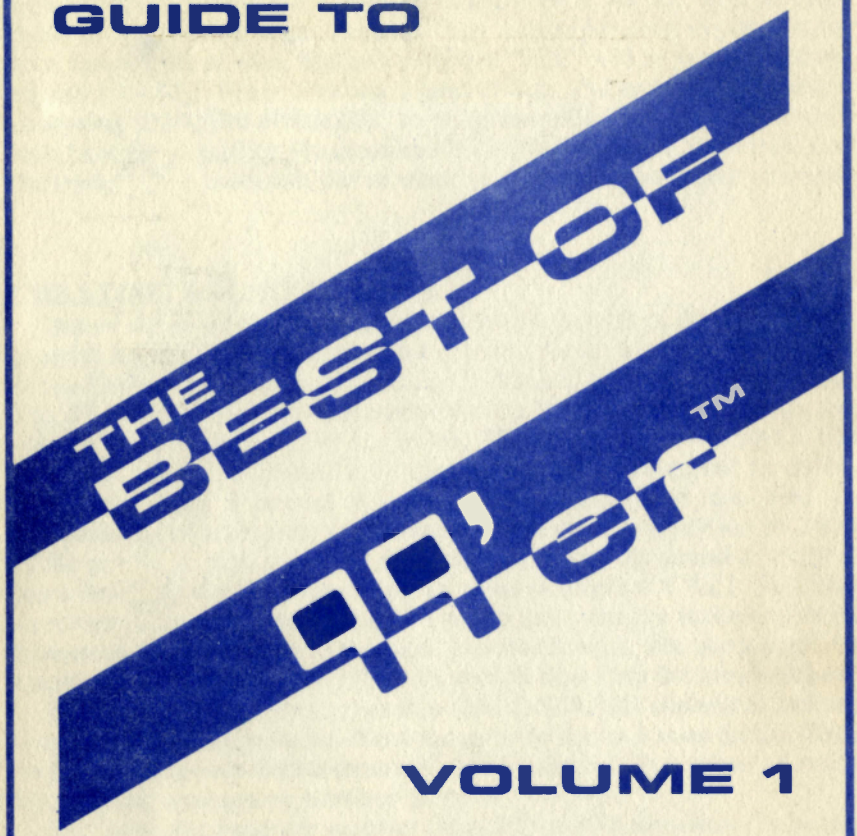
# A PROGRAMMER'S GUIDE TO




## VOLUME 1

from the Editors of
Home Computer Magazine™

# A PROGRAMMER'S GUIDE TO THE BEST OF 99'er™

From the pages of *The Best of 99'er, Volume 1,* comes this selection of programming hints, how-to's, and why's. As you read through these programming segments and their accompanying explanations, your knowledge of what makes a program "tick" will be greatly enhanced and fortified. From creating sound effects and animated graphics, to storing your files efficiently, to programming a joystick capability into LOGO, this 99'er *Programmer's Guide* will reveal the "tricks of the trade" which you can put to work in programs you create yourself.

## CREATING SOUND EFFECTS

One of the most challenging and enjoyable aspects of the home computer is its ability to create sound effects. As an example, let's look at the *Engine* program in "Livening Up Your CALL SOUNDs" (page 45). The CALL SOUND statement directs the computer to create the sounds using three types of variables. The first type of variable is the *duration*. Only one duration is specified in each statement to determine how long a sound will last. This duration is specified in thousandths of a second; so a sound of 2 seconds is specified by 2000 in the program. You also have the option of giving the sound a negative duration to make the sound stop as soon as another CALL SOUND statement is encountered. Using this procedure, you can make a smooth transition from one tone to the next. However, when you use a negative duration, you sacrifice the ability to predict how long the tone will last.

The other two variable types in a CALL SOUND statement are the *frequency* and the *volume*. You can specify up to 3 tone frequencies and one noise, each with a separate volume control. You specify a noise by entering a negative number from −1 to −8.

The following program segment (lines 760 to 800) simulates the sound made by the car as the engine turns over. The first part of this routine is a loop (lines 760 to 790):

```
750 REM ENGINE
760 FOR N=1 TO 8
770 CALL SOUND(60,220,8,-5,0)
780 CALL SOUND(60,220,8,-5,5)
790 NEXT N
800 CALL SOUND(80,220,8,-5,0)
```

Lines 760 and 790 cause the CALL SOUNDs in lines 770 and 780 to repeat 8 times. In line 770 the CALL SOUND has a duration of 60 milliseconds, (60/1000ths of a second). This statement combines a tone with a frequency of 220 cycles per second (cps) and a noise (−5). The tone has a volume of 8, and the noise a volume of 0. The volume

ranges from 0 (the loudest) to 30 (the quietest). A good way to become acquainted with the different noises is to enter them into the computer one at a time and experiment with them. Line 780 is identical to line 770 except that the volume of the noise is quieter (5 is quieter than 0). The tone created in line 800 is like the other two, but here the duration is longer, set to 80 milliseconds.

The next portion of the code simulates the car revving up after ignition. Here the loop counter variable (F) goes from 1000 to 5000 in steps of 20. The F is used in the CALL SOUND statement in line 820 to specify the third tone:

```
810 FOR F=1000 TO 5000 STEP 20
820 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
830 NEXT F
```

When you use noises −4 or −8, the frequency of the noise generated depends on the frequency of the third tone in the statement. The third tone in line 820 has a volume of 30 and is silent. The frequency of the noise (−8), which is at maximum volume (0), increases as F increases. Because the duration is a negative number (−99), the CALL SOUND does not wait the full 99/1000 second—it stops when the next CALL SOUND is encountered. This kind of frequency change sounds smoother and creates the effect of an engine revving up.

The next loop is similar to the last one—the loop counter F is used again to vary the frequency of the noise. This loop, however, counts down in steps of 50 until the loop counter F reaches 800. This makes the sound get lower in frequency to simulate the slowing down of the motor:

```
840 FOR F=4000 TO 800 STEP -50
850 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
860 NEXT F
870 END)
```

## STORING GRAPHICS IN DATA STATEMENTS

Now that we've heard a bit of the sound capabilities of the 99/4A, let's take a look at another celebrated feature of the machine—its graphics. The 99/4A was designed with graphics in mind, and you can create elaborate graphics designs in many ways. The method described in "Dynamic Manipulation Of Screen Character Graphics" (page 78) is one of the most straightforward. It uses DATA statements to store the graphics characters and their positions on the screen.

```
750 DATA POT BASE LEFT SIDE,24,6,146,POT BASE RIGHT SIDE,
    24,8,149
760 REM ----FOLIAGE----
770 DATA LO,18,1,96,LI,18,2,136,LO,17,1,96,LI,7,2,136,LO,16,1,96,LI,
    16,2,136,LO,15,2,96,LI,15,3,136
780 DATA LO,14,2,96,LI,14,3,136,LO,13,2,96,LI,13,3,136,LO,12,3,96,
    LI,12,4,136,LO,11,4,96,LI,11,5,136
```

The DATA in the statements above is defined in groups of four. The first member is simply a label and identifies the graphics being displayed (LO, for example). The next item is the *y* coordinate (18 in our example above), which indicates a row on the screen (1 = TOP, 24 = BOTTOM). The next number (1) is the *x* coordinate, or the column in which to place the graphics (1 = LEFT EDGE, 32 = RIGHT EDGE). The fourth item is the ASCII character code (96) for the character to be placed in the indicated position.

```
930 REM SCREEN LOCATION LOOP
940 HOWMANY=86
950 RESTORE 750
960 FOR CHARACTER=1 TO HOWMANY
970 READ IDENTIFICATION$,ROW,COLUMN,CHARACTER
    NUMBER
980 CALL HCHAR(ROW,COLUMN,CHARACTERNUMBER)
990 NEXT CHARACTER
```

This portion of the program READs and displays the information contained in the DATA statements. The variable HOWMANY is set to 86, indicating that there are 86 records to read. Line 950 RESTOREs the DATA pointer to line 750. The loop in lines 960 to 990 READs the DATA and displays the graphics on the screen.

## GRAPHICS WHEN MEMORY IS TIGHT

Of the many ways to place graphics on the screen, the simplest is to write out a separate CALL HCHAR or CALL VCHAR for each character you want to place on the screen. But this method will deplete available memory rapidly, and it will severely limit the graphics you can put on the screen. To avoid these pitfalls, you may want to use the alternative method found in "Preschool Block Letters and Data Compaction" (see page 165). It is similar to the method shown in "Dynamic Manipulation of Screen Character Graphics" on page 78, with a few subtle differences. In the *Xmas Tree* program, the values of the character and the coordinates were in numerical format—just as you would usually read them. This method, while easier to read and faster running, requires more memory than does the following program segment. In this example, the value of the character itself and each of the coordinates is represented by its own *single character*. For instance, to represent the *y* coordinate position 23, you would use the letter A (the letter A requires only 1 byte of memory storage area, and the number 23 requires 9 bytes). You can see that you will end up saving a great deal of memory if you use this method throughout your program. We calculated the value of 23 by taking the ASCII value for A (65), and subtracting the offset value of 42 from it. We would do the same thing for the *x* coordinate except that we would subtract 40 from the ASCII value.

```
510 RESTORE 600
520 FOR S1=0 TO 35
530 READ AR$(S1,1),AR$(S1,2),AR$(S1,0)
540 NEXT S1
600 DATA "32100000000123456777777777654"
610 DATA "00123456789:;;;;:98765432100"
620 DATA "DECJKSSSSIHBFGEDCKJVVVVHIBGF"
```

Lines 510 to 540 read the data contained in the DATA statements starting at line 600. After it is read, the DATA goes into an array, called AR$( ),which makes retrieving the information simpler.

In order to use the information in the array, the program converts the character it reads to its ASCII form and adjusts that value to get the desired number. The following code performs this task.

```
380 FOR I=1 TO LEN(AR$(S1,1))
400 COL=ASC(SEG$(AR$(S1,1)I,1))-40
410 ROW=ASC(SEG$(AR$(S1,2)I,1))-42
420 C=ASC(SEG$(AR$(S1,0),I,1))+63
430 CALL SOUND(100,300,2)
440 CALL HCHAR(ROW,COL,C)
450 FOR DELAY=1 TO 10
460 NEXT DELAY
470 NEXT I
```

Line 380 determines the length of the string contained in AR$( ) and uses it as the limit in the FOR – NEXT loop (from line 380 to 470). The loop repeats once for every character in the string. Line 400 extracts the ASCII value of one of the characters in the string and determines the *column* location by subtracting 40. The *row* coordinate is calculated in the same way (in line 410) except that the offset is 42. In line 420, the ASCII code for the graphics character is extracted with an offset of 63.

In line 600, the first character of AR$(S1,1) is 3, and its ASCII value is 51. Because 51 – 40 = 11, the column or the *x* coordinate is 11. In line 610, the first character in AR$(S1,2) is 0 (ASCII value 48), making the row or the *y* coordinate 6 (48 – 42 = 6). In line 620, the character number is extracted from the first character in AR$(S1,0). The ASCII value of the character (D) is 68, and 68 + 63 = 131. Character number 131 is then placed on the screen at column 11, row 6. This process repeats until all the members of the strings have been converted into values that can be used by the HCHAR statement in line 440.

## HIGH-RESOLUTION BAR GRAPHICS

The home computer's ability to create custom graphics characters lends itself to business applications too. In the article "Dynamic Manipulation Of Screen Character Graphics" are several examples of bar graph display techniques. The *Bar Topper* program shows you how you can increase the vertical resolution of a graph.

First, the height of the bar is calculated and placed on the screen in lines 680 – 730:

```
680 BARHEIGHT = HORSEPOWER/SCALE
690 Y = INT(BARHEIGHT)
700 REMAINDER = BARHEIGHT - INT(BARHEIGHT)
710 CALL VCHAR(22 - ,16,96,Y)
720 CALL VCHAR(22 - Y,17,96,Y)
730 CALL VCHAR(22 - Y,18,96,Y)
```

Then the new graphics character is created in lines 750 through 770 (below). The remainder of the bar height (the portion where less than a whole character is needed) is calculated in line 750, and MASTER$ is created in line 760. Line 770 calculates the STARTPOSITION in MASTER$ necessary to make the appropriate character. Line 780 then extracts 16 characters from MASTER$ using the SEG$ command with STARTPOSITION as an index. TOPPATTERN$ is set equal to the character needed to complete the graph. Line 790 assigns the shape stored in TOPPATTERN$ to ASCII character 97. Finally, line 800 completes the high resolution bar by placing the new character on the screen:

```
750 TOPPATTERN = 1 + INT((REMAINDER*8) + .5)
760 MASTER$ = "00000000000000FFFFFFFFFFFFFFFF"
770 STARTPOSITION = 2*TOPPATTERN - 1
780 TOPPATTERN$ = SEG$(MASTER$,STARTPOSITION,16)
790 CALL CHAR(97,TOPPATTERN$)
800 CALL HCHAR(21 - Y,16,97,3)
```

## THE DATA STOREHOUSE WITHIN A PROGRAM

Most people don't usually think of storing DATA *within* a program, but that is what DATA statements have been designed to do. They are ideal for storage applications that either (1) don't require a large volume of data, or (2) don't need to be constantly updated. DATA statements provide fast information retrieval with no need for an external storage medium other than the one used for storing the program itself. Using DATA statements does, unfortunately, limit the amount of information to the size of the program and thus, to the size of the computer's memory. Also, if you need to change the data, you must halt the program, enter Edit mode, and change the information in the DATA statements. Thus, you cannot change the information while the program is running.

An excellent application of the use of DATA statements to store information that won't undergo lots of changes is the houshold inventory program in "NOW WHAT?" (page 16). The information for the inventory is kept in DATA statements like this:

```
460 DATA 3,COMPUTER,1000
470 DATA 3,DESK,129.50
480 DATA 2,"MICRO. OVEN",450
```

Continued

```
490 DATA 1,PIANO,5900
500 DATA 3,PRINTER,225.50
.
.
.
570 DATA 9,ZZZ,9
```

The last line above is a *flag* to the computer. Whenever 9,ZZZ,9 appears in the program, it signals the end of the DATA for that program segment. The progam segment above shows that there are three pieces of information for each item in the house: first comes the number of the room containing the item, second, the name of the item, and finally, its value. Such a record could prove invaluable for insurance purposes. Each piece of information in DATA statements must be separated from the others by a comma. Therefore, it is essential that you do not use commas as part of the information that you are storing—the computer will interpret them as DATA separators.

Once the information is entered into the DATA statements, your program must READ it and manipulate it so that the computer can display it on the screen or a printer. This is much simpler than it sounds. To make sure the computer is READing the right DATA at the right time, you use a RESTORE statement. RESTORE simply tells the computer the line number of the first DATA statement to be READ. It is good programming practice to follow the RESTORE with the READ statement that retrieves the information. For example:

```
300 RESTORE 460
310 READ ROOM,ITEM$,COST
```

Once the computer READs the information, it uses it in various ways. For example, line 320 (below) checks for the end of DATA and branches to line 380 when the end is reached. Remember that the number 9 is used to signal the end of the DATA:

```
320 IF ROOM=9 THEN 380
```

In the following program segment, the variable CH represents the room the user has chosen to inventory. If CH is equal to 4, then the whole house has been inventoried; otherwise line 340 checks to see if the information read from the DATA pertains to the room requested. If not, the computer branches back to get another record in line 310. If the item is in the specified room, the computer continues to line 350, where the the information on the item is printed.

```
330 IF CH=4 THEN 350
340 IF ROOM<>CH THEN 310
350 PRINT ITEM$,"$";COST
```

Next, a running total of the value of everything inventoried is stored in the variable TOTAL. Then the computer branches back to line 310 to get the next record:

```
360 TOTAL=TOTAL+COST
370 GOTO 310
```

## THE BUBBLE EFFECT IN SORTING

Often in programs you have a list of data that needs to be put in order. This may be a list of numbers or a list of names that needs alphabetizing. Your TI-99/4A is made to order for sorting lists of data, and many *sort routines* have been developed. There is no practical difference between lists of numbers and lists of names because ASCII codes for letters are in numeric order, so a sort routine for numbers will need only minor alterations to work on character strings.

One of the routines in the *Homeword Helper: Fractions* program on page 168, takes up to 10 fractions as input and sorts them from smallest to largest. The sort used here is the bubble sort. Of all the sort routines used by programmers, it is one of the simplest to understand, and it is implemented with just a few lines of code as well.

In this program up to 10 fractions are entered from the computer into two arrays: NNN( ) for numerator and DDD( ) for denominator. Then the decimal values of the fractions are computed and entered into an array called FRC( ). Here's the code that receives the input of the fractions and computes the values:

```
5160 INPUT "HOW MANY FRACTIONS?" NF
5170 IF NF <11 THEN 5200
5180 PRINT "SORRY; UP TO 10 ONLY."
5190 GOTO 5160
5200 FOR I=1 TO NF
5210 PRINT : : "FRACTION ";I
5220 INPUT " NUMERATOR :":NNN(I)
5230 INPUT " DENOMINATOR:":DDD(I)
5240 FRC (I)=NNN(I)/DDD(I)
5250 FRD(I)=FRC(I)
5260 NEXT I
```

The sort begins by clearing a flag called SW (for SWitch) to zero, and initiating a FOR-NEXT loop. The loop is executed one time less than the number of fractions (NF – 1) entered.

```
5280 FOR I=1 TO NF-1
5300 IF FRC (I)< =FRC(I+1) THEN 5350
5310 FF=FRC(I)
5320 FRC(I)=FRC(I+1)
5330 FRC(I+1)=FF
5340 SW=1
5350 NEXT I
5360 IF SW=1 THEN 5280
```

Every time through the loop, each member of the array is compared to the succeeding member. If the first member is smaller than the adjacent member, then no switch is needed (the two members are already in the desired order), and the computer jumps to the NEXT in line 5350. However, if the first member is the larger number of the two,

then the computer goes on to lines 5310 through 5340, where the switch occurs. First FRC(I) is placed in the temporary storage variable FF, then FRC(I + 1) is moved is moved into FRC(I), and finally FF is loaded back into FRC(I + 1). In this way the smaller numbers "bubble up" to the top of the array. In line 5340 the SW flag is set to one. Line 5260 causes the entire FOR-NEXT loop to be executed again whenever the SW flag has been set. If, for example, the first fraction equaled .5, the second equaled .6 and the third equaled .4, then the first time the loop was executed .4 and .6 would be exchanged. But the loop would have to be executed again to switch the .4 and the .5 values. Only when the entire loop executes without any switches can we be sure that the fractions are in order.

When the list of fractions is in order, then no switches are needed; this time when the loop is executed, SW equals 0. At this point the computer continues with the program, and the sorted list of fractions is printed.

## CASSETTE FILE HANDLING

So far we have talked about manipulating data in programs and storing data in programs. Now we come to the problem of storing large amounts of data so that it can be easily brought into memory and manipulated. The TI-99/4A accesses a cassette-based data storage system that is inexpensive, reliable, and easy to use.

Cassette data files are stored in a *sequential* format. This means that information is written on the tape one record after another. When the computer reads the information back into its memory it will be in the same order in which it was written. (While diskette files can be sequential, they also have a random-access capability not possible with cassette storage.) The following program comes from "A Beginner's Guide To Cassette Operation With a Home Computer" (page 20) and illustrates how easily information can be stored and retrieved using cassettes.

```
410 REM OPEN FILE FOR OUTPUT
420 OPEN #1:"CS1",OUTPUT,INTERNAL,SEQUENTIAL,FIXED 192
430 FOR I=1 TO 60 STEP 3
440 REM PRINT THREE SETS OF DATA PER RECORD
450 PRINT #1:B_NAME$(I);B_AVG(I);B_HANDI(I);
460 PRINT #1:B_NAME$(I+1);B_AVG(I+1);B_HANDI(I+1);
470 PRINT #1:B_NAME$(I+2);B_AVG(I+2);B_HANDI(I+2)
480 NEXT I
490 CLOSE #1
```

Line 420 OPENs the file that is going to send data out to the cassette recorder. The #1 is an ID number the computer will use in referencing any data to be sent to this cassette file. The computer knows it is a cassette file because CS1 is named as the file device. Similarly, diskettes are designated with DSK1. The rest of line 420 tells the computer that file #1 is to be an OUTPUT file, using an INTERNAL (binary) and

SEQUENTIAL format for storing the data, and that each record will have a FIXED length of 192 bytes.

Lines 430-480 form a *loop* that writes to the file. The variable I is incremented by 3 each time it passes through the loop because of the STEP 3 in line 430. Line 450 through 470 then PRINTs the data to file #1 from arrays, using the loop counter I to index the array elements in order. Each record of the file includes each of the three file variables [B_NAME$(N), B_AVE(N), B_HAND(N)] three times; a total of 9 different quantities is written to each record in the file. Line 480 is the end of the output loop. Every time NEXT I is encountered, the variable I is checked to see if it has reached 60. If not, then control passes to line 440. If I reaches 60, then the program continues to line 490, where the file is closed. The file should always be closed before leaving the program; otherwise, erroneous information can be written to the file, or data can be lost.

The following program reads information back into the computer from the cassette tape file created with the last program.

```
190 REM OPEN FILE FOR INPUT
200 OPEN #1:"CS1",INPUT,INTERNAL,SEQUENTIAL,FIXED 192
210 FOR I=1 TO 60 STEP 3
220 REM READ THREE RECORDS FROM THE TAPE
230 INPUT #1:B_NAME$(I),B_AVG(I),B_HANDI(I),
240 INPUT #1:B_NAME$(I+1),B_AVG(I+1),B_HANDI(I+1),
250 INPUT #1:B_NAME$(I+2),B_AVG(I+2),B_HANDI(I+2)
260 NEXT I
270 CLOSE #1
```

This program is similar to the output program. The difference is that this program is reading information into the computer's memory (INPUT), whereas the first program was writing information to the tape (OUTPUT). In line 200 the OPEN statement includes the word INPUT in the place where OUTPUT used to be. In place of the PRINT in lines 230 through 260 you will now find INPUT. The only other difference is that the variables are now separated by a comma; in the first program they were separated by a semicolon.

## CREATING MOTION WITH SCROLLING

One of the most ingenious ways to create the illusion of motion on the TI-99/4A involves moving the entire screen image. Then the stationary object on the screen appears to be moving. Let's look at *San Francisco Tourist* (page 260), a good example of this method.

Line 170 defines a new function for the variable R. Whenever R is used in the program, the equation in line 170 makes R either +1 or −1. This random number alters what's on the screen as it scrolls.

```
170 DEF R=(-1) ^ (INT(4*RND))*(INT(4*RND))
```

The coding in lines 750-860 prints a road six characters wide. On either side a border character randomly moves from side to side by −1 or +1 character positions. Line 750, the RANDOMIZE statement,

places a random seed in the random number generator. Without this command, the random number generator would select the same series of random numbers each time the program runs. The variable J initialized to a value of 18 in line 760 keeps track of the horizontal position of the road as it is printed.

```
750 RANDOMIZE
760 J=18
```

The statements from line 790 to 1070 make up the control loop to display the scrolling road and then to accept and respond to the player's move. Let's look closely at lines 790-860. Line 790 specifies that this loop should repeat 75 times. Lines 800 through 850 check the value of J to see if it gets too close to the edge of the screen. If it does, the value is adjusted to keep it on the screen. All the action is at line 860: Each time the statement is executed, it scrolls the screen. TAB(J); in the line causes whatever is printed next on the screen to be *tabbed* over J character positions. The "())))))(" section is the road and border to be printed. The character patterns are altered by the program so they look like a road on the screen and not parentheses.

```
790 FOR I=1 TO 75
800 IF I>59 THEN 860
810 J=J+R
820 IF J<21 THEN 840
830 J=21
840 IF J>1 THEN 860
850 J=1
860 PRINT TAB(J);"())))))("
```

## BASIC ANIMATION

With computer graphics you can create effects that take cartoon artists months to draw—you can create animation. Although home computers haven't quite reached the level of quality that we expect from our Saturday morning cartoons, they have made significant progress. *County Fair Derby* (page 263) illustrates one example of animation. This program will also prove to non-believers that you do not need Extended BASIC to create animation effects.

In the following program segment the patterns for the characters are contained in an array called H$( ). The use of an array allows you to index or "flip through" the graphics characters by indicating which cell of the array is to be used.

```
480 H$(1)="0000000004020100F"
490 H$(2)="000008080F1F30F0"
500 H$(3)="0F0F102040000000"
 .
 .
 .
```

The following lines use the array to produce the animated horses:

```
3150 X=S+8
3160 FOR Y=1 to 4
3170 CALL CHAR((95+Y),H$(X))
```

Continued

```
3180 X=X+5
3190 NEXT Y
```

In line 3150 the index pointer, X, is given an initial value. The value of X determines which pattern is used from the H$( ) array. The loop from lines 3160 to 3190 flips through the patterns, assigning each horse the next pattern in sequence (using the loop counter variable Y). This creates the effect of animation. Line 3170 assigns the pattern from the array H$( ) to the character indicated by 95+Y.

## FULL SCREEN DISPLAYS WITH EXTENDED BASIC

The animation in *County Fair Derby* can be achieved in BASIC, but Extended BASIC expands your graphics capabilities. If you use the method we illustrate here, you'll spend less time and consume less memory in your full-screen graphics programs. It works by placing the characters in DATA statements, READing them, and then either PRINTing or DISPLAYing them on the screen a whole line at a time instead of a single character at a time. In *Interplanetary Rescue* (page 272) the DISPLAY AT command available in Extended Basic DISPLAYs the alien topographical map over most of the screen. The benefit here is that it took only a few seconds and used very little memory. The following code shows you how this is done:

```
470 DATA "^^!!!^^__ __ __ __ __ __ *__ ^^^!!!^^ __ __ "
480 DATA "^!!!!^^__ __ *__ ***__ __ __ __ ^!!!^^__ "
490 DATA "^!!!!!^__ ***__ __ __ __ **__ ^^^!!^__ "
500 DATA "^!!>!!^^__ ****__ ****__ ^^!!^^"
510 DATA "!!!!!!^__ __ __ *__ __ **__ __ __ ^^^!^^^"
 .
 .
 .
1660 IF OPT2=1 THEN RESTORE 1730 :: GOTO 1700
1670 IF OPT2=2 THEN RESTORE 1940 :: GOTO 1700
1680 IF OPT2=3 THEN RESTORE 470 :: GOTO 1700
1690 IF OPT2=4 THEN RESTORE 2150
1700 CALL CLEAR :: CALL COLOR(8,10,12)
1710 FOR TER=1 TO 21 :: READ TERN$ :: DISPLAY
  AT(TER,1):TERN$ :: NEXT TER
```

In lines 470 through 510 you can see one example of how DATA statements can be formatted in a screen image display. Each of the characters is redefined to different colored blocks. The DATA shown here is 26 characters wide, leaving the right two character columns for another display area. The two edge characters on either side are not used because of "bleed over" on TV sets. Lines 1660 through 1710 select the proper DATA, and DISPLAY one of four screen displays depending on the value in the variable OPT2. OPT2 is the second option selection in the game and selects the level of difficulty. The higher the level, the more complex the topographical map becomes. Lines 1660 through 1690 make this selection by testing OPT2 and restoring the appropriate block of DATA statements. Line 1700 CLEARs the screen and sets the color of character group 8 to light red on light yellow.

Line 1710 will display the DATA by setting up a loop that repeats 21 times where TER is the loop control variable. This loop causes 21 lines of DATA to be displayed on the screen. The DATA is first READ into the string variable TERN$, and then displayed using the DISPLAY AT command, with the variable TER indicating the screen row for the line of DATA.

## SPRITES IN EXTENDED BASIC

One of the main advantages of Extended BASIC over TI BASIC is that it gives you the ability to move up to 28 sprites on the screen simultaneously. Sprites given an X and Y velocity continue moving in the direction specified until changed by another sprite-controlling command. The short program *Sprite Chase* (page 267) is a good example of sprite capabilities.

```
320 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
330 CALL SPRITE(#28,96,2,90,120,0,0)
340 FOR A=1 TO TARGS
350 CALL SPRITE(#A,A+CH,2,90,120,INT(RND*50-25),
  INT(RND*50-25))
360 NEXT A
```

The routine above initializes the sprite that the player controls and the targets that the player must catch. Line 320 defines ASCII character 96 as a square block. Line 330 then designates the player's sprite as number 28, gives it this pattern, and sets its color to black. The command also specifies the sprite's location as vertical dot row 90 and horizontal dot column 120 (approximately the center of the screen). The speed for this sprite is set to 0 on both the *x* and *y* axes.

Lines 340 to 360 create a loop called A that will create a number of sprites, giving them all a different random motion. A choice is made earlier in the program, setting CH to 47 or 64 and TARGS to 10 or 26 so that either numbers or letters are displayed.

```
550 CALL COINC(#28,#A,9,HIT)
560 IF HIT=-1 THEN 620
```

Lines 550 and 560 see whether there is a coincidence between the player's sprite and the sprite number corresponding to the value of A. Line 550 will check to see if sprite #28 and sprite #A are within 9 pixels of each other. If they are, then the variable HIT is set to $-1$. If there is no coincidence, then HIT is set to zero. In Line 560, if HIT is set to $-1$ (the two sprites were within 9 pixels of each other), then the computer branches to line 620; otherwise it continues on to line 570.

## SPRITE CONTROL

In addition to being able to just place flying sprites on the screen randomly, you can also control the sprite for excellent visual effects. A routine in *Dog Fight* (page 269) lets you control the flight of a WWI biplane. You can do loops and all sorts of aerobatics while trying to shoot down your opponent. The following code shows how this is done with the keyboard. With a little ingenuity you could easily modify the program to accept the joystick as input.

```
500 CALL KEY(1,K1,S1)::CALL KEY(K2,S2)
 .
 .
530 IF K1<>5 THEN 570 ELSE IF P>3 AND P<7 THEN P=P-1 ELSE
    IF P<3 OR P=8 THEN P=P+1
540 IF P=0 THEN P=8
550 IF P=9 THEN P=1
560 GOTO 650
```

This routine reads the keyboard, checks the current direction of the plane, and changes the direction if the keyboard input corresponds to a possible move. In this program, the directions in which you can point the plane are limited by your current position. If you press S, then the plane rotates until it eventually points to the left. But if it was originally pointing directly to the right, pressing S has no effect. You must point the plane either up or down before pressing S to get the plane to go left. The plane can't do 180 degree turns with just one key press. The variable P is used to keep track of the plane's direction, from 1 to 8. 1 is to the right, 3 is straight up, 5 is left, and so on.

Line 650 changes the plane's pattern to point in the appropriate direction, then causes the computer to branch to a subroutine that sets the plane's speed.

```
650 CALL PATTERN(#1,(P*4)+92)::ON P GOSUB 910,920,930,940,950,
    960,970,980::GOTO 760
```

These subroutines go from line 910 to line 980. The variable V is determined by the level of difficulty in the game. The higher the level of difficulty, the higher the value given to V and the faster the speed of the plane.

```
910 CALL MOTION(#1,0,V)::RETURN
920 CALL MOTION(#1,-V*.6,V*.6)::RETURN
930 CALL MOTION(#1,-V,0)::RETURN
 .
 .
```

## COINCIDENCE VS. SPEED

Once a sprite is put in motion, it will continue moving until the program changes its motion. Still the program must check to see where the sprites are if you wish to have them interact. This presents a problem if the sprites are moving quickly or there are a lot of sprites to check because a fast moving sprite can pass over an area before your program can check it. In the program *Battle Star* (page 234), this problem is eliminated by limiting the moving sprites to either horizontal or vertical motion along predetermined lines. This means that only one axis needs to be checked for a sprite. By setting a flag variable when the sprite is created, the program merely tests the variable to see whether or not a sprite is on the given line. This is much faster than using CALL POSITION to first locate the sprite and then check to see if it's on the right course.

Lines 700 through 800 randomly select which enemy ships appear and whether or not they fire at you before the keyboard is scanned for your first response. As the characters and sprites are placed on the screen, flags are set. For example, let's say a ship appears from

the top. Then the first flag (SA1) is set to 1. If the ship fires a missile, then a second flag (SB1) is set.

The following lines monitor your response:

```
390 CALL KEY(0,K,S) :: IF S=0 THEN RETURN
400 IF K=69 THEN 450
```

Line 390 scans the keyboard, and if no key has been pressed, the program returns to line 320. Line 400 causes a branch to a subroutine to fire up if the letter E was pressed. Lines 450 to 490 fire your laser, check to see if there is a target, and adjust the laser accordingly.

```
450 IF SA1=0 AND SB1=0 THEN CALL VCHAR(1,16,105,10) :: CALL
  SOUND(10,800,0) :: CALL VCHAR(1,16,32,10) :: SC=SC-10 ::
  RETURN
460 IF SB1=0 THEN CALL VCHAR(2,16,105,9) :: CALL SOUND(500,
  110,2,-5,2) :: CALL VCHAR(2,16,32,9) :: SC=SC+50 :: SA1=0 ::
  RETURN
470 CALL POSITION(#1,P1,P2) :: IF P1>76 THEN 840
480 P1=INT(P1/8)+1 :: CALL VCHAR(P1,16,105,10-P1) :: CALL
  SOUND(200,110,10,-5,8) :: CALL VCHAR(P1,16,32,10-P1)
490 CALL DELSPRITE(#1) :: SC=SC+20 :: SB1=0 :: RETURN
```

In line 450 both SA1 and SB1 are checked for the existence of either the ship or the missile. If neither of them is on the screen, then the laser is fired all the way to the edge of the screen, and your score is reduced by 10 for the miss. The program then returns control to the main loop.

If either of the two flags is set to 1, then the computer will continue on to line 460. Here the flag SB1 is tested to see if a missile has been fired yet. If not, then the position of the space ship is known because it is stationary. The rest of the line fires the laser, adds 50 points to the score, resets the SA1 flag to zero, and deletes the ship from the screen.

If the missile had been fired (SB1 = 1), the computer would branch to line 470, where the position of the missile is determined, to see if it has reached your Battle Star yet. You could use a CALL COINC; however, this would not only slow down the program, but a sprite would occasionally slip through a coincidence area before the computer could detect it. Instead, the program simply checks to see if the sprite has passed a certain point (the edge of the Battle Star); if a hit was made by the missile, then the computer branches to line 840. If the missile has not reached the Battle Star, then line 480 causes the laser to fire. Line 490 deletes the missile sprite from the screen, resets its flag, and adds 20 points to the player's score. The program then returns the computer to the main control loop.

## DETECTING CHARACTERS BENEATH SPRITES

When using sprites you often want to know what character lies beneath the sprite as it crosses the screen. In *Interplanetary Rescue* (page 272), a short routine provides this information:

```
390 CALL POSITION(#1,XC,YC) :: CALL CHAR (ABS(XC+4)/8+.5),
  ABS((YC+4)/8+.5),CC )
```

This line of code returns the character's ASCII code to the variable CC that lies directly under the center of the sprite. First, the CALL POSITION command returns the dot-row and dot-column of the upper-left-corner of the sprite; so this line compensates for that movement by offsetting the position to the very center ((XC + 4)/8 + .5). By using this offset, the GCHAR statement in line 390 returns the character position based on the dot positions plus four pixels (half a character).

## PROGRAMMING FOR JOYSTICK OR KEYBOARD

Almost every arcade-type computer game either requires joysticks or is much easier to play with them. Not everyone who owns a home computer, however, has purchased a pair of joysticks. So all game programs should be written so that the option of using the keyboard in place of them is available. The four arrow keys (E,S,D,and X) on the left hand side of the keyboard are the keys most commonly used to simulate the joystick. If you wish to use the four corner positions for diagonal movement as well, you can include the W,R,Z, and C keys. The following code is from *N-Vader* on page 276.

The program lets the player choose whether to use the keyboard or joysticks in line 700:

```
700 INPUT "JOYSTICKS (Y/N)? ":X$
710 IF SEG$(X$,1,1)="Y" THEN JS=1
```

The program sets a flag (JS) in line 710 so that the computer knows which method of input you wish to use. It can then branch to the appropriate subprogram. If the program does not ask the player to choose, then the computer must scan both the joystick and the keyboard for input. This slows the game down considerably. Because in this program the computer knows which input device is being used (keyboard or joystick), it executes the proper subprogram in line 1080.

```
1080 IF JS=1 THEN CALL JOYST(1,JX,JY)ELSE CALL KEYST(1,JX,JY)
```

Notice that CALL KEYST is not a standard subprogram name. In Extended BASIC it is possible to CALL true user-defined subprogram and pass parameters between them and the main program. In this case, the CALL KEYST routine scans the keyboard and returns values that simulate those that would be returned by a joystick.

```
1390 SUB KEYST(N,X,Y)
1400 CALL KEY(N,K,S)
1410 IF S=0 THEN X,Y=0::SUBEXIT
1420 IF K=2 THEN X=-4::Y=0
1430 IF K=4 THEN X=-4 :: Y=4
1440 IF K=5 THEN X=0 :: Y=4
1450 IF K=6 THEN X=4 :: Y=4
1470 IF K=3 THEN X=4 :: Y=0
1480 IF K=14 THEN X=4 :: Y=-4
1490 IF K=0 THEN X=0 :: Y=-4
1500 IF K=15 THEN X=-4 :: Y=-4
1510 SUBEND
```

Line 1390 is the entry point to the subprogram and sets the parameters to be passed. Because this is truly a subprogram, the variables used here will not affect variables of the same name in the rest of the program. The first parameter passed is N. This variable sets the keyboard mode. It is set to 1 to scan the left half of the keyboard and set to 2 for the right half; it is used in the standard CALL KEY statement in Line 1400. The next two variables, X and Y, are set to the same values returned by the joystick if the corresponding position is selected. The value returned here will not be that character's ASCII value. To find the values of the keys in the split-keyboard mode, consult the Appendix in the *User's Reference Guide*. The third variable, S, is used to detect the keyboard status. Line 1410 checks this status, and if S is equal to zero (no key pressed), then the computer exits the subprogram, returning to the statement immediately after the calling statement. Lines 1420 through 1500 check the value of the pressed key in the variable K and set the return variables to the proper values to simulate the joystick. Line 1510 returns control back to the main program following the statement that called this subprogram.

## JOYSTICK USE WITH LOGO

Many LOGO users don't realize that they have the capability of using joysticks with LOGO procedures. This is accomplished with the undocumented JOY command. The LOGO JOY command can be useful in a number of applications. The following subroutine is part of "Fly Away With The Joy Commands Of TI LOGO" (page 121):

```
TO STICK :S
MAKE "X JOY :S
TELL :S
IF :X < 4 THEN TURNLEFT
IF :X > 6 THEN TURNRIGHT
IF :X = 6 THEN FASTER
IF :X = 4 THEN SLOWER
END
```

This procedure reads the joystick and calls the appropriate routine, depending on which direction the joystick was moved. The prodedure SETUP prompts for the choice of keyboard or joystick, and puts one choice in the variable :MODE. Then the procedure FLYAWAY chooses either CONTROL or CONTROLJOY, depending on the value of :MODE. The line MAKE "X JOY :S in the procedure STICK reads the joystick, and a number from 0 to 10 is placed in the variable :X. This value is compared to values in the succeeding lines to determine in which direction the joystick is pointed. Based upon these IF – THEN statements, the proper routine is called.


- *Scanning and Reworking by:*
*TI99 Italian User Club in the year 2015.*
*(info@ti99iuc.it)*

AN
EMERALD VALLEY
BOOK



**IMPORTANT NOTICE REGARDING BOOK MATERIALS**

Emerald Valley Publishing Co. makes no warranty, either expressed or implied, including but not limited to any implied warranties of merchantability and fitness for particular purpose, regarding these book materials and makes such materials available solely on an "as-is" basis.

In no event shall Emerald Valley Publishing Co. be liable to anyone for special, collateral, incidental, or consequential damages in connections with or arising out of the purchase or use of these book materials, and the sole and exclusive liability to Emerald Valley Publishing Co. regardless of the form of actions, shall not exceed the purchase price of this book. Moreover, Emerald Valley Publishing Co. shall not be liable for any claim of any kind whatsoever against the user of these book materials by any other party.

Emerald Valley Publishing Co.
P.O. Box 5537 ● Eugene, Oregon 97405 ● Tel. (503) 485-8796