Donald Kahn, Jr.
Nevin B. Scrimshaw

A Beginner's Guide to Real Programming

DISCOVER YOUR VIC-20

Discover Your VIC-20

A Beginner's Guide to
Real Programming

Discover Your VIC-20

A Beginner's Guide to Real Programming

Donald Kahn Jr.
and
Nevin B. Scrimshaw

Birkhäuser
Boston • Basel • Stuttgart

Library of Congress Cataloging in Publication Data

Kahn, Donald, 1951-
Discover your VIC-20.

Bibliography: p.
Includes index.
1. VIC 20 (Computer)—Programming. 2. Basic (Computer program language) I. Scrimshaw, Nevin, 1950- . II. Title.
QA76.8.V5K33 1983b 001.64'2 83-15716
ISBN 0-8176-3160-7

A B C D E F G H I J

ISBN 0-8176-3160-7

Printed in USA

Contents

Discover Your VIC-20

A Beginner's Guide to
Real Programming

1 Using the VIC Keyboard

Like all computers, your VIC-20 processes information: it receives information as input, does something with it, and sends the result out again as output. The VIC can process large amounts of data with great speed, but for all if its power it is something of a brute: you must tell it what to do and how to do it. You must feed in not only the information you want processed (data) but the instructions for processing it (programs). This book will teach you how to **program** your VIC-20; how to give the VIC the information it needs (data and programs) to perform your tasks.

Most of this book will be devoted to writing programs in the BASIC programming language. First, however, you need a guided tour around the VIC keyboard. Please be patient with these fundamentals; you will be a more comfortable VIC programmer if you know the keyboard well.

THE KEYBOARD

Your primary means of communicating with the VIC-20 is the keyboard, which is similar in many ways to that of a standard typewriter. But you'll notice that the VIC keyboard has several special keys; these keys make it much more versatile than a typewriter.

Modes

Set up your VIC as directed in the USER'S GUIDE and flip the rocker switch on the VIC's right side. When you first turn the computer on, you're in standard, or upper case/graphic mode. That blinking square under READY is the cursor; it marks the spot where what you type next will appear on the screen.

Look for the two cursor control keys on the lower right of the keyboard. In standard mode, pressing these keys moves the cursor down or to the right. Pressing the [SHIFT] key and a [CRSR] key at the same time moves the cursor up or to the left. In this book, pressing two keys together will be indicated with a colon, e.g., [SHIFT]:[CRSR]. Go ahead and experiment to get a feel for using the cursor controls.

Perhaps the most important key on the keyboard is the [RETURN] key. Like its counterpart on a typewriter, [RETURN] brings you (the cursor actually) back to the left margin, one line down; if the cursor is at the bottom of your screen, pressing [RETURN] scrolls up the display one line. Try it!

But [RETURN] on a computer has a more important function: it signals the computer that you have finished typing instructions and enters those instructions into memory. Failing to press [RETURN] at the end of any statement is like not talking to the computer at all; pressing [RETURN] means that you expect the computer to respond, by executing the statement or by storing it in memory.

Now, starting with the letter A, type in all the characters in the third row from the top of your keyboard (but don't push [RETURN]). You'll see that the VIC prints on the screen either the character on the top face of the key or the lower character when there are two; all the letters are upper-case.

Now push [RETURN]. Your screen should look like this:

```
ASDFGHJKL:;=
?SYNTAX ERROR
READY
```

By hitting [RETURN], you entered an instruction that makes no sense in the BASIC language your computer understands; the way you phrased your instruction—your **syntax**—was faulty, and your VIC responded with an ERROR message. (For a list of ERROR messages and what they mean, see Appendix N, page 16Ø, in your USER'S GUIDE.)

Now, from left to right, press all the keys in the top row. What happened? When you hit the [CLR/HOME] key the cursor shot back up to the left corner of your screen. That position is called home; the computer followed the instruction printed on the lower half of the [CLR/HOME] key, sending the cursor home. Use the down cursor key to bring the cursor back one line below the line you just typed.

On a typewriter shifting gives you a completely different set of characters. On the VIC [SHIFT]ing does the same thing, but what characters or functions you get depend on what mode you're in. Holding down the [SHIFT] key while pressing any other key will print the graphics symbol on the right front face of that key. For keys with two characters on the top (the number keys, for instance), [SHIFT]ing gives you the top character or function. Pushing a [SHIFT]ed [CLR/HOME] key thus not only takes the cursor home, but clears the screen as well.

Type in some letters and symbols and find the [COMMODORE] key in the keyboard's lower left corner. Press this and the [SHIFT] key at the same time.

Welcome to the lower case! In this second mode the VIC keyboard behaves essentially like a typewriter: the [SHIFT] key produces uppercase letters and symbols, but no graphics. Experiment for a while. To get back to the upper case (standard) mode, push both [COMMODORE] and [SHIFT] again.

Quote Mode

A third mode is **quote mode,** and you get in and out of it from either standard or typewriter mode by typing quotation marks: one " gets you in, a second " gets you out. Quote mode can be disconcerting at first because keys like [CLR/HOME], [INST/DEL], and the [CRSR] keys don't do what they did in the other modes; instead they print graphic symbols. [DEL] is the only function that works normally. For example, if you hold down the [SHIFT] key and type [CLR/HOME] you should see a white heart on a dark blue square. I guess your VIC thinks that home is where the heart is.

Quote mode is used with the BASIC command PRINT to display something in a precise format. Try this: first clear the screen by typing [SHIFT]:[CLR/HOME].

Now type PRINT and quotation marks followed by [SHIFT]:[CLR/HOME] and A LITTLE SONG. The line should look like this so far:

PRINT "$\boxed{\heartsuit}$A LITTLE SONG

Now press the down cursor key three times and type AND DANCE" so that the line looks like this:

PRINT "$\boxed{\heartsuit}$A LITTLE SONG $\boxed{Q}\boxed{Q}\boxed{Q}$ AND DANCE"

To make the VIC respond to your command, press [RETURN]. With any luck you should have:

```
A LITTLE SONG



                    AND DANCE
```

In quote mode, pressing the [CRSR] or [CLR/HOME] keys elicits a kind of delayed response from the computer. You don't want the cursor moved or the screen cleared now; rather, you want those actions incorporated into your display. The graphic symbols are placeholders representing what you want done. The computer won't print them when it "reads" them—one at a time — as part of your instruction; it performs the indicated action instead.

Your VIC-20 has still another mode: **reverse field mode.** The heart that appears when you type [SHIFT]:[CLR/HOME] is in a reverse field. To make everything appear on a reverse field, hold down the [CTRL] key (second row left) while pressing [9]. Type in some letters and graphic symbols and watch the effect. What happens when you hold down the space bar? To get back to a normal field, type [CTRL]:[Ø] or hit [RETURN].

SPECIAL KEYS

[RUN/STOP] and [RESTORE]

By now, you've probably filled the screen with symbols and are wondering how to get back to normal mode. Hold down [RUN/STOP] (third row left) and press [RESTORE] (second row right). This combination returns you to the original screen with light green border and blue lettering; programs stored in memory are not altered. When you start experimenting with complex programs, this combination will be a lifesaver.

[RUN/STOP] is also used with BASIC commands and to stop the execution of most BASIC programs. For example, typing LIST followed by [RETURN] displays any program stored in memory. If you decide you don't want a LISTing after all, hitting [RUN/STOP] will break it off. [RUN/STOP] also stops a tape loading sequence.

[CTRL] and [COMMODORE]

Whatever mode you're in, two keys, [CTRL] and [COMMODORE], work something like the [SHIFT] key, giving extra characters or performing special functions when pressed with other keys. Using [COMMODORE], [CTRL], or [SHIFT] with other keys while in quote mode allows you to PRINT an amazing array of patterns in almost any color.

On the VIC-20 [CTRL] is used primarily with the number keys [1]-[8] to set colors (see pages 18-19 in the USER'S GUIDE); when used with numbers [9] and [Ø], it turns the reverse field on and off. It can also produce some special effects when used in quote mode. Try typing:

```
PRINT "
```

Now press simultaneously the [CONTROL] and the [5] keys and a close quote. (Our notation for pressing two keys together is to separate the brackets with a colon e.g. [CONTROL]:[5].)

You should see a reverse field graphic symbol between your quotation marks. Now hit [RETURN]. The text color changes. To get back to the dark blue text color screen push [RUN/STOP]:[RESTORE].

You can also press [CTRL] to slow a BASIC program LISTing to scroll at a more readable pace.

Used with letters in standard mode, the [COMMODORE] key produces the left front graphic symbol. When combined with the [SHIFT] key, it takes you from standard upper case/graphic mode to upper/lower case mode.

[INST/DEL]

In standard mode this key allows you to correct typing mistakes: pushing it moves the cursor back one space, deleting the previous character. (Remember, [DEL] always gets rid of the character to the left of the cursor.) A [SHIFT]ed [INST/DEL] allows you to insert one letter, symbol, or space. To facilitate inserting or deleting more than one character at a time, the key repeats when held down.

In quote mode, [DEL] still deletes, but [INST] prints a reverse field graphic symbol. Pressing [SHIFT]:[INST/DEL] followed by [INST/DEL] in quote mode will print the graphic for [DEL].

Space Bar

Hitting the space bar on a typewriter puts empty space between words. On the VIC the space bar also puts a space between words, but the space is not empty: it holds a perfectly distinct character—just like A or Z—that simply looks like empty space.

This means that you can't use the space bar to advance quickly to the end of a typed line, as you can on an electric typewriter. If you try it, you'll replace each character with a space and wipe out the line. If you want to leave your text intact, you must use the cursor keys to move around.

One more note: in quote mode, where you put spaces affects how your PRINTed display will look.

[SHIFT/LOCK]

Like its counterpart on a typewriter, this key locks on the [SHIFT] functions. But a word of warning: the down and right cursor functions won't work. To unlock the [SHIFT] just push the [SHIFT/LOCK] key a second time. (This feature of pushing a key twice to get back to the original state is called a *toggle*.)

[<], [>] and [↑]

[<] means "less than," [>] means "greater than," and [↑] means exponentiate. The [SHIFT]ed [↑] key, or [$\boxed{\pi}$] allows you to use an approximate value of PI (3.1415926—there's more but no one's ever reached the last digit.) PI is a number that occupies a special place in Nature.

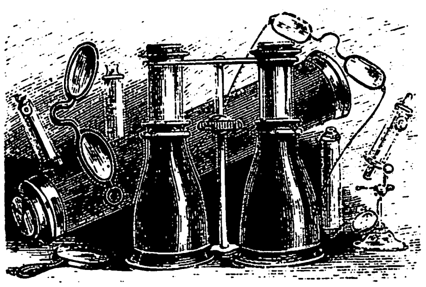
Function Keys

Our look at the keyboard ends with the four FUNCTION keys located on the extreme right. Each key has two functions: one un[SHIFT]ed and one [SHIFT]ed. We will program these keys to perform in a number of different ways in the chapters that follow.

Special Symbols

In addition to the special keys on your VIC, certain symbols have special meanings in BASIC programming language. Here is a list of these symbols; we'll explore their functions further as we use them.

$	identifies a string variable
%	identifies an integer variable
?	replaces the word PRINT (when the program LISTs the word PRINT appears)
:	separates different BASIC statements used on the same line
,	separates variables in DATA lists; prints at tab position on screen (just like tab on a typewriter except there are only two tab positions per line on the VIC)
;	indicates no [RETURN] to the next line for PRINTed displays



2 Introduction to BASIC Programming

The program language we use is an extended form of BASIC. Most home computers use BASIC, but each one uses a slightly different version. VIC-20's BASIC is easy to learn and has a wonderful range of capabilities.

One note: the programs in this book use the VIC-20's full range of sound and color. If you don't have a color TV, you can still run the programs, but the results will be less spectacular. If you wish to use the programs as models for future efforts of your own, you will need an external storage device, either a disk drive or a tape cassette recorder. Refer to Appendices A and B in the USER'S GUIDE for more information.

DIRECT MODE

Let's start in direct mode.

```
PRINT 3+4                                    [RETURN]
```

As you might have predicted, the VIC responded with:

```
7
READY
```

If your VIC responded some other way (with **?syntax error,** for instance), try again.

This example shows how the VIC can be used as a calculator in direct mode (also called immediate or calculator mode). Using the PRINT command and the +, −, *, /, and ↑ keys tells the computer to add, subtract, multiply, divide, or exponentiate and puts the results on the screen. You can perform several operations and use parentheses to govern the order of operations. For instance, type:

```
PRINT 3+2*(6-3)                                [RETURN]

9
READY
```

As in algebraic notation the VIC will perform operations within parentheses first; then exponents from left to right; then multiplications or divisions from left to right; then additions or subtractions from left to right. Note: in contrast to algebraic notation, the multiplication sign cannot be omitted.

In quote mode, the PRINT command will print whatever message is enclosed in quotation marks. Type:

```
PRINT "5+3"                                    [RETURN]

5+3
READY
```

(Please remember to press [RETURN] after each line, as the reminder to do so will no longer be spelled out at the end of each line.) Now type:

```
PRINT "HELLO THERE!"

HELLO THERE!
READY
```

Finally, one PRINT command can perform several tasks. If two calculations or quotations are separated by a semicolon or space, the outputs will be adjacent, as in:

```
PRINT 3+2; 6*2

5 12
READY
```

When two outputs are separated by a comma, the second output begins 11 spaces (or one tab) after the beginning of the first output. For example:

```
PRINT "1234567890","ABC"

1234567890 ABC
```

```
PRINT "3+5=",3+5

3+5=          8
```

A word about spaces: VIC BASIC doesn't care whether you put spaces after commands; spaces just make your commands easier to read. Try each of these:

```
PRINT2+3
PRINT 2+3
PRINT 2 + 3
```

But spaces cannot appear in the middle of a command:

```
PR INT 2+3

?SYNTAX
ERROR
READY
```

And spaces within quotation marks will be printed like any other character.

```
PRINT "A B CDE"

A B CDE
READY
```

PROGRAMMING MODE

Direct mode is useful but limited because you must instruct the computer each step of the way. In programming mode you can store many commands in the VIC's memory, then instruct the computer to execute the commands in order. Try typing:

```
10  PRINT 3 + 5
```

After pressing [RETURN], nothing happened because the initial "1Ø" told the VIC to store the command rather than execute it. Press [SHIFT]:[CLR/HOME], clearing the screen, then type: LIST. Your program will reappear. Now type: RUN and [RETURN]. The RUN command tells the VIC to execute the commands it has stored. You can store more commands:

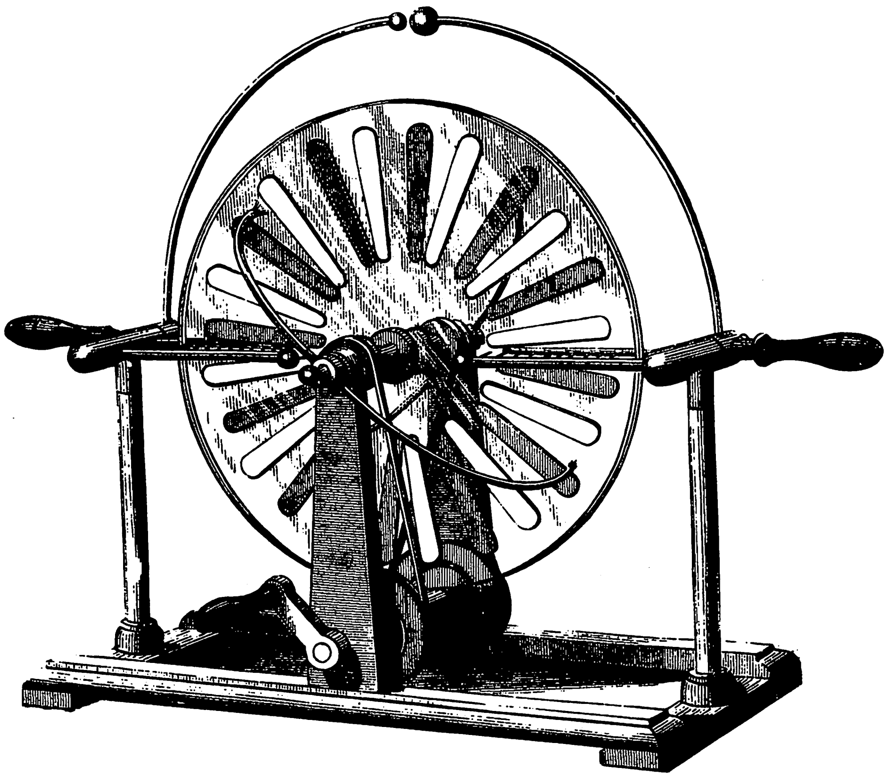
```
20  PRINT "HELLO"
30  PRINT "GOODBYE"
```

And when you type RUN, the VIC will execute your program lines in order:

```
RUN

8
HELLO
GOODBYE
READY
```

You can erase any program line by typing the line number and [RETURN]; or erase the whole program by typing NEW and [RETURN]. In Chapter 4, you will learn how the VIC's editing keys enable you to correct typing errors and otherwise change your program.



3 Bookkeeping in Basic

Bookkeeping means keeping track of large quantities of numbers. The computer is a whiz at this kind of thing, but to use this power you must learn how BASIC stores data in **variables.** Let's experiment in direct mode to give you a feel for programming with numbers and variables. Clear the computer's memory again by typing in NEW and then pressing [RETURN]. Clear the screen by pressing [SHIFT]:[CLR/HOME]. Type:

```
A = 1Ø
B = 2
PRINT A*B
```

If, as usual, you typed [RETURN] after each line, the computer responded with 2Ø, which indeed is ten times two. In this example, the numbers ten and two were stored in variables named A and B. The PRINT command then used the variables to perform the multiplication.

A variable is just a name for some numerical quantity that varies. Temperature, the amount of gas in a car and your weight are all variables. VIC will accept variable names of any length up to 255 characters but only the first two letters count. That means the VIC considers a COUNT and COUNTESS as the same thing: the variable CO. The rules for naming variables are: don't use BASIC keywords and don't use names that have keywords embedded inside them. For example, TOTAL is disqualified since it contains the keyword "TO." Numbers can be used as part of a variable name as long as the first character is a letter. For example, B2 is an accepted variable, 2B is not.

Array and string variables are introduced later in this manual. Refer also to page 113 of your USER'S GUIDE.

There are several ways to assign a value to a variable. Type NEW, then type the following program.

```
10  LET A = 3
20  B = 5
30  INPUT "ENTER A NUMBER";COP
40  LET EX = (A*B)/3+COP
50  PRINT "EX=";EX
```

Type RUN and hit [RETURN]. Enter a number as directed and hit [RETURN]. The number you enter is assigned to the variable COP which is used to evaluate the algebraic expression in Line 40. This program demonstrates the major ways of assigning a value to a variable.

Line 10 simply declares that until further notice A has the value 3.

Line 20 is similar but shows that the use of LET is optional.

Line 30 prints a message and then stops execution of the program to ask the user to INPUT a value for COP. COP will keep that value until you change it.

Line 40 demonstrates one of the really powerful ways to use variables. The variable EX is defined in terms of other variables that had previously been given values. For a sampling of what's to come, add the following line to your program:

```
60  GOTO 10
```

Before you enter RUN find the [RUN/STOP] key over on the left side of the keyboard. Now, RUN the program, and when you get tired of giving COP a value over and over again hit [RUN/STOP]:[RESTORE]. You have just escaped from an infinite loop!

SIMPLE LOOPS

The following program uses another feature of BASIC notation. When I learned math, the expression X = X+3 had to be false. Is 5 = 5+3? Not in my book! But the computer uses X = X+3 as a valid and useful expression. The VIC uses the expression to assign a new value to the variable X. First it takes the old X, adds three to it, and the result of this addition is the new value for X. So if the old value for X was 5, then the new value would be 8. This program will also be our first exposure to FOR/NEXT loops. Type NEW to clear the old program and then type the following program:

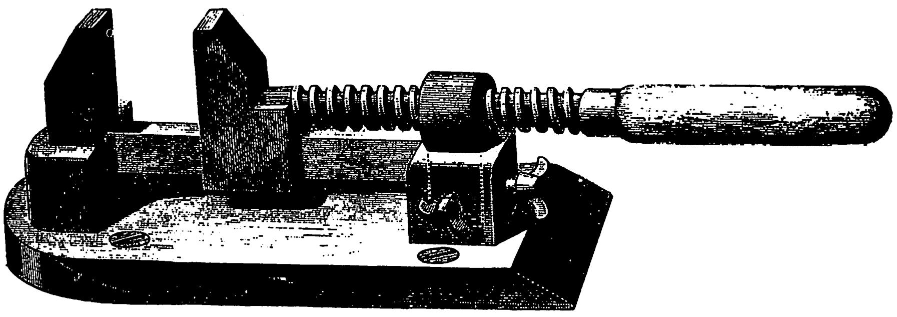
```
 5  LET SUM = 0
10  REM**THIS LINE DOES NOTHING
```

```
20  FOR I = 1 TO 10
30  PRINT I
40  LET SUM = SUM+I
50  NEXT I
60  PRINT "THE SUM IS:"
70  PRINT SUM
```

Line 10 is a REMark statement. It is ignored by the computer when it RUNs the program but can contain very useful program notes which help to decipher the program. The FOR command establishes a loop. It says, "For each value of I from 1 to 10, repeat the program lines between the FOR and NEXT commands." Clear your previous program with NEW. Here's another FOR/NEXT loop for you:

```
10  FOR T = 1 TO 10
20  PRINT 1/T
30  NEXT T
```

Many more ways that algebra does our bookkeeping for us will be explained as we go along.



4 The Basics of Editing

Your VIC-20 has some convenient features which enable you to modify the information on the screen; these features are collectively called the **screen editor**. Let's examine how it works. Type the following program:

```
 5  INPUT A,B
10  PRINT 3+5
20  GOTO 5
```

Press [SHIFT]:[CLR/HOME], type LIST and then press [RETURN]. Perhaps your screen wasn't filled with extraneous lines, but it is always a good idea to clear the screen and take a look at the current version of your program. We want to change Line 10 to add the two input numbers A and B.

Find the two cursor control keys on the lower right hand side of the keyboard. If you hit the [CRSR ↔] key (with arrows going left and right) the cursor moves to the right.

Experiment with moving the cursor by using the two cursor keys along with the [SHIFT] key. Hold [SHIFT] while you hit a cursor key and see what happens. Note also that if you hold down a cursor key, the cursor continues to move until you release the key. Move the cursor until it is positioned over the 3 in Line 10. Type A, and the 3 changes to an A. Move the cursor over to the 5, and change it to a B. When the line reads:

```
10  PRINT A+B
```

hit [RETURN], in order to register the editing change. Remember when you are editing, you must move the cursor with the cursor key. If you move it with the space bar, it will replace the characters it passes over with spaces.

Move the cursor down to an empty line and type RUN. The VIC responds with a "?" and waits for the two numbers you want to add. Type the first number and then hit return; VIC will respond with "??" and wait for you to type the second number. You may input both numbers after the first prompt (a "prompt" is a signal from VIC that it expects input) by placing a comma between them.

After you've experimented with your program a bit, you will want to stop the program execution. Hit the [RUN/STOP] and the [RESTORE] key simultaneously and the screen will clear. Here is a step-by-step explanation of the program: Line 5 tells the VIC to ask for input. Line 1Ø adds the two input numbers. Line 2Ø says GOTO 5, so the VIC goes back to Line 5 and starts over again. This is known as an infinite loop, because the program will keep repeating forever if you let it.

Let's look at another program. Use the editing features you've learned to change the program currently in memory to look like this:

```
 5  INPUT A$
1Ø  PRINT A$
2Ø  GOTO 5
```

See if you can figure out what this program will do. Did you RUN the program and did it respond as you expected?

A$ is a **string variable**, identified by the dollar sign after the variable name. The rules for naming string variables are the same as those for numeric variables; thus, these are acceptable string variables: CAT$, B2$, NAMES$; while 2B$ and TO$ are not.

When assigning characters to a string variable using the LET command, the characters must be placed within quotation marks. Try typing the following in direct mode:

```
LET NAME$ = "FRED"
PRINT NAME$
```

Now type the same commands without the quotation marks; you will get an error message because the VIC was expecting a string variable and you gave it a numeric variable.

While on the subject of variables, an **integer variable**, identified by a "%" after the variable name, will only accept whole number values. Type:

```
X% = 4.3
PRINT X%
```

```
    4
    READY
```

Integer variables take up less memory space, and so can be useful in long programs.

Now, type RUN, hit [RETURN] and input your name. When you see what the program does, hit [RUN/STOP]:[RESTORE]. Whenever you want to exit program execution, [RUN/STOP] will do it, unless VIC is waiting for input. Then you must use [RUN/STOP] and [RESTORE] simultaneously. LIST your program, and edit it so that Line 1Ø reads:

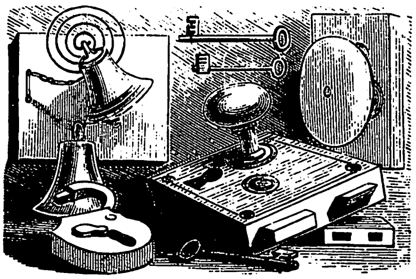
```
1Ø  PRINT A$;
```

When you RUN your program this time, your name gets plastered all over the screen. But it's going so fast that it's difficult to read. Hold down the [CTRL] key. This key will slow the VIC to a more reasonable speed.

Let's recap for a moment:

You now know how to write a simple program, but, equally important, you know how to use almost all of the editing function keys.

Editing within quotation marks is tricky, as keys don't always behave normally in quote mode. You will learn the complexities of editing in quote mode best by experimenting. Try out the cursor keys and [INST/DEL] in different situations to see how they work. Remember, you can always simply retype a line—and this is often the easiest thing to do!



5 Printing with Color

In addition to the standard BASIC your VIC has a useful feature called quote mode that can add nice touches to graphics programs. We'll explore quote mode and a few commonly used BASIC statements in a program that will flash your name in blinking color down the screen. Since we're going to issue several commands involving your name, we will store it in a string variable, in this case the string variable NAME$. You should type your name in place of "Donald."

```
10  NAME$ = "Donald"   (that's my name)
```

Since we want to use the PRINT statement repeatedly, we'll save ourselves some time and effort by using FOR/NEXT loops. Here's the code:

```
20  FOR I = 1 TO 10
30  PRINT NAME$
40  NEXT I
```

You can RUN the program at this point if you want to examine just what these lines are doing. Putting a FOR before and a NEXT after any set of commands will repeat those commands as many times as you indicate in the FOR statement.

SETTING THE STAGE

This program works so far, except it would look better starting with a clear screen. We can set the stage by using the [SHIFT]:[CLR/HOME] key combination inside the quotes of a PRINT statement. When you press these two keys a reverse heart will appear on the screen. When the VIC encounters this print statement during execution of your program, rather than printing the heart, it will clear the screen! Using the command keys in this way is what quote mode is all about. Here's the line; remember that to get the reverse heart you press [SHIFT]:[CLR/HOME].

```
5  PRINT "♥"
```

Now RUN again, and verify that this little line does indeed clear the screen. By the way, my program line numbers are usually multiples of ten to save room in between for little improvements like this. When you write a program, try to make one thing happen at a time; then you can improve the working model. If you do this, and all programmers do to some extent, widely spaced line numbers let you insert extra commands more easily.

MOVING PRINT

Using quote mode, we can move the cursor while executing a PRINT statement. To get a feel for how this works let's modify the program to move your name down the screen. We'll write your name onto the screen, wait a little, erase it, and then repeat that ten times. Just as when we PRINTed the clear command, we get special symbols when we insert other keystroke commands into a PRINT statement. Now we can PRINT NAME$, then move the cursor back to the beginning of your name, and erase it by PRINTing white over it. (Of course, this will work only if the background color is white.)

```
4Ø  FOR M = 1 TO 6
5Ø  PRINT "◫";
6Ø  NEXT M
```

This loop will move the cursor back 6 spaces. This will work if your name is 6 letters. Because names vary in length, the VIC supplies us with a useful string function to make this problem easy to solve.

Observe that we now have a new Line 4Ø; we will need to relocate the NEXT I command to later in the program.

THE LEN FUNCTION

The LEN function returns the number of characters in a string; for example, LEN("ABC") = 3. So to allow our program to handle names of different length we edit Line 4Ø to read:

```
    4Ø  FOR M = 1 TO LEN(NAME$)
```

Now our program will work with any size name. We erase by changing the character color to white and rewriting your name in invisible ink!

The following line uses quote mode to hone our program a bit further. This program line changes the printing color to white [CTRL]:[2] and moves the cursor up one screen line [SHIFT]:[CRSR ↕].

```
    7Ø  PRINT "E●"
```

Let's put in a time delay so it doesn't blink too fast.

```
    8Ø  FOR T = 1 TO 1ØØ : NEXT T
```

Here we use another important syntax feature: the multistatement program line. The colon allows you to put more than one command on a line. Not only does this take up less memory than typing 2 separate lines (keeping as many bytes of memory free as is possible is usually a good idea), it also makes the program easier to read.

PRINTING IN COLORS

Now that our PRINT cursor commands have repositioned the cursor to the beginning of the name, let's write it again, but in white this time (so we can't see it):

```
    9Ø  PRINT NAME$
```

Now we want to change the color back to blue [CTRL]:[7]:

```
    1ØØ  PRINT "←";
```

The use of a semicolon in Line 5Ø and Line 1ØØ is crucial – it keeps the cursor from moving down a line after it finishes a PRINT statement. Now we could use another time delay:

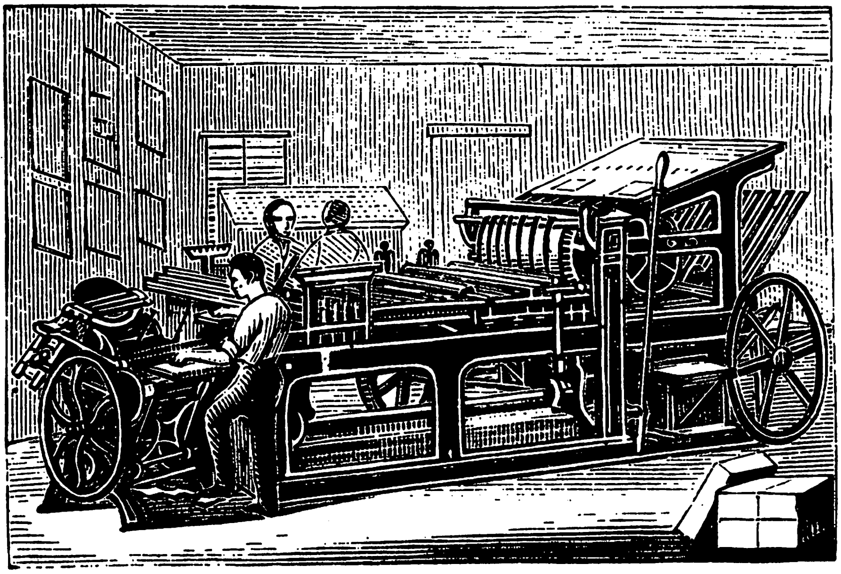
```
    11Ø  FOR T = 1 TO 1ØØ : NEXT T
```

and finally we're ready to close off our big loop with:

```
    12Ø  NEXT I
```

Sit back, dim the lights and RUN.

With this basic structure in hand, you should be able to think of all kinds of visual special effects. We could use an INPUT statement that would inquire as to the user's name; instead of a name, we could use any set of characters; we could make it blink in red instead of blue; or for that matter, any other of the colors available for printing characters. There are all kinds of things you can do, using the methods we have seen so far.



6 Color and the POKE Command

We've already seen how to print something almost anywhere on the screen in any color we like through the use of the PRINT command. But the VIC can do the same thing with the POKE command as well.

Within the VIC's memory are locations (known as addresses), where the VIC may store any number from Ø to 255. The POKE command allows us to insert the number of our choice in many of the memory locations. You have to be careful with the POKE command; making a mistake with an address can lead to spectacular crashes. (A "crash" occurs when the program is derailed and the computer no longer responds to messages from the keyboard. If this should happen, try pressing [RUN/STOP]:[RESTORE]. If that doesn't work, you'll have to turn your VIC off and then on again.) The information that the VIC refers to when determining the right color when printing on the screen is in a block of addresses starting at memory location 384ØØ.

Since the VIC screen is a 22 × 23 grid, there are 5Ø6 individual print positions on the screen. If we use the POKE command to insert a color code into one of these addresses anything already printed at that position on the screen will turn the coded color. Quote mode allows us to control print color, but the POKE command adds a new dimension to what can be done. The color memory map starts at 384ØØ which is the address number of the upper left corner of the screen. The screen is 22 spaces wide, and the addresses increase by one as you go across the row. As you run down a column the addresses increase by 22, so the first four addresses of the first column are 384ØØ, 38422, 38444, and 38466. The address of the lower right hand corner of the screen is 384ØØ +

5Ø5 or 389Ø5. (The character memory map works the same way, except it starts at 768Ø. We'll use these addresses in a later chapter.)

16K MEMORY EXPANSION

If you have a 16K memory expander, or any memory expander, the location of memory maps will be different; however, the relationships will be the same. It shouldn't to be too hard to make the appropriate adjustments. The printed information you received with the extra memory will tell you which starting addresses are changed. Throughout the book, I'll be using the memory locations relevant to the unexpanded VIC-20. 16K can be a real luxury, but an amazing amount of programming can be accommodated by the 5K that comes in the unexpanded VIC.

COLOR PRINTING, TAKE TWO

Let's rewrite our "name" program and make it blink in different colors. Once again, I'm using my name "Donald" where you will use yours.

```
1Ø  PRINT "♥";
2Ø  NAME$ = "Donald"
3Ø  A = LEN(NAME$)
4Ø  FOR I = Ø TO 9
```

(Ø to 9 is just as good as 1 to 1Ø and you'll soon see why we need it this way.)

```
5Ø  PRINT NAME$
```

ERASING

Now we want to erase your name. Since it is written at the very top of the screen, this corresponds to color memory addresses starting with 384ØØ and continuing to 384Ø5, if your name has 6 letters in it. We can erase anything that's written there by POKEing the color to the background color. I assume you're using the white background in this and all examples in this manual so the commands are:

```
6Ø  FOR K = Ø TO A-1
7Ø  P = 384ØØ+K
8Ø  POKE P,1
```

Because this will erase your name on the first line only, we need to increase P by 22 each time we PRINT your name; therefore, we always start POKEing at the beginning of your name.

```
70  P = 38400+22*I+K
```

Now you see why Line 40 uses 0 to 9. It makes Line 70 easier to write and easier to understand. An oddity worth noting: the POKE color code for white is 1; and, in fact, each color's POKE code number is one less than the key on which the color is printed.

Now we close off our two currently open loops with:

```
90  NEXT : NEXT
```

Note that I'm not using NEXT K : NEXT I in Line 90; NEXT : NEXT will work just as well, and it involves less typing for you. It also it takes up less space in memory and executes faster. The only real drawback is that it makes the program more difficult to read.

Now RUN it and you'll see that it does the same thing as our last program, except slower. What we lose in speed we make up for with more control over the color. To start flashing, we'll need to look at the color codes.

If we POKE with a three, we'll get cyan, with two we get red, and one gives us white. If we step backwards from three to one and POKE in that order, we'll get a colorful display. We want to change the color three times each time we run Lines 60 through 90. We'll actually see only two flashing colors, but that's because our third color is the background color, white. So let's insert another step in our program.

```
55  FOR J = 3 TO 1 STEP -1
```

When not specifying the STEP, VIC counts by 1. But we can make it step by −1 or +2 or just about any number. Change Line 80:

```
80  POKE P,J
```

add another NEXT to line 90, and we're done.

RUN first, and then start thinking up ways to improve the display. Here are a few:

Amend Line 10 to PRINT "[♥][E]"; and add 200 PRINT "[←]".This cleans it up a little. Have a little trouble editing Line 10? Maybe when you deleted the closing " and typed [CTRL]:[2], the cursor changed color but didn't print anything. That's quote mode for you! I find it easier to delete everything following the PRINT and then type the new info in.

Let's make the name PRINT diagonally across and down the screen:

```
50  PRINT SPC(I) NAME$
70  P = 38400+22*I+K+I
```

SPC(I) in a PRINT statement prints (I) many spaces. We've also fixed the address calculation in Line 70 by adding (I) to it (just like counting over (I) spaces).

RESETTING COLOR MEMORY

Now that we're changing colors all over the place, we may find that after a particular RUN the screen is all white and there's no blinking cursor. This may happen if we make the program an infinite loop by adding:

```
100  GOTO 10
```

and then hit [RUN/STOP] to end execution. If the character control is white at the time of [RUN/STOP], the color code will still be set at white when we return, and the screen will appear blank due to white lettering on the white background. Hitting [RUN/STOP] and [RESTORE] simultaneously returns the cursor and RESTOREs the screen, but it leaves our program intact.

After you become more adept at programming, you may find that there is more than one way to do the same job. It's possible that there's an easier or more logical way (or at least easier and more logical for you) to produce the same result as our last program. If it works for you, do it!

MORE COLOR PLEASE

You could modify your program to PRINT your name almost anywhere you want, then erase it or flash it in different colors. The "almost anywhere" qualifier is due to a limitation of the SPC function. The SPC function only works for numbers between 0 and 255, but there are 506 print positions on the screen. So, with a simple print command like SPC, we can't use both the top (positions 0-255) and the bottom (positions 256-505) parts of the screen at the same time. Of course, we could program our way out of this dilemma, but that's more than we want to tackle for now.

Now I'll tell you about a VIC function which is very useful for designing interactive programs, games or teaching programs in which the user interacts with VIC.

Let's start with the basic program from last time:

```
1Ø  PRINT "♥";
2Ø  NAME$ = "Donald"
3Ø  A = LEN(NAME$)
4Ø  FOR I = Ø TO 9
5Ø  PRINT NAME$
6Ø  FOR K = Ø TO A-1
7Ø  P = 384ØØ+22*I+K
8Ø  POKE P,1
9Ø  NEXT : NEXT
```

We're going to modify this program to flash your name at random places in the top 256 positions of the screen. Let's make it flash in red! Amend Line 1Ø:

```
1Ø  PRINT "♥£";
```

We'll need the SPC(I) function in Line 5Ø, but instead of doing SPC(I) with I going from Ø to 9, we want to SPC(R) where R is a random number between Ø and 255. If we do this, the VIC will space over R spaces before writing your name, so your name will appear at random places on the screen. RND(1) will work fine—it returns a random number between Ø and 1, including Ø but not including 1. Other numbers may be used in the RND function besides 1; see page 13Ø of your USER'S GUIDE for an explanation. For our purposes, RND(1) will suffice.

If we multiply RND(1) by 256, we get a number between Ø and 256, but not including 256.

It's considered good programming to use only whole numbers in functions such as SPC and commands such as POKE. We can make sure the number we're spacing is a whole number by using the INTeger function. If we INT(35.2) we get 35; the INTeger function turns a number with a decimal point into a whole number! (Whole numbers are known as integers in math talk). This function always returns with the next lower integer. Thus, INT(35.7) equals 35. So:

```
INT(RND(1)*256)
```

will return a whole number between Ø and 255, including both Ø and 255.

We use it as follows:

```
45  R = INT(RND(1)*256)
5Ø  PRINT SPC(R) NAME$
```

And, of course, we need to modify Line 7Ø:

```
 7Ø  P = 384ØØ+22*I+K+R
```

A few more modifications and we're done:

```
 55  FOR T = 1 TO 3ØØ : NEXT   (a time delay)
 9Ø  NEXT
 95  PRINT "♥";   (start at top of screen again)
1ØØ  NEXT
1Ø5  PRINT "←";
```

Since blue is our normal character color and we're blinking in red, Line 1Ø5 will bring us back to normal when the program's done. Now, RUN!

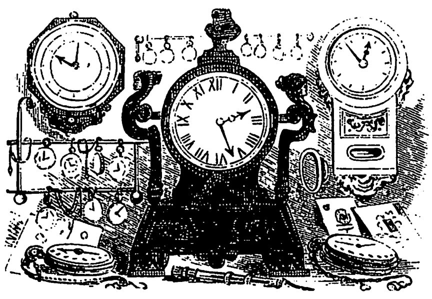
A PROGRAM NOTE

One little problem with our random flashing program—some of the time your name is printed "around the corner." Part of it is at the end of one line while the remainder is at the beginning of the next line. If we could tell VIC to *not* PRINT when we're near the end of a line, the problem would be solved. You can probably figure out how to prevent printing "around the corner" once I tell you about the IF/THEN command, which allows you to execute a statement *if* the situation warrants.

Here's an example:

```
IF (A = B) THEN PRINT "C"
```

It should be obvious that **if** the values of the variables A and B are equal **then** a "C" will be printed. We'll talk more about this very useful command when we use it in future programs. In the meantime, see if you can find a place for an IF/THEN statement in the program we just finished.



7 Loops in Loops in Loops...

We have been using loops in previous programs without examining them closely. These important creatures really deserve a chapter of their own. Loops can be nested inside of each other, and, when teamed with conditional logic, can make your program fold into itself like an Escher design. We will use loops to illustrate the Monte Carlo Method which essentially consists of keeping score while the computer performs a task over and over again.

Here's a little program that will get us started.

That reverse Q in Line 2Ø comes from pressing the down cursor and the reverse line character is the left cursor (there are eight of them below).

```
1Ø  FOR H = 1 TO 12
2Ø  PRINT "PASS# ";H;"◫◫◫◫◫◫◫◫Q";
3Ø  NEXT H
```

In Line 3Ø, the use of the H is optional but is good for bookkeeping purposes. When this program RUNs, the instructions in Line 2Ø are followed 12 times. Notice that inside each pass of the loop, the current value of the counter variable, H, is printed onto the screen. This will help you keep track of what is going on. If we are in loop number five the VIC prints:

```
PASS# 5
```

Now the computation reaches those cursor symbols you inserted in quote mode. The cursor obediently shifts back eight spaces and down one line; and now that semicolon at the end of Line 2Ø keeps the cursor from returning to the left hand column. With Line 2Ø completed, the VIC goes on to Line 3Ø. The computer follows the instruction in Line

3Ø by going on to the next candidate for H. Since there is no STEP specified, the old H is incremented by one to a new value of six.

STEP is a modification of the basic FOR/NEXT loop that allows all kinds of new possibilities. Change Line 1Ø to look like this:

```
1Ø  FOR H = 1 TO 19 STEP 2
```

Now run your program. STEP has made the variable H climb up to 19 from 1 in increments of two. Since there are nine such increments, the number of passes through the loop is ten no matter what the screen says. You might want to take a look at the description on page 121 of your USER'S GUIDE.

The STEPs can be negative as well. Clear your memory with a NEW and try this program:

```
1Ø  FOR I = 2Ø TO 1 STEP -1 : PRINT"DOWN";I : NEXT I
```

Here we have compressed a whole program into one line using the colon to separate statements. We call the colon between the statements a delimiter. RUN the program and you can trace the value of the loop counter as each line is printed. Now we are ready to use the computer to answer questions that have no precise answers.

CHANCE

The RND function generates random numbers, enabling the VIC to simulate chance events. We will use this function to simulate the result of flipping coins. There are a number of slight variations of the RND function; see the USER'S GUIDE for details of these.

Flipping a coin results in either a Head or a Tail. For the sake of this simulation we shall assign the value of 1 to getting a Head and a Ø to getting a Tail as the outcome of a coin flip. The RND function will help us by randomly selecting a Ø or a 1. The program will interpret the result and keep track of the overall tally.

With the function RND and the power of loops we are ready to build a program that will calculate the approximate probability of getting seven or more Heads when flipping ten coins. The program uses a FOR/NEXT loop to perform five trials; each trial consists of simulating flipping 1Ø coins. Every time a trial comes out with seven or more Heads, the counter CO is incremented by one.

Clear the computer with a NEW and let's get started. First we want the computer to do something five times:

```
5  ET = Ø : CO = Ø : PR = Ø : REM JUST INITIALIZING
1Ø FOR TR = 1 TO 5
```

Now to describe what that thing is:

```
2Ø REM**TEN FLIPS**
3Ø FOR FL = 1 TO 1Ø
4Ø H = INT(RND(1)*2)
5Ø ET = ET+H
6Ø NEXT FL
7Ø IF ET>6 THEN CO = CO+1
```

That will do the trick. Line 4Ø does our work for us. It randomly chooses a zero or a one and assigns that value as the new value for H. ET is our Heads counter. If RND has chosen a one, then H = 1 and ET gets incremented by 1. If H = Ø, then we add nothing, because Ø stands for Tails. CO counts how many times ET is seven or greater. After the inner loop is finished, the program reaches Line 7Ø and asks: Is ET greater than six? If so, the counter CO gets bumped up by one. Now we close off the big loop and calculate the estimated probability.

In our notation .5 means 5Ø percent probable, .3 means 3Ø percent probable and so forth. These lines close the outer loop and compute the statistics.

```
8Ø  NEXT TR
9Ø  P = CO/5
1ØØ PRINT "THE ESTIMATED PROBABILITY IS";P
1Ø5 PRINT
```

RUN the program. Do we have a bug? The answer you get looks suspicious. Has something gone wrong? Our counter ET did not come home after the first pass. Add the following line:

```
75 ET = Ø
```

This line initializes the counter ET back to Ø so that it is ready to count again. The problem was that the number of Heads in one pass was added onto the count for the next pass. By adding Line 75 we reset the counter ET. This works because the previous line has just extracted the information we needed from the old version of ET.

The program now works, and you can adapt it to answer similar questions. This program is an example of a nested loop. The inside loop flips 1Ø coins. The outside loop performs the inside loop five times. To see how accurate our estimate is we will add another loop to the program, thus flipping 1Ø coins five times 1Ø times.

Add to our previous program the following lines:

```
  7  FOR Z = 1 TO 1Ø
11Ø  PR = PR + P
12Ø  NEXT Z
13Ø  PRINT "THE AVERAGE ESTIMATE IS";PR/1Ø
```

LIST the program and take a look at the triple nested loop we just built. We have picked up the same kind of bug as before. Let's bring CO home by adding:

```
115  CO = Ø
```

The program is designed to print the estimate given by each batch of 1Ø trials. Before you run this program it might be nice to add the following line to clear the screen at the begining; the character inside the quotes comes from pressing [SHIFT]:[CLR/HOME].

```
  2  PRINT"♥";
```

Now RUN it. You could change the loop numbers in Lines 1Ø and 3Ø to other numbers, or change the conditional in Line 7Ø. Experiment! Have fun! But take a break now and then so you don't get "computer eyes."

Nested loops can be used in constructing graphics displays too. Clear the computer with the NEW command and type in the following program:

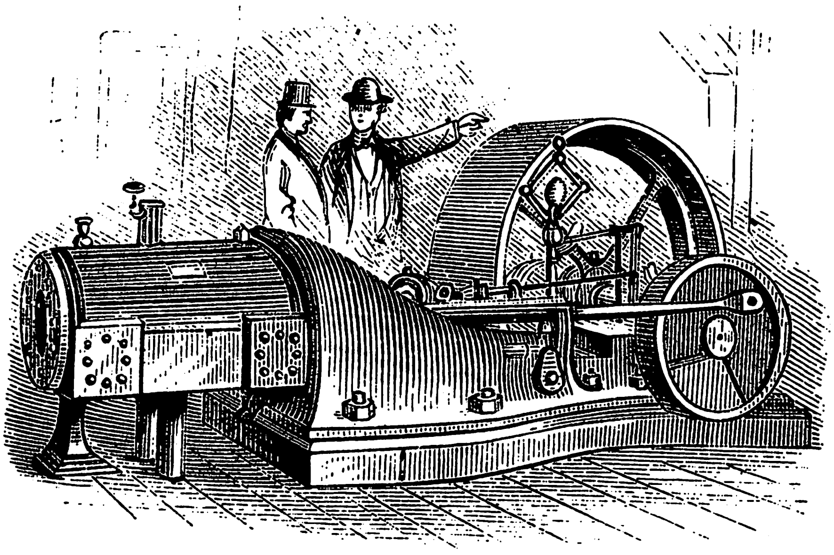
```
 5  PRINT "♥";
1Ø  FOR P = 1 TO 2Ø
2Ø  FOR H = 1 TO 2Ø
3Ø  PRINT"*";
4Ø  NEXT H
5Ø  PRINT
6Ø  NEXT P
```

The semicolon in Line 3Ø keeps the VIC from executing a return. After the inner loop has printed out a line of asterisks, it exits the loop and encounters the PRINT on Line 5Ø. Since there is nothing to print, the VIC prints nothing; but this time there is no semicolon, so the cursor does get bumped to the beginning of the next line. Now the next pass through the inner loop has a fresh line to work with. RUN this program and you will see a block of asterisks appear on the screen.

To see a classic application of loop/variable technology, LIST your program and amend Line 2Ø to read:

```
2Ø  FOR H = 1 TO P
```

Don't forget to hit [RETURN] after you enter your change. Now, before you run the amended program, try to predict what this single change will accomplish.



8 Music 1

Your VIC can be programmed to imitate an electric organ. Certain features of the organ are, at this writing, not reproducible; however, as more people program, we will see software that will enable the VIC to do just about everything an organ does. Programs already exist that can make the VIC sound like almost any instrument; such programs are beyond the scope of this book, but we can still do some impressive music-making.

We will be using the POKE command which is essential in music programs. A command such as:

```
POKE 36878,14
```

stores the value 14 in memory location number 36878. When we used the POKE command for graphics back in Chapter 6, there were over 5ØØ different addresses that were pertinent. In music we are concerned with only five. The VIC uses three addresses for soprano, alto, and bass; one address for volume, and one address for sound effects. We'll ignore the one for sound effects. The addresses that control the others are:

Bass =	36874
Alto =	36875
Soprano =	36876
Volume =	36878

Values that can be POKEed into a voice address to produce a tone range from 128 to 255. The pitch increases as the number does, except for 255, which produces the lowest note of the voice. In the Appendix, you'll find the approximate POKE values for playing customary musical notes. You'll probably get the best results from your TV speaker by using values between 163 and 225. The volume (36878) can be set from Ø to 15. Once we've done the first program, you might experiment to see which volume setting sounds best on your TV.

There is, of course, a price to pay for the simplicity of needing only five memory registers. When we're POKEing for animation or graphics, the VIC knows what we mean by, say,

```
POKE 384ØØ,1
```

That is, it remembers what color 1 represents. Consequently, when we're doing graphics, we're just manipulating information the VIC already has. It's quite a different story when we're making sound programs. While the VIC does know what POKEing 15Ø into the bass voice (36874) means, there's little connection between the numbers 128-255 and the musical scale as we know it. Sure, we can approximate middle C by POKEing in the right values, but we can't just say:

```
POKE 36875, middle C
```

In music programs, we're dealing with only a few addresses, but there's a mass of information we have to organize in order to (a) turn the typewriter keyboard into an organ keyboard, and (b) turn a range of tones into a musical scale. Generally, the most flexible way to store a related block of information in the VIC is in an array. An array is simply a list with a fancy name. When we have lists in programs, there are a number of things we have to do before we can use the list:

First, we have to say how long the list is with the use of a DIM statement.

```
1Ø  DIM A(9)
```

This command creates a list, names it A, and creates 1Ø places in it. (It's 1Ø places because the first place is Ø and the last is 9.)

A(N) is a variable array; an array gives us many variables for the price of one, because A(Ø), A(1), A(2) etc. can all have different values. They can also be conveniently manipulated using a FOR/NEXT loop, as in our program.

Next, we have to fill the list:

```
2Ø  FOR N = Ø TO 9 : READ A(N) : NEXT
```

The READ statement is similar to the INPUT statement, except that while INPUT looks to you for answers, READ expects the information to be in the same program:

```
5ØØ  DATA 195,199,2Ø1,2Ø3,2Ø7,2Ø9,212,215,217,219
```

DATA statements can be placed anywhere in a program; they are commonly put at the end.

After executing these 3 statements—DIM, a READ loop, and DATA—we can use our list A:

```
30  POKE 36878,15  (turn on the volume)
40  FOR R = 0 TO 9
50  POKE 36874,A(R)
60  NEXT
70  POKE 36878,0 : POKE 36874,0
```

If you try to RUN this, you might notice that the notes are awfully short—so put in a time delay:

```
55  FOR T = 1 TO 300 : NEXT
```

You may notice a slight difference if you add:

```
58  POKE 36874,0
```

(POKEing a voice with 0 turns it off.)

The piano program in your VIC manual uses the keys 0 thru 9 and the VALue function to indicate which number in the list is to be POKEd into the voice. Recall:

```
VAL("1") = 1
VAL("9") = 9 etc.
```

I used additional keys as I find these nine keys to be too few to make the VIC an acceptable organ. Note that your manual uses the GET statement to activate the keyboard.

```
100  GET M$ : IF M$ = "" THEN GOTO 100
```

This is the standard procedure for waiting until a key is pressed, and then remembering it. You'll be using this statement (or something similar) in any interactive program you write. (Interactive just means you get involved while the program is running.)

My organ programs use more than nine keys, and I use ones that are convenient to press: those in the middle of the keyboard. There's more than one way of identifying a particular key, say the A key, to the VIC. The simplest way is to call it "A." Each character also has its own code number, called the ASCII code number. For example, the ASCII code number for "A" is 65. You can find a table of ASCII codes in the Appendix. Note that in programming we **must** use a code for the function

(F1,F2,etc.) keys.

In the next program, you may substitute the characters for their ASCII equivalents. You might ask: what's the difference? Whenever we're writing a program, especially animation and music programs, we want the VIC to execute as quickly as possible. We can always slow the execution by inserting time delays, but the rapidity of execution is determined by the number of calculations we make—the more calculations, the slower the execution. Using characters instead of their ASCII codes speeds up the program slightly.

Now let's take a look at the program:

```
 10  DIM A(19), B(19)
 20  S2 = 36875 : S1 = 36876
 30  V = 36878 : POKE V,5
 40  POKE S1,0 : POKE S2,0
 50  FOR N = 0 TO 19 : READ A(N) : NEXT
 60  FOR N = 0 TO 19 : READ B(N) : NEXT
100  GET M$ : IF M$ = "" THEN 100
110  M = ASC(M$) : FL = 50
120  FOR I = 0 TO 19
130  IF M = B(I) THEN FL = I
140  NEXT
145  IF FL > 30 THEN 100
150  IF FL < 14 THEN POKE S2,A(FL) : GOTO 170
160  POKE S1, A(FL)
170  FOR T = 0 TO 300 : NEXT
180  POKE S1,0 : POKE S2,0
190  GOTO 100
500  DATA 195,199,201,203,207,209,212,215,217
510  DATA 219,221,223,225,227,201,203,207,209
520  DATA 212,215
530  DATA 65,87,83,69,68,70,84,71,89,72
540  DATA 85,74,75,79,76,80,58,59,42,61
```

You could make the following changes to use characters rather than ASCII codes:

```
 10  DIM A(19), B$(19)
 60  FOR N = 0 TO 19 : READ B$(N) : NEXT
130  IF M$ = B$(I) THEN FL = I

530  DATA "A", "W", "S", "E", "D", "F", "T", "G", "Y","H"
540  DATA "U", "J", "K", "O", "L", "P", ":", ";", "*", "="
```

Now you could delete the first part of Line 11Ø (M = ASC(M$))—this calculation is no longer necessary.

Since the above program without the changes is a useful stepping stone to other programs, we'll leave it in its inefficient form.

Most music programs should be broken up into sections, since they're usually long. If this is done, it's easy to LIST each part of the program separately:

```
LIST 1-9Ø
```

This part of the program is the set-up; we're dimensioning lists, naming variables, turning on the volume, turning off the voices, and reading in data. We turn off the voices first so that any tone that happened to be in there from before is turned off. The assignment statements, in Line 2Ø and half of Line 3Ø, aren't really necessary, but they make life easier: when we are typing the program or in case the volume is stuck in the "on" position. As soon as we run this program one time, VIC remembers that V = 36878, so if we have trouble with a program and end up with the volume turned on, we can turn it off quickly with:

```
P shift O V,Ø
```

or

```
POKE V,Ø
```

(Most of the commands have abreviations you may use—usually the first letter followed by the second letter shifted. You will find a complete list in Appendix D of your USER's GUIDE.)

```
LIST 1ØØ-2ØØ
```

I've done something tricky here: for the upper tones of the alto voice, I'm using the equivalent lower tones of the soprano voice, because I think it sounds better. That's why I have FL and Line 16Ø in there. (Also notice that my DATA statement Line 51Ø reads, in part, 225, 227, 2Ø1!)

The easiest way to understand any program is to pretend you're the VIC, and execute. If I had hit the W key:

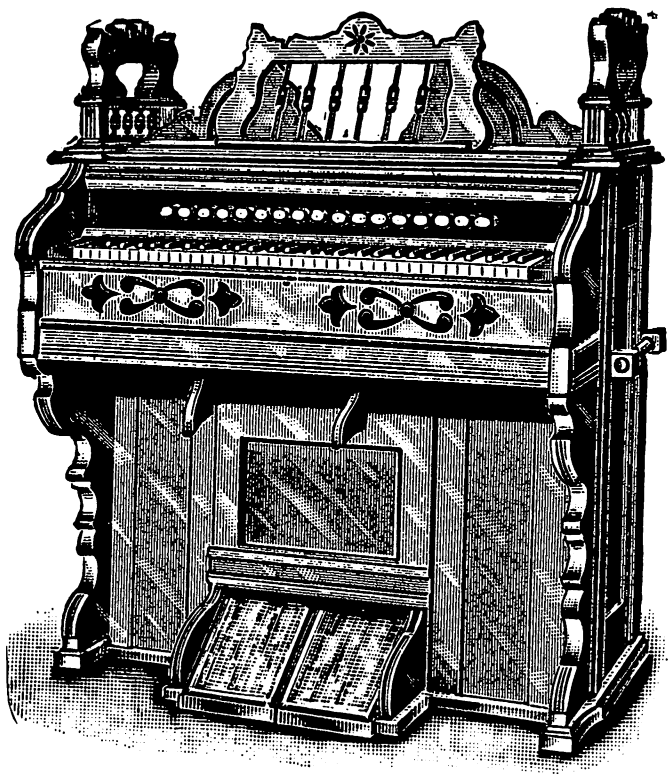
```
1ØØ  M$ = "W"
11Ø  M = 87, FL = 5Ø
12Ø  I = Ø           I = 1           I = 2  (etc.)
13Ø  87=65?(NO)   87=87?(YES)   87=83?(NO)   [FL = 1]
15Ø  1<?14 yes so POKE S2, [A(1) which is 199] and goto 17Ø
17Ø  time delay
```

180 turn off voices

190 start again

Notice that in Lines 120-140, even after I find which list number the W key represents, I still check all the rest of the list. This is clearly a waste of time, and our next program will do this more efficiently. Line 145 returns to Line 100 (GET) in case the key I've pressed doesn't correspond to the 20 keys I've told VIC about. Note that GOTO may be omitted immediately following THEN. Maybe you would like to customize this step to print out some message of mild annoyance, such as "Not that key, try another" before returning to Line 100.

This is your basic organ program. the VIC keyboard is set up like a piano keyboard: the a-s-d-f row are the white keys, and the q-w-e-r row are the black keys. This is cute, but it's not particularly useful, since I can't play chords anyway. (Just wait!) Also, I'm not using all of the available tones, or all of the usable keys. If you can't wait to hear how my organ turns out, go directly to MUSIC 2 (Chapter 10); but you'll be missing an excursion into the realm of Summeria.



9 Number Theory 1

In this chapter we put the computer to work doing the kind of tasks it does best, repetitions of simple procedures. How long would it take you to add the first 1ØØØ odd numbers? Before taking pencil in hand, let's see how to program the VIC to solve this problem. There are several ways to find the answer but one of the simplest is:

```
 5 S = Ø : CO = Ø
1Ø FOR Q = 1 TO 1999 STEP 2
2Ø   LET S = S+Q
3Ø   PRINT S
4Ø NEXT
5Ø PRINT "THE SUM IS";S
```

Note that we've left the NEXT bare. From now on, except when we illustrate a looping feature, the counter variable (Q in this case) will be left off the NEXTs. Note also that here I'm indenting within each FOR/NEXT loop; this is a good practice to get into when writing programs. It makes the code easier to read. But when you're typing it into the VIC, leave out the extra spaces. In fact, leave out as many spaces as possible; the VIC will interpret the code in the same way, but it will take up less memory. The practice of omitting unnecessary spaces also allows you to squeeze longer programs onto the VIC screen.

Let's RUN our program so far and see what we've accomplished.

The VIC will have to work a bit on this one, so be prepared to wait a minute or so. When you run the program the intermediary sums will go flashing by until you get a final answer of 1,ØØØ,ØØØ. That seems a bit suspicious. Did we really get the first 1ØØØ? How can we be sure that we didn't add 999 numbers by mistake? To check this we can insert tracers into the program and add a counter of additions performed by adding the following lines:

```
25  CO = CO+1
3Ø  PRINT S,"▯▯▯"Q,"▯▯▯▯▯"CO
45  PRINT "THE COUNTER IS AT";CO
```

The variable CO enters the loop with the value of Ø, and in the first pass through the loop, CO gets incremented up to 1. That's what we want, because this is pass #1.

Line 3Ø gives us an interim status report on our variables. This will slow our computation because the computer has to stop and print. But the visual effect is satisfying and helps make this program more transparent. What an odd looking PRINT statement Line 3Ø is! Normally, a comma between variables causes a separation of 12 spaces (one tab). Unfortunately, our VIC has only 22 spaces across. We PRINT at the tab locations, but we backspace ([SHIFT]:[CRSR ↔]) first three times, then five times, so each execution of Line 3Ø prints on one line. When you run the program now, you will see three lines of numbers scrolling up your screen. The righthand column is the number of loops so far, the center contains the odd number of the moment, and the left column contains our running sum. After a minute or so, the program halts, and the bottom of our three columns looks like this:

```
996ØØ4     1995     998

998ØØ1     1997     999

1ØØØØØØ    1999     1ØØØ

THE COUNTER IS AT 1ØØØ
THE SUM IS 1ØØØØØØ
```

Now we can start having more confidence in the answer. The counter indicates that we have added 1ØØØ numbers. Let's check further to see if there isn't some hidden logical error in our program.

Looping around 1ØØØ times is somewhat unmanageable, so we need to bring the program down to size. Use the screen editor to change Line 1Ø as follows:

```
1Ø  FOR Q = 1 TO 9 STEP 2
```

Now when you run the program it all fits on the screen and should look like this:

```
1        1     1
4        3     2
9        5     3
16       7     4
25       9     5
THE COUNTER IS AT 5
THE SUM IS 25
```

This output is easier to handle. The third column indicates that we have passed through the loop five times. The second is the list of numbers that we want to add, and it is easy to check that it contains the first five odd numbers. There is no arguing that the sum of these numbers is 25. The first column gives a running subtotal, and ends where it should, with 25. The case is now pretty strong that our answer of 1,ØØØ,ØØØ for the first 1ØØØ odd numbers is correct.

Inspect this print-out and see if you can figure the formula for getting the sum of the first N odd numbers. Compare the third column (the number of odd numbers added so far) with the first column (the running sum).

Let's use this program one more time and add up the first 5ØØ odd numbers. Five hundred is one half of 1ØØØ so it might be interesting to see how this sum will compare with 1,ØØØ,ØØØ. Change Line 1Ø again, to:

```
1Ø  FOR Q = 1 TO 999 STEP 2
```

When you run the program this time, use a watch or clock to time it. We will then delete Line 3Ø, and see how much time the PRINT statement is using.

Let's see what the sum of the first 5ØØ odd numbers is. RUN the program.

```
THE SUM IS 25ØØØØ
```

Hmmm. . . we add up half as many numbers and get one quarter the sum. The computation took about 35 seconds.

Erase Line 3Ø by typing 3Ø on any empty line and hitting [RETURN]. Now get out your timepiece again and we can see whether or not that PRINT statement really chewed up computation time. Clear the screen, check the time and RUN the program. Pretty impressive improvement!

Did you figure out the formula? The sum of the first N odd numbers is N*N.

EVEN STEVEN

Let's thicken the plot. What is the sum of the first 1ØØØ *even* numbers? A very simple modification of our program can handle this. This time we will let the VIC keep the time with its internal clock.

There is a somewhat sticky point to settle: is zero or two the first even number? We'll rule that two is the first.

Since this program takes so long, we will start with the first 5ØØ even numbers. Here's the program written for the first 5ØØ:

```
 5  SC = TI : S = Ø : CO = Ø
1Ø  FOR Q = 2 TO 1ØØØ STEP 2
2Ø  S = S+Q
25  CO = CO+1
3Ø  PRINT S,"◫◫◫"Q,"◫◫◫◫◫"CO
4Ø  NEXT
45  LET CT = (TI-SC)/6Ø
5Ø  PRINT "THE COUNTER IS AT";CO
6Ø  PRINT "THE SUM IS";S
7Ø  PRINT "COMPUTATION TIME:";CT;"SECONDS"
```

The variable TI is a reserved system variable that contains the current value of the VIC's internal clock. This variable increments at one sixtieth of a second intervals.

Let's use direct mode to see how long your computer has been on. Find an empty line and type:

```
PRINT TI/36ØØ
```

Hit [RETURN]. The number that appears on the next line shows how many minutes your computer has been on. If the answer looks wrong, it's because using your cassette program recorder has a strange effect on your VIC's internal clock. However, the internal clock will still function properly as far as evaluating elapsed time is concerned, so our program will work OK. Go ahead and RUN your program. My VIC took 36.6166667 seconds (or so) to RUN. Now for the *pizza de resistance*: erase Line 3Ø, as before. RUN the program again and measure how much this has tightened up the program. My computer took 3.2 seconds: we have improved by a factor of 1Ø.

Go ahead and amend Line 1Ø to find the sum of 2 + 4 + 6 + 8 + 1Ø + 12 + 14... + 2ØØØ. Trying to figure out a simple formula for the sum of the first N even numbers? It's not as easy as the odd numbers. If you know what Gauss did as a schoolchild you can probably figure out the formula, but that's too hard. Try N*(N+1). Isn't math fun?

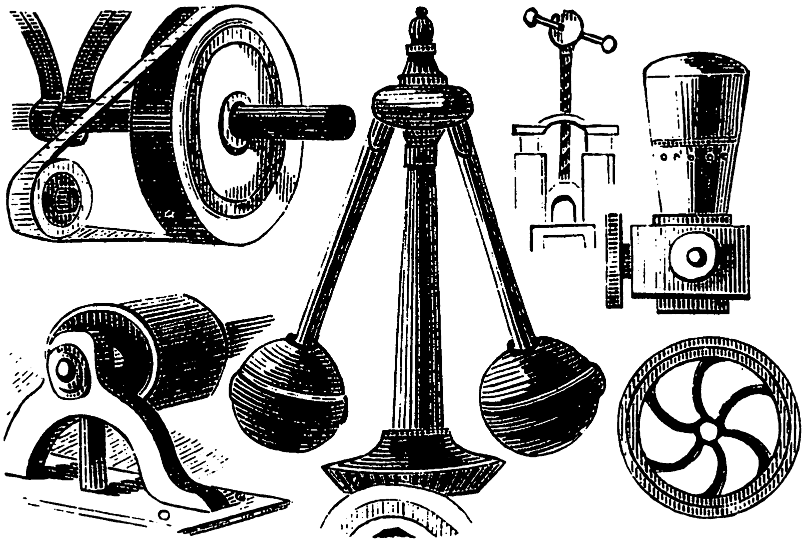
EXPONENTIALS

Here is a program that will add up the first eight powers of two:

```
 5  SUM = Ø
1Ø  FOR E = 1 TO 8
2Ø  SUM = SUM + (2↑E)
3Ø  PRINT SUM,"□□□"E,"□□□□□□"2↑E
4Ø  NEXT
5Ø  PRINT "THE SUM IS";SUM
```

And the formula is 2↑(N+1)−2.

Now you have the tools to handle all sorts of counting arguments. In Chapter 14 we'll pick up these threads again and look at the pattern that chance weaves.



10 Music 2

Here's the better organ program I promised you two chapters back. This is the first part of the program; again, we dimension, we assign variables, we fill lists (taking care of business).

Here's an editing trick that can save time: when you're done with Line 4Ø, hit [RETURN], move the cursor back up, and change the 4Ø to a 5Ø. Hit [RETURN], go back, and change it to a 6Ø, and again to 7Ø. Now LIST it, and edit Lines 5Ø-7Ø so the array names match what's below. This little editing trick can be very useful.

```
1Ø  DIM A(12), B(12), C(12), D(12)
2Ø  S1 = 36876 : S2 = 36875
3Ø  S3 = 36874 : V = 36878
4Ø  FOR N = Ø TO 12 : READ D(N) : NEXT
5Ø  FOR N = Ø TO 12 : READ A(N) : NEXT
6Ø  FOR N = Ø TO 12 : READ B(N) : NEXT
7Ø  FOR N = Ø TO 12 : READ C(N) : NEXT
```

Now we'll scan the keyboard and define the meaning of our special function keys:

```
1ØØ  GET M$ : IF M$ = "" THEN 1ØØ
11Ø  POKE V,1
12Ø  M = ASC(M$)
13Ø  IF M = 17 THEN 5ØØ
14Ø  IF M = 29 THEN POKE V,Ø : GOTO 1ØØ
15Ø  IF M < 13Ø THEN 2ØØ
16Ø  IF M = 133 THEN POKE S1,Ø : GOTO 1ØØ
17Ø  IF M = 134 THEN POKE S2,Ø : GOTO 1ØØ
```

```
180  IF M = 135 THEN POKE S3,0 : GOTO 100
```

Lines 100 through 120, you've seen before. Lines 130 to 140 and 160 to 180 set up programmed function keys. In Line 130, 17 is the ASC code for the down cursor. Hitting the down cursor ends the program, after turning off the volume. (We'll check out Line 500 later on.) No longer do we need to hit [RUN/STOP] to get out of this program! 29 is ASC for the right cursor. Hitting this key turns off the volume, but doesn't turn off the voices; so the next note key hit after this one will give an interesting effect. Code numbers 133-135 (Lines 160 to 180) turn off the individual voices; 133 is ASC for F1, 134 is ASC for F3, and 135 is ASC for F5. (Check the ASC chart in the Appendix.) Line 150 instructs the computer to skip Lines 160 to 180 if M < 130, since all the other keys in use here have ASC codes that are less than 130. This eliminates Lines 160 to 180 when they're not necessary, saving execution time.

In the program in Music 1, I searched for the key which was hit using a FOR/TO/NEXT loop, but this turned out to be inefficient because the computer kept searching through the lists even after we'd found the right key. We could have inserted an IF/THEN statement, and a GOTO out when it was time; however, leaving a FOR loop before it is finished is frowned upon. I don't think it would have hurt us this time, but it's a bad habit to develop.

Now we need a few more test statements:

```
200  I = 0
210  IF M = A(I) THEN POKE S1,D(I) : GOTO 100
220  IF M = B(I) THEN POKE S2,D(I) : GOTO 100
230  IF M = C(I) THEN POKE S3,D(I) : GOTO 100
240  I = I+1
250  IF I < = 12 THEN 210
260  GOTO 100
```

Line 200 sets I = 0; Lines 210-230 check which key was hit, and, if it's found, POKE the appropriate voice and return to the GET in Line 100. Line 240 updates I, and then Line 250 returns us to Line 210 for another pass through if I is less than or equal to 12. (Recall that lists A, B, C, and D only go up to A(12), etc.) If we never find the key, we eventually get to Line 260, and that takes us back to the GET on Line 100.

```
500  POKE V,0 : POKE S1,0
510  POKE S2,0 : POKE S3,0
```

```
800  DATA 2Ø7,2Ø9,212,215,217,219,221,223
81Ø  DATA 225,227,228,229,231
82Ø  DATA 95,49,5Ø,51,52,53,54,55,56,57,48,43,45
83Ø  DATA 81,87,69,82,84,89,85,73,79,8Ø,64,42,94
84Ø  DATA 65,83,68,7Ø,71,72,74,75,76,58,59,61,13
```

Line 5ØØ is where we GOTO from Line 13Ø. Hitting the down cursor key turns everything off (the 3 voices and the volume); then the program ends because the remaining portion of the program consists of DATA statements and these aren't active. Lines 8ØØ-81Ø fill the D list with the note values.

You will notice that although I recommended you not use note "frequencies" above 225, here I have done so. The choice depends on how musically inclined you are. Below 225 the difference between neighboring note values is at least 2, but above 225 it's often less. What this means is that note values above 225 aren't as accurate as the ones below. If you and your family can't hear the difference, you're free to use all the note values. However, if you have a more discriminating ear for music, you should stay within the range 163-225. The 225 is limited by the accuracy deviation; the 163 by your TV's puny speaker. (If you can figure out a way to hook the VIC up to your stereo, the lower limitation of 163 may go away!)

Line 82Ø fills the A list, for the S1 or soprano voice, and this voice is activated by the top row of keys on your keyboard (left arrow through minus). Line 83Ø, the B list for S2 or alto, holds the ASC codes for the second row of keys (Q through up arrow). Line 84Ø, the C list, for S3 or bass, runs from A to RETURN.

DIFFERENT SOUNDS

You may wish to use note values other than the ones I've used. If you're musically knowledgeable, you might try to program partials so you can imitate instruments other than the organ. Those of us who aren't so musically inclined can imitate a piano's sound by POKEing the same tone in two adjacent voices, and then, while they're playing, decrease the volume to Ø in a FOR loop.

Once you have an organ program you like, it's time to start experimenting! I recommend that you try any new routine on its own, before incorporating it into a long program such as the last one.

Here's a little innovation I devised:

```
1Ø  DIM A(8), R(12)
```

```
 20  S1 = 36876 : S2 = 36875
 30  V = 36878 : S3 = 36874
 40  FOR I = 0 TO 8 : READ A(I) : NEXT
 50  J = 0
 90  T = 0
100  GET N$ : IF N$ = "" THEN T = T+1 : GOTO 100
110  POKE V,5
120  N = VAL(N$)-1
130  POKE S1,A(N)
140  R(J) = A(N)
150  IF J>0 THEN R(J-1) = T
160  J = J+2
170  IF J <= 12 THEN 90
210  GOSUB 500
220  FOR J = 0 TO 10 STEP 2
230  POKE S1, R(J) : FOR T = 1 TO (R(J+1))*8 : NEXT : NEXT
240  POKE S1,0
250  GOTO 210
500  GET M$ : IF M$ = "" THEN 500
505  M = ASC(M$)
510  IF M = 32 THEN POKE S1,0 : GOTO 500
520  IF M = 133 THEN POKE S1,0 : RETURN
530  M = VAL(M$)-1
540  POKE S1,A(M)
550  GOTO 500
800  DATA 135,147,159,163,175,183,191,195,201
```

The keys 1-9 play the notes. First, you set up a refrain or repeat phrase of 6 notes; then you play on keys 1-9 until you want the refrain, at which point you press F1. The space bar (ASC = 32) POKEs the note off, so you may play the same note twice in succession or insert rests (when nothing is sounded).

Although I defined all three voices in Lines 20-30, I'm only using S1. But now it's a little easier for you to experiment—try the other voices, or try 2 different ones (say, S2 in 510-540).

Lines 10-40 you've seen before. A is my list of notes, and R is where I remember the refrain. Half of the values which fill array R are notes, and the other half are durations or time delays.

Lines 90-170 store the chorus notes and durations in R. Lines 220-240 replay the refrain.

But follow closely this sequence of events. Line 200 says something new: GOSUB 500. GOSUB stands for GO TO SUBROUTINE. (Other computer languages call them subprograms or procedures). GOSUB 500 works the same as GOTO 500 except that, should the computer encounter a RETURN statement after Line 500, it will return and execute Line 220 next. SUB 500 is where the VIC executes unless you want to play the refrain, at which point you hit F1. Line 520 RETURNs you, and Lines 220-240 play the refrain. Then we revert to Line 210, and 210 GOSUBs us back to 500.

Try some of these experiments with this program: change the 8 in Line 230 to 4, or to 15. (Any higher or lower number will do). You could increase the number of notes in the refrain by changing Lines 10, 170, and 220. For example, for a refrain of ten notes use:

```
 10  R(20)
170  J < = 20
220  J = 0 TO 18
```

Here's another variation on this playback piece:

```
115  IF (ASC(N$) < 49) OR (ASC(N$) > 57) THEN 100
515  IF M = 134 THEN POKE 650,128
516  IF M = 135 THEN POKE 650,0
525  IF (M < 49) OR (M > 57) THEN 500
543  FOR T = 1 TO 100 : NEXT
545  POKE S1,( - ((PEEK(650))/128) + 1)*A(M)
```

Lines 115 and 525 assure that, were we to hit any other key besides the numbers 1 to 9, the program wouldn't CRASH!! (A crash occurs when VIC encounters something wrong: we get an error message and the program dies.) The ASC values of 1 through 9 are conveniently within the range 49 to 57, so any other value sends us back to the previous GET.

Lines 515 and 516 define the function keys F3 and F5; if F3 is hit, all keys hit thereafter will repeat until F5 is hit, at which point we return to normal.

Line 543 is a short time delay.

That long expression in Line 545 equals A(M) in normal (no repeating keys) mode, and 0 in rreeppeeaatt mode!!! PEEK(650) returns the value stored in address 650. We'll talk more about PEEK in Chapter 12.

Remember: if you end this program while the computer's in repeat mode, POKE 650 back to 0, or your keys will play funny tricks on you.

11 Microsurgery

Until now we've been working with BASIC number variables, POKE-ing numbers into memory address registers to create screen images. In this chapter we'll consider **string variables** and build an odd Hexmas tree out of one particular string. The BASIC language has powerful functions relating to number variables. Now we will teach the VIC to use another feature: the string variable.

A string is an ordered collection of characters or numbers. This very sentence, including the spaces between the words, is a string. A string variable is a symbol that can stand for a piece of text in the same way that a number variable stands for a number. The only difference in notation is that we put a dollar sign at the end of the variable symbol so the computer knows what to expect.

There is considerable similarity of syntax in statements for number and string variables. We can even add two strings together, but there is a slight twist.

Clear the computer with a NEW, and type the following example of string addition.

```
10  A$ = "WHY"
20  B$ = "ME?"
30  C$ = A$+B$
40  PRINT C$
```

When you run this program you can see that A$ and B$ are combined to make a new, larger string C$. Combining strings in this way is called concatenation. You will note that as the program stands, A$ and B$ are bunched together. It would be nice to put a space in C$ to separate the two text blocks "WHY" and "ME?". There are several ways to do this. The easiest would be to use the screen editor to insert a space after the Y in "WHY". But we will use another method to show you that multiples of nothing make perfectly good strings. Add these lines to your program:

```
15  Z$ = " "
20  C$ = A$+Z$+B$
30  PRINT "SPOCK HERE; NOTHING ADDED TO C$"
40  PRINT "C$ IS NOW  ";C$
```

You can see that in Line 40 we have used the same syntax for PRINTing with strings as we used for numbers.

Go ahead—RUN the program.

Actually, it is not "nothing" that is added, but spaces with nothing in them. No space at all is called...

THE EMPTY STRING

The empty string is the zero of string arithmetic. We place it in a line by entering two quotation marks with no intervening space. The classic use of the empty string is with the GET statement. The GET command sets a variable equal to the last character pressed on the keyboard.

Clear the computer with a NEW and enter:

```
10  GET A$ : IF A$ = "" THEN 10
20  PRINT "THE KEY PRESSED WAS  ";A$
30  GOTO 10
```

This program prints the character that belongs to the last key pressed—with a few exceptions.

This simple loop is the foundation for all kinds of programs. Line 10 is the main consideration; if you haven't pressed a key, A$ is empty so the IF/THEN statement returns the computer to Line 10 to keep trying for a non-empty string. If you have hit a key, the corresponding character is assigned to the string variable A$.

WARNING...WARNING

There is a crucial difference between a number considered as a numerical value and the symbol for that number. Consider the following program:

```
10  A$ = "5"
```

```
20  B$ = "4"
30  C$ = A$+B$
```

This program will not add the value four to the value five. It will concatenate the two symbols "5" and "4" to make a new string "54." There are ways to convert numbers to strings and vice versa using BASIC. You may remember the VAL function which was used in the simple piano program in your VIC manual.

There's another function for "going the other way": STR$. For information on these functions, see your USER'S GUIDE.

STRING FUNCTIONS

There are a number of useful resources in BASIC for handling strings. We can find a string's length, convert a number to a string; we can even do surgery.

Convention refers to the symbols that go inside the parentheses as "arguments." For example, in Line 10 of the program below, the string "FFF" is the argument of the function ASC. X$, when it occurs as an argument, stands for any valid string to which you might want to apply the function.

ASC(X$)

This function takes the first character of whatever string is inside the parentheses and returns the ASCII code number of that character. There is a table of ASCII codes in the Appendix.

```
10  PRINT ASC("FFF")
15  F$ = "20"
20  X = ASC(F$) : PRINT X
30  Y = ASC("B") : PRINT"ASC('B') IS";Y
```

These are examples of the syntax of the ASC function.

CHR$(Y)

This function is the opposite of ASC. It takes a number between 0 and 255 and returns the character or key that is coded by that number. RUN the following program to verify this. (Don't forget to reNEW first.)

```
10  PRINT "A = ";CHR$(65)
20  Y$ = CHR$(89)
```

```
30  Y = ASC(Y$)
40  PRINT "THE ASCII CODE FOR ";Y$;" IS";Y
```

Use the screen editor to change that 89 in Line 20 to any other number in the range 0 to 255. Or, better yet, replace the 89 with a variable; put the above four lines in a FOR/NEXT loop, and check all 256 ASCII codes in one swoop. Don't forget the [CTRL] key, which slows the VIC down.

LEFT$(X$,Y)

This is a two argument function. That means you have to give the function two values with which to work. The X$ is a place for a string. Where the Y is, you put an integer. This function takes the first Y letters, starting from the left, out of the string X$. An example speaks louder than words.

```
10  Q$ = "AARDVARK"
20  A$ = LEFT$(Q$,4)
30  PRINT "THE FIRST 4 LETTERS OF AARDVARK ARE ";A$
```

LEN(X$)

This function takes the string inside the parentheses, and returns with its length: that is, the number of characters in the string.

```
10  D$ = "12345"
15  PRINT "THE STRING D$ =  ";D$
20  L = LEN(D$)
30  PRINT "THE LENGTH OF THE STRING D$ IS";L
```

MID$(X$,Y,Z)

Here we have a three-argument function. This function returns a substring of X$, starting from position Y and containing the next Z characters. Remember that a space counts as one position. You have to be a bit careful with the values of Y and Z. If they don't fall within the length of X$ funny things can happen. Again action speaks louder than cold type.

```
10  MUD$ = "THE MIDDLE OF"
15  PRINT "MUD$ =  ";MUD$
20  M$ = MID$(MUD$,5,6)
```

```
    3Ø  PRINT MUD$;" MUD$ IS ";M$
```

Line 3Ø is a little terse but if you work it out it says:

```
  THE MIDDLE OF MUD$ IS MIDDLE
```

RIGHT$(X$,Y)

This is a two-argument function that is symmetric to LEFT$. It returns the rightmost Y characters.

```
    1Ø  KK$ = "I AM ALWAYS WRONG"
    2Ø  PRINT "KK$ IS EQUAL TO ";KK$
    3Ø  R$ = RIGHT$(KK$,5)
    4Ø  PRINT "RIGHT$(KK$,5) IS ";R$
```

ODD FORESTRY

These are the instruments: now we are ready for surgery. The string that will go under the knife is "13579BDF." This is not a license plate number, it is a string of all the odd, single digit, base 16 numbers. For example, B is base 16 for 11. Base 16 is very important to languages that are closer than BASIC to the internal organization of the computer; it is also called Hex notation. Hence, our distinguished string consists of odd Hex numbers. Just the thing for odd Hexmas trees!

This program uses loops, the TAB function and a little algebra to print out a Christmas tree shape on the screen. We use the MID$ function to excise random choices from our string of odd Hex numbers.

We start by setting up the screen. The special symbol in Line 3Ø is the result of pressing [SHIFT]:[CLR/HOME].

```
    1Ø  REM**IT IS ODD**
    2Ø  REM***************
    3Ø  PRINT "♥"
    4Ø  POKE 36879,9Ø
    5Ø  A$ = "13579BCF" : COP = Ø
    6Ø  FOR Q = 1 TO 12 STEP 2
    7Ø  COP = COP+1
    8Ø  FOR J = 1 TO Q
    9Ø  D$ = "*"
   1ØØ  PRINT TAB(1Ø-COP+J) D$;
   12Ø  NEXT
```

```
13Ø  PRINT
14Ø  NEXT
```

If you run this program as it stands, it will print a handsome triangle of asterisks. The following lines will define the trunk of the tree. The two special symbols in Line 15Ø are obtained by pressing the [SHIFT]:[CRSR ↕]:and [CTRL]:[1] combinations in quote mode. The first key combination moves the cursor up one line to counteract the very last PRINT in the loop (Line 13Ø). The second key combination turns the character color to black.

```
15Ø  PRINT "●□"
16Ø  FOR P = 1 TO 6
17Ø  FOR Y = 1 TO 3
18Ø  PRINT TAB(9) D$;
19Ø  NEXT
2ØØ  PRINT
21Ø  NEXT
```

This will print a tree. Now for the magic—add the following.

```
22Ø  END
499  REM**GET A HEX**
5ØØ  X = INT(RND(1)*8)+1
51Ø  D$ = MID$(A$,X,1)
52Ø  RETURN
```

RUN the program. There is still no difference. That is because the subroutine is off floating in the sky and the program has no way to access it. Add:

```
 95  GOSUB 5ØØ
175  GOSUB 5ØØ
```

There you go—an odd Hexmas tree! See if you can figure out how to put a star at the top.

12 Game Making

So far we have been talking mostly about programming in color, programming in music, or programming in general. Our hope is that, with enough programming experience, you're now writing programs for the chores or diversions you envisioned when you bought the VIC. In this chapter we're going to design a simple game program. It's about 25 lines, and after you type it and RUN it, we'll discuss it. However, first we should talk about program design.

Almost any game tape or cartridge you buy commercially is written in machine or assembly language. Writing a game (or any program) in machine language is slow and exacting work. Assembly language is a bit easier, as it has a set of mnemonics for the machine code commands. One BASIC statement might require ten commands in assembly, but machine code is executed far faster than BASIC so the net result is a significant increase in speed.

When the VIC executes a BASIC statement, it first has to translate it into machine code commands, and this takes time. As you go from

BASIC to ASSEMBLY to MACHINE

you go

easy but slow to manageable to torture but fast.

One command we've been using which is similar to assembly is the POKE command. It's a straightforward command, and requires very little translation.

People say games have to be written in assembly, because otherwise they're too slow. This is poppycock! As long as your game programs aren't too long, or if they are long, they're as efficient as possible, speed

should not be a serious limiting factor.
Here's the program:

```
 1Ø  DEF FNR(X) = INT(RND(1)*(X+1))
 2Ø  POKE 36879,13
 3Ø  PRINT "♥"
 4Ø  HP = FNR(5Ø5) : P = FNR(14)
 5Ø  CP = FNR(5Ø5)
 6Ø  IF HP = CP THEN 5Ø
 7Ø  POKE 768Ø+HP,83
 8Ø  POKE 384ØØ+HP,2
 9Ø  GOSUB 5ØØ
1ØØ  SC = TI
11Ø  GET A$ : IF A$ = "" THEN 11Ø
12Ø  A = ASC(A$)
13Ø  IF A = 29 THEN CP = CP+1
14Ø  IF A = 157 THEN CP = CP-1
15Ø  IF A = 145 THEN CP = CP-22
16Ø  IF A = 17 THEN CP = CP+22
17Ø  IF CP<Ø OR CP>5Ø5 THEN CP = 1
18Ø  GOSUB 5ØØ
19Ø  IF CP = HP THEN 6ØØ
2ØØ  GOTO 11Ø
5ØØ  POKE 768Ø+CP,81+P
51Ø  POKE 384ØØ+CP,1
52Ø  RETURN
6ØØ  PRINT "YOU TOOK"; (TI-SC)/6Ø; "SECONDS"
61Ø  PRINT "PLAY AGAIN?"
62Ø  INPUT R$
63Ø  IF R$ = "Y" THEN 3Ø
64Ø  POKE 36879,27
```

After you RUN, a response of "Y" will let you play again (sometimes!). Let's inspect this program.

DOCUMENTATION

Line 1Ø is a little different. It defines the function FNR(X). Whenever I use FNR(X) in the rest of the program, it takes the value of a random integer (whole number) between Ø and X, including both Ø and X. DEF

FN is useful for functions of one variable which are used more than once in a program.

Line 2Ø is new, too. In the USER'S GUIDE you'll find codes for screen/border color combinations. Screen color is controlled by the value in memory register 36879, which you can change. The code for the normal screen color is 27, which is a white screen and a cyan border. In this case, we're changing our color code to 13, which is a black screen and a green border.

HP, home position, is where the heart is. CP, cursor position, is where the cursor starts. P lets the cursor symbol be a random graphics symbol between 81 (ball) and 95 (triangle). Line 6Ø makes sure the cursor doesn't start at home.

Line 1ØØ uses the internal clock (TI) of the VIC. TI is a variable which, when you turn on the VIC, equals zero. After 1/6Ø of a second TI = 1, and TI increases by 1 with every subsequent 1/6Ø of a second.

So here we initialize the game and start the timer by storing the current value of the system jiffy clock (TI) in SC. Then, in Line 6ØØ, the elapsed time is discerned in seconds by using the algebraic formula: (TI − SC)/6Ø. This makes the game much more interesting to play repeatedly; it gives you something to better.

Lines 11Ø-16Ø are pretty standard after what we did in MUSIC 2. They identify hitting the cursor keys with moving the cursor in the usual manner.

Line 17Ø ensures that we don't try to POKE a CP that's off the screen.

TRANSIENT BUGS

This program was originally designed to help a person practice using the cursor keys. Now that we're using the program as a game, we'll want to modify it so that it's easier to move the cursor left or up. (Hitting shift to move slows down the play.)

First, however, we'll deal with the problem that when you type "Y" to play again, sometimes you don't get to play again. Why is this happening? VIC reads the "Y" and any other symbols appearing on the line. The reading of symbols unequal to "Y" causes the program to end. We can modify Line 63Ø to compare only the first character in our response:

```
63Ø  IF LEFT$(R$,1) = "Y" THEN 3Ø
```

LEFT$(R$,1) represents the first character on the left of our response line. Now we'll get to play again a lot more often. However, this still doesn't cover all the possibilities, and it's still possible to type "Y" and not get to play again. I'll let you figure out why, and fix it if you want to.

REPEAT THAT PLEASE

Let's solve the other problem that's slowing our game. We don't want to [SHIFT] in order to move left or up, so let's change Lines 130-160 to reference keys that are easier to use. I'll use "Y" for up (Line 150), "B" for down (Line 160), "G" for left (Line 140) and "H" for right (Line 130). However, one nice thing about the cursor keys (and the space bar) is that if you hold down the key, it repeats its function until you release it. Other keys, like "Y" and "G" don't repeat, so if we simply change to the other keys (by changing the ASC codes), you'd have to hit the "Y" repeatedly in order for it to move up more than one space. Of course, there are ways to fix this, or I wouldn't have led you down this primrose path. One particularly useful way involves a new function PEEK(X).

PEEK

If I POKE 36878,15, I'm turning the music volume to its highest setting, 15. If I then write:

```
PRINT PEEK(36878)
```

I get a response of 15. The PEEK command does just what its name implies, it peeks in the specified location (36878 in this example) and returns the value stored there. Location 36878 in memory controls the music volume; location 197 in memory contains the code number for the current key being held down. These code numbers are not the same as the POKE or the ASC character codes. A complete list is in the VIC PROGRAMMER'S REFERENCE GUIDE. Modify Line 110 to read:

```
110  A = PEEK(197) : IF A = 64 THEN 110
```

64 is the code for no key being held down. Now delete Line 120 and change Lines 130-160 to:

```
130  IF A = 43 THEN CP = CP+1
140  IF A = 19 THEN CP = CP-1
150  IF A = 11 THEN CP = CP-22
160  IF A = 35 THEN CP = CP+22
```

The new control keys are H, Y, G and B. Try RUNning the program with these modifications and you'll find we still have a problem. The letters we used in the game are printed on the same line as our response to "PLAY AGAIN?". We certainly don't want that to happen. Here's the fix:

```
615 POKE 198,Ø
```

Address (location) 198 holds a number representing the number of characters in the keyboard buffer (a buffer is a temporary storage compartment). All those G's, Y's, B's, and H's get stored in the buffer while we're playing the game, and then print out right after Line 62Ø executes. So if we POKE 198,Ø just before Line 62Ø, in effect we're clearing the garbage out of the buffer.

ANOTHER WAY

There's a location in VIC (65Ø) that controls whether all the keys repeat. If the location contains 128, all the keys repeat; normal mode requires a Ø in location 65Ø which means only certain keys will repeat. So we could have kept Line 12Ø in our program, and added the following:

```
1Ø5 POKE 65Ø,128
19Ø IF CP ≐ HP THEN POKE 65Ø,Ø : GOTO 6ØØ
615 POKE 198,Ø
```

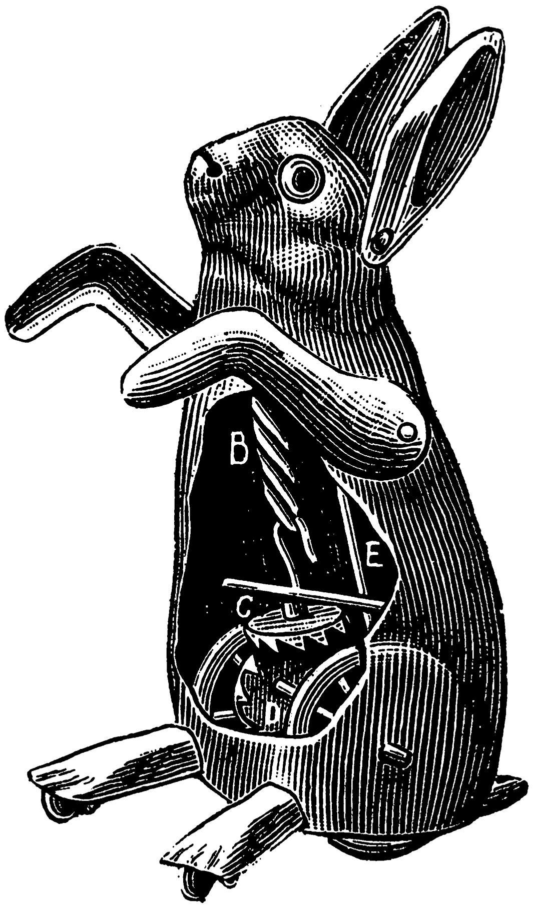
Of course, you want Lines 13Ø-16Ø to compare the variable A with ASC codes of Y, G, B, and H (or any keys—it's your choice).

COMMENTS

This game is, at best, mildly diverting; however, we did have reasons for including it. (1) it's fairly simple, and (2) it should give you ideas for other games. Once you've had a little practice writing game programs, you'll realize that the major difficulty is in incorporating sound effects. It's really too bad the VIC executes only one command at a time. If we had two computers in our VIC instead of one, we could simultaneously execute action and sound effect commands.

It seems, in programming, that as soon as you've figured out the best way to do something, somebody does it better. This is both bad and good; bad because you didn't think of it, but good because at least now you know a better way. As you get more and more involved in programming, you'll discover that many of the computer magazines on sale these days will give you programming tips and even whole programs.

The magazines that Commodore promotes are excellent sources for programs, and I recommend you read them. In addition you will find the VIC-20 PROGRAMMER'S REFERENCE GUIDE an invaluable resource. Look for it at your local Commodore dealer.



13 Moving Color

Now we'll build a program that uses the screen character map to provide animation for our graphics.

```
10  A = 5
20  PRINT "♥";
30  FOR H = 0 TO (A-1)
40  POKE 7680+H,81
50  POKE 38400+H,2
60  FOR T = 1 TO 100 : NEXT
70  NEXT
```

The VIC, as noted in Chapter 6, has a character screen map that starts at address 7680 and ends at address 8185. Again, those with memory expansion will have to change Lines 40 and 50 to conform to their screen memory maps.

If you RUN this first part of the program, you'll see we've displayed a bug 5 units long. Line 10 will eventually be an INPUT so we can control the length of our bug. Note that we usually put Line 20 first, but in this case we expect to input information, so we'll want to clear the screen after any input. 81 is the POKE code for the symbol printed on the right side of the Q key, and 2 is the POKE code for red. You will find a list of all the character codes in the Appendix.

LOCOMOTION

Now that we've drawn our insect, let's get it moving!

```
80  FOR J = 0 TO (21-A)
```

```
 9Ø  POKE 384ØØ+J,1
1ØØ  POKE 768Ø+J+A,81
11Ø  POKE 384ØØ+J+A,2
12Ø  FOR T = 1 TO 1ØØ : NEXT
13Ø  NEXT
```

The way it's written, first we turn off the symbol in the first position (upper left corner) of the screen by POKEing its color address (which is 384ØØ the first time through the J loop) to 1 (Line 9Ø). Then we POKE a circle (bug part) into the first blank space, which, the first time through the loop, is 768Ø+A. Line 11Ø sets the color to red in the same screen location. Recall that the screen character addresses start with 768Ø, which corresponds to a screen color address of 384ØØ. As we go through the values of J=Ø to (21−A), we're turning off the end of the bug and turning on another front part each time. Note that the value 1ØØ in Lines 6Ø and 12Ø can be changed to adjust the speed. I happen to think the bug moves nicely at this speed, but you can vary the speed if you want.

To keep the bug moving we need to loop:

```
14Ø  GOTO 2Ø
```

and don't forget to amend Line 1Ø to: 1Ø INPUT A.

Unfortunately, this clears the bug at the right side a little too quickly, but it's the best we can do at this stage.

This program might just as well run the bug down the first column as across the first row.

```
 4Ø  POKE 768Ø+(H*22),81
 5Ø  POKE 384ØØ+(H*22),2
 9Ø  POKE 384ØØ+(J*22),1
1ØØ  POKE 768Ø+((J+A)*22),81
11Ø  POKE 384ØØ+((J+A)*22),2
```

All that I've done is to multiply the expressions added to 768Ø or 384ØØ by 22, the length of the screen, so that each time the statements are executed the cursor counts all the way across one row and ends up just below the last screen position.

We'll return to this little bug in a later chapter and in the next chapter we'll develop another little bug that goes for a walk.

14 Number Theory 2

Back in Chapter 9 your VIC added up the first 1ØØØ odd numbers. In that case we simply told the computer to increment by two to reach another odd number to add into the sum. Now we pose the problem: how do we teach your VIC to figure out whether any given integer is even or odd? This innocent problem has a number of solutions. We will use a variant of a solution to this problem to examine a classic question in probability—the Random Walk. We will write a program that will calculate the average time it takes a drunken bug reach the edge of the screen. For the computer to realize that the little fellow has reached the edge it has to solve a simple algebraic equation very similar to the one that solves...

EVEN OR ODD?

The computer is the detective but we need to supply the clue. There are a number of ways to do this. We could ask the computer to do an exhaustive search of all even numbers:

```
 1 W = Ø
 5 REM***I DO IT***
1Ø INPUT"NUMBER PLEEYUZ";N
2Ø W = W+2
3Ø IF N = W THEN 6Ø
4Ø IF N = W-1 THEN 8Ø
5Ø GOTO 2Ø
6Ø PRINT" IT IS EVEN"
7Ø END
```

```
80 PRINT" THAT'S ODD"
```

The VIC will contentedly keep doing this all afternoon if need be. We want something that will run a bit faster when fed a large number. What we need is some property that distinguishes between even and odd numbers. Examine the following program.

```
10 REM*DO IT FASTER*
20 INPUT "INPUT NUMBER";ZZ
30 IF ZZ/2 = INT(ZZ/2) THEN 50
40 PRINT "THE NUMBER ";ZZ;" IS ODD" : END
50 PRINT ZZ;"IS EVEN"
```

The key line is 30. When you divide an even number by two the result is an integer. When you divide an odd number by two there is a fractional part of .5. Look at these examples.

```
  8/2 =  4
128/2 = 64
 15/2 =  7.5
  7/2 =  3.5
111/2 = 55.5
```

This is where the INT function goes to work by picking off the INTeger part of number. In other words, it will return the next lower integer. Examples:

```
INT(33.12345) = 33
INT(0.999999) = 0
INT(-22.00234) = -23
```

Notice how the VIC handles the negative number in the third example, it takes the next lower whole number.

Line 30 uses the INT function to test whether a given number meets our criteria for being even. If it does, the program jumps to Line 50 and prints out the happy news. If not, the program goes on to the next line, prints out the sad news and ENDs. Now we are ready to take a drunken bug on a...

RANDOM WALK

The rules for this model are simple. The bug starts at the center of the screen. Each 10 "seconds" the bug has to decide whether to sway back and forth for another 10 "seconds" or stagger in one of four directions. We want to keep track of two things, how many times the bug gets back to the starting point and on the average how long it takes for the poor thing to deep-six itself. The screen is a 22 by 23 grid, so, if we start our bug in the center of the screen, the bug can teeter over in 1Ø or 11 straight steps. With this information, go ahead and make a friendly estimate of walk expectancy. How long will it take the bug to reach the edge of the screen?

A BRIEF RESPITE

Before we get to the design of the main program let's remind ourselves of how to control the screen. The following test will fill the screen with red B's.

```
1  D = 768Ø : C = 384ØØ : FOR X = Ø TO 5Ø5
2  POKE D+X,2 : POKE C+X,2 : NEXT X
```

It is a good idea to pair the POKEing of a value in screen character memory to the operation of POKEing a color code in the corresponding location in screen color memory.

DESIGN

For a longer program like this it is often a good idea to budget line numbers. This will encourage modularity and it is easier to test sections separately. Here's the budget for "BUGS DILEMMA."

Line 1Ø Line 1ØØ	Inititialize
Line 11Ø Line 2ØØ	Subroutines
Line 3ØØ Line 4ØØ	Main Logic
Line 41Ø Line 5ØØ	Keep Score

We will start with the initialize section.

```
 1Ø  REM BUGS DILEMMA
 12  REM***************
 2Ø  POKE 36879,13 : REM**SCREEN COLOR
 3Ø  PRINT"♥" : REM**CLEAR SCREEN
 4Ø  CP = 252 : REM**STARTING VALUE
1ØØ  GOTO 3ØØ
```

(Recall that REMark statements aren't necessary; when you're typing this program, you may omit them if you wish.)

There is plenty of room here for future additions and modifications. This section of code sets up the screen and defines a constant that we will use to start the bug in the middle of the screen. You can test the screen by putting in a temporary Line 3ØØ that contains a STOP, and the program will then go through the initialization procedure only when you RUN it.

MAIN LOGIC SECTION

The main logic section of the program tests to see if the bug has reached the edge of the screen. If the poor fellow has deep-sixed, the computation jumps to the Keep Score section. Otherwise an X is printed at the current position. A subroutine call is then made to determine (randomly) where the rascal will stagger next. During this call a little square foot print shows where the bug is at the moment. When the call is finished the program loops back to Line 3ØØ. The only tricky bit is the algebra that determines when the bug has taken one step too many. You can experiment with doing all this in different orders. This sequence does a nice job of keeping our bug hopping.

```
298  REM***************
299  REM****LOGIC****
3ØØ  IF INT(CP/22) = CP/22 THEN 4ØØ
31Ø  IF INT((CP+1)/22) = (CP+1)/22 THEN 4ØØ
32Ø  IF CP < 22 OR CP > 484 THEN 4ØØ
33Ø  POKE CP+768Ø,86 : POKE CP+384ØØ,1
34Ø  GOSUB 11Ø
39Ø  GOTO 3ØØ
4ØØ  PRINT "BYE BYE BUG!"
```

The trick in Lines 3ØØ and 31Ø is similar to the odd or even criteria.

We are testing to see if CP is a multiple of 22 or if CP + 1 is a multiple of 22. For two peanuts and a used wrapper, figure out how (and why) these two lines work without looking at the explanation at the end of the chapter. Meanwhile, back at the subroutine:

```
109  REM**TAKE A STEP**
110  POKE CP+7680,108 : POKE CP+38400,7
120  L = INT(RND(1)*5)
130  IF L = 0 THEN CP = CP+1
132  IF L = 1 THEN CP = CP-1
134  IF L = 2 THEN CP = CP+22
136  IF L = 3 THEN CP = CP-22
150  RETURN
```

This is a straightforward use of conditional logic. The only subtlety here is what happens if the variable L is given the value 4. Since none of the conditions are met, the value of CP remains unchanged and the bug sways back and forth for another time period unable to make up its mind. The program will (barring your own bug) RUN now, so go ahead and see what it looks like.

KEEPING SCORE

With no delay loop the bug certainly hops around! We will now insert into the program a few more lines that will help us get a more precise measurement of what is happening as this program runs. Insert the following lines:

```
360  COP = COP + 1
365  IF CP = 252 THEN ET = ET + 1
```

and change Line 40 to initialize COP and ET:

```
 40  CP = 252 : COP = 0 : ET = 0
```

Line 360 is a simple counter that increments every time the bug has to make a decision. If we have assigned 10 "seconds" per step then the total simulated time in "minutes" is COP/6. Now we can add the following lines.

```
399  REM**KEEP SCORE**
410  PRINT "THE BUG TOOK";COP/6;"MINUTES";
420  PRINT " TO REACH THE EDGE"
430  PRINT "THE NUMBER OF TIMES RETURNED HOME";
```

```
440 PRINT " IS";ET
```

ADDENDUM

There is still the puzzle of the lines that check whether the bug has actually reached the edge after any given step. They are:

```
300 IF INT(CP/22) = CP/22 THEN 400
310 IF INT((CP+1)/22) = (CP+1)/22 THEN 400
```

These lines check to see if the bug has wandered to the last column on either side of the screen. Every value of CP on the leftmost column shares a common algebraic property: it is evenly divisible by 22. Likewise, the rightmost column of screen locations have values of CP that when 1 is added to them the sum is evenly divisible by 22. For more clues, see the screen memory map in the Appendix.

15 Advanced Color

The POKE statement is a versatile tool to weave into your programs. Now we will use it to finish our excursion in color printing with a program that moves our centipede across the screen, back across the screen one line below, and repeats the action for a specified number of lines. While I won't try to explain every detail of every line, hopefully you'll see that the important lines (the ones with POKE statements) are tied together by an overall logic. With a little thinking and a little serendipity, you can edit the basic template we develop here to get a feel for screen control.

A COUPLE OF IMPORTANT THINGS TO NOTICE

1) The program is short—the whole thing fits on the screen when you LIST it. Or at least almost the whole thing; it all depends on how many spaces you include when you type the program. In execution, VIC ignores all those extra spaces anyway.
2) The program is structured in the same way as our other programs:
   - A) INPUT statements
   - B) Clear screen
   - C) FOR loops (main body)
   - D) GOTO step A (big loop if wanted)

Having a pattern to work from makes programming much easier. You may wish to work out your own style after you get a feel for writing programs. All programmers tend to develop a pattern which suits their way of thinking. Keeping your style simple and easy to work with will help you in the long run. It's nice to have short programs—they're easier to understand or explain to a friend. When we have to write longer programs, we'll modularize them. Each part of the program that does one specific thing will be as short as possible and two parts doing two different things will be separated so we can read and understand one part at a time. Modularization (bet you can't say it five times quickly) is a cornerstone of **Structured Programming.**

Here's the program that will get us started:

```
10  INPUT A
20  PRINT"♥";
30  FOR P = 1 TO 20
40  FL = INT( ( ( (P/2) - INT(P/2) ) *2) +.5)
50  SN = 1-2*FL
60  FOR J = 0 TO 21
70  CH = (22*P)+(21*FL)+(J*SN)
80  IF J > = A THEN POKE 7680 + (22*P) + ((J-A)*SN) + (21*FL),32
90  IF J<A THEN POKE 7680 + (22*(P-1)) + ((A-J)*SN) +
    (21*FL) - SN,32
100 POKE 7680+CH,81
110 POKE 38400+CH,2
120 FOR T = 1 TO 10 : NEXT
130 NEXT : NEXT
```

Don't include all those extra spaces in Lines 80 and 90 (we've included them for clarity) or your lines won't fit into the VIC. Hopefully you've made no keying errors, so go ahead and RUN it—input 5 for A.

THE LOGIC (OR HOW I FIGURED IT OUT)

Our centipede is 5 bug units long, which means that the variable A has the value 5. To form the centipede, Lines 1ØØ and 11Ø loop until the 5 bug units appear on the screen. Then they continue to form a new bug unit while Line 8Ø follows behind erasing an old bug unit. Thus, the centipede appears to be darting across the screen. Note that Line 8Ø erases only the units that are on the same line as the bug units being turned on. Line 9Ø erases the bug units on the line above when we're "turning the corner." To program the bug's movements, I first set up a table of which addresses I wanted on or off:

TABLE 1

P	Bug	Turn on	J>=A Turn off behind	J<A Turn off above
even	→	J	J−A	(A−J)−1
odd	←	21−J	21+A−J	21+J+1−A

In Lines 1ØØ to 11Ø I'm POKEing 768Ø (and 384ØØ) + 22*P (P says which line of the screen I'm on) + either J (going left to right) or 21 − J (going right to left). When I'm erasing behind the bug (Line 8Ø), I'm adding one of the two values indicated in the table to 768Ø + (22*P), depending, again, on which way the bug is moving. When I'm erasing above the bug (Line 9Ø), I'm adding to 768Ø + 22*(P−1). Since P is the row or line of the screen I'm on, P − 1 is the line above. Note the different way I'm erasing: instead of POKEing the color to white (or the same as the background color), I'm POKEing the symbol which is represented by the number 32, which is a space or blank. Either way of erasing is perfectly acceptable.

Try putting J values in the above table for, say, Row 4 (P=4). On even rows (P even) we're going left to right, and on odd rows (P odd) we're going right to left. Imagine which bug parts we'd be turning on and off as we proceed left to right along the 4th row. For instance, we'd be turning on 768Ø + (22*4) + J, as J goes from Ø to 21.

Once you've convinced yourself that the expressions in the table work, what's the next step? Well, I could have two FOR/NEXT loops, one for P even and one for P odd. That certainly would work, if the computer could figure out whether a number is even or odd. Line 4Ø will program the computer to do just that.

TABLE 2

P	P/2	INT(P/2)	P/2 – INT(P/2)
1	1/2 (?)	Ø	1/2
2	1 (?)	1	Ø
3	3/2 (?)	1	1/2
4	2 (?)	2	Ø
5	5/2 (?)	2	1/2

How did I figure out that P/2 – INT(P/2) would alternate for P even and odd? I guessed! And I was lucky. It worked. I wasn't guessing randomly: I had in mind the way the INT function works, so let's call it an educated guess. But what's the rest of Line 4Ø doing in there? Well, note the question marks in the second column of the table. Whenever you divide on a computer, there's a chance your computed result won't be the same as the real answer. This is because of the way computers divide numbers. I want FL to alternate between 2 values I'm sure of; so I multiply 1/2 (or Ø) by 2, add 1/2, then INT it. This will definitely give me Ø when P is even, and 1 when P is odd. Then I decided to invent SN, (Line 5Ø), which is +1 when FL=Ø or P is even, and –1 when FL=1 or P is odd. Now compare the values in Table 1 with the gobbledygook in Line 1ØØ:

```
1ØØ  POKE 768Ø+CH,81   where
 7Ø  CH = (22*P) + (21*FL) + (J*SN)
```

When P is even (moving left to right) I want CH to equal 22*P + J; when P is odd (moving right to left) I want CH to equal 22*P + 21 – J:

TABLE 3

P	BUG	FL	SN	TURN ON
even	→	Ø	1	+J
odd	←	1	–1	+21 – J

21*FL + J*SN gives me what I want! When P is odd, FL=1, SN=–1, and (21*FL) + (J*SN) = 21 – J; when P is even, FL=Ø, SN=1, and (21*FL) + (J*SN) = J. I used the same logic for the steps in Lines 8Ø and 9Ø. Now I don't need two different FOR loops. Compare (J – A)*SN + (21*FL) (Line 8Ø) with J – A for P even and 21 + A – J for P odd. Do the same for Line 9Ø and you will see that I'm just letting the VIC figure out the proper addresses from three general formulas.

My time delay (Line 12Ø) may seem short but the VIC takes so long to execute Lines 7Ø to 11Ø that 1Ø works well. You may wish to lengthen it so you can watch each step execute, and then get rid of it all together for the fastest motion possible.

Of course, I didn't just sit down and write out the program off the top of my head. First, I started with two FOR loops (actually three, but here we're just talking about the loops within the P loop); then I realized I needed FL, so the computer could go to the proper loop depending on whether P was odd or even. Once I had FL, I noticed that my turn-on POKEs in the two different loops could be combined if I had SN, so I invented SN. From there on, it was easy. The tricks to successful programming are: 1) take your time; 2) imagine or write down (in English) what you're trying to do; 3) find an easy solution and try to refine it; and, 4) give up temporarily when your head starts spinning!

16 User Defined Characters

There are many internal features of the VIC which are too complicated to pursue in an introductory manual such as this. One such feature, which we will explore, permits the user to create his or her own characters. This isn't covered in the USER'S GUIDE, but designing characters is fun and fairly easy to do, so let's get started with the following program.

```
10  POKE 51,0 : POKE 52,28
20  POKE 55,0 : POKE 56,28
30  CLR
40  FOR I = 0 TO 511
50  POKE 7168 + I,PEEK(32768 + I)
60  NEXT
```

Adresses 51-52 and 55-56 are pointers for limits of the BASIC program storage area. If you combine the numbers POKEd into addresses 51 and 52 with the formula (28*256) + 0, you should get 7168. This is the start of the memory addresses where we'll store our special character set. Normally the numbers in there are 0 and 30, which, by the formula (30*256) + 0, gives us 7680 for the top of the BASIC program storage space (directly under the screen character map).

If at this point you don't have a memory map of the VIC in one of your computing manuals, by all means buy a PROGRAMMER'S REFERENCE GUIDE or any advanced VIC text. You'll be able to get a good feel for where the VIC stores its characters by messing around with the first two programs of this chapter, but for a full understanding (if that's possible!), you need at least one more reference. As an added benefit, you'll find things in the full memory map that almost nobody seems to be aware of (for instance, that's where I found POKE 650,128 which I used in the Gamemaking chapter).

The CLR in Line 3Ø tells the VIC to clear any variables which may be in the memory space we're going to "steal" and use as storage space for our special characters.

In Lines 4Ø to 6Ø we're copying the normal text characters into our freshly created character storage area. Each character takes up eight memory locations, so 512 addresses (don't forget Ø) gives us 64 characters.

Type the above program, but before RUNning it, type on the next line:

```
PRINT FRE(9)
```

The FRE function shows on the screen the number of bytes of BASIC program storage space available. When you turn on your VIC, the screen shows "3583 bytes free." FRE may be followed by any number; I use (9) because it's easy to type. (You'll see what I mean when you do it.) After typing the program,

```
PRINT FRE(9)
```

should return

```
3491
```

indicating that the program took up 92 bytes of memory. Now RUN. PRINT FRE(9) again: now there's 2972 bytes left. So another 519 bytes are gone. Type NEW, and hit [RUN/STOP]:[RESTORE]. PRINT FRE(9) now gives us 3Ø69 bytes. The program is gone (hope you SAVEd it first), but our new character memory space is still there. The only way to make it go away is to turn off the VIC! (Or rePOKE back to normal.)

Now let's put some new characters into our special space.

```
1Ø  FOR K = Ø TO 8
2Ø  FOR J = Ø TO 7
3Ø  READ A
4Ø  POKE 7168 + J + (K*8),A
5Ø  NEXT : NEXT
```

Our new characters are going in starting at location 7168. Now all we need to know is how to create a symbol.

Each time we type any character on the screen, we're actually printing a square divided up into 64 little boxes; in each box is a light, and it's either on or off. (These little lights are called pixels.) A blank space is 64 little lights, all of them turned off. When we build a symbol, we do it one row (eight boxes) at a time. If we want all the lights in the first row to be off, then we POKE Ø; if we want all the pixels on, we

POKE 255. In between Ø and 255 are all of the other combinations of turned on or off pixels. In order to figure out the combination of on and off lights corresponding to a given number, convert the binary number, which represents the combination of off and on lights, to a decimal number, like so:

```
   Ø     Ø     Ø     Ø     1     Ø     1     1

  128    64    32    16    8     4     2     1
```

thus, lights are

```
  off   off   off   off   on    off   on    on
                          8+          2+    1 = 11
```

Once you get used to the routine of converting binary to decimal, it's a much less painstaking task then you might imagine. In the meantime, there's also a chart on page 78 which will make this process less of a mathematical chore. Now let's add our DATA statements.

```
100  DATA 24,24,24,24,24,6Ø,1Ø2,195
11Ø  DATA 6Ø,126,22Ø,248,252,254,126,6Ø
12Ø  DATA 6Ø,126,223,255,255,255,126,6Ø
13Ø  DATA 56,68,68,56,16,124,16,16
14Ø  DATA 7,3,5,12Ø,152,136,136,112
15Ø  DATA 56,84,84,24,16,16,Ø,16
16Ø  DATA 24,9Ø,6Ø,231,231,6Ø,9Ø,24
17Ø  DATA 36,1Ø2,231,24,24,231,1Ø2,36
18Ø  DATA 66,189,66,165,129,153,66,6Ø
```

RUN this. Do you want to take a look at our new symbols? Type:

```
POKE 36869,255
```

Address 36869 tells the VIC where the character set and the screen character map are stored in memory. Blocks of memory (1Ø24 addresses per block) are assigned numbers, with block Ø starting at address 32768 (where the normal character set starts) and block 8 starting at address Ø. Address 7168 (our new character set) is thus block 15, and address 768Ø (our normal screen character map) is block 15.5. The formula which gives us 255 is INT(15.5)*16+15. Normally, with the character set at block Ø and the screen character map at block 15.5, we have INT(15.5)*16+Ø = 24Ø. If you don't believe me, PRINT PEEK(36869)!

The flashing cursor will disappear because it needs reverse characters in order to work, but we didn't make our special storage space big enough to hold reverse characters. (This isn't as much of an inconvenience as you might think.) Try typing @; that's our first new symbol. Press the letters A through H to see the other new symbols. How do you like the A and B characters? In the next program, we'll make them gobble their way across the screen.

Now let's experiment with our new symbols in the following program. First hit [RUN/STOP]:[RESTORE] or POKE 36869,24Ø so we can see the cursor.

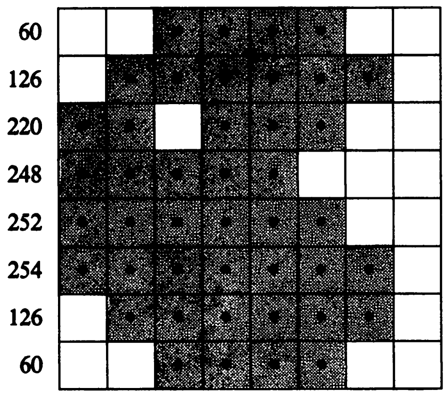
```
1Ø  PRINT "♥"
2Ø  FOR K = 1 TO 2Ø
3Ø  POKE 768Ø+K,1
4Ø  POKE 384ØØ+K,6
5Ø  FOR T = 1 TO 9Ø : NEXT
6Ø  POKE 768Ø+K,2
7Ø  FOR T = 1 TO 45 : NEXT
8Ø  POKE 384ØØ+K,1
9Ø  NEXT
```

RUN. You should see A and B flashing and moving across the top of the screen. Now POKE 36869,255 and RUN.

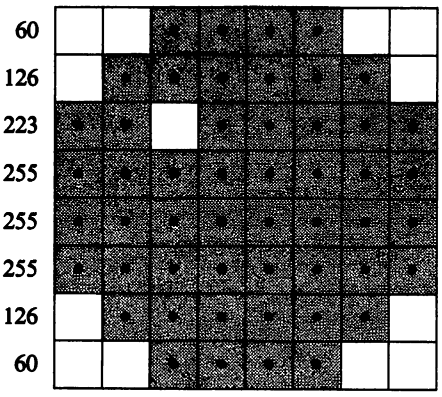
In Line 3Ø, 1 is the POKE code for what used to be an A, but is now the open munch monster. The 2 (in Line 6Ø) is the closed munch monster.

USER DEFINED CHARACTERS

Open-mouth
Munch Monster



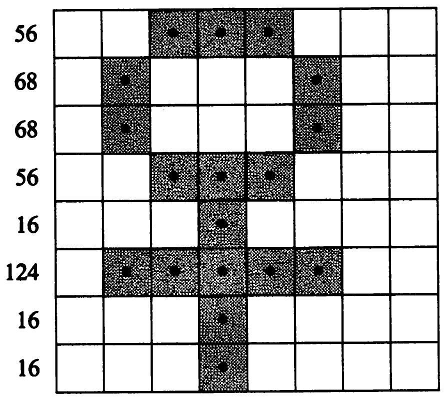
Close-mouth
Munch Monster



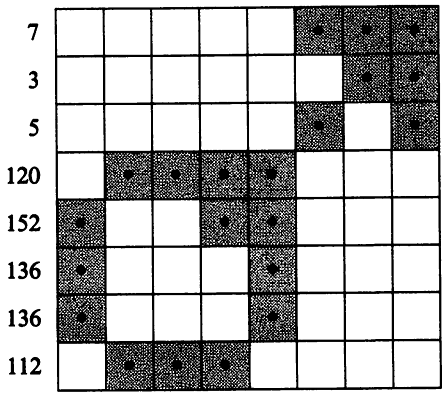
As your munch monster gobbles its way across the screen, you might start dreaming up new projects. How about adding a moving mouth to the travelling centipede? Perhaps you could give him moving feet as well.

Reviewing the various chapters of this book, now that you have an overall concept of the abilities of your VIC, will give you the tools to expand programs and invent your own. Enjoy!

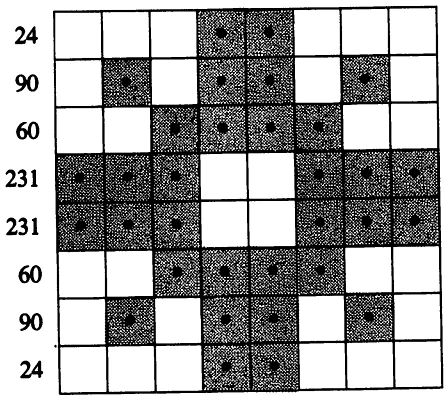
Female



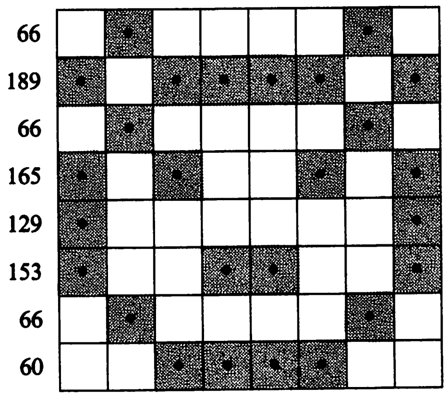
Male

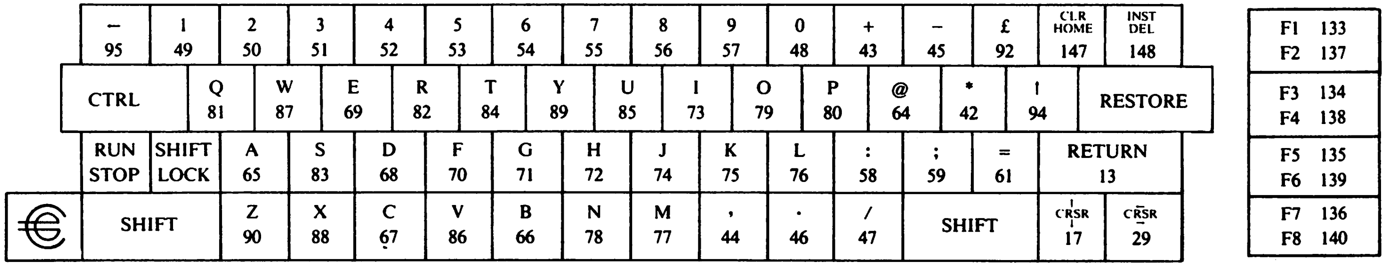


Snowflake 1



Teddy Bear Face





KEYBOARD ASCII CODES

CHARACTER DESIGN CHART

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
								0
							●	1
						●		2
						●	●	3
					●			4
					●		●	5
					●	●		6
					●	●	●	7
				●				8
				●			●	9
				●		●		10
				●		●	●	11
				●	●			12
				●	●		●	13
				●	●	●		14
				●	●	●	●	15
			●					16
			●				●	17
			●			●		18
			●			●	●	19
			●		●			20
			●		●		●	21
			●		●	●		22
			●		●	●	●	23
			●	●				24
			●	●			●	25
			●	●		●		26
			●	●		●	●	27
			●	●	●			28
			●	●	●		●	29
			●	●	●	●		30
			●	●	●	●	●	31

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
		●						32
		●					●	33
		●				●		34
		●				●	●	35
		●			●			36
		●			●		●	37
		●			●	●		38
		●			●	●	●	39
		●		●				40
		●		●			●	41
		●		●		●		42
		●		●		●	●	43
		●		●	●			44
		●		●	●		●	45
		●		●	●	●		46
		●		●	●	●	●	47
		●	●					48
		●	●				●	49
		●	●			●		50
		●	●			●	●	51
		●	●		●			52
		●	●		●		●	53
		●	●		●	●		54
		●	●		●	●	●	55
		●	●	●				56
		●	●	●			●	57
		●	●	●		●		58
		●	●	●		●	●	59
		●	●	●	●			60
		●	●	●	●		●	61
		●	●	●	●	●		62
		●	●	●	●	●	●	63

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	●							64
	●						●	65
	●					●		66
	●					●	●	67
	●				●			68
	●				●		●	69
	●				●	●		70
	●				●	●	●	71
	●			●				72
	●			●			●	73
	●			●		●		74
	●			●		●	●	75
	●			●	●			76
	●			●	●		●	77
	●			●	●	●		78
	●			●	●	●	●	79
	●		●					80
	●		●				●	81
	●		●			●		82
	●		●			●	●	83
	●		●		●			84
	●		●		●		●	85
	●		●		●	●		86
	●		●		●	●	●	87
	●		●	●				88
	●		●	●			●	89
	●		●	●		●		90
	●		●	●		●	●	91
	●		●	●	●			92
	●		●	●	●		●	93
	●		●	●	●	●		94
	●		●	●	●	●	●	95

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	●	●						96
	●	●					●	97
	●	●				●		98
	●	●				●	●	99
	●	●			●			100
	●	●			●		●	101
	●	●			●	●		102
	●	●			●	●	●	103
	●	●		●				104
	●	●		●			●	105
	●	●		●		●		106
	●	●		●		●	●	107
	●	●		●	●			108
	●	●		●	●		●	109
	●	●		●	●	●		110
	●	●		●	●	●	●	111
	●	●	●					112
	●	●	●				●	113
	●	●	●			●		114
	●	●	●			●	●	115
	●	●	●		●			116
	●	●	●		●		●	117
	●	●	●		●	●		118
	●	●	●		●	●	●	119
	●	●	●	●				120
	●	●	●	●			●	121
	●	●	●	●		●		122
	●	●	●	●		●	●	123
	●	●	●	●	●			124
	●	●	●	●	●		●	125
	●	●	●	●	●	●		126
	●	●	●	●	●	●	●	127

Note: for characters that have a dot in the 2^7 position, add 128 to the appropriate line of numbers (i.e., to form

●			●	●	●		●

add 128 to line 29 for a total of 157).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7680																						
7702																						
7724																						
7746																						
7768																						
7790																						
7812																						
7834																						
7856																						
7878																						
7900																						
7922																						
7944																						
7966																						
7988																						
8010																						
8032																						
8054																						
8076																						
8098																						
8120																						
8142																						
8164																						

Screen Character Codes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
38400																						
38422																						
38444																						
38466																						
38488																						
38510																						
38532																						
38554																						
38576																						
38598																						
38620																						
38642																						
38664																						
38686																						
38708																						
38730																						
38752																						
38774																						
38796																						
38818																						
38840																						
38862																						
38884																						

Color Codes Memory Map

Abbreviated List of Color Codes:

Code	Color
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow

TABLE OF MUSICAL NOTES

APPROX. NOTE	VALUE	APPROX. NOTE	VALUE
C	135	G	215
C#	143	G#	217
D	147	A	219
D#	151	A#	221
E	159	B	223
F	163	C	225
F#	167	C#	227
G	175	D	228
G#	179	D#	229
A	183	E	231
A#	187	F	232
B	191	F#	233
C	195	G	235
C#	199	G#	236
D	201	A	237
D#	203	A#	238
E	207	B	239
F	209	C	240
F#	212	C#	241

SPEAKER COMMANDS:	WHERE X CAN BE:	FUNCTION:
POKE 36878, X	0 to 15	sets volume
POKE 36874, X	128 to 255	plays tone
POKE 36875, X	128 to 255	plays tone
POKE 36876, X	128 to 255	plays tone
POKE 36877, X	128 to 255	plays "noise"

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	U	u	21	*		42
A	a	1	V	v	22	+		43
B	b	2	W	w	23	,		44
C	c	3	X	x	24	—		45
D	d	4	Y	y	25	.		46
E	e	5	Z	z	26	/		47
F	f	6	[		27	Ø		48
G	g	7	£		28	1		49
H	h	8	]		29	2		50
I	i	9	↑		30	3		51
J	j	10	←		31	4		52
K	k	11	SPACE		32	5		53
L	l	12	!		33	6		54
M	m	13	“		34	7		55
N	n	14	#		35	8		56
O	o	15	$		36	9		57
P	p	16	%		37	:		58
Q	q	17	&		38	;		59
R	r	18	'		39	<		60
S	s	19	(		40	=		61
T	t	20	)		41	>		62

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
?		63	▏	T	84	▐		106
─		64	╭	U	85	├		107
♠	A	65	╳	V	86	▗		108
│	B	66	○	W	87	└		109
─	C	67	♣	X	88	┐		110
▔	D	68	▕	Y	89	▂		111
▔	E	69	♦	Z	90	┌		112
▁	F	70	┼		91	┴		113
▏	G	71	▒		92	┬		114
▕	H	72	│		93	┤		115
╮	I	73	π	▒	94	▎		116
╰	J	74	◥	▧	95	▍		117
╯	K	75	SPACE		96	▐		118
⌞	L	76	▌		97	▔		119
╲	M	77	▄		98	▀		120
╱	N	78	▔		99	▃		121
⌜	O	79	▁		100	⌟	✓	122
⌝	P	80	▏		101	▖		123
●	Q	81	▒		102	▝		124
▁	R	82	▕		103	┘		125
♥	S	83	▒		104	▘		126
			◤	▨	105	▚		127

Index

If you own a VIC-20 computer, you've been ready for some serious "playing" without even knowing it. Your "toy" can perform complex calculations and balance your books, as well as enable you to produce your own arcade graphics and electronic music. **DISCOVER YOUR VIC-20** will lead you into the world of real programming.

Going far beyond your basic introduction, **DISCOVER YOUR VIC-20** doesn't merely train with unexplained exercises; it teaches the concepts and practices of BASIC programming, using specific examples. It begins with an understandable step-by-step explanation of the fundamentals, and ventures into advanced professional techniques. And at every point you will be encouraged to experiment with the skills you've learned. With **DISCOVER YOUR VIC-20,** you will be able to leave "canned" material behind, and *learn to write your own programs.*

This book is a user-paced introduction to computing. Use it to discover the hidden potential in both you and your VIC-20.

Don't stay locked in the toybox!
DISCOVER YOUR VIC-20!

DISCOVER YOUR VIC-20

ISBN 0-8176-3160-7
ISBN 3-7643-3160-7

$10.95