STRING$

String variables enable a programmer to store and manipulate words in a BASIC program. In this module campers will experiment with the various string handling features of the Atari. Upon completing this module, campers will be able to write a program that converts English to Pig Latin.

String variables were originally introduced in the module on variables. If you feel you need to brush up on strings, you may want to read the following materials.

Inside Atari BASIC: pp. 40 - 44
Your Atari Computer: pp. 64 - 65
Atari 400/800 BASIC Reference Manual: pp. 37 - 40

If you feel that any of your campers are unsure of how to dimension strings, you can use the accompanying worksheet entitled, "String Variables Worksheet," to review initializing string variables.

When you are discussing any type of variable, we recommend that you encourage your campers to use descriptive variable names. For example, NAME$ is much more meaningful to the reader than N$. Also, encourage your campers to reserve ample space for their strings in order to account for the largest conceivable input that may need to be stored in the string. String variables can be as large as you wish, as long as they will fit in the computer's memory.

Indexing Strings

Indexing strings enables you to extract or change a portion of a string. Isolating a portion of a string becomes particularly useful when strings are used to hold data.

Explain the format and procedure for accessing a portion of a string using an index. Campers should understand the following concepts.

1. One index in parentheses sets up a substring that includes the indexed element of the string to the last elemert of the original string. An example of using a single index is listed below. The program prints "LP" on the screen.

```
DIM WORD$(10)
WORD$ = "HELP"
PRINT WORD$(3)
LP
```

2. Two indices in parentheses can be used to isolate any one element, or a group of elements, in a string. A sample program using two indices is listed below.

```
DIM WORD$(20)
WORD$ = "ATARI CAMP"
PRINT WORD$(2,9)
TARI CAM
PRINT WORD$(7,10)
CAMP
PRINT WORD$(2,4)
TAR
```

Activity #1

1. Have the campers type in the following program. Campers should insert their own name in quotes on line 40. Discuss what happens to the substring that is printed on the screen each time the index in parentheses is incremented.

```
10 REM *  INDEX TO A STRING
20 REM
30 DIM NAME$(30)
40 NAME$ = "GEORGE WASHINGTON"
50 PRINT NAME$(1)
60 PRINT NAME$(2)
70 PRINT NAME$(3)
```

2. Have the campers change the index, so that only the last letter in their name is printed on the screen.

*** Leave the NAME$ program in memory for the next activity.

Activity #2

This program exemplifies how to isolate a substring by using two indices.

1. Edit the NAME$ program currently in memory by typing in the three additional lines listed below. Ask the campers to predict what will be printed on the screen.

```
50 PRINT NAME$(1,1)
60 PRINT NAME$(1,3)
70 PRINT NAME$(5,5)
```

2. Have the campers edit the program to print their initials.

Activity #3

1. A variable can be used in the place of an index to a string. For example, the variable COUNT represents an index of three in the program below.

```
COUNT = 3
NAME$ = "ROBERT"
PRINT NAME$(COUNT,6)
BERT
```

Using a variable enables you to change the index without repeating the string statement. This is particularly useful when you want to extract one element of a string at a time. In the program below, campers will print their name in big letters on the screen. By using a variable index, they will be able to print one letter of their name at a time. A delay loop holds the current letters on the screen before the next letter is printed. Without the delay loop it would appear as though all the letters were printed on the screen at the same

time. Note that the counter (LETTER) to the FOR . . . Next loop is also used to increment the position of where the next letter will be printed. Have the campers type in the program listed below. Be sure that they insert their own name on line 140 and the number of letters in their name on line 150.

```
100 REM *  VARIABLE INDEX
110 REM *
120 GRAPHICS 2 + 16
130 DIM NAME$(20)
140 NAME$ = "JOHN JONES"
150 LENOFNAME = 10
160 FOR CHAR = 1 TO LENOFNAME
170 POSITION CHAR+3,5
180 PRINT #6;NAME$(CHAR,CHAR)
190 FOR DELAY = 1 TO 500: NEXT DELAY
200 NEXT CHAR
```

2. Have the campers experiment with printing their full name on the screen, printing their name on a diagonal, etc. Challenge the campers to print the letters of their name in the reverse order. This can be done by using the following FOR . . NEXT loop on line 160.

```
FOR CHAR = LENOFNAME TO 1 STEP -1
```

Activity #4

1. It is also possible to alter a portion of a string by using indices. In the example below, new letters are being assigned to the 10th through the 13th element of the SOUP AND SALAD string. Ask the campers to predict what the program listed below will print on the screen, before to typing it into the computer. Have them RUN the program to test their hypotheses.

```
10 REM *  A SUBSTRING SWITCH
20 REM *
30 DIM LUNCH$(20)
40 LUNCH$ = "SOUP AND SALAD"
50 PRINT LUNCH$
60 LUNCH$(10,13) = "BREA"
70 PRINT LUNCH$
```

INDEXING STRINGS
(Continued)

2. Ask the campers to edit the program to print SOUP A LA CARTE. They will need to change the indices, as well as the substring in line 60.

Concatenating Strings

Explain what concatenation means. When strings are concatenated, they are linked together. Explain how two strings can be concatenated together. Discuss how it is possible to change a portion of a string or concatenate one string to another string by using indices.

Activity #1

1. The following program is an example of how two strings can be concatenated to output a longer string. Call the campers' attention to the fact that both strings are being dimensioned by one DIM statement on line 30. Have the campers type in the program listed below. RUN the program to see what is printed on the screen.

```
10 REM *  STRING CONCATENATION
20 REM *
30 DIM PREFIX$(10),ROOT$(10)
40 PREFIX$ = "TELE"
50 ROOT$ = "VISION"
60 PRINT PREFIX$;ROOT$
```

2. Now have the campers edit the program to print TELECOMMUNICATIONS, TELENETWORK, or TELEGRAM.

Activity #2

1. To see how two strings can be concatenated using indices, have the campers type in the program listed below.

```
10 REM * CONCATENATE A SUBSTRING
20 REM *
30 DIM ORIGINAL$(10),EXTENSION$(10)
40 ORIGINAL$ = "TELEPHONE"
50 EXTENSION$ = "GRAPH"
60 ORIGINAL$(5,9) = EXTENSION$
70 PRINT ORIGINAL$
```

LEN

Introduce the "LEN" string function. The length function counts the number of characters in a string, including any spaces and symbols.

Activity #1

1. Review the fact that a string can be assigned its contents from within a program or from a program user's input. Have the students type in the program listed below in order to experiment with the LEN function.

```
10 REM *  THIS PROGRAM DEMONSTRATES
20 REM *  THE LEN FUNCTION
30 REM *
40 DIM WORD$(20)
50 PRINT "TYPE IN A WORD";
60 INPUT WORD$
70 PRINT "THERE ARE ";LEN(WORD$);" LETTERS IN YOUR WORD."
```

2. The campers should be able to explain why there are semi-colons used in line 70, and why there is a space enclosed in quotes following the word "are" and before the word "letter". RUN the program.

3. Have the students RUN the program a second time. This time, type 5 spaces and then type the word. Use the same word both times. Ask you campers if the program reported the same number of letters for the word both times.

*** Leave the program in memory so that it can be edited in the next activity.

Comparing Strings

Explain how to compare strings in BASIC. Emphasize the fact that the strings being compared must be identical. Upper and lower case letters do not match. YES followed by a space does not match a YES response with no spaces before or after it.

Activity #1

1. The students should edit the LEN function program by typing in the following lines.

```
20 REM *  STRING COMPARISON
40 DIM WORD$(20),ANSWER$(3)
80 PRINT "WOULD YOU LIKE TO KNOW HOW LONG"
90 PRINT "ANOTHER WORD IS";
100 INPUT ANSWER$
110 IF ANSWER$="YES" THEN GOTO 50
```

2. Have the students RUN the program and experiment with the following responses.

NO, Yes, YES, and MAYBE

3. One way to avoid errors or confusion on the part of the user giving a response is to dimension ANSWER$ with one character and ask for a "Y" or "N" response. Then the comparison will be made with the first letter of the response regardless of what the user types in. YES, YEP, and Yes will all be treated as "yes." Note how ANSWER$ is dimensioned to one character in the following program.

Activity #2

1. Now use indices, substrings, concatenation, and LEN to write words in Pig Latin. Pig Latin takes the first letter of a word, moves it to the end of the word and adds an "ay" on the end. The students should fill in the missing information in the program listed below and type it into the computer.

```
10 REM *  PIG LATIN
20 REM *
30 DIM WORD$(20),ANSWER$(1),CHAR$(1)
40 PRINT "TYPE IN A WORD";
50 INPUT ________
55 PRINT
60 LENGTH = _____(WORD$)
70 CHAR$ = WORD$(___,___):REM CHAR$ IS ASSIGNED THE FIRST ELEMENT
OF WORD$
80 PRINT "THERE ARE ";_______________;" LETTERS IN YOUR WORD."
85 PRINT
90 PRINT WORD$(2);________;"AY IS YOUR WORD IN PIG LATIN."
95 PRINT
100 PRINT "WOULD YOU LIKE TO KNOW ANOTHER WORD"
110 PRINT "IN PIG LATIN? (TYPE Y OR N.)"
115 PRINT
120 INPUT __________
130 IF ANSWER$ = "Y" THEN GOTO 40
```

PROGRAMMING CHALLENGES USING STRINGS

1. Write an entire message in Pig Latin. Have a user type in a message in English and you print the message in Pig Latin. Be sure to reserve enough space in a string to input a very long message, in case the user is extremely verbose. Starting with the first letter of the message, compare each element of the string to a space in order to find the break between each word. Each time you encounter a space, convert the previous word to Pig Latin, print in on the screen, and read on. The following BASIC code may prove to be useful to you.

```
FOR LETTER = 1 TO LENGTH
CHAR$ = MESSAGE$(LETTER,LETTER)
IF CHAR$ <> " " THEN WORD$(INDEX,INDEX) = CHAR$
INDEX = INDEX + 1
NEXT LETTER
PRINT WORD$(2);   .   .   .
```

2. Devise your own secret code. Write a program that converts a message typed into the computer into your secret code. One technique for writing a secret code is to switch letters for other letters in the alphabet. For example, all the "A's" could be converted to "Z's." Be creative.

3. Write a program that jumbles up the letters in a word. The jumbled word is printed on the screen. The user is asked to guess what the word is and type the letters of the word in their correct order. Give the user additional guesses for incorrect responses.

***** String Indexing Programs *****

```
10 REM *  INDEX TO A STRING
20 REM
30 DIM NAME$(30)
40 NAME$ = "GEORGE WASHINGTON"
50 PRINT NAME$(1)
60 PRINT NAME$(2)
70 PRINT NAME$(3)
```

```
50 PRINT NAME$(1,1)
60 PRINT NAME$(1,3)
70 PRINT NAME$(5,5)
```

```
100 REM *  VARIABLE INDEX
110 REM *
120 GRAPHICS 2 + 16
130 DIM NAME$(20)
140 NAME$ = "JOHN JONES"
150 LENOFNAME = 10
160 FOR CHAR = 1 TO LENOFNAME
170 POSITION CHAR+3,5
180 PRINT #6;NAME$(CHAR,CHAR)
190 FOR DELAY = 1 TO 500: NEXT DELAY
200 NEXT CHAR
```

```
10 REM *  A SUBSTRING SWITCH
20 REM *
30 DIM LUNCH$(20)
40 LUNCH$ = "SOUP AND SALAD"
50 PRINT LUNCH$
60 LUNCH$(10,13) = "BREA"
70 PRINT LUNCH$
```

***** Concatenation Programs *****

```
10 REM *  STRING CONCATENATION
20 REM *
30 DIM PREFIX$(10),ROOT$(10)
40 PREFIX$ = "TELE"
50 ROOT$ = "VISION"
60 PRINT PREFIX$;ROOT$
```

```
10 REM * CONCATENATE A SUBSTRING
20 REM *
30 DIM ORIGINAL$(10),EXTENSION$(10)
40 ORIGINAL$ = "TELEPHONE"
50 EXTENSION$ = "GRAPH"
60 ORIGINAL$(5,9) = EXTENSION$
70 PRINT ORIGINAL$
```

***** LEN Function Programs *****

```
10 REM *  THIS PROGRAM DEMONSTRATES
20 REM *  THE LEN FUNCTION
30 REM *
40 DIM WORD$(20)
50 PRINT "TYPE IN A WORD";
60 INPUT WORD$
70 PRINT "THERE ARE ";LEN(WORD$);" LETTERS IN YOUR WORD."
```

***** Comparing Strings *****

```
20 REM *  STRING COMPARISON
80 PRINT "WOULD YOU LIKE TO KNOW HOW LONG"
90 PRINT "ANOTHER WORD IS";
100 INPUT ANSWER$
110 IF ANSWER$="YES" THEN GOTO 50
```

***** Pig Latin Program *****

```
10 REM *  PIG LATIN

20 REM *

30 DIM WORD$(20),ANSWER$(1),CHAR$(1)

40 PRINT "TYPE IN A WORD";

50 INPUT ________

55 PRINT

60 LENGTH = _____(WORD$)

70 CHAR$ = WORD$(___,___):REM CHAR$ IS ASSIGNED THE FIRST ELEMENT
OF WORD$

80 PRINT "THERE ARE ";_____________;" LETTERS IN YOUR WORD."

85 PRINT

90 PRINT WORD$(2);________;"AY IS YOUR WORD IN PIG LATIN."

95 PRINT

100 PRINT "WOULD YOU LIKE TO KNOW ANOTHER WORD"

110 PRINT "IN PIG LATIN? (TYPE Y OR N.)"

115 PRINT

120 INPUT __________

130 IF ANSWER$ = "Y" THEN GOTO 40
```

STRING VARIABLE REVIEW WORKSHEET

String variables must first be dimensioned with a DIM instruction. The dimension statement determines the maximum number of characters which can be held in the string.

1. Type in the following program and RUN it to see what happens.

```
10 REM *  A PROGRAM TO DEMONSTRATE
20 REM *  THE DIM FUNCTION.
30 REM *
40 DIM FRUIT$(5)
50 FRUIT$ = "APPLE"
60 PRINT FRUIT$
```

2. Note that the word being assigned to the string must be in quotes. If you change APPLE to BANANA, what will be printed on the screen? _____Try it and see.

3. A string variable can contain any combination of letters, numbers, spaces, and symbols. Also, the contents of a string can be assigned from within a program or from input. Type in the following program.

```
10 REM *  THIS  PROGRAM DEMONSTRATES
20 REM *  THE VARIOUS POSSIBLE ELEMENTS
30 REM *  OF A STRING FROM INPUT.
40 REM *
50 DIM TIME$(40)
60 PRINT "WHAT TIME IT IT";
70 INPUT TIME$
80 PRINT "IT'S ";TIME$;"!  I'M LATE!  BYE."
```

4. When you are asked what time it is, type in 10:30 am. and RUN the program.

5. What would be printed out if your typed in ten-thirty?

_____Try it and see.

READ, DATA, RESTORE

This module introduces three BASIC statements. READ, DATA, and RESTORE enable a programmer to assign values to variables more efficiently than the individual variable assignments presented thus far. Reading and storing data are explained, and campers can experiment with various types of data. After completing this module, campers should be familiar with how to write versatile programs that systematically read data and check for the last data entry.

Before participating in the activities in this module, campers must understand the difference between a numeric and a string variable. The participants also must have had experience programming SOUND on the Atari.

If you are unfamiliar with the READ, DATA, or RESTORE statements, they are explained in the following books.

Inside Atari BASIC:	pp. 60-63
Your Atari Computer:	pp. 76-78

Read and Data

1. In order to familiarize campers with the READ and DATA statements, start by writing a simple program on the board like this one.

```
10 REM *  READ DATA DEMO
20 REM *
30 DIM DAY$(20)
40 NUMOFDAYS = 7
50 PRINT
60 FOR DAYOFWEEK = 1 TO NUMOFDAYS
70 READ DAY$
80 PRINT DAY$
90 NEXT DAYOFWEEK
100 DATA SUNDAY, MONDAY, TUESDAY
110 DATA WEDNESDAY, THURSDAY, FRIDAY
120 DATA SATURDAY
```

2. Explain the format and function of the READ and the DATA statements. A READ statement enables a programmer to retrieve values from a list of data.

READ AND DATA
(Continued)

The DATA statement allows the programmer to store large quantities of data efficiently. Note that each element of data is separated by a comma. Even though we are assigning a word to the string variable DAY$, no quotation marks are needed in the list of data.

3. Explain that the READ statement looks for the first "unread" element of data. The concept of a "pointer" should be explained here. The computer maintains a pointer that points to the next data item to be READ. Each time an element of data is read, the pointer is advanced to the next element of data.

4. READ statements are customarily used in a loop. In this example, we are using a FOR . . NEXT loop. When a FOR . . NEXT loop is used, the number of times the loop is to be executed must correspond to the quantity of data. If a READ statement is executed and all of the data has been read, you will get an error message.

5. Step through the program and execute each instruction just as the computer would. Write each DAY$ on the board as you execute the PRINT DAY$ instruction. Each time an element of data is read, the "pointer" is advanced to the next data. Use your pencil to point to the data to be read as you step through the program. Call the campers' attention to the fact that data can appear on more than one line. The computer's pointer automatically advances to data on the next line. DATA lists are always read in the order of their line number, from the lowest to the highest.

Activity #1

1. Have the campers type in the days of the week program listed below.

```
10 REM *  READ DATA DEMO
20 REM *
30 DIM DAY$(20)
40 NUMOFDAYS = 7
50 PRINT
60 FOR DAYOFWEEK = 1 TO NUMOFDAYS
70 READ DAY$
80 PRINT DAY$
90 NEXT DAYOFWEEK
100 DATA SUNDAY, MONDAY, TUESDAY
110 DATA WEDNESDAY, THURSDAY, FRIDAY
120 DATA SATURDAY
```

2. Campers should RUN the program to see what is printed on the screen. Have the campers change the NUMOFDAYS assignment to NUMOFDAYS = 2. RUN the program again to see what happens. Then have the campers assign 8 days to the NUMOFDAYS variable. Once again, RUN the program to see what happens.

Activity # 2

1. A conditional statement, such as IF . . THEN, also can be used to set up a loop to read data. A GOTO, intructs the computer to re-execute a set of instructions until the IF . . THEN condition is met. The IF . . THEN statement searches for the "flag" at the end of the list of data. The flag is an unlikely element of data that has been placed at the end of the data list to indicate that there is no more data. The IF . . THEN statement checks each element of data that is READ and when the flag is encountered, the loop is terminated. In the following activity, the word "FINISH" has been placed at the end of the list of days to serve as a flag.

```
210 DATA SATURDAY,7TH,FINISH
```

When FINISH is READ, the program ends. You may want to take this opportunity to point out to the campers that using a conditional statement with a "flag" makes a program more versatile. The data can be changed or replaced with another set of data, and the program will still work, as long as the flag is the last element of data.

2. More than one variable can be READ from a list of data. The program below shows how two string variables can be assigned values from the same DATA statement. Explain to the students that the pointer continues to advance to the next element of data each time a READ instruction is executed, regardless of which variable is being assigned a value. Have the campers type in the program listed below in order to experiment with the instructions. This time, a conditional loop is used with a flag.

```
100 REM *  TWO STRING VARIABLES
110 REM *
120 DIM DAY$(15),ORDER$(10)
130 PRINT
140 READ DAY$
150 IF DAY$="FINISH" THEN END
160 READ ORDER$
170 PRINT DAY$;" IS THE ";ORDER$;" OF THE WEEK."
180 GOTO 140
190 DATA SUNDAY,1ST,MONDAY,2ND.TUESDAY,3RD
200 DATA WEDNESDAY,4TH,THURSDAY,5TH,FRIDAY,6TH
210 DATA SATURDAY,7TH,FINISH
220 END
```

If the campers' output from this program looks odd, remind them to check their spacing on line 170.

Note that when DAY$ was compared with the FINISH flag on line 150, FINISH was in quotes. When a string is compared with a word, the word must appear in quotes. Have the campers type ",FINISH" at the end of line 190. Line 190 should look like this after it has been edited.

```
190 DATA SUNDAY,FIRST,MONDAY,SECOND,TUESDAY,THIRD,FINISH
```

Ask them to predict what the program will print before RUNing it. Then insert a space before the word FINISH on line 190 and see what happens.

```
190 DATA SUNDAY,FIRST,MONDAY,SECOND,TUESDAY,THIRD, FINISH
                                                  ↑
```

Ask the campers to explain why "FINISH" did not end the program this time.

Activity #3

READ statements can be used to assign values to numeric variables as well. In fact, one DATA list can hold both numeric and string data. Ask the campers to type in the program listed below. Each camper should list the names of each member in his or her family and the person's age, separated by commas, on line 190-200.

Since there is no way of knowing how many people there are in each camper's family and therefore no way of knowing how much data will be listed on lines 190-200, a different kind of loop for reading the data is used. This time the TRAP instruction is used. Before to beginning the READ . . DATA loop, a TRAP is set to avoid an "OUT OF DATA" error message. When the READ statement on line 150 is executed, if there is no data left, ordinarily the program would stop and you would get an error message. By using the TRAP 210 instruction, we can trap that error and redirect the computer to continue with the instruction listed on line 210. Explain this third method of reading data, and have the campers experiment with the program listed below.

```
100 REM *       TRAPPED DATA
110 REM *
120 DIM PERSON$(20)
130 PRINT
140 TRAP 210
150 READ PERSON$
160 READ AGE
170 PRINT PERSON$;" IS ";AGE;" YEARS OLD."
180 GOTO 140
190 DATA MARGIE,47,JOHN,50,BETH,12
200 DATA
210 END
```

RESTORE

Activity #1

READ and DATA statements are commonly and quite conveniently used to play music on the Atari. The values for the voice, pitch, distortion, and/or loudness of the note can be stored as DATA and read as the tune is played. In this activity the RESTORE command is introduced in a program that plays music. Campers will use the RESTORE instruction to reset the pointer to the beginning of a list of SOUND data, so that the tune will be replayed.

Explain the RESTORE instruction to the campers. RESTORE resets the data pointer back to the first element of DATA listed in the program. (It is also possible to give a line number in the RESTORE instruction in order to redirect the pointer to a specific set of DATA. This will be discussed in the next activity.)

You also may need to briefly review the four components of the SOUND command. In the following program four values are being READ from DATA: PITCH, DISTORTION, LOUDNESS, and TIME. TIME indicates the length of the delay between each note. The voice is always zero. A minus one is used as a flag at the end of the DATA.

Have the campers type in the program listed below.

```
100 REM *      MUSIC
110 REM *
120 READ PITCH,DISTORT,LOUD,TIME
130 IF TIME = -1 THEN GOTO 210
140 SOUND 0,PITCH,DISTORT,LOUD
150 FOR DELAY = 1 TO TIME : NEXT DELAY
160 GOTO 120
170 DATA 121,10,10,40,91,10,10,37
180 DATA 0,0,0,3,91,10,10,40,108,10,10,28
190 DATA 0,0,0,2,108,10,10,10,91,10,10,30
200 DATA 108,10,10,10,121,10,10,80,0,0,0,0,-1
210 END
```

Note that the three variables being read can be listed with one READ instruction on line 120. RUN the program to hear the little tune.

Explain how to insert the RESTORE instruction on line 130 in order to repeat the tune. The edited format for line 130 appears below.

```
130 IF TIME = -1 THEN RESTORE: GOTO 120
```

RUN the edited program. To stop the program and turn off the sound, you must press SYSTEM RESET because the program is in an infinite loop. After this activity, you may never want to teach the READ, DATA, and RESTORE instructions using music in a classroom with 12 computers again.

Activity #2

This next activity gives the campers an opportunity to experiment with a more lengthy program that uses the READ, DATA, and RESTORE statements.

Explain to the campers that READ and DATA statements can go anywhere in a program. If the program is quite short, or if there is only one set of data used repeatedly from various places in the program, we recommend that the DATA be placed at the end of the program. Otherwise, the DATA should follow soon after the READ statement for program clarity.

Giving the DATA line number in the RESTORE command enables the programmer to dictate which part of the data the pointer will be restored to. So if a RESTORE instruction reads, RESTORE 400, the data pointer is reset to the first element of data appearing on line 400.

The following program is the beginning of an interactive art show. The person using the program selects from a list of small pictures to make a scene on the screen. The program uses the READ and DATA statements to draw the little pictures on the screen. The coordinates for each image are stored in separate DATA lists. The READ statement reads each set of coordinates to be used by the DRAWTO instruction on line 270. This is a very efficient way of storing computer illustration data. The RESTORE instruction is critical in this program. Once a shape has been selected by the program user, the RESTORE instruction is used to set the data pointer to the specific set of coordinates needed to draw the shape.

Note how the RESTORE command is used to control which list of data is being READ.

Since the READ statement on line 250 reads both X and Y, two flags are required at the end of the list of DATA. Otherwise the READ statement on line 250 will continue to look for DATA for Y before it will check for the flag on line 260. If there is no more DATA left, you will get an Out of DATA Error.

Ask the campers to explain why a TRAP instruction cannot be used to end the READ loop in lines 230 or 280. Have the campers predict what would happen if the line numbers were removed from the RESTORE commands in lines 150-170.

```
100 REM *     ART SHOW
105 REM *
110 MENU=900:REM MENU LINE NUMBER
115 GRAPHICS 7:COLOR 3
120 REM *
125 REM *****      MAIN LOOP    *****
130 GOSUB MENU
140 INPUT RESPONSE
150 IF RESPONSE<1 OR RESPONSE>4 THEN 140
160 IF RESPONSE=1 THEN RESTORE 500
170 IF RESPONSE=2 THEN RESTORE 600
180 IF RESPONSE=3 THEN RESTORE 700
190 IF RESPONSE=4 THEN RESTORE 800
200 REM *
210 REM *****      DRAW ROUTINE      *****
220 REM *
230 READ X,Y
240 PLOT X,Y:REM PICTURE START POINT
250 READ X,Y:REM GET DRAWTO DATA
260 IF X=-1 THEN 130:REM THE FLAG?
270 DRAWTO X,Y
280 GOTO 250:REM GET MORE DATA
290 REM *
500 REM *****      MOUNTAIN      *****
510 REM *
520 DATA 0,26,12,20,20,23,30,18,35,12,42,13,45,10,58,6
530 DATA 62,3,70,1,82,3,90,8,102,20,112,26,120,23
540 DATA 130,38,135,36,150,43,-1,-1
550 REM *
600 REM *****      BARN          *****610 REM *
610 REM *
620 DATA 43,50,43,46,47,46,47,50,40,50
630 DATA 40,44,45,41,50,44,50,50,40,50,-1,-1
640 REM *
700 REM *****      STAR      *****
710 REM *
720 DATA 128,10,127,11,126,11,127,12,126,13,127,13,128,14,129,13
730 DATA 130,13,129,12,130,11,129,11,128,11,128,14,-1,-1
740 REM *
800 REM *****      HORSE      *****
810 REM *
820 DATA 52,47,54,46,54,45,54,50,54,48,57,48,58,47,57,48,57,50,-1,-1
900 REM *
910 REM *****  MENU      *****
920 REM *
930 PRINT
940 PRINT "1.  MOUNTAIN      3.  STAR"
950 PRINT "2.  BARN          4.  HORSE"
960 PRINT "WHICH PICTURE (1, 2, 3, OR 4) ";
970 RETURN
```

PROGRAMMING CHALLENGES USING READ, DATA, AND RESTORE

1. Write a program that lists the necessary values for a song in DATA statements. Use the RESTORE command to repeat the chorus of the song in between the verses.

2. Add pictures to the ART SHOW program.

3. Set up a PLOT subroutine like the DRAWTO subroutine in the ART SHOW program. Add a plotted image to the list of pictures one can put on the screen. For example, offer to draw stars and PLOT tiny dots in the sky using READ and DATA statements.

4. Allow the user to draw the shapes anywhere on the screen. For example, when a person selects the barn, ask for the coordinates of the cabin's location on the screen. Be sure to tell the user the range of possible coordinates when asking for INPUT. Then add the person's coordinates to the DATA coordinates in the DRAWTO statement (DRAWTO X+XCOOR,Y+YCOOR). This way the person can put as many barns on the screen as he or she wants as well as put them anywhere in the picture.

5. Draw the shapes in different colors. Store the color of the shape as the first element of the shape data. READ the value into the COLOR instruction before drawing the image.

6. Experiment with combining different graphic modes, colors, sounds, etc. into a little show using the READ, DATA, and RESTORE statements.

READ, DATA, AND RESTORE

CAMPER COPY

***** Read DATA Demo *****

```
10 REM *  READ DATA DEMO
20 REM *
30 DIM DAY$(20)
40 NUMOFDAYS = 7
50 PRINT
60 FOR COUNTER = 1 TO NUMOFDAYS
70 READ DAY$
80 PRINT DAY$
90 NEXT COUNTER
100 DATA SUNDAY, MONDAY, TUESDAY
110 DATA WEDNESDAY, THURSDAY, FRIDAY
120 DATA SATURDAY
```

***** READ Two String Variables *****

```
100 REM *  TWO STRING VARIABLES
110 REM *
120 DIM DAY$(15),ORDER$(10)
130 PRINT
140 READ DAY$
150 IF DAY$="FINISH" THEN GOTO 220
160 READ ORDER$
170 PRINT DAY$;" IS THE ";ORDER$;" OF THE WEEK."
180 GOTO 140
190 DATA SUNDAY,FIRST,MONDAY,SECOND.TUESDAY,THIRD
200 DATA WEDNESDAY,FOURTH,THURSDAY,FIFTH,FRIDAY,SIXTH
210 DATA SATURDAY,SEVENTH,FINISH
220 END
```

```
190 DATA SUNDAY,FIRST,MONDAY,SECOND,TUESDAY,THIRD,FINISH
```

```
190 DATA SUNDAY,FIRST,MONDAY,SECOND,TUESDAY,THIRD, FINISH
                                                   ↑
```

***** Trapped DATA *****

```
100 REM *       TRAPPED DATA
110 REM *
120 DIM PERSON$(20)
130 PRINT
140 READ PERSON$
150 TRAP 210
160 READ AGE
170 PRINT PERSON$;" IS ";AGE;" YEARS OLD."
180 GOTO 140
190 DATA MARGIE,47,JOHN,50,BETH,12
200 DATA
210 END
```

***** Music *****

```
100 REM *      MUSIC
110 REM *
120 READ PITCH,DISTORT,LOUD,TIME
130 IF TIME = -1 THEN GOTO 210
140 SOUND 0,PITCH,DISTORT,LOUD
150 FOR DELAY = 1 TO TIME : NEXT DELAY
160 GOTO 120
170 DATA 121,10,10,40,91,10,10,37
180 DATA 0,0,0,3,91,10,10,40,108,10,10,28
190 DATA 0,0,0,2,108,10,10,10,91,10,10,30
200 DATA 108,10,10,10,121,10,10,80,0,0,0,0,-1
210 END
```

```
100 REM *    ART SHOW
105 REM *
110 MENU=900:REM MENU LINE NUMBER
115 GRAPHICS 7:COLOR 3
120 REM *
125 REM *****     MAIN LOOP   *****
130 GOSUB MENU
140 INPUT RESPONSE
150 IF RESPONSE<1 OR RESPONSE>4 THEN 140
160 IF RESPONSE=1 THEN RESTORE 500
170 IF RESPONSE=2 THEN RESTORE 600
180 IF RESPONSE=3 THEN RESTORE 700
190 IF RESPONSE=4 THEN RESTORE 800
200 REM *
210 REM *****     DRAW ROUTINE     *****
220 REM *
230 READ X,Y
240 PLOT X,Y:REM PICTURE START POINT
250 READ X,Y:REM GET DRAWTO DATA
260 IF X=-1 THEN 130:REM THE FLAG?
270 DRAWTO X,Y
280 GOTO 250:REM GET MORE DATA
290 REM *
500 REM *****     MOUNTAIN     *****
510 REM *
520 DATA 0,26,12,20,20,23,30,18,35,12,42,13,45,10,58,6
530 DATA 62,3,70,1,82,3,90,8,102,20,112,26,120,23
540 DATA 130,38,135,36,150,43,-1,-1
550 REM *
600 REM *****     BARN        *****610 REM *
610 REM *
620 DATA 43,50,43,46,47,46,47,50,40,50
630 DATA 40,44,45,41,50,44,50,50,40,50,-1,-1
640 REM *
700 REM *****     STAR     *****
710 REM *
720 DATA 128,10,127,11,126,11,127,12,126,13,127,13,128,14,129,13
730 DATA 130,13,129,12,130,11,129,11,128,11,128,14,-1,-1
740 REM *
800 REM *****     HORSE     *****
810 REM *
820 DATA 52,47,54,46,54,45,54,50,54,48,57,48,58,47,57,48,57,50,-1,-1
900 REM *
910 REM *****  MENU     *****
920 REM *
930 PRINT
940 PRINT "1.  MOUNTAIN      3.  STAR"
950 PRINT "2.  BARN          4.  HORSE"
960 PRINT "WHICH PICTURE (1, 2, 3, OR 4) ";
970 RETURN
```

ARRAYS

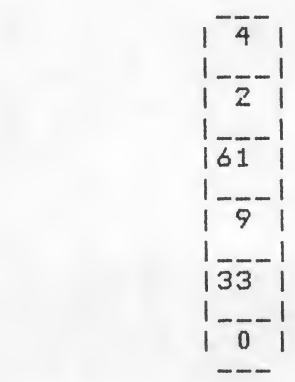
Arrays enable a programmer to store and manipulate large quantities of information systematically and efficiently. Arrays are one of the more complex features of Atari BASIC, and it may take some practice for your campers to fully understand them. You may find that you need to spend time drawing representations of arrays and stepping through programs on the board more than you have in previous modules. One objective of this module is to help campers to recognize programming problems that are well-suited to using an array. In this module the campers will use an array to reverse the order of a list of numbers and to write and store a musical tune.

To do this module you will need a chalk board or large pieces of chart paper and felt tip pens to map out arrays.

If you feel unsure of how to use an array in a program or you would like to see more examples of arrays, you may want to look over the following references.

ATARI 400/800 BASIC Reference Manual: pp. 3, 41-43
Inside Your Atari: pp. 66-74
Your ATARI Computer: pp 65-67

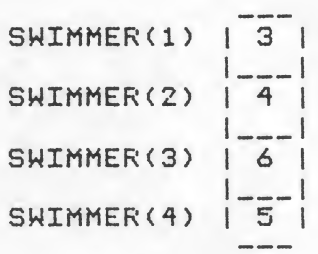
An array can be thought of as a series of boxes, each of which holds one numeric value. The following is an example of an array.



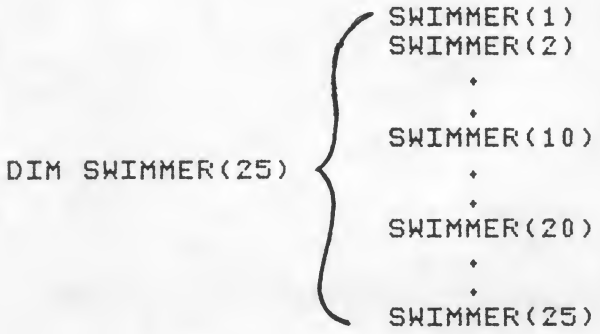
An array is a group of numeric variables that have something in common. The values in the array, for example, can be students' scores on a test or the number of raffle tickets each teacher sold for the school bake sale. Once you have stored the values in an array, you can do all kinds of calculations on the numbers.

ARRAYS
(Continued)

An array of swimmers' times in a race could be called SWIMMER. Each number in the SWIMMER array is called an "element" of the array. An index is used to identify which element of the SWIMMER array or which swimmer you are referring to. For example, in the array below, SWIMMER(1) swam the race in three minutes. SWIMMER(4) took five minutes.



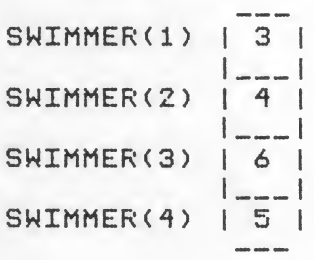
Just as we must reserve space in memory for a string, memory must be reserved for an array. You use the DIM statement to dimension the maximum number of boxes or elements your array could possibly need. DIM SWIMMER(25) enables the programmer to store up to 25 elements or swim times in the SWIMMER array.



As long as there is sufficient room in memory, there is no restriction on the size of an array that can be stored.

ARRAYS
(Continued)

Values are stored in an array by specifying the name of the array and the index of the element. For example, SWIMMER(1)=3 assigns a value of 3 to the first element of the SWIMMER array or swimmer number one. SWIMMER(3) = 6 assigns a 6 to the third element of the SWIMMER array, swimmer number three.



The index to an array can contain a numeric value or variable name. For example, the third element of the SWIMMER array is assigned a 6 in the code below as well.

```
NUM = 3
SWIMMER(NUM)=6
```

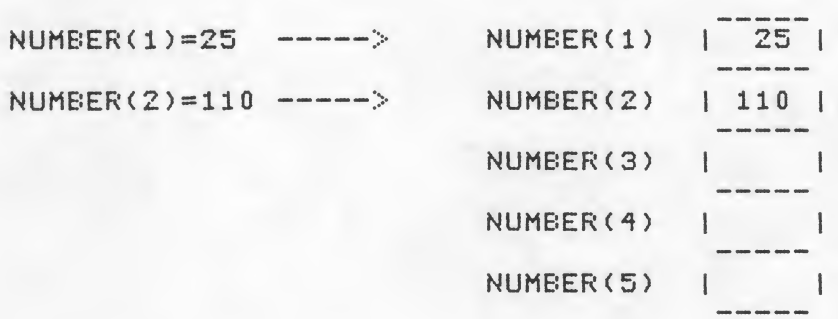
Using a variable index is particularly useful when a FOR . . NEXT loop is used to initialize an array. To inititialize an array you usually want to fill it with zeros.

```
FOR ELEMENT = 1 TO MAXELEMENTS
SWIMMER(ELEMENT) = 0
NEXT ELEMENT
```

Activity #1

1. Begin by explaining an array as a series of boxes, as was just explained in the introduction to this module. Drawing the boxes on the board is helpful when arrays are being introduced. Explain what an index is and how it is used to identify one element of an array. The format for the DIM statement of an array also will need to be explained.

2. Draw an array on the board that will hold five numbers, like the one below.



Ask campers to suggest the numbers to be stored in the array. Assign the value to the element of the array (NUMBER(3)=98) before filling the box in the array.

3. Ask the campers how they might go about printing one of these values in the array on the screen. Write out the command on the board as it is given. For example, PRINT NUMBER(1) or PRINT NUMBER(2). Be sure the campers understand the difference between PRINT 2 and PRINT NUMBER(2).

4. Explain that a variable name can be used as an index to an array, as was explained in the introduction to this module. Demonstrate the use of a variable as an index when using a FOR . . NEXT loop to PRINT out all of the elements of an array. A sample listing of the code appears below.

```
MAXNUMS = 5
FOR ELEMENT = 1 TO MAXNUMS
PRINT NUMBER(ELEMENT)
NEXT ELEMENT
```

Write this subroutine on the board alongside a series of boxes holding the elements of the NUMBER array. Step through the subroutine, printing each number on the board as the PRINT NUMBER(ELEMENT) statement is executed.

5. Given the FOR . . NEXT loop above to PRINT the numbers on the screen, ask the campers how they could program the computer to PRINT the numbers in the reverse order. A FOR . . NEXT loop like the following should evolve from the campers suggestions.

```
FOR ELEMENT = MAXNUMS TO 1 STEP -1
PRINT NUMBER(ELEMENT)
NEXT ELEMENT
```

Encourage the campers to use descriptive variable names. Again, step through the program, PRINTing the numbers that would be output on the board.

If you think your campers are catching on to arrays and variable indices, proceed with the next activity. Otherwise, review the previous examples, using a different array name and new data.

ARRAYS
(Continued)

Activity #2

1. In this activity the campers will gradually enter a program that INPUTs five numbers, stores them in an array, PRINTs them on the screen, and then PRINTs them in the reverse order. Note that the variable MAXNUMS can be used to DIMension the array. Have the campers type in the following program.

```
100 REM *      ARRAY OF NUMBERS
110 REM *
120 MAXNUMS = 5 :REM MAXIMUM NUMBERS WHICH CAN BE INPUT
130 DIM NUMS(MAXNUMS)
140 REM *
150 REM *      FILL ARRAY
160 REM *
170 FOR COUNT = 1 TO MAXNUMS
180 PRINT "TYPE IN A NUMBER";
190 INPUT VALUE
200 NUMS(COUNT) = VALUE :REM STORE INPUT VALUE IN ARRAY
210 NEXT COUNT
```

Ask the campers to add a line at the end of the program to PRINT the first element of the array. Add another line that PRINTs the last number in the array, using MAXNUMS as the index to the NUMS array.

2. Have the campers type in the following lines as a continuation of the Array of Numbers program. The following lines PRINT the entire sequence of numbers in the array. RUN the program.

```
220 REM *
230 REM *      PRINT NUMBERS IN ORDER
240 REM *
250 PRINT
260 PRINT "YOU TYPED IN THE NUMBERS IN THE"
270 PRINT "FOLLOWING ORDER:"
280 PRINT
290 FOR COUNT = 1 TO MAXNUMS
300 PRINT NUMS(COUNT);"  "; :REM PRINT VALUE + SPACES
310 NEXT COUNT
320 PRINT
```

3. Based on what the campers know, ask them to complete the following lines, which print the numbers in the array in the reverse order.

```
330 REM *
340 REM *     PRINT NUMBERS IN REVERSE ORDER
350 REM *
360 PRINT
370 PRINT "YOUR NUMBERS IN THE REVERSE ORDER ARE:"
380 PRINT
390 FOR COUNT = _______ TO ___ STEP ___
400 PRINT _____(_____);"  ";
410 NEXT _____
420 PRINT
430 END
```

The campers should RUN the program to see if it does what they intended.

Activity #3

1. This next activity demonstrates additional uses of an array. Once again, the program INPUTs numbers. This time, the numbers are used as the PITCH for a SOUND command. The person using the program can create a little tune. Start by discussing the program with the campers.

2. First, pose the question, "Suppose you wanted to use an array to write a tune. What instruction must appear in the program before the computer will allow you to use an array?" (Answer: DIM TUNE(???)) When you get the appropriate answer, draw an array of boxes on the board alongside the dimension statement.

3. Now explain that as a programmer you want to INPUT the numbers that will serve as the values for the PITCH in the SOUND command. You want to store the PITCH values in an array in order to play back the tune for the program user. List the following INPUT code and ask the campers to explain what it does.

```
100 REM *  SOUND WITH AN ARRAY
110 REM *
120 DIM TUNE(100)
130 XNOTE = 0
140 INPUT PITCH
150 IF PITCH = -1 THEN NUMNOTES = XNOTE:GOTO 200
160 XNOTE = XNOTE + 1
170 TUNE(XNOTE) = PITCH
180 GOTO 140
```

The minus one serves as a flag, indicating that the last note of the tune has been entered. Using a flag enables the user to enter as many notes as he or she wishes. The user must be told to type in a minus one as a flag for the last note of the tune.The INPUT instruction is in a loop that continually checks for the minus one flag. Explain to the campers that XNOTE is used instead of NOTE because NOTE is a reserve word. NUMNOTES is assigned the total number of notes typed in. Step through the INPUT routine a few times and place some INPUT values in the box array you drew beside the DIM statement.

4. Now that the notes are stored in the TUNE array, we need to write the BASIC routine which will play the tune. The idea here is to get the campers to generate the code from your English description of the program. To play the tune, the routine must use each of the elements of the array in the SOUND command. Ask what would be an efficient way to use each value in the SOUND instruction. Establish a FOR loop using the same variable names as were introduced in the INPUT routine.

```
FOR NUM = 1 TO XNOTE
```

In this example use voice zero in the SOUND command and use ten for both the distortion and loudness for all of the notes. Thus, use the following SOUND command:

```
SOUND 0,TUNE(XNOTE),10,10
```

You might give the campers the SOUND command, but leave out the TUNE(XNOTE) and ask the campers what they would insert.

In order to distinguish the notes, you need a delay loop after playing each note.

```
FOR DELAY = 1 TO 10: NEXT DELAY
```

When the loop ends, the SOUND must be turned off before leaving the subroutine. (SOUND 0,0,0,0)

```
200 REM *
210 REM *      PLAY TUNE
220 REM *
230 FOR XNOTE = 1 to NUMNOTES
240 SOUND 0,TUNE(XNOTE),10,10
250 FOR DELAY = 1 TO 10: NEXT DELAY
260 NEXT XNOTE
270 SOUND 0,0,0,0
```

5. Have the campers type in the entire SOUND WITH AN ARRAY program which you have just worked through. RUN the program and experiment with different notes.

```
100 REM *  SOUND WITH AN ARRAY PLUS PLAY TUNE
110 REM *
120 DIM TUNE(100)
130 XNOTE = 0
140 INPUT PITCH
150 IF PITCH = -1 THEN NUMNOTES = XNOTE:GOTO 200
160 XNOTE = XNOTE + 1
170 TUNE(XNOTE) = PITCH
180 GOTO 140
200 REM *
210 REM *      PLAY TUNE
220 REM *
230 FOR XNOTE = 1 to NUMNOTES
240 SOUND 0,TUNE(XNOTE),10,10
250 FOR DELAY = 1 TO 10: NEXT DELAY
260 NEXT XNOTE
270 SOUND 0,0,0,0
```

6. To see how the same BASIC code with a menu and a few extra PRINT statements can work, have the campers look over the listing of the SOUNDARY program on their arrays worksheets. Note how labels have been assigned to the beginning line numbers of the different subroutines in lines 170-200. Each subroutine is "called" from the main loop.

7. Note that the third option on the menu is "LIST THE NOTES." Have the campers look over lines 900,960. This FOR . . NEXT loop will PRINT the values stored in the array on the screen.

```
930 FOR XNOTE = 1 TO NUMNOTES
940 PRINT "TUNE(";XNOTE;")";"  ";TUNE(XNOTE)
950 NEXT XNOTE
960 RETURN
```

The PRINT statement in line 940 needs to be given careful explanation. Have the campers RUN the SOUNDARY program to see how it works. Take a few minutes to experiment with entering different tunes.

```
100 REM *     SOUND ARRAY
110 REM *
120 REM *                                                SOUNDARY
130 REM *  INITIALIZE VARIABLES AND ARRAY
140 REM *
150 DIM TUNE(100)
160 XNOTE=0
165 REM * ASSIGN LABELS TO LINE NUMBERS
170 MENU=300
180 VALUES=500
190 PLAY=700
200 NUMBERS=900
210 REM *
220 REM *     MAIN LOOP
230 REM *
240 GOSUB MENU
250 INPUT RESPONSE
260 IF RESPONSE=1 THEN GOSUB VALUES
270 IF RESPONSE=2 THEN GOSUB PLAY
280 IF RESPONSE=3 THEN GOSUB NUMBERS
290 GOTO 240:REM REPEAT MAIN LOOP
300 REM *
310 REM *     MENU
320 REM *
330 PRINT
340 PRINT "WOULD YOU LIKE TO:"
350 PRINT "     1.  TYPE IN A TUNE."
360 PRINT "     2.  PLAY YOUR TUNE."
370 PRINT "     3.  LIST THE NOTES."
380 PRINT
390 PRINT "TYPE IN A NUMBER";
400 PRINT :REM INPUT IN MAIN LOOP
410 RETURN
500 REM *
510 REM *     INPUT VALUES FOR NOTES
520 REM *
530 PRINT "  TYPE IN NUMBERS BETWEEN 0"
540 PRINT "  AND 255 TO BE THE NOTES"
550 PRINT "  OF A TUNE.  TYPE ONE NOTE"
560 PRINT "  PER ?.  WHEN YOU ARE FINISHED,"
570 PRINT "  TYPE A -1 FOR THE LAST NOTE."
580 INPUT PITCH
590 IF PITCH>255 OR PITCH<-1 THEN 580
600 REM *  MINUS ONE IS A FLAG FOR THE END OF THE DATA
610 IF PITCH=-1 THEN NUMNOTES=XNOTE:RETURN
620 XNOTE=XNOTE+1:REM NOTES COUNTER
630 TUNE(XNOTE)=PITCH
640 GOTO 580
700 REM *
710 REM *     PLAY TUNE
720 REM *
730 FOR XNOTE=1 TO NUMNOTES
740 SOUND 0,TUNE(XNOTE),10,10
750 FOR DELAY=1 TO 10:NEXT DELAY
760 NEXT XNOTE
770 SOUND 0,0,0,0
780 RETURN
900 REM *
910 REM *     LIST NOTES
920 REM *
930 FOR XNOTE=1 TO NUMNOTES
940 PRINT "TUNE(";XNOTE;")";"  ";TUNE(XNOTE)
950 NEXT XNOTE
```

Activity #4

Thus far we have used INPUT to fill the elements of an array. It is also possible to fill an array by READing DATA from within a program into an array. For example, music data could be stored in the program, read into an array, and sections of the array could be called at given times in order to play a verse and then repeat a melody.

However, be careful with the READ statement. In Atari BASIC the READ instruction will only accept a simple variable, not an indexed variable. For example, the computer will accept the following READ statement.

```
READ XNOTE
```

The computer will not accept the following.

```
READ TUNE(3) or READ TUNE(XNOTE)
```

Thus, you must READ the DATA into a simple variable and then transfer that value into the array.

```
READ NUMBER : TUNE(1) = NUMBER

or

READ NUMBER : TUNE(XNOTE) = NUMBER
```

Have the campers RUN the TUNE program on their BASIC Utility Disk and look over the program in their program listings. The TUNE program is essentially the same as the MUSIC program in the READ, DATA, AND RESTORE Module. However, having the data in an array enables you to repeat any note or sequence of notes in the tune. Have the campers experiment with changing the values for START and FINISH in order to make up different tunes.

```
100 REM *         TUNE ARRAY
110 REM *
120 DIM PITCH(50),DISTORT(50),LOUD(50),TIME(50)
130 INIT=500:REM INITIALIZATION LINE#
140 PLAY=300:REM PLAY TUNE ROUTINE
150 MAXNOTES=11
200 REM *
210 REM *****  MAIN LOOP  *****
220 REM *
230 GOSUB INIT
240 START=1:FINISH=5:GOSUB PLAY
250 START=6:FINISH=11:GOSUB PLAY
260 START=1:FINISH=4:GOSUB PLAY
270 END
300 REM
310 REM *****  PLAY  *****
320 REM *
330 REM *  PLAYS A SEQUENCE OF NOTES USING DATA ARRAYS.
340 REM *  INDICES DETERMINED BY VALUES OF START AND
350 REM *  FINISH IN MAIN LOOP
360 REM *
370 FOR XNOTE=START TO FINISH
380 SOUND 0,PITCH(XNOTE),DISTORT(XNOTE),LOUD(XNOTE)
390 FOR DELAY=1 TO TIME(XNOTE):NEXT DELAY
400 NEXT XNOTE
410 RETURN
420 REM *
500 REM *****  INIT ARRAY  *****
510 REM *
520 FOR FILL=1 TO MAXNOTES
530 READ PITCH,DISTORT,LOUD,TIME
540 PITCH(FILL)=PITCH:DISTORT(FILL)=DISTORT:LOUD(FILL)=LOUD:TIME(FILL)=TIME
550 NEXT FILL
560 RETURN
570 DATA 121,10,10,40,91,10,10,37,0,0,0,3,91,10,10,40,108,10,10,28
580 DATA 0,0,0,2,108,10,10,10,91,10,10,30,108,10,10,10,121,10,10,80,0,0,0,0
```

Activity #5

A Group Discussion:

In each of the programs where an array was used, all the data needed to be stored before you could do anything with it. For example, in the first program, you needed to know all the numbers before you could print them in the reverse order. In the SOUNDARY program, all the pitch values needed to be stored before they could be played back as a tune. Arrays are especially useful whenever you need to have access to all of the data before any thing is done with it. Ask the campers to think about each of the following programming problems and explain why using an array is or is not a good idea in each case.

Find the average of five numbers that are INPUT by the program user. (Answer: No, because the numbers can be added together as they are INPUT into the computer and then divided by the number of values typed in.)

```
SUM = 0
FOR COUNT = 1 TO 5
INPUT NUM
SUM = SUM + NUM
NEXT COUNT
AVERAGE = SUM/5
```

Given a list of numbers, which numbers are larger than the average of the numbers. (Answer: Yes. All the numbers must be saved in order to look back and see which are greater than the average.) Ask campers to try to solve this without an array. What if you knew there would only be five numbers?

Can you think of some more examples?

PROGRAMMING CHALLENGES USING ARRAYS

1. Store your own tune data in the SOUNDARY program. READ it into an array and play different sequences of notes from the main loop.

2. Give the person using your program the option to edit his or her tune, one note at a time. Ask the person which note he or she wants to change and use that number as the index to your array. Then ask what the new note value will be. Change the array element accordingly and return to the menu.

3. Write a program which INPUTs a series of numbers, and then lists those numbers which are larger than the average of all the numbers which were typed in. Use an array to store the values which the person types in.

4. Use two different arrays, HUE(100) and LUM(100), to create a light show. Your program should contain a statement like the following one:

```
SETCOLOR 0,HUE(COUNT),LUM(COUNT)
```

```
100 REM *      ARRAY OF NUMBERS
110 REM *
120 MAXNUMS = 5 :REM MAXIMUM NUMBERS WHICH CAN BE INPUT
130 DIM NUMS(MAXNUMS)
140 REM *
150 REM *      FILL ARRAY
160 REM *
170 FOR COUNT = 1 TO MAXNUMS
180 PRINT "TYPE IN A NUMBER";
190 INPUT VALUE
200 NUMS(COUNT) = VALUE :REM STORE INPUT VALUE IN ARRAY
210 NEXT COUNT
```

```
220 REM *
230 REM *      PRINT NUMBERS IN ORDER
240 REM *
250 PRINT
260 PRINT "YOU TYPED IN THE NUMBERS IN THE"
270 PRINT "FOLLOWING ORDER:"
280 PRINT
290 FOR COUNT = 1 TO MAXNUMS
300 PRINT NUMS(COUNT);"  "; :REM PRINT VALUE + SPACES
310 NEXT COUNT
320 PRINT
```

```
100 REM *  SOUND WITH AN ARRAY
110 REM *
120 DIM TUNE(100)
130 XNOTE = 0
140 INPUT PITCH
150 IF PITCH = -1 THEN NUMNOTES = XNOTE:GOTO 200
160 XNOTE = XNOTE + 1
170 TUNE(XNOTE) = PITCH
180 GOTO 140
```

```
200 REM *
210 REM *      PLAY TUNE
220 REM *
230 FOR XNOTE = 1 to NUMNOTES
240 SOUND 0,TUNE(XNOTE),10,10
250 FOR DELAY = 1 TO 10: NEXT DELAY
260 NEXT XNOTE
270 SOUND 0,0,0,0
```

```
100 REM *  SOUND WITH AN ARRAY PLUS PLAY TUNE
110 REM *
120 DIM TUNE(100)
130 XNOTE = 0
140 INPUT PITCH
150 IF PITCH = -1 THEN NUMNOTES = XNOTE:GOTO 200
160 XNOTE = XNOTE + 1
170 TUNE(XNOTE) = PITCH
180 GOTO 140
200 REM *
210 REM *      PLAY TUNE
220 REM *
230 FOR XNOTE = 1 to NUMNOTES
240 SOUND 0,TUNE(XNOTE),10,10
250 FOR DELAY = 1 TO 10: NEXT DELAY
260 NEXT XNOTE
270 SOUND 0,0,0,0
```

```
100 REM *      SOUND ARRAY
110 REM *
120 REM *                                                SOUNDARY
130 REM *  INITIALIZE VARIABLES AND ARRAY
140 REM *
150 DIM TUNE(100)
160 XNOTE=0
165 REM * ASSIGN LABELS TO LINE NUMBERS
170 MENU=300
180 VALUES=500
190 PLAY=700
200 NUMBERS=900
210 REM *
220 REM *      MAIN LOOP
230 REM *
240 GOSUB MENU
250 INPUT RESPONSE
260 IF RESPONSE=1 THEN GOSUB VALUES
270 IF RESPONSE=2 THEN GOSUB PLAY
280 IF RESPONSE=3 THEN GOSUB NUMBERS
290 GOTO 240:REM REPEAT MAIN LOOP
300 REM *
310 REM *      MENU
320 REM *
330 PRINT
340 PRINT "WOULD YOU LIKE TO:"
350 PRINT "      1.  TYPE IN A TUNE."
360 PRINT "      2.  PLAY YOUR TUNE."
370 PRINT "      3.  LIST THE NOTES."
380 PRINT
390 PRINT "TYPE IN A NUMBER";
400 PRINT :REM INPUT IN MAIN LOOP
410 RETURN
500 REM *
510 REM *      INPUT VALUES FOR NOTES
520 REM *
530 PRINT "  TYPE IN NUMBERS BETWEEN 0"
540 PRINT "  AND 255 TO BE THE NOTES"
550 PRINT "  OF A TUNE.  TYPE ONE NOTE"
560 PRINT "  PER ?.  WHEN YOU ARE FINISHED,"
570 PRINT "  TYPE A -1 FOR THE LAST NOTE."
580 INPUT PITCH
590 IF PITCH>255 OR PITCH<-1 THEN 580
600 REM *  MINUS ONE IS A FLAG FOR THE END OF THE DATA
610 IF PITCH=-1 THEN NUMNOTES=XNOTE:RETURN
620 XNOTE=XNOTE+1:REM NOTES COUNTER
630 TUNE(XNOTE)=PITCH
640 GOTO 580
```

TUNE

```
100 REM *         TUNE ARRAY
110 REM *
120 DIM PITCH(50),DISTORT(50),LOUD(50),TIME(50)
130 INIT=500:REM INITIALIZATION LINE#
140 PLAY=300:REM PLAY TUNE ROUTINE
150 MAXNOTES=11
200 REM *
210 REM *****  MAIN LOOP  *****
220 REM *
230 GOSUB INIT
240 START=1:FINISH=5:GOSUB PLAY
250 START=6:FINISH=11:GOSUB PLAY
260 START=1:FINISH=4:GOSUB PLAY
270 END
300 REM
310 REM *****  PLAY  *****
320 REM *
330 REM *  PLAYS A SEQUENCE OF NOTES USING DATA ARRAYS.
340 REM *  INDICES DETERMINED BY VALUES OF START AND
350 REM *  FINISH IN MAIN LOOP
360 REM *
370 FOR XNOTE=START TO FINISH
380 SOUND 0,PITCH(XNOTE),DISTORT(XNOTE),LOUD(XNOTE)
390 FOR DELAY=1 TO TIME(XNOTE):NEXT DELAY
400 NEXT XNOTE
410 RETURN
420 REM *
500 REM *****  INIT ARRAY  *****
510 REM *
520 FOR FILL=1 TO MAXNOTES
530 READ PITCH,DISTORT,LOUD,TIME
540 PITCH(FILL)=PITCH:DISTORT(FILL)=DISTORT:LOUD(FILL)=LOUD:TIME(FILL)=TIME
550 NEXT FILL
560 RETURN
570 DATA 121,10,10,40,91,10,10,37,0,0,0,3,91,10,10,40,108,10,10,28
580 DATA 0,0,0,2,108,10,10,10,91,10,10,30,108,10,10,10,121,10,10,80,0,0,0,0
```

MATRICES

An even more complex and more powerful array is called a "matrix" or a "two dimensional array." A matrix is an array of arrays, which is best explained with diagrams. The activities in this module take the campers through the steps required to write a battleship game that uses a matrix.

If you would like to read about two dimensional arrays, or review some applications for matrices, you may want to look through the following references. (These are the same resources that were listed in the Arrays Module.)

ATARI 400/800 BASIC Reference Manual: pp.3. 41-43
Inside Your Atari: pp. 66-74
Your ATARI Computer: pp. 65-67

Suppose you had a list of swimmers' times from three different swim meets. And, suppose you wanted to compare each swimmer's time for the butterfly in each of the three meets. A matrix is especially well-suited to just such a problem. The following diagram illustrates how the swimmers' data can be stored in a matrix.

	SWIM MEETS		
	1	2	3
SWIMMER #1	25	28	27
SWIMMER #2	31	29	28
SWIMMER #3	28	30	26
SWIMMER #4	37	42	35

TIMES

The swim times for each meet are stored in separate columns. The swimmers are listed in the rows of the matrix. Swimmer number one swam the race in the second meet in 28 seconds. In BASIC, this is written in the following way.

MATRICES
(Continued)

```
SWIMMER(1,2) = 28
        /   \
     ROW     COLUMN
SWIMMER#     MEET#
```

Two indices are used. One holds the row value, which is the SWIMMER number. The second index is the column value, which in this case is the swim meet number. Swimmer number three's slowest time was in the second meet.

```
SWIMMER(3,2) = 30
        /   \
     ROW     COLUMN
SWIMMER#     MEET#
```

Activity #1

1. Any data which conveniently fits into a grid or matrix format, is well-suited to a two dimensional array in a BASIC program. Describe some programming examples in which you might use a two dimensional array. Explain the format of the indices for accessing an element of the matrix. Draw a matrix similar to the SWIMMER matrix on the board. Give the campers an opportunity to practice with indices by presenting some sample statements to complete, like the following examples.

```
SWIMMER(3,1) = ___
SWIMMER(2,1) = ___
SWIMMER(3,_) = 26
SWIMMER(_,1) = 28
```

2. The format of a DIM statement for a matrix is shown below.

```
DIM SWIMMER(4,3)
        /   \
  Maximum #    Maximum #
   of rows.    of columns
        \   /
     In the matrix called
         SWIMMER.
```

MATRICES
(Continued)

Activity #2

1. The next programming activity involves developing a BASIC program for a "battleship" game. It may be helpful for the campers to be able to simulate the program on the board first. Draw a four by four matrix on the board, like the one below.

```
                    COLUMNS

                1     2     3     4
          1           X                       LOCATION(1,2)
    R
    O     2
    W
    S     3

          4
```

In the program, the random number function will be used to secretly position a boat on the board. For now, play the game on the blackboard. Ask one camper to secretly decide where the boat will be hidden on the battleship game board. Then have the other campers guess where the boat is by giving the matrix name and the indices of the guessed location. List the guesses on the board beside the matrix and place an "X" on the matrix for each incorrect guess. Do this until the boat is found. You may want to play the game a couple of times in order to give the campers practice with using two indices.

MATRICES
(Continued)

Activity #3

1. To program this problem, first we must be sure that the battleship board is cleared to zero, so that no extraneous data in memory interferes with the game. This is called initializing the array. Explain that variables can be used as indices to a matrix. For example, BOARD(ROW,COLUMN)=0, will place a zero in the specified location in the matrix. Ask the campers to suggest how they might initialize the BOARD matrix, using two nested FOR loops (like the example listed in number 2 below). Record the initialization routine on the board, as the campers plan it. Draw a 4X4 matrix on the board, like the one above, and step through the loop routine, placing zeros in the matrix everytime a zero is assigned to a location.

2. Have the campers compare their code with the initialization routine listed below. Ask the campers to type in the following subroutine. Note that a variable can be used to dimension the matrix.

```
100 REM *     BATTLESHIP
110 REM *
120 REM *       INITIALIZE VARIABLES
130 REM *
140 MAXLOCATIONS=4
150 DIM BOARD(MAXLOCATIONS,MAXLOCATIONS)
160 COLUMN=0:ROW=0
200 REM *
210 REM *       INIT MATRIX
220 REM *
230 FOR ROW = 1 TO MAXLOCATIONS
240 FOR COLUMN = 1 TO MAXLOCATIONS
250 BOARD(ROW,COLUMN)=0:REM STORE A 0
260 PRINT LOCATION(ROW,COLUMN)
270 NEXT COLUMN
280 NEXT ROW
```

Line 260 prints the contents of each element of the matrix. RUN the program to confirm that each element of the matrix got a zero as planned. Be sure the campers understand how the nested FOR loops work in lines 230 and 240 by "playing computer" and filling in a matrix on the blackboard in the order the computer does it.

MATRICES
(Continued)

3. Now the ship must be hidden. Two random numbers must be generated, one for the row value and one for the column value in order to hide a ship in the matrix. A one is placed in the matrix in the location of the hidden ship in order to distinguish it from the empty spaces (zeros) on the board. Go over the following routine with the campers. Explain why and how the random numbers were used for the ship's location. Discuss why a one is put in the location of the ship. Ask campers what they think BOARD(SHIPROW,SHIPCOL) = 1 will do. Have the campers add the following code to their program which initializes the battleship matrix.

```
400 REM *
410 REM *      PLACE SHIP
420 REM *
430 SHIPROW = INT(RND(0)*40+1 :REM A RANDOM NUMBER
440 SHIPCOL = INT(RND(0)+4)+1 :REM BETWEEN 1 AND 4
450 BOARD(SHIPROW,SHIPCOL)=1  :REM ONE IS ASSIGNED TO RANDOM LOCATION
```

4. To play the game you must INPUT the user's guess and see if there is a "one" stored in that location. If the contents of the guessed location is one, the ship has been found. If not, let the user guess again. Explain, in English, the sequence of steps the program must take in order for someone to play the game. Have the campers come up with the BASIC code. Record the campers suggestions on the board, as they are given.

1. First, get the user's INPUT for the ROW value.

```
PRINT "ROW: ";: INPUT ROWGUESS
```

2. Get the COLUMN value.

```
PRINT "COLUMN: ";: INPUT COLGUESS
```

3. Check to see if the location in the matrix guessed by the program user holds a ship. If so, PRINT "YOU FOUND IT!".

```
IF BOARD(ROWGUESS,COLGUESS)=1 THEN PRINT "YOU FOUND IT!":END
```

4. Otherwise PRINT "TRY AGAIN" and go back to the line which INPUTs the ROW value.

Compare the campers' listing with the game play routine below.

```
700 REM *    PLAY
720 REM *
730 PRINT "TYPE IN THE COORDINATES OF"
740 PRINT "YOUR GUESS.  THE NUMBER MUST"
750 PRINT "BE BETWEEN 1 AND 4.
760 PRINT
770 PRINT "ROW: ";
780 INPUT ROWGUESS
790 PRINT "COLUMN: ";
800 INPUT COLGUESS
810 IF BOARD(ROWGUESS,COLGUESS)=1 THEN PRINT "YOU FOUND IT!":END
820 PRINT
830 PRINT "TRY AGAIN"
840 GOTO 770
```

Have the campers type in the PLAY routine. Remember, the matrix is only a 4X4 matrix. Remind campers that if they type in a coordinate for the ship which is less than one or greater than four the program will bomb.

Activity #4

1. To see how the same code can be dressed up with a few PRINT statements, RUN the SHIP program on the BASIC Utility Disk.

PROGRAMMING CHALLENGES USING MATRICES

1. Place more than one ship on the battleship board in a random location for the user to find.

2. Make the ship larger, so that it takes up two or three locations on the grid of elements in the matrix. Give the outer limits of the ship different values than the middle of the ship. This way, you can give the player feedback on his or her guess. (eg. "You hit the front of the ship.")

3. Give the player hints in response to his or her guesses. For example, if the program player's guess for the row position is one away from the ship, PRINT "YOUR ROW GUESS IS HOT!" If the next row guess is more than one away from the ship's location, PRINT "YOUR ROW INDEX IS GETTING COLDER." Do the same for the column values.

4. Place friendly ships and enemy ships on the game board. Give the friendly ships one value and the enemy ships another value in order to differentiate the two. Keep score for the player. When an enemy ship is encountered, the player gains points. When an allies ship is hit, the player looses points.

5. Experiment with different graphics modes, drawing a grid, and displaying the actual ships on the screen.

***** BATTLESHIP *****

```
100 REM *      BATTLESHIP
110 REM *
120 REM *       INITIALIZE VARIABLES
130 REM *
140 MAXLOCATIONS=4
150 DIM BOARD(MAXLOCATIONS,MAXLOCATIONS)
160 COLUMN=0:ROW=0
200 REM *
210 REM *       INIT MATRIX
220 REM *
230 FOR ROW = 1 TO MAXLOCATIONS
240 FOR COLUMN = 1 TO MAXLOCATIONS
250 BOARD(ROW,COLUMN)=0
260 PRINT LOCATION(ROW,COLUMN)
270 NEXT COLUMN
280 NEXT ROW
```

***** PLACE SHIP *****

```
400 REM *
410 REM *      PLACE SHIP
420 REM *
430 SHIPROW = INT(RND(0)*40+1 :REM A RANDOM NUMBER
440 SHIPCOL = INT(RND(0)+4)+1 :REM BETWEEN 1 AND 4
450 BOARD(SHIPROW,SHIPCOL)=1 :REM ONE IS ASSIGNED TO RANDOM LOCATION
```

***** PLAY *****

```
700 REM *     PLAY
720 REM *
730 PRINT "TYPE IN THE COORDINATES OF"
740 PRINT "YOUR GUESS.  THE NUMBER MUST"
750 PRINT "BE BETWEEN 1 AND 4.
760 PRINT
770 PRINT "ROW: ";
780 INPUT ROWGUESS
790 PRINT "COLUMN: ";
800 INPUT COLGUESS
810 IF BOARD(ROWGUESS,COLGUESS)=1 THEN PRINT "YOU FOUND IT!":END
820 PRINT
830 PRINT "TRY AGAIN"
840 GOTO 770
```

```
100 REM *      BATTLE SHIP
110 REM *
120 REM *
130 REM *  INITIALIZE VARIABLES
140 REM *
150 MAXLOCATIONS=4
160 DIM BOARD(MAXLOCATIONS,MAXLOCATIONS)
170 COLUMN=0:ROW=0
175 REM ASSIGN NAMES TO SUBROUTINE LINE NUMBERS
180 INITMATRIX=500
190 PLACESHIP=700
200 PLAY=900
210 WIN=1300
300 REM *
310 REM *      MAIN LOOP
320 REM *
330 GOSUB INITMATRIX
340 GOSUB PLACESHIP
350 GOSUB PLAY
360 END
500 REM *
510 REM *      INIT MATRIX
520 REM *
530 FOR ROW=1 TO MAXLOCATIONS
540 FOR COLUMN=1 TO MAXLOCATIONS
550 BOARD(ROW,COLUMN)=0
560 PRINT BOARD(ROW,COLUMN)
570 NEXT COLUMN
580 NEXT ROW
590 RETURN
700 REM *
710 REM *      PLACE SHIP
720 REM *
730 SHIPROW=INT(RND(0)*4)+1:REM RANDOM NUMBER
740 SHIPCOL=INT(RND(0)*4)+1:REM BETWEEN 1 AND 4
750 BOARD(SHIPROW,SHIPCOL)=1:REM PLACE SHIP IN RANDOM LOCATION
760 RETURN
900 REM *
910 REM *      PLAY
920 REM *
930 GRAPHICS 2
940 POSITION 0,0
950 PRINT #6;"        columns"
960 FOR NUMBER=1 TO 4
970 POSITION 2,NUMBER*2:REM ROW COORDINATES
980 PRINT #6;NUMBER
990 POSITION NUMBER*4,1:REM COLUMN COORDINATES
1000 PRINT #6;NUMBER
1010 NEXT NUMBER
```

```
1020 POSITION 0,3:PRINT #6;"R"
1030 POSITION 0,4:PRINT #6;"O"
1040 POSITION 0,5:PRINT #6;"W"
1050 POSITION 0,6:PRINT #6;"S"
1060 PRINT "  TYPE IN THE COORDINATES OF"
1070 PRINT "  YOUR GUESS.  THE NUMBER MUST"
1080 PRINT "  BE BETWEEN 1 AND 4."
1090 PRINT "ROW: ";
1100 INPUT ROWGUESS
1110 IF ROWGUESS<1 OR ROWGUESS>4 THEN 1090
1120 PRINT "COLUMN: ";
1130 INPUT COLGUESS
1140 IF COLGUESS<1 OR COLGUESS>4 THEN 1120
1150 IF BOARD(ROWGUESS,COLGUESS)=1 THEN GOSUB WIN:RETURN
1160 POSITION COLGUESS*4,ROWGUESS*2:REM PUT * ON BOARD
1170 PRINT #6;"*"
1180 PRINT :PRINT "TRY AGAIN"
1190 GOTO 1090
1300 REM *
1310 REM *       WIN
1320 REM *
1330 PRINT :PRINT "YOU FOUND IT!"
1340 FOR COUNT=1 TO 10
1350 POSITION COLGUESS*4,ROWGUESS*2
1360 PRINT #6;" ";:REM ERASE *
1370 FOR DELAY=1 TO 75:NEXT DELAY
1380 POSITION COLGUESS*4,ROWGUESS*2
1390 PRINT #6;"*";:REM FLASH *
1400 FOR DELAY=1 TO 75:NEXT DELAY
1410 NEXT COUNT
1420 RETURN
```

PEEK AND POKE

In the same way that human beings have all sorts of pieces of information stored in their brain like telephone numbers and lock combinations, the computer also maintains hundreds of pieces of information. When you turn on your computer the operating system stores all the necessary information the computer needs to process your programs. The PEEK instruction enables you to look at the contents of any memory location. The POKE instruction allows you to change a value stored in memory.

To read more about the PEEK and POKE instructions, you may want to consult the following books.

Atari 400/800 BASIC Reference Manual: pp. 35
Inside Atari BASIC: pp 132-139
Your Atari Computer: pp 113, 379, 398

All the information stored in memory is stored in the form of numbers. Each number is held in a separate location. Memory can be thought of as a long series of boxes, each holding one piece of information, a number. There are 65,536 memory locations in a 64K Atari. Each box or memory location has an address which enables the programmer to identify which of the memory locations he or she is referring to. When you use the PEEK or POKE instructions you must specify the address of the memory location you want.

The PEEK instruction enables us to peer in at the contents of the specified memory location. For example, PEEK(82) is the value stored in memory location 82. The address of the memory box you want to look at in this example, 82, is listed in parentheses following the PEEK instruction. Memory location 82 happens to hold the number of spaces used by the computer to set the left margin on the screen. To see the number of spaces that the left margin is currently set to, type the following instruction.

```
PRINT PEEK(82)
```

Have you ever noticed that there are two blank spaces on the left hand side of the screen. This is because the left margin, memory location 82, is set to 2 by the operating system, when you turn on your computer.

PEEK AND POKE
(Continued)

The POKE instruction enables you to change a value stored in memory. For example, we could use a POKE instruction to change the left margin.

```
POKE 82,5
```

To poke a value into memory, the address of memory is listed, followed by a comma and the new value to be stored in memory. Have the campers type in the POKE 82,5 instruction to see what happens. Experiment with poking the following values into memory for the left margin. Retype the entir instruction using the values listed below.

```
5, 10, 20, 39
```

After poking 39 into memory location 82, have the campers try to move the cursor to the left or to the right. Ask the campers why the cursor will not move to the left or right.

It is also possible to use a variable as the value to be poked into memory with the POKE instruction. For example, if COUNT is 1 then POKE 82,COUNT will store a 1 in memory location 82. Have the campers type in the following routine that uses a variable in a FOR . . NEXT loop to change the setting of the left margin.

```
10 REM **   POKING THE LEFT MARGIN
20 REM *
30 PRINT
40 FOR LEFTMARGIN = 0 TO 39 STEP 5
50 POKE 82,COUNT
60 PRINT
70 PRINT "LEFT MARGIN "
80 NEXT LEFTMARGIN
```

RUN the program. Then ask the campers to LIST their program. Why does everyone's code look so peculiar? Encourage the campers to experiment with changing the values in this routine and running the program.

The value for the right margin is stored in memory location 83. Have the campers type the following instruction to see what the right margin has been set to.

```
PRINT PEEK(83)
```

The right margin is routinely set to 39, the last column on the graphics zero screen. Try poking a 20 into memory location 83.

```
POKE 83,20
```

Have the campers try all sorts of values. If they get to a point where they are unable to return the margin to a reasonable setting, press SYSTEM RESET to reset the margins to their customary setting. Have the campers type in the following routine and RUN it. The program is simply a loop which continuously decrements the right margin by 5.

```
100 REM *****     DECREMENT RIGHT MARGIN
110 REM *
120 PRINT
130 FOR RIGHTMARGIN = 39 TO 0 STEP -1
140 POKE 83,RIGHTMARGIN
150 PRINT "RIGHT MARGIN";
160 FOR DELAY = 1 TO 50: NEXT DELAY
170 NEXT RIGHTMARGIN
```

Once again, when the program is completed the margins are so close together that the computer does not understand any instructions which are typed in. Press SYSTEM RESET to return the margins to normal. Then have the camper LIST the routine. Typing SYSTEM RESET does not affect the program in memory.

Now have the campers type in the following POKE instruction.

```
POKE 755,6
```

PEEK AND POKE
(Continued)

All the letters on the screen should be inverted. To return the letters to their upright position type the following POKE instruction.

```
POKE 755,2
```

For more locations to PEEK, POKE, and play with, see the following resources.

Atari Connection, Summer,1983: pp 31-32
Inside Atari BASIC: pp 132-139
Master Memory Map

All of these references are available in the camp library.

The summer issue of The Atari Connection has an article entitled "PEEKS AND POKES, Commonly Used and Helpful Memory Locations." This article has lots of fun ideas for experimenting with PEEK and POKE and an explanation for each example.

Inside Atari BASIC also suggestS various locations to PEEK and POKE for fun.

The Master Memory Map, produced by Educational Software Inc., lists all the significant memory locations, their contents, and what changes you can make. Encourage the campers to look over the memory map and experiment with changing the suggested locations. And finally, reassure the campers that it is impossible to damage or permanently alter memory. Have fun!

PEEK AND POKE
CAMPER COPY

***** PEEK AT THE LEFT MARGIN SETTING *****

```
PRINT PEEK(82)
```

***** POKE THE LEFT MARGIN WITH A NEW VALUE *****

```
5, 10, 20, 39
```

***** POKING THE LEFT MARGIN USING A VARIABLE *****

```
10 REM **   POKING THE LEFT MARGIN
20 REM *
30 PRINT
40 FOR LEFTMARGIN = 0 TO 39 STEP 5
50 POKE 82,COUNT
60 PRINT
70 PRINT "LEFT MARGIN "
80 NEXT LEFTMARGIN
```

***** PEEK AT THE RIGHT MARGIN *****

```
PRINT PEEK(83)
```

***** POKE THE RIGHT MARGIN *****

```
POKE 83,20
```

***** POKING THE RIGHT MARGIN WITH A VARIABLE *****

```
100 REM **      DECREMENT RIGHT MARGIN
110 REM *
120 PRINT
130 FOR RIGHTMARGIN = 39 TO 0 STEP -1
140 POKE 83,RIGHTMARGIN
150 PRINT "RIGHT MARGIN";
160 FOR DELAY = 1 TO 50: NEXT DELAY
170 NEXT RIGHTMARGIN
```

***** INVERTED PRINT *****

```
POKE 755,6
```

```
POKE 755,2
```

***** REFERENCES WITH MORE IDEAS FOR MEMORY LOCATIONS TO PEEK AND POKE *****

Atari Connection, Summer,1983: pp 31-32
Inside Atari BASIC: pp 132-139
Master Memory Map