than the total number of pages of memory available, we are saving 1K of memory of the screen and display list that we just initialized.

Line 70 resets the display list and screen. Because we changed the amount of memory that the computer thinks it has, the first display list and screen display memory is intact. The screen memory and display list that the computer set in this line are lower in memory than those

established for the preceding screen. We now have two display lists and screen memories in the computer's RAM.

Line 80 calculates the new display list and the location of the new screen memory. This is the second display list and screen in RAM.

Line 90 removes the cursor and prints a message on the screen. This message appears on the screen that you are looking at.

Line 100 pokes the memory location of the first screen into memory locations 88 and 89. This is where the computer looks to see where the screen display area is. If we change these bytes to the location on the first screen, the computer will print or draw in that memory area. This will not affect the display that we are looking at.

Line 110 is the message that the computer will print on the screen. This message will not appear on the screen that we are looking at because locations 88 and 89 have been changed.

Line 120 sets the variable C to 0. C is used as a flag in this program. When it is 0, a second message will be printed on the second screen. When it is 1, the message will not be printed. The message between the quotes will be printed after C is set to 0. Before printing any message, memory locations 88 and 89 must be changed. If they were not changed back to the screen memory of the screen that we are looking at, we would not see this message.

Line 130 checks memory location 764. When its value is not 255, a key has been pressed. The program will loop here until a key is pressed.

Line 140 clears memory location 764 by resetting it to 255. If this location was not reset when the program looped back to line 130, it would think that a key had been pressed whether one was or not.

Line 150 pokes the values for the first screen memory into the display list. Now the message that was poked into that memory area will be displayed on the screen.

Line 160 is a timing loop to give you a chance to read the message.

Line 165 clears this screen. This will not clear the screen that you are looking at! This will clear screen 2. If C is set, the program will go on to line 180.

Line 170 prints a message on the screen and sets C to 1. This message is not printed on the screen that you are looking at, but at the other screen.

Line 180 prints the rest of the message.

Line 190 resets the screen by poking the screen memory area for the second screen into the display list.

Line 200 sends the computer back to line 130 to repeat the program.

To end the program press the system reset key. This will reset the display list and all the registers. If you press the break key, you will only stop the program but not reset any of the registers.

This method of screen flipping can be used with other modes as well. In the next program, the screen will flip between mode 0 and mode 1.

Listing 9-2. Simple Page Flipping: Two Different Modes

```
10 REM LISTING 9.2
20 REM SIMPLE PAGE FLIPPING - TWO DIFF
ERENT MODES
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 17
```

Listing 9-2. Simple Page Flipping: Two Different Modes (continued from page 135)

```
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM RESET THE DISPLAY LI
ST
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:? "THIS MESSAGE IS ON
PAGE 2":? "WHILE YOU ARE READING IT":?
 "I'M WRITING ON SCREEN 1"
100 POKE 88,SCR1:POKE 89,SCR2:POKE 87,
1
110 ? #6;"GOOD WORK"
120 POKE 88,SC1:POKE 89,SC2:POKE 87,0:
C=0:? "    I'm ready press any key!"
130 IF PEEK(764)=255 THEN 130
140 POKE 764,255:REM CLEAR THE REGISTE
R
150 POKE 560,DLL1:POKE 561,DLH1:REM SH
OW SCREEN 2
160 FOR Y=1 TO 200:NEXT Y:REM TIMING L'
OOP
165 ? "}CLEAR}":IF C THEN 180
170 ? "This was printed while you
were        watching the other screen."
:C=1
180 ? "press any key"
190 POKE 560,DLL2:POKE 561,DLH2:REM RE
SET THE SCREEN
200 GOTO 130
```

Line 40 sets the display list for mode 1 without the text window. The rest of the listing is identical to the previous one with one exception.

Line 150 changes the display list that the computer is using for the second screen to the display list for the first screen.

Line 190 resets the display list for the second screen.

By changing memory locations 560 and 561 for the different display lists, you can flip between screens that are in different modes.

What if you want to display two screens simultaneously? It can be done. But with all the capabilities of the ATARI with one screen display, is there really a need to display two screens at

once? The next four programs illustrate different approaches that can be taken to display two screens at the same time. In all cases, there will be some flickering.

Listing 9-3. Simultaneous Page Flipping—in BASIC

```
10 REM LISTING 9.3
20 REM SIMULTANEOUS PAGE FLIPPING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1:? "}CLEAR}"
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM SET THE DISPLAY LIST
 TO ANOTHER MODE
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:POSITION 2,9:? "THIS CAN
 CREATE"
100 POKE 88,SCR1:POKE 89,SCR2
110 ? "INTERESTING EFFECTS":REM 2-
ESC-DOWNARROWS, 2-ESC-TABS
120 POKE DL2+4,SCR1:POKE DL2+5,SCR2:PO
KE DL2+4,SC1:POKE DL2+5,SC2:GOTO 120
```

Listing 9-4. Simultaneous Page Flipping—Two Modes

```
10 REM LISTING 9.4
20 REM SIMULTANEOUS PAGE FLIPPING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 17
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 18:REM SET THE DISPLAY LIS
T TO ANOTHER MODE
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POSITION 2,3:? #6;"THIS CAN CREATE"
100 POKE 88,SCR1:POKE 89,SCR2:POKE 87,
1
110 ? #6;"INTERESTING EFFECTS"
```

Listing 9-4. Simultaneous Page Flipping—Two Modes (continued from page 137)

```
130 IF PEEK(764)=255 THEN 130
140 POKE 560,DLL1:POKE 561,DLH1
160 POKE 560,DLL2:POKE 561,DLH2:GOTO 1
30
```

Listing 9-5. Simultaneous Page Flipping: Machine Language Subroutine

```
10 REM LISTING 9.5
20 REM SIMULTANEOUS PAGE FLIPPING - MA
CHINE LANGUAGE SUBROUTINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1:? "}CLEAR}"
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM SET THE DISPLAY LIST
 TO ANOTHER MODE
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:POSITION 2,9:? "THIS CAN
 CREATE"
100 POKE 88,SCR1:POKE 89,SCR2
110 ? "INTERESTING EFFECTS"
120 POKE 203,SCR1:POKE 204,SCR2:POKE 2
05,SC1:POKE 206,SC2:REM ADDRESS OF BOT
H SCREENS
130 POKE 208,DLH2:POKE 207,DLL2+4:REM
WHERE SCREEN ADDRESS IS STORED
140 FOR X=0 TO 30:READ C:POKE 1536+X,C
:NEXT X
150 DATA 104,160,0,165,203,145,207,200
,165,204,145,207,136,165,205,145,207,2
00,165,206,145,207,136,173,252,2
160 DATA 201,255,240,229,96
170 Q=USR(1536)
```

Listing 9-5A. Simultaneous Page Flipping: Machine Language Subroutine—Horizontal Blank

```
10 REM LISTING 9.5A
20 REM SIMULTANEOUS PAGE FLIPPING - MA
```

```
CHINE LANGUAGE SUBROUTINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1:? "}"
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM SET THE DISPLAY LIST
 TO ANOTHER MODE
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:POSITION 2,5:? "THIS CAN
 CREATE"
100 POKE 88,SCR1:POKE 89,SCR2
110 ? "INTERESTING EFFECTS"
120 POKE 206,SC2:REM ADDRESS OF ONE SC
REEN
130 FOR X=0 TO 19:READ C:POKE 1536+X,C
:NEXT X
140 POKE 1553,DLH2:POKE 1552,DLL2+5:PO
KE 209,4:REM WHERE SCREEN ADDRESS IS S
TORED
150 DATA 72,169,4,69,209,133,209,165,2
06,24,101,209,141,10,212,141,0,0,104,6
4
170 POKE DL2+28,130
180 POKE 512,0:POKE 513,6:POKE 54286,1
92
190 GOTO 190
```

These four listings are nearly identical.

Line 40 sets the mode to 0 (except in listing 9-4 where it is set to 17, which is mode 1 without a text window), removes the cursor, and clears the screen. The display list is now set for the first screen.

Line 50 finds the beginning of the display list, and the location of the screen memory for this mode. These values will be used later in the program.

Line 60 moves the end of RAM 1K. This protects the display list when we reset the graphics mode.

Line 70 sets a new display list by setting the graphics mode. This new display list will be located before the old one. The screen display area for this mode will also be located before the other screen. The other display list is protected because 106 was poked with a number that was 1K less than the actual amount of memory in the system.

Line 80 finds the new display list and the screen memory locations for the second screen.

Table 9-1. Machine Language Subroutine to Flip Screen.

```
decimal code       assembly language listing
   104             PLA         ;Pull a number from the stack.
   160             LDY #0      ;Load index Y with 0.
     0
   165             LDA 203     ;Load the accumulator with the
   203                         value in location 203.
   145             STA (207),Y;Store it in memory location
   207                         207 plus the value of index Y.
   200             INY         ;Increment index Y.
   165             LDA 204     ;Load the accumulator with the
   204                         value in location 204.
   145             STA (207),Y;Store it in memory location
   207                         207 plus the value of index Y.
   136             DEY         ;Decrement index Y.
   165             LDA 205     ;Load the accumulator with the
   205                         value in location 205.
```

Line 90 prints a message on the screen that we are looking at.

Line 100 sets locations 88 and 89 for the first screen memory. In listing 9-4, location 87 is set for mode 1. This will place the message in the correct position on the screen.

Line 110 prints the message on the screen. This message will appear on the first screen, not the screen that we are looking at.

Line 120 is a loop in listing 9-3. The screen memory location in the display list is constantly changed between the first and second screen. Although both messages appear on the screen, there is noticeable flickering.

In listing 9-4, line 130 waits for a key to be pressed. Each time one is pressed, the address of the display list is changed back and forth between the first display list and the second. This program flips the screens by changing the entire display lists because they are in two different modes.

Listing 9-5 uses a machine language subroutine to flip the screens. See Table 9-1 for the assembly language listing of this subroutine. Once the machine language subroutine is poked into memory, the computer uses a USR command to access it. The screen will continue to flip until a key is pressed. There is less flickering with this method than the previous one, but it is still

```
145             STA (207),Y;Store it in memory location
207                       207 plus the value of index Y.
200             INY       ;Increment index Y.
165             LDA 206   ;Load the accumulator with the
206                       value in location 206.
145             STA (207),Y;Store it in memory location
207                       207 plus the value of index Y.
136             DEY       ;Decrement index Y.
173             LDA 764   ;Load the accumulator with the
253                       value from location 764
  2
201             CMP #255  ;See if a key was pressed.
240             BEQ 229   ;Branch if no key was pressed
229                       backwards 27 bytes.
 96             RTS       ;Return to BASIC if key was
                          pressed.
```

noticeable. Lines 120 and 130 poke the addresses of both screens and the address of the display list where the screen memory should be placed—in memory locations that are not used by BASIC or the operating system.

Listing 9-5A uses a display list interrupt to change the screens. Table 9-2 shows the assembly language listing for this routine. By using a display list interrupt, the screen is flipped every time ANTIC comes to that line in the display list. This method causes the least amount of flickering.

Listing 9-5B. Simultaneous Page Flipping: Machine Language Subroutine—Vertical Blank

```
10 REM LISTING 9.5B
20 REM SIMULTANEOUS PAGE FLIPPING - MA
CHINE LANGUAGE SUBROUTINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 752,1:? "}"
50 DLL1=PEEK(560):DLH1=PEEK(561):DL1=D
LL1+DLH1*256:SCR1=PEEK(DL1+4):SCR2=PEE
```

Listing 9-5B. Simultaneous Page Flipping: Machine Language Subroutine—Vertical Blank (continued from page 141)

```
K(DL1+5):SCR=SCR1+SCR2*256
60 POKE 106,PEEK(106)-4:REM MOVE SCREE
N UP 1K
70 GRAPHICS 0:REM SET THE DISPLAY LIST
 TO ANOTHER MODE
80 DLL2=PEEK(560):DLH2=PEEK(561):DL2=D
LL2+DLH2*256:SC1=PEEK(DL2+4):SC2=PEEK(
DL2+5):SC=SC1+SC2*256
90 POKE 752,1:POSITION 2,5:? "THIS CAN
 CREATE"
100 POKE 88,SCR1:POKE 89,SCR2
110 ? "INTERESTING EFFECTS"
120 POKE 206,SC2:REM ADDRESS OF ONE SC
REEN
130 FOR X=0 TO 21:READ C:POKE 1536+X,C
:NEXT X
140 POKE 1553,DLH2:POKE 1552,DLL2+5:PO
KE 209,4:REM WHERE SCREEN ADDRESS IS S
TORED
150 DATA 72,169,4,69,209,133,209,165,2
06,24,101,209,141,10,212,141,0,0,104,7
6,95,228
160 FOR X=1560 TO 1571:READ C:POKE X,C
:NEXT X
170 DATA 104,104,104,168,104,104,170,1
04,104,76,92,228
180 Q=USR(1560,0,6,6)
190 GOTO 190
```

Listing 9-5B uses a slightly different method of screen flipping. The machine language subroutine is accessed on the vertical blank. Every 60th of a second, the raster scan reaches the bottom right corner of the screen. When it shuts itself off to go back to the top-left corner, there is a vertical blank. By flipping the screen at this time, each screen is displayed for 1/60th of a second 30 times in one second. Theoretically, you should not see any flickering here because the eye detects movement at 1/20th of a second. This is the most reliable way to display two screens at the same time.

The machine language subroutine used here is executed during the vertical blank. The computer has certain routines that it must perform during this period. There is time, though, to insert your own code for the computer to execute in addition to its own. The trick is to steal the vector or address location for your own use. In this program we are using the immediate vertical blank vector for our routine. Any time that a machine language subroutine is executed during an immediate vertical blank, it should end with a jump to 58463. A machine language subroutine can also be used during the deferred vertical blank. This routine should end with a jump to 58466. These addresses contain the routines that the computer needs to perform during the vertical

Table 9-2. Machine Language Subroutine—Horizontal Blank.

```
decimal code      assembly language listing
    72            PHA         ;Push the value in the
                              accumulator on the stack.
   169            LDA #4      ;Load the accumulator with 4.
     4
    69            EOR 209     ;Exclusive OR it with the
   209                        contents of location 209.
   133            STA 209     ;Store the result in location
   209                        209.
   165            LDA 206     ;Load the accumulator with the
   206                        value in 206.
    24            CLC         ;Clear the carry.
   101            ADC 209     ;Add with carry the value in
   209                        location 209.
   141            STA 54282   ;Store it - wait for
    10                        horizontal sync.
   212
   141            STA 0       ;Store it in location 0.
     0
     0
   104            PLA         ;Get the value back from the
                              stack.
    64            RTI         ;Return from the interrupt.
```

blank. As the name implies, an *immediate vertical blank* routine is performed as soon as the blanking period begins. A *deferred vertical blank* is executed after other routines are completed.

The routines that you want executed should be fast and to the point. There is a time limit for these routines. If they are too long they could cause problems with the display by changing registers while the scan is turned on. Routines performed during the immediate vertical blank should not exceed 300 machine cycles. Routines using the deferred vertical blank can be about 25,000 machine cycles. Remember, a machine language instruction is not a machine cycle long. Some instruction can use up to 7 machine cycles.

The USR routine passes the address of the machine language subroutine that the vertical blank will use to another machine language subroutine. ATARI has a routine in its operating system to set up immediate and deferred vertical blank routines. To set the address for the routine, you must call a routine at location 58460. When this routine is called, the address of your machine language subroutine must be stored in the Y (low order address) and X (high order address). A 6 must be in the accumulator if it's an immediate vertical blank routine; a 7 if it's a deferred. The machine language subroutine at location 1560 accomplishes this. See Fig. 9-3 for the assembly language listings for these routines.

CREATING SLIDES

With some screen flipping and a disk drive, an entire presentation can be displayed. Slides, pictures, or graphs can be prepared ahead of time and stored on disk. Then, under program control, the slide could be loaded into the computer's memory. While one picture is on the screen, another could be loaded into another part of memory. By pressing a key, the computer would 'flip' to the other picture and load a new picture into the screen memory that is not being shown any longer. The following program will allow you to create slides that will be stored on disk. The program after it will retrieve those slides in any order than you want.

Table 9-3. Machine Language Subroutine for Vertical Blank.

```
76      JMP 58463 ;Jump to this address to
95                computer's routines.
228
```

replace the return from interrupt with this jump to complete the vertical interrupt routines.

decimal code	assembly	language instructions
104	PLA	;Get value from stack (?)
104	PLA	;Get value from stack (0)
104	PLA	;Get value from stack (0)
168	TAY	;Transfer it to index Y.
104	PLA	;Get value from stack. (0)
104	PLA	;Get value from stack. (6)
170	TAX	;Transfer it to index X.
104	PLA	;Get value from stack (0)
104	PLA	;Get value from stack (6)
76	JMP 58460	;Jump to address to set up subroutine to run during
92		vertical blank.
228		

— change for machine language subroutine Boot routine to set up subroutine during vertical blank.

Listing 9-6. Slide Editor

```
10 REM LISTING 9.6
20 REM SLIDE EDITOR
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
32 FOR X=1536 TO 1548:READ B:POKE X,B:
NEXT X:UP=1536:REM ROUTINE TO MOVE UP
34 DATA 104,160,0,200,177,205,136,145,
205,200,208,247,96
36 FOR X=1552 TO 1564:READ B:POKE X,B:
NEXT X:DOWN=1552:REM ROUTINE TO MOVE D
OWN
38 DATA 104,160,255,136,177,205,200,14
5,205,136,208,247,96
40 DIM NAME$(14),N$(8)
50 NAME$(1)=" ":NAME$(12)=" ":NAME$(2)
=NAME$:NAME$="D:":REM CLEAR THE STRING
60 ? "}CLEAR}PLEASE ENTER THE NAME O
F THE PICTURE":INPUT N$:NAME$(3)=N$:NA
ME$(11,14)=".DRW":IF N$="" THEN 50
70 GRAPHICS 21:REM SET DISPLAY FOR GRA
PHICS MODE 5 - NO TEXT WINDOW
80 PM=PEEK(106)-8:POKE 54279,PM:POKE 2
05,0:POKE 206,PM+2:PM=PM*256:REM BEGIN
NING OF PLAYER
85 DLIST=PEEK(560)+PEEK(561)*256
90 POKE 559,46:REM 2-LINE RESOLUTION F
OR PLAYER
100 POKE 53277,3:REM ENABLE PLAYER/MIS
SILE GRAPHICS
110 FOR X=PM+512 TO PM+640:POKE X,0:NE
XT X:REM CLEAR MEMORY FOR PLAYER
120 RESTORE 130:FOR X=PM+525 TO PM+532
:READ B:POKE X,B:NEXT X:REM MAKE CURSO
R
130 DATA 24,24,24,231,231,24,24,24
140 POKE 704,12:REM COLOR PLAYER WHITE
150 HZ=47:VT=0:X=1:Y=0:C=2
160 POKE 53248,HZ
170 IF PEEK(764)=255 THEN 220
180 B=PEEK(764)-30:IF B=1 THEN C=B:GOT
O 220
190 IF B=0 THEN C=2:GOTO 220
200 IF B=-4 THEN C=3:GOTO 220
210 IF B=-6 THEN C=4
220 POKE 764,255:IF PEEK(53279)=6 THEN
```

Listing 9-6. Slide Editor (continued from page 145)

```
 350
230 IF STRIG(0)=0 THEN COLOR C:PLOT X,
Y
240 IF STICK(0)=15 THEN 170
250 IF STICK(0)=7 AND X<79 THEN X=X+1:
HZ=HZ+2:POKE 53248,HZ:GOTO 170
260 IF STICK(0)=11 AND X>1 THEN X=X-1:
HZ=HZ-2:POKE 53248,HZ:GOTO 170
270 IF STICK(0)=13 AND Y<47 THEN Y=Y+1
:Q=USR(DOWN):Q=USR(DOWN):GOTO 170
280 IF STICK(0)=14 AND Y>0 THEN Y=Y-1:
Q=USR(UP):Q=USR(UP):GOTO 170
290 IF STICK(0)=6 AND X<79 AND Y>0 THE
N X=X+1:HZ=HZ+2:POKE 53248,HZ:Y=Y-1:Q=
USR(UP):Q=USR(UP):GOTO 170
300 IF STICK(0)=5 AND X<79 AND Y<47 TH
EN X=X+1:HZ=HZ+2:POKE 53248,HZ:Y=Y+1:Q
=USR(DOWN):Q=USR(DOWN):GOTO 170
310 IF STICK(0)=9 AND X>1 AND Y<47 THE
N X=X-1:HZ=HZ-2:POKE 53248,HZ:Y=Y+1:Q=
USR(DOWN):Q=USR(DOWN):GOTO 170
320 IF STICK(0)=10 AND X>1 AND Y>0 THE
N X=X-1:HZ=HZ-2:POKE 53248,HZ:Y=Y-1:Q=
USR(UP):Q=USR(UP)
330 GOTO 170
340 REM SAVE SCREEN TO DISK
350 SCRB=PEEK(DLIST+4)+PEEK(DLIST+5)*2
56:SCRE=SCRB+959
360 OPEN #2,8,0,NAME$
370 FOR X=SCRB TO SCRE:PUT #2,PEEK(X):
NEXT X:CLOSE #2
380 POKE 53248,0:GOTO 50
```

Line 32 places into memory the machine language subroutine that moves the cursor up. This subroutine will be located on the 6th page of memory. This page is left blank by ATARI for program routines or other uses that you may have for particular memory locations. The contents of this memory will not be changed by pressing the system reset key. The variable UP is set to the first address of this routine.

Line 36 moves the machine language subroutine from line 38 into the memory locations that follow the up routine. This machine language subroutine will move the cursor down. The variable DOWN is set to the address of this routine. It is important that the numbers in these data lines are copied exactly as listed here. The wrong number could cause the program to crash.

Line 40 sets aside string space for the name of the picture that will be drawn on the screen.

Line 50 clears the string and sets the first two bytes of the string to D:. This will be the string that will place the name of the screen into the disk directory.

Line 60 clears the screen and asks you to enter the name of the picture. This name is stored in N$. It is then transferred to NAME$ beginning with the 3rd position. The string will end with the extender .DRW. If you tag an extender to all files created by a particular program, you can retrieve them just by looking at the extenders. This line also checks to see that a name was actually entered. If one hasn't been, it will repeat itself until one has.

Line 70 sets the graphics mode to 5 with no text window. We could actually use any mode for this program. However, since 5 is neither the highest nor lowest resolution, we will use it in this program.

Line 80 sets the beginning of the player/missile graphics area 2K below the end of memory. This value is poked into 54279. Now the computer knows where the player/missile area begins. Memory location 205 is poked with a 0, and location 206 with the beginning of the player/missile area plus 2. We will be using the first player for this program. We will also be using the 2-line resolution for the player. This means that the first player will begin 512 bytes after the beginning of the player/missile area. 512 is 2 pages of memory (512/256=2). This is the value placed in 206. The memory locations 205-206 are used by the machine language subroutine to move the player up and down. It needs to know where in memory the player begins. The value PM is then multiplied by 256 to get the actual decimal value for the beginning for the player/missile area.

Line 85 stores the beginning address of the display list in the variable DLIST.

Line 90 pokes memory location 559 with 46. This enables the player/missile graphics for 2-line resolution.

Line 100 pokes 53277 with 3. This enables the player/missile graphics. If this location is not poked, the players will not be displayed.

Line 110 clears the memory for the first player. This is done because there could be data in those bytes from a previous program or garbage from power-up. This would be displayed on the screen along with our cursor.

Line 120 tells the computer where the information to draw the cursor will begin. The data from the next line is read and stored in the first player area.

Line 140 places the color in the color register for the first player. We will be using white. This number can be changed for any color.

Line 150 sets the variables that will be used in moving the cursor and drawing the colors on the screen. The variable HZ is the horizontal position of the cursor. Position 47 places the cursor along the left side of the screen. It is equivalent to the graphics position for the first column. The variable X is the column for the plot command; the Y is the row. The variable C is used for the color. The program begins with the color set to 2.

Line 160 pokes the value of HZ into location 53248. Now the cursor will appear on the screen.

Line 170 checks location 764 to see if a key has been pressed. When a key has been pressed, the value of this location will not be 255. The computer will move directly to line 220 if no key has been pressed.

Line 180 takes the value in location 764, subtracts 30 from it, and stores it in the variable B. The value at location 764 will not be the ATASCII value of the key pressed. It is a hardware value for that key. If B is 1 (the hardware value was 31), then the 1 key was pressed, and the variable C will be 1. The color in color register 1 will be drawn on the screen.

Line 190 will set the value of C to 2 if key number 2 was pressed. The color in color register 2 will now be used to draw on the screen.

Line 200 sets the value of C to 3 if key 3 was pressed. The color in color register 3 will be used.

Line 210 sets the color value to 4. There is no color value 4 since mode 5 uses three colors plus the background color. This value will have no color value and can be used to erase the lines that were drawn.

Line 220 resets the value in location 764 go 255. This clears the location for a new input. Whenever the keyboard is read this way, the program must reset that register. Then the computer checks to see if the start button has been pressed. If this button has been pressed, the computer will be directed to the part of the program that saves the screen onto disk.

Line 230 checks to see if the red button on the joystick is pressed. This is the only way to draw on the screen. If the red button is not pressed, the cursor may be moved without drawing lines on the screen.

Lines 240-320 check the position of the joystick. If it has not moved, the program repeats itself with line 170. If the joystick has been moved, the program checks to see if the cursor can move. The highest numbered column on the right side of the screen is 79. If the variable X is less than 79, then the cursor can move to the right. The lowest value that X can be is 1. The 0th column does not show up on all screens. Only when the value of X is greater than 1 can the cursor be moved to the left. The top of the screen is row 0. If the variable Y is greater than 0 then the cursor can move up. The bottom row on the screen is 47. The cursor can move down as long as Y is less than 47. Even though the variables X and Y have only 1 added to them whenever the cursor moves, the horizontal and vertical positions must be moved by 2. One row or column in mode 5 is two rows or columns for the player. Every time the horizontal variable (HZ) is changed, the new value must be poked into 53248. To move the cursor up or down, execute the machine language subroutine to move the cursor twice. To move the cursor on a diagonal, both the machine language subroutine and the horizontal register must be used. The lines to move the cursor diagonally (290-320) check two edges of the screen before moving the cursor. The routine ends by sending the computer back to line 170. It continues this loop until the start button or the system reset key has been pressed.

Lines 350-380 save the screen to disk. The variable SCRB is set to the first byte of the screen. The ending byte is calculated by adding 959 to the first byte. (The screen only uses 960 bytes of memory in mode 5.) Line 360 opens the buffer to write a file to the disk. The name of the file is contained in NAME$. The for....next loop in line 370 peeks at every screen memory location from the first byte to the last and stores the bytes on the disk. When the entire screen has been saved to disk, the buffer is closed. The player is moved off the screen and the program returns to line 50. If you want to draw another screen, you can enter its name. If you want to quit, you can press the system reset key.

Listing 9-7. Slide Show

```
10 REM LISTING 9.7
20 REM SLIDE SHOW
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 SCR1=(PEEK(106)-4)*256:S2=SCR1/256:
S1=SCR1-S2*256:REM FIRST SCREEN MEMORY
50 POKE 106,PEEK(106)-4:REM SAVE THAT
AREA OF MEMORY
```

```
60 GRAPHICS 0:REM RESET NEW SCREEN AND
 DISPLAY LIST
70 DIM NAME$(140),N$(8),SCR$(14):REM R
OOM FOR 10 SCREENS
75 NAME$(1)=" ":NAME$(140)=" ":NAME$(2
)=NAME$:SCR$=NAME$:N$=SCR$
80 ? "HOW MANY SCREENS    ";:INPUT S
:REM THREE SPACE AND BACKSPACES
90 IF S>10 THEN 80:REM NO MORE THAN 10
 SCREENS
100 FOR X=1 TO S:? "ENTER ";X;" SCREEN
";:INPUT N$:REM GET THE NAMES OF THE S
CREENS
110 SP=X*14-13:NAME$(SP)="D:":NAME$(SP
+2,SP+9)=N$:NAME$(SP+10)=".DRW":REM EN
TIRE NAME
120 NEXT X
130 GRAPHICS 21:REM SET 2ND SCREEN
140 DLIST=PEEK(560)+PEEK(561)*256:SCR2
=PEEK(88)+PEEK(89)*256:REM SECOND SCRE
EN AREA
150 F=0:FOR X=1 TO S:REM NOW GET THE S
CREENS
160 SP=X*14-13:SCR$=NAME$(SP,SP+13):RE
M NAME OF SCREEN TO BE DISPLAYED
170 SCREEN=SCR2:IF X/2=INT(X/2) THEN S
CREEN=SCR1:REM GET THE RIGHT SCREEN
180 OPEN #2,4,0,SCR$
190 FOR Z=SCREEN TO SCREEN+959:GET #2,
S:POKE Z,S:NEXT Z:CLOSE #2:REM GET SCR
EEN FROM DISK AND PUT IN MEMORY
200 IF X=2 THEN F=1
210 IF F THEN GOSUB 300:REM WAIT FOR S
TART
220 NEXT X
230 GOTO 230
300 IF PEEK(53279)=7 THEN 300
310 IF PEEK(88)<>PEEK(DLIST+4) THEN PO
KE DLIST+4,PEEK(88):POKE DLIST+5,PEEK(
89):RETURN
320 POKE DLIST+4,S1:POKE DLIST+5,S2
330 RETURN
```

Line 40 subtracts 4 from the amount of RAM shown in location 106. This 1K of memory will be set aside for one screen. The variable S2 is the high order byte of the screen memory, S1 is the low order byte.

Line 50 changes the value in 106 by poking it with the value that was there less 4 (1K). Now the computer thinks that it has 1K less of memory and will not touch the memory past this address.

Line 60 resets the display list and screen. Since there is less memory for the computer to use, the display list and screen are relocated.

Line 70 sets aside string space for the names of the screens that will be shown. NAME$ will contain the names of all the screens that will be shown. Each screen name needs 14 bytes or character spaces. There is room for 10 titles as the program stands. Increase the amount of string space by adding multiples of 14 to 140.

Line 75 clears all the strings. There can be data in the strings that can interfere with the program. The easiest solution is to clear out strings that have been fielded.

Line 80 asks for the number of screens that you are planning to display. After the word SCREENS come three spaces and three backarrows. This will clear out a previous entry if it was more than 10. The number of screens that you plan to show are stored in variable S.

Line 90 checks S to see if it is greater than 10. If it is, line 80 will repeat. If you changed NAME$ for more than 10 titles, change the 10 in this line to reflect the number of screens that can be entered.

Lines 100-120 allow you to enter the titles of the screens that you want displayed. Be sure to enter the names correctly. There is no trap in this program for wrong titles. The titles should be entered exactly as you entered them when you saved them in the previous program. The computer will add the D: to the beginning of the name and the .DRW to the end.

Line 130 sets the screen for mode 5 with no text window. This is the mode that the pictures were drawn in, so this is the mode that they must be shown in.

Line 140 finds the beginning of the display list and the beginning of the screen memory for this display list. In addition to the fifth and sixth locations in the display list, the beginning location of the screen memory area is also stored in decimal locations 88 and 89.

Line 150 starts the loop that loads the screens into memory. The loop will be repeated the number of times that S is set for.

Line 160 extracts the name of the first screen from NAME$. The variable SP will be the first byte for the name of the screen. The value of X will be multiplied by 14 (there are 14 bytes for each name) and 13 will be subtracted from that answer. That points SP to the first letter of the name of the screen. Since D: was added to every name, SP should be pointing to a D. The string SCR$ will hold the name of the screen. The next 13 bytes after and including the byte that SB is pointing to will be stored in SCR$.

Line 170 stores the value of SCR2 in SCREEN. SCR2 is the first byte of the second screen. Then the variable X is checked to see if it is even or odd. If X is even, X/2 will be equal to the INT(X/2). The even screens are displayed on the first screen. The variable SCREEN will be changed to the first byte of the first screen. Now the screens will load, alternating between the first and second screen.

Line 180 opens the buffer to read the file from the disk. If SCR$ contains the wrong information, the program will crash.

Line 190 gets the bytes from the disk and stores them in memory from the first byte of the screen to the last. After the entire file is read to the screen, the buffer is closed. You will be able to watch the first picture being drawn on the screen.

Line 200 checks the value of X. If it is greater than 1, then the computer will go to the subroutine that waits for the start button to be pressed.

Line 210 finishes the loop. If there are more screens to be loaded, the computer will loop back and load the next screen. This time, the picture will be loaded on the screen that is not being displayed.

Line 220 loops until the system reset key is pressed.

Lines 300-330 changes the screen that is being viewed. While one screen is displayed, the computer loads in another screen. When the start button is pressed, the computer compares the value of location 88 with the value of the screen in the display list. If these are not the same, the first screen is being displayed and the computer loads the values of locations 88 and 89 into the screen display in the display list. Now the second screen is displayed, and the routine returns to the main program. If the second screen is being displayed, the computer places the first screen address into the display list.

This method of page flipping can be used for any number of screens. The only thing to remember is—DO NOT PRESS START UNTIL THE DRIVE HAS SHUT OFF. The program can also be designed to show a series of slides without any human intervention. Every slide ends with a .DRW. The program can load in the files by checking every entry on the disk for the .DRW. Use the * for the name of the program and .DRW for the extender. The computer will only choose those files that end in .DRW. A timing loop will leave the pictures on the screen long enough to be viewed without getting monotonous.

With a little ingenuity, the program can load entire display lists as well as the pictures so that the slides can use multimode resolutions and alternate character sets.

Chapter 10
Sound Generators

All sounds are generated by the same mechanism—the movement of air. How fast, how long, and with how much force this vibration occurs will govern the sound that we hear. A kitten's purr is quite distinguishable from a lion's roar!

By careful manipulation of the sound registers, a variety of sounds and effects can be created. The four sound or voice channels that the ATARI has can be used alone or in combination with each other. Each channel can be set for a different distortion. This can be used to create strange sound effects for programs.

THE AUDIO CHANNEL CONTROL

Every sound has the same characteristics—attack, decay, sustain, and release. The actual tone depends on the frequency or number of pulses generated in a given time period. The higher the frequency, the higher the note or tone.

In the sound command four different options must be set: SOUND v,p,d,1.

v=voice. There are four different voices or sound channels that can be used. Each is set by using the numbers 0-3. Each voice must be set with a separate sound statement.

p=pitch. This is the frequency of the tone. Any number from 0-255 can be used. The higher the number, the lower the tone. A zero will produce no tone—just a clock from the speaker. The actual sound of the tone will depend on which distortion setting is used.

d=distortion. The distortion here means the noise content of the sound that will be generated. The distortion value will tell the computer how to generate the pulses that will become sound. Only values of 10 or 14 will produce pure tones.

l=loudness or volume. The tones can be loud or soft. Each voice channel can be set independently. The only restriction is that the sum of the volume of the voices used cannot exceed 32.

The following program demonstrates how the tone (p) can vary depending on the level of distortion (d). The number of the pitch as well as the distortion will be printed on the screen while you listen to the tone.

Listing 10-1. Sounds

```
10 REM LISTING 10.1
20 REM SOUNDS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 FOR P=5 TO 255 STEP 5:? P,:REM STEP
 THE NOTES BY 5
50 FOR D=0 TO 14 STEP 2:? D,:REM CHANG
E THE DISTORTION - USE ONLY EVEN NUMBE
RS
60 SOUND 0,P,D,10:FOR Y=1 TO 200:NEXT
Y:REM LISTEN TO THE SOUND
70 NEXT D:? :REM GO THROUGH ALL THE SO
UNDS - THE PRINT LETS THE NEXT PITCH S
TART ON A NEW LINE
80 NEXT P
```

Line 40 begins the for . . . next loop to change the pitch that the sound register will use. The number of the pitch will be printed on the screen.

Line 50 begins the for . . . next loop to change the distortion of the sound register. Only the even numbers from 0-14 can be used for the distortion. Each pitch or tone will be heard in all 8 distortions.

Line 60 plays the sound. The value of the pitch and/or distortion will be different each time this line is executed. The for . . . next loop here gives you time to listen to the sound created.

Lines 70-80 complete the loops.

As you can hear, some of the distortions of a pitch are very similar to others in the same pitch. Others have no sound at all. The only pure tones are generated by distortions 10 and 14.

The volume of the sound can be used to enhance the sound created. Too often the sound is a "set it and forget it" function. Listen again to the sounds generated by the last program. Each sound came on and went off. True sounds do not occur this way. Listen to a piano key being struck, then a drum, and finally a horn (wind instrument) being blown. Each instrument produces its own unique sounds. Although both an organ and a piano are keyboard instruments, they have their own sound qualities. The time that a tone takes to reach its amplitude (height of sound) is called *attack time.* The *decay time* is the time it takes for the sound to start to fade, the *sustain time* is the length of time that the tone is heard (still vibrating). At the *release time,* the tone has faded away completely. The piano is the best example of these four factors. When you strike the key, you immediately hear the tone. It is loud and clear. This is the attack time. That loudness does not continue for an extended period of time. Very quickly the tone starts to fade. This is the decay time. The tone does not die away completely. The length of time that you can still hear the tone is the sustain level. When the tone finally fades completely, it is the release time.

The following program creates an interesting effect when attack, delay, sustain, and release time are inserted into the program.

Listing 10-2. Sounds with Attack and Decay

```
10 REM LISTING 10.2
20 REM SOUNDS-WITH ATTACK & DECAY
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 FOR X=1 TO 14:READ P:REM GET THE TO
NE VALUE
50 FOR V=0 TO 14 STEP 2:SOUND 0,P,10,V
:FOR Y=1 TO 10:NEXT Y:NEXT V:REM ATTAC
K
60 FOR V=14 TO 8 STEP -2:SOUND 0,P,10,
V:FOR Y=1 TO 20:NEXT Y:NEXT V:REM DECA
Y
70 FOR Y=1 TO 100:NEXT Y:REM SUSTAIN
80 FOR V=6 TO 0 STEP -2:SOUND 0,P,10,V
:NEXT V:FOR Y=1 TO 10:NEXT Y:REM RELEA
SE
90 NEXT X:REM DO ENTIRE SONG
100 DATA 121,121,81,81,72,72,81,91,91,
96,96,108,108,121
```

Line 40 reads the pitch values from the data line. These are the tones that the computer will play.

Line 50 simulates an attack. The volume begins at 0 and gradually works its way up to 14. At each level, a timing loop holds the tone and volume level.

Line 60 is the decay. The volume gradually decreases. The timing loop is longer here so that it will take a longer time for the sound to fade.

Line 70 is the sustain. This is the length of time that you will hear the sound. The volume will remain the same.

Line 80 reduces the volume of the sound to 0. This is the release time. The timing loop here gives each tone generated its own time. If there was not a clean break between the tones, the tone would appear to run into each other.

Line 90 completes the loop.

The sound created by this program gives the effect of an accordian. You can almost hear the bellow opening and closing with each attack and decay. The next program uses a slightly different technique to create attack and decay. There is a very definite vibrato to the melody.

Listing 10-3. Sounds with Attack and Decay—Vibrations

```
10 REM LISTING 10.3
20 REM VIBRATIONS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM A$(12)
```

Listing 10-3. Sounds with Attack and Decay—Vibrations (continued from page 155)

```
50 A$="678987654345"
60 FOR X=1 TO 14:READ P:REM GET THE TO
NE VALUE
70 FOR T=1 TO 3:FOR Z=1 TO 12:V=VAL(A$
(Z,Z)):SOUND 0,P,10,V:NEXT Z:NEXT T
80 FOR Y=1 TO 10:NEXT Y:SOUND 0,0,0,0
90 NEXT X
100 DATA 121,121,81,81,72,72,81,91,91,
96,96,108,108,121
```

Line 40 dimensions A$ for the volume settings.

Line 50 sets the volume sequence in A$.

Line 60 reads the pitch from the data line at the end of the program.

Line 70 contains the timing loop and the sound statement. This time the entire range of volumes will be changed rapidly a number of times rather than changing the volume and holding it for a period of time. The volume sequence will be used three times. The variable V will contain the volume. As Z is increased from 1 to 12, the correct volume will be removed from A$.

Line 80 holds the last volume of the sound for a few seconds, then turns it off.

Line 90 completes the loop.

DIRECT ACCESS

Like most commands in ATARI BASIC, there are hardware registers for the sound generators that can be set by poke commands. For each voice that you want to set, there are two addresses that must be poked.

Voice #	Frequency Register	Audio Control
0	53760	53761
1	53762	53763
2	53764	53765
3	53766	53767

The frequency register can be poked with any value from 0-255. This has the same effect as the P variable in the sound command.

The audio control register is a combination of the volume variable (V) and the distortion variable (D). To find the number that should be poked here, multiply the distortion value by 16 and add the volume.

By knowing where these register are, it is possible to use only the register that you need in a program. This will make it execute the sound changes faster because the one register will be set directly, and BASIC will not have to reset registers that are already set; for example, if the volume and distortion are set, and you have no reason to change them, you can poke the frequency register with the pitch or tones. If you use the sound statement, BASIC will recompute the distortion and volume each time it executes the sound command.

There is one more control register for the audio. This register can change the way that the tones are generated. In the previous examples, we used one sound generator. We could easily

change the program to use two, three, or all four generators. For an actual tune, this would generate some harmony. For sound effects, a second or third sound register will add to the realism of the sound. It is also possible to use two registers together to form a 2-byte tone generator. This will increase the range of sounds that the computer can generate.

In the next program, address 53768 is poked with 16. This will couple sound register 0 with 1. Register 0 is the low order byte and register 1 is the high order byte. As the tones begin, you will notice that as long as register 1 is set to 0, the tones sound the same as those generated with a single sound register. Once the value of register 1 changes, the tones become deeper, until they become so very deep that they do not seem to change their pitch.

Listing 10-4. Variations on Tones

```
10 REM LISTING 10.4
20 REM VARIATIONS ON TONES
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 POKE 53768,16:REM LINK VOICES 0 & 1
50 POKE 53761,170
60 FOR P=0 TO 255:POKE 53762,P:? "SOUN
D REGISTER 1=";P,
70 FOR P1=0 TO 255 STEP 5:POKE 53760,P
1:? P1,
80 FOR Y=1 TO 100:NEXT Y:REM TIMER
90 NEXT P1:? :NEXT P
```

Line 40 pokes location 53768 with 16. This joins sound registers 0 and 1 into one register.

Line 50 pokes location 53761 with 170. This is equivalent to a distortion of 10 and a volume of 10.

Line 60 begins the for . . . next loop that changes the pitch content of register 1. This register will increase in value once every time the register 0 reaches 255.

Line 70 begins the for . . . next loop to change the pitch value in register 0. Every time this register cycles from 0 to 255, register 1 will be increased by 1.

Line 80 is a timing loop to give you a chance to hear the tones being generated.

Line 90 continues the loop until all the tones have been generated.

Knowing where the tone generator registers are located is very helpful if you want to write a machine language subroutine for the sounds and/or music that you need for your program. Some of the arcade games play music while the program is running. It is possible to write a machine language subroutine that uses the vertical blank to produce music. The procedure is very similar to the one used in the last chapter to display two screens at the same time.

Listing 10-5. Music: Machine Language Subroutine

```
10 REM LISTING 10.5
20 REM MUSIC - MACHINE LANGUAGE SUBROU
TINE
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
```

Listing 10-5. Music: Machine Language Subroutine (continued from page 157)

```
40 A=PEEK(106)-1:REM SAVE 256 BYTES
50 POKE 106,A:REM MOVE EVERYTHING UP 2
56 BYTES
60 GRAPHICS 0:REM RESET THE SCREEN AND
 DISPLAY LIST
70 POKE 206,1:POKE 0,20:REM SET THE LE
NGTH OF THE NOTE PLAY
80 POKE 207,0:REM OFFSET FOR THE NOTE
BUFFER
90 POKE 203,0:POKE 204,A:REM ADDRESS O
F BUFFER FOR NOTES
100 BUF=A*256:OFF=191:REM BEGINNING OF
 BUFFER AND LENGTH
110 FOR X=BUF TO BUF+OFF:READ C:POKE X
,C:NEXT X:POKE 205,OFF:REM READ IN THE
 NOTES
111 DATA 0,60,64,60,81,60,64,60,96,121
,128,121,162,121,128,121
112 DATA 162,60,64,60,81,60,64,60,72,9
1,96,91,121,91,96,91
113 DATA 108,53,60,53,72,53,60,53,64,8
1,85,81,108,81,85,81
114 DATA 96,47,53,47,64,47,53,47,60,72
,76,72,96,72,76,72
115 DATA 60,85,96,85,60,85,96,85,64,81
,85,81,64,81,85,81
116 DATA 64,96,108,96,64,96,108,96,72,
85,96,85,72,85,96,85
117 DATA 72,108,121,108,72,108,121,108
,81,96,108,96,81,96,108,96
118 DATA 81,114,128,114,81,114,128,114
,85,108,114,108,85,108,114,108
119 DATA 81,85,81,108,72,81,72,108,64,
72,64,108,60,64,60,108
120 DATA 53,85,81,72,64,64,53,53,47,40
,53,40,60,40,64,40
121 DATA 72,72,85,96,85,85,64,64,60,53
,64,60,72,68,72,81
122 DATA 81,81,81,81,81,81,81,81,81,81
,0,0,0,0,0,0
130 FOR X=0 TO 33:READ C:POKE 1536+X,C
:NEXT X
140 DATA 72,169,170,141,1,210,198,206,
208,20,165,0,133,206,164,207,177,203,1
```

```
96,205,208,2,160,255,200,132,207
150 DATA 141,0,210,104,76,95,228
160 FOR X=1570 TO 1581:READ C:POKE X,C
:NEXT X
170 DATA 104,104,104,168,104,104,170,1
04,104,76,92,228
180 Q=USR(1570,0,6,6)
190 ? PEEK(207);"OFFSET",PEEK(206);"CO
UNTER"
200 GOTO 190
```

Line 40 moves the end of memory down by 256 bytes. This is the area that will be our buffer for the music. The music buffer can be as large or small as needed. This program will restrict the length of the melody to 256 bytes.

Line 50 pokes this new end of memory value into location 106. Now the computer will not use the last page of memory.

Line 60 resets the screen and the display list using the value in 106 as the end of memory.

Line 70 pokes a 1 into location 206. This location will be used as the timer or duration of the note being played. We set it to a 1 to begin with as a dummy value. Location 0 will hold the duration value. Location 206 will change while the program is being executed. Every time it counts down to 0, the computer will have to restore the duration value for the next note. Location 0 will not change when the program is executed.

Line 80 uses location 207 as the offset for the buffer. This location will increment every time a note is played. By adding the number in this location to the beginning address of the buffer, the computer will always know where the next note is.

Line 90 sets locations 203 and 204 to the buffer address. Location 203 is set to 0 because we know that the music buffer begins on an even page. Location 204 is set to the high order byte of the beginning of the music buffer. Since we set the last page of memory aside for the music buffer, this value is poked into location 204.

Line 100 computes the decimal address of the first byte of the music buffer by multiplying the value of A by 256. The variable OFF is set to one less than the number of notes that will be played.

Line 110 is the for . . . next loop that moves the data on the next 12 lines into the music buffer. Each number in the data lines represents one note.

Line 130 reads the machine language subroutine into page 6 of the computer's memory. This is the routine that will be executed everytime the vertical blank is executed. Be sure that the data in lines 150 and 160 are entered correctly. If there is an incorrect number in the routine the system will crash or lock up.

Line 160 places into memory the machine language subroutine that will initialize the routine that runs during the vertical blank. Again, it is important that the line of data is entered correctly.

Line 180 executes the second machine language subroutine. The beginning address of the vertical blank routine and the type of routine is passed to the machine language subroutine with the USR command.

Line 190 simply prints the offset value from location 207 and the counter or duration value from location 206. Every time the number in 207 changes, the note will also change. Once it

reaches the 191st note of the melody, it will reset to 0. The counter shows how long the note is being held. Every time it reaches 0, a new note is pointed to by location 207.

To stop this program, you must press the system reset key. If you press only the break key, you will only stop the BASIC program. The machine language subroutine running in the vertical blank will not be affected.

By experimenting with different distortions, durations, and frequencies, you can change the entire effect of the melody.

Chapter 11

Interpreting the Keyboard

There are times when a simple input statement is not the best way to get information from the keyboard: the program may not lend itself to question marks on the screen; you may want single key input; or you may want the program to continue with its task and the keyboard to be read only when a key is actually pressed. (This is sometimes called reading it on the fly.)

When you use the input command to retrieve information from the keyboard the program stops and a question mark appears on the screen. The entry is placed in either a string or a variable. If letters or other characters are entered into a numeric variable instead of numbers, an error will result. This can be resolved by using strings for all inputs. The input can be checked only after the return key has been pressed. Using this command is the simplest way to get information. It is used most often.

The second method of getting an entry is with the get command. Although the program must still stop to get the input, this method is much faster than the previous way. The characters entered can be screened immediately. If the character is not correct, the program can disregard it and wait for another entry to be made. The buffer length can be set for any length, and when full, the program can continue without waiting for the return key. The keys pressed do not have to appear on the screen.

The following program is a useful routine that can be used in any program that needs a read-keyboard routine.

Listing 11-1. Read the Keyboard

```
10 REM LISTING 11.1
20 REM READ THE KEYBOARD
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM BUF$(10)
50 POKE 752,1:? ")":OPEN #2,4,0,"K:":R
EM CLEAR SCREEN - OPEN KEYBOARD FOR A
READ
60 FOR X=1 TO 10:REM BUFFER IS 10 CHAR
ACTERS LONG
```

Listing 11-1. Read the Keyboard (continued from page 161)

```
70 GET #2,B:REM WAIT FOR A KEY TO BE P
RESSED - STORE ATASCII VALUE IN B
80 IF B>127 THEN B=B-128:POKE 694,0:RE
M IF ATARI KEY HAS BEEN PRESSED RESET
90 IF B>90 THEN B=B-32:POKE 702,64:REM
 MAKE IT UPPER CASE
100 IF B<65 THEN 70:REM IT'S NOT A LET
TER
110 BUF$(X)=CHR$(B):REM PUT IT IN THE
STRING
120 POSITION X,5:? CHR$(B):NEXT X
130 ? BUF$
140 CLOSE #2
```

Line 40 sets the buffer length—(BUF$) to 10 characters.

Line 50 removes the cursor, clears the screen, and opens the keyboard for a read.

Line 60 begins the for . . . next loop that will accept keyboard input without using a return key.

Line 70 waits for a key to be pressed. The keyboard must be opened before the get command will work. The ATASCII value of the key pressed will be placed in the variable B.

Line 80 checks the value of B. If it is greater than 127, the ATARI or inverse key was accidentally pressed. Correct this by poking 0 into location 694. Subtract 128 from the value of B to get the normal code for the key pressed.

Line 90 checks the value to see if it is upper or lowercase. If the caps/lower key was pressed, all the inputs would be in lowercase. Rather than check each key against 2 values, it is easier to subtract 32 from the value in B and reset the keyboard for uppercase by poking location 702 with 64.

Line 100 now checks the value of B to make sure that it is a letter. If the ATASCII value is less than 32, it is not a letter, and the program goes back to line 70 to wait for another input. The key that was pressed is disregarded completely. It is not displayed on the screen and it is not stored in the buffer.

Line 110 places the letter for the key pressed into BUF$. As each correct key is pressed and entered, the value of X will increase. This will point to the next vacant position in BUF$.

Line 120 places the letter on the screen. The row is set, the column position will increase with X.

Line 130 prints BUF$ on the screen under the letters that were printed as they were pressed. BUF$ contains exactly what was printed on the screen. If any numbers or control characters were pressed, they were ignored by the program.

Line 140 closes the keyboard.

In the next program, we will use this input routine for a tile game. This game is based on the 5×5 letter tile game. Each tile can only slide to a vacant spot. Try to arrange the letters correctly.

Listing 11-2. Tiles

```
10 REM LISTING 11.2
20 REM TILES
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM BUF$(25),TEMP$(1)
50 BUF$="ABCDEFGHIJKLMNOPQRSTUVWX ":RE
M PLACE ALL THE TILES IN A STRING
60 FOR T=1 TO 5:FOR X=1 TO 25:R=INT(RN
D(1)*25)+1:REM PICK A LETTER
70 TEMP$=BUF$(R,R):BUF$(R,R)=BUF$(26-X
,26-X):BUF$(26-X,26-X)=TEMP$:REM MOVE
THE LETTERS AROUND
80 NEXT X:NEXT T:REM DO IT SEVERAL TIM
ES
90 GRAPHICS 18:REM MODE 2 WITHOUT TEXT
 WINDOW
100 X=1:FOR R=4 TO 8:FOR C=7 TO 11:POS
ITION C,R:? #6;BUF$(X,X):REM PRINT THE
 LETTER
110 X=X+1:NEXT C:NEXT R:REM PRINT THE
WHOLE STRING
120 REM READ THE KEYBOARD
130 POSITION 1,1:? #6;"Press a letter"
:OPEN #2,4,0,"K:":REM OPEN THE KEYBOAR
D FOR A READ
140 GET #2,B:CLOSE #2:REM GET A KEYSTR
OKE
150 IF B>127 THEN B=B-127:POKE 694,0:R
EM RESET THE INVERSE FLAG
160 IF B>90 THEN B=B-32:POKE 702,64:RE
M SET FOR UPPER CASE ONLY
170 IF B<65 OR B>88 THEN 130:REM TRY A
GAIN - NOT A GOOD LETTER
180 FOR X=1 TO 25:IF ASC(BUF$(X,X))=B
THEN 200
190 NEXT X:GOTO 130:REM NOT THERE??
200 R=INT((X-1)/5)+4:REM COMPUTE THE R
OW
210 C=(X-(R-4)*5)+6:REM THIS IS THE CO
LUMN
220 OPEN #2,4,0,"K:":FOR T=1 TO 2:SOUN
D 0,50,10,10:FOR Y=1 TO 10:NEXT Y:SOUN
D 0,0,0,0:NEXT T:REM GET KEY
230 POSITION 1,1:? #6;"Press an arrow"
```

Listing 11-2. Tiles (continued from page 163)

```
:GET #2,B:CLOSE #2:REM GET THE KEYSTRO
KE
240 IF B=42 THEN 300:REM RIGHT ARROW
250 IF B=43 THEN 320:REM LEFT ARROW
260 IF B=45 THEN 340:REM UP ARROW
270 IF B=61 THEN 360:REM DOWN ARROW
280 GOTO 220:REM NOT A 'MOVE' KEY
290 REM NOW SEE IF THE LETTER CAN MOVE
300 IF X=25 THEN 130
305 IF BUF$(X+1,X+1)<>" " OR X/5=INT(X
/5) THEN 130
310 POSITION C,R:? #6;" ":POSITION C+1
,R:? #6;BUF$(X,X):BUF$(X+1,X+1)=BUF$(X
,X):BUF$(X,X)=" ":GOTO 130
320 IF X=1 THEN 130
325 IF (X-1)/5=INT((X-1)/5) OR BUF$(X-
1,X-1)<>" " THEN 130
330 POSITION C,R:? #6;" ":POSITION C-1
,R:? #6;BUF$(X,X):BUF$(X-1,X-1)=BUF$(X
,X):BUF$(X,X)=" ":GOTO 130
340 IF R=4 THEN 130
345 IF BUF$(X-5,X-5)<>" " THEN 130
350 POSITION C,R:? #6;" ":POSITION C,R
-1:? #6;BUF$(X,X):BUF$(X-5,X-5)=BUF$(X
,X):BUF$(X,X)=" ":GOTO 130
360 IF R=8 THEN 130
365 IF BUF$(X+5,X+5)<>" " THEN 130
370 POSITION C,R:? #6;" ":POSITION C,R
+1:? #6;BUF$(X,X):BUF$(X+5,X+5)=BUF$(X
,X):BUF$(X,X)=" ":GOTO 130
```

Line 40 sets aside 25 bytes for the letters (BUF$) and one byte for a temporary storage (TEMP$).

Line 50 places the first twenty-four letters of the alphabet into BUF$. The twenty-fifth letter is the space.

Lines 60-80 shuffle the letters. A random letter (R) is picked. That letter is placed in TMP$. The last letter minus the value of X is placed in the random position. The letter in TEMP$ is moved to the position that was just vacated. The loop continues until all the letters have moved five times. This method ensures that all the letters in the string will be thoroughly mixed.

Line 90 sets the screen for graphics mode 2 with no text window.

Line 100 places all the letters in BUF$ in a 5×5 grid on the screen. The variable X will point to the letter in the string. R is the row and C is the column. To center it on the screen we begin with the 4th row and 7th column.

Line 110 increments X each time a letter is printed. This moves the pointer up one letter.

The loop continues until all the letters have been printed.

Line 130 prints the direction on the screen. First a letter must be pressed. The keyboard is opened for a read.

Line 140 sets the value of the key pressed and closes the keyboard.

Line 150 checks the value of B. If it is greater than 127, then the inverse or ATARI key has been pressed. 127 must be subtracted from this value and location 694 must be poked with a zero to set it back to normal.

Line 160 checks to see if the value is for a lowercase letter. If it is, 32 is subtracted from the value and location 702 is poked with a 64. The keys pressed now will be returned as upper case.

Line 170 checks the value of B. If it is not a letter between A and X the program will return to line 130 for another input.

Line 180 finds the position of the letter pressed in BUF$. The value of X will be its position.

Line 190 sends the program back to line 130 for another input if the letter is not found.

Line 200 calculates the row of the letter. The value of X minus 1 is divided by 5. The integer of the result is added to 4. Since there are 5 letters in each row, dividing X by 5 will give the row number. However, the fifth letter in each row would be put in the wrong row. By subtracting one from X, the resulting integer is the correct row (the first row is counted as row 0). Since we began printing the letters in position 4, this number is added to the result. Now we know which row on the screen the letter is in.

Line 210 calculates the column. 4 is subtracted from the value of X to give the true row. This answer is multiplied by 5 (there are 5 letters in each row) and the result is subtracted from the position that X is pointing to. This answer is added to 6 because it will be in the range of 1-5. The columns on the screen begin with position 7. Now that we know where the letter is located on the screen, we can get a keystroke for the direction that the letter should be moved.

Line 220 opens the keyboard for a read. A short tone will indicate that the program is ready for another input.

Line 230 prints the new message on the screen. This time the program wants an arrow key to be pressed. When a key has been pressed, the computer will close the keyboard.

Lines 240-270 will send the computer to the correct routine depending on whether the up arrow, down arrow, right arrow, or left arrow was pressed. Actually, the code that the program is comparing the inputs to are for the asterisk, the minus sign, the equals sign, and the plus sign. By using these codes instead of the control-arrow codes, the program is truly a one keystroke program.

Line 280 sends the program back to line 220 if an arrow key was not pressed.

Line 300 checks to see if X is pointing to 25. If it is, then it is at the end of the buffer and the letter cannot be moved to the right. The program will send the computer back to line 130 for another letter input.

Line 305 checks to see if the next place in the buffer is empty. It also checks to see if this letter is in the fifth column on the screen. If either of these conditions are true, then the letter cannot move to the right, and the computer will go back to line 130 for a new letter.

Line 310 erases the letter from its position on the grid and moves it over one to the right. Then it moves the letter up one space in the buffer. The position that the letter was occupying becomes a space. The program continues with line 130.

Line 320 checks to see if X is the first position. This routine moves the letters to the left. If the letter to be moved to the left occupies the first position in the string, it cannot be moved to the left. The program returns for another letter input.

Line 325 checks to see if the letter occupies the first column of the grid. It also checks the position just before it in the buffer. If this position is full the letter cannot be moved to the left. If the letter cannot be moved because of either of these conditions, the program waits for another input at line 130.

Line 330 uses the same procedure, but in reverse, to move the letter to the left. The letter is first erased from the screen, and then reprinted in the new position. The letter is moved down one position in the buffer and the old position becomes a space.

Line 340 checks to see if the row (R) is 4. If it is, then the letter is in the top row. This routine moves the letter up one row. A letter in the first row cannot be moved up.

Line 345 checks the buffer five positions before the position of the letter that will be moved. The letter will move up one row. The square that it is moving to is five positions before its position. If that position is not empty, the program will go back and wait for a new command.

Line 350 erases the letter from the grid and reprints it one row up. It then moves the letter up five positions in the buffer and replaces it with a space.

Line 360 checks the value of the row (R) to see if it is the last row of the grid. This routine moves the letter down one row. The letters cannot be moved past the last row.

Line 365 checks the buffer five positions past the letters position to make sure that it is a space. If it is not, the program will go back to line 130 for another entry.

Line 370 erases the letter and prints it one row down. It then moves the letter over five positions in the buffer. The letter's original position becomes a space.

This entire game is played with single keystroke entries. There are actually two entries for each move, but each is treated separately. If the first entry is correct, the program will ask for the second. Inputs that are not considered legal are ignored.

KEYBOARD CODE

The third method of entry from the keyboard is to read the keyboard on the fly. This means that the computer is busy running the program, but it keeps checking to see if a key was pressed. If it was, then it will process that information. If no key was pressed, it will continue with the main program.

This method could be used in a game where the computer is working out some possible moves. If the player had to wait for the computer to make its move after the player made his/her move, the game could be excessively long. If, however, the computer could do its thinking while the player did, the time that the computer needed for its moves would appear to be shortened. A chess game is a good example of a situation where you would want the computer to think while the player did.

The keyboard code is not ATASCII. It does not seem to follow any pattern. The only exception is that there is one bit set in the code if it's an uppercase letter, or if the control key has been pressed. Table 11-1 shows the hardware key code and the corresponding key. Try the following one line program:

```
10 ? PEEK(764),:GOTO 10
```

There will be a stream of 255s on the screen until a key is pressed. Once a key has been pressed, that value will be printed on the screen until another key has been pressed. This is the hardware code for the keys. To convert the keycode into ATASCII so that you would know which key was actually pressed could be done with all the possible characters in a string buffer. There is

no need to do duplicate work. Beginning with memory location 65278, all the ATASCII codes are listed according to keycode.

The following program will show you the internal or hardward code of the key and the character.

Listing 11-3. Keyboard Conversion

```
10 REM LISTING 11.3
20 REM KEYBOARD CONVERSION
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 CONVERSION=65278
50 ? "}CLEAR}":POKE 752,1:REM CLEAR SC
REEN - REMOVE CURSOR
60 POSITION 3,5:? "PRESS ANY KEY ON TH
E KEYBOARD          CEXCEPT THE BREAK K
EY]"
70 IF PEEK(764)=255 THEN 70:REM NO KEY
 HAS BEEN PRESSED
80 KC=PEEK(764):POKE 764,255:REM SAVE
THE CODE - CLEAR THE LOCATION
90 POSITION 3,10:? "INTERNAL CODE OF K
EY PRESSED       ";KC:REM SHOW HAR
DWARE VALUE - 6 SPACES-5 ESC-BKARROW
100 POSITION 3,15:? "KEY PRESSED -
  ";CHR$(PEEK(CONVERSION+KC)):REM
 6 SPACES-5 ESC-BKARROW
110 GOTO 60
```

Line 40 sets the variable CONVERSION to the first byte of the ATASCII values. This location is the ROM or the operating system.

Line 50 clears the screen and removes the cursor.

Line 60 asks you to press any key with the exception of the break key.

Line 70 loops until a key has been pressed. When a key has been pressed, the value of location 764 will not be 255.

Line 80 stores the value of location 764 in variable KC. Location 764 is poked with 255. This clears it for another input.

Line 90 prints the internal or hardware code for the key that was pressed. Be sure to enter six spaces and five escape control-backarrows in the print line. This will erase the previous code from the screen.

Line 100 adds the keycode to the first byte of the table in ROM. The program then prints the character of the peek of that location. This character will be in lowercase unless the shift key or control key was pressed for the input.

Line 110 loops back to line 60 for another entry.

Because of this table in ROM, it is very easy to convert keycode into the actual ATASCII values for a program.

letter or character	ATASCII code	hardware code
<space>	32	33
!	33	95
"	34	94
#	35	90
$	36	88
%	37	93
&	38	91
'	39	115
(	40	112
)	41	114
*	42	7
+	43	6
,	44	32
-	45	14
.	46	34
/	47	38
0	48	50
1	49	31
2	50	30
3	51	26
4	52	24
5	53	29
6	54	27
7	55	51

Table 11-1. ATASCII and Hardware Values.

READING THE KEYBOARD

The following program is an example of how the keyboard can be read while the computer is executing a main program. This simple keyboard program will keep letters flying across the screen. If you press the correct letter, the letter will stop and you will be awarded points. The entire time that the letters are moving across the screen, the computer is checking location 764 to see if a key has been pressed.

Listing 11-4. Letter Attack

```
10 REM LISTING 11.4
20 REM LETTER ATTACK
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 CONVERSION=65278
50 GRAPHICS 17:TL=30:REM MODE 2 NO TEX
T WINDOW.
60 K=INT(RND(1)*26)+65:REM GET ATASCII
 VALUE FOR CHARACTER - DON'T USE A SPA
CE
70 POSITION 2,0:? #6;"score";SC
80 R=INT(RND(1)*22)+1:REM PICK A RANDO
M ROW FOR THE LETTER
90 FOR X=1 TO 19:POSITION X-1,R:? #6;"
 ";CHR$(K)
100 FOR T=1 TO TL
110 IF PEEK(764)=255 THEN 140:REM NO K
EY HAS BEEN PRESSED
120 KC=PEEK(764):POKE 764,255:REM SAVE
 THE CODE - CLEAR THE LOCATION
130 IF PEEK(CONVERSION+KC)-32=K THEN S
C=SC+20-X:T=TL:X=19
140 NEXT T
150 NEXT X:POSITION 19,R:? #6;" "
160 IF SC<1000 THEN 60
170 IF SC>=1000 THEN GRAPHICS 17:POSIT
ION 4,4:? #6;"YOU MADE IT":POSITION 4,
6:? #6;"score - ";SC
180 ? #6;"PLAY AGAIN":OPEN #2,4,0,"K:"
190 GET #2,B:CLOSE #2:IF B=89 OR B=121
 THEN TL=TL-5:GRAPHICS 17:GOTO 60
```

Line 40 sets the variable CONVERSION to 65278. This is the first byte of the ROM table to convert the internal or hardware code into ATASCII.

Line 50 sets the screen to mode 2 with no text window. The variable TL will be used in the timing loop. It can be changed to any number to make the letters move faster or slower.

Line 60 chooses a number for the letter that will fly across the screen. It chooses a number

from 0-25 because there are 26 letters in the alphabet. This number is added to 65. The letter A is ATASCII 65.

Line 70 prints the current score on the screen. This line will update the score after every letter.

Line 80 picks a random row for the letter to travel on. The letter cannot be on the 0th line because that's where the score is printed.

Line 90 begins the for . . . next loop that moves the letter across the screen. The letter begins on the left side of the screen and continues across to the right.

Line 100 begins the timing loop. If there was no timing loop, the letter would travel across the screen too fast to be read.

Line 110 checks location 764 for a value other than 255. If no key has been pressed, the computer is directed to line 140 to continue the timing loop.

Line 120 stores the value of location 764 in the variable KC. The location is cleared by poking it with 255.

Line 130 checks the key pressed with the letter that is being displayed. The program looks at the peek of the keycode added to the first byte of the ROM table. Since this is the lowercase ATASCII code for the letter, 32 must be subtracted from the code. If this result is equal to the value of KC, the correct key has been pressed. A new score is calculated by adding 20 (20-X) to the old score. X is the horizontal position of the letter at the time the correct response key was pressed. The variables T and X are set to their highest values and the for . . . next loop continues. Since T and X are at their limit, the program will continue with line 160. If the letter reaches the edge of the screen, it will be erased from the screen. If the correct key has been pressed, the letter will stop on the screen and remain there until another letter erases it.

Line 160 sends the computer back to line 60 for another letter if the score is less than 1000.

Line 170 will end the game if the score reaches or surpasses 1000. The screen will clear and a message will appear. The ending score will also be printed.

Line 180 asks if you want to play again. The keyboard is opened for a read.

Line 190 gets an input from the keyboard and closes it. If the letter Y has been pressed, the program will subtract 5 from the counter and go to line 60 for another game. Each time a new game is played, the letters will fly faster on the screen. If any other key is pressed, the program will end.

Chapter 12

Understanding the Screen Editor

Everything that you see on your screen is processed by the screen editor. It keeps track of where the cursor is, where the screen memory begins, whether or not there is a text window, the tab locations, the mode the screen is in, the right and left margins, etc. This chapter will list most of the memory locations that the screen editor uses and the function of each location. These locations can be changed by the program. If they are changed correctly, they can add features to your program. If they are not, the program could crash or produce results that are less than desirable.

GET/PUT CHARACTERS

When the program contains the locate command, it is getting a character from the screen. This command contains the row and column that you want looked at. It returns the ATASCII value of the character at that location.

Listing 12-1. Locate, Poke, and Peek

```
10 REM LISTING 12.1
20 REM LOCATE, POKE AND PEEK
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:REM CLEAR THE SCREEN
50 ? "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
60 LOCATE 2,0,B:POSITION 2,0:? CHR$(B)
:REM GET THE ATASCII VALUE AND PUT CHA
RACTER BACK
70 POSITION 5,5:? "THE ATASCII OF ";CH
R$(B);" IS ";B:REM SHOW ATASCII VALUE
80 MEM=PEEK(88)+PEEK(89)*256:REM BEGIN
NING OF SCREEN MEMORY
90 B1=PEEK(MEM+2):REM SAME LOCATION
100 POSITION 5,6:? "POSITION IN CHARAC
TER SET ";B1:REM SHOW VALUE
110 POKE MEM+42,B1:REM MOVE IT DOWN ON
E LINE
```

Line 40 sets the graphics mode and clears the screen. The graphics command must before using the locate command.

Line 50 prints the letters of the alphabet across the screen.

Line 60 uses the locate command to get the ATASCII value of the character in location 2,0. The value will be stored in variable B. Immediately after using the locate command, the character must be printed back into that location. When the location is examined for its contents, the location becomes blank. To reinstate the character, the print command must be used.

Line 70 shows the ATASCII value of the character at the location 2,0.

Line 80 calculates the beginning address of the screen. This address is stored in locations 88 and 89.

Line 90 uses the peek command to look at that memory location. This time the result is not the ATASCII value, but the position of the character in the character set.

Line 100 prints this value on the screen.

Line 110 pokes this value back into memory. The location that will be poked is 40 more than the address that was peeked at. This will move the character down one line.

As you can see, you can get different results depending on which commands you use to look at a character on the screen. The locate command is the easiest to use since it calculates the screen position, and returns the ATASCII value of the character. There may be times, though, when the program that you are writing will work better with pokes and peeks.

CONTROL CHARACTERS

There are sixteen control codes for the ATARI computer. Each of these codes has its own ATASCII value. They can be printed to the screen within a string, within quotes, or with the CHR$ command. The following list gives the ATASCII code for the control codes and their function.

ATASCII code (decimal)	Command and function
27	**Escape:** The character following this code will be treated as data. This means that if the next character you want is a control character (for example, clear screen), pressing the escape key first will display the character rather than clearing the screen. Use the escape key when there are control characters in a string or line that will be printed on the screen.
28	**Cursor up.** The cursor is moved up one line on the screen. If the cursor is on the top line, it will appear on the bottom of the screen. This key is used in editing. Under program control it could space messages that are printed on the screen without using the position command.
29	**Cursor down.** This code moves the cursor down one line on the screen. If the cursor is on the bottom line, it will move to the top of the screen. Again, this can be used instead of the position command, when you are printing text on the screen.
30	**Cursor Left.** This code moves the cursor one position to the left. If the cursor is at the left side of the screen, it will move to the right side of the screen, but remain in the same screen line. This character is often used to clear a previous input to the same prompt. If the previous line prints three or four spaces, then

ATASCII code (decimal)	Command and function
	printing one fewer cursor-left character will clear any information that was on the line, and the question mark will appear in the correct position for the next entry.
31	**Cursor Right.** This code moves the cursor one position to the right in the same line. If the cursor is at the right edge of the screen, it will move to the left edge in the same line.
125	**Clear.** This code represents one of the most frequently used control characters. It clears everything from the screen and homes the cursor. The home position of the cursor is the upper left corner of the screen.
126	**Backspace.** This code moves the cursor back one space. If the cursor is at the left margin, and this line is the beginning of a logical line, the cursor will not move any further. If this line is the continuation of a previous line, the cursor will move to the right edge of the screen one line higher. (There are three screen lines to one logical line.) When the cursor moves to the left, it removes or deletes the characters on the screen.
127	**Tab.** This code moves the cursor several positions to the right. The tab positions are set when the computer is turned on. They can be reset under program control. The computer considers three screen lines to be one logical line for the tab function.
155	**End of Line.** This code ends the logical line for the computer. It is also used to indicate that the return key has been pressed to enter an input. When the screen editor receives an end-of-line (EOL) it returns the cursor to the left side of the screen, one line down. Printers usually use this code to issue a carriage return and line feed. Disks and cassettes use it to indicate the end of a record.
156	**Delete Line.** This code removes all the information that is on the line that the cursor is on. This line is cleared on the screen. If there is text printed below the line, all the lines move up one line.
157	**Insert Line.** This code adds a blank line in the line that the cursor is in. If there is information in this line, it and any text below it are moved down one line. Any information on the last line of the screen will be moved off the screen.
158	**Clear Tab.** This code removes the tab indicator from the point that the cursor is at. If the tab was not set at this location, nothing happens. There is no clear-all-tabs command, but this function can be accomplished with pokes.
159	**Set Tab.** This code places a tab indicator at the location of the cursor. Since the tabs can be set for three screen lines, it is important to know exactly where the cursor is.
253	**Bell.** This code makes a tone using the speaker on the computer. This sound is more like a squawk than an actual bell. This character has no effect on the display.
254	**Delete Character.** This code removes the character "under" the cursor. If there are characters in the line to the right of the cursor, they will move one position to the left. If the logical line is longer than one screen line, the contents of the lines below the line with the character that is being deleted will move up also.

ATASCII code (decimal)	Command and function
255	**Insert Character.** This code adds one position to the line. A space is inserted at the position of the cursor. The character under the cursor and any characters to the right of the cursor are moved one position to the right.

In some programs, it is possible to place machine language subroutines into strings. If it is a relocatable subroutine, this is a good place to store it since it can be accessed by using the USR(ADR(STRING$)) command. But, very often, some of the codes for the subroutine are the control codes listed above. If you try to print the string, you may find that you cannot see all the characters in it because the control characters don't print! The next program shows you which location to poke to make control characters visible on the screen.

Listing 12-2. Printing Control Characters

```
10 REM LISTING 12.2
20 REM PRINTING CONTROL CHARACTERS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM A$(20)
50 A$="}}~";REM PUT CONTROL CH
ARACTERS IN THIS STRING
60 POKE 766,1
70 ? "A$ CONTROL CHARACTERS ";A$
80 POKE 766,0
```

Line 40 sets aside 20 places for characters in A$.

Line 50 places an assortment of control characters into A$. These are - bell, clear, insert character, delete character, cursor down, cursor up, cursor right, cursor left, tab, insert line, and delete line. To print these characters within the quotes, use the escape control keys for all but the last two; use the escape shift keys for those.

Line 60 pokes location 766 with a 1. Anytime this location is not zero, the control characters will be printed on the screen as characters.

Line 70 prints the contents of A$ on the screen. If you do not clear the screen before running this program, you will see that A$ is printed exactly as it is set in line 50.

Line 80 resets location 766 with a 0. Now if you try to print A$, you will get the bell, the screen cleared and all the other control characters executed.

OTHER MEMORY LOCATIONS

The following list of memory locations are used by the screen editor, display handler, etc, to display information on the screen. They are all located in RAM, so they can be changed under program control. Some values may be immediately reset by the operating system; others will be ignored; and changing some can cause strange results, or make the system crash.

Memory Location	Function
88,89	This is the beginning of the screen memory. When the contents of these locations are changed, the computer will print in memory other than that being displayed on

Memory Location	Function
	the screen. This location was changed in the program *Slide Shows* (Chapter 9).
675	This byte and the next 14 bytes make up the bit map for the tab. Every bit that is set is a tab. For example, input POKE 675,17. Now press the tab key. The cursor will stop under the E in READY—the fourth screen position, and under the space after the Y—the eighth screen position. All tabs can be cleared by this statement.

```
FOR X=675 TO 689:POKE X,0:NEXT X
```

694	This location sets or clears the inverse flag. If this location is 128, all the characters will be treated as inverse characters. It must be set to 0 for normal characters. This location was used in Chapter 11.
702	This location is set for lower or uppercase letters. If this location is 64, the shift-lock has been set and the characters are uppercase. If it is 0, the characters are lowercase. This location was also used in Chapter 11.
703	This location can only be 4 or 24. If it is 24, it sets the normal screen size. A 4 sets the text window at the bottom of the screen. Poke this location with a 4; then list a program. The entire program will scroll in the bottom four lines of the screen.

By experimenting with these locations, you will learn how to create the effects that you need or want for your programs.

Chapter 13

Disk Use

Convenient program and data storage is provided by the floppy disk system available with the ATARI. By understanding how the computer handles the disks and how the disks themselves are organized, you can better control and utilize the system.

DISK FILE MANAGER

The file manager provides the commands that allow BASIC to access the disk drive. Up to four drives can be accessed with the manager that comes with the ATARI Disk Operating System (DOS). Throughout this chapter and the next, the ATARI DOS will be used for the examples. If you are using a different DOS on your system, some of the commands or formats may differ.

IOCBs

The ATARI computer has a portion of memory set aside for input and output control. This area of memory is called the input/output control buffer (IOCB). The memory locations in this area are set for the device(s) that will be handled. This area is not exclusively for the disk drives. It can also be used by the screen editor, cassette, or any other device that can access it. It can also be used under program control.

The IOCB stores information concerning the device that is being used, the process that will be performed (writing to the device or reading from it) the length of the buffer that will be written from or read into, and location of the buffer.

CIO Functions

The central input/output (CIO) functions control most of the disk operations. Each disk operation has its own specific function. Some operations have options that can be accessed by using the correct auxiliary code.

Open. The open command can be used to create a new file, append an existing file, or update an existing file. The file can also be opened for a read. To use the open command, you must specify which drive, the name of the file and the option, for example, OPEN #2,8,0,"D:NAMES". The file NAMES would be opened using buffer #2. It is opened for a write only. If a file already exists with that name, it will be written over. If it does not exist, it will be created. Its name will be added to the directory.

Using the format OPEN #2,9,0,"D:NAMES" opens the file for an append. The file will not

be destroyed. The information sent to it will be placed immediately following the data that is already there.

If you use the format OPEN #2,12,0,"D:NAMES", the file will be opened for update. Any part of it can be changed without affecting the rest of the file if the file was created properly.

You can also use the open command to read the directory from BASIC by using OPEN #2,6,0,"D:*.*". In the next program, you can read the directory from BASIC and run any program by entering that program's number.

Listing 13-1. Directory Listing

```
10 REM LISTING 13.1
20 REM DIRECTORY LISTING
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM DIR$(768),BUF$(14)
45 DIR$(1)=" ":DIR$(768)=" ":DIR$(2)=D
IR$
50 ? "}CLEAR}":REM CLEAR THE SCREEN
60 OPEN #2,6,0,"D:*.*":N=1:X=1:REM OPE
N THE FILE FOR THE DIRECTORY
70 INPUT #2,BUF$:REM GET AN ENTRY
80 DIR$(X,X+11)=BUF$(3,13):REM PLACE I
T IN THE DIRECTORY BUFFER
90 IF ASC(BUF$(1,1))<58 AND ASC(BUF$(1
,1))>47 THEN N=N-1:CLOSE #2:GOTO 110
100 X=X+12:N=N+1:GOTO 70
110 R=1:C=2:FOR X=1 TO N:POSITION C+(
NOT INT(X/10)),R:? X;".";DIR$(X*12-11,
X*12):REM PRINT THE FILE
120 C=24-C:IF C=2 THEN R=R+1:REM NEXT
COLUMN, NEXT ROW
130 NEXT X:REM FINISH DIRECTORY
140 TRAP 140:POSITION 2,22:? "ENTER NU
MBER OF PROGRAM      ";:INPUT P
150 IF P>N OR P<1 THEN 140
160 BUF$(1)=" ":BUF$(14)=" ":BUF$(2)=B
UF$:BUF$="":BUF$="D:":FOR X=P*12-11 TO
 P*12-1:IF DIR$(X,X)=" " THEN 180
170 NEXT X:X=P*12-3
180 X=X-1:BUF$(3)=DIR$(P*12-11,X):X=LE
N(BUF$)
190 X=X+1:BUF$(X,X)=".":X=X+1:BUF$(X,X
+2)=DIR$(P*12-3,P*12)
200 RUN BUF$
```

Line 40 sets aside 768 bytes for the directory and 14 for the buffer. There is a maximum of 64 files that can be stored on disk. There are 12 bytes fielded for each file.

Line 45 clears the garbage from the string.

Line 50 clears the screen.

Line 60 opens buffer #2 to read the directory. The number 6 indicates a directory read. The asterisks (*) are used for the file name so that all the file names will be read. The variable N will record the number of files read. X will point to the position in DIR$ that the file name will begin at.

Line 70 gets an input from buffer #2. The contents of this input is stored in BUF$. The program is reading the entry from the buffer, not from the disk.

Line 80 places the contents of BUF$ into DIR$. All the file names will be stored this way.

Line 90 checks the first character of BUF$. If it is a number, there are no more files in the directory. One is subtracted from N so that N will be the number of files listed in the directory.

Line 100 adds 12 to the value of X. This moves the pointer up 12 bytes to point to the beginning of the next field. The variable N is incremented by 1, and the program continues with line 70.

Line 110 uses the variable R to indicate the row that the file information will be printed on and the variable C for the column. The for....next loop accesses DIR$ to print the names of the files on the screen. The column position is calculated to keep them straight. When X is greater than 10, the number (X) must be printed one column to the left to keep the numbers in a straight line. The logical operation NOT returns a 1 when the integer of C/10 is 0. It returns a 0 when it is a number greater than 0. By adding this value to the variable C, we can keep the numbers in a straight column. The number (X) of the file and the file name are printed on the screen. Again, X is used to calculate the beginning and ending positions of the file name.

Line 120 recalculates the value of C. The two positions that the column can be are 2 and 22. By subtracting the value of C from 24, the variable C will be 2 and 22 alternately. Everytime C is 2, the variable R is incremented so that the next file name can be printed in the next row on the screen.

Line 130 continues the loop.

Line 140 asks for the number of the program that you want to run. The line ends with five spaces and three backarrows. The trap keeps the program from crashing if a letter or other character is entered instead of a number. The number of the program is stored in the variable P.

Line 150 checks the value P. If it is larger than the number of the last program on the screen, or less than 1, the program returns to line 140.

Lines 160-180 place the name of the program into BUF$. The first two characters of the string are set to D:. This is the code that the computer needs to access the disk. The program looks for the first space. The part of the name from the first character to the character before the first space is placed in BUF$.

Line 190 places the . after the name of the program. The extender is added to the name.

Disk Buffer

Located in the IOCB is the address of the disk buffer. When the computer inputs information from the disk, it moves the data into the buffer. When the program uses the get command or put command, it goes into this buffer to retrieve or place information. The get and put commands work with only one byte at a time.

The input command also gets its information from the buffer. All the data up to the end-of-line code is placed into the string. If the string is not long enough for the information, it will be lost.

The following program will print a hard copy of a program that has been listed to the disk. The length of the line can be specified. The program uses the data in the disk buffer to print the listing.

Listing 13-2. Print from Disk

```
10 REM LIST 13.2
20 REM PRINT FROM DISK
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 ? "}CLEAR}ENTER THE LENGTH OF A LIN
E";:INPUT PL
50 ? "ENTER THE NUMBER OF LINES TO A P
AGE";:INPUT LN
60 OPEN #2,8,0,"R4:"
70 OPEN #1,4,0,"D:LISTING"
80 L=1
90 TRAP 300:FOR C=1 TO PL
100 GET #1,B
110 ? #2;CHR$(B);
120 IF B=155 THEN 140
130 NEXT C:? #2;CHR$(155);
140 L=L+1:IF L<LN THEN 90
150 FOR L=1 TO 10:? #2:NEXT L:GOTO 80
300 CLOSE #1:CLOSE #2:END
```

Line 40 clears the screen and asks for the length of the line. This value is stored in variable PL. It can be any value. Of course, if your printer can only print 40 characters on a line, a number larger than 40 would not work.

Line 50 asks for the number of lines to a page. This is the number of lines that you want printed at one time. After every page, the program will print ten line feeds to separate the pages.

Line 60 opens buffer #2 for the printer. The printer that I use is a serial printer out of port #4 of the interface.

Line 70 opens buffer #1 to the disk drive. The program that you want printed should be listed to the disk with the LIST "D:LISTING" command instead of being saved. Use the name LISTING for the program. By listing the program to the disk, it is not saved in the token form, but byte for byte the way it appears on the screen when it is listed.

Line 80 sets the variable L to 1. This variable will count the number of lines that are printed.

Line 90 sets a trap for line 300. We don't know how long the program is, so when an error occurs, we will end the program. The for....next loop to print the program begins here.

Line 100 uses the get command to retrieve a byte from the buffer. The disk will turn on and 256 bytes will be read into the buffer. This command will get each byte from the buffer one at a time. It keeps track of which byte it got last, so it always gets the next byte. When the end of the buffer is reached, the next 256 bytes will be read in off the disk.

Line 110 prints the CHR$ of the byte to the printer. The semicolon after the CHR$ keeps every character on the same line until the carriage return and line feed (EOL) is issued to the printer.

Line 120 checks the value of B. If it is 155, the EOL has been sent to the printer, and the printer has returned to the beginning of the line and fed the paper up one line. The program is directed to line 140 since there will be no more characters printed on this line.

Line 130 continues the loop. After the number of characters specified by PL is sent to the

printer, the program sends an EOL to the printer. If there are more characters for this program line, they will be printed on the next line.

Line 140 adds one to the value of L. This is the number of lines that have been printed. If the number of lines required for one page have not been printed, the program will go back to line 90. This will start the for....next loop again.

Line 150 will separate one page from the next. Ten prints will be sent to the printer. This number can be changed if you want more or less spacing between the pages.

Line 300 closes the buffers and ends the program.

SPECIAL FUNCTIONS

There are some functions that can be accessed both from the DOS and BASIC. Although they are not supported by one word commands, they can be executed with the XIO command. They are: lock, unlock, delete, rename, and format. The XIO commands are:

Operation		Command
LOCK	XIO	35,#1,0,0,"D:name"
UNLOCK	XIO	36,#1,0,0,"D:name"
DELETE	XIO	33,#1,0,0,"D:name"
RENAME	XIO	32,#1,0,0,"D:name newname"
FORMAT	XIO	254,#1,0,0,"D1:"

These commands can be used within a program to access the disk. The following program includes some of these commands and the note and point functions.

Listing 13-3. Calendar

```
10 REM LISTING 13.3
20 REM CALENDER
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM DAY$(30),BUF$(20),MT$(8),TEMP$(
25)
50 ? "}CLEAR}":POSITION 5,2:? "PLEASE
CHOOSE:":POSITION 5,4:? "1.  FORMAT NE
W DISK"
60 POSITION 5,6:? "2.  CHECK CALENDER"

70 TRAP 70:POSITION 10,8:? "  ";:INP
UT C:REM 2 SPACES & 2 LEFT ARROWS TO C
LEAR ENTRY
80 IF C<1 OR C>2 THEN 70:REM ONLY ACCE
PT 1 OR 2
90 ON C GOTO 100,250
100 POKE 752,1:? "}CLEAR}":POSITION 2,
10:? "THIS SELECTION WILL FORMAT A DIS
K FOR THE CALENDER PROGRAM.  PLACE A"
110 ? "NEW DISK INTO THE DRIVE AND PRE
```

Listing 13-3. Calendar (continued from page 181)

```
SS     RETURN ANY OTHER KEY WILL RETUR
N YOU  TO THE MENU"
120 IF PEEK(764)=255 THEN 120
130 LN=50:IF PEEK(764)=12 THEN LN=150
140 POKE 764,255:GOTO LN
150 XIO 254,#1,0,0,"D1:":REM FORMAT TH
E DISK
160 FOR X=1 TO 12:READ BUF$:DAY$(1)="
":DAY$(30)=" ":DAY$(2)=DAY$:OPEN #2,8,
0,BUF$
170 FOR D=1 TO 31:DAY$(1)=STR$(D):DAY$
(30)=" ":PRINT #2;DAY$:NEXT D
180 CLOSE #2:NEXT X:REM CREATE THE MON
TH FILES
190 DATA D:JANUARY,D:FEBRUARY,D:MARCH,
D:APRIL,D:MAY,D:JUNE,D:JULY,D:AUGUST
200 DATA D:SEPTEMBER,D:OCTOBER,D:NOVEM
BER,D:DECEMBER
210 POKE 752,0:GOTO 50
250 ? "}CLEAR}":POSITION 5,10:? "ENTER
 THE MONTH";:INPUT MT$
260 TRAP 270:RESTORE 190:FOR X=1 TO 12
:READ BUF$:IF LEN(MT$)+2<=LEN(BUF$) TH
EN IF BUF$(3,LEN(MT$)+2)=MT$ THEN 280
270 NEXT X:GOTO 50
280 TRAP 280:? :? "   ENTER THE DAY";:
INPUT D
290 TRAP 40000:OPEN #2,4,0,BUF$:FOR X=
1 TO 31:NOTE #2,S,B:INPUT #2,DAY$
300 IF VAL(DAY$(1,2))<>D THEN NEXT X:C
LOSE #2:GOTO 50
310 CLOSE #2
320 ? "}CLEAR}":? "DATE REQUESTED - ";
MT$;" ";D
330 ? :? DAY$
340 ? :? "PRESS K-TO KEEP INFORMATION
& RETURN  TO MENU"
350 ? :? "PRESS U-TO UPDATE INFORMATIO
N"
360 ? :? "PRESS D-TO REVIEW ANOTHER DA
TE"
370 OPEN #2,4,0,"K:":GET #2,C:CLOSE #2
380 IF C>127 THEN C=C-128:POKE 694,0
390 IF C=75 OR C=107 THEN 50
```

```
400 IF C=85 OR C=117 THEN 450
410 IF C=68 OR C=100 THEN 250
420 GOTO 370
450 ? :? "PLEASE ENTER NEW INFORMATION
"
460 TEMP$(1)=" ":TEMP$(25)=" ":TEMP$(2
)=TEMP$:INPUT TEMP$:IF LEN(TEMP$)<>25
THEN TEMP$(25)=" "
470 ? "VERIFY - THIS IS CORRECT (Y/N)"
480 OPEN #2,4,0,"K:":GET #2,C:CLOSE #2
490 IF C>127 THEN C=C-128:POKE 694,0
500 IF C=89 OR C=121 THEN 530
510 IF C=78 OR C=110 THEN 450
520 GOTO 470
530 DAY$(4)=TEMP$:DAY$(30)=" ":REM STO
RE THE NEW INFORMATION
540 OPEN #2,12,0,BUF$:POINT #2,S,B:REM
 RE-OPEN THE BUFFER AND POINT TO THE S
ECTOR
550 ? #2;DAY$:CLOSE #2:GOTO 50:REM STO
RE THE INFORMATION ON DISK
```

Line 40 sets aside space for the strings. DAY$ will hold the day's activity. BUF$ is used as a buffer for the disk file name. MT$ is the month and TEMP$ holds information temporarily.

Line 50 clears the screen and places the first menu option on the screen.

Line 60 places the second menu option on the screen.

Line 70 places a question mark on the screen and waits for an entry. The trap will keep the program from crashing as a letter or other character is entered. Two spaces and two left arrows are printed before the input. This will erase the entry if the program does not accept it.

Line 80 checks the value of C. If it is not a 1 or 2, the program will return to line 70.

Line 90 branches to the correct routine.

Line 100 removes the cursor and clears the screen. It prints part of the message on the screen.

Line 110 prints the rest of the message on the screen.

Line 120 waits until a key has been pressed.

Line 130 sets the variable LN to 50. This is one of the two lines that the program can be directed to. If the return key was pressed, the value of LN changes to 150, the other line number.

Line 140 clears the key input and sends the computer to the correct line.

Line 150 formats the disk. It is very important that the disk in the drive does not have programs that you want to keep on it. When the disk is formatted, all the information on the disk will be erased.

Lines 160-180 place the files for the months on disk. The X loop goes from 1 to 12. The month is read into BUF$. The contents of DAY$ are cleared, and the file is opened with the name that is in BUF$. Line 170 begins the second loop. This loop places a buffer for each day of the month on

the disk. The first byte(s) of the DAY$ contains the number of that day. The last byte is set to a blank space. The entire buffer is printed to the disk. After all the days have been printed to the disk, the file is closed and the X loop continues.

Lines 190-200 contain the months in the format needed to open files on the disk.

Line 210 puts the cursor back on the screen and returns to the main menu. The disk is now set up as a calendar.

Line 250 begins the calendar routine. The screen is cleared, and the message to enter a month is placed on the screen.

Line 260 restores the data line and begins the for....next loop to look for a match between the month entered, and the months on file. The contents of BUF$ beginning with the third byte is compared to the contents of MT$ (the month entered). If a match is found, the computer goes on to line 280.

Line 270 continues the loop if no match is found. If the month cannot be found because of a spelling error, the program goes back to the main menu.

Line 280 waits for a day to be entered. All the files are set up for 31 days. The entry here is not checked for a correct day. If you need more days in your year, here's your chance to lengthen February!

Line 290 clears the trap. The file for the month specified by BUF$ is opened for a read. The loop gets every day from 1 to 31. The program has no way of knowing where in the file the day is located. The note command places the sector number of the day being read into the variable S. The first byte of the day is placed in B.

Line 300 checks the day in the buffer against the day entered. The loop continues until the days match. If no match is made, the file is closed and the program returns to the main menu.

Line 310 closes the file if the days match.

Line 320 clears the screen and prints the month and day entered on the screen.

Line 330 places a blank line on the screen, then prints the contents of DAY$.

Lines 340-360 print a mini-menu. If you want to keep the information that is on the screen and return to the main menu, press K. If you want to change the information, press the U. Press D if you want to check another date.

Line 370 opens the keyboard for a read. When a key is pressed, the keyboard will be closed.

Line 380 checks the value of C. If it is greater than 127, the inverse or ATARI key has been pressed. 128 must be subtracted from the value of C. A zero is poked into memory location 694 to reset the flag for normal text.

Lines 390-410 check the value of C. If it is a K, the program will return to the main menu. If it is a U, the program will continue with line 450. Entering a D will send the computer back to line 250 for another entry.

Line 420 will send the computer back for another entry because the key that was pressed was invalid.

Line 450 issues a blank line, and then asks for the new information to be entered.

Line 460 clears any previous information or garbage from TEMP$. It then waits for a new input. If the length of the input is less than 25, the last character of the string will be set to a space.

Line 470 asks you to verify what you typed.

Line 480 opens the keyboard for a read. Once a key has been pressed, the keyboard will be closed.

Lines 490-520 check the value of C. If a Y or an N was not entered, the program will loop back for another entry. If an N was pressed, the program will go back to line 450 for a new input. If the

entry is correct, the program will continue with line 530.

Line 530 places the information from TEMP$ into DAY$. The last byte of DAY$ is set to a space. The disk is fielded for 30 bytes per record. If fewer bytes are sent back to the disk, the pointers would be changed, and we could not retrieve information from the disk.

Line 540 opens the file in BUF$ for read/write or append. The point command is used to set the pointer to the correct sector and byte. The information in DAY$ must be placed back on the disk in the exact spot that it was taken from.

Line 550 prints DAY$ to the disk and closes the buffer. After each update, the routine will return to the main menu.

DISK HANDLER

Another way to access the information on the disk is by using the disk handler. The disk handler is twelve bytes long and can transfer one sector (128 bytes) of data to or from the disk. In order to transfer the information, the disk handler must be set up by the program. The disk handler begins at memory location 768. It must contain the number of the disk drive that will be accessed, the command byte (get sector, put sector, format, or status request), the buffer address, and the number of the sector to be accessed.

The next program will load in a specified track from a disk and display it as characters on the screen. The characters can be changed, and resaved onto the disk. This program will not allow you to change the data on the disk, just to look at it. It will tell you the status of the track. If the status is not OK, it is a bad track. Either the information on it got scrambled, or it was made into a bad track for copy protection purposes.

Listing 13-4. Displaying Sectors

```
10 REM LISTING 13.4
20 REM DISPLAYING SECTORS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM BUFFER$(128),DROUT$(5)
50 BUFFER$(1)=" ":BUFFER$(128)=" ":BUF
FER$(2)=BUFFER$:REM CLEAR THE BUFFER
60 FOR X=1 TO 5:READ B:DROUT$(X,X)=CHR
$(B):NEXT X
65 DATA 104,32,83,228,96
70 ADDR=ADR(BUFFER$):REM DECIMAL ADDRE
SS OF BUFFER$
80 BUFHI=INT(ADDR/256):REM HIGH ORDER
BYTE
90 BUFLO=ADDR-(BUFHI*256):REM LOW ORDE
R BYTE
100 DCB=768:REM DISPLAY CONTROL CHARAC
TERS - SET DEVICE CONTROL BLOCK
110 POKE DCB+1,1:REM DRIVE #1
120 POKE DCB+2,82:REM GET SECTOR
130 POKE DCB+4,BUFLO:REM BEGINNING OF
BUFFER - LOW ORDER ADDRESS
```

Listing 13-4. Displaying Sectors (continued from page 185)

```
140 POKE DCB+5,BUFHI:REM BEGINNING OF
BUFFER - HIGH ORDER ADDRESS
150 POKE 766,0:TRAP 150:? "INPUT SECTO
R NUMBER";:INPUT SECTOR:POKE 766,1:REM
 GET A SECTOR NUMBER
160 SECTORHI=INT(SECTOR/256):REM GET H
IGH ORDER BYTE OF SECTOR
170 SECTORLO=SECTOR-(SECTORHI*256):REM
 GET LOW ORDER BYTE OF SECTOR
180 POKE DCB+10,SECTORLO:REM STORE IT
190 POKE DCB+11,SECTORHI
200 X=USR(ADR(DROUT$)):REM CALL THE O.
S. ROUTINE TO READ DISK
210 DSTAT=PEEK(DCB+3):REM GET THE STAT
US
220 ? "DISK STATUS=";DSTAT;:IF DSTAT=1
 THEN ? " - OK":GOTO 240
230 PRINT :REM PRINT SECTOR INFORMATIO
N ON NEW LINE
240 ? "SECTOR DATA:";? BUFFER$
250 GOTO 150
260 END
```

Line 40 sets aside space for two strings. BUFFER$ will contain the data in the sector. DROUT will contain the address of the machine language subroutine in the operating system that will use the disk handler.

Line 50 clears the buffer.

Line 60 reads the machine language subroutine into DROUT$. This subroutine is a jump to a subroutine in the operating system. We need this subroutine to access the operating system subroutine because we are accessing this subroutine through BASIC. The first thing that the subroutine must do is pull a byte off the stack. If we accessed the routine directly, it would not return properly since the byte would not be pulled off the stack.

Line 70 places the decimal address of the first byte of the buffer into the variable ADDR.

Line 80 divides this number by 256. This integer is the high order address of the string location.

Line 90 subtracts this number from the address. This gives the low order address.

Line 100 sets the variable DCB to 768, the beginning of the device control block, and pokes location 766 with a 1. Now if there are any control codes in the sector, they will be printed on the screen.

Line 110 pokes the second byte of the device control block with the number of the drive that will be read.

Line 120 pokes the next byte with the command byte. 82 is the command to get a sector from the disk.

Lines 130-140 place the low order and high order byte of the address into the device control

block. This address is the beginning of BUFFER$.

Line 150 asks for a sector number. The trap is set for inputs that are not numbers. The sector number will be stored in the variable SECTOR.

Lines 160-170 divide this number for the low order and high order bytes.

Lines 180-190 place these bytes into the correct addresses of the device control block.

Line 200 calls the machine language subroutine that calls the operating system's disk interface routine.

Line 210 checks the status of the sector read.

Line 220 prints the status of that sector. If it is a 1, the sector was read incorrectly.

Line 230 forces a line feed to the screen. There is a semicolon after the status value in line 220. If the status was not OK, the line feed would not occur.

Line 240 prints the sector on the screen. Some sectors look like they are displaying garbage on the screen. Others are a string of hearts, and still others are perfectly readable.

Line 250 sends the program back to line 150 to get another sector to be displayed.

Press the break key when you are finished displaying sectors. The following list shows which bytes are used for the DCB and which commands could be used.

Address	Function
769	This byte is set to the number of the drive that will be accessed (1-4).
770	This is the command byte. The commands are: 82-get sector; 87-put sector with verify; 83-status request; 33-format disk.
771	This is the status byte. After a successful read, this byte will be a 1.
772-773	This is the buffer address that indicates where the data will be placed or taken from.
778-779	This is the number of the sector that the routine will access. The sector could be written to or read from.

FILE MANAGEMENT SYSTEM

In addition to the disk file manager, there is also a file management system. This governs the format of the disk, the location of the boot record, the file directory, the volume table of contents, and any other information needed to keep files on a disk. If any or all of the files that the file management system (FMS) uses are destroyed, the disk may not boot, copy correctly, or read in the files. By understanding how the files are structured on the disk, bad sectors can be fixed; the table of contents can be altered; and files can be deleted or restored without using the disk file manager.

Track Format

When a disk is formatted, all the data on the disk is erased. There are 720 sectors on the disk. Every byte in every sector is set to 0. The first sector of the disk is reserved for the boot record. If there is no boot record, the disk will not load from a cold start. When DOS is written to the formatted disk, the boot record is written to the disk along with the DOS and DUP files. When the disk boots, it brings DOS into the computer.

When a file or record is sent to the disk, it may require one or more tracks. The FMS begins with the first available sector and stores the program on subsequent available sector. The key word here is available. If, for example, a program used two sectors for storage; the next program

used four; and the third used five. Now, you delete the second program from the disk. Those four sectors are available for another program. The next program that you write needs 10 sectors. The FMS uses the first four sectors from the deleted program and the six sectors after the third program. How does it know where the rest of the program is???

Every sector of every program has its own *linking* bytes. There are 128 bytes in every sector. Bytes 1-125 contain the data for that sector. It could be a program listing, a file, or whatever else was saved to the disk. The 126th byte contains the file number. Every sector that pertains to this program will have the same file number. If there is a mismatch between file numbers, an error 164, file number mismatch, will occur.

Byte 127 is the forward pointer. It contains the sector number for the next sector that has more data for this file. It usually is, but does not have to be, the following sector. This is how the computer knows where the rest of the program is. If sectors 4-8 are used for a program and the next available sector is 14, the number 14 will be stored in byte 127 on sector 8. When the computer finishes reading in the data from sector 8 it will continue on to sector 14 and skip sectors 9-13. If this is the last sector for the file, this byte will be 0.

Byte 128 is the byte count. It contains the number of bytes that should be read in from this sector. A full sector contains 125 bytes. Anything less than 125 is considered a short sector.

If you use the *Displaying Sectors* program from this chapter, you can examine the sectors of the disk and see how the sectors are linked.

Volume Table of Contents

When you want to save a program onto the disk, the computer has to know where there is room to save it. It can't sit there and examine every sector on the disk to decide whether or not the program should be placed there. Sector 360 is the volume table of contents (VTOC). Use the *Displaying Sectors* program to examine this sector. Now examine a newly formatted disk.

The VTOC for the disk that has been used should have a string of hearts, with some inverse insert characters. The disk that was just formatted is almost filled with the inverse insert characters. This is a bit map that tells the computer which sectors have been used and which ones are empty. The fourth and fifth bytes of this sector indicate how many sectors are available. On a new disk these two characters should be an inverse C and a control B. This is a two byte number: 2*256+195=707 free sectors. On a used disk, this number will vary. If the 4th and 5th bytes are set to 0, the computer would think that the disk is full. The bit map begins with the 11th byte or character. On the new disk, this byte will be 15, the character will be a control 0. The next 44 bytes are inverse insert characters (ATASCII 255). The 45th and 46th bytes are a heart and insert character. The next 43 bytes are inverse insert characters (ATASCII 255). Every time something is stored on the disk, this sector is checked. If the bit is a 1, then the sector is free and the information can be placed in that sector. The bit in the bit map is then set to 0. The 45th and 46th bytes are sectors 360-368. These sectors are reserved for this VTOC and the directory. If the directory sectors are changed to ones, they could be written on. The computer would not be able to list a directory or load or store programs.

FILE DIRECTORY FORMAT

Beginning with sector 361, the computer keeps the names of the programs on the disk. There are sixteen bytes used for each entry. Up to eight names can be stored on each sector. There are eight sectors set aside for the directory. Up to 64 files or names can be stored on a disk. This

means that even if there are free sectors on the disk, the disk will not hold more than 64 programs or files.

Look at sector 361 using the *Displaying Sectors* program. You should see up to eight file names. Before each name, there are some characters. The first byte of the file name is the flag byte. If this character is a B, the file is available. If it is a b, the file is locked. If it is an inverse heart, the file has been deleted.

The next two bytes contain the sector count. The second byte is the low byte, the third, the high byte. Multiply the ATASCII value of the third byte by 256, and add the ATASCII value of the second byte to find out how many sectors are used for the file.

The fourth and fifth bytes contain the sector number at which this file starts. Again, this is a two byte number with the low byte first and the high byte second. Use the ATASCII values of these characters to find out which sector the file begins in.

The next eight bytes are the name of the file. The last three are the extender. There is no period separating the name from the extender. When the directory is read into the computer, the period is added by the program. If the file name or extender contains more characters than there is space for in the directory, the extra characters will be ignored. Table 13-1 shows the format of the file directory.

BOOTING YOUR OWN DISK

When you place a disk into the drive, and turn on the computer, the drive turns on and a program boots. This program could be the DOS or another program. Most commercially available machine language programs will boot themselves when you turn the computer on. BASIC programs are usually loaded into the computer.

AUTORUN.SYS

You may have noticed a program on your ATARI disk called AUTORUN.SYS. If you examine the disk directory from various software firms, you may find the AUTORUN.SYS is anywhere from one to three or more sectors long. You may have also found that when you write DOS to a new disk, the AUTORUN.SYS does not get written to the new disk. It has to be copied.

Table 13-1. File Directory Format.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

byte 1	flag byte
byte 2&3	sector count
byte 4&5	starting sector
byte 6-13	file name
byte 14-16	extender

The purpose of the AUTORUN.SYS is to give the programmer a way to boot a program or load a machine language program into the computer during the boot process. Let's say that you have a machine language program that will disable the break key. You want this program executed immediately when the system is turned on. If the program had to be loaded by the user and executed, chances are it wouldn't be done. A short routine could be inserted into your BASIC program to load and execute this routine, but this would take up memory space and time.

When you turn your computer on and DOS is loaded in, it checks to see if there is a program on the disk called AUTORUN.SYS. If there is, it loads this program and executes it before it turns control over to the user. The AUTORUN.SYS program must be written in machine language. It cannot be a BASIC program.

In the following program, you will create your own AUTORUN.SYS. It will change the right and left margins on the screen, and the background color before BASIC takes over.

Listing 13-5. AUTORUN.SYS

```
10 REM LISTING 13.5
20 REM AUTORUN.SYS
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 OPEN #2,8,0,"D:AUTORUN.SYS":REM OPE
N THE BUFFER TO WRITE THE NEW AUTORUN
50 PUT #2,255:PUT #2,255:REM LEADING B
YTES
60 PUT #2,0:PUT #2,6:REM STARTING ADDR
ESS
70 PUT #2,13:PUT #2,6:REM ENDING ADDRE
SS
80 FOR B=1 TO 14:READ D:REM GET THE DA
TA FOR THE ROUTINE
90 PUT #2,D:REM PUT IT IN THE BUFFER
100 NEXT B:REM CONTINUE THE LOOP
110 PUT #2,226:PUT #2,2:REM AUTO ADDRE
SS
120 PUT #2,227:PUT #2,2:REM START FROM
 THIS ADDRESS
130 PUT #2,0:PUT #2,6:REM STARTING ADD
RESS
140 CLOSE #2:REM CLOSE THE BUFFER - PR
OGRAM GOES TO DISK
150 END
200 DATA 169,5,133,82,169,30,133,83,16
9,208,141,198,2,96
```

Line 40 opens the disk buffer to write to the disk. The name of the program is AUTORUN.SYS. If there is a program by that name on the disk, it will be replaced by this one.

Line 50 puts the first two bytes of this program into the disk buffer. The first two bytes of any machine language program is 255.

Line 60 places the beginning address of the program into the buffer. The machine language program will begin at memory location 1536 (600 hex).

Line 70 places the ending address of the program into the buffer. The machine language program is 14 bytes long, so the ending address will be 1549 (60D hex).

Line 80 begins the for....next loop that reads the machine language program into the disk buffer. There are 14 bytes in this program.

Line 90 puts each byte into the buffer.

Line 100 continues the loop until all the bytes have been read and placed into the buffer.

Line 110 puts the low order address of the autorun routine into the disk buffer.

Line 120 puts the high order address of the autorun routine into the buffer.

Line 130 puts the beginning address of the machine language program into the buffer. This is the address that will be put into the preceding addresses. For a program to load and run, the running address must be stored in memory locations 738 and 739.

Line 140 closes the disk buffer. When the buffer is closed, its contents will be sent to the disk.

Line 200 contains the machine language program that will be run when the disk is booted in.

After you enter this program, run it. Be sure that the disk that is in your drive does not have an AUTORUN.SYS on it or it will be destroyed. It is best to have a new initialized disk in the drive. The DOS must be on the disk. When you run the program, the disk will turn on and you can hear the program being saved to the disk. After it is saved, turn the computer off and then turn it back on. You will hear the disk boot in. The cursor will move to the right and the screen will turn black. The word READY will appear in green. If you load in a program and list it, you will see that your margins have moved. See Table 13-2 for the machine language listing of the program.

Boot Record

The AUTORUN.SYS is one way to load and execute programs. But, what if you are writing programs in machine language, and you do not want a directory on the disk, but do want the program to load and run when the disk is turned on?

In the last chapter, we discussed how the disk was structured. The first sector is called the boot sector. This is the sector that the computer will look at to bring in a program from the disk. If you copied the DOS on the disk, you have the ATARI boot on this sector. It will bring in the DOS. If you placed your own boot on this sector, it will bring in whatever program you want it to. This is not something that can be easily accomplished from BASIC, but that's not to say that it can't be done. If you are ready to place your own boot records on the disks, you must have a good understanding of assembly language and how the boot process works. In this chapter, we will only describe the boot process.

The boot sector has its own specific format. The first byte is stored in memory location 576. It is not used by the boot and should be set to a zero. The second byte tells the computer how many sectors are in the boot. It should include this sector. This number can be any number from 1 to 255. If it is 256, it will be set to zero. The third and fourth bytes tell the computer where to start loading the program. This does not have to be where the program begins, just where to start loading the file. The fifth and sixth bytes tell the computer where the program begins. After all the sectors for the boot have been loaded, the computer has to know where it should transfer control to. This address will be the beginning of the program. The third and fourth bytes and the fifth and sixth bytes are two byte addresses. Be sure that the third and fifth bytes contain the low order address and the fourth and sixth bytes contain the high order address.

Table 13-2. Machine Language Routine for AUTORUN.SYS.

```
decimal code    assembly language listing

   169          LDA #5       ;Load the accumulator with 5.
     5
   133          STA 82       ;Store it in location 82.
    82
   169          LDA #30      ;Load the accumulator with 30.
    30
   133          STA 83       ;Store it in location 83.
    83
   169          LDA #208     ;Load the accumulator with 208.
   208
   141          STA 710      ;Store it in location 710.
   198
     2
    96          RTS          ;Return from the routine.
```

The following is an example of the structure of a boot record. It is not the complete record.

```
  0  ignore
  3  number of sectors in boot
  0  low order address
  7  high order address—place boot here
 64  low order address
 21  high order address for initialization
 76  jump to following address
 20  low order byte
  7  high order byte
  3  ??
  3  ??
  0  ??
124  ??
 26  ??
  1  data—loads into the Y register
 24
```

```
  0
125
203  data
  7
172  load the Y index use-next address
 14  low order byte for address
  7  high order byte for address
240  branch if it's 0
 54  ahead 54 bytes
```

This portion of a disk boot shows how the information in the first sector is used. The first byte (0) is ignored. The second byte indicates how many sectors are in this boot. This number is stored at location 577. The next two bytes are stored at locations 578 and 579. They tell the computer where this boot should begin. The first six bytes and all the byte following will be moved to the location in memory beginning with the address specified here. The next two bytes are the initialization bytes. After the boot is executed, the address in this memory location will be jumped through. The next instruction in the boot is a jump. Once the three sectors have been moved into memory the computer will perform a jump subroutine. The address that it will jump to is the sixth address after the beginning address. In this case it is location 1798 (706 hex). The instruction at this address is to jump to memory location 1824 (720 hex). At this location, the computer will load the Y index with the value stored at 1812 (714 hex). If the value is zero, the computer will move ahead 54 bytes. If we count 15 bytes down in the table, we will see that the value at that location is a one. The computer will proceed to the next instruction.

The boot process then consists of:

1. Reading the first sector and storing the first six bytes, which indicate the number of sectors, the load address, and the initialization address.
2. Placing the boot sectors into memory beginning with the location specified in the third and fourth bytes of the boot sector.
3. Jumping to the subroutine that begins in the seventh location after the beginning of the boot.
4. Jumping through the subroutine whose address is stored in locations 12 and 13 (fifth and sixth bytes of the boot record).
5. Transferring control to the program by jumping to the address in locations 14 and 15.

The machine language program loaded into the computer will now be fully operational. If the addresses are set correctly when system reset key is pressed, the program will restart itself. If they are not, the drive may turn on and the computer may try to boot the program again, or the computer will go to BASIC or the memo pad.

Chapter 14

Cassette Use

Like the disk, the cassette is a convenient form of storage that can be utilized more effectively when the way the computer handles it, and its physical structure are understood.

THE CASSETTE HANDLER

The cassette handler is similar to the disk manager. Files can be written to or read from the cassette. The buffer can be opened for get and put commands. Programs can be loaded using BASIC or using an autoload command. However, there are some important differences between the disk and cassette. The cassette can only read and/or write data to the portion of tape under the recorder head. Files cannot be stored or retrieved from just any position. When a program is saved to the cassette, it is saved in one continuous stream. There is no positioning of the head. You cannot specify tracks like you can point to sectors and bytes on the disk. Likewise, you can only read what is passing under the cassette head. You cannot point to only one section of tape like you can read in one sector from the disk. The cassette is, however, a fairly reliable way to store and retrieve information.

Format of Data

If you have a recorder, save a program to it. It doesn't have to be very long—four or five lines will do. After the program is saved, rewind the tape and enter these commands in the direct mode:

```
OPEN #7,4,0,"C:":FOR X=1 TO 128:GET #7,B:? B;" ";:NEXT X:CLOSE #7
```

This opens the cassette for a read. Now press the play button on the recorder and the return key on the keyboard. You will hear one tone. Press the return key again. After you hear a record of data being read in, you will see numbers being printed on the screen. This is the code for the program that was saved. The computer is printing it from the cassette buffer, in the same manner as it was printed from the disk buffer.

Now enter this without clearing the screen:

```
FOR X=0 TO 130:?PEEK(1021+X);" ";:NEXT X
```

The first three bytes that are printed on the screen are different from the first three bytes that were printed from the buffer. But, if you compare the first byte of the buffer with the fourth byte, you will see that both number patterns are identical. Memory location 1021 begins the cassette buffer. The buffer is 131 bytes long.

The first two bytes in the buffer are 85. These two bytes are fixed and are used by the computer to measure the speed of the cassette. The third byte is the control byte. If this byte is 252, the record is full—128 bytes. A 250 in this position indicates a partially full record and a 254 is an end of file record.

The first bytes after the speed and control bytes for the first record consist of the table entries. This includes the variable table, the value table, etc. The values are adjusted by the amount of memory in your machine. This is why a program that requires more storage memory than your machine has will produce an error message immediately when you try loading it rather than half way through the load.

The last byte that the computer reads from the cassette is a checksum. When the program was saved to the cassette, every byte that was sent to the cassette in the 128 byte record was added to the previous bytes. The two markers are also included in the addition. After the two markers, the control byte, and the record were sent to the cassette, the sum of the bytes were also sent. When the records are read in, the bytes are added together again. If the sum of the bytes match the checksum byte that is read in, then the record is assumed to be correct and the computer will continue with the loading process. If the two bytes do not match, the load will stop and an error message will appear on the screen.

Cassette Frames

Each record of 128 bytes is considered a frame. The speed at which the data is sent or received is called the baud rate. The ATARI uses the baud rate of 600. This means that 600 bits are sent per second. (Each byte is 8 bits). The two marker bytes are read in by the computer. The computer determines how long it took to read them in and calculates the correct baud rate for the tape. It does this with every frame of data that is read in. The input baud rates can be adjusted to read faster or slower. This adjustment process allows for variations in motor speeds, stretched tape, etc. It does not, however, allow for alignment problems between the recorder that the program was originally saved on and the one that it is being read on.

You may have noticed that when you CSAVE a program to cassette, the records seem to be sent faster than when you use the list command. There are two different modes that the computer can use when sending records to the cassette. One is called the normal IRG (Inter-Record Gap), and the other is the short IRG. The computer uses the short IRG for the CSAVE and CLOAD commands. The computer reads in the record, checks the checksum, places it in RAM, and goes back for another record. The recorder is running the entire time. The computer must do its work quickly so that it won't miss the next record.

When the computer uses the normal IRG for saving records to the tape, the recorder stops after every record is read into the computer. The information can be processed; then next record can be read in. The baud rate for saving and reading the data is the same for both modes. It is the time between records that varies. In the normal mode there is about 3 seconds of tone between records. In the short mode there is about ¼ of a second of tone time between records.

The IRG is set up by the computer when the save or load command is entered. If the wrong command is entered for the tape, the program will not load and an error message will be displayed on the screen.

BOOTING YOUR OWN CASSETTES

There are two different ways to make your cassettes load and run automatically. The first way is with a BASIC program. A short boot program is saved to the tape. Then, without moving the tape, the second program is saved to the cassette. By using the RUN "C:" command, the boot program will load, run, and load in the second program. The following program is a short boot program that will load and run a second program.

Listing 14-1. BASIC Boot Load

```
10 REM LISTING 14.1
20 REM BASIC BOOT LOAD
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 GRAPHICS 0:POKE 710,114:POKE 712,11
4
50 POKE 65,0:POKE 752,1
60 POSITION 23,2:? "copyright 1983"
70 POSITION 13,10:? "Now loading":POSI
TION 15,12:? "programs":POSITION 14,15
:? "by"
80 POSITION 15,16:? "your name"
90 COLOR 35:PLOT 10,7:DRAWTO 27,7:DRAW
TO 27,18:DRAWTO 10,18:DRAWTO 10,7:COLO
R 32:PLOT 2,20
100 POKE 752,0:POKE 764,12
110 RUN "C:"
```

Line 40 sets the graphics mode to 0. The graphics mode must be set before plots and drawtos. The background color and border colors are changed.

Line 50 pokes 65 with a 0. This will turn off the noise channel and you won't be able to hear the program loading. This line also removes the cursor from the screen.

Line 60 prints the copyright notice on the screen.

Lines 70-80 prints your message on the screen.

Line 90 sets the color to 35. This is the ATASCII value of the pound sign. Using the plot and drawto commands, a border is drawn around your message.

Line 100 turns the cursor back on and places a carriage return in the location that holds the last key entered.

Line 110 executes the RUN "C:" command. You won't have to press the return key because the code is already poked into location 764.

The program that is saved on the tape after this boot program had to be saved using the SAVE "C:" command. If it was CSAVEd, this procedure won't work. If you want the next program to load in only, and not run, change line 110 to CLOAD, and be sure that the program that you want to load was CSAVEd to the cassette.

The noise location was set to zero in this program so that music or instructions could be heard over the television speaker. This command is not needed if you will not be using the second track on the cassette.

Boot Record

If you wrote a machine language program that you want to load in from cassette without using the editor/assembler, you need to write a boot record as the first record of the cassette program. The boot record for the cassette is almost identical to that for the disk.

There are six bytes for the first record of the boot. The first byte is ignored. The second byte contains the number of records that should be initially read. The third and fourth bytes contain the address where the boot records should be stored. The fifth and sixth bytes contain the address that should take control of the program after the boot is completed.

The following is the partial coding of a machine language program's boot record.

```
  0   ignore
 32   number of records
191   low order memory load address
 11   high order memory load address
184   low order initialize address
 27   high order initialize address
169   load the accumulator with next byte
 60
141   store it in the next address
  2   low order address
211   high order address
169   load the accumulator with next byte
  3
141   store it in the next address
 15   low order address
210   high order address
```

The computer will ignore the leading zero. The next number, 32, is the number of records that will be read in. This value will be saved. The computer will begin storing these records at location 3007. The fifth and sixth bytes will be stored at memory locations 2 and 3. After all the records are read in, the computer will issue a jump to the subroutine at the seventh byte loaded in. This byte will load the accumulator with 60 and store this value at location 54018. This shuts off the cassette recorder. The program continues. When it has returned, the computer will jump through the initialization routine. The address for this routine was stored in locations 2 and 3. Once the initialization is complete, the computer will jump through the address at location 14-15 and begin the program.

Although it is possible to write a BASIC program that will load and automatically run a machine language program, it really should be done using the editor/assembler. The procedure to load a cassette with a machine language boot on it is:

Turn off the computer.
Hold down the start key and turn the computer on.
When you hear the single tone, press the play button on the recorder and press the return key on the computer.

The records on the tape must be created using the short mode for it to load properly.

USING THE CASSETTE WITH SOUND

In the last program, we used a BASIC boot program to bring in the program. Using the RUN "C:" command, the computer loaded in the first program and ran it. This program contained the instruction to either CLOAD or RUN "C:" the next program on the tape. It also poked the memory location that directed the sound of the data so that the program loaded silently.

The ATARI recorder is a stereo recorder. Most other computers use a monaurial recorder. With a monaurial recorder, the computer places two tracks of data on the tape, one track on one half and one on the other. With a stereo recorder, the computer only uses half of the track for data. A stereo recorder places four tracks on each tape, two on one half of the tape and two on the other. See Fig. 14-1 for the data tracks.

The other two tracks on the tape are left blank. They can have sound or spoken instructions on them if you have the facilities to record and duplicate stereo tapes. The procedure is fairly simple, but it does require some planning and timing.

First, load the BASIC boot program into the computer. Save it to a new cassette using the SAVE "C:" command. After the BASIC boot program has been saved to the cassette, load in the program that you want to save. If you have a disk drive, you can load the program in from the disk. If you have only a cassette recorder, carefully remove the new cassette from the recorder and place the cassette with the program on it in the recorder and load it into the computer. You want to handle the cassette with the new program on it carefully because the tape is in position for the second program. If you rewind the tape, you will be saving the program over the BASIC boot program. If the tape advances too far, there could be a loading problem.

Once the second program is loaded into the computer, it can be saved to the new tape with either the SAVE "C:" command or the CSAVE command. If your boot program is using the RUN "C:" command, then use the SAVE "C:" command and the second program will run immediately after it is loaded in. If your BASIC boot program uses the CLOAD command, CSAVE your program to the cassette.

After the second program has been saved, rewind the cassette, type RUN "C:", press the play button on the recorder and the return key on the computer twice. If you have not poked

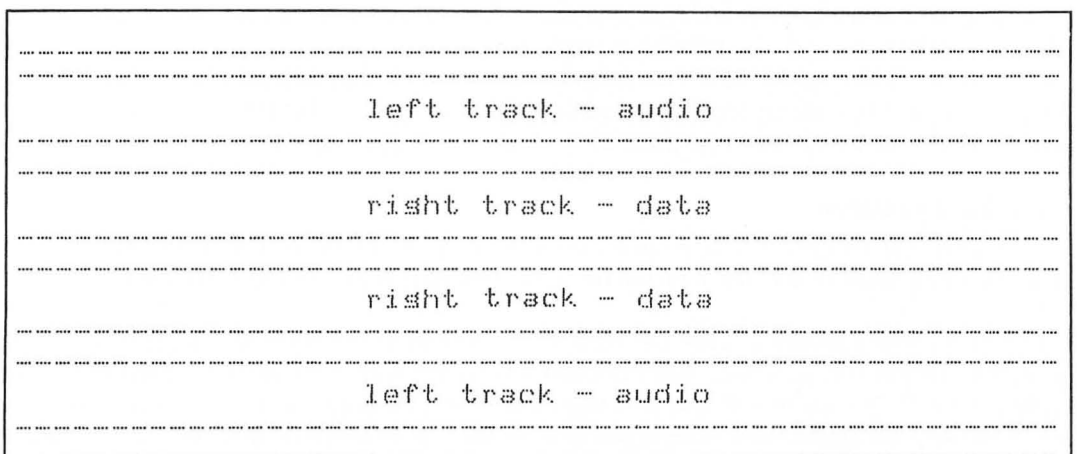


Fig. 14-1. Track formats on cassette.

location 65 with a 0, you will hear the BASIC boot program load in. The screen will change colors and display your message on the screen. The tone will sound and the computer will begin to load in the second program. You don't even have to press the return key. The BASIC boot program did it all. Now listen to the program load. If you can hear it, you either didn't have a POKE 65,0 in the boot program, have a problem with separation in the cassette recorder, or have very good ears. The only sounds that should come through the television speaker are the sounds recorded on the left track. Since we did not place any sounds there, you shouldn't hear any.

Adding Sounds or Instructions

In order to record sounds, music, or instructions on the left track, you need a stereo cassette recorder/duplicator. Place the cassette into the recorder and listen to it. First you will hear a pure tone. Then you will hear the date. Each record is separated by the pure tone. The BASIC boot program will be about four or five records long. The last record of this program will have a slightly different sound. After listening to the records several times, you will be able to tell the difference between the program records and the ending record. After the last record for the first program, you will hear the pure tone again for a longer period of time before you hear the first record. This is the tone that the computer sends out before it starts to send the program. It seems longer here because there is no leader tape between the first and second program. This is where you should start the music, sound, or instructions on the left track.

Place a new tape into deck B of the recorder/duplicator. Place the tape that you have just saved the two programs on into deck A. Press the play and record buttons for cassette B and the play button for cassette A. If you are using a microphone, be sure that it is connected to the left track. If you will be recording music directly, be sure that your patch cord is plugged in for the left channel. Listen to the cassette as it is being duplicated from deck A to deck B. When you hear the tone for the beginning of the second program, you can begin speaking, playing the music, etc. Be sure to stop recording on the left track just before the last record. Otherwise, the recorder will stop loading in the program and you will not hear the remaining words or music.

Rewind both cassettes, remove the newly created cassette from deck B and place it in your program recorder. Type RUN "C:" and press the return key. This time when the second program loads into the machine, you should hear your music, instructions, or whatever you recorded on the left track coming through the television speaker.

Because of the speed variances between recording machines, head alignments, and the nature of cassettes, you may have to repeat the duplication process before you create a tape that will load. This process cannot be used with a high speed tape duplicator.

Voice Synchronization

Another use for the cassette recorder is for voice synchronization. Poking memory location 54018 with 52, turns on the motor on the recorder. Poking that location with a 60 turns the motor off.

The following program turns the recorder on and off at the correct time. It is a spelling program. To use this program, you will need a recorder with a microphone. First enter this program into the computer and save it on tape. Either turn the tape over and rewind it so that you are at the beginning of the back side, or just remove the tape from the program recorder and place it in the other recorder. Now record the words in the data line onto the tape. Space the words at five second intervals. Now rewind the cassette to the beginning of the words and run the program.

Listing 14-2. Listen and Spell

```
10 REM LISTING 14.2
20 REM LISTEN AND SPELL
30 REM BY L.M.SCHREIBER FOR TAB BOOKS
40 DIM WORD$(15),ANS$(15)
50 N=N+1:? "}CLEAR}":IF N>10 THEN 170:
REM CLEAR THE SCREEN
60 POSITION 3,5:? "LISTEN TO THE WORD"
70 POKE 54018,52:REM TURN ON THE RECOR
DER
80 FOR T=1 TO 2000:NEXT T:REM LET IT R
UN
90 POKE 54018,60:REM TURN IT OFF
100 W=0:READ WORD$
110 ? :? "WHAT WAS THE WORD";:REM ASK F
OR WORD
120 INPUT ANS$
130 IF WORD$=ANS$ THEN POSITION 3,20:?
 "VERY GOOD":FOR T=1 TO 500:NEXT T:GOT
O 50
140 W=W+1:IF W<>3 THEN 110
150 POSITION 3,20:? "THE WORD WAS -- "
;WORD$
160 FOR T=1 TO 500:NEXT T:GOTO 50
170 POSITION 3,10:? "YOU HAVE FINISHED
 YOUR SPELLING FOR   TODAY!!!"
180 END
200 DATA BOOKS,FAVORITE,FOOTBALL,HOBBI
ES,CARNIVAL,MOUNTAIN,BIONIC,FAILURE,AT
TITUDE,SECURE
```

Line 40 sets aside the string space for the word (WORD$) and the answer (ANS$).

Line 50 adds one to the value in N. This counts the words as they are being given. If the value is greater than 10, then all the words have been given and the program will go to the end at line 170 after the screen is cleared.

Line 60 instructs the user to listen to the word.

Line 70 turns on the cassette recorder by poking location 54018 with a 52. The play button on the recorder must be pressed down for this program to work. If that button is not down, you will hear a click from the television speaker, but no word.

Line 80 is a timing loop. This is set for about five seconds. You may have to adjust this number for your own program recorder.

Line 90 turns the recorder back off.

Line 100 sets the variable W to zero. This variable will count the number of times the user tries to spell the word. It also reads the word from the data line. This is the word that was just spoken on the tape.

Line 110 asks the user for the word.

Line 120 waits for an entry to be made.

Line 130 checks the answer entered against the word. If it is correct, a message will be printed on the screen; the program will pause and then go on to line 50 for another word.

Line 140 adds one to the variable W and checks to see is this was the third try. If it wasn't, the program goes to line 110 and asks for the word to be entered again.

Line 150 prints the correct word if the user tried to spell it three times and couldn't.

Line 160 is a timing loop to give the user time to study the word. Then the program continues at line 50.

Line 170 tells the user that all the words have been spelled.

Line 200 contains the words used in this program.

This program can be a very effective way to learn, but it does have its drawbacks. In this case, the words must be asked in the same order every time. After a while, the user will know the order of the words. Since the tape cannot be rewound or fastforwarded from the computer, the information will always be presented in the same order. If the word was entered wrong, or if you want to hear it again, the tape cannot be rewound to that word.

On the other hand, if you are presenting a very structured lesson or a story with animation, there would be no need to rewind or repeat parts of the tape.

Index

Index

Advanced Programming Techniques for Your ATARI®, including Graphics and Voice Programs

by Linda M. Schreiber

- Get the most from your ATARI's sound and graphics capabilities for both practical *and* entertainment purposes!
- Master the special techniques that let you write your own advanced programs for almost any application you can think of!
- Go beyond the limitations imposed by BASIC . . . to become the master of your machine rather than merely its user!

Here's a book that shows you how to understand the special characteristics of your ATARI *and* how to use them for all sorts of new and exciting sound and graphics effects. You can create your own character sets . . . mix graphics modes . . . use player/missile graphics and screen flipping . . . create animated games . . . understand and use interrupts . . . create your own self-booting disk and cassette programs . . . even enter a machine language subroutine to play music while a BASIC program is running!

If you have a good understanding of BASIC but want to go beyond the limitations it imposes, this book is *the place* to begin. You'll learn to manipulate your machine using professional tips and tricks perfected by an author thoroughly knowledgeable in both program design *and* the ATARI's capabilities. Every program is described in detail so that you'll be able to use the illustrated techniques to begin writing your own original programs. You'll even find a listing of all ATARI memory locations used by the operating system, discover how to change their values for different programming effects, and learn how to use the disk file structure to give you control over the drive. EVERYTHING you need to become an advanced programmer, able to use *all* of your ATARI's unique capabilities, is included in this outstanding sourcebook!

Linda M. Schreiber is a professional programmer and expert on microcomputers. She is the author of TAB's *ATARI Programming . . . with 55 Programs.*

FPT > $14.50

ISBN 0-8306-1545-8

PRICES HIGHER IN CANADA

1395-0783