

CHAPTER 4

ADVANCED BASIC

- Introduction
- Simple Animation
 - Nested Loops
- INPUT
- GET
- Random Numbers and Other Functions
- Guessing Game
- Your Roll
- Random Graphics
 - CHR\$ and ASC Functions

INTRODUCTION

The next few chapters have been written for people who have become relatively familiar with the BASIC programming language and the concepts necessary to write more advanced programs.

For those of you who are just starting to learn how to program, you may find some of the information a bit too technical to understand completely. But take heart. . . because for these two fun chapters, **SPRITE GRAPHICS** and **CREATING SOUND**, we've set up some simple examples that are written for the new user. The examples will give you a good idea of how to use the sophisticated sound and graphics capabilities available on your **COMMODORE 64**.

If you decide that you want to learn more about writing programs in BASIC, we've put a bibliography (Appendix N) in the back of this manual.

If you are already familiar with BASIC programming, these chapters will help you get started with advanced BASIC programming techniques. More detailed information can be found in the **COMMODORE 64 PROGRAMMER'S REFERENCE MANUAL**, available through your local Commodore dealer.

SIMPLE ANIMATION

Let's exercise some of the Commodore 64's graphic capabilities by putting together what we've seen so far, together with a few new concepts. If you're ambitious, type in the following program and see what happens. You will notice that within the print statements we can also include cursor controls and screen commands. When you see something like {CRSR LEFT} in a program listing, hold the **SHIFT** key and hit the CRSR LEFT/ RIGHT key. The screen will show the graphic representation of a cursor left (two vertical reversed bars). In the same way, pressing **SHIFT** and **CLR/HOME** shows as a reversed heart.

NEW

```
10 REM BOUNCING BALL
20 PRINT "{CLR/HOME}"
25 FOR X = 1 TO 10 : PRINT "{CRSR/DOWN}": NEXT
30 FOR BL = 1 TO 40
40 PRINT"!●{CRSR LEFT}";:REM (● is a SHIFT-Q)
50 FOR TM = 1 TO 5
60 NEXT TM
70 NEXT BL
75 REM MOVE BALL RIGHT TO LEFT
80 FOR BL = 40 TO 1 STEP -1
90 PRINT"!{CRSR LEFT}{CRSR LEFT}●{CRSR LEFT}";
100 FOR TM = 1 TO 5
110 NEXT TM
120 NEXT BL
130 GOTO 20
```

: INDICATES NEW
COMMAND

THESE SPACES
ARE INTENTIONAL

TIP:

All words in this text will be completed on one line. However, as long as you don't hit **RETURN** your 64 will automatically move to the next line even in the middle of a word.

The program will display a bouncing ball moving from left to right, and back again, across the screen.

If we look at the program closely, (shown on page 44) you can see how this feat was accomplished.

Line 10 is a REMark that just tells what the program does; it has no

```

10  REM BOUNCING BALL
20  PRINT "{CLR/HOME}"
25  FOR X = 1 TO 10 : PRINT "{CRSR/DOWN}": NEXT
30  FOR BL = 1 TO 40
40  PRINT " ●{CRSR LEFT}"; REM (● is a SHIFT-Q)
50  FOR TM = 1 TO 5
60  NEXT TM
70  NEXT BL
75  REM MOVE BALL RIGHT TO LEFT
80  FOR BL = 40 TO 1 STEP -1
90  PRINT " {CRSR LEFT}{CRSR LEFT}●{CRSR LEFT}";
100 FOR TM = 1 TO 5
110 NEXT TM
120 NEXT BL
130 GOTO 20

```

effect on the program itself. Line 20 clears the screen of any information.

Line 25 PRINTs 10 cursor-down commands. This just positions the ball in the middle of the screen. If line 25 was eliminated the ball would move across the top line of the screen.

Line 30 sets up a loop for moving the ball the 40 columns from the left to right.

Line 40 does a lot of work. It first prints a space to erase the previous ball positions, then it prints the ball, and finally it performs a cursor-left to get everything ready to erase the current ball position again.

The loop set up in lines 50 and 60 slows the ball down a bit by delaying the program. Without it, the ball would move too fast to see.

Line 70 completes the loop that prints balls on the screen, set up in line 30. Each time the loop is executed, the ball moves another space to the right. As you notice from the illustration, we have set up a loop within a loop.

This is perfectly acceptable. The only time you get in trouble is when the loops cross over each other. It's helpful in writing programs to check yourself as illustrated here to make sure the logic of a loop is correct.

To see what would happen if you cross a loop, reverse the statements in lines 60 and 70. You will get an error because the computer gets confused and cannot figure out what's going on.

Lines 80 through 120 just reverse the steps in the first part of the program, and move the ball from right to left. Line 90 is slightly different from line 40 because the ball is moving in the opposite direction (we have to erase the ball to the right and move to the left).

And when that's all done the program goes back to line 20 to start the whole process over again. Pretty neat! To stop the program hold down **RESTORE** and hit **RUN/STOP**.

For a variation on the program, edit line 40 to read:

```
40 PRINT "Q";
```

← TO MAKE THE Q, HOLD THE SHIFT KEY DOWN AND HIT THE LETTER "Q."

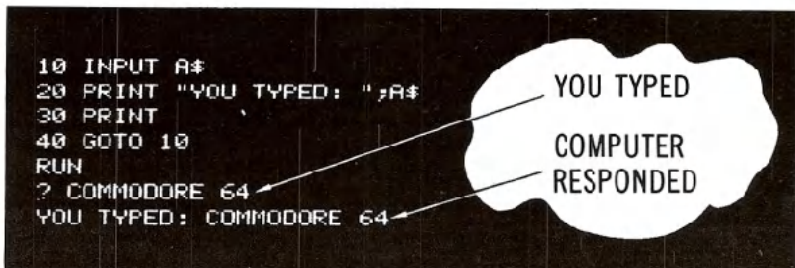
Run the program and see what happens now. Because we left out the cursor control, each ball remains on the screen until erased by the ball moving right to left in the second part of the program.

INPUT

Up to now, everything within a program has been set before it is run. Once the program was started, nothing could be changed. INPUT allows us to pass new information to a program as it is running and have that new information acted upon.

To get an idea of how INPUT works, type **NEW** **RETURN** and enter this short program:

```
10 INPUT A$
20 PRINT "YOU TYPED: ";A$
30 PRINT
40 GOTO 10
RUN
? COMMODORE 64
YOU TYPED: COMMODORE 64
```



What happens when you run this program is simple. A question mark will appear, indicating that the computer is waiting for you to type something. Enter any character, or group of characters, from the keyboard and hit **RETURN**. The computer will then respond with "YOU TYPED:" followed by the information you entered.

This may seem very elementary, but imagine what you can have the computer do with any information you enter.

You can INPUT either numeric or string variables, and even have the INPUT statement prompt the user with a message. The format of INPUT is:

```
INPUT "PROMPT MESSAGE";VARIABLE
```

← PROMPT MUST BE 38 CHARACTERS OR LESS.

Or, just:

INPUT VARIABLE

NOTE: To get out of this program hold down the **RUN/STOP** and **RESTORE** keys.

The following program is not only useful, but demonstrates a lot of what has been presented so far, including the new input statement.

NEW

```
1 REM TEMPERATURE CONVERSION PROGRAM
5 PRINT "{CLR./HOME}"
10 PRINT "CONVERT FROM FAHRENHEIT OR CELSIUS
    (F/C)": INPUT A#
20 IF A# = "" THEN 20
30 IF A# = "F" THEN 100
40 IF A# <> "C" THEN 10
50 INPUT "ENTER DEGREES CELSIUS: "; C
60 F = (C*9)/5+32
70 PRINT C;" DEG. CELSIUS = "; F;" DEG.
    FAHRENHEIT"
80 PRINT
90 GOTO 10
100 INPUT "ENTER DEGREES FAHRENHEIT: "; F
110 C = (F-32)*5/9
120 PRINT F;" DEG. FAHRENHEIT = "; C;" DEG.
    CELSIUS"
130 PRINT
140 GOTO 10
```

NO SPACE
HERE

DON'T
FORGET
TO HIT
RETURN

If you enter and run this program, you'll see INPUT in action.

Line 10 uses the input statement to not only gather information, but also print our prompt. Also notice that we can ask for either a number or string (by using a numeric or string variable).

Lines 20, 30, and 40 do some checks on what is typed in. In line 20, if nothing is entered (just **RETURN** is hit), then the program goes back to line 10 and requests the input again. In line 30, if **F** is typed, you know the user wants to convert a temperature in degrees **F**ahrenheit to **C**elsius, so the program branches to the part that does that conversion.

Line 40 does one more check. We know there are only two valid choices the user can enter. To get to line 40, the user must have typed some character other than **F**. Now, a check is made to see if that character is a **C**; if not, the program requests input again.

This may seem like a lot of detail, but it is good programming prac-

tice. A user not familiar with the program can become very frustrated if it does something strange because a mistake was made entering information.

Once we determine what type of conversion to perform, the program does the calculation and prints out the temperature entered and the converted temperature.

The calculation is just straight math, using the established formula for temperature conversion. After the calculation is finished and answer printed, the program loops back and starts over.

After running, the screen might look like this:

```
CONVERT FROM FAHRENHEIT OR CELSIUS (F/C): ?F
ENTER DEGREES FAHRENHEIT: 32
32 DEG. FAHRENHEIT = 0 DEG. CELSIUS

CONVERT FROM FAHRENHEIT OR CELSIUS (F/C): ?
```

After running the program, make sure to save it on disk or tape. This program, as well as others presented throughout the manual, can form the base of your program library.

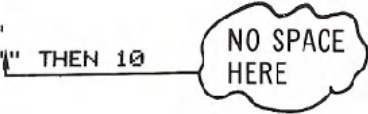
GET

GET allows you to input one character at a time from the keyboard without hitting **RETURN**. This really speeds entering data in many applications. Whatever key is hit is assigned to the variable you specify with GET.

The following routine illustrates how GET works:

NEW

```
1 PRINT "{CLR/HOME}"
10 GET A$: IF A$ = " " THEN 10
20 PRINT A$;
30 GOTO 10
```



NO SPACE
HERE

If you RUN the program, the screen will clear and each time you hit a key, line 20 will print it on the display, and then GET another character. It is important to note that the character entered will not be displayed unless you specifically PRINT it to the screen, as we've done here.

The second statement on line 10 is also important. GET continually works, even if no key is pressed (unlike INPUT that waits for a response), so the second part of this line continually checks the keyboard until a key is hit.

See what happens if the second part of line 10 is eliminated.

To stop this program you can hit the **RUN/STOP** and **RESTORE** keys.

The first part of the temperature conversion program could easily be rewritten to use GET. LOAD the temperature conversion program, and modify lines 10, 20 and 40 as shown:

```
10 PRINT "CONVERT FROM FAHRENHEIT OR CELSIUS  
<F/C>"  
20 GET A$: IF A$ = "F" THEN 20  
40 IF A$ <> "C" THEN 20
```



NO SPACE
HERE

This modification will make the program operate smoother, as nothing will happen unless the user types in one of the desired responses to select the type of conversion.

Once this change is made, make sure you save the new version of the program.

RANDOM NUMBERS AND OTHER FUNCTIONS

The Commodore 64 contains a number of functions that are used to perform special operations. Functions could be thought of as built-in programs included in BASIC. But rather than typing in a number of statements each time you need to perform a specialized calculation, you just type the command for the desired function and the computer does the rest.

Many times when designing a game or educational program, you need to generate a random number, to simulate the throw of dice, for example. You could certainly write a program that would generate these numbers, but an easier way to call upon the RANdOm number function.

To see what RND actually does, try this short program:

NEW

```
10 FOR X = 1 TO 10  
20 PRINT RND(1),  
30 NEXT
```

IF YOU LEAVE OUT THE COMMA YOUR LIST
OF NUMBERS WILL APPEAR
AS 1 COLUMN

After running the program, you will see a display like this:

```
.789280697      .664673958  
.256373663      .0123442287  
.682952381      3.90587279E-04  
.402343724      .879300926  
.158209063      .245596701
```

Your numbers don't match? Well, if they did we would all be in trouble, as they should be completely random!

Try running the program a few more times to verify that the results are always different. Even if the numbers don't follow any pattern, you should start to notice that some things remain the same every time the program is run.

First, the results are always between 0 and 1, but never equal to 0 or 1. This will certainly never do if we want to simulate the random toss of dice, since we're looking for numbers between 1 and 6.

The other important feature to look for is that we are dealing with real numbers (with decimal places). This could also be a problem since whole (integer) numbers are often needed.

There are a number of simple ways to produce numbers from the RND function in the range desired.

Replace line 20 with the following and run the program again:

```
20 PRINT 6*RND(1),  
  
RUN  
  
3.60563664      4.53660853  
5.47238963      8.40850227  
3.19265054      4.39547658  
3.16331095      5.50620749  
9.32527884      4.17090293
```

That cured the problem of not having results larger than 1, but we still have the decimal part of the result to deal with. Now, another function can be called upon.

The **INTEger** function converts real numbers into integer values.

Once more, replace line 20 with the following and run the program to see the effect of the change:

```
20 PRINT INT(6*RND(1)),  
  
RUN  
  
2      3      1      0  
2      4      5      5  
0      1
```

That took care of a lot, getting us closer to our original goal of generating random numbers between 1 and 6. If you examine closely what we generated this last time, you'll find that the results range from 0 to 5, only.

As a last step, add a one to the statement, as follows:

```
20 PRINT INT(6*RND(1))+1,
```

Now, we have achieved the desired results.

In general, you can place a number, variable, or any BASIC expression within the parentheses of the **INT** function. Depending on the range desired, you just multiply the upper limit by the **RND** function. For example, to generate random numbers between 1 and 25, you could type:

```
20 PRINT INT(25*RND(1))+1
```

The general formula for generating a set of random numbers in a certain range is:

$$\text{NUMBER}=\text{INT}(\text{LOWER LIMIT}+(\text{UPPER}-\text{LOWER}+1)*\text{RND}(1))$$

GUESSING GAME

Since we've gone to some lengths to understand random numbers, why not put this information to use? The following game not only illus-

trates a good use of random numbers, but also introduces some additional programming theory.

In running this program, a random number, NM, will be generated.

NEW

```
1 REM NUMBER GUESSING GAME
2 PRINT "{CLR/HOME}"
5 INPUT "ENTER UPPER LIMIT FOR GUESS ";LI
10 NM = INT(LI*RND(1))+1
15 CN = 0
20 PRINT "I'VE GOT THE NUMBER.":PRINT
30 INPUT "WHAT'S YOUR GUESS"; GU
35 CN = CN + 1
40 IF GU > NM THEN PRINT "MY NUMBER IS
    LOWER": PRINT : GOTO 30
50 IF GU < NM THEN PRINT "MY NUMBER IS
    HIGHER": PRINT : GOTO 30
60 PRINT "GREAT! YOU GOT MY NUMBER"
65 PRINT "IN ONLY "; CN ;"GUESSES.":PRINT
70 PRINT "DO YOU WANT TO TRY ANOTHER (Y/N)";
80 GET AN$: IF AN$="" THEN 80
90 IF AN$ = "Y" THEN 2
100 IF AN$ <> "N" THEN 70
110 END
```

: INDICATES NO
SPACE AFTER
QUOTATION MARK

You can specify how large the number will be at the start of the program. Then, it's up to you to guess what the number is.

A sample run follows along with an explanation.

```
ENTER UPPER LIMIT FOR GUESS? 25
I'VE GOT THE NUMBER.

WHAT'S YOUR GUESS ? 15
MY NUMBER IS HIGHER.

WHAT'S YOUR GUESS ? 20
MY NUMBER IS LOWER.

WHAT'S YOUR GUESS ? 19
GREAT! YOU GOT MY NUMBER
IN ONLY 3 GUESSES.

DO YOU WANT TO TRY ANOTHER (Y/N) ?
```

IF/THEN statements compare your guess to the number generated. Depending on your guess, the program tells you whether your guess was higher or lower than the random number generated.

From the formula given for determining random number range, see if you can add a few lines to the program that allow the user to also specify the lower range of numbers generated.

Each time you make a guess, CN is incremented by 1 to keep track of the number of guesses. In using the program, see if you can use good reasoning to guess a number in the least number of tries.

When you get the right answer, the program prints out the "GREAT! YOU GOT MY NUMBER" message, along with the number of tries it took. You can then start the process over again. Remember, the program generates a new random number each time.

PROGRAMMING TIPS:

In lines 40 and 50, a colon is used to separate multiple statements on a single line. This not only saves typing, but in long programs will conserve memory space.

Also notice in the IF/THEN statements on the same two lines, we instructed the computer to PRINT something, rather than immediately branching to some other point in the program.

The last point illustrates the reason behind using line numbers in increments of 10: After the program was written, we decided to add the count part. By just adding those new lines at the end of the program, numbered to fall between the proper existing lines, the program was easily modified.

YOUR ROLL

The following program simulates the throw of two dice. You can enjoy it as it stands, or use it as part of a larger game.

```
5 PRINT "Care to try your luck?"
10 PRINT "RED DICE   = ";INT(6*RND(1))+1
20 PRINT "WHITE DICE = ";INT(6*RND(1))+1
30 PRINT "HIT SPACE BAR FOR ANOTHER ROLL";PRINT
40 GET A$: IF A$ = " " THEN 40
50 IF A$ = CHR$(32) THEN 10
```

Care to try your luck?

From what you've learned about random numbers and BASIC, see if you can follow what is going on.

RANDOM GRAPHICS

As a final note on random numbers, and as an introduction to designing graphics, take a moment to enter and run this neat little program:

```
10 PRINT "{CLR/HOME}"
20 PRINT CHR$(205.5 + RND(1));
40 GOTO 20
```

As you may have expected, line 20 is the key here. Another function, CHR\$ (Character String), gives you a character, based on a standard code number from 0 to 255. Every character the Commodore 64 can print is encoded this way (see Appendix F).

To quickly find out the code for any character, just type:

```
PRINT ASC("X")
```

where X is the character you're checking (this can be any printable character, including graphics). The response is the code for the character you typed. As you probably figured out, "ASC" is another function, which returns the standard "ASCII" code for the character you typed.

You can now print that character by typing:

```
PRINT CHR$(X)
```

If you try typing:

```
PRINT CHR$(205); CHR$(206)
```

you will see the two right side graphic characters on the M and N keys. These are the two characters that the program is using for the maze.

By using the formula $205.5 + \text{RND}(1)$ the computer will pick a random number between 205.5 and 206.5. There is a fifty-fifty chance of the number being above or below 206. CHR\$ ignores any fractional values, so half the time the character with code 205 is printed and the remaining time code 206 is displayed.

If you'd like to experiment with this program, try changing 205.5 by adding or subtracting a couple tenths from it. This will give either character a greater chance of being selected.