SAMS PRESENTS

22089

VIC 20 Programmer's Notebook

Earl R. Savage

BLACKSBURG CONTINUING EDUCATION SERIES™
edited by Titus, Titus & Larsen

VIC 20

PROGRAMMER'S NOTEBOOK

Earl R. Savage

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING — 1983

International Standard Book Number: 0-672-22089-X
Library of Congress Catalog Card Number: 83-50494

Edited by: *Welborn Associates*
Illustrated by: *David K. Cripe*

Printed in the United States of America.

Preface

This Notebook is dedicated to the proposition that program writing on the VIC-20 can be both easier and better. That statement includes every type of program—game, tutorial, financial, mathematical, and record keeping — in short, whatever you set your VIC 20 to do.

All of us begin our programming with efforts that are both simple and crude. Enough effort and debugging made them do the job in spite of their crudeness. No one starts out with flawless programming. In fact, it is extremely doubtful that anyone ever achieves perfection because there is just too much to learn, especially considering the continuous improvements made in both hardware and software. So, this Notebook is designed for you, the VIC 20 programmer-user, whether you are a beginner or an old hand.

If you are getting started, you can achieve better programming results by using the techniques and subroutines in this book because here you will find shortcuts and "tricks of the programmer's trade." There are subroutines and other statement sequences that are designed to "dress up" your programs and make them more professional in appearance. Also, there are techniques for increasing their effectiveness and efficiency. You will find instructions and explanations for making your VIC 20 do things that have not yet occurred to you. Best of all, any of these Notes can be copied directly into your programs because they are written specifically for the VIC 20.

Be advised that there are few programs in this book. There are many subroutines and program fragments that you can use; however, the programs will have to be yours. These Notes will help you do the job — they won't do it for you. There is no substitute for your imagination and sense of purpose.

If you are an advanced programmer, you will find this book a "cafeteria of ideas." There are techniques that you may not have thought of yet. More importantly, these new and different ideas have been worked out for you to save you time. It isn't that you could not work them out for yourself — it's just quicker and easier this way. Why spend your valuable time reinventing the wheel?

ALL of the program statements in this book are *written in the VIC 20 language* — no translation is necessary. The statements make use of the specific features of the VIC 20 keyboard, codes, memory locations, and so on.

Each topic and each subroutine in this Notebook is stated clearly and explained in detail — how it works, why it works, and when and how to use it. Flowcharts are used when needed to clarify the logic further. Interesting and useful variations are presented or suggested.

As indicated previously, no one knows all the possible programming "tricks." Therefore, you won't find them all here. I'm still learning new ones and you probably know some that I haven't yet found. Others will develop as we continue programming. Make your own additions to the Notes in this book. In that way, your cafeteria will always have food for your programming tray.

For now, take full advantage of this smorgasbord. GOOD PROGRAMMING TO YOU!

EARL R. SAVAGE

To Ben and Brian
for whom this will become
child's play.

Contents

How To Use This Book

ORGANIZATION OF THE CONTENT

The VIC 20 Programmer's Notebook is generally organized into alternating chapters of the "how and why" things function as they do and chapters of actual Notes of "how-to-do-it." The Notes in any one chapter are grouped around one or two topics as indicated in the title, with the exception of the last chapter which contains a potpourri of useful routines not fitting into the other topics.

The major purpose of this book is to serve as a reference. It is a source of solutions to problems that you will encounter sooner or later in your program writing. This is not a book, however, which you should put away until you have a programming problem. Its value to you will increase as your knowledge of its content increases. At least, read it through so that you will know where to find the solutions when the problems do arise.

The best way to use this book is to experiment with each Note until you understand how it functions. Key in the statements and run them. Take your time and use care when typing any of the listing and variation statements. Type them exactly as shown. Be particularly careful to avoid confusing the digit 1 with the letter I and the digit 0 with the letter O. Watch out for the punctuation marks, especially trailing semicolons which are quite easy to overlook.

Sometimes, it will be necessary to supply a simple program that will call the subroutine that you are studying. As an alternative, you can use an existing program and substitute the new technique for the one in that program.

In any case, when the program is up and running, study the analysis and the flowchart. When you have a fairly good idea of what is going on, try the variations suggested. Then, you should be ready to apply the Note to your own programs.

LISTING CONVENTIONS

Throughout this book, reversed letters (e.g., CTRL) designate a single keypress. That is, when this designation is used, it means the reversed letters are a single keystroke on the keyboard (or keys that are pressed simultaneously to produce a desired result) instead of being input as individual characters.

The VIC 20 and some other machines make use of special symbols when certain special-purpose keys are used in writing a program statement. In your VIC 20, for example, an R in a color block means that reverse print is activated and a heart symbol in a block means that the SHIFT CLR/HOME keys have been used. Often you will see program listings that show those symbols.

You will not find that type of program listing in this Notebook. I'm a slow learner, perhaps, but the fact is that I had used the VIC 20 for a long time before I knew what to key in when one of those symbols appeared in a printed listing. (Would you believe I still don't recognize all of them??) In any case, there is no need to add to your possible confusion so you won't find those symbols here. You can key in the routines in this book without knowing a one of them.

The listings on the following pages show you exactly which keys to press — not what appears on the screen as a result of the keypress. When you see CTRL-9, just press the Control and 9 keys together — never mind what symbol is displayed. If two reversed keys are joined by a hyphen, as shown here, press them at the same time (well, press the first one a bit before the second). When a reversed key is not joined to another by a hyphen, just press it alone as you would with D CRSR and SPACE. When you see CMDR, just press the Commodore symbol key.

A single or double reversed keypress may be followed by a "modifier" in parentheses. For example R CRSR (4 times) means to press the right cursor 4 times. The modifier is applied to the last keypress only — not all those in the line.

The use of this convention in the listings does have one potential drawback. It is done at the sacrifice of brevity, of course. The listings sometimes look much longer than they are. Don't be discouraged by the apparent length — remember that all those words just represent individual keypresses and they eliminate the guessing by telling you exactly which ones to press.

The "Build" Utility Program

While working on the manuscript for this book, I developed a little program called Build. It saved a lot of time in running, rerunning, modifying, and remodifying the routines here. It can do the same for you as you study these routines and modify them to suit your special needs. I would like to share it with you.

Program Listing

```
15   SC = 256 * PEEK (648) : CO = 38400 : IF SC < 5000 THEN CO =
     37888
20   S1 = 36874 : S2 = 36875 : S3 = 36876 : NO = 36877 : VO =
     36878
.
.
.
85   POKE VO + 1, 8 : POKE 646, 3
90   PRINT CHR$ (147);
.
.
.
999  : : : : : :
1000 POKE VO, 15 : POKE S2, 235 : FOR U = 1 TO 100 : NEXT : POKE
     VO, 0 : POKE S2, 0
1005 PRINT "*";
1010 POKE 198, 0
1015 GET U$
```

```
1020 IF U$ = "" THEN 1015
1025 U = ASC (U$)
1030 IF U = 134 THEN RETURN
1035 IF U = 135 THEN RUN
1040 IF U = 133 THEN 1000
1045 IF U = 136 THEN LIST -998
```

Analysis

15 sets variable SC equal to the start of the Screen RAM and variable CO equal to the start of the Color RAM regardless of any memory that you may have added to your VIC 20.

20 sets variables S1, S2, and S3 to the addresses of the three tone generators and variables NO and VO to the addresses of the noise generator and the volume control, respectively.

.

.

.

85 changes the display to a black background with cyan printing for ease on the eyes (use your favorite colors).

90 clears the screen and homes the cursor.

.

.

.

Notice that now the machine is ready to execute a program involving sound and PEEKs/POKEs into Screen RAM and Color RAM. This is a general "setup" that provides plenty of space for the addition of specifics in any given instance. Normally, you would begin the program or routine that you are studying with line number 100.

999 simply provides a convenient reference number (it's easy to hit the 9-key three times) and some separation from the preceding program/routine.

1000 sounds a short tone as a reminder that it is time for you to do something.

1005 PRINTs an asterisk in case you didn't hear the tone.

1010–1025 clears the keyboard buffer, waits for you to press a key, and determines the ASCII value of the key you press.

1030 if the key was F3, it transfers back to the main program/routine which must have called it with a GOSUB999.

1035 if the key was F5, the program is RUN again.

1040 if the key was F1, it transfers back to the specified line number which must be set according to need (if you start off with a 1000, you will be reminded if you have forgotten to set a needed line number).

1045 if the key was F7, it LISTs the program up to line 998 (normally, you aren't interested in seeing this part of the program).

Notice that any other keypress will END the program and bring up the usual "READY" sign.

Of course, you can use any keys in lines 1030–1045 or even add more. The function keys are convenient though. On my machine, I have stuck little labels beside those keys (GOTO, RETURN, RUN, and LIST).

When you have a program or routine to work on, just load Build and write the statements between lines 90 and 999. An alternate procedure for use with an existing program is to Append lines 1000–1045 to it. Instructions for appending one program (or part) to another will be found in Chapter 8.

Build looks so simple that it may seem not worth the effort to key in and use it but do give it a try. To paraphrase that old potato chip slogan, you can't use it just once.

SLOW SCROLL

It is often necessary to check the LISTing when you are writing a program. Under ordinary circumstances the LIST scrolls up the screen so fast that you don't have time to read it. If you hold down the CTRL key, the scrolling will be slowed considerably.

CAVEAT

Occasionally, you may do everything just right yet be surprised to find the display quite different from your expectations. More often than not, you will fail to see something that you expected. When this happens, check your colors carefully.

It sounds strange, but you can get yourself easily into a situation where you are trying to create a display with PRINT and/or POKE with characters of the same color as the background! You may think that this will never happen but the VIC 20 has so many ways to change character and background colors that you will lose track of them sooner or later. Having the characters and the background the same color is like whistling in the dark — it doesn't help because you can't see anything anyway.

Watch out for the colors but if it does happen, make a quick check by changing the background color. If your characters show up, you know that you have found the problem.

SUMMARY

The final and most important suggestion for using this book is to let it become the nucleus of your own personal notebook. Add any further variations that you discover. Add notes about other useful techniques and routines. If you do this, you will have an increasingly valuable resource for improving your programs.

CHAPTER 1

Programming Considerations

This is the first of alternating chapters in the VIC 20 Programmer's Notebook containing information that will enable you to use the chapters of Notes more effectively. Impatience, however, may prompt you to skip over to the chapters of Notes to see what information is there that you can apply to your program writing right now. That is a normal desire. Go ahead — take a look — but don't forget to come back. If you fail to read this chapter, you could have real problems later.

In order that you won't miss anything, the following statement will mark time until you return.

```
FOR X = 0 TO 10000 : FOR Y = 0 TO 10000 : NEXT Y : NEXT X
```

Welcome back! Now that you have that urge under control, it is time to do a bit of thinking about how you can get the most out of the Notes in this book. Oh, all of them will work just the way they are written, but you can make them work even better. For example, look back at the delay statement. Though it does run just as it is shown, surely you would not want to waste all that memory space. We will consider that and other things in this chapter.

CHARACTERISTICS OF A GOOD PROGRAM

Quite possibly there are as many definitions of what constitutes a "good" program as there are programmers (and would-be programmers). Even the recognized experts disagree on the relative importance of accepted characteristics and on the finer points of good programming.

Nevertheless, every programmer develops his or her own definition — sometimes deliberately with considerable thought but, more often, by default. After all, when you stop working on a program, you must have decided that it is "good." For best results, however, you should make the decision rationally rather than by default. You can do this by considering carefully some of the most widely accepted program characteristics.

You must be aware, too, that a program that is "good" for one type of use may not be at all satisfactory in another use. A program that you write strictly for yourself must meet only your personal requirements. You may be quite satisfied that it does what you wish to have done and no other criteria are important.

If you give copies of your program to friends, however, minimum quality standards are not enough. Further, if you sell or plan to sell your program, it must meet very high standards, indeed.

There is, then, no absolute answer to the question, "What is a good program?" The answer is relative, or as is often stated, "Beauty is in the eye of the beholder!" With that in mind, here are some characteristics that have been offered as being essential to a "good" program.

1. *A good program does what it was designed to do every time it is used, provided the operator follows simple instructions.*

   This characteristic is on everyone's list. It is the bare-bones necessity. A program that still has "bugs" cannot be considered good by any stretch of the imagination.

2. *A good program is foolproof.*

   Another way of stating this is to say that a good program cannot be made to crash or otherwise misbehave even when a fool sits at the keyboard. As a very minimum, an effective error trap is required. Additionally, types of acceptable responses should be made known to the user.

Ideally, your program should simply reject any improper responses, such as those that would result in dividing by zero and those containing a letter when a numerical answer is requested.

3. *A good program "looks" good.*
   While the visual appearance of displays and printouts may be considered as cosmetic, actually it can enhance the user's understanding simply by being clearer to him or her. Charts, graphs, data tabulations, and the like should be concise yet complete, well labeled, and well designed (good spacing and good proportions). Even directions and explanations must be well presented both in wording and spacing. It takes very few additional bytes to space material out on two or three frames (displays) rather than crowding it all on the screen at once.

4. *A good program is self-prompting.*
   A self-prompting program is one in which the directions to the operator appear on the display when they are needed. A user faced with just a question mark on the screen may be at a total loss in understanding what is required. Tell the user what and/or how to make the response — after all, he may have been interrupted by the telephone and lost both his place and his train of thought.

5. *A good program is well organized.*
   In this characteristic, reference is made to the organization of the program statements and not to the organization of the program contents that must be assumed to be reasonable and logical. There is no generally accepted form of statement organization so the following format is just one suggestion. The basic requirement is that statements be placed in logical groups.

   A. Setup (initialization of string space, variables, etc.)
   B. Subroutines
   C. Main Program (separated into logical, distinguishable sections)
   D. Data (before main program if used very often)
   E. Title
   F. Operating Instructions
   G. Variables List

You will note that the order of the items in this list is not the order in which most people write them. In addition, it is evident that it is not the order in which they are used in the program. If this is confusing, read on — there are good reasons for the sequence shown.

6. *A good program has a liberal number of remark statements.*

   All the program sections and all the unusual techniques should be identified right in the program. Not only will this help the user, but you will find it invaluable six months later when you decide to make a modification or two. The only excuse for inadequate REMark statements is that the memory is not sufficiently large to hold them and the working program, too (see documentation).

7. *A good program has a variables list.*

   This is nothing more than a list of the variables and an indication of how each is used in the program. The advantages of having a variables list are the same as those for having REMark statements. The actual list may be in the program, or it may be in the accompanying documentation.

8. *A good program is appropriately documented.*

   A program that is not documented is a literal pain to all who deal with it. Such a program will not be successful in the marketplace. Nor should it be given to friends because of the frustration it engenders (unless, of course, you wish to terminate the friendship). Having no documentation will even be frustrating to you at some future time.

   The word "documentation" does not imply any particular type or quantity. The operative word is "appropriate." A very simple program requires very little documentation and the appropriate types and amount of documentation increase with program complexity.

   Note, too, that documentation does not necessarily mean hard copy. Some programs can be documented quite adequately in the programs themselves. Liberal use of REMark statements and a variables list tacked on the end of a program may be entirely sufficient. Other programs will require printed documentation.

Very basically, documentation tells the user what is required of him, what the program results will be, and how and why the program does its job. Other things, such as background information and references, may be needed.

9. *A good program uses memory efficiently.*
10. *A good program is written to operate as fast as the language allows.*

Efficient memory usage and speed of operation are discussed in detail below.

There are, of course, additional characteristics ascribed to "good" programs. Those listed previously are usually considered the most important. As mentioned, each programmer arrives at his or her own definition in one way or another. Each one will modify his definition with experience and time. If you start off with reasonable expectations for quality in your programs, the results will be obvious to you and to others.

PROGRAMMING FOR MEMORY CONSERVATION

The program statements in this book are printed in such a way as to make them as easy as possible to read and understand. They will function as they are shown. You can use them in your programs if you type them in exactly as they appear.

While they are written for maximum clarity, they are not written for maximum efficiency in the use of available memory. In order to conserve memory space, you may wish to take any or all of the following actions after you understand the particular technique presented. Further, you will find that the changes suggested here will have the bonus effect of speeding up program operation.

Remove the Spaces

Having spaces between words makes for easier reading but it does use up lots of memory. For example, these two statements result in the same program action:

```
250 IF E = 10 OR E = 15 OR E = 22 THEN 30
250 IFE=10ORE=15ORE=22THEN30
```

The second version requires 13 fewer bytes which are then available for other use. In removing spaces, you must be careful of spaces included between quotation marks. You cannot omit these spaces without causing significant program changes.

Place Several Statements on One Line

Multiple-statement lines offer two advantages. First, they save space because fewer line numbers (and internal links and line terminators) are used. Second, such statements are executed faster than are the same statements placed on individual lines.

Multiple-statement lines are written by using colons to separate the formerly individual lines as in this example:

```
145 FOR X = 1 TO 10
150 READ E
155 PRINT E,
160 NEXT X
```

```
145 FOR X = 1 TO 10 : READ E : PRINT E, : NEXT X
```

Each separate statement you eliminate saves at least 5 bytes that were used for line number, link, and terminator. Because you do have to add the colon, the net saving may be only 4 bytes. Four bytes does not seem like much but if you eliminate only 50 lines in a short program, you have saved 200 bytes (or more) — enough for several additional lines. In this process, however, remember that 88 characters (four lines on the screen) is the maximum line length in the VIC 20.

In writing multiple-statement lines, you must be aware of some special cases. When there is a statement referred to by a GOSUB, GOTO, or THEN, you cannot place that statement in the middle of a combined line. If that were done, there would be no correct line number available for referencing.

IF . . . THEN statements also require special treatment. For example, consider these lines:

```
310 IF X = 5 THEN W = 0
315 RETURN
```

Note that line 315 will be executed regardless of the value of X. However, if combined in the following manner:

```
310 IF X = 5 THEN W = 0 : RETURN
```

the RETURN will be executed *only* if X has a value of 5. Obviously, this is not the same as lines 310 and 315. So, watch out for all IF . . . THEN statements to keep from altering the program in unexpected ways.

Miscellaneous Savings

1. Most programmers begin writing with line number 10 or 100 and progress in increments of 10 in order to leave space for inserting additional lines later. This is good procedure when writing but it is wasteful of memory if lines are left numbered in that manner.

Numbers in the text take up one memory location for each digit. Therefore, if a line is numbered 10520 instead of 360, you lose three bytes every time that line is referenced by a GOTO, GOSUB, or THEN. That is one of the reasons, too, for placing the subroutines near the beginning of the program. In any case, your final version of the program should use the lowest possible line numbers and an increment of one; i.e., 1, 2, 3, 4, et cetera.

Fortunately, your VIC 20 makes it easy to change line numbers. All you have to do is to edit the line numbers — place the cursor on the 0 of 120, for example, and DELete twice, then press 2 and RETURN. This changes the line number to 2.

Line 120 is still there as you can see by LISTing the program. In other words, you have duplicated line 120 in the new line 2. Therefore, the last step is to delete line 120.

2. You should make ample use of subroutines. If there is an action that is taken more than once in a program, make those statement lines into a subroutine. The second time you use GOSUB 9, for example, you have saved memory bytes — just how many depends upon the length of the subroutine. Every time you use GOSUB 9 after that, you save a number of bytes equal to just less than that length.

3. The previous reasoning applies equally to the use of variables. If your program uses 36878 (the location for setting speaker volume) several times, you will be ahead of the game by making it equal to V% and using that name in place of the number.

4. There are two types of numerical variables: integer and floating point. An integer variable (whole number) requires 2 bytes while a floating point variable requires 5 bytes of memory. Obviously, you save with each variable you can define as an in-

teger (A%) rather than as a floating point (A).

5. You know, of course, that

```
380 NEXT X
385 NEXT Y
```

can be replaced by 380 NEXT X : NEXT Y at a saving in memory. They can be replaced also by

```
380 NEXT X, Y
```

In fact, the naming of a single variable after the word NEXT is for the benefit of humans — it enables us to follow the flow of the program statements. The computer, however, does not need to be told except in very rare instances. Thus, you can save bytes in any program by writing NEXT instead of NEXT Z. In your early programming efforts, it is advisable to write the variable names to keep from getting confused. When your program is "up and running," you can go back and delete them.

6. Certainly, it should be clear that THEN GOTO and THEN GOSUB are redundant and waste both time and memory. Neither word should follow THEN, in spite of the many times you will see it written that way in programs.

7. There is yet another shortcut involving the IF . . . THEN statement. Suppose, for example, your program has a flag (variable W) which may have a value of 0 or 1. When testing the flag, it is not necessary to write

```
285 IF W = 1 THEN 60
```

The statement can be written as follows:

```
285 IF W THEN 60
```

Note that program execution transfers to line 60 if W equals anything other than zero. It is a shorthand way of saying IF W <> 0 THEN . . . .

8. There are many instances when the usual semicolon may be omitted from statements. Examine the following examples which are written without one or more semicolons where you normally find them:

```
90 PRINT "HELLO, "A$", WELCOME!"     (2 omitted)
95 PRINT A$ CHR$ (99) B$ TAB (14) C$ D$     (4 omitted)
```

Be careful, however, of statements like the following:

```
125 PRINT J M
```

On encountering line 125, your VIC 20 will search for a variable named "JM." If J and M are two variables, they must be separated by a semicolon or by a comma or your program will crash.

9. If your program makes use of one or more identical calculations several times, use the Define Function of your VIC 20. DEF FN will save a few bytes beyond those saved by a subroutine to perform the calculation.

10. When you have finished writing your program and debugged it for the last time, go back and edit the BASIC commands (keywords) in each line. Replace them with their abbreviations: ? for PRINT; R shift I for RIGHT$ — see Appendix B. Do not list the program after converting to abbreviations or the VIC 20 will change them back to the full words. When all abbreviations are made, immediately SAVE the program and you will save memory bytes at the same time.

11. This memory saver has been held until last because it is in direct opposition to what has been said about "good" programming. When memory is at a premium, you may delete REMark statements, of course. If you do have to take this extreme measure, be doubly sure that your documentation on paper is complete.

Memory Summary

You will find few, if any, of the previous memory-saving techniques mentioned in the Notes. Their presence would make it more difficult for you to understand the how and why of the program fragments.

In addition, there are instances in this book where the overall efficiency of one Note can be improved by incorporating the technique of another Note. In the interest of clarity, that was not done because it is easier to grasp just one new concept at a time. By all means, combine a Note with others when you can but only after you thoroughly understand the Note as presented. That is precisely how you improve your programming.

PROGRAMMING FOR SPEEDY OPERATION

As a writer of programs, you will become increasingly aware of the speed at which they execute. The BASIC language works relatively slowly but there are steps you can take to get as much speed as possible from your program. Most of them are listed in memory conservation because almost everything you do to save memory will save time as well.

Any subroutines that are used frequently should be placed at the beginnng of the program. The reason for doing so is that when the program must use one of them, the computer goes back to the beginning and starts searching. Obviously, the quicker it finds them, the quicker it can get on with program execution. The same is true of DATA statements the first time they are used and after each RESTORE command, so it is often advantageous to place them near the beginning of the program, also.

For some reason, programmers seem to fall into a natural writing pattern which places subroutines and DATA statements last in the program. Because of the built-in ability of the VIC 20 to move lines around, you can continue to write in this way. Just be sure to edit the line numbers and move them to their proper places before "tying up" the program — see Low Line Numbers for the procedure.

You may be surprised to learn that variables often RUN faster than do constants. That's right! In most cases, defining DA as 2.5 and using DA in a computation is faster than using 2.5. We agree that this does not seem logical but take a few minutes to prove it to yourself with this little program:

```
10 A = 10 : B = 0.55
20 TI$ = "000000"           REM — 6 zeros
30 FOR X = 1 TO 1000
40 T = T + A - B
50 NEXT
60 PRINT "TIME = " TI / 60
70 PRINT T
```

You will learn more about parts of this program later. For now, be aware that line 20 resets the VIC 20 clock to zero and line 60 PRINTs the time in seconds. Of course, line 70 PRINTs the result of the calculation in line 40.

The time PRINTed on the screen, then, is the number of seconds it takes to do the arithmetic 1000 times. RUN the program and make a note of the time given. Fast, isn't it?

Now, change line 40 to read

```
40 T = T + 10 - 0.55
```

RUN the new program and compare this time with that of the previous version.

We won't tell you the times — RUN the programs and see the difference. The surprising thing is that the second version takes more than twice as long to RUN. It's almost unbelievable but believe the evidence and keep it in mind as you write your programs.

Speed Summary

These are the most important steps you can take to speed execution of your programs (and save memory, too):

1. **Remove spaces in lines.**
2. **Use low line numbers.**
3. **Use multiple-statement lines.**
4. **Place subroutines at/near the program start.**
5. **Never reference a REMark with GOTO, GOSUB, etc.**
6. **Restrict or delete REMark statements.**

FLOWCHARTS

You will find flowcharts throughout this book. The function of a flowchart is simply to provide a diagram of the actions taken by the computer when executing a sequence of statements. There is an old saying that one picture is worth a thousand words. The same may be said of a diagram. A flowchart diagram will help you form a mental picture of what is happening in the program. The more complex the program, the more necessary the flowchart.

There are two types of activities in which flowcharts are especially helpful. One of them is program writing, of course. Programmers seem to fall into three categories. Almost all beginners, but very few who are experienced, begin a new program by typing statements into the computer. Some experienced programmers begin by writing out a complete, detailed description of each step in the program — what it does, how it does it, inputs and outputs. Others begin by drawing a flowchart to simulate and illustrate program action.

Making a flowchart for a complex program can be an involved and time-consuming process, even for the experienced. It is worth the effort, however, because it greatly simplifies the writing process. Most programmers can write a good program from a good flowchart.

The real pay-off, though, comes in the form of time that is saved in "debugging" (troubleshooting) the new program. A program that was not well planned before it was written often requires more time to debug (to get it running) than it took to enter on the keyboard. Then, too, there is the extremely frustrating experience of having a program finally run through, only to discover that it refuses to do what you intended — it goes off on some esoteric mission of its own!

The second activity in which a flowchart is especially helpful is that of attempting to decipher a program written by someone else. Digging into another's program is a fairly common task for a programmer. It is a case of trying to find an answer to the question, "Now, how did he do that?"

The cause of the inquiry may be nothing more than simple curiosity but more often, it is a serious question. One frequent cause is the desire to modify a program to better suit a particular task (if you don't understand it, you can't change it). Another reason is the one which applies to you and this book.

Flowcharts have been included here to provide another means of understanding the subroutines and sequences of program statements given in the Notes. If you don't understand the Notes, you probably won't be able to use them in your own programming. The flowchart serves as a supplement to the written analysis of the program statements.

Even with a program listing and its analysis in front of you, it is sometimes difficult to follow the sequential actions that the computer will take. The flowchart will help you form a mental picture of what is happening. The flowchart will not only help you incorporate the technique or subroutine into your own programs but it will also help you make modifications to suit your needs. You should study the flowcharts in these Notes to ensure your understanding of the execution of the statements.

Flowchart Fundamentals

Each symbol or shape used in a flowchart indicates a specific type of program action. This is done to make it even easier to comprehend the way in which the statements function.

Normally, several different types of symbols appear in a flowchart.

Because this book contains very few programs, it has been possible to reduce the number of shapes or symbols to only two. They and their meanings are:

A rectangular box (Fig. 1-1) indicates an unequivocal action taken by the program. Usually there is a box for each statement but, at times, one box may represent two or more statements.

A diamond-shaped box (Fig. 1-1) indicates a decision point in the program. The diamond contains a question. When program execution reaches this point (diamond), it makes a choice of two (or more) possible subsequent actions. That choice may be based upon operator input or upon information in the memory which provides an answer to the question.

In addition to these two shapes, you will find two other conventions used in the flowcharts:

A series of three dots (Fig. 1-1) is used in place of some program action that is not shown in the chart. In most cases, this symbol is found at the beginning and end of the flowchart to represent the remainder of the total program.

Arrows (Fig. 1-1) show the direction of program flow (execution). The flowchart can be read only by moving from box to box in the direction indicated by the arrows.

Flowchart Examples

To avoid possible confusion later, here are two examples using these symbols and conventions. Fig. 1-1 represents a subroutine in which the operator's answer is rejected unless it matches the correct answer in A$. The first box shows the program GETting the input (B$). In the diamond, a decision is made as to whether or not B$ = A$. If not, the execution goes back to the "GET B$" box. If they are equal, the action goes to the RETURN box (execution returns to the main program).

The ever-present FOR . . . NEXT loop is illustrated in Fig. 1-2, a portion of a flowchart in which a string (A$) is created to consist of 10 randomly selected letters. First, X is set to equal 1. Then, a random letter is generated and concatenated (added) to A$. A decision is made: is X equal to 10? If not, X is incremented (increased) by 1 and the action goes back to generate another random letter. Note that X remains equal to the number of

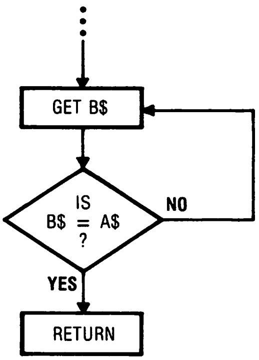


Fig. 1-1. Example flowchart fragment.

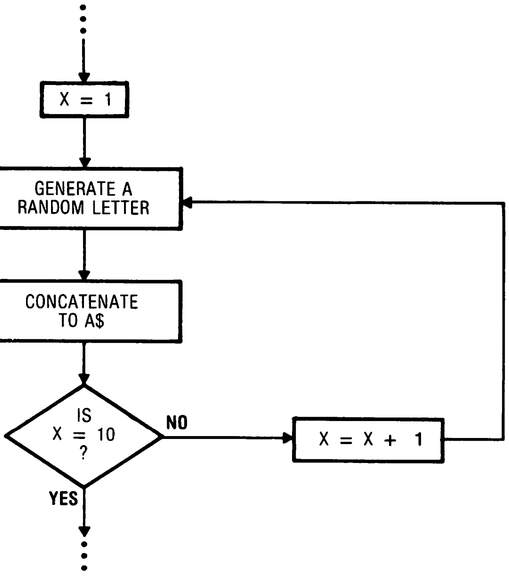


Fig. 1-2. Flowchart example of a FOR . . . NEXT loop.

letters. When X does equal 10, A$ is 10 letters long and execution leaves the loop to continue in the statements that follow.

If you keep these explanations in mind, there is nothing to prevent you from understanding the action shown in any flowchart in this book. Remember to read the program listing, the analysis, and the flowchart together. Each of them is about the same subject and they supplement each other.

Additional important programming considerations will be discussed in Chapter 3.

CHAPTER 2

Notes: Input Controls

There are many reasons for wishing to control the permissible input of the program user. We will take a look at some of them and show you ways to do so. Without some additional programming, the VIC 20 built-in functions do not allow much latitude.

When a response is desired from the user of your program, it is a simple matter, indeed, to insert the statement:

```
60 INPUT "HOW MANY "; A
```

The operator may type and enter up to about 250 characters for the program to use.

Another response mechanism is:

```
80 PRINT "HOW MANY ?";
90 GET A$
```

In this case, the operator is limited to a one-character answer. The operator won't even get a chance to type one character because of the way the GET function works. Unless GET A$ is followed by

```
95 IF A$ = "" THEN 90
```

the user will have no real opportunity to hit a key. The program GETs and finding nothing, moves on to the next statement. Line 95 forces it to wait until a key is actually pressed.

Even so, the operator may not have a chance to answer if, for example, he has been playing with the keyboard before the question is asked. The GET statement should be preceded by

```
85 POKE 198, 0
```

in order to be sure that no "old" keypress is lurking around in the VIC 20. If there is one, the GET will take it as the operator answer and that could give some confusing results. Line 85 clears the keyboard buffer and readies it for new typing.

The preceding methods do allow for user input of data but they are rather restrictive. INPUT requires that the RETURN key be pressed and that may be disadvantageous in some situations. Further, use of the variable A will not allow the entry of alphabetic answers. Using A$ will allow alpha entries but will accept numerics also. To top it all off, INPUT will not accept certain punctuation and symbols and the ones that it rejects depend upon whether the variable is numeric or string.

GET, on the other hand, requires no RETURN key and it will accept anything, even graphics, if a string variable is used but it will accept only one character. Further, A$ will accept both alpha and numeric characters and, even worse, the numeric A will end the program with an error message if the operator should press an alpha key. Obviously, both INPUT and GET can be used in only the simplest programs as shown so far.

You will have other reasons for wishing to control user input. Suppose, for example, that you want to limit the amount of time the operator has to answer the question. You may wish to permit only the correct answer. How about refusing to accept the repetition of an earlier input?

Techniques for these and other controls are given in the Notes that follow. Many of the techniques given in this chapter can be used with either GET or INPUT. You should give special attention to the collected variations in the last section of this chapter.

MULTICHARACTER RESPONSE WITH GET

Because GET will accept punctuation which INPUT rejects, it is often desirable to use the former for operator input. A problem that must be overcome, however, is the fact that GET will accept only one character each time it is called. This subroutine will permit the program to GET more than one character in a response.

Listing

```
.
.
.
610 B$ = ""
620 POKE 198, 0
630 GET A$
640 IF A$ = "" THEN 630
650 IF ASC (A$) = 13 THEN RETURN
660 B$ = B$ + A$
670 GOTO 630
```

Fig. 2-1 shows a flowchart of multicharacter response with GET.

Analysis

610 sets the contents of variable B$ to null.
620 clears the keyboard buffer.
630 assigns the character in the keyboard buffer to A$.
640 transfers execution back to 630 if the buffer contained no character.
650 transfers back to the main program if RETURN is pressed.
660 concatenates A$ to B$.
670 transfers back to 630 for another character.

Use

This subroutine permits you to make use of the advantages of the GET statement when seeking user input. For example, the program can now accept answers containing commas. The operator response is returned to the main program in the variable B$.

Variations

Often, when numerical responses are required, they are to be used in a numerical rather than a string variable. This accommodation can be made in two ways. The first is to convert B$ to a numeric variable after RETURN from the subroutine. If

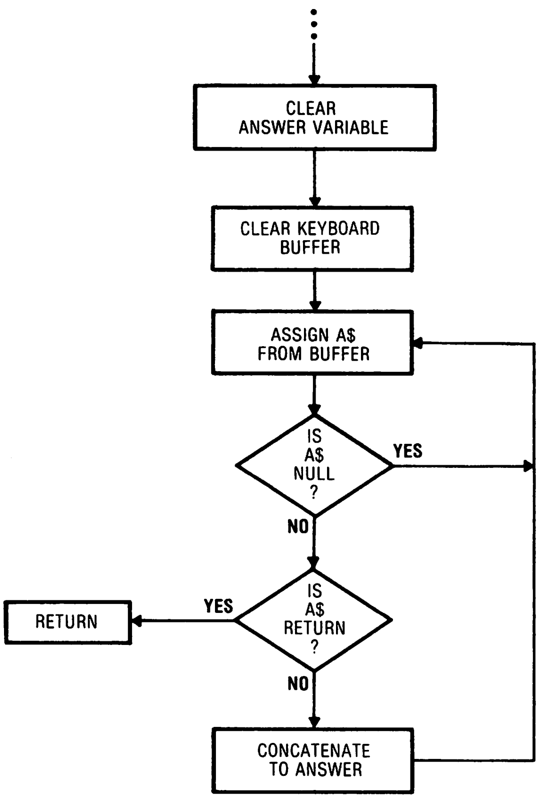


Fig. 2-1. Flowchart of multicharacter response with GET.

this is to be done several times, however, save memory by making the conversion in the subroutine:

```
650 IF ASC (A$) = 13 THEN B = VAL (B$) : RETURN
```

Now, after the response is made, the main program looks to B for numerical answers as well as to B$ for string answers.

If your application requires that the response begin with an alpha character, you could insert these lines:

```
650 IF ASC (A$) = 13 THEN 680
680 IF VAL (B$) < 1 THEN RETURN
690 PRINT "IMPROPER INPUT, TRY AGAIN"
700 GOTO 610
```

To accept only those responses that begin with a digit, change the < 1 to > 0.

EVALUATING VALUES WITH INPUT

There are many times when the characteristics of INPUT are perfectly suitable for program use. In such cases, INPUT should be used as it will save some memory over the use of GET. When it is desirable, responses via the INPUT statement can be limited by type or length as shown below.

FORMATTED INPUT

The formatted input subroutine is used to pre-structure (format) the "answer blank" into which the user's response will be displayed. Even more than that, you may specify almost any characteristic of acceptable responses. This subroutine can be used in any program that requires operator input and it is unusually versatile. Formatted input will add variety, interest, and exactness to your programs.

Though the subroutine appears long in this easy-reading form, it can be compressed considerably through the use of multiple-statement lines. We'll begin with a simple version and add features as needed.

Listing

```
10  UA = 209
12 U$ = " SHIFT-Z (10 times)"
.
.
.
190 PRINT "YOUR ANSWER ?";
200 UL = 5 : GOSUB 840
210 —evaluation of numeric (B) or string (B$) answer—
.
.
.
840 B$ = ""
850 U1 = PEEK (UA) : U2 = PEEK (UA+1) : U3 = PEEK (UA+2)
860 UT = 11 - UL
870 PRINT " CTRL-9 " MID$ (U$, UT) " CTRL-0 "
880 POKE 198, 0
890 POKE UA, U1 : POKE UA + 1, U2 : POKE UA + 2, U3
900 GET A$
910 IF A$ = "" THEN 900
920 IF ASC (A$) = 13 THEN 970
```

```
930 PRINT A$;
940 B$ = B$ + A$
950 IF LEN (B$) = UL THEN 970
960 GOTO 900
970 B = VAL (B$)
980 PRINT CHR$ (32)
990 RETURN
```

See Fig. 2-2 for a flowchart for formatted input.

Analysis

10 sets the first address of cursor position locator.
12 sets the form of the answer blank.
.
.
.
190 asks for the user input (the trailing semicolon causes the answer blank to be displayed right after the question).
200 sets the length of the answer and calls subroutine.
210 program continues.
.
.
.
840 sets the answer variable to null.
850 determines the location of the cursor.
860 determines how much of the answer blank to display.
870 displays the answer blank in reverse color.
880 clears the keyboard buffer.
890 places the cursor back at the beginning of the blank.
900 takes a character from the keyboard buffer.
910 transfers back to 900 if no key was pressed.
920 transfers to 970 if the key was RETURN.
930 PRINTs the character (in A$) in the blank.
940 concatenates the character to B$.
950 transfers to 970 if the length of the answer (B$) is equal to that specified before the subroutine call (this is an automatic RETURN).
960 transfers back to 900 to get another character.
970 sets variable B to the numerical equivalent of B$.
980 PRINTs a space over (erases) the end-of-blank marker.
990 RETURNS to the main program.

Use

This formatted input subroutine can be used in any program requiring operator input. It displays an answer blank with a length as specified before it is called. The operator answer is accepted when its length is equal to the length of the blank. A shorter answer is accepted if the RETURN key is pressed. The

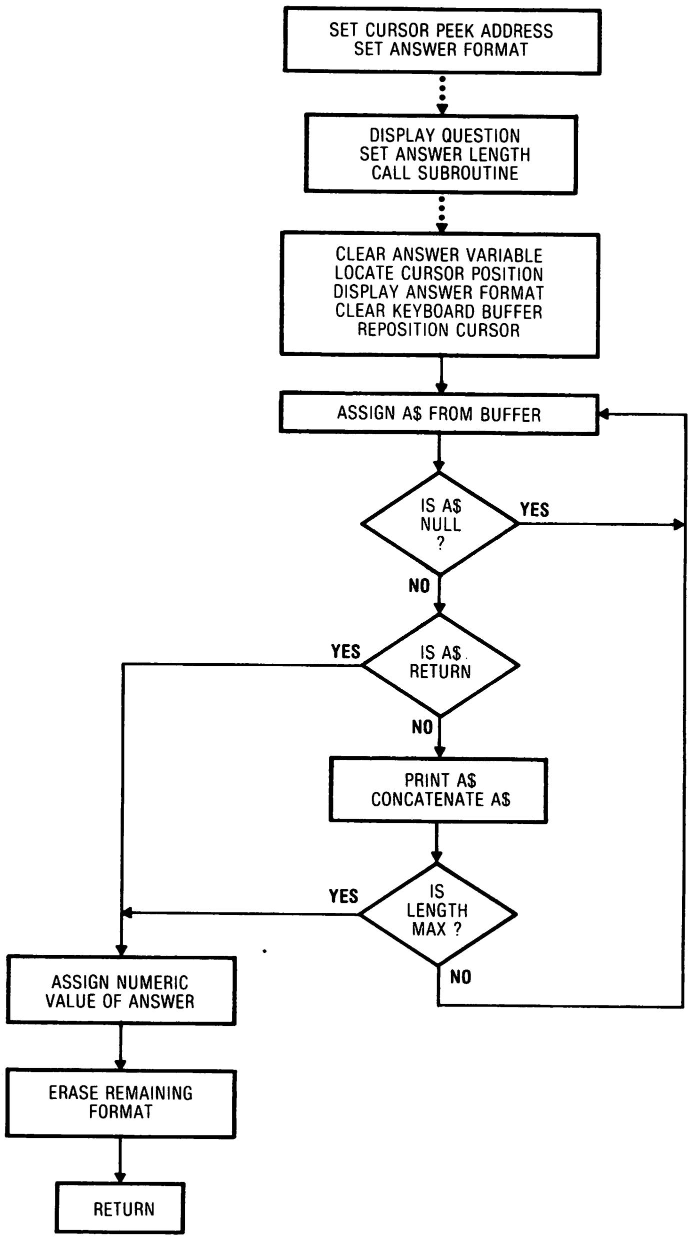


Fig. 2-2. Flowchart for formatted input.

response is in numeric variable B and in string variable B$. Of course, B will equal zero if the first character is a letter.

If the acceptable length of the answer is the same for consecutive questions, it need not be specified each time. Variable UL retains the last value given so you need not set it again unless it is to be changed.

As shown, the subroutine will accept any keyboard entry but methods of restricting the characters it will accept are given as follows:

Variations

1. You may simplify the subroutine a bit by eliminating the automatic RETURN feature. If you delete line 920, pressing the RETURN key will count as a response character. To ignore it altogether, change the 970 in line 920 to 900.

2. The form of the answer blank can be changed by replacing the 10 SHIFT Z graphic characters in line 12. You may want to try such characters as –, CTRL I, or SHIFT W. If the character is changed, the corresponding modifications should be made to any blinking cursor additions (see the following).

3. The maximum length of any answer in the program is determined by the length of the string in U$ (line 12) minus one (for the end marker). It may be changed by shortening or lengthening that string. If this is done, the number (11) in line 860 must equal the total length of U$.

4. The answer end marker (<) can be removed from line 12. If you do this, change the 11 in line 860 to 10.

5. To move the position of the answer blank one space to the right, make this change in line 850:

```
850 . . . : U3 = PEEK (UA + 2) + 1
```

6. If the answer blank (format) is not displayed, responding will be more difficult for the user. This can be done by deleting lines 12, 860, and 870. All of the other advantages of the subroutine are retained.

7. The operator will be less likely to lose his or her place on the screen if the input point is made to flash. A side benefit of the feature is the added attractiveness of the display. These changes to the original subroutine will create a blinking cursor:

```
11   UX = 122
892  PRINT CHR$ (UX);
894  FOR Z = 1 TO 30 : NEXT
910  IF A$ = "" THEN GOSUB 1000 : GOTO 892
925  GOSUB 1000
960  GOTO 892
1000 POKE UA + 2, PEEK (UA + 2) - 1
1010 IF UX = 122 THEN UX = 32 : RETURN
1020 IF UX = 32 THEN UX = 122
1030 RETURN
```

The subroutine at lines 1000 through 1030 serves two functions. First, the cursor is moved to the left by one space to keep it from marching across the screen. Then, the value in variable UX is switched between 32 and 122 in order to blink the cursor.

8. The rate of flashing can be changed by replacing the 30 in the delay loop (line 894) with another number.

9. If you wish to provide additional contrast to the input point, you can change the flashing character. See how you like replacing 122 with 175 in lines 11, 1010, and 1020.

10. A change in color will give further contrast. Assuming the printing is in green, this will change the cursor to red:

```
892 PRINT " CTRL-3 " CHR$ (UX) " CTRL-6 ";
```

11. The entire answer may be displayed in the contrasting color by removing the " CTRL-6 " from 892 and inserting it between the words PRINT and CHR$ in line 980. It may be made a third color by inserting " CTRL" (and your color choice) after PRINT in line 930.

12. As presented to this point, the subroutine will not permit the correction of any typing errors. To enable the DEL function, insert this line in the blinking cursor variation:

```
915 IF ASC (A$) = 20 THEN B$ = LEFT$ (B$, LEN (B$) - 1) : GOSUB
     1000 : GOSUB 1000 : GOTO 892
```

13. There will be times when you do not want the entered characters displayed. That can be accomplished by deleting line 930 and line 925, if the latter is used.

14. There is no limit to the variations on this formatted input subroutine. As one last example, try these lines in the blinking cursor if you would like to add sound:

```
13  POKE 36878, 15
893 POKE 36875, 240
895 POKE 36875, 0
```

DEFAULT INPUTS

In your programs, there will be occasions when you wish a response to have a predetermined value if the operator enters nothing. Often you will see statements like this:

```
50 INPUT "YOUR ANSWER "; A$
60 IF A$ = "" THEN A$ = "NONE"
```

Of course, NONE will be the default answer but this technique wastes memory. A better approach is:

```
80 A$ = "NONE"
90 INPUT "YOUR ANSWER "; A$
```

Lines 80 ad 90 use less memory, yet they act exactly as do lines 50 and 60. Of course, one multistatement line

```
80 A$ = "NONE" : INPUT YOUR ANSWER "; A$
```

is even more economical, as in all the Notes.

In many of the forms, GET routines also keep a preset response if only **RETURN** is pressed. Whether or not a default answer will be carried through depends upon the structure of the routine. If it does not do so initially, usually it can be modified, otherwise you may have to use the technique in lines 50 and 60.

NO-ERROR RESPONSE (WORD)

This subroutine will prevent the user from entering an incorrect response. Only a preset answer will be accepted. Acceptance or rejection is based on the entire word(s). For letter-by-letter evaluation, see the following Note. This technique can be used to advantage in many types of programs.

Listing

```
.
.
.
40  A$ = "PANAMA"
50  GOSUB 700
.
.
.
700 INPUT "YOUR ANSWER "; B$
```

```
710 IF A$ <> B$ THEN PRINT "SORRY, THAT'S NOT RIGHT. TRY
    AGAIN " : GOTO 700
720 RETURN
```

See Fig. 2-3 for a flowchart for no-error word response.

Analysis

40 **sets the correct response.**
50 **calls the subroutine.**
.
.
.
700 **asks for the response.**
710 **PRINTs a message if the response and the preset answer are not the same, and transfers back to 700 for another response.**
720 **transfers back to the main program only when the answer is correct.**

Use

In certain types of tutorial situations, it is very desirable to disallow an incorrect response. It may not be good practice to permit the learner to "get by with" entering BRAZIL when the correct response is PANAMA — negative learning could result. Of course, the operator could be corrected immediately, but why not simply refuse to accept the wrong answer? Refusing would prevent guessing, especially if a limit were set on time or number of trials.

As another example of its use, there is an instance in which you must reject an incorrect response. That is when you ask for a codeword or password. Of course, you may wish to "hide" the correct response in the program per instructions in another chapter.

Variations

Useful variations include limiting the number of trials permitted. See other Notes in this chapter for details on this and other modifications.

NO-ERROR RESPONSE (CHARACTER)

This subroutine evaluates and accepts or rejects a user response on a character-by-character basis. Otherwise, it is similar in action to the subroutine in the previous Note.

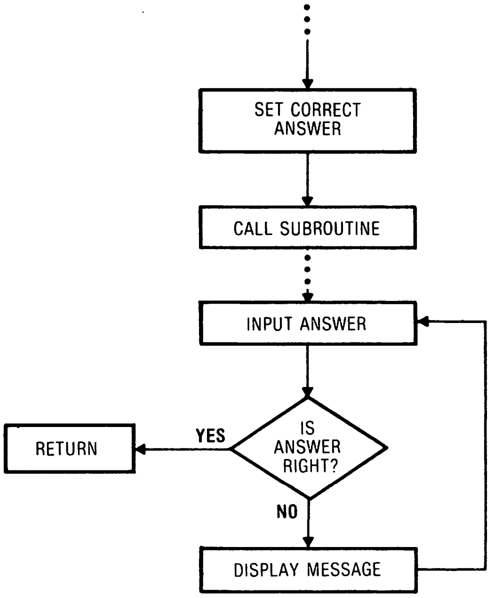


Fig. 2-3. Flowchart for no-error word response.

Listing

```
30  A$ = "CODEWORD"
40  PRINT "ANSWER ?";
50  GOSUB 700
.
.
.
700 Z=0
710 B$ = " "
720 POKE 198, 0
730 GET T$
740 IF T$ = " " THEN 730
750 IF ASC (T$) = 13 THEN RETURN
760 Z = Z + 1
770 IF T$ <> MID$ (A$, Z, 1) THEN PRINT "WRONG" : RETURN
780 B$ = B$ + T$
790 GOTO 720
```

Fig. 2-4 shows a flowchart for no-error character response.

Analysis

30 sets the correct response.
40 poses the question.

50 calls the subroutine.

.

.

.

700 sets the counter to zero.
710 sets the answer variable to null.
720 clears the keyboard buffer.
730 gets a character from the buffer.
740 if the buffer was clear, transfers back to 730.
750 if the character is RETURN, transfers to the main program.
760 increments the counter.
770 checks the current character against the one expected and, if not the same, PRINTs a message and RETURNs to the main program.
780 concatenates the character to answer variable B$.
790 transfers back for the next character.

Use

This technique may be used in the same manner as the preceding Note. The major difference is that the response is checked after each character is entered. Depending upon the program, this may make it less "secure" than the earlier No-Error Response subroutine.

Variations

Time and/or trial limits can be added if desired. See the appropriate Notes in this chapter for details.

MULTIPLE-TRIAL RESPONSE

There are many instances when it is desirable to permit more than one attempt to respond to a question or direction. This subroutine provides for multiple trials.

Listing

```
30  A$ = "PASSWORD"
40  TR = 3 : REM (number of trials permitted)
50  GOSUB 800
.
.
.
800 C = 0
810 C = C + 1
820 IF C = TR + 1 THEN PRINT "THAT'S ENOUGH GUESSING!" :
    RETURN
```

```
830 INPUT "YOUR ANSWER "; B$
840 IF B$ <> A$ THEN PRINT "INCORRECT — TRY AGAIN" :
    GOTO 810
850 RETURN
```

Analysis

30 sets the response equal to A$.
40 sets the number of permitted trials.
50 calls the subroutine.
800 clears the counter.
810 increments the counter by 1.
820 PRINTs a message and RETURNs to the main program if all the permitted trials have been taken.
830 assigns answer to variable B$.
840 if the answer is not correct, transfers back for another trial.
850 transfers back to the main program.

Use

In tutorial type programs, especially, it is often desirable to allow more than one try at making a correct response. The learner may not become as discouraged if he or she can get the answer on the second (or fifth!) attempt. In other types of programs, this technique can allow for typing errors on the first entry.

Variations

1. You can change the number of trials by putting another number in place of 3 in line 40.

2. The number of permitted trials need not be specified for a question if it is not different from the number permitted for the previous question.

3. In many programs, a score is determined by the number of correct answers. When only one answer try is permitted, there is no problem determining the score. When more than one try is allowed, you will want to allow less score for a correct answer on the second or third (or sixth) trial. These changes to the preceding program will compute the score:

```
35 VA = 10 : REM (point value of the question)
55 SCORE = INT (TR / C * (VA / TR) + .5)
```

Variable VA maintains its value until it is changed. The score formula in line 55 decreases the point value for the question with each unsuccessful try. It may be modified to suit your needs.

TIME-LIMITED RESPONSE

This subroutine will allow you to limit the amount of time a user has to input an answer or command. If the response time is too slow in a game, you may want the player to lose a turn or lose points. When a student exceeds a given time, your program can score the question wrong or force him or her to review some material. This time-limited technique has many uses.

Listing

```
40  TM = 20 : REM (equals about 20 seconds)
45  PRINT "YOUR ANSWER ?";
50  GOSUB 800
.
.
.
800 Z = 0 : B$ = ""
810 POKE 198, 0
820 Z = Z + 1
830 GET A$
840 IF Z = TM * 75 THEN PRINT "TIME'S UP!" : RETURN
850 IF A$ = "" THEN 820
860 IF ASC (A$) = 13 THEN RETURN
870 PRINT A$;
880 B$ = B$ + A$
890 GOTO 820
```

See Fig. 2-5 for the flowchart for time-limited response.

Analysis

40 sets the maximum line (1 unit = about 1 second).
45 presents the question.
50 calls the subroutine.
.
.
.
800 sets the timer to zero and sets B$ to null.
810 clears the keyboard buffer.
820 increments the timer.
830 GETS a character from the buffer.
840 checks the time and, if expired, PRINTs a message and transfers back to the main program.
850 transfers back to increment the timer and checks the buffer if the character was null.

860 if the character is RETURN, transfers back to the main program.
870 PRINTs the character.
880 concatenates the character to B$.
890 transfers back to increment the timer and checks the buffer.

Use

This time-limited subroutine is well suited to any application requiring responses during a specified interval. Games, training programs, and tests of memory power, observation, and speed-of-reaction often fall within this category.

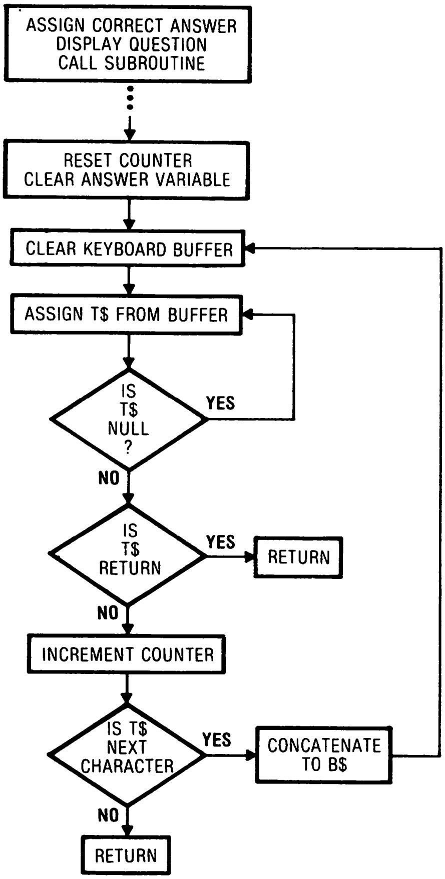


Fig. 2-4. Flowchart for no-error character response.

Variations

1. As the subroutine is given, each unit in the TM variable is worth approximately 1 second. This is determined by the 75 in line 840. Of course, that number can be changed to make a TM unit worth 10 seconds, a minute, or whatever desired.

2. You can take some special action if the user does not respond within a given time. This statement admonishes him or her to get busy!

```
835 IF Z = INT (TM / 2) AND LEN (B$) = "" THEN PRINT "WELL, MAKE
      A GUESS!": GOTO 820
```

In this case, the (TM / 2) displays the message when one-half the time has lapsed provided no acceptable key has been pressed. The denominator can be changed to provide the warning at any desired time.

3. Answers can be automatically entered by making these changes:

```
42  N = 8           : REM (number of characters in answer)
805 P = 0
875 P = P + 1
885 IF P = N THEN RETURN
```

Variable N is the number of characters (including spaces) in the correct answer. Variable P counts the number of entered characters and when it is equal to N, an automatic transfer is made back to the main program.

4. If only certain characters are acceptable, you may wish to build in a penalty for pressing an unacceptable key. To do this, the previous statement could be modified by:

```
840 IF Z = > TM * 75 THEN PRINT "TIME'S UP!" : RETURN
865 IF ASC (A$) < 65 OR ASC (A$) > 90 THEN Z = Z + INT (TM / 10) :
      GOTO 820
```

Here, the user is penalized one-tenth of the permissible time for each wrong key.

5. More accurate timing can be achieved through the use of the timer built into the VIC 20. To use that timer, make the following changes to the original subroutine.

```
800 TI$ = "000000" : REM— 6 ZEROS
820 (delete)
840 IF INT (TI / 60) = > TM THEN PRINT "TIME'S UP!" : RETURN
```

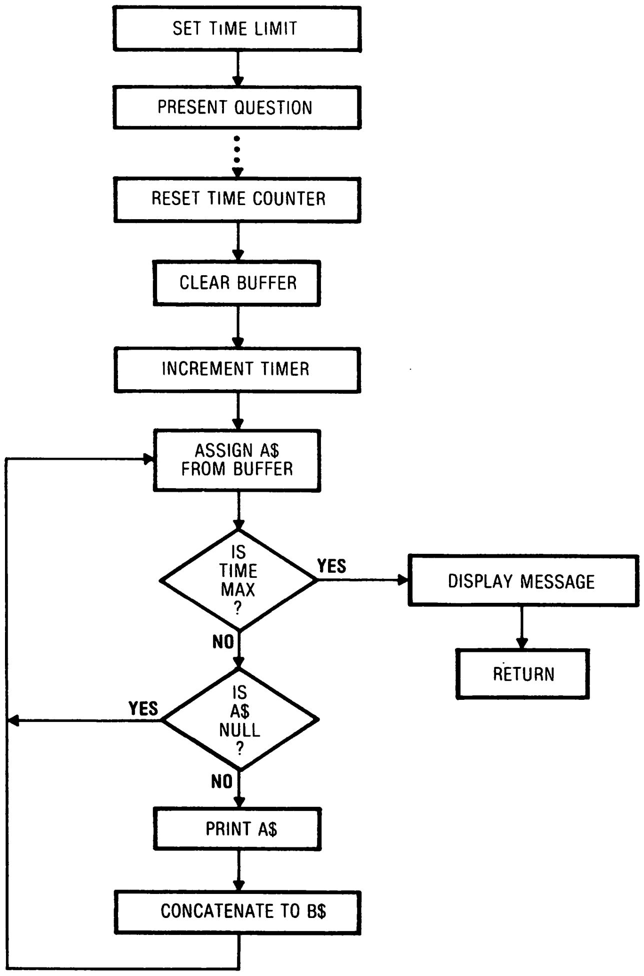


Fig. 2-5. Flowchart for time-limited response.

The preset value of TM is in units of 1 second. Line 800 resets the clock/timer to zero. Resetting is quicker than would be reading the timer in 800 and in 840 and computing the difference in the two readings for comparison with TM. Of course, if this modification to the subroutine is used as shown, the resetting of the clock/timer may cause a problem with some other timing function (if any) in the program.

TIME-WEIGHTED RESPONSE

Just as the number of trials can be used to weight the score for a response, the amount of time can also be used for the same purpose. This subroutine will allow your program to change the score for an answer according to the user's speed.

Listing

```
40  PRINT "YOUR ANSWER ?";
50  GOSUB 800
60  (score routine using TS)
.
.
.
800 TI$ = "000000" : REM— 6 ZEROS
810 POKE 198, 0 : B$ = " "
820 GET A$
830 TS = INT (TI / 600 + .5)
840 IF A$ = " " THEN 820
850 IF ASC (A$) = 13 THEN RETURN
860 B$ = B$ + A$
870 GOTO 820
```

See Fig. 2-6 for the flowchart for time-weighted response.

Analysis

40 poses the question.
50 calls the subroutine.
60 scores the answer using the value of TS as a weighting factor.
.
.
.
800 resets the clock/timer.
810 clears the keyboard buffer and sets B$ equal to null.
820 gets a character from the buffer.
830 sets variable TS equal to the time interval.
840 transfers back to 820 if the character is null.
850 transfers to the main program if the character is RETURN.
860 concatenates the character to B$.
870 transfers to 820 for another character.

Use

The amount of time taken by the operator to make his or her response is returned to the main program in the variable TS. The TS value can be used to give less credit for slower responses. This

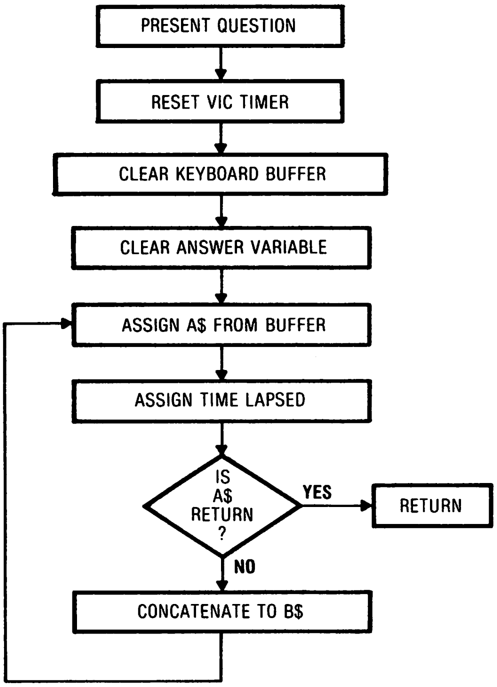


Fig. 2-6. Flowchart of time-weighted response.

subroutine can be quite valuable in such programs as games, tutorials, and reaction speed tests.

Variations

1. As presented previously, the subroutine counts in units of 10 seconds. The value of a unit may be set to any convenient number. For example, changing the 600 in line 830 to 60 will cause the units to be counted in seconds. These figures are based on the fact that each TI unit of the internal clock/timer has a value of one-sixtieth of a second.

2. There are many ways in which you can make use of the time in TS. The operator can be rewarded for faster time and penalized for a slower time, as in this example:

```
30 TU = 3 : REM-- "normal" time
35 VA = 8: REM-- point value of question
```

```
60 IF TS < 1 THEN TS = 1
70 IF B$ = (answer) THEN SCORE = INT (TU / TS * TU + .5)
```

Of course, the statements in lines 60 and 70 determine the effect on the score of the time taken. As a further example, if you change line 60 to

```
60 IF TS < 3 THEN TS = 3
```

slower times will be penalized but faster times will not be rewarded.

A completely different approach to weighting is illustrated by these changes to the preceding statements:

```
60 IF TS < VA / 2 THEN T = 4
65 IF TS > VA + VA / 2 THEN T = -4
70 IF B$ = (answer) THEN SCORE = VA + T
```

Now, if the operator uses less than half of the normal time, he or she is rewarded with an extra 4 points. If he or she uses 50% more than the normal time, he or she is penalized by 4 points. Of course, you can change the times and points to suit your needs.

3. You can have RETURN to the main program executed automatically at the end of some specified time. These lines will do so:

```
30  TU = 3   : REM— maximum time
835 IF TS = > TU THEN PRINT "TIME'S UP!" : RETURN
```

REVERSE RESPONSE

The reversal of entered characters can be advantageous in a number of ways. One obvious application is in the entry of a password for program or data access. In a tutorial for young children, the direction might be to enter a word "backwards." If the program is a game or puzzle, reverse input can be used to introduce additional challenge or even confusion on the part of the user.

With INPUT

In subroutines using INPUT, these statements should be placed immediately before RETURN:

```
nn T$ = ""
nn FOR X = LEN (B$) TO 1 STEP -1
nn T$ = T$ + MID$ (B$, X, 1)
nn NEXT
nn B$ = T$
```

With GET

GET subroutines can reverse the response as shown previously or much simpler by swapping the variables on the right of the concatenation statement:

instead of **B$ = B$ + T$**
use **B$ = T$ + B$**

You can further scramble responses by setting up a counter and inserting:

```
IF Z = > 4 THEN B$ = T$ + B$ : GOTO (GET-line)
B$ = B$ + T$ : GOTO (GET-line)
```

COLLECTED VARIATIONS ON THE PRECEDING NOTES

Many variations on response routines are equally applicable to all or most others. In order to avoid duplication, a number of them have been collected here. Recheck this collection when using any of the Notes in this chapter. The following statements assume that an input character is in T$ and that B$ is the variable in which the response is returned.

1. It would be wasteful of memory space to have one response routine for strings and another for numbers. Because string variables can hold both alpha and numeric characters, response routines usually use strings. To accommodate both needs, the following line is commonly included in the routine:

```
xx B = VAL (B$)
```

If this statement is executed immediately before the RETURN, the main program can look to B$ for a string response and/or to B for a numeric response. This technique is useful in any response routine.

2. GET routines, especially, can incorporate response "filters" very easily. A filter consists of several statements that reject some characters and pass others through. In the routine, the filter is normally positioned immediately before the input character is concatenated to the answer variable.

The first statement assigns the ASCII code of the input character:

```
xx T = ASC (T$)
```

This is followed by a line (or more) in which the actual filtering is done. This line passes only digits:

```
yy IF T < 48 OR T > 57 THEN (GS—# of GET line)
```

Referring to the ASCII table in Appendix G, you will see that an ASCII value less than 0 or more than 9 will be rejected; i.e., it is ignored because the execution goes back to get another character before the current one is used. This statement rejects everything except the letters A through Z:

```
yy IF T < 65 OR T > 90 THEN (GS)
```

Of course, filter statements can be used in various combinations. These pass only digits and letters:

```
yy IF T < 48 OR T > 90 THEN (GS)
zz IF T > 57 OR T < 65 THEN (GS)
```

Using the ASCII table, you can devise statements to filter any character or characters.

3. There will be times when you want to give a hint and another try if an answer is wrong. This can be done as shown in this example:

```
xx IF B$ = "CHICAGO" OR B$ = "DALLAS" THEN PRINT
   "CLOSE—TRY AGAIN" : GOTO (beginning of response routine)
```

4. You can cause special actions if certain characters are entered. In the GET routine, this statement ENDs the program abruptly if the pressed key is anything other than a letter:

```
xx IF ASC (T$) < 65 OR ASC (T$) > 90 THEN END
```

This line can be used in either GET or INPUT routines:

```
xx IF B$ = "JONES" THEN END
```

5. This line will display characters as they are entered in a GET routine:

```
xx PRINT A$;
```

6. Response routines can be made to RETURN automatically under specified conditions as in these examples:

```
xx IF LEN (B$) = > 8 THEN RETURN
xx IF B$ = "YES" THEN RETURN
xx IF ASC (T$) > 57 THEN RETURN
```

7. Color and sound can be used for many effects. Examples are given in the Note on Formatted Input.

CHAPTER 3

Further Programming Considerations

MEMORY CHANGES

If you go beyond the simplest type of programming, undoubtedly you will discover that you must add memory to the basic VIC 20 that you purchased. There is, after all, just so much you can do in the originally available 3583 bytes of free memory regardless of how well you conserve memory in programming. That 3583 bytes will hold only 3583 characters (including spaces and "overhead," such as line numbers, links, and end markers which are discussed later). Why, with no overhead counted, this short paragraph has over 500 characters!

Further, it is a very rare program, indeed, in which you can write into all the available memory. Almost all programs require at least some data manipulation and there are those that require more space for that purpose than is used for the program statements, themselves. Don't forget that there must be free memory for data manipulation such as user-input responses, error trapping, counters for loops, concatenating and parsing strings, mathematical operations, and a host of others.

The point is that the original 3583 free bytes will only satisfy your needs for so long. When your programs begin to bump into that dreaded "OUT OF MEMORY" message, it is past time to add more usable memory (RAM). At that time, you are certainly in good company — every beginning programmer soon runs out of the minimal memory that comes in most computers.

You are fortunate in that the VIC 20 makes it quite easy to add RAM (free memory). All you have to do is to plug it into the back of the machine. Currently available memory modules are 3K, 8K, and 16K in size. (Each "K" is equal to 1024 bytes so "3K" is 3072 bytes, though it is referred to as 3 "thousand.")

Adding memory is easy — but it can cause you programming problems if you are not aware of some of the resulting changes in your system. It is NOT simply a matter of adding more available bytes to the end of what you had.

Depending upon just how much memory is added, the locations of certain things in the VIC 20 shift to different places. For example, when 3K of RAM is added, the beginning of BASIC programs shifts from memory location 4069 to 1024 (and other additions will shift it to 4608). Other significant changes include the locations of the blocks of memory that control the video characters and the video color.

Shifts such as these will cause no discernible change in the functioning of many programs. Others which refer to these locations directly (those which use commands such as SYS, PEEK, and POKE), will "crash" or "bomb" (cease to function) until they are modified to accommodate the changes.

It is important for you to know that such changes do take place and just what they are so that you can write/modify your programs accordingly. Chapter 5 contains that information as well as suggestions for making your programs self-adjustable for machine memory size.

ERROR TRAPPING

Few things, if any, are more frustrating to someone running a program than to have it abruptly quit for no apparent reason. A programmer can earn a lot of enemies that way! In an attempt to win friends and influence people, a programmer must do all he or she can to anticipate confusion and even beginners' mistakes on the part of program users. Further, he or she must build into the program every device he or she can to prevent both accidental and purposeful errors from interfering with its execution.

You cannot prevent an operator of your program from making an error, of course. The operator may enter an alpha character when a numerical character is required. The operator may enter figures that will cause your program to try to divide by zero. The operator may enter a "7" when the largest number in your menu is "5".

The ways a user can make mistakes have never been numbered. There probably never will be such a count because operators are still finding new ways to cause programs to crash or bomb.

One of your big tasks in writing your program is to predict/discover as many potential operator errors as possible. At least, you must find the most likely ones. One way to be sure that you have not overlooked any major possibilities for error is to let a few people run the program while you observe. Usually, the less they know about computers, the better because such folk make the most mistakes.

Rejecting Errors

An "error trap" is nothing more than a procedure you build into the program to capture user mistakes and render them harmless. Your error trap either throws out the mistake or changes it to something that will not ruin your good program. Look at these lines:

```
.
.
.
50 INPUT"WHAT LEVEL OF DIFFICULTY DO YOU WANT (1-3)";B
60 IF B < 1 OR B > 3 THEN 50
.
.
.
```

Line 50 asks the operator to enter a difficulty level of 1, 2, or 3. If the operator enters some other number, it may cause unpredictable results or even crash your program. You can bet on the fact that users will input numbers outside the acceptable range by accident or even on purpose, so you trap their errors.

In line 60, any number less than 1 or greater than 3 is rejected. It will cause the program execution to go back to line 50 and repeat the question. The user simply cannot get past line 60 until he or she enters a number in the specified range. If the operator enters a decimal number, 1.3 for example, it will be treated usually as the integer portion (a 1 in this case). Of course, an acceptable entry will cause execution to "fall through" line 60 and continue with subsequent statements. You have trapped the user's potential errors.

If, in the trap shown previously, you want to make sure that the answer is an integer, you may do so in two ways. First, you could use B% as the variable instead of B. A second method would be to insert this line in the trap:

```
55 B = INT (B)
```

Here is an example of an error trap that is used frequently:

```
120 INPUT "MILES YOU COMMUTE TO SCHOOL OR WORK (ONE WAY)
      "; D$
130 D = ASC (LEFT$ (D$,1)) : IF D < 48 OR D > 57 THEN 120
140 F = VAL (D$)
```

Operators often will enter words when you have expected numbers. Line 120 accepts the input answer as a string so that the user will not get the cryptic "? REDO FROM START" message if he does input alpha characters. Line 130 then determines the ASCII value of the first characters of D$ and checks to see if it is between 48 (the digit 0) and 57 (the digit 9). Thus, if the first character is not a digit, the question will be repeated.

If all is well in line 130, line 140 converts D$ into a numerical value for use in the following program lines. Of course, you could add a trap for unacceptable values to line 140 (: IF F < 3 or F > 100 THEN 120).

If you believe the operator may be confused by having the question repeated, you can modify the last part of line 130 in this manner:

```
130 . . . D > 57 THEN PRINT "FROM 3 TO 100 ONLY" : GOTO 120
```

Changing Errors

Instead of rejecting errors, you can change them into some acceptable response. For example, if you have a menu of five items, you could use these statements:

```
.
.
.
60 INPUT "YOUR CHOICE ";A$
65 A = VAL ( A$ )
70 IF A< 1 OR A > 5 THEN A = 5
.
.
.
```

This little trap changes both alpha entries and entries outside the acceptable 1 to 5 range into 5. Obviously, you can change an error into any response and you would choose one that would represent a reasonable action.

Error Summary

This short treatment of error trapping is sufficient to enable you to catch the major errors. The examples can be adapted to the specific needs of your programs in order to make them "foolproof."

Error traps that become more involved (statements more lengthy) and that are used more than once in a program should be placed in a subroutine so that you can save memory by calling the same statements repeatedly.

NUMBERING SYSTEMS REVIEW

Your VIC 20 uses numbers in all its operations. Even when you might suppose that letters or symbols are used, they are converted to numbers. The letter "B," for instance, is stored and used in the VIC 20 as the number 66. A plus sign (+) is a 43. The BASIC word "GOSUB" is 141.

So it goes — the VIC 20 accepts all manner of input and converts it into numbers for storage and use, then reconverts them before outputting to you. If you are going to be a programmer, you must know something about how the machine handles those numbers.

Even if you don't "like" math, stick around. What you need to know is not all that bad and we'll hit just the high spots. At the very worst, grit your teeth and read along to see that the fundamental ideas are not difficult at all. The best part is that the arithmetic is really simple and there are tables of values to help.

Decimal System

First, let's take a look at our own numbering system that we use every day — the decimal system. It is based on 10, perhaps because we have 10 fingers on which we and our distant ancestors learned to count. (After all, aren't both fingers and single numbers called ''digits''?)

Our decimal system goes like this:

1
2
.
. (We left some out!)
.
8
9
10 — Now it gets interesting. When we get to the tenth count, we move one space to the left to show the number of ''tens'' (in this case, it's one ten). Let's go on . . .
11
12 — This, for example, means one ten and two ones (or two ''units'').
13
.
.
.
19
20 — And this, of course, indicates two tens and no ones or units.
21
.
.
.
99 — Nine tens and nine ones
100 — Since we can't show more than nine tens in one space (digit), we now move another space to the left to show ten tens or one ''hundred''.
243 — Two hundreds and four tens and three ones

So it goes, as high as you wish to count in our decimal or ''tens'' system which is described reasonably enough as a ''base 10'' system.

Binary System

You must have heard that a computer is just a stupid machine. You are about to get your first concrete evidence of the truth of that statement.

We count by tens (decimal) and we give the machine instructions using that numbering system. You may be surprised to know that the VIC 20 is so dumb that it can't count by tens. It can count only by twos! (Does that mean that it came from a place where people had only two fingers?)

Even though there are circuits in the VIC 20 to translate decimal numbers for its use, you need to know that it is a BASE 2 or BINARY system at heart. There will be times when you will have to do some decimal/binary translating, too, for inputs and for determining "bit values" within a byte (more about that later).

The binary system works just like the decimal system except that it is based on 2 instead of 10.

1 — one one
10 — one two and no ones (= 2 decimal)
11 — one two and one one (= 3 decimal)
100 — one four, no twos and no ones (= 4)
101 — one four and one one (= 5)
110 — one four and one two and no ones (= 6)
111 — one four, one two, and one one (= 7)
•
•
•
10101 — one 16, no eights, one four, no twos and one one (= 21)

And so on — by powers of two, of course. It is a simple but cumbersome system with which to work. A conversion table for the binary system is located in Appendix H.

Incidentally, computers use the binary system because they consist of little more than a very large number of electronic switches. A switch can be either on or off, so it has only two "fingers." On is 1 and off is 0. Thus, you might say the dumb machine is nothing but thousands and thousands of two-fingered hands with which to count!

You may wonder why we don't simply use binary numbers in all instructions since the computer "understands" them best. Some programmers do exactly that (use "machine" language) but most of us find it much quicker and easier to use 249 instead of 11111001.

Did you notice the relationships of the "place values" to the base numbers (10 and 2) of these systems? Those values are related to their bases in exactly the same way.

Chart 3-1. Relationships of Place Values

	4	3	2	1	UNIT
DECIMAL (10)	10*10*10*10 10^4 10000	10*10*10 10^3 1000	10*10 10^2 100	10 10^1 10	1 10^0 1
BINARY	2*2*2*2 2^4 16	2*2*2 2^3 8	2*2 2^2 4	2 2^1 2	1 2^0 1

Each place to the left increments the power to which the base is raised. This, of course, gives you a quick way of converting from binary to decimal:

$$\begin{aligned}110110 &= 2^5 + 2^4 + 2^2 + 2^1\\ &= 32 + 16 + 4 + 1\\ &= 53\end{aligned}$$

Systems Summary

As previously stated, there is no need to get "up tight" about numbering systems. Make no effort to memorize any conversion tables. The binary system is simple and, if you use it enough, you will begin to think in binary terms automatically. In the meantime, it is sufficient to have a general idea how it works and know how to change from one to another with the easy conversion table in Appendix H.

NUMBER STORAGE

The VIC 20 and most other "personal" computers operate on a base of eight bit "bytes" or "words" or characters — that is, each memory location has eight bits (electronic switches). Thus, each location can hold a number up to the capacity of eight places in the binary system. Count it up: from 00000000 (zero decimal) to 11111111 (255 decimal). That's a capacity of 256 different numbers — sure, zero is a number. How, then, can the VIC 20 keep a number like 65,000 decimal?

Of course, it could put each digit in one memory location and, even though that is wasteful of space, in some situations it is done. Usually, however, such a number is kept in just two locations (bytes). The designers were very ingenious. They decided that the byte in one of the two memory locations would be automatically multiplied by 256 and the other would be added to the result to give the original number. Further, they decided that the smaller number (the one not multiplied) would be put in the first of the two locations.

Look at it this way. Imagine two boxes representing the two adjacent memory locations:

LOCATION 1	**LOCATION 2**
$\boxed{\textbf{low byte}}$	$\boxed{\textbf{high byte}}$

Location 1 has the lower address (1) and the lower number (the one not multiplied by 256). Location 2 has the higher address (2) and the higher number because it is multiplied by 256.

Now suppose we examined (PEEKed at) two memory locations and found the following:

LOCATION 43	**LOCATION 44**
$\boxed{1}$	$\boxed{16}$

The low byte is 1 and the high byte is 16. Following the previous rules, we multiply 256 times 16 to get 4096 and add 1 to get the actual stored number of 4097. The process is quite straightforward and is used often as illustrated.

The example above is not fictitious. Locations (memory addresses) 43 and 44 are real. They hold the "pointer" to the beginning of the BASIC program that you have in the VIC 20. This pointer in the unexpanded machine does contain 4097, meaning that the first program statement you enter starts at that location/address (4097). We will use many such pointers in the Notes so you should know how to interpret them.

Just as you sometimes need to PEEK at certain locations to determine their contents, there will be occasions when you must POKE numbers into memory locations. There will be times, for example, when you will wish to change the place (address) where a BASIC program begins. In order to do this, you must figure out what numbers to POKE into the pointer.

Suppose you need to have a program begin loading at 4322. What numbers would you POKE into 43 and 44 to make it happen? Well, you would do just the reverse of the preceding procedure. First, get the high order byte by taking the integer (whole number) portion of the result of dividing the number by 256.

4322 / 256 = 16.8828

The integer portion is 16, which becomes the high order byte.
Then, get the low order byte by subtracting from the original number (4322) the product of 256 and the high order byte.

256 * 16 = 4096
4322 – 4096 = 226 (low order byte)

Now, with the low and high order bytes, you are ready to change the pointer so that your BASIC program will load at 4322.

```
POKE 43, 226 : POKE 44, 16
```

Before leaving this matter of decimal byte conversion, let's have a short check-up. Don't read the answer until you have figured out this little problem:

What is the largest decimal number that can be held in two VIC 20 memory locations?

Write your answer on a piece of paper and then see if you are correct.

Well now, two locations will hold two bytes and the largest number that can be in a byte is 255. That means

LOCATION 1	LOCATION 2
255	255

Both the high order byte and the low order byte are 255.
Multiplying the high byte by 256, we have

255 * 256 = 65280

and adding the low byte gives

65280 + 255 = 65535

Thus, 65535 is the largest number the VIC 20 can handle in this way. If you do not thoroughly understand how to change back and forth from a decimal number to decimal high and low order bytes, you should review this section. Your understanding of several of the following Notes is dependent upon your familiarity with this process.

CHAPTER 4

Notes: Cursor Control and Graphics

This chapter contains Notes about two very important phases of programming. You can get by without knowing how to control the cursor — just let it fall where it may. When you can move it at will, however, your programs can include much more useful and pleasing displays. Several approaches to cursor control are shown on the following pages. Check out each one because you will find occasions when your favorite method will not be effective.

The use of graphics can vary from the simple to the ridiculous. Seldom have we seen a plain program of any type that could not be improved by the judicious use of graphics. Do not limit your graphics to games and charts. They can add to the user's understanding as well as his pleasure.

Effective use of graphics often depends upon effective control of the cursor. These topics are grouped together naturally.

DETERMINING THE CURSOR POSITION

If you do much programming at all, you will need a method of determining the screen location of the cursor within a program (while it is being executed).

Listing

```
610 PRINT "01234";
620 A = PEEK (209)
630 B = PEEK (210)
640 C = PEEK (211)
650 D = PEEK (214)
660 PRINT "SCREEN RAM ADDRESS = " B * 256 + A + C
670 PRINT "LINE NUMBER = " (B * 256 + A - 7680) / 22
680 PRINT "CHARACTER POSITION = " C
690 PRINT "214 POSITION = " D
```

Analysis

610 displays 5 digits and positions the cursor on the 5th character position (first position is 0).
620–650 set variables A, B, C, D.
660 displays the Screen RAM address (see Appendix) of the cursor location as computed by formula.
670 displays the number of the line on which the cursor is located as computed (top line is #0)
680 displays the cursor position within the line
690 displays the value in address 214

Use

As given, this routine shows the relationship of several memory addresses to the location of the cursor. For experimental purposes, it may be RUN alone to illustrate those relationships. Varying the number of digits in line 610 and the place on the screen where RUN is executed will change the locations of the cursor and the results.

Addresses 209 and 210 hold the low and high order bytes of the line address in Screen RAM. Address 211 holds the character position. Address 214 holds the number of lines from the top of the display. Seldom are all four addresses needed for any one use but all of them will be needed for various purposes.

Ordinarily, the line position value will vary between 0 and 22 and the character position, between 0 and 21. Note the number 7680 in line 670 is used as the beginning of Screen RAM. Remember that most of the LISTings in this book use addresses for the unexpanded VIC 20, as stated earlier. That number must be changed if you have added memory to your machine (see Chapter 5 for details).

Portions of this routine can be used whenever it is necessary to know or record the location of the cursor. An example of such a need would occur if you wished the program to take some special action if the cursor got as low as line number 18 of the display. This and other examples may be found in the following Notes.

POSITIONING THE CURSOR WITH SPC

Often, you will wish to place the cursor at a given spot on the screen in order to have PRINTing or game action proceed from that point. This routine presents one method you may use to accomplish that.

Listing

```
130 PRINT " SHFT-CLR";
140 INPUT "SPACES ";A
150 PRINT SPC(A) "X"
```

Analysis

130 clears the display and "homes" the cursor.
140 sets variable A equal to the number of spaces you wish to skip.
150 PRINTs X after skipping A spaces.

Use

This routine shows that the SPC function causes the cursor to skip the specified number of spaces before the execution continues. In this case, the skip-count begins on line number 1 (the second line from the top).

SPC can be used in your program whenever the new location is to be below the current location. The maximum number of skipped spaces is 255. Note that there must be no space between the word SPC and the parenthesis.

Variations

1. To prevent the operator from entering an illegal number of spaces, insert these lines:

```
144 IF A < 1 THEN A = 1
146 IF A > 255 THEN A = 255
```

2. You can cause the skip-count to begin at the upper left corner of the display by inserting this line:

```
149 PRINT " HOME ";
```

Of course, you would use SHFT-CLR in place of HOME to clear the display.

3. If you wish to specify only the number of lines to be skipped, make these changes:

```
140 INPUT "NUMBER OF LINES ";A
150 PRINT SPC(A * 22)"X"
```

To retain the safety feature mentioned above, change the 255s in line 146 to 11s.

4. To add the line position to the number of lines, make these changes:

```
142 INPUT "CHARACTER POSITION "; B
150 PRINT SPC(A * 22 + B)
```

If you are using the safeties, change the 255s to 10s in line 146 and insert:

```
147 IF B < 0 THEN B = 0
148 IF B > 21 THEN B = 21
```

POSITIONING THE CURSOR WITH FOR/NEXT

This positioning routine is like the previous routine except that it is not limited to 255 spaces or 11 lines. It is based upon the fact that when cursor movement keys are entered within quotation marks in a program statement, those actions are taken when the statement is executed.

Listing

```
250 INPUT "LINES TO SKIP ";A
260 FOR X = 1 TO A
270 PRINT " D CRSR ";
280 NEXT
290 PRINT "X"
```

Analysis

250 sets variable A to the number of lines.
260 sets up the loop for a count of A.
270 moves the cursor down one line.
280 transfers to 260 if the count is not equal to A.
290 displays X to show skipped lines.

Use

This routine is used as is the preceding one. The maximum number of lines that can be skipped is virtually unlimited. You can even make the count large enough to scroll a current display right off the screen — the actual number to use will depend upon the size of the display and the location of the cursor.

Variations

1. The specification of line position is easily added to the routine:

```
255 INPUT "LINE POSITION ";B;
282 FOR X = 1 TO B
284 PRINT " R CRSR ";
286 NEXT
```

2. To begin counting at the home position, insert:

```
257 PRINT " HOME ";
```

Substituting SHFT-CLR will clear the screen as well.

3. Of course, you can use SHFT-U CRSR and SHFT-L CRSR in the routine to move the cursor "back" into the previously passed display area.

POSITIONING THE CURSOR WITH POKE

At times you will find the previous positioning methods unsatisfactory. Here is a routine that avoids some of the problems they generate. It is closely related to the first Note in this chapter — in fact, they are often used together.

Listing

```
700 PRINT " SHIFT-CLR";
710 INPUT "LINE NUMBER "; A
720 INPUT "CHARACTER POSITION "; B
730 L = A * 22 + 7680
740 L1 = INT (L / 256)
750 POKE 210, L1
760 POKE 209, L - L1 * 256 + B
770 POKE 214, A
780 PRINT "X"
```

Analysis

700 clears the display and homes the cursor.
710 sets variable A equal to the line number.
720 sets variable B equal to the character position.
730–740 computes the high order byte of the address (watch the 7680 if you have added memory).
750 POKEs the high order byte.
760 computes and POKEs the low order byte of the address.
770 POKEs the line number.
780 displays X at the designated screen location.

Use

This routine will place the cursor at any selected location. All subsequent PRINTing will be based on that beginning point.

Variations

1. If you wish to designate how many spaces to skip, make these changes:

```
710 INPUT "SPACES ";A
720 delete
730 L = A + 7680
760 POKE 209, L - L1 * 256
770 POKE 214, A / 22
```

2. You can cause the change of position to be only temporary by deleting line 770. All PRINT statements on line 780 will be made at the new location but any subsequent PRINTing will be done at the old location; i.e., where the cursor was before the POKEs were made.

3. In addition to specifying the new position, you can combine this routine with the first in this chapter. To do so, use that routine when the cursor is positioned where you want to PRINT later. You can PRINT some more lines and then go back to the earlier spot with:

```
920 POKE 209, A
925 POKE 210, B
930 POKE 211, C
935 POKE 214, D
```

In practice, you can have several pre-PEEKed places on the display and position the cursor at them as you wish.

PROHIBITING LINES TO THE CURSOR

In some programming situations, you will want to keep the operator from moving the cursor into selected areas of the display. This technique will permit the placing of such areas "off limits" to the cursor.

Suppose you wished to protect the display below line 15. This statement can be put into the program to keep the cursor above line 16:

```
xx IF PEEK (214)> 15 THEN HOME
```

Here, we are sending the cursor to the home position when it gets out of bounds, but any type of cursor or other action can be taken. By changing the line number and/or substituting a less-than sign for the greater-than sign, you can keep the cursor above or below any line.

Of course, the cursor can be restricted to lines anywhere in the display as in the following example:

```
xx IF PEEK (214) <12 OR PEEK (214) >17 THEN  HOME
```

RANDOM PLACEMENT OF THE CURSOR

In programs of all types, it is often desirable to place characters or graphics randomly on the display. This routine will generate random locations for such use.

Listing

```
900 PRINT " SHFT-CLR ";
910 FOR X = 1 TO 100
920 A = INT (RND (0) * 500)
930 L = A + 7680
940 L1 = INT (L / 256)
950 POKE 210, L1
960 POKE 209, L - L1 * 256
970 PRINT "*";
980 NEXT
```

Analysis

900 clears the display and homes the cursor.
910 sets a loop to count to 100.
920 sets variable A to a random number between 0 and 500.
930–960 computes and places the cursor A spaces from home.

970 PRINTs an asterisk at the cursor position.
980 transfers back to the loop if the count is not 100.

Use

With this routine, the user will find an asterisk (*) "popping up" in various spots on the display. In a tutorial program, randomly placed designs, such as faces, symbols, and words, can be used to reward or reinforce a correct answer or to admonish or punish for a wrong answer. The designs could be smiling or scowling faces, big check marks, "GOOD", "WRONG", "EXCELLENT", "TERRIBLE", and so on.

In a game, random placement can be used to advantage in such ways as placing obstacles in unpredictable locations in the path of a user-controlled car.

The additional variety of randomly placed designs increases operator interest and motivation.

Variations

1. The designs can be superimposed on an existing display if this change is made:

```
900 PRINT " HOME ";
```

2. To change the number of designs, replace the number 100 in line 910 with another.

3. You can limit the area covered by the random designs with:

```
920 A = INT (RND (0) * 88)
```

This statement will shift the area to the middle of the display:

```
920 A = INT (RND (0) * 88) + 150
```

Of course, the numbers 88 and/or 150 may be changed to any desirable value(s).

4. Further variety can be introduced by PRINTing a random number of designs:

```
910 FOR X = 1 TO RND (0) * 200
```

5. If you want to limit the random spots to certain locations, try this change:

```
950 PRINT " HOME ";
920 A = INT (RND (0) * 40) * 11
970 PRINT "GREAT!";
```

Or you might prefer this for some uses:

```
910 FOR X = 1 TO 5
915 PRINT " HOME ";
920 A = INT (RND (0) * 4) * 110
922 B = RND (0)
924 IF B > .5 THEN 930
926 A = A + 11
```

PROGRAMMING GRAPHICS

The effectiveness of your graphic displays is related to the time given to designing and to your imagination in putting the blocks together. Of course, the graphics on the front of the keys can be entered on the screen in the direct mode. To make the same display in a program it is necessary only to place the graphics in quotation marks.

As with many things, however, there are more ways than one to achieve the same result. For example, these statements put a small design on the screen:

```
10 PRINT " SHFT-CLR ";
20 PRINT " CMDR-+ CMDR-U CMDR-+ "SPC(19)" CMDR-+
   CMDR-0 CMDR-+"
```

Note that the graphics symbols are enclosed in quotes just as is CLR/HOME. As you see, the SPC(19) function PRINTs a sufficient number of spaces to cause the bottom of the design to appear in the correct place. For more complex designs, line 20 can be continued until it reaches the maximum length — four screen lines. Beyond that, the design can continue on additional statement lines.

For another way of showing the same design, substitute 19 actual spaces for SPC(19) in line 20. In this example, the use of spaces is impractical, of course, but they may be more effective in other situations. A third method is shown by the following substitution for SPC(19):

```
20 . . . D CRSR SHFT-L CRSR SHFT-L CRSR SHFT-L
   CRSR . . .
```

In this case, you are directing the cursor to move down one line and back up three spaces. In print the line appears to be a great deal longer than it is. Actually, only four characters have been inserted.

You may be wondering why you should bother learning three ways to do the same thing. That's a good question! The answer will be apparent later when you discover that one and sometimes two of the methods will not work in certain situations.

GRAPHIC FACE

We will present a graphic design in the form of a face and then illustrate a number of ways in which graphics and cursor control can be combined to make effective displays. (Note that the design statements appear to be very long but that is deceiving.)

Listing

```
20  QT$ =" SHFT-N CMDR-T CMDR-T SHFT-M D
    CRSR SHFT-L CRSR SHFT-L CRSR SHFT-L CRSR
    SHFT-L CRSR SHFT-L CRSR CMDR-G SHFT-W
    SPACE SHFT-W CMDR-M D CRSR SHFT-L CRSR
    SHFT-L CRSR SHFT CRSR SHFT-CRSR SHFT-L CRSR
    CMDR-G SPACE CMDR-E SPACE CMDR-M"
22  Q$(1) = " VERY GOOD! 5 SPACES CMDR-G SHFT-J
    SHFT-* SHFT-K CMDR-M"
26  QB$ = " D CRSR SHFT-L CRSR SHFT-L CRSR SHFT-L
    CRSR SHFT-L CRSR SHFT-L CRSR SHFT-M
    CMDR-@ CMDR-@ CMDR-@ SHFT-N"
35  Z = 1
880 PRINT " SHFT-CLR ";
980 PRINT QT$ + Q$ (Z) + QB$;
996 POKE 198, 0
998 WAIT 197, 64, 64
```

Analysis

20–26 set the design of the face.
35 sets variable Z equal to 1.
880 clears the screen and homes the cursor.
980 displays the face.
996–998 hold the display without other PRINTing until a key is pressed.

See Fig. 4-1 for flowchart of the expanded face routine.

Use

This little smiling face can be used in games and tutorials as a "reward" for a good score. Line 880 and the following statements would normally be in a subroutine for repeated use in the program.

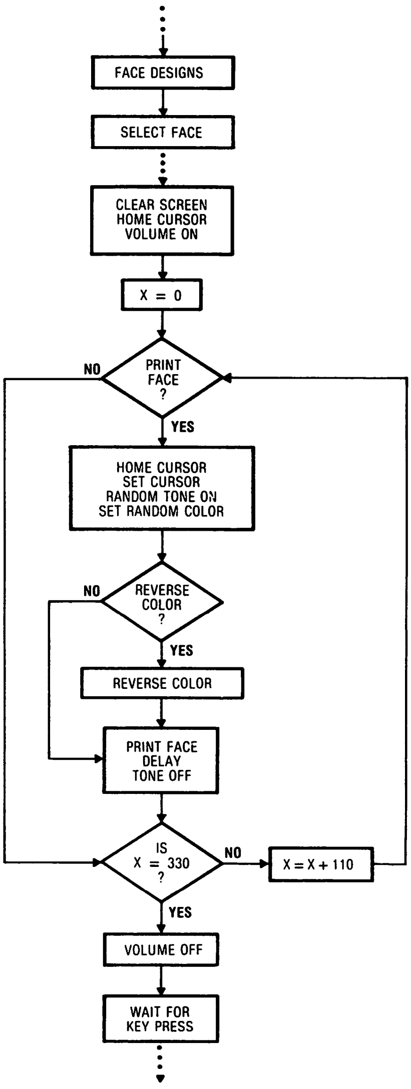


Fig. 4-1. Flowchart for the expanded "face" routine.

Variations

1. Our first variation will be to display several faces at selected locations on the screen:

```
890 FOR X = 0 TO 330 STEP 110
920 PRINT " HOME ";
930 L = X + 7680
940 L1 = INT (L / 256)
950 POKE 210, L1
960 POKE 209, L - L1 * 256
970 POKE 214, INT (X / 22)
990 NEXT
```

Now, four faces appear on the display. This could be done in a simpler manner, but we'll use this method because we want to make changes that it will accommodate.

The FOR/NEXT loop is set up to provide values of X that will place four faces along the left side of the screen. Lines 930 and 940 make intermediate calculations to determine the face locations based on the value of X. Lines 950 through 970 place the cursor as computed.

2. You can change the column in which the faces are PRINTed by placing some quoted spaces in line 980 between PRINT and QT$.

3. To add to the interest, you can cause a random number of faces to be displayed each time the routine is RUN. These additions will do the trick:

```
900 B = RND (0)
910 IF X < 330 AND B > .4 THEN 990
```

Line 900 sets variable B equal to a number between 0 and 1. In line 910, the face is not PRINTed if the value of B is greater than 0.4 unless it is the fourth face; i.e., faces 1, 2, and 3 may or may not appear but face 4 is always PRINTed.

4. There are times when you wish to admonish rather than reward the operator. With the following addition, you can choose a smiling or a frowning (sad) face:

```
24 Q$(2) = " TRY HARDER! 5 SPACES CMDR-G SHFT-U
   SHFT-C SHFT-I CMDR-M
```

Now, all you have to do is to set the value of Z (as in line 35) before calling the subroutine. If Z = 1, the face smiles; if Z = 2, the face frowns.

5. The selection of a smiling or frowning face can be made on a question-by-question basis as shown previously or on the basis of total score. The choice can be made by the program itself. These statements will serve as examples of program selection:

```
924 Z = 1
926 IF SCORE < 80 THEN Z = 2 2
```

You may wish to expand this variation by having a neutral face for mid-range scores.

6. The faces can be reverse-PRINTed with this:

```
978 PRINT " CTRL-9 ";
985 PRINT " CTRL-0 "
```

7. Add these lines and the reverse will be chosen only about half the time:

```
976 C = RND (0)
977 IF C > .5 THEN 980
```

8. Of course, the color may be changed but how about having it chosen randomly! Try this interesting addition:

```
973 A = INT (RND (0) * 7) + 1
974 POKE 646, A
```

You can restore the original color by inserting these lines:

```
882 CC = PEEK (646)
992 POKE 646, CC
```

9. If the face routine is still not fancy enough for you, just add some sound:

```
885 POKE 36878, 15
972 POKE 36875, 200
987 POKE 36875,0
991 POKE 36878,0
```

To sound the tone longer and delay the appearance of subsequent faces, you may prefer to add:

```
986 FOR ZZ = 1 TO 50 : NEXT
```

10. It is clear that the number of variations is limitless for all practical purposes. Here is one final change that will randomly vary the pitch of the sound:

```
971 D = INT (RND (0) * 100) + 152
972 POKE 36875, D
```

MOVING GRAPHICS NO. 1

It is not enough to simply put a graphic design some place on the screen. Very often it is necessary to move it around. This routine for moving a graphic design across the screen is quite straightforward. You just display the design in successive positions and "erase" the old one after each move. In this example, a small dog moves from left to right.

Listing

```
.
.
.
40  A$ = " SHFT-J CMDR-R CMDR-R SHFT-W SHFT-L
        CRSR SHFT-L CRSR SHFT-L CRSR SHFT-L CRSR"
50  PRINT " SHFT-CLR"
60  PRINT " D CRSR D CRSR D CRSR D CRSR"
70  FOR X = 1 TO 17
120 PRINT " SPACE " A$;
130 FOR Y = 1 TO 150 : NEXT Y
140 NEXT X
.
.
.
```

Analysis

40 sets variable A$ equal to the dog design and the proper number of LEFT CURSORs to cause each PRINT to be one space to the right (that number is equal to the number of characters in the design).
50 clears the display and homes the cursor.
60 moves the cursor a few lines down the screen (just for convenience).
70 sets up a 17 count loop.
120 displays a space and the dog.
130 delays a bit to keep the motion from being too fast.
140 transfers back to the loop if the count is not 17.

Use

Moving graphics can be used in many programs to add interest and clarity. It is essential in games. Any characters can be moved, including alphanumerics. Another name for this routine is "animation" though it is rather crude by Disney standards.

Note that it is the trailing blank (space) which erases the dog's tail each time it moves. Of course, it is not necessary to erase the rest of the dog because the "new" dog is PRINTed over it.

Variations

1. The speed of the motion is controlled by the delay loop in line 130. That speed can be varied by changing the maximum count to a number larger or smaller than 150.

2. You can make the routine much more effective by having the dog come out of the left edge of the screen and disappear into the right edge. These additions will illustrate how that is done:

```
70  FOR X = 1 TO 25
80  C$ = " SPACE " + A$
90  IF X < 4 THEN C$ = MID$ (A$, 5 - X, 2 * X)
110 IF X  >  21 THEN C$ = " SPACE " + LEFT$ (A$, 25 - X) +
     RIGHT$ (A$, 25 - X)
120 PRINT C$;
```

The loop count is changed to allow the dog to walk on off the screen. Line 80 is needed because we will PRINT pieces of A$ and we must leave it intact so that we can select those pieces repeatedly. Line 90 selects an increasing number of dog and LEFT CURSOR characters to make it appear that he is coming from behind the border. Line 110 does the same thing to get him behind the right border.

3. If you are getting bored with the walking dog, you might insert this line:

```
100 IF X = 15 THEN FOR Z = 1 TO 500 : NEXT : C$ = " CMDR-D"
     + A$
```

4. The dog can be made to walk backwards by putting the space in front of his nose and using additional LEFT CURSORs in the design.

5. Of course, designs can be made to move up and down by using the same principles. This is illustrated by the following statements which cause a design to "fall" from the top of the screen:

```
200 PRINT " SHFT-CLR";
210 FOR X = 1 TO 20
220 PRINT ""       :REM 3 spaces between quotes
230 PRINT" CMDR-+ CMDR-+ CMDR-+ SHIFT-U CRSR"
240 FOR Z = 1 TO 100 : NEXT Z
250 NEXT X
```

The spaces in line 220 serve as erasers for the previously PRINTed design. The U CRSR in line 230 causes the design to move only one line at a time. There should be as many of them as there are lines in the design.

MOVING GRAPHICS NO. 2

This technique for moving graphics differs from the preceding one in that it allows operator control. The operator can move the cursor as he or she wishes. The example given is actually a small sketch program because the user can PRINT or erase characters as he or she goes.

Listing

```
50  PRINT " SHFT-CLR"
70  PRINT " D CRSR(10 times) R CRSR(10 times) . SHFT-L
    CRSR";
110 POKE 198, 0
120 GET A$
130 IF A$ = " " THEN 120
140 A = ASC (A$)
150 IF A = 13 THEN 200
160 IF A < 32 OR A = 145 OR A = 157 THEN PRINT A$ : GOTO 180
170 PRINT A$" SHFT-L CRSR";
180 FOR Y = 1 TO 100 : NEXT Y
190 GOTO 120
200 GET A$
210 IF A$ = " " THEN 200
```

Analysis

50 clears the screen and homes the cursor.
70 moves the cursor to the middle of the display, PRINTs a period there, and backs the cursor one space.
110 clears the keyboard buffer.
120–130 assign to variable A$ anything found in the buffer.
140 assigns to variable A the ASCII value of A$.
150 transfers to the end routine if the key pressed was RETURN.
160 moves the cursor in the direction indicated by any cursor key that is pressed.
170 displays other characters.
180 executes a delay.
190 transfers to GET another character.
200–210 hold the display clear of other PRINTing.

Use

As it stands, this routine is a complete program that permits the user to write and/or draw anywhere on the screen. It can be used "as is" for many hours of pleasure.

The primary purpose for including the program here, however, is so you can study how the cursor is handled. With that knowledge, you can incorporate the technique into your own programs easily.

Variations

1. This change will test your ingenuity because it starts at a random location on the screen and doesn't display the period to show where it is:

```
70 R = RND (0) * 21
72 FOR X = 1 TO R
74 PRINT " D CRSR";
76 NEXT
80 R = RND (0) * 21
82 FOR X = 1 TO R
84 PRINT " R CRSR";
86 NEXT
```

If you wish to show the starting point, insert

```
90 PRINT ".SHFT-L CRSR";
```

2. You can make interesting and unpredictable patterns with this change:

```
152 R = INT (RND (0) * 21)
154 IF R< 5 THEN PRINT CHR$ (19);
156 IF R> 4 AND R< 11 THEN PRINT CHR$ (29);
158 IF R> 10 AND R< 16 THEN PRINT CHR$ (145);
160 IF R> 15 THEN PRINT CHR$ (157);
```

You may get anything now!

3. With this one, you can control where but not what is PRINTed:

```
165 R = INT (RND (0) * 32) + 95
170 PRINT CHR$ (R) " SHFT-L CRSR";
```

Variable R ranges between 96 and 127. Of course, you can make the range any desirable values (see Appendix G for CHR$ codes).

4. Don't forget the additional possibilities available if you add sound and/or color variations.

ACCELERATING/DECELERATING MOTION

This little routine illustrates a simple method of causing graphic motions of any type to occur faster and faster (accelerate) or slower and slower (decelerate).

Listing

```
.
.
.
300 PRINT " SHFT - CLR "
310 PRINT " D CRSR (10 times) "
320 FOR X = 1 TO 21            REM — decelerate
330 D = X * 50
340 PRINT " SHFT - Q ";
350 FOR T = 1 TO D : NEXT T
360 NEXT X
.
.
.
```

Analysis

300 clears the screen and homes the cursor.
310 moves the cursor down 10 lines.
320 sets up a loop to count to 21.
330 sets variable D equal to 50 times the value of X (therefore, D increases with each pass through the loop).
340 PRINTs a ball.
350 delays for a count equal to the value of D.
360 transfers back to 320 until the count is 21.

Use

Acceleration and deceleration are quite useful for making graphics more realistic. Moving objects can brake or speed up just as they do in real life. The same speed changes seem to take place when an object is moving away from or toward the observer (into or out of the screen).

This technique can be applied also to flashing words or graphics. Thus, it can add variety and emotional content to a program — suspense, excitement, and so on.

Variations

1. The rate of deceleration is determined by the calculation of D in line 330. Obviously, the 50 can be changed to other numbers. For different effects, try various formulas, such as

```
330 D = X UP ARROW 2 * 50
```

2. Of course, the action can be made to accelerate by making the value of D smaller with each pass through the loop if you use such formulas as these:

```
330 D = 1500 / X
330 D = 1500 / X UP ARROW 1.5
```

3. You can impart a "jerking" or uneven motion with

```
330 D = RND (0) * 2000
```

which may be somewhat clearer to see if line 340 is changed to

```
340 PRINT " SPACE SHFT-Q SHFT-L CRSR";
```

CHAPTER 5

Memory Organization and the Operating System

Your VIC 20 is called a 64K machine because it can "address" 65536 (64K) memory locations. Each location has a unique numerical address just like the houses on Pine Street in your home town. Due to the fact that the VIC 20 has only 16 binary address lines, the highest address of the last memory location is 1111111111111111 (that's 16 ones or 65535 decimal). Counting zero (0000000000000000) as the first memory location, we have a total of 65536, which is 64K.

It would appear, then, that we have a potential 64K of memory in which to load our programs. This would be the case except for the fact that some of the memory must be used to hold the operating system. Even the most primitive computer must have some fundamental instructions to tell it how to do what the program directs. Those fundamental instructions constitute the operating system. In general terms, the more sophisticated the computer, the larger its operating system.

Certainly, the VIC 20 does not offer 64K of memory for programming even when the maximum amount has been added. Table 5-1 shows the uses to which the 64K available addresses have been assigned. The first two columns give the addresses of certain points of interest in decimal form. The last three columns show the use of the addresses (memory) between those points for an unadorned 5K VIC 20, for a machine with a 3K memory addition, and for one with 8K or more added.

Examination of Table 5-1 indicates that there is a maximum of about 31K available for your programming use. Let's take a look at where that is located and at the uses of the other memory locations. Keep in mind that ROM is "Read-Only-Memory" from which you can get instructions, values, etc., but you cannot change the contents of this memory.

The other memory type is RAM — a complete misnomer because it stands for "Random-Access-Memory." In truth, both ROM and RAM are random access memories. When you see "RAM," you should think "Read-And-Write" memory which you cannot only read (instructions, values) but you can write there, too; i.e., you can change the contents of RAM to suit your needs.

The first 1K of addresses (0-1023) is used identically, regardless of the amount of memory in a machine. This is RAM but it is not available to hold programs. It is "reserved" for use by the operating system — in fact, you can consider it part of the operating system. A great many interesting things are to be found in the reserved RAM and we will discuss it in some detail in Chapter 7.

The next 3K addresses (1024–4096) are unused in the 5K machine but this is where the first 3K expansion memory goes. With that addition, 1024 becomes the beginning of BASIC RAM for your programs instead of the previous 4097. Note, too, that if more than 3K is added, this area becomes unavailable for BASIC programs even though there may be RAM present (it may be used for machine language programs and other specialized uses).

Because this is the first instance of differences in the three VIC 20 configurations, you should take care to understand how such things can affect your programs. Suppose, for example, that your BASIC program refers directly to the actual location of the beginning of BASIC. There are a number of such programs in this book. Such a program cannot refer to a specific number unless its use is restricted to one "size" of VIC 20.

If that program were to include the statement "POKE 4097, 1", it would put a 1 into the first address of the BASIC RAM in a

Table 5-1. General Memory Organization

DEC	5K	+3K	+8K & UP
0			
1024	(1K) Reserved RAM	Same	Same
4096	(3K) Not Used	BASIC RAM	Not Used
4608	(.5K) BASIC RAM	BASIC RAM	Screen RAM
7680	(3K) BASIC RAM	BASIC RAM	BASIC RAM
8192	(.5K) Screen RAM	Screen RAM	BASIC RAM
16384	(8K) RAM Expansion	Same	BASIC RAM
24576	(8K) RAM Expansion	Same	Same
32768	(8K) RAM Expansion	Same	Same
36864	(4K) Character ROM	Same	Same
37136	(.3K) VIC: Video Chip	Same	Same
37888	(.7K) I/O Port	Same	Same
38400	(.5K)		Color RAM
38912	(.5K) Color RAM	Same	
40960	(2K) I/O Ports	Same	Same
49152	(8K) ROM Expansion	Same	Same
57344	(8K) ROM BASIC	Same	Same
65535	(8K) ROM KERNAL	Same	Same

5K machine. In an 8K machine, the program would likely crash because that 1 would go in about 3K past the beginning of the program. If the VIC 20 happened to have 13K or more RAM, the letter "A" would appear in the upper left corner of the screen!

Now, your program may require that any one of these actions be taken (but not all three of them) with the selection depending on the amount of memory in the machine. If you write a program that uses the commands PEEK and/or POKE, be very sure that you make provisions for different memory configurations. Failure to do so will earn you dirty thoughts (at least) from others who use your programs.

Continuing with Table 5-1 and the VIC 20 memory organization, note that when the memory exceeds 8K, the Screen RAM area is shifted from 7680 to 4096 and the beginning of BASIC RAM is raised to 4608. The video display is said to be "memory mapped" because 506 memory locations are designated to correspond to the 506 display blocks on the screen (23 lines of 22 characters). Anything that you (or the VIC 20) put into one of those memory locations will be placed automatically on the screen (though, of course, it may be "invisible" if it is the same color as the screen). For example, if you POKE the number 81 into the middle of the Screen RAM area, a ball will appear in the middle of the display — if it doesn't appear, change the color of the ball or of the screen as fully described in Chapter 9. To place anything on the display, then, you (and the VIC 20) must know which RAM is being used for the screen.

Addresses up to 32768 are used for BASIC RAM provided memory has actually been added there. Your BASIC program and the working areas it may require can extend from 4608 to 32768 — a total of 27.5K. Keep in mind that this refers just to BASIC space. There are other addresses here and there that can be used for other programming purposes, such as the 3K from 1024 to 4096. In fact, any one of the entire 64K of memory addresses might be used by your programs.

A 4K character generator ROM is located from 32768 to 36863. It holds the instructions for making upper and lower case characters and graphics, both normal and reversed.

Input/Output addresses start at 36864. An I/O "port" may be thought of as a delivery gate through which material (information in the form of numerical values) can be sent out of or brought into the computer. The tv set/monitor is a port through which visual images are sent to you, the operator. The keyboard is a port through which you send information to the VIC 20.

Other ports handle the cassette recorder/player, disk drive, printer, joystick, paddles, light pen, and so on. Note that these ports can be two-way gates or one-way only and that they have specific addresses; i.e., they are memory mapped (like the Screen RAM).

The first I/O locations are assigned to the Video Interface Chip, called the VIC. (Now you know where your computer got its name!) This remarkable integrated circuit generates the actual audio and visual signals that go to the monitor/tv set. As if that were not enough, it also calculates the positions/values of the light pen and paddles.

Right in the midst of the I/O port addresses is the Color RAM area — 37888 to 38911. Note that the exact location of the Color RAM shifts a bit in this area with different memory sizes. Just as the Screen RAM holds the character or graphic block to be displayed, the Color RAM holds the color to be used in displaying that character/graphic.

After the port addresses, an 8K section is set aside for ROM expansions (40960–49151). It is usually used by game cartridges. The next 8K (49152–57343) contains the all-important ROM BASIC. In a manner of speaking, that 8K acts as a translator between the high-level "BASIC" language that we usually use (PRINT, PEEK, FOR, IF, etc.) and the machine language understood by the VIC 20 (all those ones and zeros we have discussed before). Using that translator (interpreter), we can have the VIC 20 display the digits 0 to 9 with the simple instruction, "For X = 0 TO 9: PRINT X ; : NEXT". That is much more difficult to do in machine language.

The final 8K addresses (57344-65535) are used by the ROM KERNAL. Whatever the reason for naming this part of the system KERNAL, it should have been as a take-off on the word COLONEL because the KERNAL is, in fact, the officer in charge of the VIC 20. Did you wonder how the various areas of RAM were shifted and assigned? — well, that is one of the functions of the commanding officer.

Among the many tasks performed by the KERNAL is an initial check of the amount of memory in the machine. Based upon what it finds, the KERNAL assigns the RAM as shown in Table 5-1. Further, it sets all the pointers found in the first 1K "housekeeping" or "reserved" RAM (see Chapter 7). It serves as a machine language "jump table" and, finally, it displays the opening power-up message on the screen and turns the VIC 20 over to you and BASIC.

As you see, the functions of several address areas change considerably as RAM is added to the VIC 20. These shifts make little or no difference in the operation of many BASIC programs — just load them in and changes are automatically accommodated. Other BASIC programs, however, must be adapted by the user unless the original programmer has made adequate provisions for RAM shifts. In Chapter 7, you will find instructions for identifying such programs and modifying them to run in various RAM organizations.

BASIC LANGUAGE CONSIDERATIONS

If you are to communicate with the VIC 20, you must learn a language that it "knows." As mentioned previously, BASIC is the language built into your VIC 20. Though you can arrange to use other languages with the machine, concentrate on learning to use BASIC well. Not only is BASIC already there, but it is an excellent general purpose language that is not unlike English.

The words of the VIC 20's rather extensive BASIC language are listed in four functional groups in Appendix B. For your review, a very brief definition is given for each word as well as its "token." The use of tokens is discussed in Chapter 13.

If you are a beginner, here is a good way to learn to use the BASIC language of your machine: Never type a word in unless you know how and why it is used. If you will follow that rule when copying lines from this book, you will be surprised at how quickly BASIC begins to make sense. As necessary, refer to the fundamental definitions and examples in your owner's manual.

SCREEN RAM USE

As shown in Table 5-1, the screen RAM begins at 7680 (or 4096) and extends to 8185 (or 4601). As our discussions deal with a "plain" 5K VIC 20, we will be using the higher set of figures; i.e., 7680 to 8185. You will have noted that these figures represent 506 memory addresses. That is the same number, of course, as the number of "spaces" on the display screen: 22 characters × 23 lines = 506 spaces. The logical conclusion is that each memory address is related in some way to each space on the screen and so it is!

Appendix C shows a sketch of the display screen, that is, 23 lines of 22 spaces each (0 to 21, across the top). The numbers down the left side are memory addresses of the spaces in the left-most column. With these figures, you can determine the address of any space on the screen. For example, the address of the upper left space is 7680 (that is 7680 + 0). The upper right space is 7701 (7680 + 21). The bottom right space is 8185 (8164 + 21).

Let's see how we can put this information to use. First, take the following steps:

1. Turn the VIC 20 off and then on, so that the screen displays the usual sign-on message

```
****  CBM BASIC V2  ****
3583 BYTES FREE
READY.
```

2. Type this instruction

POKE 36879, 8 RETURN

3. Press CTRL-2, then SHFT-CLR
4. Press RETURN four times

Do not be concerned at this time about steps 2 through 4. You will learn more about them in Chapter 9. For now, we just want to place the cursor (white) at the beginning of the fifth line on the screen (black). If this does not describe the appearance of your display, repeat the steps.

Now, type POKE 7788, 102 and press RETURN. Immediately, a checkered square appears near (at?) the end of that fifth line. If the square does not appear, you did not change the colors properly — repeat steps 1 through 4. Now, looking again at Appendix C, we can figure that the design has been placed in the next-to-the-last square of the line (7768 + 20). Let's see if that is correct.

Type POKE 7789, 81 RETURN and, then, POKE 7790, 83 RETURN. The two graphic blocks appear just where you expect them. Now, check that address/display correlation one last time with these entries:

POKE 7680, 102 RETURN
POKE 8185, 102 RETURN

At this time it should be quite clear that whatever you put into screen RAM appears on the screen display in a corresponding position provided, of course, that the colors are different. This is very handy information and we will use it often in the Notes. Appendix C tells you where to put the character but how do you know what to put there?

Here is another little experiment after having executed the previous steps 1–4:

1. Clear the screen (press SHFT-CLR)
2. Press RETURN 15 times
3. Type FOR X = 0 TO 127 : POKE 7680 + X, X : POKE 7834 + X, 128 + X : NEXT RETURN

When this line is executed, the first six lines are filled with letters and graphics; the seventh line is blank; and the next six are like the first six except the characters/graphics are reversed. Poking X (0 to 127) produced the first set and poking 128+X (128 to 255) produced the second.

Now, press the Commodore symbol and SHIFT keys together. This action changes many, but not all, of the characters/graphics in each set. You may switch them back and forth by repeatedly pressing these same two keys.

The chart in Appendix D will help you keep track of all these figures. Note that the two sets are given beside the number that is poked to display them. There is a second way to shift between the sets. Try poking the values 240 and 242 into address 36869.

SUMMARY

As you write and modify programs, it is very important that you keep in mind the way in which the memory is organized and how that organization changes as memory modules are added to the VIC 20. Adaptations of your programs must be made for the various memory configurations or your programs will not function properly, if at all.

Special attention has been given here to the Screen RAM area of memory. An understanding of screen RAM and its use is essential if you are to make the most of a number of the Notes. Other parts of the memory are discussed elsewhere in this book.

CHAPTER 6

Notes: The Display

The appearance of the display is the second most important characteristic of your program that will influence users. The most important, certainly, is whether or not it does what it is supposed to do. If your program presents a clear and pleasing series of pictures to the users, they will like it. Of course, "clear" refers to clarity of organization leading to user understanding rather than to sharpness of image.

This chapter contains techniques that will assist you in presenting clear and pleasing displays. They include placement of characters (words or graphics), manipulation of screen sections and other useful routines.

PRINTING IN GROUPS

There are times when the quantity of items to be presented is too great to fit on one screen. At other times, you will wish to present items in groups so the operator will not be overwhelmed.

Listing

```
.
.
.
190 PRINT " SHFT-CLR";
200 FOR X = 1 TO 100
210 Z = RND (0) * 60 + 32
230 PRINT CHR$ (Z)
250 IF INT (X / 20) = X / 20 THEN WAIT 197, 64, 64 : PRINT
300 NEXT
.
.
.
```

Analysis

190 clears the screen and homes the cursor.
200 sets up a 100-count loop.
210 generates a random character code and assigns its value to variable Z.
230 PRINTs the character represented by Z.
250 stops the PRINTing after each 20 characters, WAITs for a key to be pressed, then PRINTs a blank line before proceeding.
300 continues the loop unless the count has reached 100.

Use

This routine can be used whenever you wish to present material in sections. The one given previously uses a series of single characters just for the purpose of illustrating the technique. The material can be of any type, alphanumerics and/or graphics, and it can consist of single characters or entire lines.

Variations

1. To change the number of items in each group, it is necessary only to change the two numbers (20s) in line 250.

2. You can do anything practical to the display after a single group has been PRINTed. As shown, a blank line is PRINTed and the list continues, scrolling up the screen. To see other possible effects, try these changes after the WAIT, 64, 64 : in line 250:

```
250 . . . : PRINT " SHFT-CLR ";
```

clears the screen and homes the cursor to start anew with each group.

```
250 . . . : PRINT "----"X "----"
```

identifies each group by its sequential number.

```
250 . . . : PRINT " CMDR - + (5 times)"
```

separates the groups with a graphic.

```
250 . . . : PRINT " D CRSR" : PRINT "PRESS ANY KEY TO CONTINUE"
```

skips down a line and gives the user instructions. Obviously, several of these changes can be used together if desired.

3. When the items to be PRINTed are short as in this example, you can have several displayed simultaneously. To do so, make this change:

```
220 IF C > 0 THEN FOR Y = 1 TO C : PRINT " R CRSR (5 times) " ; :
    NEXT
250 IF INT (X / 20) = X / 20 THEN PRINT " SHFT - U CRSR (20 times)
    " : C = C + 1
```

Now the cursor is returned to the top line for each group (line 250) and each subsequent group is PRINTed five spaces to the right (line 220). As before, these numbers can be changed to suit the material being presented.

4. If, regardless of how it is displayed, your material will not fit on one display, there is a technique to take care of the problem. Insert this line into the previous Variation.

```
260 IF INT (X / 40) = X / 40 THEN WAIT 197, 64, 64
: PRINT " SHFT - CLR " ; : C = 0
```

5. Don't forget both color and sound to make your display clearer and more pleasing. Use color for emphasis and attention-getting. A short "beep" can be effective after each group is PRINTed to remind the operator to take some action.

SPLIT-SCREEN OPERATION

A split screen gives you the ability to display information on one portion of the screen while being able to change information on another portion. There is no convenient way in BASIC to have true split-screen operation; i.e., have independent scrolling on part of the screen. Fortunately, you can approximate this function with a simple routine that, however, is limited to re-using the top portion of the display. See also Adjustable Clear.

Listing

```
.
.
.
120 PRINT " SHFT-CLR ";
130 FOR X = 1 TO 22 : PRINT "** (21 times) " : NEXT
140 LN = 7
150 GOSUB 600
160 FOR X = 1 TO 20
170 GOSUB 700
180 PRINT X
190 NEXT X
.
.
.
600 PRINT " HOME ";
610 FOR Y = 1 TO LN : PRINT "(21 spaces)" : NEXT
    : PRINT " HOME ";
620 RETURN
.
.
.
700 IF PEEK (214) > LN - 1 THEN WAIT 197, 64, 64
    : GOSUB 600
710 RETURN
.
.
.
```

Analysis

120–130 set up a demonstration display.
140 sets variable LN to the number of lines to split.
150 calls the line clear subroutine.
160–190 set up a demonstration of use.
160 establishes a 20-count loop.
170 calls the line number check subroutine.
180 PRINTs the value of X.
190 continues the loop.

.
.
.

600 homes the cursor.
610 PRINTs LN blank lines and homes the cursor.
620 transfers back to the main program.

.
.
.

700 if the line number is greater than LN, WAITs for a key to be pressed, then calls 600 to clear lines and homes the cursor.
710 transfers back to the main program.

Use

Split screen operation is quite useful when there is material on the bottom part of the display that must remain in view while it is explained or while questions are asked about it. The material may be alphanumerics and/or graphics and it would be inconvenient to redisplay it time after time. Using a split screen, the information can be left intact while the remaining portion of the screen is used and reused as often as necessary.

You will find use for this technique in all types of programs.

Variations

1. The number of lines cleared is determined by the value assigned to LN before the subroutine is called.

2. You can use this method in any situation in which the operator is entering characters. Here is an example of its use with the GET command:

```
330 PRINT "ANSWER? ";
340 GET A$
350 IF A$ = "" THEN 340
360 IF ASC (A$) = 13 THEN RETURN
370 GOSUB 700
380 PRINT A$;
390 GOTO 340
```

3. Of course, you could have line 700 PRINT a warning message, flash a color, sound a beep, or take any number of actions before clearing the screen and proceeding.

ADJUSTABLE CLEAR NO. 1

The technique shown in this Note will permit you to clear any part of the screen and reuse it without disturbing the remainder of the display. It is not limited to the top of the screen.

Listing

```
.
.
.
```

```
110 PRINT " SHFT - CLR ";
120 FOR X = 1 TO 22 : PRINT "** (21 times) " : NEXT
130 LB = 2
140 LN = 7
150 GOSUB 600
160 FOR X = 1 TO 20
170 GOSUB 700
180 PRINT X
190 NEXT
.
.
.
600 GOSUB 620
610 FOR Y = 1 TO LN : PRINT " (21 spaces) " : NEXT
620 PRINT " HOME ";
630 FOR Y = 1 TO LB :PRINT " D CRSR "; : NEXT
640 RETURN
.
.
.
700 IF PEEK (214) > LN + LB - 1 THEN WAIT 197, 64, 64
    : GOSUB 600
710 RETURN
```

Analysis

110–120 set up a demonstration display.
130 sets the number of lines at the top to skip.
140 sets the number of lines to reuse.
150 calls the clear-and-position cursor subroutine.
160–190 demonstrate the use of the technique.
.
.
.
600 calls the last portion of itself to position the cursor.
610 PRINTs the specified number of blank lines ("erases").
620 homes the cursor.
630 positions the cursor at the start of the reuse space.
640 transfers back to the main program.
.
.
.
700 if the cursor line is greater than the total of skip and reuse, WAITs for a key to be pressed, and calls the "eraser" subroutine.
710 transfers back to the main program.

Use

This technique is very similar to the previous split screen Note except that it is not limited to the top of the screen. In a game, perhaps you need to give directions without removing the game field or board. In a tutorial, there are times when a chart or diagram should remain on the screen while the operator enters answers or reads directions. In such cases, the graphic display can stay on the screen as different questions, directions, etc., are successively PRINTed on the cleared portion.

Variations

1. The line on which the cleared space begins is dependent entirely on the value assigned to variable LB (line 130) before the subroutine is called.
2. The extent of the cleared space is determined by the value assigned to variable LN (line 140) before the subroutine is called.
3. Depending upon the nature of your program, it may be satisfactory simply to write over the characters on the selected portion of the display. If so, you can delete line 610.
4. You can change line 700 to have some other action performed when the cursor extends beyond the cleared space.
5. The 600-subroutine can be used just to erase a portion of the display without having any intention to reuse it. Delete line 630.
6. The technique used here for positioning the cursor was selected because it is readily understandable. Of course, you can use other methods, for example, the memory addresses of 209, 210, 211, and 214. See Chapter 4 on cursor control for further details.

ADJUSTABLE CLEAR NO. 2

The second Adjustable Clear routine is a specialized form of the previous Note. Here, only a portion of a line is cleared for reuse.

Listing

```
110 PRINT " SHFT-CLR";
120 FOR X = 1 TO 22 : PRINT "** (21 times)" : NEXT
130 LB = 12
140 LP = 15
150 GOSUB 600
160 GET A$ : IF A$ = " " THEN 160
170 IF ASC (A$) = 13 THEN 200 (continue program)
```

```
180 PRINT A$;
185 B$ = B$ + A$ : IF LEN (B$) > 21 - LP THEN WAIT 197, 64, 64 : B$
    = " " : GOSUB 600
190 GOTO 160
.
.
.
600 GOSUB 620
610 FOR Y = 1 TO 22 - LP : PRINT " SPACE " ; : NEXT
620 PRINT " HOME ";
630 FOR Y = 1 TO LB : PRINT " D CRSR "; : NEXT
635 POKE 211, LP
640 RETURN
```

Analysis

110–120 PRINT a demonstration display.
130 sets the line number of the space.
140 sets the line position of the space.
150 calls the space-clear subroutine.
160–190 demonstrate use of technique.
160 GETs a character from the keyboard buffer.
170 if the character is RETURN, continues the main program.
180 PRINTs the character.
185 concatenates the answer and checks the length of the answer to see that space is not exceeded; if so, WAITs for a keystroke, clears the answer and space.
190 transfers back to GET another character.
600 calls a portion of own subroutine to place the cursor at the start of space.
610 clears the space to the end of the line.
620 homes the cursor.
630 places the cursor at the start of the specified line.
635 places the cursor at the specified line position.
640 transfers back to the main program.

Use

This subroutine is used most often in programs requiring the operator to enter a word, a number, or the answer to a question. You can apply it any time one line or less must be cleared.

Variations

1. Line number and line position are specified in lines 130 and 140, respectively. If the values differ from those previously set, they must be respecified before the subroutine is called.

2. Space may be cleared for reasons other than for creating space for a response. That is determined by the program state-

ments following the GOSUB. Just to clear space, the subroutine need not be changed.

3. You do not have to clear all space to the end of the line. Change "21 – LP" and "22 – LP" in lines 185 and 610 to suit your needs. You can make these values into variables that can be changed as specified before each GOSUB.

THE RISING DISPLAY

Here is a subroutine that will allow you to have your display rise from the bottom of the screen just as the sun rises above the horizon. With it, you can "reveal" a fully PRINTed display.

Listing

```
.
.
.
(screen filled with alphanumeric or graphic characters)
120 GOSUB 900
.
.
.
900 FOR X = 131 TO 24 STEP -1
910 POKE 36881, X
920 FOR Y = 1 TO 75 : NEXT Y
930 NEXT X
940 RETURN
```

Analysis

120 calls the subroutine after the screen is PRINTed.

.
.
.

900 sets up a loop counting down from 131 to 24.
910 POKEs the value of the loop-counter (X) into the memory address 36881.
920 delays for a 75-count.
930 continues the loop until 24 is exceeded.
940 transfers back to the main program.

Use

The subroutine brings a fully PRINTed display — words, graphics, colors and everything — up from the depths of the sea, as it were. On the other hand, you can cause a display to sink slowly out of sight. Further, a variation given below will permit you to "pop" a complete display into view. These presentation techniques can be quite effective in almost any type of program: game, tutorial and so on.

Variations

1. You can make the display rise above its normal position if you POKE in values less than 24. The display will not rise high enough to disappear, however. Values greater than 131 (up to 255) will cause the display to go further below the screen but there seems little point in doing that.

2. The speed of the motion is controlled by the step value in line 900 and by the delay count (75) in line 920. The most effective speed depends on the nature of the material in the program.

3. For a "sinking" display, make this change:

```
900 FOR X = 24 TO 131
```

4. This subroutine will cause the display to appear to jump into existence:

```
110 POKE 36881, 131
.
.
.
900 FOR X = 1 TO 1000 : NEXT
910 POKE 36881, 24
920 RETURN
```

5. In a similar manner, a display can be made to wink out with this subroutine:

```
900 POKE 36881, 131
.
.
.
(erase the display while it is out of sight)
.
.
.
970 POKE 36881, 24
980 RETURN
```

6. In all the previous uses, except the disappearing displays, the effectiveness is usually lessened if the operator sees the display before it disappears to rise again. To avoid this situation, the dis-

plays should be "drawn" while the screen is out of sight. That's right — your VIC 20 does not care whether or not the display is in view! These statements show how it is done:

```
80  POKE 36881, 131
.
.
.
(statements to create the display just as though it were visible to the
operator)
.
.
.
120 GOSUB 900
```

You may be able to program a display in your mind but most of us have found it best to leave out line 80 until we have worked out the statements for the display!

THE UNFOLDING DISPLAY

When you want your display to unfold like a map from the top of the screen, use this subroutine.

Listing

```
.
.
.
230 POKE 36883, 0
(draw invisible design)
290 GOSUB 900
.
.
.
900 FOR X = 0 TO 46 STEP 2
910 POKE 36883, X
920 FOR Y = 1 TO 200 : NEXT Y
930 NEXT X
940 RETURN
```

Analysis

230 causes the display to be invisible while the design is created.
290 calls the subroutine.

.
.
.

900 sets up a loop incremented by 2.
910 POKEs the value of X into address 36883, each change causing one more line to appear.
920 delays for a count of 200.
930 continues the loop unless the value is over 46.
940 transfers back to the main program.

Use

This subroutine is used much like the preceding one. Some of the variations given as follows make it more versatile, however.

Variations

1. An existing display can be made to fold out of sight by changing line 900:

```
900 FOR X = 46 TO 0 STEP -2
```

2. This variation on the subroutine will cause the invisible display to appear in sections:

```
900 S = 36883
910 POKE S, 16
920 FOR Y = 1 TO 500 : NEXT
930 POKE S, 46
940 RETURN
```

This produces two sections, but you can have any number up to line-by-line.

3. To make the whole display invisible, simply POKE 36883, 0. A POKE of 46 will bring it all into view. Thus, you can pop a design back and forth at will. For a real attention-grabber, try this subroutine:

```
900 FOR X = 1 TO 25
910 POKE 36883, 0
920 FOR Y = 1 TO 100 : NEXT Y
930 POKE 36883, 46
940 FOR Y = 1 TO 100 : NEXT Y
950 NEXT X
960 RETURN
```

4. You can make only a part of the display invisible. This technique can be used in fun and games as well as some more serious endeavors. By this time, you have discovered that each two-step count between 0 and 46 represents one line. If you

```
POKE 36883, 40
```

the last three lines will be invisible but fully functional. They can be written to and drawn on quite normally. Any characters that are on the invisible lines will scroll up to the visible portion of the screen when more is written below.

THE SHIFTING DISPLAY

Not only can the display be moved up and down or into and out of visibility, but it can be shifted from side to side as well.

Listing

```
.
.
.
500 FOR X = 0 TO 17
510 POKE 36864, X
520 NEXT
530 FOR X = 17 TO 0 STEP -1
540 POKE 36864, X
550 NEXT
560 POKE 36864, 5
.
.
.
```

Analysis

500 sets up an 18-count loop.
510 moves the display progressively from left to right.
520 continues the loop until 17 is reached.
530 sets up a decrementing 18-count loop.
540 moves the display progressively from right to left.
550 continues the loop until 0 is reached.
560 places the display in its normal position.

Use

Of course, this routine can be used to attract attention. In addition, you can POKE a value other than 5 into 36864 in order to compensate for an off-center display.

Variations

1. Placing the routine within a loop can be hard on the eyes but it will attract attention:

```
490 FOR Y = 1 TO 25
```

```
.
.
.
555 NEXT
```

2. Adding little delay loops at lines 515 and 545 will produce a less frenetic action.

3. Values above 17 in address 36864 will cause strange things to happen. You may wish to try changing the 17 in one of the loops just for the experience.

4. These lines will cause the display to jump from side to side:

```
600 FOR Z = 1 TO 25
610 POKE 36864, 0
620 FOR Y = 1 TO 100 : NEXT
630 POKE 36864, 17
640 FOR Y = 1 TO 100 : NEXT
650 NEXT Z
660 POKE 36864, 5
```

SAVING THE DISPLAY

How often have you created a simply terrific display and wished there was some way to SAVE it? This routine will do just that. It is designed for tape but can be adapted easily for disk use.

Listing

```
10 DIM A$ (25)
.
.
.
50 POKE 36879, 42 : POKE 646, 7
60 FOR X = 7680 TO 8174 : PRINTCHR$(113); : NEXT
.
.
.
200 FOR X = 1 TO 25
210 FOR Y = 1 TO 21
220 Z = Y + 7679 + (X - 1) * 21 : IF Z > 8185 THEN 260
230 B$ = STR$ (PEEK (2))
240 IF LEN (B$) < 4 THEN B$ = CHR$ (32) + B$ : GOTO 240
250 A$ (X) = A$ (X) + B$
260 NEXT Y, X
270 OPEN 1, 1, 1, "FILE"
280 FOR X = 1 TO 25
```

```
290 PRINT #1, 1; A$ (X) ; CHR$ (13)
300 NEXT
310 CLOSE 1
.
.
.
410 OPEN 1, 1, 0, "FILE"
420 FOR X = 1 TO 25
430 INPUT #1, A$ (X)
440 NEXT
450 CLOSE 1
.
.
.
600 PRINT " SHFT CLR ";
610 FOR X = 1 TO 25
620 FOR Y = 1 TO 21
630 Z = Y + 7679 + (X - 1) * 21 : IF Z > 8185 THEN 650
640 POKE Z, VAL (MID$ (A$ (X), Y * 4 - 1, 4 ))
650 NEXT Y,X
```

Fig. 6-1 shows a flowchart for saving the display.

Analysis

10 dimensions the A$ array.
50 sets the colors of the screen and characters.
60 creates a design on the screen.
200 sets up a loop for the array.
210 sets up a loop for the display characters.
220 sets the variable Z to the computed value of the current screen position, then bypasses the reading if the value is beyond the Screen RAM area.
230 assigns to B$ the string representation of the value held in location Z.
240 assures that all B$ values are 4 spaces in length.
250 concatenates B$ to A$.
260 continues both loops until requirements are met (At this point, all Screen RAM values are stored in the A$ array. The SAVE routine follows. Be sure the machine is ready. If necessary, insert: 265 PRINT "READY TAPE" : WAIT 197, 64, 64).
270 identifies and OPENs the correct tape file.
280 sets up a loop for the array.
290 prints each array element to tape.
300 continues the loop until all elements are printed.
310 CLOSEs the file.
.
.
.
(The LOAD routine follows.)

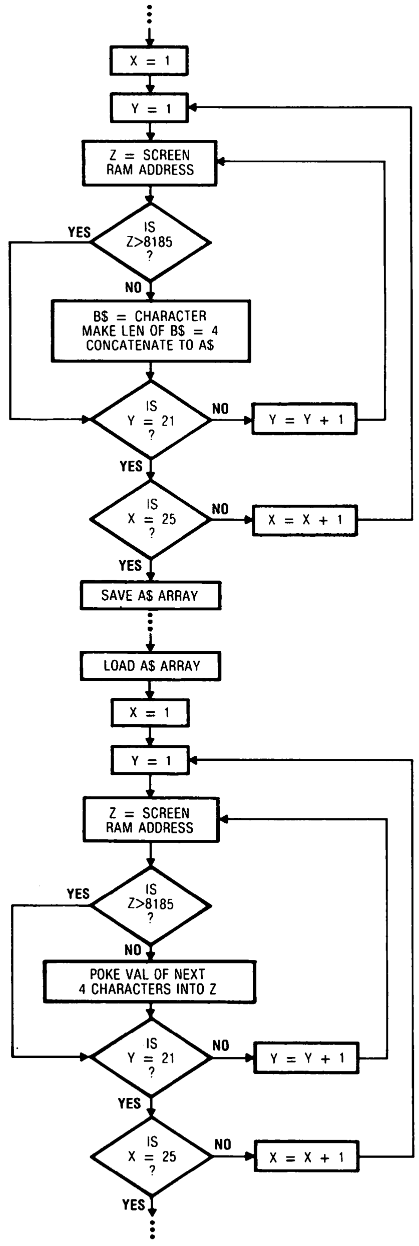


Fig. 6-1. Flowchart for saving the display.

410 OPENs the correct tape file.
420 sets up the array loop.
430 stores each array element from the tape to the A$ array.
440 continues the loop until the array is filled.
450 CLOSEs the file.

.
.
.

(The following routine shows how to use the array to recreate the original display.)
600 clears the display and homes the cursor.
610 sets up the array loop.
620 sets up the character loop.
630 assigns to variable Z the current Screen RAM address and skips ahead if it exceeds the boundaries.
640 POKEs into address Z the numerical value of the current 4-character section of the current A$.
650 continues the loops until requirements are met.

Use

There are many occasions when you wish to preserve an entire display or a part of one. The previous routines will SAVE it to tape, LOAD it from tape, and re-create the display.

Variations

1. As given, this series of routines functions on the entire display screen. It can be modified to operate on any desired part of the screen by proper selection of the loop sizes to "cover" that part.

2. You will have noticed that the original colors are not retained. The re-created display takes on whatever colors happen to be in the Color RAM when it is PRINTed. The following changes and additions will SAVE the colors and re-create the design exactly like the original. It does require more RAM than that built into the VIC 20. The numbers given below are for a VIC 20 with an 8K RAM cartridge.

In lines 60, 220 and 630, change 7680 to 4096; 8170 to 4590; 8185 to 4601; and 7679 to 4095.

```
10  DIM A$ (25) , D$ (25)
70  FOR X = 1 TO 100 : Y = INT (RND (0) * 500) : POKE 37881 + Y,
    RND (0) * 8 : NEXT
```
(adds some random color to the sample display)

```
232 C = PEEK (Z + 33792)
234 IF C > 7 THEN C = C - 8 : GOTO 234
236 C$ = STR$ (C)
245 IF LEN (C$) < 4 THEN C$ = CHR$ (32) + C$ : GOTO 245
```

```
255 D$ (X) = D$ (X) + C$
290 . . add to the end . . ; D$ (X) ; CHR$ (13)
430 . . add to the end . . , D$
645 POKE Z + 33792, VAL (MID$ (D$ (X), Y * 4 - 1, 4 ))
```

CHAPTER 7

"Housekeeping" in RAM

In any computer operating system, there must be areas of RAM that are used by the system itself. There are many values such as pointers, links, clocks, and buffers, which vary as the machine operates. Since they do vary, they cannot be held in ROM. Your VIC 20, like other computers, reserves sections of RAM for its own use. These areas go by a variety of names but, most commonly, they are referred to as "reserved RAM" or "housekeeping RAM."

You should be familiar with the housekeeping areas if you wish to make the most of the capabilities of your VIC 20. Many operators and many programs, both BASIC and machine language, function quite well without making any access to these special RAM areas. Knowledge of them, however, will enable you to do many things more efficiently and to do some things that cannot be done otherwise.

The first housekeeping area we will discuss is the first 1K of memory. It is the section labeled "Reserved RAM" in Table 5-1, the general memory map found in Chapter 5. In every VIC 20, regardless of how much memory has been added, the first 1K of RAM is used in the same way — to hold data for use by the operating system.

Normally, that first 1024 bytes of RAM are unavailable to BASIC programs or to BASIC direct commands except through the PEEK and POKE statements. With PEEK, of course, you can see/get the value stored in any RAM or ROM location. POKE, on the other hand, allows you to insert a value into any RAM (only) location. Since reserved areas are RAM, you can use both PEEK and POKE with this memory.

There is one warning that you must observe in dealing with housekeeping RAM. Use caution when poking values into this area of memory. Do not experiment in this manner when there is a program or data in the machine that would be difficult to reproduce. One simple POKE can cause a crash of the system, resulting in a loss of program and data when the system is restarted.

Do your experimenting with reserved RAM when you have nothing valuable in the VIC 20. It is too easy to lose it all when you are not SURE of what you are doing. Certainly, this applies to POKE only. You can PEEK anywhere, at any time, without danger to memory contents.

THE ZERO PAGE

The uses of the zero page, the first 256 locations (0-255), are quite interesting. While all 256 addresses are useful, certain ones are especially so. Many are used in Notes throughout this book. A number of useful addresses are given in Appendix A and we shall discuss some of them.

The following little program will allow you to check (PEEK) addresses at will. It can be used as shown or it can be made into a subroutine in an existing program in which you wish to check memory values. Note that it gives you the value in the specific location you select AND the value in the next location. Further, in case these two numbers are two bytes of a single number, you are given the equivalent of the two values.

```
10 INPUT "ADDRESS " ; A
20 B = PEEK (A)
30 C = PEEK (A + 1)
40 PRINT SPC (22) A " = = > " B
50 PRINT A + 1 " = = > " C
60 PRINT SPC (30) "DEC= " C * 256 + B
70 PRINT SPC (66)
80 GOTO 10
```

You can use this program to "look into" the reserved RAM discussed here.

Among the most useful addresses are those from 43 to 52. These are "pointers" to RAM locations that are used by a BASIC program.

Locations 43/44 point to the start of the RAM area where BASIC programs are stored. If you run the little program above and answer ADDRESS ? with 43, you will see this display:

```
 43 = = > 1
 44 = = > 16
DEC = 4097
```

The number stored at address 43 is 1 and at 44 is 16. When these low and high bytes are converted, the result is 4097, a number that should be familiar to you as the beginning of the available programming area in a 5K VIC 20. It is easy to see why addresses 43 and 44 are referred to as "pointers" to the beginning of BASIC programs. (With an 8K memory cartridge installed, the numbers would be 1, 18, and 4609 as expected.)

To give you an idea of the value of such knowledge, let's look at but one example of its use. In one of the Notes (Append in Chapter 8), there are directions for loading a second (or third) program into the VIC 20 without overwriting the first as is done normally. This is a very handy technique because it allows you to do such things as append a set of subroutines to a program without having to retype them. As you will see in that Note, you change the 43/44 pointers to make the VIC 20 "forget" the first program. Then you LOAD the subroutines and put the original values back in 43/44 so the whole thing runs as one complete program. Neat!

Addresses 45/46 are pointers to the beginning of the variable storage area just above the BASIC program. The location pointed to is actually the first memory address above the program. Among other uses, you can subtract the value of the bytes in 43/44 from the value in 45/46 and find out just how many bytes your program takes up. In appending an existing program to one in the memory, the 45/46 values are put into 43/44 to tell the VIC 20 where to start loading the second program.

The array storage area begins as indicated in 47/48 and ends at the address pointed to by 49/50. String storage space begins at the 51/52 address (normally the top of memory) and moves down as it is used. Incidentally, when the addresses in 49/50 and 51/52 meet, your program has run out of memory!

Another particularly important pointer is found in 55/56, which gives you the top of RAM memory available to your program. These values must be changed when you wish to protect some high memory from program use, for example, to keep your BASIC program from destroying a machine language routine you have placed there.

The foregoing pointers are so important to the programmer that they are listed separately in Table 7-1.

Addresses 57/58 contain the number of the currently executing line and 59/60 contain the previously executed line number. These can be useful in controlling the execution of a complex program.

Often it is advantageous for the program to determine quickly that a specific key has been depressed. This is especially true in machine language. Addresses 197 and 203 appear to be duplicates. In any case, they hold a number corresponding to the depressed key only as long as that key is held down. Think how a PEEK at one of these locations could be useful in a test of reaction speed or a fast-moving game. The contents of 197 and 203 will be 64 when no key is depressed. The number found there for each key is given in Table 7-2.

ABOVE THE ZERO PAGE

There are, of course, three more "pages" of reserved RAM above the zero page before we reach the end of that first (bottom) 1K of memory. After all, it takes four pages of 256 locations to equal 1024.

Some of the more useful addresses in this area are given in Appendix A. Many are used in the Notes in this book. You should experiment with these locations in order to see just how they can be used. Then you will be ready to put them to work in your programs.

Your VIC 20 contains many other useful addresses beyond the first 1K reserved section. These will be taken up as they are used in following chapters and notes. Probably you will find those related to sound and color to be the most important to you in the early stages of programming.

Table 7-1. Program Space Pointers in Reserved RAM

Decimal Address	Description
43/44	To the start of BASIC (43 = low byte; 44 = high byte)
45/46	To the start of numeric variables (one location above end of program)
47/48	To the start of arrays (one location above end of numerics)
49/50	To end of arrays
51/52	To bottom of string storage space (moves down from top of free memory)
55/56	To top of free RAM (to protect RAM, 51/52 and 55/56 must be changed)

Table 7-2. Table of "Current Key" Values in 197 and 203. A Value of 64 Indicates That No Key Is Being Pressed

Val	Key	Val	Key	Val	Key	Val	Key
0	1	16		32	space	48	Q
1	3	17	A	33	Z	49	E
2	5	18	D	34	C	50	T
3	7	19	G	35	B	51	U
4	9	20	J	36	M	52	O
5	+	21	L	37	.	53	@
6	Eng pound	22	;	38		54	up arrow
7	INST/DEL	23	LR cursor	39	f1	55	f5
8	lft arrow	24	STOP	40		56	2
9	W	25		41	S	57	4
10	R	26	X	42	F	58	6
11	Y	27	V	43	H	59	8
12	I	28	N	44	K	60	0
13	P	29	,	45	:	61	–
14	*	30	/	46	=	62	CLR/HOME
15	RETURN	31	UD cursor	47	f3	63	f7

SELF-ADJUSTING PROGRAMMING

In Chapter 5, it was pointed out that many memory locations change when additional RAM is added to the VIC 20. You should keep in mind that the functions of the first 1024 addresses do not change. Location 646 changes the color of the printed character in any memory configuration.

Basically, three areas shift with added RAM: user RAM (for programming), Screen RAM, and Color RAM. If your program uses PEEks or POKEs into any specific address in even one of these areas, it will not run properly when loaded into a computer of another size.

Certainly one way to avoid this problem is to stay away from PEEks and POKEs. These commands are quite unnecessary in many programs; thousands have been written without them. Yet, if you place these commands off-limits in all your programming, you will severely restrict the variety and speed of graphics and color in those programs. Sooner or later, that restriction will chafe and you will use them.

When that happens, design your program so that it will run properly with any amount of RAM plugged in (or out). Making your program self adjusting is not difficult. You can do it by making your addresses in these areas relative instead of specific. Here's how.

PEEK and POKE in the BASIC Program Area

You already know that zero page locations 43/44 hold pointers to the beginning of the BASIC programming area. Normally, they point to 4097 in an unexpanded VIC 20. Suppose you POKEd into your program at address 4202 to pack it, for example (see Chapter 13). In a machine with an additional 3K of RAM, that POKE would go way above where you intended. With +8K or more of added RAM, it would go right into the Screen RAM and ruin your display.

On the other hand, if you POKEd at address PEEK(44) * 256 + PEEK(43) + 105, the value would go into the same place regardless of how much RAM was in the VIC 20. If you plan to make several POKEs, set AD = PEEK(44) * 256 + PEEK(43) and then make your POKEs in the form of

```
POKE AD + (whatever), value
```

PEEK and POKE in the Screen RAM

In order to create or change a display, it is often more satisfactory to POKE values into the Screen RAM instead of using a series of PRINT statements. You know that the absolute location of Screen RAM does vary. The advice for writing self-adjusting routines is quite similar to that discussed previously: use relative instead of absolute addresses.

In this case, the addresses are relative to the beginning of Screen RAM, of course. A quick glance back into Chapter 5 at Table 5-1 shows that the beginning address does not change until you reach the +8K level of additional RAM, at which time Screen RAM shifts to 4096. The need for this shift can be determined by reference to our old favorite 43/44 addresses.

As an example of self-adjusting POKE into Screen RAM, suppose you wished to POKE a value into the first position of the 11th display line. You might be tempted to write

```
POKE 7900, value
```

but resist the temptation because it would not be transportable to all versions of the VIC 20.

Self-adjustment will require a routine similar to this

```
20 S = 7680
30 AD = PEEK(44) * 256 + PEEK(43)
40 IF AD > 4500 THEN S = 4096
.
.
.
160 POKE S + 220, value
```

"Value" depends upon what character/graphic you want on the display. The various values are given in Appendix C. Note that they are not the ASCII values.

In effect, lines 30 and 40 tell the program to change the start of Screen RAM from 7680 to 4096 when the BASIC area begins above 4500. If every Screen RAM address is stated as in line 160, the program will be completely transportable.

Note that line 30 is the same as given earlier under POKing into the program area. If both types of POKEs are made, that statement need be made only once.

PEEK and POKE in the Color RAM Area

By this time, you are surely ahead of me! This potential problem is handled just as are those previously. See if you can determine how the following lines work. Note that they take care of Screen as well as Color RAM.

```
.
.
.
```

```
25 C = 38400
30 AD = PEEK(44) * 256 + PEEK(43)
40 IF AD > 4500 THEN S = 4096 : C = 37888
.
.
.
180 POKE C + 230, value
```

In this case, "value" is any digit between 0 and 7 (see Chapter 9).

SUMMARY

The values that are stored in the reserved section of RAM are quite useful to the BASIC programmer. Addresses there can be PEEKed to get information on which to base program action. Many can be POKEd with new values to achieve some desired effect.

Diligent use of reserved RAM will enable you to write faster and more attractive programs — in some cases, even saving valuable space in the program area. Certainly, it will make possible programs that could not be written without its use.

Study of, and experimentation with, reserved RAM will pay rich dividends in the quality of your work.

CHAPTER 8

Notes: PEEK and POKE

Two of the most powerful commands available to you are PEEK and POKE. With a knowledge of the purpose of the most important memory addresses in your VIC 20, you can use PEEK and POKE to do many things that otherwise would be impossible. In addition, many "screen" and other operations can be done much more efficiently by using these commands. A full understanding of the use of PEEK and POKE is an essential component of your programming arsenal. Chapters 7 and 13 will provide you a firm foundation for this understanding.

We have used PEEK and POKE in routines throughout this book. In many instances, no special point was made of the commands. Now, we will discuss specific uses and show you some of the "tricks of the trade." First, however, here is a quick review.

The PEEK command "looks" into a specified memory address and determines its contents. The form is

```
PEEK (address)
```

but if you enter this at the keyboard or place it in a program, you will get a "? SYNTAX ERROR" on the screen. In this case, the

VIC's message reminds you that you did not tell it what to do with the information. In other words, you told the VIC 20 to look into the address and the VIC 20 responds, "So what!" You may write more properly, of course, PRINT PEEK (address) or another command.

Most frequently, PEEK is used as in one of these examples:

```
A = PEEK (209)
W = 60 * PEEK (162)
IF PEEK (8142) < 33 THEN PRINT "NOTHING"
```

In each case the VIC 20 is told what to do with what it found. The first statement instructs the VIC 20 to assign to variable A the value of the contents of memory address 209. In the second, variable W is to be assigned 60 times the value found. The third statement causes some special action to be taken based solely on the contents of a memory location.

PEEK, then, can be used anywhere a number would be used in program statements. It is powerful because of the wide range of things the VIC 20 is doing down there below the keyboard where there is a veritable beehive of activity even while you go out for a cup of coffee. It buzzes along (quietly, of course) at a very rapid pace every moment that it is not turned off. The PEEK command can tap selectively into that activity just as a radio receiver can tap into the hundreds of thousands of radio waves crisscrossing your antenna and allow you to hear the beep-beep of a satellite, the local disk-jockey, or a symphony from London. Powerful? . . . you bet it is!

The memory location that is PEEKed can be anywhere from 0 through 65535. You can look into any and all of the VIC 20's possible memory locations — yes, even if there is no memory of any kind actually there. You cannot harm, distort, or change in any way the contents of any address by PEEKing at it. With an easy mind, you may PEEK around to the full extent of your curiosity.

POKE, on the other hand, is a different kettle of fish. POKE and PEEK are generally associated in programmers' minds. In the sense that they both deal with memory locations, they are alike. In other respects, they are different in what they do and the effects they generate.

Though you may POKE any address from 0 through 65535, we urge caution in doing so. To experiment, POKE all you like, where you like, but be sure that you do not have any irreplaceable material in the program memory when you do so. You cannot hurt the VIC 20 with a POKE to any location. Indiscriminate POKEs, however, can cause programs to be lost, because the machine "locks up" and you lose all control. When this happens, you can regain control only by turning the VIC 20 off and back on.

The form of the POKE command is

POKE address, value

where the value may be any number from 0 to 255. This command tells the VIC 20 everything about what it is to do; i.e., place the stated value into the location whose address is given (and, in the process, wipe out the current contents). Thus, you can see why you must be careful with POKE.

A POKE to a ROM address will do no harm. Also, it will do no good because the contents of a ROM location cannot be changed in this way (nor in any other readily available way). To RAM locations, POKEs can wreak havoc or be a terrific asset — it all depends upon what is POKEd where.

You can POKE into the program RAM area to change (ruin or improve) a resident program and some of our Notes do just that. A POKE into areas of RAM used by the VIC 20 will often change its operating characteristics for better or for worse, and many such examples are found in this book.

The moral of the PEEK and POKE story is to use the former as you will, but be careful about the POKEs until you are sure what will happen. PEEKs can teach you a lot about what the VIC 20 is doing — well, so can POKEs even when they lock up the machine!

With these commands used so frequently throughout the other chapters, it is not practical to attempt to go into all their uses here. We will treat only a few topics of special interest and value here. In addition, you should be on the lookout for them elsewhere. Most of the memory addresses useful in programming are listed in Appendix A, to which you will be referring often during your own programming.

TIMERS AND DELAYS

There is an old saying that time is money. Well, it isn't money in programming unless you are in that business or, of course, unless you are taking bets on the speed with which one can successfully complete a program — but forget that last part! Even though it is not money, time is often of vital importance to the programmer as other Notes illustrate.

Since time can be important, you must have a way of measuring it. Preferably, there should be several ways so that you can choose the one best suited for a particular application. Fortunately, your VIC 20 gives you a choice of methods for determining lapsed time.

One "timer" is the simple and commonly used FOR/NEXT delay loop that looks like this:

```
FOR X = 1 TO 1000 : NEXT
```

The 1000 was chosen for this loop because it provides a delay equal to approximately 1 second (for greater accuracy, try 1025). Often, you will see this delay written in one of these forms:

```
FOR X = 1 TO A * 1000 : NEXT
FOR X = 1 TO A * 60000 : NEXT
```

The loop is placed in a subroutine and variable A can be reset each time before it is called. In the first case, the delay extends for approximately A-seconds and, in the second, for A-minutes.

So slight a change as placing NEXT on the following line will lengthen the time through the loop a bit. If you put any line inside the loop (even REM), the timing will be changed considerably — the amount of change depending upon what the VIC 20 must do in each pass through the loop.

The FOR/NEXT loop method works quite adequately when the task is to provide a delay while the operator reads a screen, for example. It does have its limitations, however, because while the VIC 20 is counting, it cannot do anything else. Suppose you want a timer to prevent the operator from taking too long to enter a response. Then, you want to measure time and allow use of the keyboard simultaneously. The simple loop cannot do this with any degree of accuracy so you must use another method.

Let's see what we can learn from this little program:

```
10 PRINT "TI$ = " ; TI$
20 TI$ = "000000"        : REM — 6 zeros
30 GET A$ : IF A$ = "" THEN 30
40 PRINT "TI$ = " ; TI$
50 PRINT "TI = " ; TI
60 PRINT "TI/60 =  " ; TI / 60
70 PRINT : GOTO 40
```

The first time you run it, line 10 will tell you how long your VIC 20 has been turned on! There are two digits for hours, two for minutes, and two for seconds in the form of HHMMSS. Line 20 resets the clock to zero and line 30 waits for you to press any key.

A keypress causes three numbers to be displayed (and the execution goes back to wait again). You know that the first of the three numbers (TI$) is the time since the clock was reset. The third number is equal to the number of seconds shown on the first unless you waited more than a minute, in which case you will have to convert to seconds. (Actually, there may be a one second difference because you may catch the clock in the process of updating). The second number (TI) is 60 times the number of seconds.

So you know that TI measures time in 1/60 second intervals and, when divided by 60, the result is the total elapsed seconds. Press a key a few more times and check this information out for yourself.

There are three memory locations where you can PEEK at the time. Those addresses and their contents are:

162 increments every 1/60th of a second.
161 increments every 4.2 seconds (approximate).
160 increments every 18.2 minutes (approximate).

Because each location can count no higher than 256, the maximum count in 162 is 4.2 seconds. In 161, it's 18.2 minutes and in 160, it's 77.6 hours. All three counts are restarted when the main clock is reset as shown previously.

When using the values in these addresses for delays and/or timers, you must be careful about the statement that checks them. For instance, if the program is waiting for input from the user and it checks a time address occasionally, you cannot use

```
375 IF PEEK (161) = 15 THEN . . .
```

The statement might check 161 when it held 14 and not get back to it again until its value is 16 or 17. Then, the THEN part of the statement would never be executed. Your statement should read

```
375 IF PEEK (161) > 14 THEN . . .
```

You can use the contents of these locations for a variety of purposes. Of course, they will provide very accurate timing. As another example, consider that they provide an excellent source of a random seed for the random number generator, or even a

random number itself.
For further Delay Notes, see Chapter 16.

REPEATING KEYS

There will be times when you wish to control the repeat-action of keys. Under normal, power-up conditions the repeating keys are limited to SPACE, INST/DEL, and the two CRSR control keys. Your program may require that these keys not repeat when held down or that all keys do so.

A memory location controls the key-repeat function in this manner:

POKE 650, 0 — normal repeat action
POKE 650, 100 — no keys repeat
POKE 650, 128 — all keys repeat

The indicated conditions continue until they are changed by another command or until the VIC 20 is turned off and back on. These commands may be included in your program statements and/or issued directly from the keyboard.

RESERVED MEMORY NO. 1

As your programming grows more advanced, there will be many times when you will have one or more machine language subroutines. These must be placed in a protected part of memory where your BASIC program will not interfere with them. There are two ways to reserve memory and the most common is discussed here. The second method will be found in the following Note.

You may recall from Chapter 7 that memory locations 55 and 56 contain pointers to the top of RAM memory. In a 5K VIC 20, the values found there are 0 and 30, respectively. If you change those low- and high-order bytes, your BASIC program will not recognize the existence of anything above the memory to which they point.

For example, if you POKE 56, 29 the entire top 1K will be off limits to BASIC except by the USR function which is used to call machine language subroutines and by PEEK and POKE which can be used to store and retrieve data. Thus, the top 1K is available for such purposes and BASIC will not write over it. Of course, you can reserve as little or as much memory as you need by putting the appropriate values in 55 and 56.

Changing the 55/56 values may be done in keyboard commands or in program statements. *If the latter, you must POKE the same values into addresses 51 and 52 which tell the VIC 20 where to store strings* (and that always begins at the top of memory). In addition, the change must be made early in the program or you may "trap" values up there that would be unavailable for use by BASIC. In other words, if your program has stored the value of A$ before you lower the memory pointers, the program cannot use A$ because it cannot find it. Change the top of memory at the beginning of the program.

RESERVED MEMORY NO. 2

A less commonly used but equally effective method of protecting space for data storage and/or a machine language subroutine is to place it below the BASIC program. Addresses 43 and 44 contain the pointers to the bottom of BASIC RAM. Normally, the values there are 1 and 16. If these values are increased, a subsequently LOADed program will be placed above the newly specified bottom of RAM. The area below will be available, yet protected from interference by your program except by USR, PEEK, and POKE.

Not only is the bottom of memory an alternate to the top for reserving space, but the two may be used in the same program. You could have some protected memory at the top for a machine language routine and at the bottom for data that is to be passed to the next program.

Be aware that the bottom memory must be set before the program is LOADed. Do so after LOADing and your program will not run properly, if at all.

APPEND

It is often advantageous to append a program or subroutines to the end of an existing program. Being able to do this will save you a great deal of re-typing. Your machine has no built-in Append function but you can do it quite easily. It is important, however, that this technique be used before a program is RUN or there is possibility of unpredictable results. Further, it is good practice to be sure the appended statements have higher line numbers than those in the first program.

After LOADing a program, execute the following commands from the keyboard:

```
POKE 43, PEEK (45) - 2 : POKE 44, PEEK (46)
```

If, as usual, the screen now displays the normal READY sign, all is well and you may proceed to the next paragraph. If luck is against you and an ERROR statement is displayed, it means that PEEK(45) – 2 was a negative value; i.e., 45 contained a value less than 2. In such cases, you must execute this command:

```
POKE 43, PEEK (45) + 254 : POKE 44, PEEK (46) - 1
```

This accomplishes the same purpose, of course, but both the high and low order bytes are changed appropriately.

The *bottom* of your RAM is now considered to be *just above* the first program. LOAD the subroutine package or the second program in the normal way and it goes above (after) the first. To fit the pieces together, you must now lower the beginning pointers to their original values with this command:

```
POKE 43, 1 : POKE 44, 16
```

If you are using added RAM, change the values as appropriate.

Now the second LOAD is firmly attached to the first and you have, in fact, one program. If you LIST it, you will see that it is together. You can RUN and SAVE your "new" program in the normal manner.

DUAL PROGRAMS

There may be occasions when you find it useful to have two independent programs resident in the memory at the same time. It may be that one is a utility that operates on the other or perhaps you just want both of them handy.

LOAD program No. 1 and execute these commands:

```
PRINT PEEK (45) , PEEK (46) : POKE 43, PEEK (45) : POKE 44, PEEK (46)
```

and make a note of the values from addresses 45 and 46. Now, LOAD program No. 2

If you enter RUN, only the second program will be executed. POKE the original values (1 and 16) back into locations 43 and 44 in order to RUN program No. 1 as though the other program was not there. If you POKE the previously noted values from 45 and 46 into 43 and 44, the second program is back. Thus, you can RUN either program selectively.

As you have discovered, the difference between this and the Append technique is only a difference of two memory locations in where the second program is LOADed. Here, we leave the two zeros which tell the VIC 20 that the first program is ended. With Append, we write over those two zeros because we don't want execution to stop there. See Chapter 13, Statement Structure, for further details.

Of course, more than two independent programs can be LOADed simultaneously into the memory. It is only necessary to note the proper values for each program and to be sure that you do not exceed the capacity of the memory.

MERGE

To merge two programs or parts thereof is to "interleave" their program statements according to their line numbers. For example, if program No. 1 has lines 10 and 20 and program No. 2 has a line 15, a true merge would place line 15 between lines 10 and 20.

Unfortunately, we have found no way to merge programs in one step, but it can be done in three. The technique requires that the second program be appended to the first by following the previous instructions. Then the line numbers of the second are changed by the regular editing process — just type over the numbers and the lines will be inserted in the proper places. The third and final step is to delete the original lines of the second program.

This merge method is somewhat cumbersome but, even so, it can save you a lot of time if there are many lines involved.

ACCOMMODATION TO VARIOUS MEMORY SIZES

As you know, the locations of BASIC RAM, Screen RAM, and Color RAM change as varying amounts of memory are added to your VIC 20. You must take this into account if your programs are to be transportable; i.e., if they are to function in a machine of any size. Of course, the amount of memory can be determined easily with a PEEK to address 44.

You can build in transportability with this technique. First, place the following lines at or near the beginning of your program:

```
nn AB = 4096 : AS = 7680 : AC = 38400
nn X = PEEK (44) : IF X < 16 THEN AB = 1024
nn IF X > 16 THEN AB = 4608 : AS = 4096 : AC = 37888
```

Now, AB is the start of BASIC RAM; AS is the start of Screen RAM; and AC is the start of Color RAM.

If you will not be using PEEK or POKE with one or two of these RAM areas, omit the corresponding parts of the previous lines. There is no need to set up an address that will not be used and to do so will take up valuable memory space in the program and in the variables storage area.

All PEEK and POKE references to addresses in these three areas should be stated in relative terms; i.e., how far above the beginning of the areas they are. As an example, consider that

```
POKE AS + 10, 34
```

will place a quotation mark in the tenth position of the top screen line on a computer of any size.

Of course, there are several techniques that will automatically adjust relative addresses to memory size. Another location that is used frequently is 648, which holds the screen memory "page." Here is a statement that permits relative addressing of Screen (SC) and Color (CO) RAM:

```
nn SC = 256 * PEEK (648) : CO = 38400 : IF SC < 5000
THEN CO = 37888
```

DATA TRANSFER BETWEEN PROGRAMS

When using a VIC 20 with little or no added memory, longer programs cannot be accommodated. It would be possible to break the long program into smaller pieces and LOAD and RUN each one in succession if there were some way to transfer information from one part to the next. This need to pass data from one program to another can arise regardless of the size of the VIC 20 memory or the length of the program. Fortunately, there is a way to accomplish just such transfers.

Let's suppose that you want to pass 24 bytes of data from one program to another. The procedure is to have the first program reserve some memory for the purpose by using one of the methods given previously (we will use Reserved Memory No. 1 in this example). Then, the program POKEs the information into that memory space. When the second program is LOADed and RUN, it retrieves the information from there with a few PEEKs and proceeds on its way. Here is how to do it.

First Program

```
10 POKE 51, 232 : POKE 52, 29 : POKE 55, 232 : POKE 56, 29 : REM —
reserves 24 byte-spaces
.
.
.
— assumes the data is in A(n) —
350 FOR X = 7656 TO 7679
360 POKE X, A(X)
370 NEXT
```

Second Program

```
80 FOR X = 7656 TO 7679
90 A(X) = PEEK (X)
100 NEXT
```

Note that the second (and any subsequent) program does not have to reserve the memory as was done in the first program — it does not change when other programs are LOADed and RUN. Further, the example assumes that the values of A(n) do not exceed 255. If the values are larger, they will have to be POKEd digit by digit or in some other manner.

If the data had been in a string, the following changes would be needed:

```
360 POKE X, ASC (MID$ (A$, X - 7655, 1) )
```

and

```
90 B = PEEK (X)
95 A$ = A$ + CHR$ (B)
```

In this case, we are using the ASCII values of each letter in turn to be POKEd into the protected area. Later, the second program gets those values with PEEKs, changes them back to letters, and concatenates them into A$. The variable A$, then, is identical to its structure in the first program.

KEY DISABLES

For various reasons you may wish to disable a key or function of the VIC 20. It may be to prevent the user from accidentally hitting a key or to prevent easy access to the program. Here are a few which will prove useful.

LIST

This command will disable the LIST function though a program will RUN normally:

```
POKE 775, 200
```

If the operator tries to LIST a program, the operator will get an error statement but can reRUN the program. If the value 198 is POKEd, instead of 200, trying to LIST the program will lock up the VIC 20 so that it must be turned off and on to regain control.

The normal value in 775 is 199 and either this must be POKEd or the machine will have to be re-powered to enable regular LISTings.

STOP

The normal value in 808 is 112. The STOP key can be disabled by

```
POKE 808, 0
```

Save

If 134 (instead of the normal 133) is POKEd into 818, the user will receive only a "DEVICE NOT PRESENT" message when he or she tries to SAVE the program.

All Keys Except STOP

The value in 649 controls the size of the keyboard buffer. Normally, it is set for 10 characters and sometimes you may want to decrease that number. If you reduce the value to 0 (POKE 649, 0), the buffer can hold no characters and the VIC 20 cannot get any from the keyboard. That effectively removes the keyboard from the machine so the operator can enter nothing except STOP unless you have disabled that, also.

Be careful about disabling the keyboard by typing in the POKE 649, 0. If you do that, you lose access to the VIC 20 and can enable the keys again only by pressing **STOP**-**RESTORE** or by powering down and restarting the machine. Usually, the keyboard is disabled by an early program statement and enabled again before the program ends. Of course, you can turn it on and off at will within a program.

CHAPTER 9

Color

Now and then someone says that color is just a "frill" that can be added to a program. They say that color is pretty but it contributes nothing to a program's effectiveness. Such a view is narrow, to say the least. Even "pretty" can be worthwhile as long as it does not get in the way of the purpose of the program.

One of the last places one might expect color to be useful is in a staid business program — payroll, for example. Yet even here, it can contribute to understanding and accuracy. When carefully applied, color can function as a device to focus attention on, and add emphasis to, words and actions. A graph in which the bars or pie sections are in colors is far clearer and more understandable than one in shades of grey or black-and-white designs.

In programs of other types, color can and does have much greater value. Color can help to establish the mental attitude of the operator. It can help to generate interest and add to the sense of reality. Color can contribute to the excitement of a program.

It is true, however, that color can be a frill. Moreover, it can decrease the effectiveness of a program if it is used indiscriminately. Such adverse results are more likely to occur when color is an afterthought instead of planned as an intricate part of the program.

We cannot tell you how to use color with discretion. We can tell you only various methods of creating and using color. You must supply the discretion and that is most easily done as the program is being designed.

COLOR GENERATION

There are four primary ways to determine the color of all or part of the display. They are to use PRINT statements (from keyboard or program), to POKE codes into selected Reserved RAM locations, to POKE color codes into the Color RAM, and to POKE codes into locations associated with the chip that controls the video signal. We shall consider each of these methods in turn.

Color With PRINT Statements

You know that you can change the color of characters that are to be PRINTed from the keyboard. This is done by pressing CTRL and a digit from 1 to 8. Subsequently displayed characters will be in the color indicated on the front of that digit key. If you press CTRL-6, for example, further PRINTing will be in green letters.

This same action can be executed within a program but it must be done in a PRINT statement. The statement

```
35 PRINT " CTRL-8 "
```

will cause the subsequent characters to be yellow. Note that the instruction must be enclosed in quotation marks and, thus, be a part of a PRINT statement, though it need not be alone as in this example:

```
35 PRINT " CTRL-8 Who is there ? "
```

Note further, that when CTRL and 8 are pressed within quotation marks, a reversed Pi symbol appears on the screen. This symbol does not appear when the command is given from the keyboard. When the program is RUN, the symbol is not PRINTed on the screen and the space it occupies in the statement is filled in as though it did not exist.

Once the color setting statement is executed, that color remains in effect until it is changed. Color changes may be made several times within one quotation if desired. Thus, you can have each word or even letter in different colors.

Characters may be PRINTed in reverse colors, also. CTRL and 9 activate REVERSE ON; that is, the background color becomes the character color and vice versa. When this is done in a PRINT statement, a reverse R appears between the quotation marks. REVERSE OFF (CTRL and 0) can be executed in the same manner but it also turns off automatically at the end of a statement unless a final comma or semicolon is attached.

Color With POKEs Into Reserved RAM

There is one location in reserved RAM that is of special interest when dealing with color. That is address 646 which was mentioned briefly in Chapter 7.

You can POKE a value into 646 to set/change the color of all subsequent display characters. The effect is the same as that achieved by using CTRL-digit as discussed previously. The form of the command/statement is

POKE 646, value

and it may be entered from the keyboard or placed in a program statement.

The "value" may be any number from 0 to 255. Numbers 0 through 7 are the standard colors found on the digit keys except that the value for a given color is one less than the digit on the key. The values and corresponding colors are:

0	BLACK	4	PURPLE
1	WHITE	5	GREEN
2	RED	6	BLUE
3	CYAN	7	YELLOW

The next eight digits (8-15) place the displayed characters in the "multicolor" mode. When these values are used, the characters lose in resolution but they are made of dots of three colors: the border color, the auxiliary color, and the character color. Normally, this effect is not useful with letters and numbers but it can be very effective with graphics.

Values from 16 through 255 repeat the action of the first 16 values, switching into and out of the multicolor mode every eighth number. You can see the effect with various borders and backgrounds by running the following program. Just hold the space bar down until you see something you like. Then copy the values from the left columns. The function of line 20 will be explained in the next section.

```
10 FOR Y = 0 TO 255
20 POKE 36879, Y
30 FOR X = 0 TO 15
40 POKE 646, X
50 GET A$
60 IF A$ = "" THEN 50
70 PRINT Y; X " A B C CMDR-+ SHFT-Z CMDR-I"
80 NEXT X, Y
```

Color With POKEs Into Color RAM

You know from your reading in Chapter 5 that there is a section of memory called Color RAM. This area of memory has PRINT positions on the screen, in the same manner as does the Screen RAM. The difference is that values POKEd into these addresses do not put a character on the screen but they set/change the color of any character PRINTed there.

Color RAM addresses for the 5K VIC 20 extend from 38400 to 38905 as shown on the Color RAM map in Appendix E. If you POKE a value (0–255) into one of those addresses, any character will assume the color determined by the value POKEd. No effect is seen if there is no character at that screen location. Later PRINTing at that location assumes the color dictated by the cursor, so you can affect only preexisting characters with this method.

This short program will enable you to experiment with using Color RAM:

```
10 PRINT " SHFT-CLR ";
20 FOR X = 1 to 176
30 PRINT " SHFT-Q ";
40 NEXT
50 FOR Y = 1 TO 8
60 INPUT "WHERE "; A
70 INPUT "VALUE "; B
80 POKE A + 38400 , B
90 NEXT
```

RUNning this program PRINTs eight rows of balls. Answer "Where?" with any number from 0 to 175 to select a ball at location 38400 plus the number. Answer the "Value?" question with numbers between 0 and 255 to see the selected ball change color.

You will find that the values function in exactly the same way that they would if POKEd into 646 (previously). In groups of eight, they alternate between normal colors and multicolors. The difference is that only one position is affected. Note once again that when a number corresponding to the background color is POKEd, the ball seems to vanish.

Color With POKEs to the "VIC"

Earlier we said that the VIC 20 probably got its name from the 6560 Video Interface Chip (VIC), an integrated circuit that controls the video signal produced by the machine. Now you will begin to learn something about the great power of that chip. We will limit our attention at this time to locations 36878 and 36879, which affect the color of the display.

Location 36879 is the primary color determiner in the VIC 20. Actually, the byte POKEd into 36879 determines three separate but related colors. In order to understand and use it fully, it is necessary to consider the individual bits within the eight-bit "word" of the byte; that is, you must convert decimal into binary numbers and vice versa.

Chapter 2 contains instructions for these conversions. We will just hit the high spots here to give you the idea and you can take it from there. If you prefer not to get down to the bit level at this time, skip down three paragraphs for empirical methods for using 36879.

Location 36879 (and all the others) will hold a byte that consists of 8 bits (ones and zeros). The first three bits — the Least Significant Bits (LSB) — set the frame color of the display. The fourth bit sets the background/foreground (0 is equivalent to REVERSE ON and a 1, REVERSE OFF). The last four bits — Most Significant Bits (MSB) — set the screen color.

Diagrammatically, it looks like this:

```
MSB < - - - - - - - - > LSB
screen      rev      frame
```

The color values are:

0 BLACK	4 PURPLE	8 ORANGE	12 LT. PURPLE
1 WHITE	5 GREEN	9 LT. ORANGE	13 LT. GREEN
2 RED	6 BLUE	10 PINK	14 LT. BLUE
3 CYAN	7 YELLOW	11 LT. CYAN	15 LT. YELLOW

Now let's look at a couple of examples. Suppose you wanted a green screen with yellow border and reverse off. The screen would be 5 (0101); the reverse, 1; and the frame, 7 (111). String the binary numbers together and you have 01011111, which is equal to 95 decimal. POKE 36879, 95 and you have it. You can turn on the reverse by changing the fourth bit to 0: 01010111, which is equal to 87. Any permissible color combination can be set in the same manner.

Rather than go through these number conversions, you can go to the Screen and Border Color Table in Appendix F. The numbers in this matrix will set the screen and border colors as indicated. Reverse will be OFF. Reverse can be turned on for any given combination by subtracting eight from the number in the matrix.

Then, too, you can run this little program to see what is available and choose what you want. It will show all the combinations.

```
10 FOR X = 0 TO 255
20 POKE 36879, X
30 PRINT " SHFT-CLR " SPC (185) X
40 GET A$
50 IF A$ = "" THEN 40
60 NEXT
```

Address 36878 is an interesting one in the color realm. The four MSBs can be used to set the multicolor mode to the desired color. Since there are 4 bits, the value can be from 0 to 15 with the color corresponding to the color code table. There are two caveats in the use of this address.

First, the four MSBs of 36878 do set the multicolor mode color but it is effective only on those screen locations that previously have been set up in that mode. Refer to the previous sections on how to POKE multicolor into Reserved RAM and Color RAM. The second caveat is that the four LSBs of this address are used to set the volume of the sound generators (see Chapter 11 for details).

You must be careful to leave these bits undisturbed if any of the sound generators are being used.

SUMMARY

This business of color on the VIC 20 can be quite confusing at first because there are so many colors, color combinations, and methods to achieve them. Perhaps the best way to start out is to experiment with them as suggested here in order to know just what is available. When you know that, you can always refer back here to find out how to create the effect you want.

CHAPTER 10

Notes: Data Management

Any program that you write will involve the management of data; i.e., the use of information. That data may be in numeric or string form and its manipulation may be simple or complex, but the processes are there, for the VIC 20's whole operation depends upon data. It is for this reason that a knowledge of data management is essential to the programmer.

There is yet another reason why data management is of great importance to you. When data is manipulated effectively in numeric and string variables, you avoid wasting space, both in the memory and on the tape or other storage medium. The saving of space on the tape, especially, means a considerable saving of time for you and the operator, as the data is sent to and from the tape.

The main functions for handling data are:

ASC	MID$
CHR$	RIGHT$
LEFT$	STR$
LEN	VAL

These functions are straightforward and they are explained adequately in your owner's manual. There is a bit of a quirk, however, in the use of MID$ so we will mention that before getting on to the Notes.

In a manner of speaking, the MID$ function can be used in two ways. The first and most common appears in this form:

```
B$ = MID$ (A$, 5, 3)
```

When this command/statement is executed, B$ becomes equal to the part of A$ that begins with the fifth character and is three characters long; i.e., a string consisting of the fifth, sixth, and seventh characters of A$.

The second way of using MID$ is often quite useful. The third term in the parentheses is omitted and, following the previous example, would look like this:

```
B$ = MID$ (A$, 5)
```

Now, B$ becomes equal to the fifth character and all following characters in A$. If A$ = "ABCDEFGHIJ", then B$ would equal "EFGHIJ" instead of "EFG" as it would previously.

In dealing with data management, you should be aware that a string variable can contain digits, in fact, it can be all numbers. (Numeric variables, however, cannot contain alpha characters.) In many instances, it is advantageous or even necessary to collect digits as strings of characters or, if they are already numeric, to change them to strings.

The reason for preferring strings is that they can be manipulated in many ways that cannot be applied to numerics. For instance, how would you go about getting the last four characters of numeric variable A = 4279643 ? It would take some mathematics. If A$ = "4279643", however, it is an easy matter to get thing like that, you would have to compute some new mathematics. If A$ = "4279643," however, it is an easy matter to get the last four digits with the LEFT$ (or MID$) function.

When numbers are collected as, or are converted to, strings in order to manipulate them, they may be used as strings if they are simply to be listed on a report or something of that nature. If they are to be used mathematically (added, subtracted, divided, etc.) they are easily converted to (or back to) numeric values with the VAL function.

ARRAYS OF RANDOM CHARACTERS

It is often necessary to build arrays of random characters. They are used in some types of programs and they are needed to test programs in their development stages.

Listing

```
10  DIM A$ (15)
.
.
.
170 FOR X = 1 TO 15
175 FOR Y = 1 TO 5
180 W = INT (RND (0) * 26) + 65
185 A$ (X) = A$ (X) + CHR$ (W)
190 NEXT Y
195 NEXT X
.
.
.
200 FOR X = 1 TO 15 : PRINT A$ (X), : NEXT
```

Fig. 10-1 shows a flowchart for generating random strings.

Analysis

10 dimensions the A$ array.
170 sets up a 15-count loop (number of strings).
175 sets up a 5-count loop (number of characters per string).
180 generates a random number between 65 and 90.
185 concatenates the character represented by the value of the random number to A$ (X).
190–195 continue the loops until conditions are met.
200 PRINTs the strings for demonstration purposes.

Use

This sequence of statements generates a series of 15 strings of 5 characters each. Such random strings can be used to test string handling operations in your programs. For example, it is advisable always to test sorting and recording subroutines before "buttoning up" a program. In case of errors, you can get tired of repeatedly entering test strings manually.

Of course, there are programs that directly make use of random strings. Among these are programs for advanced typewriting and for testing the operator's memory. An interesting appli-

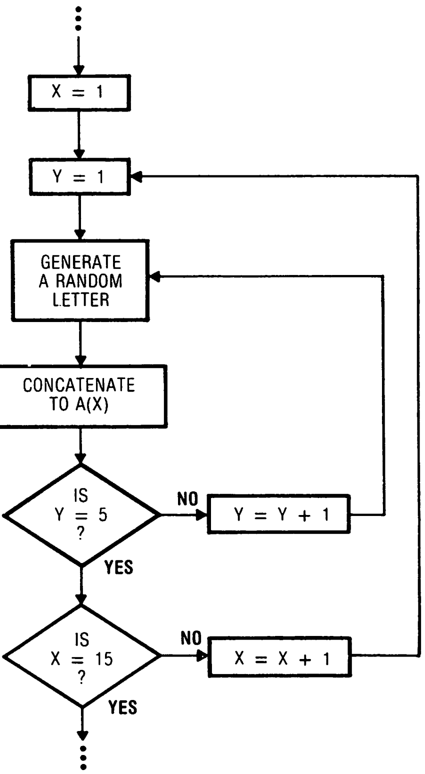


Fig. 10-1. Flowchart for generating random strings.

cation is the generation of "random code groups" for Morse code study by Scouts and ham radio operators.

Variations

1. The number of strings can be changed by changing the 15 in lines 170 and 200. The string length is determined by the 5 in line 175.

2. If strings of random numerics are needed, make these changes:

```
10  . . . , B (15)
180 W = INT (RND (0) * 10) +48
182 IF Y = 1 AND W = 48 THEN 180
192 B (X) = VAL (A$ (X) )
```

Note that Line 182 prevents the first digit from being a zero so that all variables will be of the same length.

3. The nature of the strings can be made to suit your needs. Use Appendix G to find the proper code ranges. This variation in the original Listing generates a mixture of letters and digits:

```
180 W = INT (RND (0) * 43) + 48
183 IF W > 57 AND W < 65 THEN 180
```

Line 180 generates numbers between 48 and 90 that include digits and letters. Line 183 rejects the numbers from 58 to 64 which represent symbols not wanted.

4. As illustrated by line 183, you can lock out any characters or groups of characters you wish.

EQUALIZING VARIABLE LENGTHS

In data management, there will be occasions when you want all the data to be of the same length. This is especially true in some instances of concatenating strings as pointed out later. Equalizing variable lengths is a simple matter of inserting one line in your program.

This line will make all affected variables seven characters long:

```
230 IF LEN (D$) < 7 THEN D$ = " " + D$ : GOTO 230 : REM -- one space
      between quotes
```

Here, if D$ is less than seven characters long, a space is added to the front of it and execution goes back to measure the length of the string again. If the length is not seven, another space is added and if it is, execution proceeds to the next line of the program.

Of course, the space(s) can be added to the end of D$ by reversing the positions of D$ and " " in line 230. Any character can be added instead of a space — just put it between the quotes in line 230.

CONCATENATING DATA

Concatenate — that is just a fancy word for "add." Well, not quite. If you add 3 and 5, you get 8. If you concatenate 3 and 5, you get 35. Some of the confusion arises because the same symbol (+) is used for both add and concatenate. Further, there is the difference between numeric and string data as shown in these statements:

```
80  A = 3 : B = 5
90  PRINT A + B
100 A$ = "3" : B$ = "5"
110 PRINT A$ + B$
```

Line 80 defines A and B as numeric variables, so line 90 adds the two numbers and displays 5. On the other hand, line 100 identifies two string variables that are concatenated to 35 in line 110. The process is somewhat clearer when dealing with letters or words. Apples and oranges cannot be added but they can be concatenated to "applesoranges."

Concatenation, then, can be thought of as tacking one piece of data to the end of another. Webster says that to concatenate is "to connect or link in a series or chain." The process is quite valuable when dealing with strings. You will find concatenation used in many of the Notes throughout the book.

Concatenation of items has a special advantage when handling strings containing related pieces of information. For example, consider a stock inventory. Instead of assigning each piece of information about an item to an individual variable, you could concatenate the information into something like this:

```
B$ (1) = "A13976123"
B$ (2) = "C17022056"
```

A LEFT$(B$(n),1) statement would always produce the department (A or C in this case). MID$(B$(n),2,5) would give the stock number (13976 or 17022) and RIGHT$(B$(n),3), the quantity (123 or 056). This approach would save a lot of memory and time.

Suppose, however, the corresponding items were of different lengths. Your variables would look like this:

```
B$ (1) = "A13976123"
B$ (2) = "AF627056"
```

Obviously, you could not separate it by the method indicated previously and get any useful information. You would have a classic case of GIGO — Garbage In, Garbage Out!

One way around this problem would be to fill in the shorter items to correspond in length to the longest item of the same type with the method shown in a previous Note. That would mean a lot of spaces or blanks that would waste memory though not as much as using individual variables.

Of course, there is another solution to the problem of different length items. That is to concatenate them but insert a "separator" symbol to indicate where one ends and the next begins. Consider the following example (it is numbered to use the random strings generated by the previous Note).

Listing

```
300 N = 0
310 FOR X = 1 TO 5
320 FOR Y = 1 TO 3
330 N = N + 1
340 D$ (X) = D$ (X) + A$ (N) + "/"
350 NEXT Y, X
370 FOR X = 1 TO 5 : PRINT D$ (X) : NEXT
```

Analysis

300 sets the counter variable to zero.
310 sets up a 5-count loop for the concatenated strings.
320 sets up a loop to put 3 small strings in each one.
330 increments the small-string counter.
340 concatenates the small strings with each followed by a separator.
350 completes the loops.
370 PRINTs the concatenated strings for demonstration.

Use

Though the statements are shown to use previously generated data, A$(N) could be data from any source. It could be read from data statements in the program or entered by the operator. In any case, they are concatenated and stored in this manner:

```
D$ (1) = "FISHING LURES/BOWMAN BEN/POP FISH/JUN 83/"
D$ (2) = "MODEL ROCKET CAMERA/WISE AD/..../..../"
D$ (n) = "..../....../........./......../"
```

This concatenation process is useful any time you have data of variable length to use or SAVE/LOAD. Especially, if tape cassettes are used, the time saving can be considerable.

The following Note will show you how to re-create the original pieces of data from the concatenated strings.

Variations

1. A slash (/) has been used as the separator here. You should choose a separator symbol that will not appear in your data or

the data will not be properly re-created later.

2. You should use loop specifications in lines 310 and 320 to match those of your items and the number of storage variables you wish to create.

3. The manual input of data to be concatenated requires a different treatment than that shown previously. The following sequence of lines will allow manual input.

Listing

```
10  DIM D$ (12)     : REM — the maximum number of expected
    storage variables
300 N=1
310 PRINT " SHFT-CLR ENTER @ TO END STOCK ITEM; * TO END
    LIST"
315 B$ = "/" : INPUT "DATA"; A$
320 E$ = RIGHT$ (A$, 1)
325 If E$ = "@" OR E$ = "*" THEN GOSUB 350
330 D$ (N) = D$ (N) + A$ + B$
335 IF E$ = "@" THEN N = N + 1 : GOTO 310
340 IF E$ = "*" THEN 380
345 GOTO 315
350 IF LEN (A$) = 1 THEN B$ = E$ : A$ = " " : RETURN
360 A$ = LEFT$ (A$, LEN (A$) - 1) : B$ = B$ + E$ : RETURN
380 FOR X = 1 TO N : PRINT D$ (X) : NEXT
```

Analysis

10 sets the dimensions of the storage array.
300 sets the counter to 1.
310 PRINTs the input directions to the user.
315 establishes the separator and asks for a response.
320 assigns the last character to E$.
325 if E$ is equal to the symbol for end-of-item or end-of-list, calls the subroutine.
330 concatenates response plus separator or end symbol.
335 at the end of the item, increments the counter and seeks another item.
340 transfers to demonstration at the end-of-list symbol.
345 transfers back for additional data on the item.
350–355 subroutine.
350 changes input and separator if the length is 1.
355 shortens the response and changes the separator.
380 demonstrates the storage variables.

Use

This routine is useful whenever data of undetermined number and length are to be entered manually. The operator is required to furnish indication of the end-of-item and end-of-list. All markers are automatically inserted at the appropriate places.

The storage variables appear quite different from those of the previous example. Now they are in this form:

```
D$ (1) = "WINTER BIRDS/BOWMAN BRIAN/POP AV/JUL 83/@"
D$ (2) = "WORDPROCESSING/KENNEDY CHARL/......../@"
D$ (3) = "ADV..../.../...../...../...../*"
```

Re-creation of the original form is shown below.

PARSING DATA

Parse — well, that is another fancy word used in data management and it means breaking something up into its parts. When data is concatenated to improve the efficiency of handling it in variables and/or data statements, sooner or later it must be broken up into its original pieces. That is exactly what we do when we parse data. The proper parsing method to use will depend on the nature of the concatenated data.

Parsing Method 1

When items of a concatenated string were the same size or were made to be the same size, their separation is quite straightforward. All you have to do is to use the LEFT$, MID$, and RIGHT$ functions incorporated in BASIC. These functions are useful in this case simply because the various data items are of a known size. LEFT$(B$(n),10) picks off the 10 left-most characters of B$(n) each time it is applied.

Here is an example that uses strings in DATA statements, though they could be from any source:

```
200 FOR X = 1 TO 20
210 READ A$
230 PRINT LEFT$ (A$, 12)
240 PRINT MID$ (A$, 13, 14)
250 PRINT RIGHT$ (A$, 5)
260 NEXT
.
.
.
900 DATABROADWELL CMNORTH CAROLINA12345
```

```
910 DATAWILLIAMS AH NEW MEXICO 54321
920 DATABELL BW IOWA 23456
 .
 .
 .
```

This program segment will always PRINT the name, state, and zip of the DATA list. This is true because shorter items have been "padded" to make their lengths predictable. It would not work if the spaces were omitted from the DATA.

Notice that all those spaces in this type of DATA string use up memory. That is why it should be employed only when the items are of fairly consistent length. Other types of concatenating/parsing should be used when item length varies (as it usually does).

Parsing Method 2

This type of data separation makes use of the fact that items have been concatenated without any extra spaces for padding to a standard length. Instead, the variables contain separator symbols to prevent items from blending into each other.

In this example, a single list is to be parsed into individual items that are separated by a slash (/). The variables were concatenated by a method shown previously. This time, the data is in program statements for the sake of clarity.

Listing

```
10  DIM F$ (25)    : REM — to accommodate the new list
90  D$ (1) = "WORREL BJ/BOX 3421/COLUMBIA SC/87654/"
95  D$ (2) = "HUXTABLE SB/3306 THIRD AVE/NEW ROCHELLE
    NJ/23456/"
 .
 .
 .
410 N = 1
420 FOR X = 1 TO 10 : REM — the number of variables to be parsed
430 FOR Y = 1 TO LEN (D$ (X) )
440 B$ = MID$ (D$ (X), Y, 1)
450 IF B$ = "/" THEN N = N + 1 : GOTO 470
460 F$ (N) = F$ (N) + B$
```

```
470 NEXT Y
480 NEXT X
520 FOR X = 1 TO N : PRINT F$(N), : NEXT
.
.
.
```

These statements separate the original space-saving variables into individual items. It can be used because the number of items is known.

Parsing Method 3

This method of parsing variables is somewhat more complex than the preceding ones. It is designed for use with manually input information or other data that is of unknown length and quantity. It requires that the parts be separated by a slash (/), the items separated by an @ symbol, and the end of the list identified with an asterisk (*). Again, the variables are shown in the program for clarity.

Listing

```
10  DIM F$ (15)
90  D$ (1) = "QWERTY/123456/A/GOOD/TRY/@"
92  D$ (2) = "NOW/IS/THE/TIME/@"
94  D$ (3) = "THE/QUICK/BROWN/FOX/JUMPED/OVER/*"
.
.
.
410 N = 1
420 FOR X = 1 TO 50    : REM — more than the number to be
    parsed
430 FOR Y = 1 TO LEN (D$ (X) )
440 B$ = MID$ (D$ (X) ,Y, 1)
450 IF B$ = "*" THEN N = N - 1 : GOTO 520
460 IF B$ = "@" THEN 500
470 IF B$ = "/" THEN N = N + 1 : GOTO 490
480 F$ (N) = F$ (N) + B$
490 NEXT Y
500 NEXT X
520 FOR X = 1 TO N : PRINT F$ (N) : NEXT
.
.
.
```

Analysis

410 sets the new-variable counter to 1.
420 sets up a loop large enough to handle the concatenated data.
430 sets up a loop to the length of the current variable.
440 extracts sequential characters from the variable.
450 if the character is an asterisk, decrements the counter and leaves the parsing routine.
460 if the character is @, changes to a new input variable.
470 if the character is a slash, increments the counter and starts a new variable.
480 concatenates the new variable.
490–500 continues the loops.
520 demonstrates the parsed variables.

Use

This parsing program segment is labeled to match the concatenation of manually entered data in the previous Note. It can be used with data of any origin and almost of any form if matching separators and symbols are used. With this parsing method it is not necessary to know much of anything about the stored data — just enough to dimension the arrays so that they are large enough to store it.

FINDING BURIED DATA

DATA statements are read serially and sequentially. The first READ gets the first datum, the second READ gets the second datum, and so on. A **RESTORE** statement resets some pointers so that the following READ statement will cause the first datum to be read again. It is not always convenient to access the data sequentially. Sometimes you will want (or need) to begin with the sixteenth or forty-second datum. Here are two methods that can be used to access these values in the order you choose.

Access Method 1

In this method, you read and discard all items up to the one you want or the one with which you wish to begin READing. No special preparation of the DATA statements is required.

Listing

```
.
.
.
50 FOR X = 1 TO 16
55 READ A$
60 NEXT X
65 PRINT A$
.
.
.
120 RESTORE
125 FOR X = 1 TO 41
130 READ A$
135 NEXT X
140 FOR X = 1 TO 9
145 READ A$ : PRINT A$
150 NEXT X
.
.
.
```

Fig. 10-2 shows a flowchart for finding buried data, Method 1.

Analysis

50–60 READ the first 16 data values. Note that the first 15 are "discarded" because each one is replaced in variable A$ by the next one.
65 PRINTs the sixteenth datum (the final one read).
120 resets the data pointer to the first datum.
125–135 READ the first 41 data values.
140–150 READ and PRINT the next 9 values.

Access Method 2

This method is somewhat like the first except that you need not know the number of items to discard to get to the one you want. This is possible because the DATA statements have been specially marked with coded locations.

Listing

```
.
.
.
30 READ A$
```

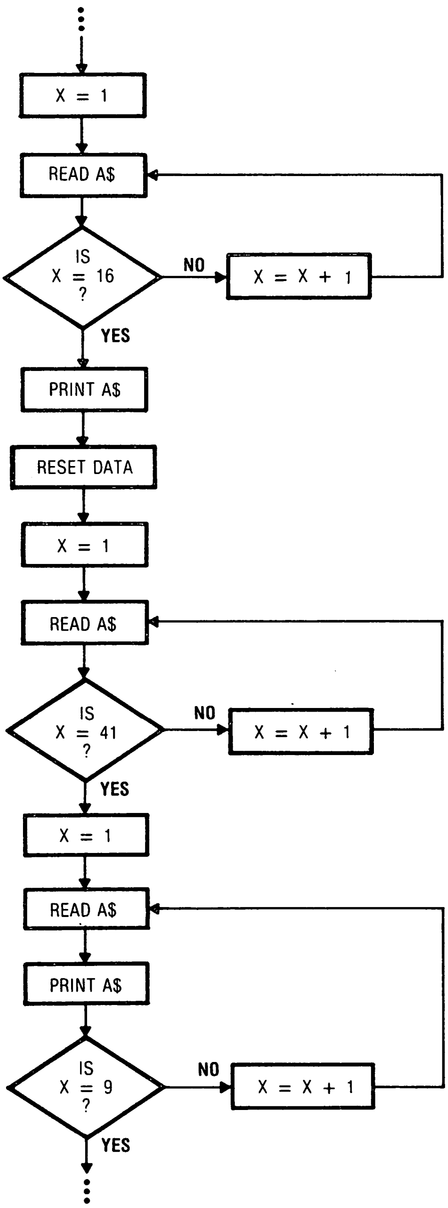


Fig. 10-2. Flowchart for finding buried data, Method 1.

```
35 IF A$ <> "C" THEN 30
40 FOR X = 1 TO 6
45 READ A$ : PRINT A$
50 NEXT X
.
.
.
90 RESTORE
95 READ A$
100 IF A$ <> "B" THEN 95
105 FOR X = 1 TO 3
110 READ A$ : B$ (X) = A$
115 NEXT X
.
.
.
300 DATA A,ONE,TWO,THREE,FOUR,FIVE
305 DATA B,SIX,SEVEN,EIGHT,NINE,TEN
310 DATA C,TWENTY,THIRTY,FORTY,FIFTY,SIXTY,SEVENTY
.
.
.
```

Fig. 10-3 shows a flowchart for finding buried data, Method 2.

Analysis

30–35 keep READing and discarding data until it READs C.
40–50 READ and PRINT the next 6 datum entries.
90 resets the DATA pointer.
95–100 READ and discard until it finds B.
105–115 assign the next 3 data items to array B$ (X).

Use

This method is preferred over the preceding one because you don't have to count and recount the DATA to get the number of discards correct. To use it, however, you must properly mark the places you will want to find later. Obviously, the two methods can be used on the same DATA statements.

It might appear that neither of these methods of finding buried DATA is necessary — just rearrange the order of the DATA statements and you won't need them. You are correct, of course, for these simple examples. You will discover, however, that as your programs become more complex these methods are quite useful and are sometimes necessary to prevent your having to repeatedly re-type lines and lines of DATA statements in a program.

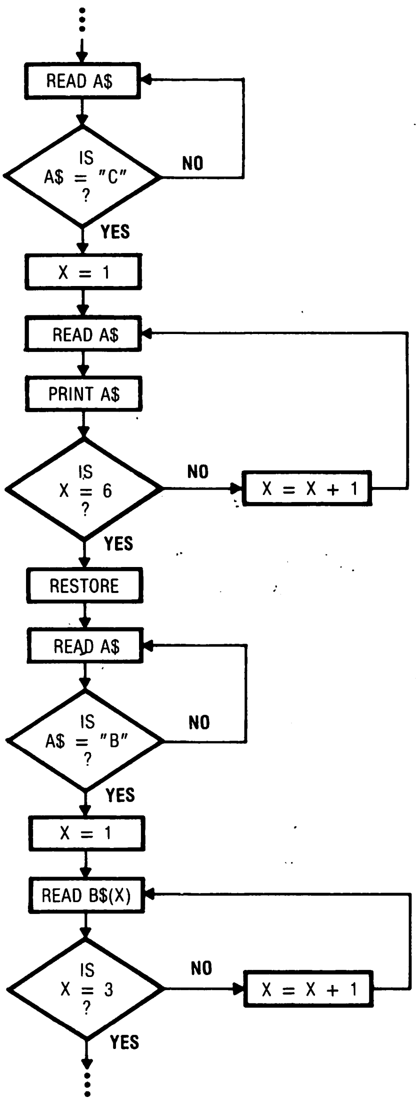


Fig. 10-3. Flowchart for finding buried data, Method 2.

Here is an example of a program in which these methods can be used to reduce your work. It is a program in which you must keep items of DATA in the same relative order even when items are added and deleted. Suppose you want to display or print four columns of groups of words from DATA statements.

You could read the first word in each column (four words) and print them, read four and print those, and so on to the end. The result would be:

```
w1      x1      y1      z1
w2      x2      y2      z2
w3      x3      y3      z3
w4      x4      y4      z4
```

and it would use a DATA statement that would look like this:

```
500 DATAw1,x1,y1,z1,w2,x2,y2,z2,w3,x3,y3,z3 . . .
```

All would be fine until you wanted to insert a new "w" word between w1 and w2. To keep the words in order and in the same columns, it would be necessary to re-type all of the w words following w1 — some fun!

On the other hand, using Method 2 and putting the vertical groups in separate DATA statements makes the task simple. This example uses just two columns for simplicity:

```
35 N = 1
40 READ E$
45 IF E$ <> "A" THEN 40
50 FOR X = 1 TO N
55 READ A$
60 IF A$ = "999" THEN 120
65 NEXT X
70 READ E$
75 IF E$ <> "B" THEN 70
80 FOR X = 1 TO N
85 READ B$
90 NEXT X
95 N = N + 1
100 RESTORE
105 PRINT A$,B$
110 GOTO 40
120 . . . (continue program) . . .
.
.
.
500 DATA A,w1,w2,w3,w4,w5, . . .
505 DATA w23,w24,w25, . . .,999
510 DATA B,x1,x2,x3,x4,x5,x6, . . .
515 DATA x23,x24,x25,x26, . . .
```

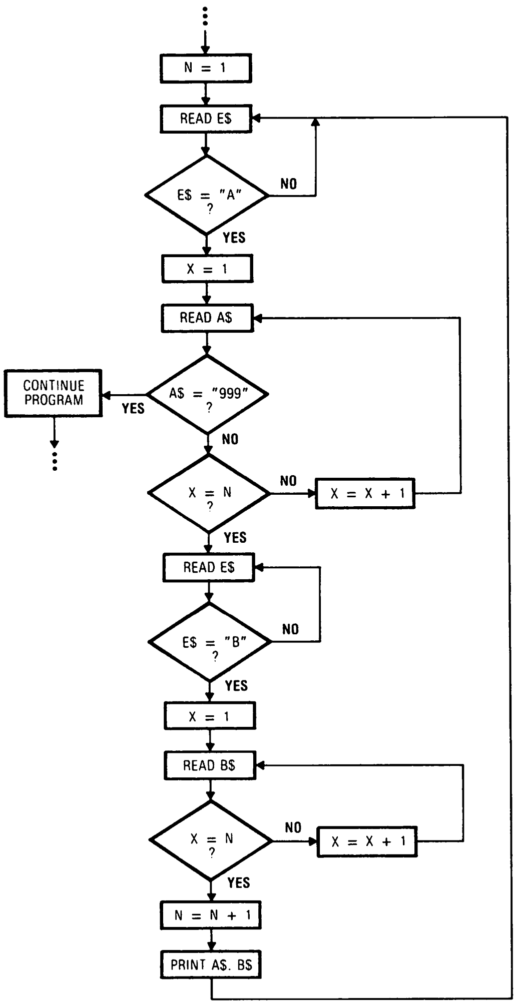


Fig. 10-4. Flowchart for buried data, special method.

The flowchart in Fig. 10-4 will help you follow the action in this routine. Note how much easier it is to add, delete, and/or change items in these DATA statements. You might also take a good look at the way in which variable N is used. With each pass through the loop, data is read until it gets to the first value that has not been printed.

BUBBLE SORT

It is often desirable, at least, to put lists of various types into some kind of logical order. Then, too, some types of lists must be ordered and reordered repeatedly when they are used for several purposes.

A mailing list, for example, may be sorted (ordered) by ZIP codes for the benefit of postal regulations and then re-sorted alphabetically for membership checking. Every program used for handling lists should have a sorting subroutine built into it. Of course, such lists must be sorted again every time a new name is added or an old name is deleted.

Listing

(Assumes the list is in array A$(N) with N being the largest subscript.)

```
.
.
.
500 FOR J = 1 TO N -1
510 FOR K = J +1 TO N
520 IF A$ (J) < = A$ (K) THEN 560
530 L$ = A$ (J)
540 A$ (J) = A$ (K)
550 A$ (K) = L$
560 NEXT K
570 NEXT J
580 RETURN
.
.
.
```

Fig. 10-5 shows a flowchart for the bubble sort routine.

Analysis

500 sets a loop to count to 1 less than the array size.
510 sets a loop to count one more than the first loop throughout the range.
520 if the upper item is equal to or smaller than the lower item, skips to the next comparison.
530–550 swap the upper for the lower item if the latter is smaller.
560 checks the K-loop for completion.
570 checks the J-loop for completion.
580 transfers back to the main program.

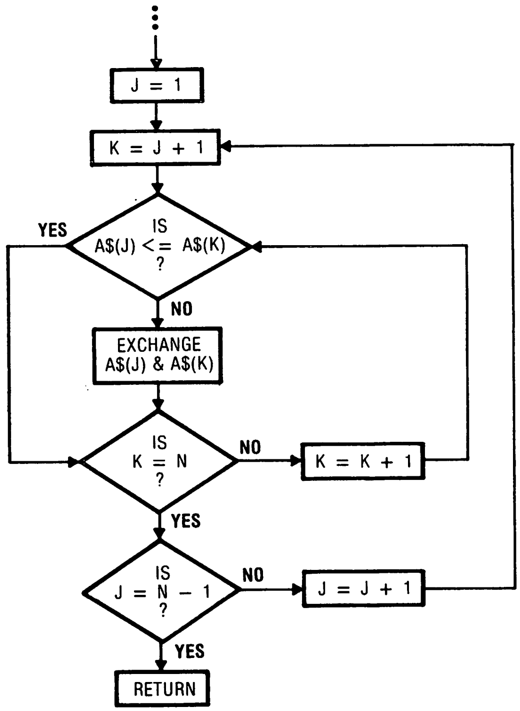


Fig. 10-5. Flowchart for the bubble sort routine.

Use

This bubble sort subroutine orders the list (array) from low to high. A numeric list will begin with the lowest number and progress to the highest. An alpha list will begin with the A's. A mixed alphanumeric list will begin with the numbers (in order) followed by the ordered alphabet.

It is advisable for you to check the functioning of a sorting subroutine whenever you write one into a program. To avoid the tedium of entering test lists manually, see the earlier note on generating random strings.

Variations

This subroutine can be made to reverse the order of the sorted list. For high-to-low sorting, simply change the less-than symbol (<) in line 520 to a greater-than symbol (>).

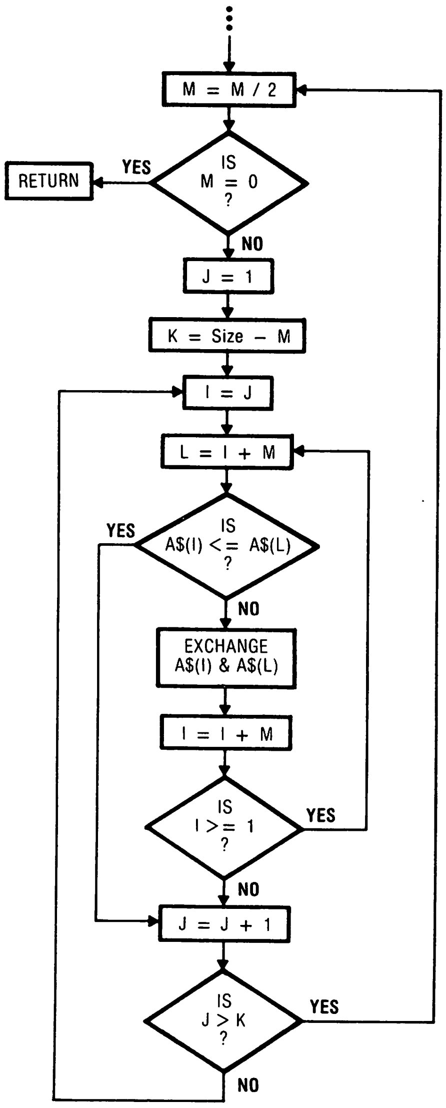


Fig. 10-6. Flowchart for the Shell-Metzner sort routine.

SHELL-METZNER SORT

The Shell-Metzner sorting technique is not used as frequently as the bubble sort, perhaps because it is somewhat more complex. However, it does the job faster than the bubble sort. The time difference is slight for short lists but it becomes quite significant with lists that contain hundreds of entries.

Listing

(Assumes a list in array A$(N) with N being the largest subscript.)

```
.
.
.

500 M = N
510 M = INT (M / 2)
520 IF M = 0 THEN RETURN
530 J = 1
540 K = N - M
550 I = J
560 L = I + M
570 IF A$ (I) < = A$ (L) THEN 630
580 H$ = A$ (I)
590 A$ (I) = A$ (L)
600 A$ (L) = H$
610 I = I - M
620 IF I > = 1 THEN 560
630 J = J + 1
640 IF J > K THEN 510
650 GOTO 550
```

Fig. 10-6 shows a flowchart for the Shell-Metzner sort routine.

Use

This Shell-Metzner sort is much more efficient than the bubble sort. The amount of time that it saves increases at an exponential rate as the number of sorted items increases. For example, as a rough approximation, it requires only one-half of the bubble sort time for 10 to 15 items, one-fifth for 100 items, one-twentieth for 500, and so on.

The efficiency of a sorting routine is determined by the number of times each item is compared with the others and the number of times their positions in the array must be switched. The bubble sort requires that each item be compared with every other item in the list. The S-M sort requires fewer comparisons and, with an identical list of items, fewer switches.

It does take a little longer to type the S-M sort into your program. With a hundred or more items, however, you will more than make up that time after a few runs of the program.

CHAPTER 11

Sound Generation

Your VIC 20 is such a versatile machine that all of the variations of its great capabilities have yet to be explored. In preceding chapters, we have been able to do little more than skim the surface of color, for example. It is unfortunate but true that try as we might, we will be able to do even less on the subject of sound here and in the next chapter. That gives you an indication of the potential of VIC 20 sound.

Many computers can produce only a limited type and number of sounds without add-on devices and intricate programming (usually in machine language). On the other hand, built right into the VIC 20 are three tone generators and a noise generator, all of which can be controlled with statements in plain ole BASIC. It is possible, even, to produce the effect of a fourth (and fifth?) tone generator!

THE CONTROLS

Well, we can't do it all but let's begin by talking about the basic controls we will use. You can, of course, turn the sounds on and off, adjust the volume, and set the tone (frequency) of the sound generators. Each of these functions can be controlled independently.

Volume

The VIC 20, like any sound generator, has a volume control with an ON/OFF switch. Search as you might, however, you will find no familiar knob to twist. The volume is adjusted with POKEs to address 36878. The values are limited to a range of 0 to 15 which have this effect:

0 = off
1 = lowest volume
•
•
•
15 = highest volume

Let's hear how that works. RUN the following program:

```
10 POKE 36875, 222
20 FOR X = 1 TO 15
30 POKE 36878, X
40 FOR Y = 1 TO 300 : NEXT Y
50 NEXT X
60 POKE 36878, 0
70 POKE 36875, 0
```

For now, ignore lines 10 and 70 (actually, they control the tone, but we will see that later). Lines 20 and 50 set up a FOR/NEXT loop to increment the value of X from 1 to 15 (the value for the volume control). Line 30 POKEs the value of X into 36878, the address of the volume control. Line 40 is a simple delay loop to "stretch" the duration of each step and line 60 turns the volume OFF after the main loop concludes.

When this program is RUN, you will hear a tone that increases in volume in step with the value of X. Experiment with the program a bit to become familiar with the volume control. For example, if you insert

```
25 PRINT X
```

you can see the values being POKEd into the volume location. Change line 20 to read FOR X = 15 TO 0 STEP −1 in order to decrease the volume from a loud beginning. (If this is done, line 60 is not needed . . . right?) The "300" in line 40 can be changed to increase or decrease the speed of volume change.

Actually, unless the volume is a specific part of the sound effect you are creating, the general practice is to use just two volume commands:

```
POKE 36878, 0 (= OFF)
POKE 36878, 15 (= ON)
```

Then, the volume control on your monitor/tv is used to adjust the loudness as desired. In order to simplify our discussion, we will use the "volume" control in this way.

Be aware that this control only regulates the amount of sound signal getting to your monitor/tv. It does not turn ON or OFF any of the sound generators.

Tone

As previously stated, your machine has three tone generators. They are operated exactly alike and may be used together or individually because each has a different address. These generators do cover a different range of frequencies (tones) that overlap.

The three memory addresses used for control are 36874, 36875, and 36876. The first (−74) produces the lowest tones and the last (−76) produces the highest. Each generator spans a range of three octaves and has a two octave overlap with the adjacent generator(s) in this manner:

```
(HIGHEST) 36876:                 OCT 3 OCT 4 OCT 5
          36875:           OCT 2 OCT 3 OCT 4
 (LOWEST) 36874: OCT 1 OCT 2 OCT 3
```

As indicated, all three tone generators are controlled exactly alike, with POKEs into the desired memory locations: 36874, 36875, or 36876. We will use 36874 in our examples but keep in mind that the other two addresses (generators) function the same way.

The values POKEd may range from 0 to 255. All values from 0 to 127 turn the generator *off* so the convention is to use the value of 0 to do so and ignore the others in that range. We will use only the 0 in our examples.

We are left, then, with values 128 through 255 which turn the generator *on* and set the tone. The higher the number you POKE, the higher the frequency will be. All of the numbers in this range will produce a tone in each of the generators but a given value will not produce the same tone in each one. You should not be surprised to find that values in the 250s in 36876 are inaudible — exactly where the tone cuts off will depend on the high frequency capability of your monitor/tv sound system and of your ears!

In order to get a feel for the ranges of the tone generators, key in this program:

```
10 POKE 36878, 15
20 X = 128
30 POKE 36874, X
40 PRINT X;
50 FOR Y = 1 TO 200 : NEXT
60 X = X + 1
70 IF X < 256 THEN 30
80 POKE 36878, 0
90 POKE 36874, 0
```

Line 10 turns on the volume. Note that both the volume and the generator must be *on* or no sound will be heard. Line 20 sets the beginning value at 128, which is POKEd into the lowest sound generator. In line 40, the value of X is PRINTed for your reference. Line 50 is a simple delay that keeps the tone on while it counts to 200. Lines 60 and 70 increase the value of X by one and, if it is less than 256, transfer the execution back to line 30.

When 256 is reached, lines 80 and 90 turn the volume and the generator *off.* Take note of the fact that either of these lines may be omitted with no apparent effect. If the volume (36878) only is turned *off,* the generator continues to run even though the sound cannot reach the monitor/tv. If the generator (36874) only is turned *off,* there is no sound to reach the speaker through the open volume control. For this reason, it is good practice to close down both before leaving a program or subroutine. Of course, if both lines are omitted, the last tone continues after the program ends.

You will find it interesting to change the tone generator (line 30) in order to hear the entire range of which the VIC 20 is capable. Changing the 200 in line 50 will change the duration of the tone, of course.

Noise

You may believe that there is quite enough noise in the world. If so, be prepared for a surprise because you will find that the noise you are about to add is constructive rather than destructive.

The fourth generator in your VIC 20 makes a special kind of sound. It is called "white noise" and sounds something like your tv set when you switch to an unused channel. The big difference here is that you have control over the noise, including its pitch. As you will see, this is very valuable in creating certain kinds of sounds.

The noise generator is controlled through address 36877. The 0 through 255 values POKEd at that location affect the *on/off* and pitch exactly as they do in the tone generators. You can experiment with the noise generator by using its address (36877) in the preceding program.

In practice, the noise generator is used most often to create sound effects — explosions, phaser guns, wave action, and the like. Such effects may or may not require the simultaneous operation of a tone generator but they usually do require manipulation of the volume control. This fundamental program is modified in many ways to create a wide variety of sounds:

```
10 POKE 36877, 235
20 FOR X = 15 TO 0 STEP -1
30 POKE 36878, X
40 FOR Y = 1 TO 70 : NEXT Y
50 NEXT X
60 POKE 36877, 0
```

Experiment with this program by changing the frequency of the noise (line 10), the loop parameters (line 20), and the duration (line 40). Try adding one or more tone generators as in these lines:

```
15 POKE 36875, 135
65 POKE 36875, 0
```

SUMMARY

The following chapter contains a number of examples of sound creation with the VIC 20. All of them are based on the information given previously. They can be understood and, thus, modified for your needs by referring back here as necessary.

CHAPTER 12

Notes: Combining Graphics, Color, and Sound

Graphics, color, and sound have been discussed and used in Chapters 4, 9, and 11. In fact, they have been used throughout the book because it is a rare program, indeed, which does not incorporate one or more of these features. The "trick," if there is one, is in combining them within a program.

You can mix graphics, color, and sound quite successfully if you remember one principle. Our old human minds, in spite of their apparent speed now and then, are really slow in many respects when compared with the VIC 20. As an example, a word or design that the VIC 20 flashes on and off the display will go right over our heads unless we build in a delay to give us time to absorb it.

Consider that an animated graphic would streak across the screen much too fast unless we insert a bit of a delay. If a sound is not lengthened, it just seems like a "pop" to us. A quick flash of color may as well be colorless for all practical purposes. Because we must slow down the VIC 20, as it were, there is plenty of time to do other things. There is no point in having the machine sit there and count like a dunce when it could be making a sound effect, changing a color, and/or moving a graphic!

Correlative to the principle of our slowness is the fact it often seems to us that the VIC 20 can do several things at the same time. Of course, it cannot actually do two things at once — it just appears that way. In any case, the important point is that you can usually stick another command or two here and there in an existing routine, and not discern any difference in the routine's execution. Of course, those "extra" commands can be for color, sound, graphics or whatever.

As indicated earlier, there is no trick to manipulating these three functions together. The basic technique is to "interleave" the commands for the functions we want. The only timing problems this process may cause is that some of the delays may have to be shortened a bit.

The recommended method of interleaving these functions is to build just one and then stick the others in where they belong. For example, first build an animated graphic routine, then go back and interleave color commands, and finally, go back and add the sound commands. The order in which the ultimate routine is built is not important — just do one thing at a time to reduce the chance of confusion and the possibility of hours spent on debugging.

The technique is not quite as simple as it may sound. If the routine has any complexity, you will have to do some trial and error experimentation — juggling — to get the effects just as you want them. That is only to be expected. When did you write the last program that ran perfectly the first time? — probably not since you wrote "10 PRINT 3 + 5 :: 20 END."

In building the types of routines we have been discussing, you will be doing much RUNning, LISTing repeatedly. The task will be much quicker if you use the little "Builder" utility program given in the section entitled How To Use This Book.

Beyond what has been said, there is not much to add in terms of "how-to-do-it." In the following Notes, we will just present a few examples and analyze each briefly. The potential number of variations is all but limitless. To avoid undue complications, the form of the Notes is changed a bit in order to simulate the way you would build the given routine — graphics, then color, then sound. The fact is that these routines were built exactly that way.

MUSICAL INTERLUDE

This first routine includes no complex actions in order to better illustrate the interleaving process. Moreover, it provides examples of several interesting techniques — storing changes in DATA statements, parsing two concatenated items, and cursor positioning. In this case, first we will generate the sound and then add a bit of graphics.

Program Listing (Sound)

```
5    POKE 36879, 8
20   G = 36875 : V = 36878
40   DATA2254,2254,2354,2354,2374,2374,2358
42   DATA2324,2324,2314,2314,2284,2284,2258
44   DATA2354,2354,2324,2324,2314,2314,2288
46   DATA2354,2354,2324,2324,2314,2314,2288
48   DATA2254,2254,2354,2354,2374,2374,2358
50   DATA2324,2324,2314,2314,2284,2284,2258
.
.
.
100  POKE V, 15
130  FOR X = 1 TO 42
140  READF$
160  POKE G, VAL (LEFT$ (F$, 3) )
170  FOR Y = 1 TO 100 * VAL (RIGHT$ (F$, 1) ) : NEXT
180  POKE G, 0
190  NEXT X
200  POKE V, 0
```

Analysis

5 sets the screen color (black).
20 sets variables to the tone generator and the volume control addresses.
40–50 contain the sequential concatenated frequency and duration information for the tune (the first three are frequency and the last one is the duration).
100 turns the volume control on.
130 sets a 42-count loop for the notes.
140 reads a (string) datum with each pass through the loop.
160 determines the value of the three left-most characters and POKEs it into the tone generator.
170 determines the value of the right-most character and counts the correct delay (duration of the note).
180 turns the tone off.
190 completes the note-count loop.
200 turns the volume off.

Notice that the duration of each tone is stored with its frequency. This makes for shorter data statements and it is much less confusing to write in than are separate items for each. Having to parse the data before use does not cause any discernible delay. Notice, too, that the DATA is read as strings in order to make the parsing easier. Of course, it has to be converted to numerics (with the VAL function) before it can be used in the POKEs.

Now, let's make the routine a bit more interesting by adding the words in a manner similar to that used in the old movie "sing-along."

Program Listing (Words Addition)

```
10  DIM A$ (42)
30  DATATWINK,TWINKLE,LIT,LITTLE,STAR,HOW,I,WON,WONDER,
    WHAT,YOU,ARE
32   DATAUP,A,ABOVE,THE,WORLD,SO,HIGH,LIKE,A,DIA,DIAMOND,
    IN,THE,SKY
34  DATATWINK,TWINKLE,LIT,LITTLE,STAR,HOW,I,WON,WONDER,
    WHAT,YOU,ARE
110 FOR X = 1 TO 42 : READ A$ (X) : NEXT
120 PRINT " CTRL-8"
150 PRINT " SHFT-CLR" SPC(95) A$ (X)
210 PRINT " SHFT-CLR"
```

Analysis

10 dimensions variable A$ to hold the words.
30–34 contain the words in DATA statements.
110 READs the words sequentially into array A$(X).
120 sets the PRINT color to yellow.
150 clears the screen, positions the cursor, and PRINTs the word corresponding to the note to be sounded.
210 clears the display after the "performance."

GALLOPING HORSE

Have you ever noticed in western movies how the horses make a different sound when they cross a bridge or a stretch of rock? Now, you can do the same thing. First, we will sketch a little horse moving across the screen.

Program Listing (Graphics)

```
290 POKE 36879, 8 : SC = 256 * PEEK (648)
310 PRINT CHR$ (147);
330 FOR Y = 0 TO 21
340 POKE SC + Y + A - 1, 32 : POKE SC + Y + A, 94
400 FOR X = 1 TO 100 : NEXT
430 NEXT Y
440 A = A + 22 : IF A > 500 THEN 460
450 GOTO 330
460 PRINT CHR$ (147)
```

Analysis

290 sets the screen color and sets variable SC equal to the first address of the Screen RAM (whatever its location).
310 clears the screen and homes the cursor.
330 sets up a 22-count loop.
340 POKEs a blank and a (horse) character which move to the right with each pass of the loop.
400 delays for a count of 100.
430 continues the loop until conditions are satisfied.
440 adds 22 to variable A (putting the horse down one line) and transfers to 460 when the horse gets to the last screen line.
450 transfers back to 330 to move the horse across the next screen line.
460 clears the screen and homes the cursor.

Well, that is a pretty crude horse — or is it a series of horses crossing successive screen lines? In any case, the animation is good enough to illustrate how sound can be mixed with graphics.

Program Listing (Sound Addition)

```
300 S2 = 36875 : V = 36878
320 POKE V, 15 : F = 240
350 FOR X = 1 TO 3
360 POKE S2, F
370 POKE S2, 0
380 FOR Z = 1 TO 50 : NEXT
390 NEXT X
410 F = 240
420 IF PEEK (197) <> 64 THEN F = 248
470 POKE V, 0
```

Analysis

300 sets variables to the sound and volume addresses.
320 turns the volume on and sets variable F equal to 240.

350 sets up a 3-count loop.
360 sets the sound generator to the value in F.
370 turns the sound generator off.
380 delays for a count of 50.
390 completes the 3-count loop.
410 sets F equal to 240 (in case it has been changed).
420 changes the value in F if any key is pressed.
470 turns off the volume control.

Now, the horse gallops along and you can hear the clomp of his hoofs. If you press a key, the sound will change as though the horse were crossing a bridge.

This program was put together just as indicated previously: the horse was programmed and, then, the sound statements were interleaved. In writing a program mixing sound, color, and/or graphics, if things get too complex, you can develop any of the parts independently and put them together after you have the result you want. Even so, you still will have to do some fine-tuning to make things mesh properly.

JUMPING JACK

In this routine, we begin with a simple animation of a fellow doing a jumping jack exercise. Then we add color and, last, sound. If you key in this little program in the parts as indicated, you will see how it is improved by each addition and how the inserted lines interact.

Program Listing (Graphics)

```
5 POKE 36879, 8 : POKE 646, 3
10 SM = 36881
.
.
.
100 PRINT " SHFT-CLR D CRSR(5 times)" : D = 1
110 FOR X = 1 TO 31
160 IF X / 2 = INT (X / 2) THEN PRINT SPC(10) " SHFT-J
    CMDR-I SHFT-K" : GOTO 180
170 PRINT  SPC(10)  " SHFT-U  CTRL-9  CMDR-Y  CTRL-0
    SHFT-I"
180 PRINT SPC(11) " CTRL-9 SPACE CTRL-0"
190 IF X / 2 = INT (X / 2) THEN PRINT SPC(10) " SHFT-N SPACE
    SHFT-M  SHFT-U  CRSR  SHFT-U  CRSR  SHFT-U
    CRSR" : GOTO 220
```

```
200 PRINT  SPC(10) " CMDR-N SPACE CMDR-H SHFT-U
     CRSR SHFT-U CRSR SHFT-U CRSR"
220 FOR Y = 1 TO 150 * D : NEXT
240 POKE SM, 22 : FOR Y = 1 TO 50 : NEXT : POKE SM, 24
250 NEXT X
260 PRINT SPC(10) " SPACE (3 times)" : PRINT SPC(11) "CMDR-@
     CMDR-P CMDR-I SHFT-I" : PRINT SPC(10) " SPACE
     CMDR-@ CTRL-9 SPACE SPACE CTRL-0 SHFT-I"
270 FOR Y = 1 TO 3000 : NEXT
.
.
.
```

Analysis

5 sets screen and printing colors.
10 sets the variable to the address of screen "jump" location.
100 positions the cursor and sets the time delay variable.
110 sets a 31-count loop for the number of jumps.
160 on every even-numbered jump, PRINTs the arms up.
170 on other (odd-numbered) jumps, PRINTs the arms down.
180 PRINTs the body.
190 on even jumps, PRINTs the legs apart and moves the cursor up three lines (ready to PRINT the next time).
200 on odd jumps, PRINTs the legs together and moves the cursor up three lines.
220 counts a delay.
240 moves the entire screen up and then down to simulate jumping action.
250 completes the jump-counting loop.
260 PRINTs the figure in a prone position.
270 counts a delay.

That is quite a little fellow, but he creates little interest until he stops to rest. Or did he pass out? Well, let's add some color and see if that helps.

Program Listing (Color Addition)

```
130 IF X > 20 THEN POKE 646, 4 : D = 2
150 IF X > 25 THEN POKE 646, 5 : D = 4
```

Analysis

130 when the jump-count exceeds 20, changes the color to purple and slows the jump rate.
150 when the jump-count exceeds 25, changes the color to green and further slows the jump rate.

That is an improvement. We know the guy apparently is not feeling too well. Notice how the color statements were stuck right in there with no discernible difference in the execution of the routine. Obviously, slowing the rate also added to the effect.

For the final touch, let's turn this routine from a silent movie into one with sound. In practice, sound could have been added before the color.

Program Listing (Sound Addition)

```
15 S2 = 36875 : V = 36878
105 F = 240 : POKE V, 15
120 IF X > 15 THEN F = 255 - X
140 IF X > 22 THEN F = 260 - X * 3
210 POKE S2, F
230 POKE S2, 0
280 POKE V, 0
```

Analysis

15 sets variables to the addresses of a tone generator and the volume control.
105 sets the initial frequency (tone) and turns the volume on.
120 when the jump-count exceeds 15, decreases the frequency with each jump.
140 when the jump-count exceeds 22, decreases the frequency at a faster rate.
210 turns the tone generator on to the designated frequency.
230 turns the tone generator off.
280 turns the volume control off at the end of the routine.

"THAT" DOG, AGAIN!

Do you recall the little dog we created in Chapter 4 to walk across the screen? Well, here he is again. We will use him to illustrate the addition of sound and color to an existing graphics routine. First, let's re-create the dog.

Program Listing (Graphics)

```
25  POKE 36879, 8 : POKE 646, 1
40  A$ = " SHFT-J CMDR-R CMDR-R SHFT-W SHFT-L
    CRSR SHFT-L CRSR SHFT-L CRSR SHFT-L CRSR"
50  PRINT " SHFT-CLR"
60  PRINT " D CRSR D CRSR D CRSR"
70  FOR X = 1 TO 25
```

```
80  C$ = " SPACE " + A$
90  IF X < 4 THEN C$ = MID$ (A$, 5 – X, 2 * X)
100 IF X = 15 THEN FOR Y = 1 TO 500 : NEXT : C$ = " CMDR D "
    + A$
110 IF X > 21 THEN C$ = " SPACE " + LEFT$ (A$, 25 – X) + RIGHT$
    (A$, 25 – X)
120 PRINT C$;
130 FOR Y = 1 TO 100 : NEXT
140 NEXT X
```

Analysis

25 sets the screen and PRINT colors.
40 sets A$ equal to the dog design plus 4 left-cursor movements.
50 clears the screen and homes the cursor.
60 moves the cursor down three lines.
70 sets a 25-count loop to move the design.
80 sets C$ equal to a space plus the design.
90 changes the design as it comes from the left border.
100 on a count of 15, inserts another delay and makes a small addition to the design.
110 changes the design as it "goes behind" the right border.
120 PRINTs the design.
130 counts a delay.
140 completes the loop.

Now, we are back where we were in Chapter 4. Let's add a bit of color.

Program Listing (Color Addition)

```
95 IF X = 14 THEN PRINT " CTRL 3 " ;
125 PRINT " CTRL 2 " ;
```

Analysis

95 changes the color on a count of 14.
125 returns the color to the original white.

The addition of sound will complete the picture.

Program Listing (Sound Addition)

```
30  S = 36874 : V = 36878
35  POKE V, 15
122 IF X = 14 THEN FOR Y = 1 TO 150 : POKE S, 202 + Y / 140 : NEXT :
    POKE S, 0
135 POKE S, 240 : POKE S, 0 : FOR Y = 1 TO 18 : POKE S, 253 : POKE S,
    0 : NEXT
150 POKE V, 0
```

Analysis

30 sets variables to the addresses of the tone generator and the volume control.
35 turns the volume on.
122 on the count of 14, inserts an extra delay and generates a sound.
135 generates a sound with each movement of the design.
150 turns off the volume.

BOUNCING BALL EXPLODES

The final example of a graphics/color/sound routine is longer than the preceding ones. Here, a green ball bounces against a wall and when it falls to the ground, it explodes with flashing colors and sound effects.

We won't analyze this routine in detail — just enough to point out the most interesting techniques incorporated in it. You will find that the results make it worth keying in this listing.

Program Listing

```
10  CO = 37888 : SC = 4095
20  S2 = 36875 : NO = 36877 : V = 36878
30  POKE 36879, 8
.
.
.
100 PRINT " SHFT-CLR CTRL-6";
110 POKE V, 15
120 FOR Y = 1 TO 21 : POKE SC + 22 * X, 97 : NEXT
130 FOR Y = 1 TO 19 : PRINT : NEXT
140 FOR X = 1 TO 19
150 PRINT SPC(X)" SPACE SPACE SHFT-U CRSR SHFT-L
    CRSR SHFT-Q SHFT-U CRSR"
160 POKE S2, 254 - X
170 FOR Y = 1 TO 15 * (5 + X) : NEXT
180 NEXT X
190 POKE CO + 21, 2 : POKE S2, 175
200 D = 2 : GOSUB 700
210 PRINT "HOME";
220 FOR X = 1 TO 20
230 Z = X : IF Z > 9 THEN Z = 9
240 PRINT SPC(20 - Z) " SPACE SPACE D CRSR L CRSR L
    CRSR SHFT-Q SHFT-U CRSR"
250 POKE S2, 234 + X
260 IF Z = 9 THEN Z = 23
270 FOR Y = 1 TO 15 * (25 - Z) : NEXT
280 NEXT X
```

```
290 POKE S2, 0
300 PRINT SPC(11) " CTRL-3 SHFT-Q SHFT-U CRSR"
310 G = S2 : F = 130 : D = 5 : GOSUB 700
320 PRINT SPC(11) " SHFT-W SHFT-U CRSR"
330 G = NO : F = 240 : D = 7 : GOSUB 700
340 PRINT SPC(10) " CTRL-8 CTRL-9 CMDR-* CTRL-0
    CTRL-3 SHFT-W CTRL-8 CTRL-9 SHFT-ENG PND
    CTRL-0 SHFT-U CRSR SHFT-U CRSR"
350 D = 3 : GOSUB 700
360 PRINT SPC(9) " CMDR-* SPACE SPACE SPACE SHFT-
    ENG PND"
370 PRINT SPC(9) " CMDR-I CTRL-9 CMDR-* CTRL-0
    SPACE CTRL-9 SHFT-ENG PND CTRL-0 CMDR-I
    SHFT-U CRSR SHFT-U CRSR SHFT-U CRSR" : GOSUB
    700
380 PRINT SPC(8) " SHFT-M " SPC(5) " SHFT-N " : PRINT :
    PRINT SPC(8) " CMDR-P " SPC(5) " CMDR-P SHFT-U
    CRSR SHFT-U CRSR SHFT-U CRSR " : D = 8 : GOSUB
    700
400 FOR X = 1 TO 7
410 POKE 646, X
420 FOR Y = 1 TO 3 : PRINT SPC(8) " SPACE (7 times)" : NEXT :
    PRINT " SHFT-U CRSR (4 times)"
430 D = 1 : GOSUB 700
440 PRINT SPC(8) " SHFT-M" SPC(5) " SHFT-N" : PRINT SPC(9)
    " CMDR-* SPACE SPACE SPACE SHFT-ENG PND" :
    PRINT SPC(8) " CMDR-P CMDR-I CTRL-9 CMDR-*
    CTRL-0 SPACE CTRL-9 SHFT-ENG PND CTRL-0
    CMDR-I CMDR-P SHFT-U CRSR SHFT-U CRSR
    SHFT-U CRSR"
450 D = X : GOSUB 700
460 NEXT X
470 POKE NO, 0 : POKE V, 0
480 PRINT " SHFT-CLR"
500 GOTO (continue program)
700 POKE G, F
710 FOR Z = 1 TO D * 100 : NEXT
720 POKE G, O
730 RETURN
```

As the ball goes higher into the air, its speed decreases just like the real thing (line 170) and, on the way down, its speed increases (line 270). The sound changes accordingly. When it reaches the ground, the ball changes color (line 300) and it explodes as blast streaks shoot out accompanied by sound (lines 340-380).

You may find lines 400 through 460 of special interest. The initial blast is followed here by a series of additional explosions with sounds, flashing streaks, and changing colors.

This routine was designed just like the others: the picture was created then the color and sound were interleaved separately. Follow this procedure and all that will be left to do is some fine-tuning of the effect.

CHAPTER 13

Program Statement Structure

PROGRAM STATEMENTS IN MEMORY

If you intend to do more than casual programming, you will find it very useful to know how the program statements are stored in your VIC 20. Not only will such knowledge help you to make more efficient use of available RAM, but it will be a real time and effort saver in many advanced situations.

Once again, let's use the VIC 20 as well as this text to get a handle on this topic. It may pain you a bit to do so, but remove any add-on memory that you may have in place. Then, your stripped-down VIC 20 will have the same addresses you find here. (For later use with additional memory, translate these addresses to match the amount of memory as discussed in Chapter 5.)

Every program statement in memory can be divided into four segments with these names and space requirements:

LINK	LINE #	TEXT	END
(2)	(2)	(varying)	(1)

The LINK is a two-byte number that points to the next line; i.e., the link on line 10 will point to the location of line 20. This number is in the standard form of low byte followed by high byte. The purpose of the link is to enable your VIC 20 to quickly locate a line number. For example, when the program line GOTO 1120 is executed, the search begins at the start of BASIC and jumps from one link to the next until it finds one that is followed by 1120. If there were no links, such a search would have to check every memory location until 1120 was found — a much slower process. Altogether, a great deal of time is saved. Consider all the line finding caused by the multitude of GOTOs, GOSUBs, THENs, inserting, deleting, editing and so on.

The *line number* is necessary to keep track of the order of program execution. It, too, is kept in low byte/high byte form.

The TEXT is not stored on a letter-for-letter basis as you type it in. When they are not within quotation marks, BASIC "keywords" are stored as tokens (codes or numbers) that stand for those words. A keyword may be a command (LIST, RUN); an operator (AND, OR); a statement (CLR, DATA); or a function (FRE, RND). A complete list of VIC 20 keywords and their tokens is found in Appendix B. All other text characters are stored as ASCII values (see Appendix G). The use of keyword tokens saves a considerable amount of memory space since, for example, the number 153 requires only one byte because it is not greater than 255, while "PRINT" takes five bytes.

The final segment of a program statement is the END or terminator. This requires only one byte which is always a zero.

Let's put all this in perspective with an example of "real" statements. We will be talking about the following program in memory, so load it into your 5K machine exactly as shown (and perhaps you should SAVE it because we will be using it quite a bit):

```
10  PRINTCHR$(147)
20  PRINT"THIS IS A TEST"
30  PRINT"       PROGRAM"
245 PRINTTAB(10);:FORX=4097TO4099:PRINTPEEK(X);:NEXT
250 FORX=4100TO4184
255 IFX/5=INT(X/5)THENPRINT"     ";
260 A$=STR$(PEEK(X))
265 IFLEN(A$)<4THENA$ = " " + A$:GOTO265
270 PRINTA$;
275   NEXT
280 END
```

When you RUN this program, the screen will appear as shown in Chart 13-1 (without the numbers in parentheses). Note that the four-digit numbers in parentheses are the addresses (memory locations) of the first byte in each line. Thus, the byte stored at 4105 is 52 and at 4106, it is 55.

Links and Line Numbers

Address 4097 is the beginning of the program — the first statement — in this case, the line number 10. One way you know this is because the third and fourth bytes hold the number 10 (high byte is 0 x 256 = 0; low byte is 10; 0 + 10 = 10).

Looking at the contents of memory, we find a zero at 4108. This indicates the end of the statement, so the next statement must begin at 4109. Back at bytes 1 and 2 (the link at 4097 and 4098), the values are 13 and 16, respectively. Thus, 16 x 256 = 4096 and 4096 + 13 = 4109, which is where we already found the beginning of the statement containing line 20.

The next link is in the first two locations of the second statement — 4109 and 4110. Those numbers are low byte 35 and high byte 16 which work out to 4131. Going down to 4131 and backing up one address to 4130, we find a zero indicating the end of a statement.

Just in case you are skeptical of the two bytes for line numbers, PEEK at 4225 and 4226, where line number 260 is stored. There you will find 4 and 1, respectively, which translate into 260. Of course, you will want to look at a few bytes before and after those to make sure you have the right ones.

There is one more point on links that you should check out. The last statement (line 280) has a link of 191/16 or 4287. If you follow that pointer, you will find that both 4287 and 4288 contain zeros. A link of zero is the VIC 20's notation for the end of a program. In fact, counting the end of statement zero at 4286, there are three consecutive zeros — a sure end-of-program sign.

That should be enough on links and line numbers for you to follow statements through any program. If you have trouble converting these low-byte and high-byte numbers, you may wish to review the pertinent sections in Chapter 3.

Tokens (Keyword Codes)

Take a look at the statement of line 10 again (4097–4108). After the line number, you find the values 153, 199, 40, 49, 52,

Chart 13-1. Screen Display From Example Program

```
                    THIS IS A TEST
                       PROGRAM
(4097)                         13     16     10
(4100)         0     153      199     40     49
(4105)        52      55       41      0     35
(4110)        16      20        0    153     34
(4115)        84      72       73     83     32
(4120)        73      83       32     65     32
(4125)        84      69       83     84     34
(4130)         0      55       16     30      0
(4135)       153      34       32     32     32
(4140)        32      32       80     82     79
(4145)        71      82       65     77     34
(4150)         0      87       16    245      0
(4155)       153     163       49     48     41
(4160)        58     129       88    178     52
(4165)        48      57       55    164     52
(4170)        48      57       57     58    153
(4175)       194      40       88     41     59
(4180)        58     130        0    104     16
READY
```

55, 41, and then zero, signaling the end. The 153 and 199 are keyword tokens for PRINT and CHR$. The remaining numbers are ASCII values for (, 1, 4, 7, and). In line 245, the 153 is PRINT, 163 is TAB(, 129 is FOR, 178 is TO, and so on. All the values higher than 100 are tokens. Indeed, they save a considerable amount of memory.

There are, of course, other series of codes that extend beyond 100. You can distinguish between them by the way in which they are used. Tokens, for example, are not found in link or line number locations nor are they PRINTed, POKEd, or PEEKed.

Insert and Delete

Further insight about the way your VIC 20 handles statements may be found by "doctoring" the previous program and observing the results. What happens when you insert a line? — the best way to find out is to try it and see. If you type

```
15 PRINT
```

and RUN the program, you will find that the VIC 20 has moved everything after line 10 up in memory just far enough to put that statement between lines 10 and 20. Of course, the links are changed accordingly.

If, on the other hand, you delete line 20, the VIC 20 simply does the reverse. It changes the links and lowers the upper part of the program to fill the vacated memory locations.

A little more experimenting with that program will show you that the same things happen when characters are inserted and deleted within a line. The entire program expands or contracts to meet the need. Of course, no line numbers are changed; however, statement locations do change and the links are modified to track them properly.

APPLICATIONS

You will recall that it was stated that this knowledge would come in handy for the advanced programmer. Well, you can use this information even if you are not yet "advanced." Some of the Notes in this book get right into program statement modification.

First of all, you will now have some appreciation for what the VIC 20 has to do when you insert or delete even a single character in a line. Especially if that line is near the beginning of a long program, a great deal of moving and switching is done. Now, perhaps you won't be so impatient when there is a delay of a few seconds before the READY sign comes up.

From previous discussions, you are aware that just as you can PEEK into a memory location and determine the value stored there, you can POKE new values as well. That fact leads to a number of interesting applications.

You know that you cannot CONTinue a program after it has been STOPped if any editing has been done. One of the reasons is that all variables have lost their values and there is nothing with which to CONTinue. Now, you can do certain kinds of editing and still CONTinue or GOTO a given program line with variable values intact.

Going back to the original "workhorse" program, RUN it and then enter PRINT A$ (direct mode). Of course, the value 16 is printed. If you were to delete the 7 in 147 in the first line in the regular way (edit) and ask for A$, you would draw a blank. However, if you POKE 4106, 32, the seven will be replaced with a blank and leave 14 in the parentheses. Now, you can still get the value of A$, indicating that the variables are still there. Later, you can return to the LISTing and DELete the space after the four.

Any part of a statement can be edited directly but the length cannot be changed in this way. Normally, this restriction is an inconvenience at most. You may even change the line numbers as long as they are not referred to by a GOTO, GOSUB or the like. A renumbered line LISTs in the same place as the original line — a situation that may confound users of your program. Some line number changes will give startling results, so practice a bit to discover the limitations, such as not having a higher numbered line before a referenced one.

Did you ever want to "hide" a piece of information in one of your programs — your identification, perhaps? Load up our test program and try this:

```
POKE 4124, 34 : POKE 4125, 0
```

LIST the program and you will find that you have lost part of line 20. RUN it to see that it is missing, indeed. If you examine the PEEKs, however, you will discover that the last characters are still there but they are not executed. The VIC 20 thinks the line ended with the zero. Those characters could be your initials, the date, or whatever and no one will suspect that they are there if you don't overdo it.

You can, of course, POKE zeros into the two addresses of any link and cause the program to end at that point. There is little practical use for that procedure but a related one is quite interesting. With the original program in the machine (not again!),

```
POKE 4097, 35
```

and RUN. You will see no difference in the operation of the program. LIST it and you will *not* see line 20 — it simply does not show up. Now, POKE 55 in the same location and both lines 20 and 30 will "disappear" even though they execute just as before.

What you have done, of course, is to change a link to jump one or more lines. The "skipped" lines function but do not LIST. This technique is used in the next chapter. When you use it, be careful not to jump a referenced line or your program will crash because the VIC 20 won't be able to find that line number!

UTILITY PROGRAMS

On a more practical level, this chapter will conclude with three utility programs: Compress, Pack, and Renumber. They are given here because each one illustrates the application of the preceding information. In addition, of course, each is very useful to anyone who is writing or modifying programs.

These utilities are written entirely in BASIC. This means that you can follow the action easily and modify it to suit your special requirements. Another consequence of the language is that the utility programs execute at the relatively slow speed of BASIC. Even so, they perform their functions much faster than these jobs can be done manually.

While these three utilities can be used independently, they are most valuable when used sequentially. For example, Pack leaves many spaces in a program and they should be eliminated by using Compress.

The same procedure is used with each of these programs — each must be loaded into the VIC 20 right along with the program on which it is to operate. The steps of this procedure are:

1. LOAD the program to be Packed, Compressed, or Renumbered.
2. Enter: POKE 43, PEEK (45) : POKE 44, PEEK (46).
3. LOAD and RUN the utility.
4. After the run, execute the two POKEs as instructed.

As you will recall, step 2 moves the "beginning of program" pointers (43/44) to the top of the previously loaded program (45/46). This, of course, causes the utility to load above that program.

Note, too, that just as promised earlier, the values in these program listings are the ones you need for an unadorned (5K) VIC 20. If you have added memory, the value of 4097 must be changed to accommodate that addition (review Chapter 5 for details).

Compress

The Compress utility program does precisely what its name indicates: it squeezes all the spaces out of a program. If your program writing is done with spaces to make for easy reading, Compress will eliminate all except those in quotation marks and, thereby, make your program take up less memory and execute faster. It is especially useful after running Pack which leaves plenty of spaces where it removes unnecessary links and line numbers.

You may enter the Compress program directly from Listing 13-1. As usual, check your typing carefully. Of course, you may leave out the "easy reading" spaces in the listing or type it just as shown and, then, squeeze them out with Compress, itself.

Looking at both the listing and the flowchart (Fig. 13-1) of the Compress program, you will see three major sections. The first includes lines 100 through 210 and its purpose is to examine each character in each line of the program. If it encounters a quotation mark (34), it skips along quickly until it finds a second one — thus, not removing any spaces inside quotes. Any other spaces (32), however, will send execution to the second and third sections.

The second section, lines 230-300, decrements (subtracts one from) every link beginning with the current one and continuing to the top of the program. When this is completed, lines 320–360, the third section, begin at the memory location where the space was found and move the value in every higher address down one location. Then, the execution goes back to the examination of character after character, looking for another space that is not in quotes.

Line 355, by the way, prints a dash on the screen every time a space is removed. It has no real value except to reassure you that Compress is working. Otherwise, you might begin to wonder when you compress a long program and see nothing happening!

Renumber

Renumber is a very handy utility for the program writer/modifier. Most of us write lines in increments of 10 or more and begin with 10 or 100. We do that for good and sufficient reason, yet we are left with large and irregular line numbers to carry in our programs forever. Renumber will take care of that problem and give you small, regular numbers for efficiency and good looks.

This Renumber program (Listing 13-2) does not simply change the line numbers. It changes any referenced numbers within the statements to match the new line numbers. Thus, every THEN, GOTO, and GOSUB will reference the new lines so the program can run properly.

Examination of the listing and the flowchart (Fig. 13-2) will show you that this utility is divided into two major sections. The first, lines 100–210, renumbers the lines according to the

Listing 13-1. Compress Program

```
100 X = 4097 : T = PEEK(44) * 256 + PEEK(43) - 2
110 W = PEEK(X) : W1 = PEEK(X+1)
120 FOR Y = X + 4 TO T
130 Z = PEEK(Y)
140 IF Z = 0 THEN X = Y + 1 : GOTO 110
150 IF Q THEN 180
160 IF Z = 34 THEN Q = 1 : GOTO 190
170 IF Z = 32 THEN X1 = X : GOTO 240
180 IF Z = 34 THEN Q = 0
190 NEXT
200 POKE 43, 1: POKE 44, 16
210 T = T + 2 : W = INT (T/256) : PRINT "POKE 45," T - W * 256 "
    AND 46," W : END
220 REM — DECREMENT LINKS
230 W = PEEK(X1) : W1 = PEEK(X1 + 1)
240 W2 = W1 * 256 + W
250 IF W2 = 0 THEN 320
260 W2 = W2 - 1
270 W4 = INT (W2/256) : W3 = W2 - W4 * 256
280 POKE X1, W3 : POKE X1+1, W4
290 X1 = W2 + 1
300 GOTO 230
310 REM — MOVE ALL DOWN ONE
320 T = T - 1
330 FOR X1 = Y TO T
340 POKE X1, PEEK(X1 + 1)
350 NEXT
355 PRINT "- ";
360 GOTO 110
```

specifications you enter for starting number and increment. In addition, it makes a very important record: the old line number, D(C), and the corresponding new number, E(C). Without this matched list of old and new, the referenced numbers could not be corrected.

Lines 220–410 examine the program, character by character and line by line, searching for GOTO (a token value of 137), GOSUB (141), and THEN (167). When one is found, the following characters are checked to determine whether or not they represent a line number. If so, the corresponding new number is POKEd in its place.

Usually, Renumber is used to lower the line numbers in a program. When the new number is shorter than the old referenced number, line 360 pads out the memory locations with spaces. If,

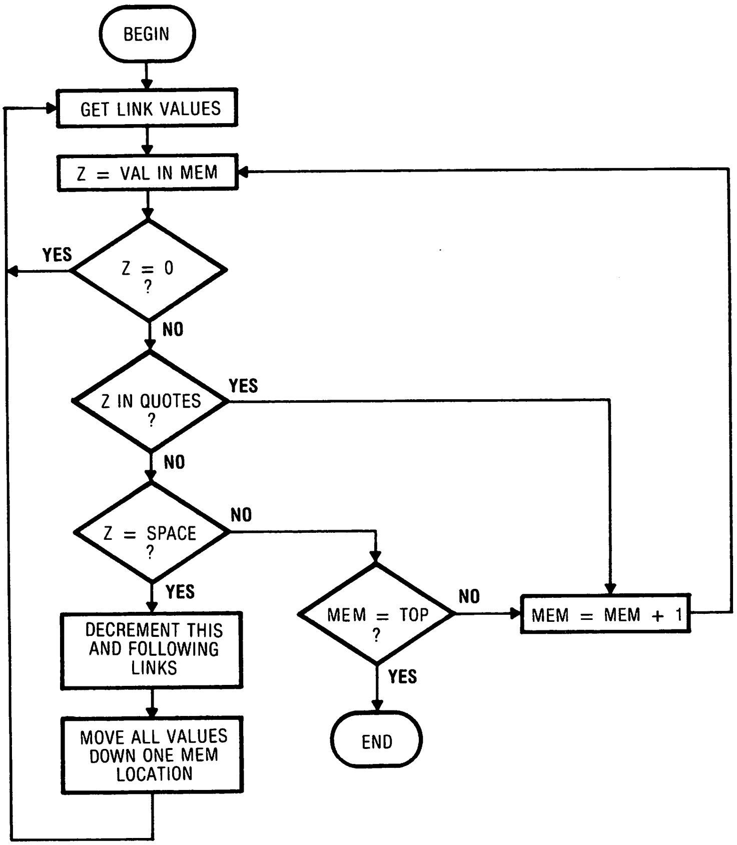


Fig. 13-1. Flowchart of compress utility.

for some reason, you are changing to longer line numbers (100, 200 instead of 10, 20 or 1, 2), additional space must be made in the references to lines. The simplest way to do this is to insert one or two colons or spaces after each reference number; then run Renumber and Compress the spaces away (or delete the colons).

Of course, you could add a section to the program to take care of longer line numbers automatically but it hardly seems worth the extra baggage since this facility would be used so seldom. The same thought applies to adding the ability to specify the old line numbers with which the action is to begin and end.

Listing 13-2. Renumber Utility Program

```
100 DIM D(50), E(50): X = 4097 : T=PEEK(44)*256+PEEK(43)-2
110 INPUT "START #";A
120 INPUT "INCREMENT ";B
130 Y = X + 2
140 Z = PEEK(Y) : Z1 = PEEK(Y+1)
150 W = Z1 * 256 + Z
160 C = C + 1 : D(C) = W : E(C) = A
170 POKE (Y+1), INT (A/256): POKE Y, A - INT (A/256) * 256
180 A = A + B
190 Y = PEEK(Y-1) * 256 + PEEK(Y-2)
200 IF Y = > T THEN 220
205 PRINT "#";
210 Y = Y + 2 : GOTO 140
220 FOR Y = X + 4 TO T
230 Z = PEEK(Y)
240 IF Z = 0 THEN X = Y + 1 : GOTO 220
250 IF Z <> 137 AND Z <> 141 AND Z <> 167 THEN 410
260 T$ = " " : FOR N = Y + 1 TO Y + 6
270 Y2 = PEEK(N)
280 IF Y2 < 48 OR Y2 > 57 THEN N = Y + 6 : GOTO 300
290 T$ = T$ + CHR$(Y2)
300 NEXT
310 P = VAL (T$) : IF P = 0 THEN 410
320 FOR N = 1 TO C
330 IF P = D(N) THEN M = N : N = C
340 NEXT
350 G$ = STR$(E(M)) : G$ = MID$(G$,2)
360 IF LEN(T$) > LEN(G$) THEN G$ = G$ + "" : GOTO ~~360~~ 420
370 FOR N = 1 TO LEN(G$)
380 POKE Y + N, ASC (MID$ (G$,N,1))
390 NEXT
400 Y = Y + N
410 NEXT
420 POKE 43, 1 : POKE 44, 16
430 T = T + 2 : W = INT (T/256)
440 PRINT " POKE 45," T - W * 256 "AND 46," W : END
```

Pack

The preceding utilities are nice to have around but their results could be duplicated with some judicious editing. The editing work to compress or renumber is not too bad unless the program is a long one. The Pack utility, however, is a different matter.

As has been pointed out, it is advisable to write programs in short, single-statement lines. Long, multistatement lines are hard to read and hard to debug. So, we want to write the former

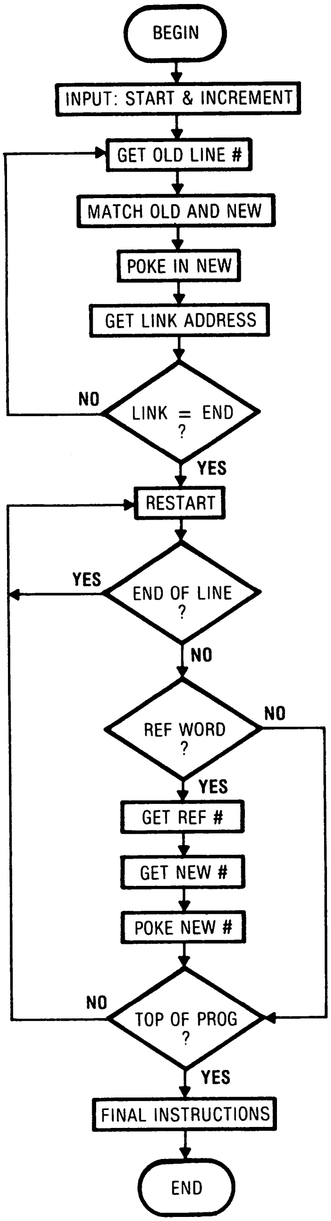


Fig. 13-2. Flowchart of renumber.

but run the latter in order to conserve memory and get the most speedy execution. Manually changing from single to multistatement lines in an existing program is a time-consuming process even when that program is short. A great deal of retyping and deleting is required. The Pack utility will "pack" your program into multistatement lines and save you all that work.

Of course, a referenced line cannot be packed into the midst of a multistatement line where it would lose its number and could not be found by the program. For instance, line 130 cannot be packed behind 120 if there is a GOTO 130 in your program. The utility must not allow this to happen. Also, REMark and IF statements must not be packed except at the end of a line. A line packed in behind a REM will be ignored and behind an IF, it would be executed sometimes and ignored at other times.

As shown in Listing 13-3 and Fig. 13-3, the first part of Pack (lines 100-290) finds all referenced line numbers so that they will not be "buried" in the middle of a line. In addition, it finds the line numbers of lines following IF and REM statements to prevent them from being buried and changing the program design.

The second section, lines 300–490, determines the number and length of each line. If the total length of lines one and two is less than 85, they are combined into one and that length plus line three is checked. When a referenced line number is found, the combining stops regardless of the lengths and the process is restarted to see if anything can be added to the referenced line.

When Pack ends, each line that has been packed to another contains four space characters — two in place of the original link and two in place of the original line number. A colon has been substituted for the original "end of statement" marker (0). The packed program should be compressed and then renumbered for both practical and aesthetic reasons.

UTILITY SUMMARY

It is obvious that a number of refinements could be added to the three utilities presented previously. Some have been mentioned. Another that comes to mind is to combine the three into one big menu-driven utility. Because these programs are written in BASIC, refinements and modifications are easily accomplished. Before fancying them up, however, consider the consequences carefully.

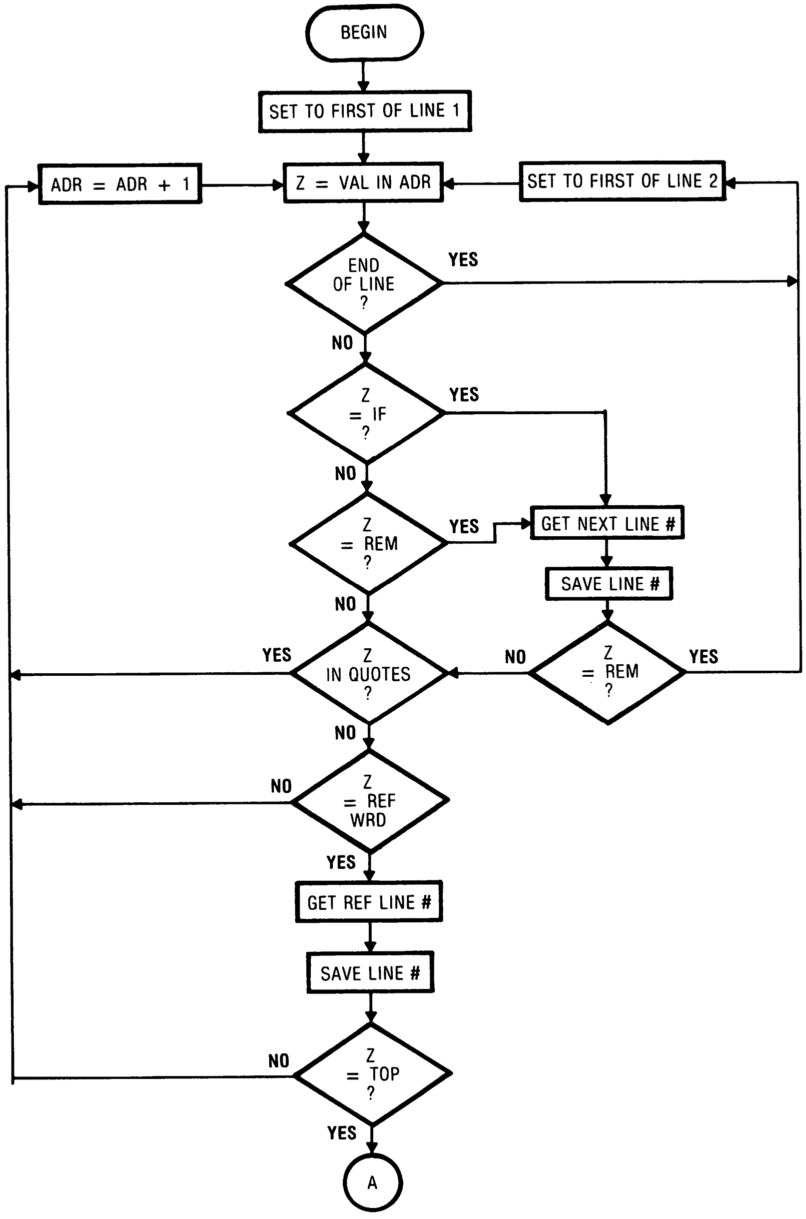
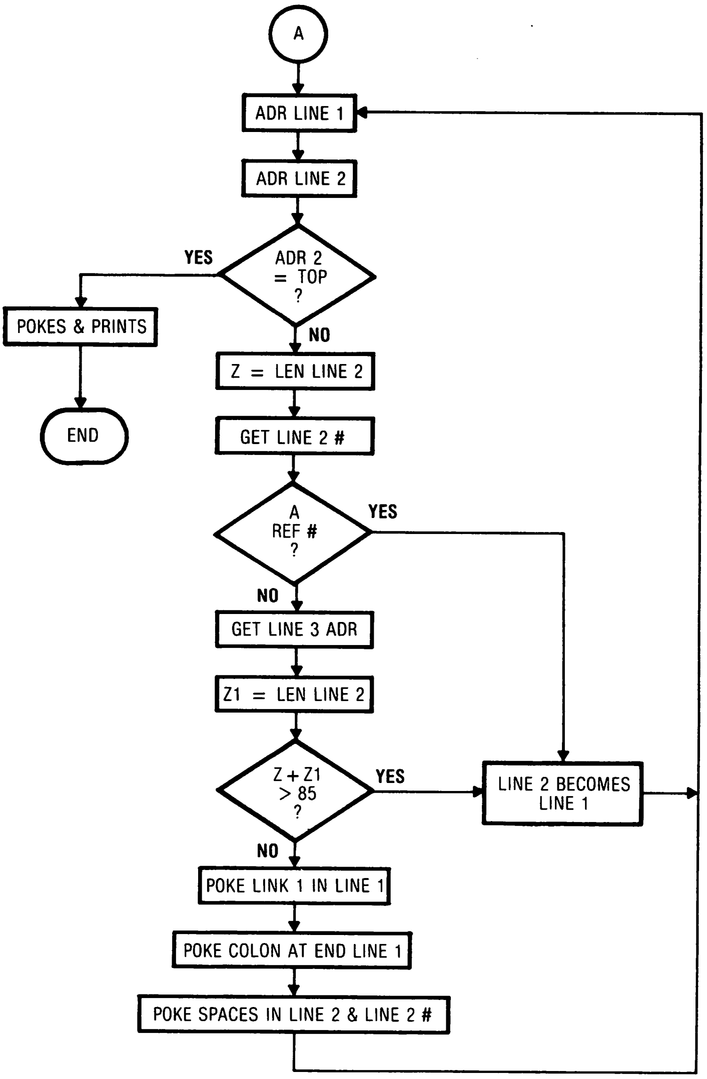


Fig. 13-3. Flowchart

The greatest potential disadvantage is the amount of memory required by a long utility. The longer it is, the more space it requires and that decreases the amount of memory available to hold the program on which the utility is working. This could be a serious problem on a 5K or 8K machine. The three short "parts"



of pack utility.

Listing 13-3. Pack Utility Program

```
100 DIM A(50) : X = 4097 : T = PEEK(44) * 256 + PEEK(43) - 2
110 FOR Y = X + 4 TO T : Z = PEEK(Y)
120 IF Z = 0 THEN X = Y + 1 : Q1 = 0 : GOTO 110
130 IF Z = 139 THEN Q1 = 1 : GOTO 150
140 IF Z <> 143 THEN 180
150 U = PEEK(X + 1) * 256 + PEEK(X) : P = PEEK(U + 3) * 256 +
    PEEK(U + 2)
160 N = N + 1 : A (N) = P
170 IF Q1 = 0 THEN X = U : GOTO 110
180 IF Q THEN 280
190 IF Z = 34 THEN Q = 1 : GOTO290
200 IF Z <> 137 AND Z <> 141 AND Z <> 167 THEN 290
210 T$ = "" : FOR Y1 = Y + 1 TO Y + 6
220 Y2 = PEEK(Y1)
230 IF Y2 < 48 OR Y2 > 57 THEN Y1 = Y + 6 : GOTO 250
240 T$ = T$ + CHR$(Y2)
250 NEXT
260 P = VAL (T$) : IF P = 0 THEN 280
270 N = N + 1 : A (N) = P
280 IF Z = 34 THEN Q = 0
290 NEXT
299 REM — COMBINE LINES
300 X = 4097
310 W = PEEK(X) : W1 = PEEK(X + 1)
320 X1 = W1 * 256 + W
330 IF X1 = > T THEN 500
340 Z = X1 -X
350 X2 = PEEK(X1 + 2) : X3 = PEEK(X1 + 3)
360 X4 = X3 * 256 + X2
370 FOR V = 1 TO N
380 IF A(V) = X4 THEN X = X1 : GOTO 310
390 NEXT
400 Y = PEEK(X1) : Y1 = PEEK(X1 + 1)
410 X2 = Y1 * 256 + Y
420 Z1 = X2 - X1
430 IF Z + Z1 > 85 THEN X = X1 : GOTO 310
440 POKE X, Y : POKE X + 1, Y1
450 POKE X1 - 1, 58
460 FOR V = 0 TO 3
470 POKE X1 + V, 32
480 NEXT
490 GOTO 310
500 POKE 43, 1 : POKE 44, 16
510 T = T + 2 : W = INT (T/256)
520 PRINT "POKE 45," T - W * 256 "AND 46," W
530 END
```

as given here are much less likely to cause a memory problem.

Whether or not you make changes in these utilities, don't forget to accommodate any differences caused by added memory. Address 4097 is not the beginning of BASIC programming after you add memory and neither is POKE 43, 1 : POKE 44, 16.

CHAPTER 14

Joystick, Paddles and Light Pen

One of the great advantages of the VIC 20 is that it is so easy to add external controls. The most common such controls are the joystick, paddles, and light pen. Each one is simply plugged directly into the game port on the right side of the machine. No adapters or other connections are needed.

Of course, a bit of programming is needed in order to "read" the values of the external control(s) and make them available for your use. Because of the VIC 20's circuitry, the programming is short and straightforward.

Each of these three external controls will be examined in this chapter. There is a simple, illustrative program given for each device. The programs will show you how to incorporate the controls into your programs and help you to understand how they work.

Joysticks, paddles, and light pens are usually perceived as adjuncts to game programs. In fact, you will find them as controllers of action in games almost exclusively. As you study this chapter and as you write your own programs, keep in mind that they can be quite useful in other types of programs. For example, any one of them can be valuable to an operator who is unable to use the keyboard effectively because of age or physical handicap. Be on the lookout for such applications.

THE JOYSTICK

The joystick is the most common external controller for the VIC 20. The stick or handle is used to control the direction of action on the screen. The "fire-button" or switch is used to control any type of special action, usually the firing of a weapon.

There are four switches in the joystick exclusive of the fire-button. Using the four points of the compass for reference, there is a switch at North, East, South, and West. When the stick is pushed in one of these directions, the corresponding switch is closed. Further, if the stick is pushed in a direction between these points, the two adjacent switches are closed. For example, when the stick is pushed Northeast, both the North and East switches are closed.

By "reading" the joystick switches, the VIC 20 determines the direction toward which you have pushed and can take action accordingly. There are, then, two major tasks for a program using this controller. The first is to read the switches and the second is to translate that reading into terms that can be acted upon.

Reading the switches is a matter of checking the ports connected to the game control socket. This is done in the following program in the subroutine in lines 900–940. The values found in the ports are converted to X and Y coordinates.

Lines 200–220 convert these coordinates to compass points as shown in Fig. 14-1. Thus, if the North switch is closed, the value of P in line 310 is 1. If the direction is Northeast, the value is 2 and so on.

Joystick Art

This program enables the user to make graphic designs on the screen. Colors and graphic blocks are selected at the keyboard and the joystick determines the direction of movement when the fire-button is pressed. Note, especially, the manner in which the joystick input is handled.

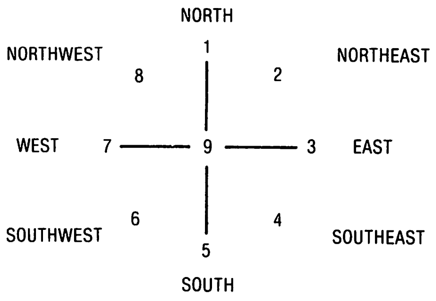


Fig. 14-1. The joystick "compass."

Listing

```
90  POKE 36879,10
100 DIM Q (2,2) : R1 = 37139 : R2 = 37154
110 PL = 3
120 POKE R1, 0
200 Q (0, 0) = 8 : Q (1, 0) = 1 : Q (2, 0) = 2
210 Q (0, 1) = 7 : Q (1, 1) = 9 : Q (2, 1) = 3
220 Q (0, 2) = 6 : Q (1, 2) = 5 : Q (2, 2) = 4
290 PRINT " SHFT-CLR D CRSR (9 times) R CRSR (11 times) .";
300 GOSUB 900 : IF F = 0 THEN 300
310 P = Q (X, Y)
320 IF P = 9 THEN P = PL
330 PL = P
340 GET A$ : IF A$ = " " THEN A$ = B$
350 IF A$ = " ENG POUND " THEN POKE R1, 128 : END
360 B$ = A$
370 ON D GOTO 400, 410, 430, 420, 440, 450, 460, 470, 480
400 PRINT " SHFT-U CRSR "; : GOTO 480
410 PRINT " SHFT-U CRSR "; : GOTO 430
420 PRINT " D CRSR ";
430 PRINT A$; : GOTO 300
440 PRINT " D CRSR "; : GOTO 480
450 PRINT " D CRSR SHFT-L CRSR "; : GOTO 480
460 PRINT " SHFT-L CRSR "; : GOTO480
470 PRINT " SHFT-U CRSR SHFT-L CRSR ";
480 PRINT " SHFT-L CRSR " A$ ; : GOTO300
900 POKE R2, 127 : R = -((PEEK (37152) AND 128) = 0) : POKE R2,
    255
910 V = PEEK (37137) : D = -((V AND 8) = 0)
920 L = (V AND 16) = 0 : U = (V AND 4) = 0 : F = -((V AND 32) = 0)
930 X = L + R + 1 : Y = U + D + 1
940 RETURN
```

Analysis

90 sets the color of the screen.
100 dimensions the Q variable and sets R1 and R2 equal to the addresses of the direction controls for the ports.
110 establishes the initial direction of movement (East).
120 sets port A for input.
200–220 set up the array to correspond with the "compass" in Fig. 14-1.
290 clears the screen and PRINTs a dot in the center.
300 calls the subroutine to read the joystick and, if the fire-button was not pressed, repeats the call.
310 sets P equal to the compass direction as determined by the returned values of X and Y.
320 if P is 9 (stick centered) continues the previous direction.
330 updates the direction "holder."
340 checks the keyboard buffer and, if empty, continues the previous input character or color.
350 if the character is an English pound, reverses the direction of port A and ends the program.
360 updates the character holder.
370 transfers execution to the line containing action corresponding to the compass direction of movement.
400–480 these lines move the cursor in various directions; PRINT the proper character, graphic, color or space; and transfer back to the call routine.
900 changes the direction of port B to input long enough to check the No. 3 (East) switch.
910 reads the input at port A and checks the 5/South switch.
920 checks the 7/West and 1/North switches and the fire-button.
930 combines the switch readings to values for X and Y.
940 transfers back to the main program.

Use

The statements and technique in this program can be transferred directly to any other. The program, itself, can be used to create very intricate designs.

Variations

The possible variations on this program and the use of the joystick are extremely wide. Use your imagination to apply the technique to all manner of programs.

You may wish to have the cursor (and design) move whenever the stick is pushed away from the center position. Then the fire-button could be used to randomly select a color or a graphic block. Don't overlook the possibility of using the joystick for special actions within a keyboard-operated program. For example, the stick could be used to select an answer from among several presented.

PADDLES

In many respects, paddles are very much like joysticks. They control the action on the display usually by moving the cursor or a graphic design. They, too, have push switches that serve as fire-buttons or initiators of other special actions. Of course, paddles come in pairs and they are often used together.

There are two major differences between a paddle and a joystick. The first is that the paddle normally controls in only one dimension. It causes the cursor or design to move horizontally or vertically. Of course, you can use the paddle values to cause action in two dimensions but that takes more than just a bit of programming.

The second way in which a paddle differs from a joystick is that it permits a finer degree of control. Instead of on/off switches, the paddle contains a variable resistor that may have an effective value from zero to 255 as read by the ''VIC'' chip. This means that you can control something in 256 steps rather than in just two (on/off). That degree of fineness of movement is seldom used but it does give the potential of quite realistic action.

Music Master

This little program will demonstrate how the paddle values are read. Here, they are used to control sound generators, so you can create music — if you have the talent!

Listing

```
100 POKE 36879, 8 : POKE 646, 7
110 PRINT CHR$ (147) ;
120 POKE 37139, 0
130 DR = 37154
140 X = 200 : Y = 10
200 GOSUB 400
210 IF X2 <> X THEN POKE SG, X
220 IF Y2 <> Y THEN POKE 36878, Y
230 IF F1 THEN POKE SG, 0 : SG = 36876
```

```
240 IF F2 THEN POKE SG, 0 : SG = 36874
260 X2 = X : Y2 = Y
270 FOR Z = 1 TO 500 : NEXT
290 GOTO 200
400 X = 254 - INT (PEEK (36872) / 2) : Y = INT (PEEK (36873) / 17)
410 POKE DR, 127 : F2 = -(( PEEK (37152) AND 128) = 0) : POKE DR,
    255
420 F1 = -(( PEEK (37137) AND 16) = 0)
430 RETURN
```

Analysis

100 sets the display and character colors.
110 clears the screen and homes the cursor.
120 changes the direction of port A in VIA No. 1.
130 sets DR equal to the address of port B in VIA No. 2.
140 sets the initial values for frequency and volume.
200 calls the subroutine to read the paddle values.
210 if the value of the frequency variable has changed, POKES the new value into the sound generator.
220 if the value of the volume variable has changed, POKES the new value into the volume control.
230 if fire-button No. 1 was pressed, turns off the sound generator and changes the variable address to the other generator.
240 does the same as line 230 if fire-button No. 2 was pressed.
260 updates the values in X2 and Y2.
270 delays for a 500-count.
290 transfers back to repeat the process.
400 sets X to the value of paddle No. 1 after converting it to range from 127 to 254 (as suitable for the sound generator) and sets y to the value of paddle No. 2 after converting it to range from 0 to 15 (as suitable for the volume control).
410 changes the direction of port B long enough to set F1 to a value of 1 if the fire-button was pressed.
420 sets F2 to a value of 1 if the No. 1 fire-button was pressed.
430 transfers back to the main program.

Use

Aside from the fact that sound is generated as determined by the positions of the paddles, this program shows how the paddle values are read. The subroutine (lines 400–430) can be placed in any program in which you wish to use paddle input. Two additional factors must be accommodated. First, the value of DR must be set before the subroutine is called or 37154 can be put in line 410.

The second factor is the adjustment of line 400 to place the values of X and Y in the proper ranges for the application. The "raw" readings will vary between 0 and 255, which may exceed the values permissible in your program. The raw readings can be made with these statements:

```
X = PEEK (36872)
Y = PEEK (36873)
```

Another example of adjusting the raw readings will be found in the following program.

Variations

The number of possible variations on this program is all but limitless. For a beginning, you could use other sound/noise generator combinations; leave the current generator on when changing from one to another (delete POKE SG, 0 from lines 230 and 240); change the speed with which tones can be changed (line 270); set the volume to a constant value and let each paddle control an individual sound generator; and add some visual effect, such as

```
105 SC = 256 * PEEK (648)
250 POKE SC + INT (RND (0) * 500), 42
280 IF RND (0) < .05 THEN PRINT CHR$ (147);
```

Capture Game

This small game allows two players to compete by capturing (erasing) the graphics scattered about the screen by the opponent while protecting his own. Its main purpose, however, is to provide another illustration of limiting the range(s) of the paddle values to those suitable for the application.

Listing

```
    lines 100-130 from music program, above
140 X = 10 : Y = 10
150 B$ = " SHFT-L CRSR SHFT-L CRSR SHFT-U CRSR
    SHFT-U CRSR SPACE D CRSR SHFT-L CRSR SHFT-L
    CRSR SPACE SHFT-Q D CRSR SHFT-L CRSR SHFT-L
    CRSR SPACE"
200 GOSUB 400 : IF X2 = X AND Y2 = Y THEN 200
210 PRINT CHR$ (19);
220 IF X> 0 THEN FOR Z = 0 TO X : PRINT " D CRSR "; : NEXT
230 IF Y< 0 THEN FOR Z = 0 TO Y : PRINT " R CRSR "; : NEXT
240 IF RND (0) < .5 THEN IF F2 THEN POKE 646, 3 : GOTO 260
```

```
250 IF F1 THEN POKE 646, 7
260 PRINT B$;
270 X2 = X : Y2 = Y
280 GOTO 200
400 X = 21 - INT (PEEK (36872) / 12) : Y = 20 - INT (PEEK (36873)
    /13)
    lines 410-430 from music program, above
```

Analysis

140 sets variables to place ball in middle of screen.
150 sets B$ to a ball design surrounded by spaces.
200 calls the subroutine and repeats if X and Y values are unchanged.
210 homes the cursor.
220 counts the lines for design placement.
230 counts the columns for design placement.
240 half the time changes the design color if fire-button No. 2 was pressed.
250 half the time changes the design color if fire-button No. 1 was pressed.
260 PRINTs the design at the location determined by the paddles.
270 updates the values of X2 and Y2.
280 transfers back to repeat the procedure.
400 sets the values of X and Y as determined by the paddles after conversion of the raw readings to line and column ranges.

Use

Note how the X and Y readings are made appropriate for the specification of screen line and column.

This little game, though rather crude, can get fairly interesting. Each player controls one direction of movement (and speed) as well as one color of the design. In play, there is neat combination of aggression, defense, and sabotage. The winner, of course, is the player with the greatest number of dots of his color on the screen.

Variations

As in the previous example, only your imagination limits the variations on this basic game.

THE LIGHT PEN

The light pen, like the joystick and paddles, is easy to attach to your VIC 20. Just plug it into the game port — no other hardware is needed. Programming, on the other hand, is a bit more complex. The reason for this is that all light pens are not made alike and you have to calibrate your pen, VIC 20 and tv/monitor.

The first thing you must do is connect the pen and find out what range of vertical and horizontal numbers it returns. These statements PRINT two columns of numbers on the screen. The first column is row (vertical) numbers and the second is column (horizontal). By moving your pen from top to bottom of the display and from side to side, you can determine the row and column ranges.

```
10 POKE 36879, 25 : POKE 646, 6
20 PRINT CHR$ (147);
30 R = PEEK (36871)
40 C = PEEK (36870)
50 PRINT R, C
60 GOTO 30
```

Let's use an actual example to illustrate how the calibration is done. In running the program, the "row" numbers ranged from 18 to 120. Of course, there are only 23 lines on the display, so we must change this range to run from 0 to 22. Inserting this statement will do just that:

```
45 R = INT ( (R – 17) / 4.7)
```

Now, R varies neatly between 0 and 22 and can be used to determine the line at which the pen is pointed. In calibrating your setup you may have to juggle the figures a bit to make them come out just right. The column range is handled in the same way.

To keep things simple and clear, we will use only the line (row/vertical) readings in a practical application. Study the following program which uses this calibration.

Listing

```
10  SC = 256 * PEEK (648) : CO = 38400 : IF SC < 5000 THEN CO =
    37888
20  POKE 36879, 29 : POKE 646, 6
90  PRINT CHR$ (147);
100 FOR X = 1 TO 20 : PRINT : NEXT
110 PRINT "TOUCH HERE TO CONTINUE"
120 SK = 1 : GOSUB 800 : SK = 0
130 IF A < 17 THEN 120
190 PRINT CHR$ (147);
200 PRINT "THE CAPITAL OF VA IS" : PRINT
```

```
210 PRINT SPC (5) "WASHINGTON" SPC (34) "RICHMOND" SPC (36)
    "NORFOLK"
220 GOSUB 800
230 PRINT : IF A = 4 THEN PRINT "VERY GOOD !" : GOTO 250
240 PRINT "THAT'S NOT RIGHT"
250 FOR X = 1 TO 5000 : NEXT
260 GOTO 90
800 R = PEEK (36871) - 20
810 A = INT (R / 4.8)
820 C = B : B = A : IF A <> C THEN 800
830 IF SK THEN 860
840 POKE SC + A * 22 + 3, 81
850 POKE CO + A * 22 + 3, 5
860 B = 0 : C = 0
870 RETURN
```

Analysis

10 sets Screen RAM and Color RAM addresses regardless of memory size.
20 sets display and PRINT colors.
90 clears screen and homes cursor.
100 places cursor on line 21.
110 PRINTs message.
120 sets SK flag (to prevent PRINTing of indicator); calls pen subroutine; and resets flag.
130 checks to see if a line below 17 was touched.
190 clears screen and homes cursor.
200 PRINTs a question.
210 PRINTs three answers.
220 calls the pen subroutine.
230 PRINTs "GOOD" message if line 4 was touched.
240 PRINTs "WRONG" message if another line was touched.
250 delays for a 5000-count.
260 repeats the example.
800–810 read the row number and convert it to a line number (note that the conversion needed some fine tuning to function in the program).
820 transfers back to read again unless three consecutive readings are the same.
830 if the SK flag was set, skips the PRINTing of the indicator.
840 POKEs a graphic (ball) at the chosen line.
850 POKEs a color for the graphic.
860 resets variables B and C to zero.
870 transfers back to the main program.

Uses

This program illustrates the use of a light pen in selecting responses. This technique can be copied directly into your programs but you may have to recalibrate the pen so that the line numbers come out right.

Note that the light pen can be especially useful to users who are physically handicapped and find it difficult to operate the keyboard. It is equally valuable when working with children too young to read and/or type (their task might be to match colors, shapes, and so on).

External controllers can add a great deal to many types of programs — not just games. Don't provide for their use simply because they exist. Keep them in mind as you write your programs and add them when they can make a real contribution to its effective operation.

CHAPTER 15

Privacy and Program Protection

From the time of the earliest computers, programmers have been concerned with controlling who had access to the fruits of their labors. From the same time, users of programs and other programmers have been trying to gain access to programs the writers attempted to protect. It is a game that many computerists play — "if you hide it, I will find it." The levels on which the game is played vary from friendly neighbors to antagonistic nations. The methods used vary from appeals for fair play to elaborate encryption.

As a programmer, there will be times when you want to protect your programs from others. Such occasions will fall into one or both of two general categories. You may wish to prevent a user from easily looking into the program to find the answers to (test) questions or to discover the winning strategy for your newest game. On the other hand, you may want to keep others from discovering how to use a technique you have developed or, even, from making a copy of your program. We will look at both types of protection.

Often, the protection of a program is a legitimate and real need. Be advised, however, that there are no known methods that will guarantee secrecy. That which one can hide, another can find sooner or later.

We cannot get into elaborate protection methods but we will discuss a few simple techniques that will keep out unsophisticated users. Varying degrees of effort and expertise will be required to get around them. They will be sufficient to put up a smoke screen or a barrier of sorts around the things you wish to hide. You will have to take it from there if you wish to go further.

DELETE REMARKS

One of the first rules of programming is to make liberal use of REMark statements to help you remember a few months later how your own program works. The first rule of hiding anything about a program is to take out all REMarks (keeping a copy for yourself, however). Then, anyone unfamiliar with the program will have to spend more time working out the techniques and algorithms.

PACK YOUR PROGRAM

A program in which every statement is on a separate line is one that is easier to figure out. Multistatement lines are harder to follow. Use a ''pack'' program such as the one in Chapter 13, to condense your program as much as possible.

INVISIBLE LINES

When someone examines your program listing, you can throw him or her a few curves by making some of the more important lines invisible. If you change the link in line No. X to refer to line No. X+2 instead of line No. X+1, then X+1 will not list. The program will RUN quite normally as long as that line is not referenced by a GOTO, GOSUB, or THEN.

DISABLE CERTAIN FUNCTIONS

If you disable one or more normal VIC 20 functions, it will cause the curious some trouble. If they can't use the keyboard or STOP the program, for example, they can't get at the program once it starts RUNning. Of course, they may have read these Notes and know exactly what to do. Even so, it will slow them down, at least.

Here are some memory locations with which you can begin. Undoubtedly, you will discover others.

Location 649 holds the size of the keyboard buffer. The value is 10 normally. If your program POKEs 0 here, the keyboard is non-existent to the VIC 20 except for the RUN/STOP key.

Location 775 is a part of the tokens link. The normal value is 199. POKE in 200 to disable the LIST command but allow the program to be re-RUN. If you POKE 198, the LIST command will cause the VIC 20 to lock up tightly. A value of 175 will cause LIST to clear the screen and disable both the RUN and the RUN/STOP RESTORE functions.

Location 808 is a part of the STOP vector and normally holds a value of 112. A value of 114 disables RUN/STOP and 100 disables RUN/STOP RESTORE.

Location 818 is a part of the SAVE link. If the normal value of 133 is replaced with 134, the SAVE command is disabled.

CONCEALED IDENTIFICATION

Writers often begin their programs with several REM statements that contain an identifying name, organization, and address. Certainly, these lines are easy to delete if one wishes to make an unauthorized copy of the program. You may wish to conceal identifying data as well. Such data might be name, initials, and/or serial number.

Data can be concealed within a program if you are willing to do some manipulation in memory. The method depends heavily upon information given in Chapter 13. You will have to refer to that chapter in order to use this technique. We can give only an outline of the steps to take because the details will differ from program to program — you will have to fill in the numbers.

The example we will use is a program containing lines numbered 210, 230, and 240. Assume that there are five characters you wish to hide, though you may use any number.

1. Insert the following lines in the program:

```
220 REMXXXX,XXX
235 REM 12345 < = = =the characters to be hidden
```

2. Look into memory (Chapter 13) to find the link given in line 235 and find the address of the link in line 230.
3. If the high-order bytes in line 230 and 235 are different, go back to step 1 using this line in place of the one given for 220:

   ```
   220 REMXXXX,XX:HXXXX,XXX
   ```

4. Return to the LISTing and replace line 220 with:

   ```
   220 POKEnnnn,mmm
   ```

   (Note that nnnn is the 4-digit address of the low-order link in line 230 and mmm is the 3-digit low-order link found in 235 which is padded with an initial zero if necessary to give 3 digits.)
5. If both link bytes were to be changed (step 3), repeat the substitution in the second-half POKE statement using the high-order byte and its address.

What you are doing here, of course, is to put the line 235 link into line 230 also. The business of line 220 is necessary because SAVing and LOADing the program will undo the hiding if you POKE the numbers from the keyboard. When all is done, line 235 will no longer exist as far as the program and its LISTing are concerned. You will have to PEEK in there to get the hidden information.

Be aware of the fact that you do not have to have line 220 so close to line 235 — it just has to execute before line 235. Further, you do not have to have all those POKE numbers hanging there for everyone to see. You can use variables that have gone through several modifications before they are used, all of which will discourage the decoders.

HIDING WORDS

There are many ways to hide answers or other words in a program. Here are several techniques that will get your thought processes started. In each case the word to be hidden is "AFRICA" and it ends up in the variable B$. Note that in a real program, the statements would not be grouped together and, thus, would be less obvious.

Method 1 — Using ASCII Values for the Letters

```
10 B$ = CHR$(65) + CHR$(70) + CHR$(82) + CHR$(73) + CHR$(67)
   + CHR$(65)
20 PRINT B$
```

Method 2 — Using Data Statements With ASCII − 50 Values

```
10 B$ = " " : FOR N = 1 TO 6 : READ A
20 B$ = B$ + CHR$ (50 + A)
30 NEXT : PRINT B$
40 DATA 15, 20, 32, 23, 17, 15
```

Method 3 — Using Concatenation of Every Second Letter From a Disguised Word

```
10 A$ = "RATFIRMIOCRA"
20 B$ = " " : FOR N = 2 TO LEN (A$) STEP 2
30 B$ = B$ + MID$ (A$, N, 1)
40 NEXT : PRINT B$
```

Method 4 — as Method 3 Except Concatenation in Reverse

```
10 A$ = "RATCLIPRAFTAM"
20 B$ = " " : FOR N = 2 TO LEN (A$) STEP 2
30 B$ = MID$ (A$, N, 1 ) + B$
40 NEXT : PRINT B$
```

Method 5 — as Method 4 Except A$ Is Concatenated From Quite Normal Appearing Words in a DATA Line

```
10 A$ = " " : FOR N = 1 TO 4 : READ T$
20 A$ = A$ + T$ : NEXT
30 B$ = " " : FOR N = 2 TO LEN (A$) STEP 2
40 B$ = MID$ (A$, N, 1) + B$
50 NEXT : PRINT B$
60 DATA RAT, CLIP, RAFT, AM
```

ENCODING

The subject of coding or cryptography is well beyond the scope of this book. There is a whole body of literature on the subject. We will give you a brief sample, however, in case you have an urgent need to protect something to a greater degree right now before you have time to do research in that field.

This technique makes use of the Dual Programs Note explained in Chapter 8. Here are the steps that result in an encoded program:

1. LOAD the plain-text program to be encoded. DO NOT RUN IT!
2. Execute the following line from the keyboard, making a note of the resulting numbers and labeling them W, X, Y, and Z, respectively:

   ```
   FOR N = 43 TO 46 : PRINT PEEK (N) : NEXT
   ```

3. Determine the values of B and E with these formulas using the values for W, X, Y, and Z from step 2:

   ```
   B = W + 256 * X
   E = Y + 256 * Z - 4
   ```

4. Clear the VIC 20 (NEW), type in and SAVE the following program using the previously determined values for B and E:

   ```
   10 FOR N = B TO E
   20 A = PEEK (N)
   30 IF A > 127 THEN 70
   40 IF A > 63 THEN A = A - 64 : GOTO 60
   50 IF A < 64 THEN A = A + 64
   60 POKE N, A
   70 NEXT
   ```

5. Clear the VIC 20 and LOAD (do not RUN) the program to be encoded. Using the appropriate values, execute this line from the keyboard:

   ```
   POKE 43, Y : POKE 44, Z
   ```

6. LOAD and RUN the coding program (from step 4).
7. Execute this line:

   ```
   POKE 43, W : POKE 44, X : POKE 45, Y : POKE 46, Z
   ```

8. Your program is now ready to be SAVEd in its encoded form in the normal manner.

The same coding program (step 4) is used to decode the program. Follow the same steps, though it is not necessary to make another copy of the coding program. On occasion when the encoded program is LOADed, the machine appears to lock up. If this happens to you, just press RUN/STOP - RESTORE and proceed.

You should be aware of several points regarding this coding process. First, complete access is denied anyone who does not have (or figure out) the decoding program. Thus, for anyone to use your work, you must decode it for them or furnish the decoding program. For many uses, this defeats the purpose.

The second point is that you don't have to encode an entire program. Sometimes only a small coded portion will be sufficient. Then, the decoding may be done with a "key" that is hidden away in the main program.

Finally, this coding program is a very simple one. You can change it in many ways to keep out "snoopers."

SUMMARY

We have made only a bare-bones beginning on security of programs and contents. Even so, these approaches will function like locks — they will keep out the honest people. Don't overlook the possibilities in combining some of these methods.

The subject of privacy and program protection is an extensive one. Governments and businesses, large and small, devote a great deal of time and effort to attempts to find unbreakable techniques. At the same time, each one is hard at work trying to break the systems in use by the others. In a way, it is amusing in spite of its seriousness.

CHAPTER 16

Miscellaneous Notes

In assembling Notes for the preceding chapters, there were those that did not fit into the various topics under discussion. They were put aside thinking they would be suitable elsewhere and many were used subsequently. Now, however, we reach the final chapter and still have a number of Notes "left over." Because they will be useful to you, some of them are given here under the heading of miscellaneous — which is no heading at all! So, here they are: a potpourri of Notes.

NONREPEATING SELECTION

Many types of programs include the random selection of an item from a limited list. Perhaps the selection is from a group of questions or of numbers. Usually, you will not want the same item presented in two consecutive passes — or three or four. There are occasions when you will want no item chosen more than once.

Listing

```
.
.
.
```

```
30 DATA ALABAMA, ALASKA, ARIZONA, ARKANSAS, etc.
.
.
.
210 FOR X = 1 TO 50
220 Z = INT (RND (0) * 50) + 1
230 IF Z = R THEN 220
240 R = Z
250 FOR Y = 1 TO Z : READ A$ : NEXT
260 PRINT "WHAT IS THE CAPITAL OF " A$;
270 INPUT B$
.
. (scoring routine here)
.
310 NEXT
.
.
.
```

Fig. 16-1 shows a flowchart for nonrepeating selection.

Analysis

210 sets up a 50-count loop.
220 sets variable Z equal to a random number from 1 to 50.
230 if the value of Z equals the value of R, transfers back to reset Z to another random number.
240 sets R equal to the current value of Z.
250 READS to the Zth item of DATA.
260 PRINTs the question about the chosen item.
270 allows the user to input an answer.
310 continues the loop until conditions are satisfied.

Use

Often you will need to present a series of items in games, tutorials, tests, and many other types of programs. At times, you will want the items used in a fixed order. At others, your purpose would be defeated if they appeared in a fixed order. In the latter cases, this routine will give a varying order and, as a bonus, prevent the duplication of items.

As shown, the routine avoids the presentation of an item that has just been presented. In this example, we are using the names of the states. The "items" can be of any nature, even randomly generated numbers for math problems or letter groups for memory testing.

In this routine, if any selected random value of Z is equal to the preceding value, it is rejected and another random value is

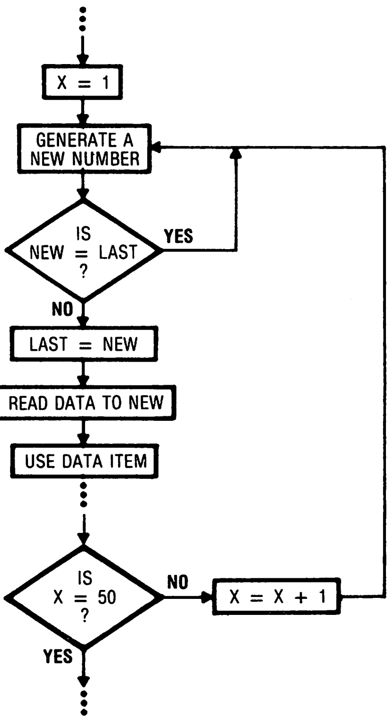


Fig. 16-1. Flowchart for nonrepeating selection.

generated. This will ensure that no two consecutively chosen values are equal. Because the successive values of Z are used to select states from the list, no state will be repeated right after it has been presented.

Variations

1. If you wish to prevent the use of an item that was presented within the last two, make these changes:

```
230 IF Z = R OR Z = S THEN 220
240 R = Z : S = R
```

2. Of course, you could expand the number of items that would not be repeated by expanding on the previous variation. Just add T, U, V, and as many as you like. Very quickly, however, the programming becomes long and tedious and slow-RUNning.

3. You have noticed that the previous routines can prevent the repetition of recent and not-so-recent selections. Unless you go to extreme lengths, there will be repetitions in the selections and some of the items in the list (states) will not be used at all. The following is a good way to disallow all repetitions and, within the 50-count loop, select every state. To do so, make these changes:

```
200 DIM B(50)
230 IF B(Z) THEN 120
240 B(Z) = 1
```

Now, when an item is selected, a corresponding "flag" is set in the B(n) array; e.g., if 10 is selected, the value of B(10) is changed to 1. Of course, all the B(n) values start out as 0 from program initialization. In line 230, B(Z) is tested to see if the value is not zero (that statement is a shorthand way of saying IF B(Z) <> 0 THEN 120). Thus, if 10 ever comes up again in line 220, as you can bet it will, it is rejected because B(10) is not equal to 0.

Each state is selected once and only once. Nothing is free, however. As you noticed when you ran the new routine, the last states were slow to appear. That is because dozens of selections may have been made before an unused one came up. Still, it's better than having the states (or other items) presented in the same order each time the program is RUN.

4. Though shown in DATA statements, the items to be chosen can be in other forms. An array is used frequently: A(1) = "ALABAMA" : A(2) = "ALASKA" : . . . To save program space, a long item list may be read into an array before the selection process begins: FOR X = 1 TO 50 : READ A(X) : NEXT.

THE FUNCTION KEYS

The function keys can be a great asset to you because they can cause virtually any type of action. They can be most helpful in program writing itself in addition to executing complex statements for the program operator. You saw an example of the former in the Build utility program in the section entitled "How To Use This Book."

Listing

```
2000 GET U$ : IF U$ = "" THEN 2000
2010 U = ASC (U$)
```

```
2020 IF U = 133 THEN . . .
2030 IF U = 134 THEN . . .
.
.
.
2090 IF U = 140 THEN . . .
2100 RETURN
```

Analysis

2000 waits until a key has been pressed.
2010 sets U equal to the ASCII value of the key that was pressed.
2020 if the key was F1, takes the specified action.
2030 if the key was F3, takes the specified action.
2090 if the key was F8, takes the specified action.
2100 transfers back to the main program if the key pressed was not a function key.

Use

As you have learned, a function key must be "programmed" or it will have no effect. The ASCII value returned by each key is

F1	133	F5	135
F2	137	F6	139
F3	134	F7	136
F4	138	F8	140

Note that the even-numbered functions are realized by pressing the SHIFT key with the function key.

There is no practical limit to the actions that may be taken when a function key is pressed. Even the types of actions are limited only to those of which the VIC 20 is capable. You can PRINT (the program variables for debugging), SAVE (a program or data), change a program component (variable, data), NEW (erase the entire program), permit special user INPUT, GOTO a special routine, RUN, LIST, RETURN, END and so on. Use your imagination to get the most from the function keys.

Variations

The possible variations are so numerous that we will discuss only a slightly different application for the function keys. You recall that memory location 197 holds a particular value for any key that is pressed. The following lines will cause program execution to pause until the specified function key is pressed:

```
200 PRINT "PRESS F1 TO CONTINUE"
210 IF PEEK (197) <> 39 THEN 210
```

Note that 39 is the "current key" value for F1. The others are:

47 = F3 55 = F5 63 = F7

As with the other keys, the current key values for SHIFTed function keys are the same as the un-SHIFTed values.

SETTING THE ODDS

In some programs, one of two or more possible operations must be performed on the basis of probability or odds. In a game, you may wish to set the odds just like they do on the one-arm bandits (slot machines), so that a player will lose more often than he will win. (Now, would you like to gamble against my program?) In many types of programs, possible courses of action can be chosen by chance (using your odds) to prevent repetition of a fixed sequence that can become predictable or boring or both.

Listing

```
100 FOR Z = 1 TO 100
110 X = RND (0)
120 IF X < .5 THEN 150
130 PRINT "H" ;
140 N = N + 1
150 NEXT
160 PRINT "N = " N
```

Analysis

100 sets up a 100-count loop.
110 sets X equal to a random number between 0 and 1.
120 if the number is less than 0.5, loops to get another number.
130 PRINTs an H (for heads).
140 increments the counter N.
150 continues the loop until the conditions are satisfied.
160 PRINTs the number of "heads."

Use

One of the most frequently encountered demonstrations of probability is a heads/tails coin toss, which is a 50-50 proposition. The odds in dice and cards are almost as well known. With this little programming technique, you can change the probability or odds to make all players equal, to favor the "house" or to suit any particular purpose.

A fixed sequence of reinforcement or admonition graphic displays in an instructional or game program can become tiring. Now you can mix them up as you wish. You will find many applications for this odds-maker, which can be placed in the main program or in a subroutine if it is to be used several times.

The statements actually creating the odds are in lines 110 and 120. The routine shown is suitable for studying how the odds fall (in this case, 50-50) with any given number in line 120.

Variations

1. The LISTing sets the odds at 1 to 1. The number 0.5 in line 120 can be changed to produce other probabilities. For example, use 0.66 for 1 out of 3; 0.33 for 2 out of 3; 0.1 for 9 out of 10 and so on.

2. If a variable is used in place of the number in line 120, you can change the odds at any time during the program RUN by changing the value of the variable. These statements illustrate how to do this and place the odds-maker in a subroutine:

```
.
.
.
110 V = .4 : GOSUB 700
.
.
.
190 V = .7 : GOSUB 700
.
.
.
700 IF RND (0) < V THEN . . .
710 RETURN
```

3. In variation No. 2, the odds can be both variable and unpredictable at the same time. Even you won't be able to predict the odds with this statement:

```
220 V = RND (0) : GOSUB 700
```

INDEFINITE DELAYS

It is a rare program, indeed, that does not use any delay statements. One type of delay takes some given action at the end of a specified length of time. That type was discussed in the Timers and Delays Note in Chapter 8.

Another type of delay is of indefinite length. You want the program to stop executing until the operator takes some action. Such delays are used to allow time for the operator to read instructions, for example. When he is ready to go on, he presses a key and the program continues. Indefinite delays are used to allow time for studying a chart, making a decision, and similar user actions.

We will show you several ways to create an indefinite delay. When you are familiar with them, you can choose the one that best meets the needs of the various programming situations you will encounter.

nnn INPUT W$

This statement is used often. It halts program execution until RETURN is pressed. In this use of INPUT, you are not seeking user input beyond that signal that he is ready to proceed. The variable (W$) is ignored in the following program lines.

nnn GET W$: IF W$ = "" THEN nnn

This indefinite delay is commonly seen in programs. The advantage over using INPUT is that the program proceeds when any key is pressed. The disadvantage is that it requires a separate program line, while INPUT can be placed in the middle of a multistatement line. Both of these methods, however, have a common disadvantage that may or may not be important in a given program — they do use string space.

nnn GET W$: IF W$ <> "K" THEN nnn

You will find this GET statement useful when the operator is to press a specific key. In this example, the program will proceed only when the K-key is pressed.

nnn WAIT 197, 64, 64

It has been said that the use of the WAIT statement is archaic, yet it is often useful. As shown, the statement halts program execution until any key is pressed. It can be placed in the midst of a multistatement line (as well as on a separate line) and it uses no string space. In addition, it is shorter than the GET delay and, so, saves memory. Altogether, this is a very useful indefinite delay statement for most purposes. WAIT can be made to respond to only one key but the statement does become rather cumbersome in this use except for one key.

```
nnn WAIT 653, 1
```

This short WAIT statement halts program execution until a SHIFT key is pressed. Its uses and advantages are the same as those of the previous WAIT statement. Use this one when it is important not to respond to just any key.

FLASHING SCREEN

On many occasions you will want to attract the attention of the user. It may be when he or she must watch something closely or when he or she has made an error. This subroutine will alternately flash the screen colors to let the user know that something special is happening. We consider the flashing screen superior to the alternative of beeping or screeching with sound because the tv/monitor volume may be off but the screen is always on.

Listing

```
20 CZ = 36879
.
.
.
130 GOSUB 800
.
.
.
800 ZY = PEEK (CZ)
810 FOR XX = 2 TO 21
820 ZZ = ZY
830 IF XX / 2 = INT (XX / 2) THEN ZZ = 25
840 POKE CZ, ZZ
850 FOR XY = 1 TO 300 : NEXT
860 NEXT XX
870 RETURN
```

Analysis

20 sets the variable CZ to the screen color address.
130 calls the subroutine.
800 sets the variable ZY to the current value of the screen colors.
810 sets up a 20-count loop.
820 makes ZZ equal to the (color) value of ZY.
830 on even-numbered counts, changes the value of ZZ to 25 (white).

840 POKEs the specified screen colors.
850 delays for a count of 300.
860 continues until the loop specifications are complete.
870 transfers back to the main program.

Use

This subroutine alternately changes the screen colors between white and the original colors. At the end of the sequence, the colors are as they were before the subroutine was called.

In addition to the previously stated use for attracting attention, you can use this technique effectively to heighten the sense of excitement in a game or other type of program.

Variations

1. By changing the constants in lines 810 and 850, you can change the number of flashes and/or the delay between flashes.

2. As written, the alternate flashing color is white. If you prefer another color or, especially if your screen is normally white, you will wish to change the alternate color. This is done by replacing the 25 in line 830 with a number of your choice.

3. You can add sound to this "attention-getter." A short beep routine can be inserted anywhere between lines 810 and 860 to sound each time the screen flashes.

4. Flashing the entire screen is rather drastic. You can flash only the border color by changing these lines:

```
800 ZY = PEEK (CZ) AND 7
830 IF XX / 2 = INT (XX / 2) THEN ZZ = 1
840 POKE CZ, PEEK (CZ) AND 248 OR ZZ
```

Note that the border flashes between white (the 1 in line 830) and the original color.

STANDARD SUBROUTINE PACKAGE

If you do much programming at all, you will find that there are certain subroutines that you use with most of your programs. You can save a great deal of time and effort if you prepare them into a standard subroutine package, SAVE it, and then append it to the ends of programs as they are written. (See the Append Note in Chapter 8.) This Note suggests a package that includes several useful subroutines.

Listing

```
20  S2 = 36875 : V = S2 + 3 : CZ = S2 + 4
.
.
.
900 FOR XX = 1 TO 100 * T : NEXT : RETURN
910 ZY = PEEK (CZ) : FOR XY = 2 TO 11 : ZZ = ZY : IF XY / 2 = INT
    (XY / 2) THEN ZZ = 25
920 POKE CZ, ZZ : T = 2 : GOSUB 910 : NEXT : GOTO 950
930 POKE V, 15 : T = 1 : FOR XY = 1 TO 10 : POKE S2, 240 : GOSUB
    900 : POKE S2, 0 : GOSUB 900 : NEXT : POKE V, 0 : RETURN
940 PRINT " HOME " : FOR XX = 1 TO 21 : PRINT : NEXT : PRINT
    " SPACE = = >PRESS ANY KEY< = = "; : WAIT 197, 64, 64
950 PRINT " SHFT CLR "; : RETURN
```

Analysis

20 sets the sound, volume, and color variables.
900 delays for a count determined by variable T.
910–920 flash the color of the screen and transfer to 950.
930 sounds a series of beeps.
940 PRINTs a message at the bottom of the screen and WAITs for a key to be pressed.
950 clears the screen, homes the cursor, and transfers back to the main program.

Use

This standard subroutine package is composed of several "stand alone" and integrated subroutines. Any line except 920 can be called independently. Of course, they can be called sequentially.

When, modified to suit your particular needs and SAVEd, it is available to be appended to any program you are writing. If you do not need all the subroutines in a package for a given program, append it anyway because it is easier to delete a few lines than to type in all the rest. The saving in time will be significant to you.

Variations

The beauty of a standard subroutine package is that the variations are limitless. You can include just those subroutines that are most useful to you. They may be independent or interlocked (as when one subroutine calls another). Through the use of flags, you can have a great deal of versatility in your package.

APPENDIX A

Useful Memory Locations

Decimal Address	Description
43/44	Pointer to start of BASIC program area
45/46	Pointer to start of variable storage
47/48	Pointer to start of array storage
49/50	Pointer to end of arrays
51/52	Pointer to end of string storage (moves down from top)
55/56	Pointer to top of free RAM (program area)
57/58	Line number being executed
59/60	Previous line number executed
152	Number of files open
160	Mini-clock (18.2 minute increment)
161	Mini-clock (4.2 second increment)
162	Mini-clock (1/60 second increment)
197	Number of the depressed key
198	Number of characters in keyboard buffer (0 clears buffer)
199	Screen reverse flag
201/202	Input cursor (row, column)
203	Number of the depressed key

205	Cursor timing
206	Character at the cursor
209/210	Pointer to screen line
211	Position of cursor on line
214	Cursor row
243/244	Screen color pointer
631–640	Keyboard buffer
646	Color code for printing characters
647	Color at the cursor
648	Screen memory page
649	Size of keyboard buffer (maximum = 10) (set to 0 to disable keyboard)
650	Repeat key action (0 = normal; 100 = disable repeat; 128 = all repeat)
651	Repeat key speed
653	SHIFT control
657	Shift mode switch (0 = enabled; 128 = locked)
774/775	Print tokens link
808/809	STOP vector
818/819	SAVE link
828–1019	Cassette buffer
1024	Begin BASIC area (VIC + 3K)
4096	Begin BASIC area (VIC)
4096	Begin Screen RAM (VIC + 8K and up)
4608	Begin BASIC area (VIC + 8K and up)
7680	Begin Screen RAM (VIC and VIC + 3K)
36872	Location for reading No. 1 paddle
36873	Location for reading No. 2 paddle
37137	Port B
37139	Direction (in/out) of port A
37152	Port A
37154	Direction (in/out) of port B
36864	Horizontal position of display (5 = normal)
36870	Horizontal position of light pen
36871	Vertical position of light pen
36874	Sound generator (low)
36875	Sound generator (mid)
36876	Sound generator (high)
36877	Noise generator
36878	Volume control
36879	Screen and frame color(s)
36881	Vertical position of display (24 = normal)
36883	Visible text lines (46 = normal)
37888	Begin Color RAM (VIC + 8K and up)
38400	Begin Color RAM (VIC and VIC + 3K)

APPENDIX B

BASIC Words, Abbreviations, and Tokens

Section 1. Commands

Brief Meaning	Keyword	Abbreviation	Token
Continue run	CONT	C shift O	154
List program	LIST	L shift I	155
Load program	LOAD	L shift O	147
New memory	NEW		162
Run program	RUN	R shift U	138
Save program	SAVE	S shift A	148
Verify save	VERIFY	V shift E	149

Section 2. Statements

Brief Meaning	Keyword	Abbreviation	Token
Close files	CLOSE	CL shift O	160
Clear variables	CLR	C shift L	156
Command	CMD	C shift M	157
Data	DATA	D shift A	131
Define Function	DEF	D shift E	150
	FN		165
Dimension array	DIM	D shift I	134

End execution	END	E shift N	128
For . . To . . Step	FOR	F shift O	129
	TO		164
	STEP	ST shift E	169
Get from keyboard	GET	G shift E	161
Get# from device	GET#		
Gosub (line #)	GOSUB	GO shift S	141
Goto (line #)	GOTO	G shift O	137
	GO		203
	TO		164
If . . Then	IF		139
	THEN	T shift H	167
Input from keyboard	INPUT		133
Input from device	INPUT#	I shift N	132
Let	LET	L shift E	136
Next (after FOR)	NEXT	N shift E	130
On n GOTO/SUB	ON		145
Open (device)	OPEN	O shift P	159
Poke to memory	POKE	P shift O	151
Print to display	PRINT	?	153
Print to device	PRINT#	P shift R	152
Read from DATA	READ	R shift E	135
Remark	REM		143
Restore DATA pointer	RESTORE	RE shift S	140
Return from GOSUB	RETURN	RE shift T	142
Stop execution	STOP	S shift T	144
System	SYS	S shift Y	158
Wait for change	WAIT	W shift A	146

Section 3. Operators

Brief Meaning	Keyword	Abbreviation	Token
Add	+		170
Subtract	−		171
Multiply	*		172
Divide	/		173
Raise to power	↑		174
Greater than	>		177
Equal	=		178
Less than	<		179
And (logical)	AND	A shift N	175
Or (logical)	OR		176
Not (logical)	NOT	N shift O	168

Section 4. Functions

Brief Meaning	Keyword	Abbreviation	Token
Absolute value	ABS	A shift B	182
Arctangent	ATN	A shift T	193
Cosine	COS		190
Exponent value	EXP	E shift X	189
Function value	FN		165
Integer value	INT		181
Logarithm (natural)	LOG		188
Peek at mem	PEEK	P shift E	194
Random number	RND	R shift N	187
Sign of number	SGN	S shift G	180
Sine of angle	SIN	S shift I	191
Square root	SQR	S shift Q	186
Tangent	TAN		192
User machine jump	USR	U shift S	183
ASCII value	ASC	A shift S	198
Character (ASC)	CHR$	C shift H	199
Left of string	LEFT$	LE shift F	200
Length of string	LEN		195
Middle of string	MID$	M shift I	202
Right of string	RIGHT$	R shift I	201
Number to string	STR$	ST shift R	196
String to number	VAL	V shift A	197
Memory left	FRE	F shift R	184
Column number	POS		185
Skip spaces	SPC	S shift P	166
Tabulate	TAB	T shift A	163

APPENDIX C

Screen RAM Map

This chart will help you find the memory address of any location on the 22-character by 23-line screen. The first number of each pair is the address of the adjacent block in a VIC 20 with no added memory. The second number is the appropriate address when 8K or more memory is added.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
7680/4096																							7701/4117
7702/4118																							7723/4139
7724/4140																							7745/4161
7746/4162																							7767/4183
7768/4184																							7789/4205
7790/4206																							7811/4227
7812/4228																							7833/4249
7834/4250																							7855/4271
7856/4272																							7877/4293
7878/4294																							7899/4315
7900/4316																							7921/4337
7922/4338																							7943/4359
7944/4360																							7965/4381
7966/4382																							7987/4403
7988/4404																							8009/4425
8010/4426																							8031/4447
8032/4448																							8053/4469
8054/4470																							8075/4491
8076/4492																							8097/4513
8098/4514																							8119/4535
8120/4536																							8141/4557
8142/4558																							8163/4579
8164/4580																							8185/4601

↑ ↑
+0K +8K
+3K UP

↑ ↑
+0K +8K
+3K UP

APPENDIX D

Display Screen Codes

When the codes in this Appendix are poked into Screen RAM, the letter/symbol/graphic shown in either of the two Set columns will appear on the screen. To shift between "Set A" and "Set a" you may

1. press the **SHIFT** and Commodore symbol keys together or
2. poke 240 or 242 into address 36869.

Set A	Set a	Code	Set A	Set a	Code	Set A	Set a	Code
@		0	J	j	10	T	t	20
A	a	1	K	k	11	U	u	21
B	b	2	L	l	12	V	v	22
C	c	3	M	m	13	W	w	23
D	d	4	N	n	14	X	x	24
E	e	5	O	o	15	Y	y	25
F	f	6	P	p	16	Z	z	26
G	g	7	Q	q	17	[		27
H	h	8	R	r	18	£		28
I	i	9	S	s	19	]		29

Set A	Set a	Code	Set A	Set a	Code	Set A	Set a	Code
↑		30		?	63	SPACE		96
←		31	⊟		64	▌		97
	SPACE	32	♠	A	65	▄		98
	!	33	│	B	66	▔		99
	"	34	─	C	67	▁		100
	#	35	▔	D	68	▏		101
	$	36	▔	E	69	▒		102
	%	37	▁	F	70	▕		103
	&	38	▏	G	71	▄		104
	'	39	▕	H	72	◤	▨	105
	(	40	╮	I	73	▕		106
	)	41	╰	J	74	├		107
	*	42	╯	K	75	▗		108
	+	43	└	L	76	└		109
	,	44	╲	M	77	┐		110
	−	45	╱	N	78	▁		111
	.	46	┏	O	79	┌		112
	/	47	┓	P	80	┴		113
	0	48	●	Q	81	┬		114
	1	49	▁	R	82	┤		115
	2	50	♥	S	83	▏		116
	3	51	▏	T	84	▌		117
	4	52	╭	U	85	▐		118
	5	53	╳	V	86	▔		119
	6	54	○	W	87	▀		120
	7	55	♣	X	88	▄		121
	8	56	▕	Y	89	▁	✓	122
	9	57	♦	Z	90	▖		123
	:	58	┼		91	▝		124
	;	59	▒		92	┘		125
	<	60	│		93	▘		126
	=	61	π	▒	94	▚		127
	>	62	◥	▧	95			

Note 1: When only one character is shown, that character is the same for each Set.

Note 2: Adding 128 to the above codes will produce reverse characters.

APPENDIX E

Color RAM Map

This chart will help you find the memory address of any color location on the 22-character by 23-line screen. The similarity to the Screen RAM Map is expected because each is based on the same screen display. The first number of each pair is the address of the adjacent block in a VIC 20 with no added memory. The second number is the address for a VIC 20 with 8K or more of added memory.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
38400/37888																							38421/37909
38422/37910																							38443/37931
38444/37932																							38465/37953
38466/37954																							38487/37975
38488/37976																							38509/37997
38510/37998																							38531/38019
38532/38020																							38553/38041
38554/38042																							38575/38063
38576/38064																							38597/38085
38598/38086																							38619/38107
38620/38108																							38641/38129
38642/38130																							38663/38151
38664/38152																							38685/38173
38686/38174																							38707/38195
38708/38196																							38729/38217
38730/38218																							38751/38239
38752/38240																							38773/38261
38774/38262																							38795/38283
38796/38284																							38817/38305
38818/38306																							38839/38327
38840/38328																							38861/38349
38862/38350																							38883/38371
38884/38372																							38905/38393

↑ ↑

+0K +8K

+3K UP

↑ ↑

+0K +8K

+3K UP

APPENDIX F

Color Tables

COLOR CODE TABLE

This table contains the colors that can be displayed. Characters can have only colors 0 through 7. See Chapter 9 for information on using these codes.

0 BLACK	4 PURPLE	8 ORANGE	12 LT. PURPLE
1 WHITE	5 GREEN	9 LT. ORANGE	13 LT. GREEN
2 RED	6 BLUE	10 PINK	14 LT. BLUE
3 CYAN	7 YELLOW	11 LT. CYAN	15 LT. YELLOW

TABLE OF COLORS FOR SCREEN AND BORDER

The numbers from this table determine the border and screen colors as indicated. They are used in the statement

POKE 36879, value

If you POKE a number 8 less than that shown in the table, the REVERSE will be turned ON.

	Border							
Screen	**BLK**	**WHT**	**RED**	**CYN**	**PUR**	**GRN**	**BLU**	**YEL**
BLACK	8	9	10	11	12	13	14	15
WHITE	24	25	26	27	28	29	30	31
RED	40	41	42	43	44	45	46	47
CYAN	56	57	58	59	60	61	62	63
PURPLE	72	73	74	75	76	77	78	79
GREEN	88	89	90	91	92	93	94	95
BLUE	104	105	106	107	108	109	110	111
YELLOW	120	121	122	123	124	125	126	127
ORANGE	136	137	138	139	140	141	142	143
LT ORN	152	153	154	155	156	157	158	159
PINK	168	169	170	171	172	173	174	175
LT CYN	184	185	186	187	188	189	190	191
LT PUR	200	201	202	203	204	205	206	207
LT GRN	216	217	218	219	220	221	222	223
LT BLU	232	233	234	235	236	237	238	239
LT YEL	248	249	250	251	252	253	254	255

APPENDIX G

CHR$ and ASCII Codes

This chart shows the characters corresponding to the numbers from 0 to 191 (the others are repeats). The ASCII values are the same as those for CHR$.

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0	SWITCH TO LOWER CASE	14		25	&	38
	1		15		26	.	39
	2		16		27	(	40
	3	CRSR ↓	17	RED	28	)	41
	4	RVS ON	18	CRSR →	29	*	42
WHT	5	CLR HOME	19	GRN	30	+	43
	6	INST DEL	20	BLU	31	,	44
	7		21	SPACE	32	–	45
	8		22	!	33	.	46
	9		23	''	34	/	47
	10		24	#	35	0	48
	11			$	36	1	49
	12			%	37	2	50
RETURN	13						

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$		
3	51	[	91		131	▕	167
4	52	£	92		132	▄	168
5	53	]	93	f1	133	◤	169
6	54	↑	94	f3	134	▐	170
7	55	←	95	f5	135	├	171
8	56	─	96	f7	136	▗	172
9	57	♠	97	f2	137	└	173
:	58	│	98	f4	138	┐	174
;	59	─	99	f6	139	▁	175
<	60	▔	100	f8	140	┌	176
=	61	▔	101	SHIFT RETURN	141	┴	177
>	62	▁	102	SWITCH TO UPPER CASE	142	┬	178
?	63	▏	103		143	┤	179
@	64	▕	104	BLK	144	▎	180
A	65	╮	105	↑ CRSR	145	▍	181
B	66	╰	106	RVS OFF	146	▐	182
C	67	╯	107	CLR HOME	147	▔	183
D	68	└	108	INST DEL	148	▀	184
E	69	╲	109		149	▄	185
F	70	╱	110		150	▕	186
G	71	┌	111		151	▖	187
H	72	┐	112		152	▝	188
I	73	●	113		153	┘	189
J	74	▁	114		154	▘	190
K	75	♥	115		155	▚	191
L	76	▏	116	PUR	156		
M	77	╭	117	← CRSR	157		
N	78	╳	118	YEL	158		
O	79	○	119	CYN	159		
P	80	♣	120	SPACE	160		
Q	81	▕	121	▌	161		
R	82	♦	122	▄	162		
S	83	┼	123	▔	163		
T	84	▒	124	▁	164		
U	85	│	125	▏	165		
V	86	π	126	▒	166		
W	87	◥	127				
X	88		128				
Y	89		129				
Z	90		130				

APPENDIX H

Decimal to Binary and Binary to Decimal Conversion

Dec	Binary	Dec	Binary	Dec	Binary	Dec	Binary
0	00000000	64	01000000	128	10000000	192	11000000
1	00000001	65	01000001	129	10000001	193	11000001
2	00000010	66	01000010	130	10000010	194	11000010
3	00000011	67	01000011	131	10000011	195	11000011
4	00000100	68	01000100	132	10000100	196	11000100
5	00000101	69	01000101	133	10000101	197	11000101
6	00000110	70	01000110	134	10000110	198	11000110
7	00000111	71	01000111	135	10000111	199	11000111
8	00001000	72	01001000	136	10001000	200	11001000
9	00001001	73	01001001	137	10001001	201	11001001
10	00001010	74	01001010	138	10001010	202	11001010
11	00001011	75	01001011	139	10001011	203	11001011
12	00001100	76	01001100	140	10001100	204	11001100
13	00001101	77	01001101	141	10001101	205	11001101
14	00001110	78	01001110	142	10001110	206	11001110
15	00001111	79	01001111	143	10001111	207	11001111
16	00010000	80	01010000	144	10010000	208	11010000
17	00010001	81	01010001	145	10010001	209	11010001
18	00010010	82	01010010	146	10010010	210	11010010
19	00010011	83	01010011	147	10010011	211	11010011

20	00010100	84	01010100	148	10010100	212	11010100
21	00010101	85	01010101	149	10010101	213	11010101
22	00010110	86	01010110	150	10010110	214	11010110
23	00010111	87	01010111	151	10010111	215	11010111
24	00011000	88	01011000	152	10011000	216	11011000
25	00011001	89	01011001	153	10011001	217	11011001
26	00011010	90	01011010	154	10011010	218	11011010
27	00011011	91	01011011	155	10011011	219	11011011
28	00011100	92	01011100	156	10011100	220	11011100
29	00011101	93	01011101	157	10011101	221	11011101
30	00011110	94	01011110	158	10011110	222	11011110
31	00011111	95	01011111	159	10011111	223	11011111
32	00100000	96	01100000	160	10100000	224	11100000
33	00100001	97	01100001	161	10100001	225	11100001
34	00100010	98	01100010	162	10100010	226	11100010
35	00100011	99	01100011	163	10100011	227	11100011
36	00100100	100	01100100	164	10100100	228	11100100
37	00100101	101	01100101	165	10100101	229	11100101
38	00100110	102	01100110	166	10100110	230	11100110
39	00100111	103	01100111	167	10100111	231	11100111
40	00101000	104	01101000	168	10101000	232	11101000
41	00101001	105	01101001	169	10101001	233	11101001
42	00101010	106	01101010	170	10101010	234	11101010
43	00101011	107	01101011	171	10101011	235	11101011
44	00101100	108	01101100	172	10101100	236	11101100
45	00101101	109	01101101	173	10101101	237	11101101
46	00101110	110	01101110	174	10101110	238	11101110
47	00101111	111	01101111	175	10101111	239	11101111
48	00110000	112	01110000	176	10110000	240	11110000
49	00110001	113	01110001	177	10110001	241	11110001
50	00110010	114	01110010	178	10110010	242	11110010
51	00110011	115	01110011	179	10110011	243	11110011
52	00110100	116	01110100	180	10110100	244	11110100
53	00110101	117	01110101	181	10110101	245	11110101
54	00110110	118	01110110	182	10110110	246	11110110
55	00110111	119	01110111	183	10110111	247	11110111
56	00111000	120	01111000	184	10111000	248	11111000
57	00111001	121	01111001	185	10111001	249	11111001
58	00111010	122	01111010	186	10111010	250	11111010
59	00111011	123	01111011	187	10111011	251	11111011
60	00111100	124	01111100	188	10111100	252	11111100
61	00111101	125	01111101	189	10111101	253	11111101
62	00111110	126	01111110	190	10111110	254	11111110
63	00111111	127	01111111	191	10111111	255	11111111

Index

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials that will permit students, engineers, scientists, technicians and others to quickly learn how to use new technologies and electronic techniques. We continue to do this through several means, textbooks, short courses, seminars and through the development of special electronic devices and training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

–The design and development of what is considered to be the first popular hobbyist computer. The Mark-B was featured in Radio-Electronics magazine in 1974. We have also designed several 8080-based computers, including the MMD-1 system. Our most recent computer is an 8085-based computer for educational use, and for use in small controllers.

—The Blacksburg Continuing Education Series™ covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

–We have pioneered the use of small, self-contained computers in hands-on courses for microcomputer users. Many of our designs have evolved into commercial products that are marketed by E&L Instruments and PACCOM, and are available from Group Technology, Ltd., Check, VA 24072.

–Our short courses and seminar programs have been presented throughout the world. Programs are offered by The Blacksburg Group, and by the Virginia Polytechnic Institute Extension Division. Each series of courses provides hands-on experience with real computers and electronic devices. Courses and seminars are provided on a regular basis, and are also provided for groups, companies and schools at a site of their choosing. We are strong believers in practical laboratory exercises, so much time is spent working with electronic equipment, computers and circuits.

Additional information may be obtained from Dr. Chris Titus, the Blacksburg Group, Inc. (703) 951-9030 or from Dr. Linda Leffel, Virginia Tech Continuing Education Center (703) 961-5241.

Our group members are Mr. David G. Larsen, who is on the faculty of the Department of Chemistry at Virginia Tech, and Drs. Jon Titus and Chris Titus who work full-time with The Blacksburg Group, all of Blacksburg, VA.

SAMS

VIC 20 Programmer's Notebook

- Has a collection of programming techniques, hints and kinks, tricks, subroutines, and shortcuts.
- Explains procedures for adding memory and error trapping.
- Proclaims that KERNAL is the officer in charge of the VIC 20.
- States that two of the most powerful commands are PEEK and POKE.
- Discusses the main functions for handling data: ASC, CHR$, LEFT$, LEN, MID$, RIGHT$, STR$, VAL, concatenation, parsing data, finding buried data, bubble sort, and Shell-Metzner sort.
- Explains methods of combining graphics, color, and sound.
- Covers joystick, paddles, and light pen.
- Discusses privacy and program protection.

This book is equally useful to the beginner and experienced VIC 20 programmer.

Earl R. Savage graduated magna cum laude from Hampden Sydney College with a B.S. degree. He went on to receive a M.Ed. degree from the University of Virginia, and studied further at the University of Chicago, College of William and Mary, and Old Dominion University. While in the U.S. Air Force, he received training in Electronic Countermeasures.

Mr. Savage's experience has included being a science teacher and science supervisor, a university instructor, a principal of both elementary and secondary schools, and a Superintendent of Schools. As the Hobby Editor for **Radio-Electronics,** a position he has held since 1977, he has written several articles and a monthly column, the "Hobby Corner." He also writes the monthly column "Education 80" in **80 Microcomputing**. In addition, he has written articles for many technical and educational magazines and house organs. He is the author of the SAMS book **BASIC Programmer's Notebook.**

Although Mr. Savage includes electronic activities in his hobbies of ham radio (K4SDS), electronics, and microcomputer programming, he also plays chess and bridge, and enjoys photography.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$14.95/22089 ISBN: 0-672-22089-X