DATAMOST

$14.95

INTERMEDIATE COMMODORE 64

by Guy Grotke

Take that intermediate step from elementary BASIC to machine language programming!

Intermediate Commodore 64

Intermediate Commodore 64

by
Guy Grotke

Illustrated by
Robert J. Peters


DATAMOST™

20660 Nordhoff Street
Chatsworth, CA 91311-6152
(818) 709-1202

DATAMOST™

ISBN 0-88190-253-5

Printed in U.S.A.

Acknowledgements

I would like to take this opportunity to thank Bill Sanders for asking me to write this book as the sequel to his book, *The Elementary Commodore 64*, for editing the manuscript, and for his valuable comments and suggestions.

I must also thank Commodore, first for building and mass marketing such an interesting computer, and second for loaning me a system for so long.

Finally, I want to thank all of my relatives and friends, and especially Dawn, my best friend, for months of patience.

Table of Contents

Editor's Introduction

William B. Sanders, Ph.D.

There comes a time when a programmer has mastered the elementary level of programming. He or she knows how to use the fundamental BASIC statements and starts wondering about POKE, PEEK, SYS and other statements that seem to go into the very heart of the computer. Various aspects of their computer come to light that are also beyond the scope of introductory books, such as enhanced graphics and disk operations. They have heard about machine and assembly language, but jumping into these complex, lower level areas of programming is a bigger step than they want or need at this point. This book is intended for those persons who are beyond the elementary level but do not want to jump the chasm to the advanced level without first taking an intermediate step.

Intermediate Commodore 64 begins by showing you how to write better BASIC programs. Like most non-professional programmers, beginners just want their programs to execute. If the program runs, then that's all that counts. However, if you could spend less time creating programs, have them execute faster, have them execute more often, and take up less memory, you'd not only be a better programmer, you would be less frustrated and have more fun with your Commodore 64. You won't become a professional programmer, but you will be a better programmer.

After showing you program structures and how to develop algorithms, *Intermediate Commodore 64* then shows you how to work with your disk files. It goes beyond creating simple sequential files, delving into all the nooks and crannies of your disk system. You will be able to do far more than you thought possible

using BASIC and, with the help of some simple machine language subroutines, how to do even more. Machine language is handled in an easy, straightforward way. You will be shown how to create simple routines and how to access built-in subroutines in your Commodore 64. All of this can be done from BASIC, but you are also shown some steps in using an assembler to simplify creating machine code.

If you like graphics, the next part of the book will give you incredible flexibility with visual effects. Besides the elementary keyboard and sprite graphics you may already know, you will see how to change the appearance of the character set, create high resolution drawings, and do things with sprites that are truly amazing. The special graphics chip in your Commodore 64 is revealed in all of its utility.

Finally, the last chapter shows you how BASIC actually works. Not only can you see how BASIC is executed inside the machine, but you can actually rewrite BASIC to meet your requirements. For example, you can create BASIC statements that will handle graphics functions that would otherwise require several POKE statements.

About the Author

The author, Guy Grotke, is an experienced programmer who has created many professional software packages and hardware devices for computers. His knowledge of programming and computers is complete, and, as you will see with this book, he is also an excellent writer. The material is presented in a clear, straightforward manner and there are plenty of examples. Just reading the manuscript, I learned more about the Commodore 64 than I thought there was to learn, and I have no doubt you will find the same to be true. Guy has programmed in most of the major programming languages and has even written a version of FORTH for microcomputers. He is currently the technical director of a company writing software for medical applications.

Preface

This book is titled *Intermediate Commodore 64* because it is part of a series. If you have not already mastered the introductory concepts covered in *The Elementary Commodore 64*, you may have some difficulty following along. General BASIC commands and programming are not covered by this book. Instead, each chapter has been written in greater depth on a specific topic. Examples are given to illustrate methods, not just routines and programs to be entered and used by rote. By reading each chapter and then incorporating the information and techniques into your programs, you can advance well into the intermediate level of programming the Commodore 64.

Chapter 1 covers structured programming and organization techniques that can help you write complex programs without getting lost in the details. These methods are considered from a practical sense, and are not idealized or excessively theoretical. Programming aids discussed include top-down design, control and flow diagrams and variable charts.

Chapter 2 deals with algorithms. This chapter should take some of the mystery out of choosing the correct algorithms for your programs. Making the right choice can mean that a program will run much faster or use much less memory than it would with another algorithm.

Chapter 3 is about the Commodore 64 disk file system. Both sequential and random access data files are explained and used in example programs. Some of the peculiarities of the disk commands are shown, along with details on BASIC and machine language program files.

Chapter 4 explains the use of machine language programs and subroutines. This is not a tutorial on 6510 microprocessor programming, but reasons for using machine code and the techniques available for use with the Commodore 64 are explained.

Subjects include modifying the behavior of vectored routines and a form of multi-tasking through the use of the Interrupt Request (IRQ) routine.

Chapters 5, 6 and 7 are all about video graphics and the VIC integrated circuit. These three chapters are devoted to graphics, and for a good reason: The Commodore 64 has some of the most advanced video generation hardware of any home computer sold today. The sophisticated hardware opens many graphics techniques to the intermediate level programmer that require expertise and tedious labor on other computers. Topics include character and bit-mapped modes, redefined character sets, sprites, missiles, sprite animation and real-time multiplexing.

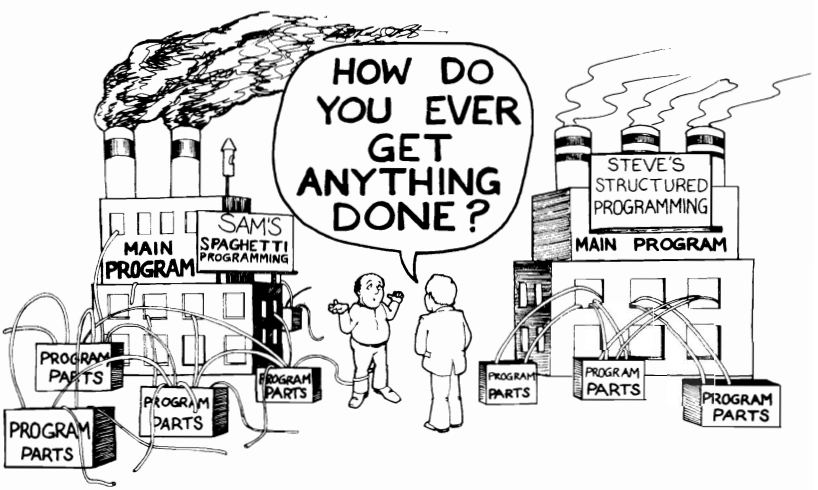
Finally, Chapter 8 contains information about the way BASIC stores program lines and data structures. The trade-offs between increasing program execution speed and decreasing program memory use are considered. There are also discussions of advanced techniques such as passing parameters to machine code routines, extending the BASIC interpreter with new commands, and writing machine code routines that interpret BASIC program lines.

Chapter 1

Structured Programming

Structured programming concerns more than just the physical layout of program lines. It consists of planning, logical design, coding with careful attention to good practices, and full documentation of a program. If this chapter is the only part of this book that you incorporate into your programs, you will have made a terrific investment. By using structured programming techniques, you will make your programs:

1. Readable
2. Portable
3. Correct
4. Easy to maintain



Why Bother?

Readability means that your program will be easier for you to work with, both as you write it and, months later, when you need to fix a problem.

Portability means that another programmer will be able to understand what your program does, so that he or she could move it to another type of computer. This especially helps make your program suitable for publication.

Correctness means that your program will probably work the first time that you RUN it. By following these procedures you won't spend a lot of effort rewriting or renumbering your program should you find major conceptual errors. The first draft of a program may not be the fastest possible version, but it is much easier to speed up a slow-running program than to patch up a program that doesn't run at all.

Ease of maintenance means that fixing problems or adding features at a later date will be accomplished in minutes instead of hours. Many BASIC programs are designed so poorly that even their authors call them "fragile" and are afraid to try to fix bugs for fear of creating new ones. Even some expensive commercial programs have this problem.

Writing a Readable Program

The first step in writing a program is to decide exactly what the program is to do. This is called "specification." There are many formal systems of specification used by large companies that contract for programs, but for our needs we can just follow this checklist:

1. What are the data inputs?
2. What is done with the data?
3. How does the user interact?
4. How are errors handled?
5. What are the variables needed?
6. What is output from the program?

Each of these items deserves careful consideration and words on paper. I can't stress this enough. I spent an entire year writing and rewriting a single large program because there wasn't a written specification. For the next program I spent three weeks writing a specification and four weeks writing the program. The second program was even larger than the first, but everyone was happy with it the first time.

You may not be able to answer all of the questions very well before designing the program, but just trying will help you solidify ideas about how the program will work.

Divide and Conquer

The next step is to divide the program into five to 10 logical parts. If your program is very simple, there may not be five parts, but usually the opposite case is true. It is important not to divide it into more than 10 parts at this point. If you can't see 10 or fewer logical divisions, think some more about the similarities of the different parts of your program. Look for functions that will be used several times within the program. Once you have divided the program, examine each section and ask yourself: "Can this section be divided further?" If it makes sense to do so, then divide that section again into five to 10 parts.

As you divide the program into more and more logical sections, each "leaf of the tree" will become a trivial programming exercise. This entire process is called "top-down design." Using top-down design will prevent you from programming your way into a corner.

You may use flowcharts or any other type of diagram to assist you as you break your program up into reasonable chunks. You don't need to buy a flowchart template or special paper unless you are compulsively neat. You can write excellent flowcharts with only three symbols.

Symbol	Meaning
Box	Action of program
Diamond	Decision point
Circle	Connector to other circle

Here is a small flowchart of a game program which uses each of these.

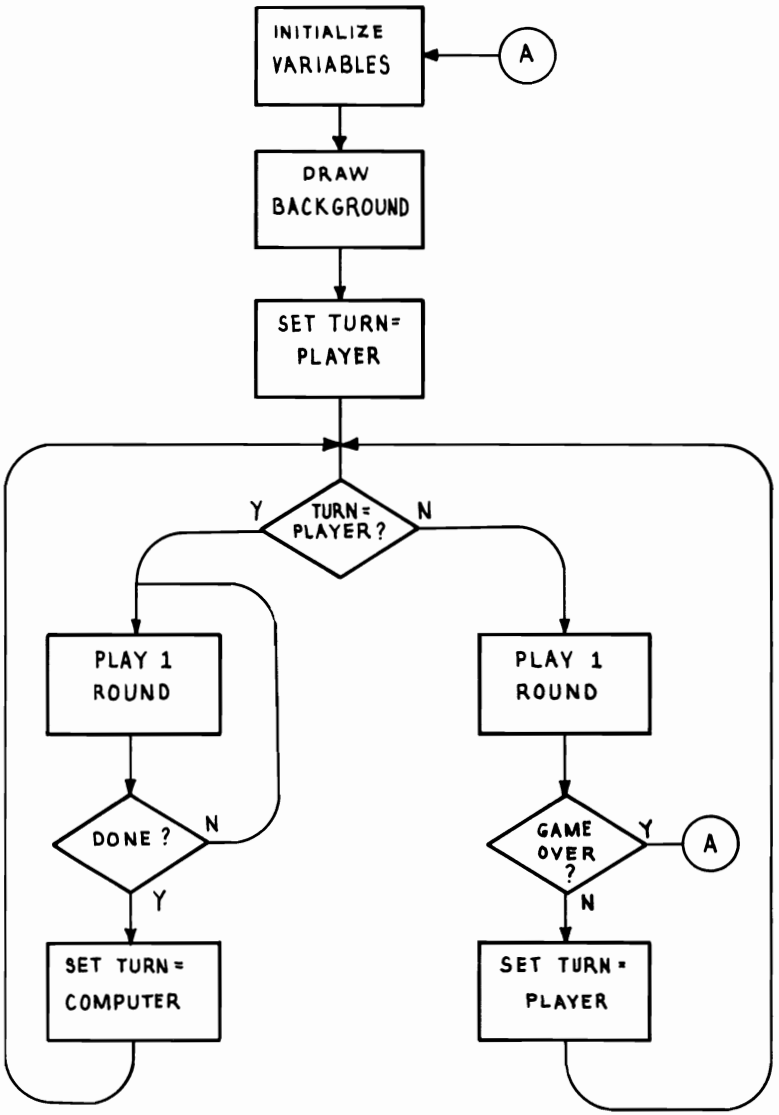


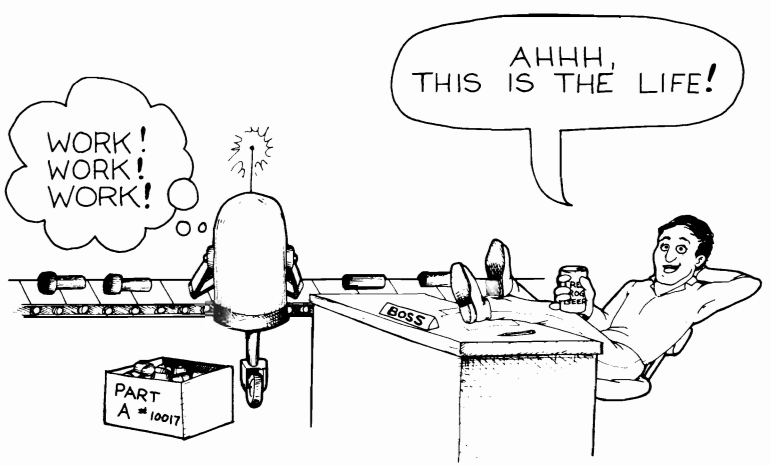
Figure 1.1: A Sample Flowchart

There is another type of chart that is useful to BASIC programmers, and that is the variable chart. Since all variables and arrays in BASIC are global, each may be changed by any part of the program. In many other languages variables may be made local to individual parts of the program. This means that the same variable name could be used by more than one part of the program without interference. Unfortunately, this is not the case with the BASIC in the Commodore 64.

If you keep a list of variable names as you write lines of BASIC, you will avoid the common trap of reusing a variable name accidentally. Of course you may want to reuse a variable in program sections that don't interfere, but by keeping track you'll never do it unintentionally.

NAME	TYPE	SET BY	USED BY	MEANING
A	Floating	20, 50, 250	30, 60, 70	Address pointer
SC	Floating	300	1010	Game score
D%	Integer	120, 240	390	Difficulty factor
G$	String	10	200	Name of game
Z(100, 10)	Floating Point Array	2010-2090	180, 190 220, 240	Object space coordinates
R$	String	40, 80, 250	50, 90, 206	User reply to a question

Figure 1.2: A Sample Variable Chart



Translating Flowcharts into BASIC Lines

Most of the boxes on your flowchart will translate into one or perhaps a few BASIC lines. A word is in order here about BASIC's ability to hold multiple statements on one line. The word is NO, don't use it! On some computers putting more than one statement per line decreases the execution time by sacrificing any chance of readability. This is not the case with the Commodore 64. Interpreting those extra statements without line numbers does not seem to improve the execution time to any measurable extent. The only time that you should put more than one statement on a line is when it improves the readability of the program. For example:

```
10 PRINT:PRINT:PRINT
```

is perfectly clear and acceptable. Also, since the IF . . . THEN construct conditionally executes only the remainder of the line, it is sometimes clearer to add statements rather than add branches to meaningless line numbers.

```
40 IF SUM>200 THEN FLAG=TRUE:X=1
50 PRINT X
```

is better than

```
40 IF SUM<=200 THEN 70
50 FLAG=TRUE
60 X=1
70 PRINT X
```

Controlling Program Flow

There are several commands in Commodore 64 BASIC for controlling the order of execution of program lines. The two commands END and STOP act in a similar manner, except that STOP tells you where the program actually stopped. After either of these have been executed, you can restart the program at the next line by typing CONT. END corresponds to the boxes on your flowchart that contain "END OF PROGRAM". Any pro-

gram may have several logical ending places, so there may be several END statements. One common structured programming concept is that every program should have only a single END. To implement this in BASIC, you may use the GOTO command to jump to the END statement. The advantage of this practice may not be apparent now, but what if you decide later that certain actions need to be performed before the program stops running? If there is only one END statement then you could add the needed lines at that point and know that they would ALWAYS be executed before the program stopped running. On the other hand, if you have several END statements, you would have to find and replace them all with the needed lines or GOTO statements.

ENDs throughout:

```
100 IF A$ = "" THEN END
110 C = B
120 PRINT "THANK YOU"
130 INPUT "ENTER NEXT NUMBER",N
140 IF N = 0 THEN END
150 GOSUB 1000
160 END
```

A single END:

```
100 IF A$ = "" THEN 10000: REM END
110 C = B
120 PRINT "THANK YOU"
130 INPUT "ENTER NEXT NUMBER",N
140 IF N = 0 THEN 10000: REM END
150 GOSUB 1000
160 GOTO 10000: REM END
10000 CLOSE 4
10010 CLOSE 5
10020 INPUT "RUN AGAIN? (Y/N)  ;R$
10030 IF LEFT$(R$,1) = "Y" THEN 10
10040 END
```

The STOP command is very useful for program debugging since it prints its line number when it is encountered. If you write lines or whole routines that you're not sure of (as I do), you can check on what is going on by adding a few STOPs. After each STOP you can print variables or strings and then tell BASIC to CONTinue. Once a program is running well, all of the STOP commands should be removed or changed to GOTOs directing the program flow to an error handling routine.

What IF?

The IF . . . THEN command in Commodore 64 BASIC permits conditional execution of other BASIC commands. The correspondence between IF . . . THEN and the decision diamonds of your flowchart should be obvious. Most computer languages have some sort of an IF . . . THEN command. Some of these languages, including advanced BASICs, also permit IF . . . THEN . . . ELSE, but not Commodore 64 BASIC.

If the condition is met, the commands after THEN will be executed, until the end of the line. On the Commodore 64 a line may contain 80 characters, so quite a few statements may be on that one line.

```
50 IF A = 1 THEN B = 24:C = 35:PRINT "A NOW
   EQUALS ONE.":PRINT
60 PRINT
70 D$ = "LAST ONE"
```

If this seems difficult to read, or if all of the necessary statements would not fit on the single line, then it could be written instead as:

```
50 IF A <> 1 THEN 100 : REM SKIP
60 B = 24
70 C = 35
80 PRINT "A NOW EQUALS ONE."
90 PRINT
100 PRINT
110 D$ = "LAST ONE"
```

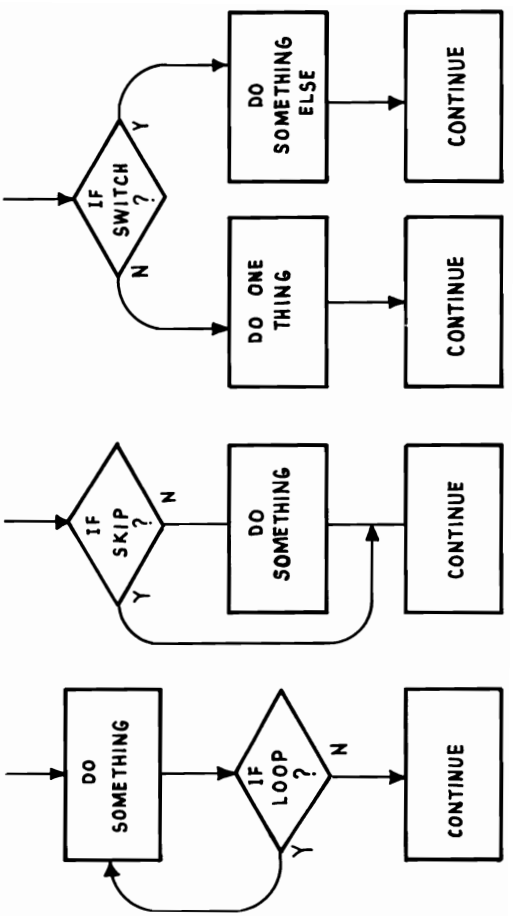


Figure 1.3: IF . . . THEN Command

Note that the conditional test (< >) replaced the former test (=) in line 50. This type of "inverted logic" is commonly needed when the conditional action is changed to a line number in order to skip following statement lines. You should remember that the opposite of < is >= and not just >. Likewise, the opposite of > is <=.

If you flowchart a program which will display a menu of choices, you probably found that you needed a "multi-way branch." In some languages there is a CASE statement for this very purpose. In Commodore 64 BASIC you may write a multi-way branch by using several IF . . . THENs:

```
10 PRINT "A . . . ADD A TRANSACTION"
20 PRINT "R . . . REMOVE A TRANSACTION"
30 PRINT "E . . . END SESSION"
40 PRINT
50 INPUT R$ : REM GET MENU CHOICE
60 IF R$ = "A" THEN 1000
70 IF R$ = "R" THEN 2000
80 IF R$ = "E" THEN 9000
90 PRINT "PLEASE CHOOSE AGAIN"
100 GOTO 10
```

The program lines above give the user three choices, and only three choices! Planning for unknowledgable or even malicious use is very important when you are trying to write a professional quality program. Compare this to:

```
10 INPUT R$ : REM NO PROMPT!
20 IF R$ = "A" THEN 1000
30 IF R$ = "R" THEN 2000
40 GOTO 9000 : REM TOUGH LUCK!
```

If the user accidentally misses the "A" or "R" keys, the program may end even though that might be a big mistake.

There is another way to write multi-way branches, and that is the ON statement. Personally, I have found the ON statement to be so restrictive that it's use tends to force you to write an unfriendly user interface:

```
10 PRINT "ENTER PROCEDURE NUMBER"
20 INPUT P : REM GET NUMBER CHOICE
30 ON P GOTO 1000,2000,9000
40 PRINT "1, 2, OR 3 PLEASE."
50 GOTO 10
```

If the user enters a 1, the GOTO will jump to line 1000. If he enters a 2, 2000; a 3, 3000. If he chooses any other number from 0 to 255, line 40 will be executed. An error will result if the number is negative or greater than 255.

You may find use for ON . . . GOTO or ON . . . GOSUB in programs when the index variable is one internal to the program rather than the result of a user input. It is quite common to have a "mode" variable which has values suitable for this type of branch. If there are illegal values for the index, they should point to an error handling routine:

```
100 ON M GOTO 1000,2000,110,1500
110 PRINT "M = 3 ERROR"
120 GOTO 10 : REM START OVER
```

Loop de Loop

Computers are especially good at repetitive tasks, so every computer language needs a "loop" command. In Commodore 64 BASIC, looping may be done with simple program statements such as:

```
80  REM THIS IS A HANDMADE LOOP
90  PRINT C
100 C = C+1
110 IF C < 100 THEN 90
```

In most cases, the FOR . . . NEXT structure is the method of choice:

```
80  REM THIS IS A FOR . . . NEXT LOOP
90  FOR C = 0 TO 99
100 PRINT C
110 NEXT C
```

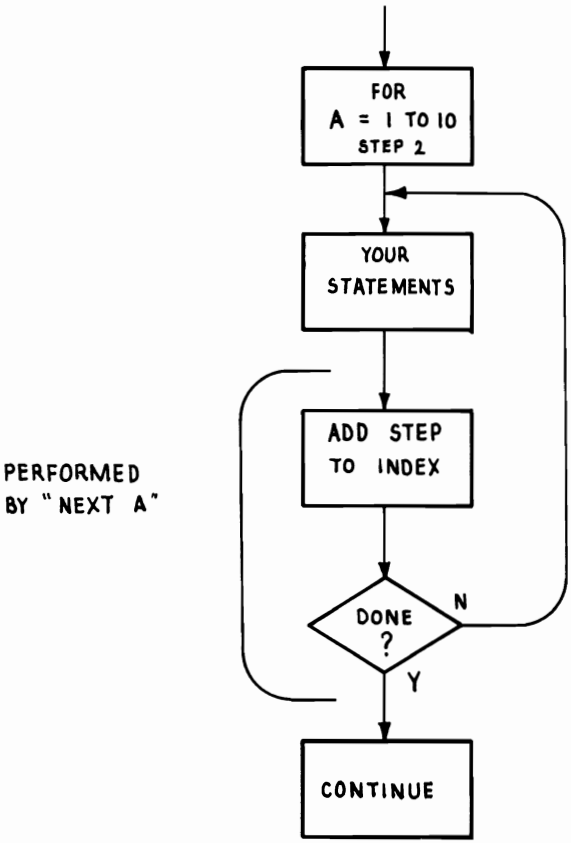


Figure 1.4: Flowchart of FOR . . . NEXT

FOR . . . NEXT loops generally run faster and are easier to read, but programmers always run into one problem: What if I want to force an exit from the loop? The common solution just happens to be the wrong solution:

```
80  REM THIS WILL BOMB THE PROGRAM
90  FOR C = 0 TO 99
100 PRINT C
110 GET A$
120 IF A$ < > "" THEN 200
130 NEXT C
```

The problem with the program above is that the BASIC interpreter keeps a list of where each NEXT should loop back to. By leaving the FOR . . . NEXT in an abnormal manner, the "loop back address" was not removed from the list. If the program did this a few hundred times it would fill up the memory space available and stop running. Since some other statement might actually add the fatal last item to the list, this problem may cause error messages referring to perfectly valid lines. Beware of this problem, as it can take hours or days to discover what is really going on.

Instead of jumping out of the FOR . . . NEXT, you can force it to leave on the next time through by changing the loop counter:

```
80  REM THIS WORKS!
90  FOR C = 0 TO 99
100 PRINT C
110 GET A$
120 IF A$ < > "" THEN C = 99
130 NEXT C
```

If you wanted to skip some of the loop you could include a GOTO statement:

```
80 REM SKIP 120 IF EARLY EXIT
90 FOR C = 0 TO 99
100 GET A$
110 IF A$ < > "" THEN C = 99: GOTO 130
120 PRINT C
130 NEXT C
```

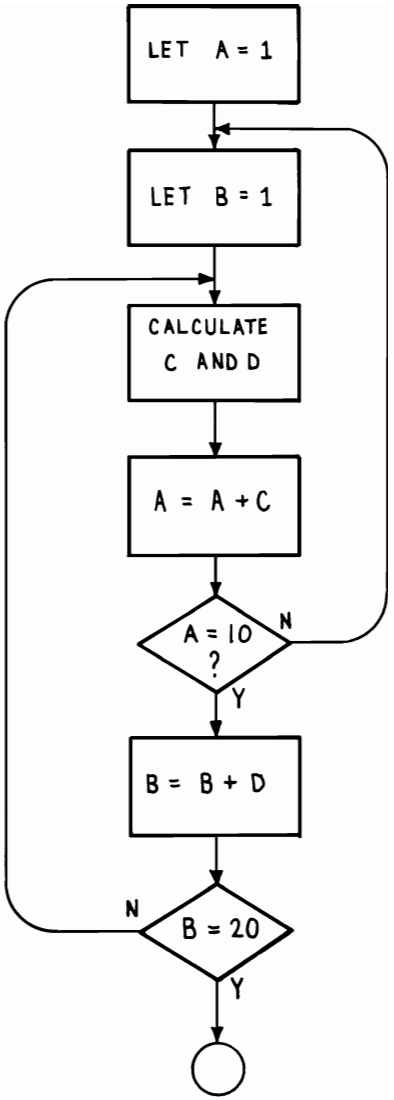


Figure 1.5: Overlapping Loops

FOR . . . NEXT loops may be nested inside of other FOR . . . NEXT loops, but you can't have them overlap:

```
80  REM THIS WON'T WORK
90  FOR C = 1 TO 200
100 FOR D = 10 TO 20
110 PRINT D,C
120 NEXT C
130 NEXT D
```

The example above actually has no meaning, but you may find that you need something like it in a program. If you can't use FOR . . . NEXT loops, write your own as shown in the first example in this section. There are many useful loop-type program control structures other than the simple FOR . . . NEXT loop, but you can create any of them using FOR . . . NEXT and IF . . . THEN.

Subroutines

Many advocates of structured programming suggest that anything worth doing in a program should be placed in a subroutine. That may be true in the case of a language which permits calling subroutines by name, but in BASIC, subroutine calls by line number tend to be confusing. The only time you should use a subroutine is if the subroutine will be called from more than one point in your program. To make subroutine calls more readable, include a comment line:

```
100 GOSUB 2020 : REM FORMAT+PRINT
110 A = B
2000 REM FORMAT N AS $XXXX.00
2010 REM PRINT AT (10,5)
2020 N$ = "$" + STR$(INT(N/100)) + ".00"
2100 RETURN
```

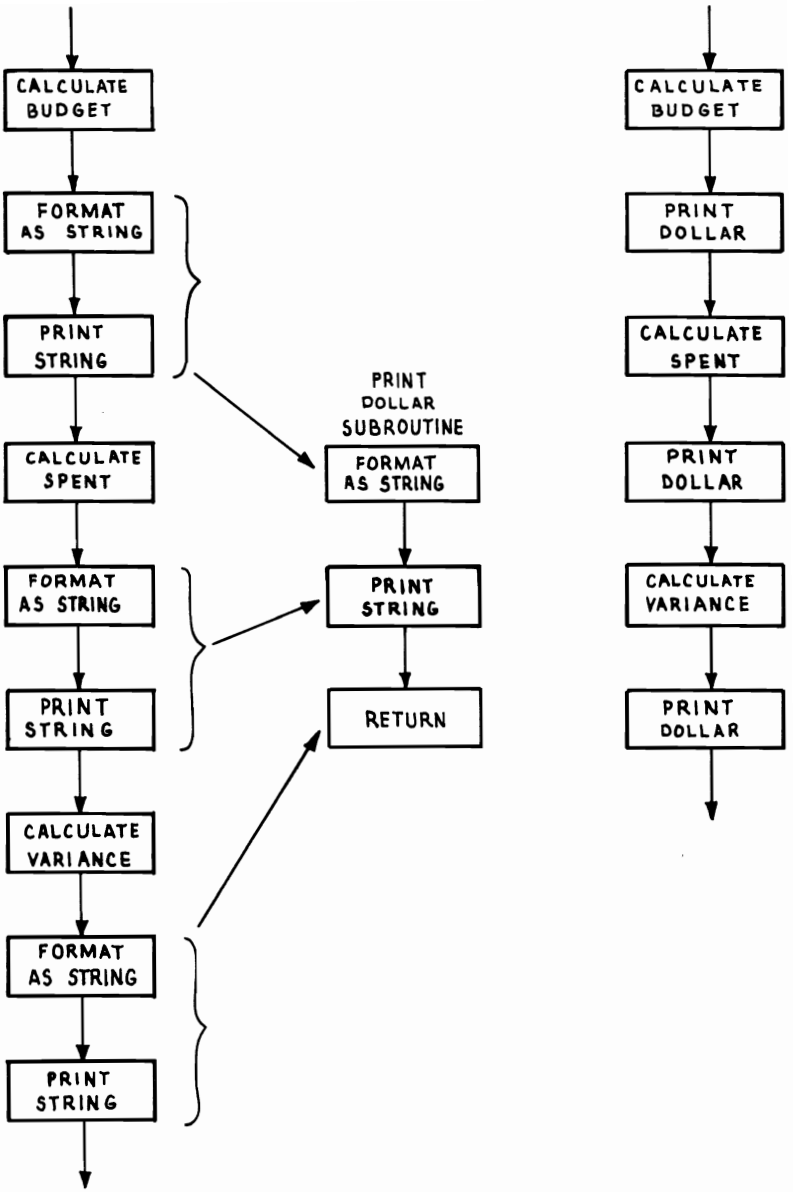


Figure 1.6 Finding Subroutines

It is a good practice to start all of your subroutines on even line number increments like 100 or 1000, with a blank line between each. In the example above, the GOSUB statement contained the line number of the first non-REMark line in the subroutine. This was not necessary, but it would speed up execution compared to using GOSUB 2000.

```
1999 :
2000 REM SUBROUTINE 1
2010 (BODY OF SUBROUTINE 1)
2180 RETURN
2190 :
2400 REM SUBROUTINE 2
2410 (BODY OF SUBROUTINE 2)
2530 RETURN
2540 :
2800 REM SUBROUTINE 3
2810 (BODY OF SUBROUTINE 3)
2970 RETURN
```

ON . . . GOSUB works in a manner similar to the ON . . . GOTO command, and also shares the same limitations. Subroutine return addresses are also stacked like FOR . . . NEXT loop addresses. To avoid bombing your program, never jump out of a subroutine. Instead, jump to the RETURN statement to leave a subroutine immediately. Jumping from the middle of one subroutine to the middle of another may work, but it isn't worth the trouble it will cause when you try to decipher the program flow from a listing.

BASIC subroutines can be very useful, but beware of the limitations:

1. You can't pass different variable names to a subroutine from different calling locations. You must set up the values you want assigned to the variables that the subroutine requires, call it, and then store the results from the variables that the subroutine uses to return values.

2. GOSUBs look quite harmless in a listing of a program that you are trying to debug. Each subroutine may alter many variables or even fail to RETURN as expected. Tracing the execution of a program that has many subroutines has caused many a good programmer to give up and call it "spaghetti code."

3. Using a subroutine will always be slower than writing the same code "in line," that is, substituting the actual statements of the subroutine in place of GOSUB statements.

What Could be Lower Than a Subroutine?

Commodore 64 BASIC permits you to define functions using the DEF FN statement. Functions work like single line, single parameter subroutines:

```
100 DEF FNA = (A * 9/5) + 32
350 TEMP = FNA(C) : REM C->F
360 PRINT TEMP;" DEGREES F"
```

Maybe you noticed that some of GOSUB's problems are not present in DEF FN; Functions are passed any variable or constant you like and return the result to any variable specified. Using a function can only change the value of the variable assigned in the calling program line. These are significant advantages over GOSUB. Use DEF FN rather than GOSUB whenever you can.

Commodore 64 BASIC permits two letter function names, so you will never run out of functions. The only limitation on the use of functions is that they must be defined before they can be called in your program:

```
10 REM THIS WON'T WORK
20 X = FNA(Y)
30 DEF FNA = A * 3 + 2
```

Define all of the functions that you will use at the start of the program or in an initialization subroutine. (More about initialization subroutines in the speed section.)

GOTO or GO TO

The lowly GOTO command has been singled out for an undeserved bad reputation among computer language philosophers. You have the choice: You can use only a few GOTOs where they make sense and have a readable program, or you can stick GOTO anywhere you run out of room between line numbers or feel like jumping into the middle of some other part of the program and end up with a plate full of spaghetti. There are instances in BASIC programs when only GOTO makes sense. For example, most game programs must repeat until you stop them:

```
10 REM START OF GAME
20 (INITIALIZE)
300 (PLAY ONE ROUND)
900 (REPORT SCORE)
990 GOTO 10 : REM START OVER
```

The most readable programs contain the minimal number of GOTOs required for the control of program flow, no more and no less.

Not for SYSsies

The SYS command offers the possibility of introducing very fast machine language subroutine calls into your BASIC program. These can often speed up a program by 10 or even 100 times, but you must understand the use of the 6510 processor instruction set to write them. As was stated earlier, this book will not make you a machine language programmer. There are already many books available for learning 6510 (6502) programming. If you don't understand 6510 programming I want you to run right out now and buy a book or two on the subject. You won't want to miss out on the fun once you've mastered BASIC . . . . Okay, now we're all expert machine language programmers, right?

The examples of the use of machine language routines in BASIC in most published material involves hand assembling, converting hex to decimal, and adding the decimal values to your program as DATA statements. This will work, but it is so tedious that you

won't want to do it more than a few times unless you are desperate for speed. Buy an assembler. If you have ever used the former method, you will be astonished by how easy machine language programming can be with a good assembler.

Join a USR Group?

The USR(X) function is similar to SYS in that it calls a machine language subroutine. The difference is that while SYS jumps to any address specified with no parameters passed, USR jumps to a fixed address with one parameter and returns one result. In general, USR tends to be too limited compared to SYS, especially after you read the chapter on using SYS routines from BASIC. You will learn how a SYS routine can use or modify any BASIC variable or array.

WAIT, There is Only One More

The WAIT statement provides a simple way for you to synchronize your program with some real-time event. Almost every input/output device, VIC or SID chip register, or IRQ routine variable may be read as a memory location by WAIT. Of course, any of these could also be checked with PEEK and IF . . . THEN, but WAIT is a much "tighter loop." If your program must do nothing until a memory location or register changes, and then respond immediately, WAIT is the method to use. Unfortunately, BASIC requires the parameters for WAIT as decimal numbers, so that there is little correspondence between the parameters and the bit(s) of interest. You can get around this and speed up your WAIT loop by setting up variables in advance:

```
100 B = 16 : REM BIT 4
1210 WAIT 1,B : REM CASSETTE
```

Chapter 2

Developing Algorithms

What is an Algorithm?

One of the most important parts of a program design is the algorithm. An algorithm is simply a plan for performing a task. You use algorithms every time you work arithmetic problems, without even thinking about them. When you solve a long division problem you follow certain steps that you learned in school which always result in a correct answer if the steps are followed correctly. How do you know when to multiply, when to subtract? You know because you learned the "division algorithm" which specifies the exact order of steps to use. Imagine writing a complete step-by-step guide for long division. Your guide would have to give precise instructions general enough to work with any divisor and any dividend. No fair using examples to force the reader to intuitively grasp the order of the steps or number of times to repeat the steps.

In writing the guide, you would find that there are only three steps needed to perform long division, but that they need to be repeated "until done." The three steps are, of course:

1. Estimate the partial quotient;
2. Multiply the partial quotient by the divisor; and
3. Subtract the result from the dividend.

You would also need to define what is meant by being "done." In defining algorithms this is often the most difficult part. For our example, let's use this definition:

Repeat until the dividend equals zero

Now we can write the division algorithm:

1. Estimate partial quotient;
2. Multiply partial quotient by divisor;
3. Subtract result from dividend; and
4. Repeat until dividend equals zero.

What's missing? What about signs? decimal points? divisors larger than dividends? Once your guide takes all of these into account, it will be a complete algorithm for division. Obviously, we take the complexities of long division for granted.

Fortunately, the Commodore 64 knows all about division. Algorithms that you write can assume that the computer already knows a great deal about arithmetic. The result is that most of the algorithms you will want to use to describe your programs will be simpler than the algorithm for long division!

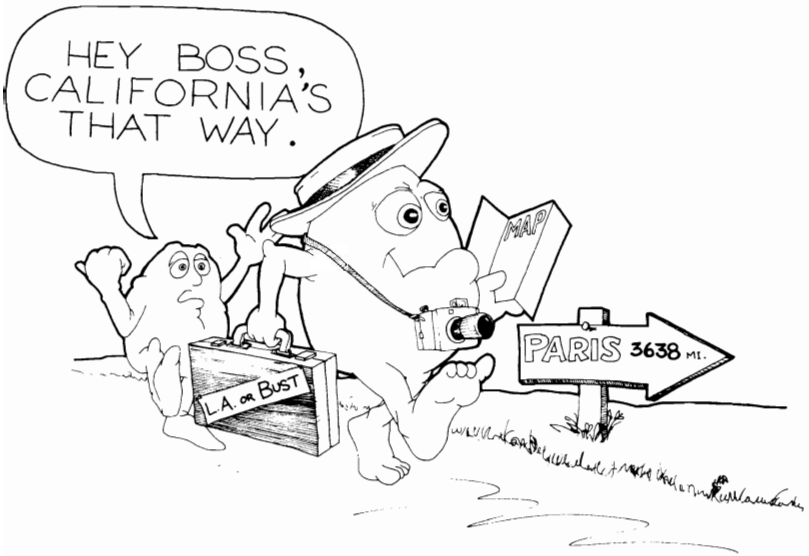
Writing a general algorithm for the problem that you want to solve results in two benefits:

1. You will be forced to state exactly what the program must do.
2. You will be forced to consider the general case rather than any specifics.

There are almost as many ways to write algorithm descriptions as there are programmers. Sometimes these descriptions take the form of mathematical equations and symbols. Unless you are going to include an algorithm description in a computer science degree dissertation, simple English sentences will do nicely. Writing such a description serves as a natural bridge in getting from a program specification to a flow chart.

How Many Ways Are There From Here to There?

Consider the problem of getting from New York City to Los Angeles by airplane. There are many possible routes that you could take. If the fares were lowered as an incentive, you might



even stop over in New Orleans for a few hours. Algorithms for solving problems are just the same; some ways of solving a problem are fast but expensive, some are slow but cheap. Watch out though, some ways may be slow and expensive! What you want to find is a method that is fast and inexpensive in terms of using the computer's resources. Searching a list of alphabetized names is a good example. The simplest algorithm is:

1. Start at the beginning of the list
2. Is this the one?
3. If it is, you're done, name found
4. Move to the next name
5. End of the list?
6. If it is, you're done, not found
7. Go to step two

In BASIC:

```
100 REM N$ = NAME L$(M) = LIST
110 P = 0
120 FOR C = 1 TO M
130 IF N$ < > L$(C) THEN 160
```

```
140 P = C
150 C = TOP
160 NEXT C
170 REM NOW P IS 0 IF NOT FOUND
180 REM OR P IS THE LIST ELEMENT NUMBER
```

To find a name in the list will take no time at all for the first name, while the last name will require every name in the list to be checked. The average time to find a name in the list will be:

(number of names) * (time per name)/2

For purposes of comparison, the time per name will be the same for any method we use, so we can say that this is an "N/2" method for searching. This method is very simple to write as an algorithm, flowchart, or program. But what if we had a hundred thousand names to examine? There must be a faster way.

There is a faster way, and you use it every time you look up a name in the telephone book. What you do is to estimate where in the list to start looking, and then zero in on the name by making more and more accurate estimates by reading names from the pages. One way to describe this process to a computer is called a "binary search":

1. Look at the name in the middle of the whole list
2. Compare the list name with the name wanted
3. If they're equal, you're done
4. Otherwise, decide which half of the list the name is in
5. Look at the name in the middle of that (partial) list
6. Go to step two

In BASIC:

```
100 REM N$ = NAME L$(M) = LIST
110 S = 1 : REM START OF SUBLIST
120 F = M : REM FINISH OF SUBLIST
130 A = (F-S)/2 : REM COMPARISON ELEMENT
140 IF L$(A) = N$ THEN P = A : GOTO 250
150 IF N$ < L$(A) THEN 210
```

```
160 :
170 REM CHOOSE TOP HALF OF SUBLIST
180 S = A+1
190 GOTO 130
200 :
210 REM CHOOSE BOTTOM HALF
220 F = A-1
230 GOTO 130
240 :
250 REM P IS THE ELEMENT NUMBER FOUND
```

After each comparison the list is divided into two parts, and one of the parts is then ignored because you know the name is not in that part. This method is very fast compared to the "brute force" method of comparing every name in the list. The average time required to find a name is:

log base 2 (number) * time per name

If there are 256 names in the list, it will require at most eight comparisons (2 to the 8th = 256). For a list of 1024 names it would require 10 comparisons (2 to the 10th = 256). And finally, for that list of 100,000 names, only 17 comparisons would be required! That's quite a bit better than the 50,000 comparison average using the brute force method.

The binary search is just about the fastest general method devised yet for searching large lists, but there may be faster ways under special circumstances. For example, if you knew in advance that the list would be small, or that most of the names to find would be near the top of the list, looking at each name from the start might be faster.

In choosing the best algorithm to implement in a program, you should consider known facts about the data and program use. Often the best general algorithm is not the best choice for a particular application. In order to analyze which is the best algorithm for a program, you must thoroughly understand why each algorithm behaves the way it does. To properly analyze a number of algorithms is beyond the scope of this book. Several excellant books have been written on this topic. One of the classics is *The Art of Computer Programming* by Donald E. Knuth.

Translating Equations into Programs

The simplest class of algorithms may be written as single algebraic equations. BASIC is particularly good at handling equations. Most equations can be written in BASIC almost indentically to their mathematical representations. For example, the conversion of Celsius to Farenheit temperatures is defined as:

Equation: $F = (C \times 9)/5 + 32$

BASIC: $F = (C * 9)/5 + 32$

The only difference, in this case, is the BASIC symbol for multiplication. The parentheses in both the equation and the BASIC statement are added only for clarity. BASIC assumes the same precedence of operations used in mathematics:

Exponentiation	<--- Done first
Multiplication and division	
Addition and subtraction	<--- Done last

In either system, the normal precedence can be defeated by adding parentheses.

There are thousands of useful equations like the previous example which you can find in mathematics or reference books. Most of these equations can be written as a single BASIC statement. More complex equations may require additional program lines. A straightforward example is finding the arithmetic mean of a list of numbers:

Equation: Mean X = Sum of X's / number of elements

In BASIC:

```
100 SUM = 0
110 FOR C = 1 TO NUMBER
120 SUM = SUM + X(C)
130 NEXT C
140 MEAN = SUM/NUMBER
```

Some Useful Algorithms

Here is a list of some algorithms that may help you. Additional algorithms can be found in computer magazine articles, computer science reference books, and, best of all, in your imagination. In this list, capital letters are used to signify valid BASIC expressions, small letters are used for derived or descriptive expressions.

```
LOG base 10 (X) = LOG(X)/LOG(10)

degrees = radians * (180/PI)
arcsin(X) = ATN(X/SQR(-X*X+1))
arccos(X) = -ATN(X/SQR(-X*X+1))+PI/2
sinhyp(X) = (EXP(X)-EXP(-X))/2
coshyp(X) = (EXP(X)+EXP(-X))/2
tanhyp(X) = -EXP(-X)/(EXP(-X)+EXP(X))*2+1

Celsius degrees = (Fahrenheit - 32) * 5 / 9
Fahrenheit degrees = Celsius * 9 / 5 + 32
mean of list X(1 to n) = (sum X(1 to n))/n
```

Here are a Few More Complex Algorithms

Sorting arrays into ascending order: Bubble Sort
Where DATA[n] is an array to be sorted.

```
FOR I = N TO 1 STEP -1
    FLAG = 0
    FOR J = 1 TO I-1
        IF DATA[J] > DATA[J+1] THEN
            SET FLAG = 1
            SWAP DATA[J] AND DATA[J+1]
    NEXT J
    IF FLAG = 0 THEN I = 1
NEXT I
```

Shell's Sort (much faster for large arrays)

```
distance = closest power of two to n/3, plus 1
begin
    for j = 1 to (n – distance)
        if data(j) > data (j + distance) then
        swap them
    next j
    if distance = 1 then exit routine
    distance = (distance–1)/2 + 1
    if distance = 2 then distance = 1
    (this generates a 2 to the k + 1 series)
repeat
```

Searching a sorted array: Binary Search
Where DATA[1 to n] is the array and pattern is the item to find.

```
trys = integer(log base 2(n)) + 1
first = 1
last = n
pointer = 0
for i = 1 to trys
    test = int(last-first)/2
    if pattern = data(test) then
        pointer = test
        terminate loop
    if pattern > data(test) then
        first = test + 1
    if pattern < data(test) then
        last = test – 1
next i
```

Searching an unsorted array: Brute Force method

```
pointer = 0
for i = 1 to n
    if pattern = data(i) then
        pointer = i
        terminate loop
next i
```

Integration of a function:
Where FN[x] is integrated from i to j and DX is selected by the programmer.

```
SUM = Ø
FOR X = I TO J STEP DX
(Now you know where this "I" came from.)
    SUM = SUM + fn[X]
NEXT X
INTEGRAL = SUM * DX
```

Polar to rectangular conversion:
Where r = radius, theta = angle in radians.

```
X = R * COS[THETA]
Y = R * SIN[THETA]
```

Rectangular to polar conversion:

```
R = SQR[X*X + Y*Y]

THETA = ATN[Y/X]
IF X < Ø THEN
    THETA = THETA + PI RADIANS
```

In Conclusion

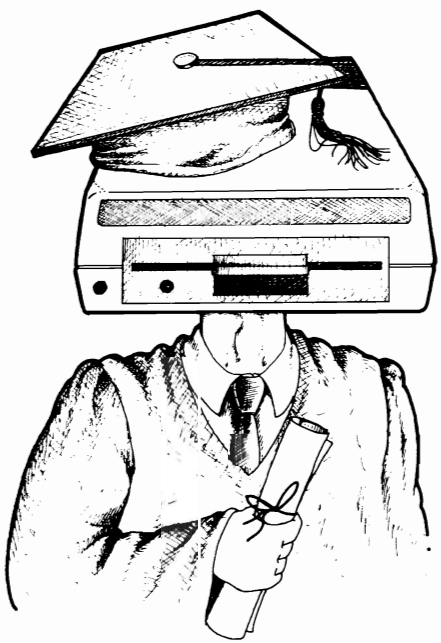
Algorithms aren't just a formality given to us by computer science academics. Thinking about the algorithms that you use in your programs will help you become a better programmer. If there is a fault in your logic, it will be easier to find if you understand the algorithm used. Choosing the correct algorithm from many that are possible can result in a program that runs in seconds instead of hours.

Most algorithms are quite simple. Even more complex algorithms are made up of simple steps, because that is what an algorithm is; a sequence of simple steps that accomplishes a given task. The next time, and everytime, you start designing a program, spend a few moments thinking about the different ways that the program tasks could be performed. It will be well worth the time invested.

Chapter 3

Advanced File Handling

Your 1541 disk drive is an extremely capable and intelligent device for storing information. You may have already noticed that the disk drive seems to continue executing commands like SCRATCH even after the computer prints the READY prompt. This can only take place because Commodore designed the 1541 disk drive as an independant intelligent device with its own built-in microcomputer. When you or your program sends a command to the disk drive, the drive doesn't receive step-by-step instructions, but rather the actual characters of the command. The drive microcomputer interprets the command and then acts upon it.



This is very different from most microcomputer disk systems which require a great deal of the central processor's time to operate the disk drive. With this "distributed processing" approach to disk operation, the Commodore 64 has more time to spend on graphics, sound, and the actual running of your program.

What Your Disk Drive Can Do

The 1541 disk drive stores up to 174,848 bytes of data on a disk in 683 blocks of 256 bytes each. These are stored as variable numbers of blocks per track, on 35 tracks. The numbers of actual blocks per track are higher on the longer outside tracks (toward track 1), and lower on the shorter inside tracks (toward track 35). Here is a table of the blocks per each range of tracks:

Track Range	Block Range	Total Blocks/Track
1 – 17	0 – 20	21
18 – 24	0 – 18	19
25 – 30	0 – 17	18
31 – 35	0 – 16	17

Figure 3.1: Blocks per Track

This variable number of blocks per track scheme does complicate the use of the disk drive in a few cases, but without it the designers would have had to use the worst-case number of 17 blocks per track. Then the disk would have only held 595 blocks, a loss of more than 12 percent.

The disk drive can be used in three different ways. The way that you are probably most familiar with is to LOAD and SAVE BASIC programs. You may not know that the same commands can be used to store and retrieve machine language programs, graphics or sprite data, or any part of system memory. This will be covered later in this chapter in sufficient detail for you to incorporate into your programs.

The second method is called "sequential access." This is the way that BASIC supports the use of named disk files for storage of one long string of characters, much the way it uses tape files. Sequential files have their limitations, but they are certainly a lot better than filling programs with thousands of DATA statements or re-entering the data during each run of the program. The use of sequential disk files in conventional as well as more elaborate ways will be discussed in the next section.

The third method of using the disk is what Commodore calls "random access." They may call it that, but it is not strictly accurate. They really did not implement random access files, but rather "direct read/write sector" capability. These are two entirely different things. More will be said about the uses of this capability in the section titled "Random Access" Files.

Using the 1541 Disk Commands

If you have had your 1541 disk drive for a while, you know all about loading programs or a directory list. If you study the manual that came with your disk drive, you should be able to figure out the use of LOAD, SAVE, OPEN, CLOSE, PRINT# 15, NEW, COPY, RENAME, and SCRATCH. There are three disk commands that need additional clarification.

VALIDATE erases the Block Allocation Map (BAM) and then rebuilds it by following the links between sequential file blocks to reorganize the disk. If you have used the BAM to protect disk blocks being used for direct sector access, they may be wiped out by VALIDATE. (Are they trying to tell us something about random files like, "they aren't really supported"?) At any rate, use VALIDATE with extreme caution, if at all.

INITIALIZE is much more useful than the manual claims. If you want to turn off that nasty flashing red light after a disk error, you can open the command channel (channel 15) and send the "I" command:

```
OPEN 15,8,15
PRINT#15,"I"
```

Actually, any valid disk command would turn off the light, so "I" seems superfluous. The "I" command is essential when you have two 1541 disk drives. You may find, as I have, that your second drive won't initialize when the Commodore 64 power is first turned on. My second drive even stops communication between the Commodore 64 and my first drive until I open the command channel and send an "I" to device 9 (the offending second drive). Then everything works fine. You will need to use "I" in this same manner if you put a friend's unmodified disk drive together with yours, and then change the device number of one of them through software, so that you can copy whole disks. What the manual left out of the explanation of how this is done is:

1. Connect the cables for both drives, but leave one of them powered off.

2. Open the command channel and send the memory write command string as per the book to change then powered drive to device number 9:

```
OPEN 15,8,15
PRINT#15,"M-W:"CHR$(119)CHR$(0)CHR$(2)
CHR$(41)CHR$(73)
```

3. Close the command channel.

```
CLOSE 15
```

4. Turn on the power to the second drive.

5. Open the command channel to device 8 (now the second drive).

```
OPEN 15,8,15
```

6. Send the "I" command to the second drive.

```
PRINT#15,"I"
```

7. Close the command channel.

```
CLOSE 15
```

8. The first drive should now respond as device 9, the second drive should respond to its old device address 8.

SAVE "Ø:filename" has been reported to be unreliable by some commercial software authors. They recommend that you save your updated file under another name and then use the SCRATCH command to get rid of the old copy. Then you can rename your file to the old name. Personally, I haven't had any trouble with "SAVE and replace," so I will keep on using it until (if ever) it sneaks up and gets me. Commodore may have already fixed this problem (if there ever was one), but I think it won't hurt you to know about it. If you ever have trouble using SAVE and replace, you might remember the solution given above.

Loading Programs From the Disk

Loading programs from the disk is simple, right? Yes and no. Loading BASIC programs into the standard BASIC work area at address 2049 is simple, but that isn't all that LOAD can do. Adding the optional ",1" command to the end of a LOAD instruction tells the Commodore 64 operating system to LOAD the program starting at an absolute address in memory. The disk drive manual tells you that this is useful for machine language and character sets, but it doesn't tell you how it works.

Each program-type file contains a two-byte load address before any actual memory data. When you type LOAD "filename",8 you are telling the operating system to ignore this load address and store the program at the start of the BASIC work space (normally 2049). When you add the ",1" and type LOAD "filename",8,1 the operating system will use the load address and begin storing the program at that memory location.

Saving Program Files on the Disk

You can make anything in your system RAM into a LOADable program file. All you need to do is:

1. OPEN a program file for write. This is done just like a sequential file, except with a file type of "P".

```
OPEN 2,8,2,"Ø:MYPROG,P,W"
```

2. Use "PRINT#<channel #>,CHR$(load address low byte)+CHR$(load address high byte);" to send the load address to the file.

```
PRINT#2,CHR$(Ø)+CHR$(128); (address = 32768)
```

3. Use a FOR . . . NEXT loop to PEEK at, and then write, the bytes of data to the file. Note that the load address sent does not have to agree with the address that you are PEEKing the data from. In fact, the data doesn't even have to be in memory, as long as you PRINT# it as CHR$(data); to the program file.

```
FOR A = 32768 TO 4Ø98Ø
PRINT#2,CHR$(PEEK(A));
NEXT A
```

4. CLOSE the file.

```
CLOSE 2
```

If you buy a Commodore 64 machine language monitor cartridge or program, you can use the "S" function to make program files from any part of memory. If you don't have a monitor program, the BASIC program below will do the same thing. Don't forget to POKE the data into memory to begin with. A good area for machine language programs is the 4096 bytes starting from 49152 ($C000). If you have a program that POKEs some sprites or machine code routines to RAM, you can run that program first to set up the RAM. Then use this program to save RAM areas that contain data or routines useful in other programs:

```
5  REM MAKE PROGRAM UTILITY
1Ø PRINT CHR$(147);"MAKE PROG UTILITY"
2Ø PRINT : PRINT
3Ø INPUT "ENTER STARTING ADDRESS
   (DECIMAL) ";S
4Ø INPUT "ENTER LENGTH (DECIMAL) ";L
```

```
50 INPUT "ENTER PROGRAM NAME ";N$
60 INPUT "ENTER DRIVE # ";D
70 REM NOW OPEN PROGRAM FILE
80 OPEN 4,D,4,"0:"+N$+",P,W"
85 REM AND COMMAND DOESN'T WORK
   ABOVE 32767
90 AL = (S+(65536*(S>32767))) AND 255
95 REM SEND LOAD ADDRESS
100 AH = INT(S/256)
110 PRINT#4,CHR$(AL);CHR$(AH);
120 REM NOW SEND DATA TO FILE
130 FOR A = S TO S+L-1
140 PRINT#4,CHR$(PEEK(A));
150 PRINT A
160 NEXT A
170 CLOSE 4
180 REM NOW CHECK FOR FAILURE
190 OPEN 15,8,15
200 INPUT#15,A,B$,C,D
210 CLOSE 15
220 IF A = 0 THEN 260
230 PRINT "SORRY, ";B$;" ON TRACK ";C;"
    SECTOR ";D
240 INPUT "TRY AGAIN ";R$
250 IF LEFT$(R$,1)="Y" THEN 10
260 END
```

There are two problems associated with the use of LOAD "filename",8,1:

Program files each contain a load address. When you LOAD a file using the ",8,1" option, it is placed in RAM at the file's load address, even though another program may have just been loaded at that same address. If you wish to load more than one machine language program at a time, you will have to plan the memory use before saving the programs on the disk. Then programs won't ever accidentally overwrite one another.

Planned program memory use:

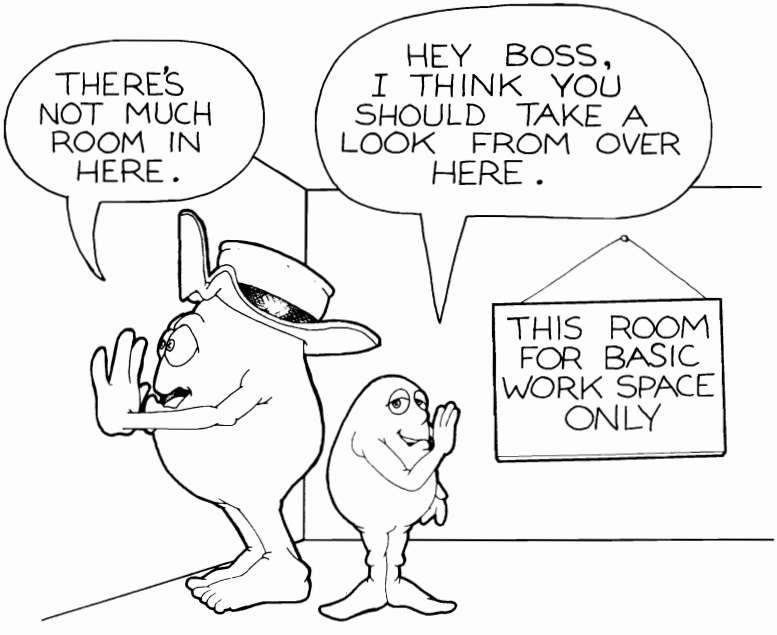
49152–49183	SCREEN.CLR
49408–49791	HIGHRES.PLOT
49920–49945	COLOR.FILL

Each of the first addresses in the above pairs marks the beginning of a new 256 byte memory page. (Each is exactly some integer times 256.) If we look at the same table in hexadecimal, it makes more sense.

Planned program memory use:

C000–C01F	SCREEN.CLR
C100–C27F	HIGHRES.PLOT
C300–C319	COLOR.FILL

If you want to use machine language subroutines or sprite data in your programs, you will have to plan for memory use anyway, so this doesn't really require any extra work. The operating system knows how many bytes are in each program file and only loads that many bytes. If the exact length of each program file is known, you can organize the memory use in a more compact manner.



Planned program memory use:

C000-C01F	SCREEN.FILL
C020-C19F	HIGHRES.PLOT
C1A0-C1B9	COLOR.FILL

The other problem that you may encounter is that using LOAD "filename",8,1 in the immediate mode destroys BASIC's memory pointers. When you try to run the regular BASIC program or add new lines, you may get an error message "out of room," but this is printed only when the load address is beyond the normal BASIC work area. What actually happens is that BASIC is left thinking that the end of the normal program is at the last address of the machine language program. Even if you don't get the error message, your program's work area will be severely restricted. Here is a way around the problem:

1. Find out what's at address 45 and 46 by typing:

```
?PEEK(45),PEEK(46)
```

2. Write down or remember the two numbers.

3. Load your machine language programs using ",8,1".

4. Restore the pointer by typing:

```
POKE 45,n:POKE 46,m
```

Where n and m are the two numbers, respectively.

Finally, the last quirk about LOAD is that within a BASIC program LOAD acts to both load a program and then to run it. This is done so that sections of a very large BASIC program can load-and-run other sections from the disk. If the program loaded isn't a BASIC program starting at the BASIC work area, the BASIC program that is there will be run! That is why you may have seen program listings that looked like this:

```
10 REM LOAD MACHINE LANGUAGE FIRST
20 IF A=0 THEN A=1:LOAD "MACHINE",8,1
30 IF A=1 THEN A=2:LOAD "MACH2",8,1
40 REM MACHINE LANG PROGRAMS LOADED
```

The first time that this program is run, A is equal to zero, so A is set to 1 and the program MACHINE loaded to wherever its load address pointed. Assuming that address was not in the BASIC area of memory, the BASIC program starts running again at line 10. This time A is equal to 1, so line 30 loads the program MACH2. When the BASIC program starts running the third time at line 10, A will be equal to 2, so that neither line 20 nor line 30 will do anything and execution will proceed from line 40. The value of the variable A was passed from one run of the BASIC program to the next because the BASIC work area wasn't disturbed. If you use LOAD within a program section to load-and-run another section of BASIC, the variables will all be reset to zero.

Using Sequential Data Files

Sequential files can be used for storing data on the disk, taking the place of BASIC DATA statements or keyboard input. A program that uses a disk file for input can be used with many different files, each containing a different set of data. In this way, the program becomes a "tool" for manipulating any disk file containing data in the proper format. Other simple programs which perform different tasks can be written to use the same file format. In this way, the programs become a "tool set" for manipulating the data in the disk files. UNIX, the rising star in operating systems for larger computers, consists mostly of a very complete set of program tools for working with sequential disk files.

A sequential disk file is made up of one long string of characters. The characters can be alphabetic, numeric, punctuation, graphics, or control characters. In fact, a character can have any value from 0 to 255, just like a memory location. Some of the characters may be CHR$(13), or RETURN. In some uses, these RETURN characters can act as separators for strings of other characters.

If the file MYDATA contains:

```
123.45<RETURN>32.98<RETURN>1298
<RETURN>
```

then the statements

```
100 OPEN 2,8,2,"0:MYDATA,S,R"
110 INPUT#2,A,B,C
120 CLOSE 2
```

will open the file and read the three values into variables A, B, and C respectively.

The key to the sucessful use of sequential files is to write the file in a format that is easily read by a program. To write a sequential file, a program must open the file, and then print a series of characters into the file. The various PRINT formatting options affect the number of <RETURN>s and superfluous spaces, so it is usually best to print one item per PRINT statement:

```
100 OPEN 2,8,2,"0:MYDATA,S,W"
110 PRINT#2,A
120 PRINT#2,B
130 PRINT#2,C
140 CLOSE 2
```

These program lines will create a file named MYDATA containing three numbers, each followed by a single <RETURN> character. If you wanted a file to contain strings, then instead of PRINT#2,A it could PRINT#2,A$. Sequential files can be created from anything that can be output with a PRINT statement.

Sequential files are often used to save and later restore numeric or string arrays. Often a program will include a program segment like those above, but with a FOR . . . NEXT or other loop:

```
100 OPEN 2,8,2,"MUCHDATA,S,R"
110 FOR N = 1 TO 100
120 INPUT#2,A(N)
130 NEXT N
140 CLOSE 2
```

These program lines open a sequential file already on the disk named MUCHDATA, and then fill the array A with 100 values read from the file. In many programs there is more than one array which needs to be filled with data. In that case, the program can load each array completely before going on to load the next, or, more commonly, fill all of the arrays within a single loop:

```
100 OPEN 2,8,2,"MIXEDATA,S,R"
110 FOR N = 1 TO 100
120 INPUT#2,A(N),B$(N),C%(N)
130 NEXT N
140 CLOSE 2
```

The file MIXEDATA would consist of 100 groups, each containing a floating point number for array A, a string for array B$, and an integer number for array C%. Another name for a group of related (but not identical) items read together from a disk file is "a record." In a complex program like a payroll check printer, there would probably be one record per employee per pay period, and each record might consist of several different items:

Record format for payroll file

last name
first name
middle initial
pay rate
Monday hours
Tuesday hours
Wednesday hours
Thursday hours
Friday hours
insurance deduction
retirement deduction

Given a file of data following this format, a simple program could:

1. Read the entire record from the file.
2. Calculate total hours, gross and net pay.
3. Print a payroll check for the net pay.
4. Go back to 1 until all the checks are printed.

If there were a mistake and the program accidentally read too many or too few items to make up an entire record, the program would be "out of synchronization" with the disk file, and all subsequent payroll checks would be printed incorrectly. This type of problem is more commonly found in complicated programs with many multi-way branches. An important part of verifying the correctness of such a program is to force it to follow every possible branch and check for inconsistencies like program paths that read different numbers of items from the disk file.

An interesting aspect of the payroll program described above is that each record is read, processed, and then discarded in turn. The items would not need to be stored in arrays in memory. This means that the data file could be much larger than the available memory. In fact, the data file could even fill an entire disk. This is a very important concept to consider when designing a program which will use data files:

Will all of the data need to reside in arrays, or can a single record be processed and then discarded?

The effect on the program can be profound. Some operations are just not practical unless all of the data is in memory at once. A good example of this is sorting. If you wanted to sort the data file of the payroll program so that the checks would be printed in alphabetical order, all of the employee names would have to be in a string array. The string array could be sorted by just a few program lines and then the checks could be printed:

```
998 REM SORT ARRAY NAME$(100)
999 REM AND ARRAY R(100) OF RECORD
    NUMBERS
1000 FOR N = 100 TO 1 STEP -1
1010 FLAG = -1
1020 FOR M = 1 TO N-1
1030 IF NAME$(M)<NAME$(M+1) THEN 1110
1040 FLAG = 0
1050 T$ = NAME$(M): REM SWAP NAMES
1060 NAME$(M) = NAME$(M+1)
1070 NAME$(M+1) = T$
1080 T = R(M): REM SWAP RECORD NUMBERS
1090 R(M) = R(M+1)
1100 R(M+1) = T
1110 NEXT M
1120 IF F = -1 THEN N = 1
1130 NEXT N
1140 RETURN
```

If the data file was too big to fit in memory all at once, then you would just be out of luck. Or would you?

There are actually several ways to get around this limitation, and some are a lot better than others. The first method I'll describe is often called "merge sort" and is based on the idea that smaller portions of a large data file can be loaded into arrays, sorted, and then written to separate disk files. Once all of the data has been divided into these sorted smaller files, the files can be merged together, a record at a time, and written to another large file. The disadvantage of this method is that you need at least as much disk space to hold the temporary files as the initial file takes up,

and you can't scratch the initial file until all of the temporary files are created. The net result is that your "large file" can only be half as large as the total space on the disk.

A more sophisticated way to handle a file that is larger than the computer's memory is to use a key file. When a program sorts the records of the payroll file described above, it doesn't really need all of the information on pay rate, daily hours, and so forth. All that it really needs are the employee names and the check amounts. The program could create a key file consisting of just the names and record numbers extracted from the large data file. Then the key file could be sorted alphabetically in arrays. The last step would be to read records from the large file in the order specified by the sorted key information and print the checks. Unfortunately, Commodore 64 BASIC does not permit you to pick the position to start reading in a sequential file.

The only way to read an out-of-order record from the main file would be to read (and ignore) each record preceding the desired record. To read a record with a lower record number, the program would have to close the file, reopen the file, and then read records until it read the one needed. Now you understand the main disadvantage of using sequential disk files: Each of the records of a sequential file must be read in sequence. With a large file, this could take several minutes for each record extracted. To print all of the records in alphabetical order, the time required would be:

$$T = t*N*N/2$$

where T = total time
t = time to read a record
N = number of records

The important factor in the equation is that N, the number of records, is squared. This means that a 400 record file would require four times as long as a 200 record file, and a 1600 record file would require 64 times as long. Obviously, this method is not practical for large files unless you only want one check printed every 10 minutes! In the section on direct access, you will see a much more efficient solution to this problem.

Now Where Did I Leave That Record?

Searching a sequential file for a certain string or number can be quite simple. Suppose we use our payroll file example and say that you wanted a program to print one employee's record data on the computer screen when the user entered the employee's name. The program would have to open the data file and read the records until it found the right name. Then it would display the information. Here is a short program to do just that.

```
10 OPEN 2,8,2,"MUCHDATA,S,R"
20 INPUT "EMPLOYEE NAME";N$
30 INPUT #2,LNAME$,FNAME$,MNAME$
40 INPUT#2,RATE,MON,TUE,WED
50 INPUT#2,THU,FRI,INS,RET
60 REM RECORD HAS BEEN READ
70 IF LNAME$ <>N$ THEN 30
80 REM RECORD HAS BEEN FOUND
90 PRINT "RECORD FOUND"
100 PRINT "LAST NAME: ";LNAME$
110 PRINT "FIRST NAME: ";FNAME$
120 PRINT "INITIAL: "; MNAME$
130 PRINT "PAY RATE: ";RATE
140 PRINT "HOURS: ";MON;TUE;WED;THU;FRI
150 PRINT "INSURANCE: ";INS
160 PRINT "RETIREMENT: ";RET
170 CLOSE 2
180 END
```

So much for searching for a record; but what about changing one of the items in case of an error? Computer errors are very rare, but operator errors happen all of the time. The difficulty in changing a sequential file is that a file must be opened either for reading or writing, not both. Also, the only place that you can start writing a sequential file is at the beginning, not at the record that is in error. Think of the sequential file as a long railroad train. To insert a car in the middle, you would have to separate the train into two parts, add the new car, and then rejoin the two parts. That is exactly what you have to do to change or insert a record into the middle of a sequential file. The program would have to open the old file for reading and a new file for writing. Then each record would have to be read from the old file and written into the new file until the proper record was found.

Changes or insertions could be made and the record written to the new output file. Then the remaining records would have to be read from the old file and written into the new file. At the end, the program would scratch the old file and rename the new one to the old file's name. For 20 records it would be a bit time consuming. But you must know by now that I wouldn't complain unless there were a better way. So here it is:

"Random Access" Files

You may remember that I whined and moaned a bit earlier in this chapter about Commodore calling these "random access files." Well, I still think that "direct disk access" is a more truthful name, but I'm not going to complain any more. There are many file applications that are easy using direct access, but very difficult using sequential files. For example, what if you wanted to search through a data file of business transactions from the most recent to earlier records. This is a "worst case" for use with sequential files. For each record, your program would have to open the file and read all of the earlier records. Using a truly ran-

dom access file, your program could go directly to the record wanted for the data. Also consider that random access channels for reading and writing can both be opened at the same time. This means that records can be read, corrected, and then written back to the disk without reading or writing any other records.

"Real" Random Access Files

Using real random access files on other computer systems is quite simple. File names and fixed record sizes are defined with an OPEN statement, something like this:

```
10 REM THIS ISN'T FOR COMMODORE-64
20 OPEN 3,"MYFILE",L=128
```

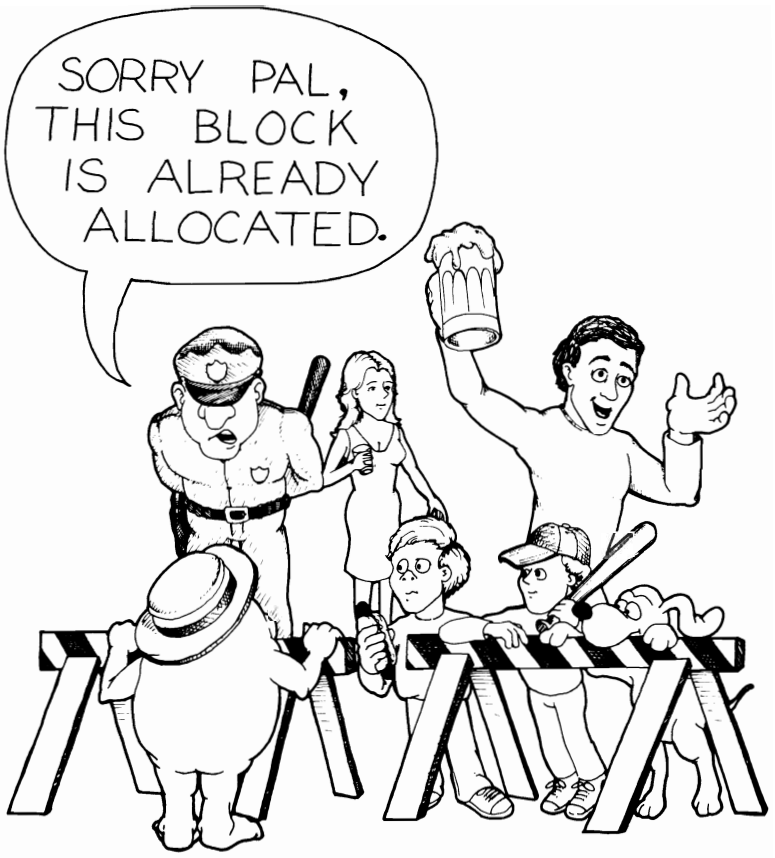
Individual records are accessed with special purpose instructions like this:

```
40 GET 3,R=26,A$,B$,C,D
50 PUT 3,R=38,X$,Y$,I,J
```

This type of truly random access simplifies data file programming tremendously. Unfortunately, Commodore has not made it that simple. What they have given us instead is the capability for programs to reserve, release, read from, and write to individual disk sectors. Commodore 64 random files do not have file names and they are not represented in the disk directory.

A Block Party First Needs a Block

Random files are created by a program repeatedly executing a BLOCK-ALLOCATE (B-A) command. This command asks the disk operating system to allocate a certain disk sector (or block) for use by the program. If the specified block is available, the DOS will mark it as used in the disk Block Allocation Map (BAM) and then return an error equal to zero. If the specified block is already marked as used, DOS will return an error of 65, along with the track and sector of the next free block.



The possibility that a desired block may already be in use should always be considered in your program design. Even on a newly formatted disk, some of the blocks of track 18 are used for the directory and BAM. The following routine obtains one disk block from the disk operating system and then records the track and sector. A program using this routine should set up T and S to contain the desired track and sector numbers before calling the routine. If the desired block is already marked as used in the BAM, the routine will get the next available block.

```
100 PRINT#15,"B-A:"0,T,S
110 INPUT#15,A,B$,C,D
120 IF A <> 65 THEN 150
130 T=C : S=D
```

```
140 GOTO 100
150 REM NOW RECORD T AND S
160 AR(N,1)=T
170 AR(N,2)=S
```

As your program is allocated disk blocks, it must keep track of the track and sector of each one. DOS does not keep a list of tracks and sectors associated with a particular random file. Also, the sectors of a random file are not linked together in any way, so your program must keep track of where a particular record is on the disk. There are two fairly simple ways to do this.

The "brute force" technique is to only store random file blocks on parts of the disk that you know will be available. For example, on a newly formatted disk, tracks 1 to 17 are completely unused, and what's more, each of these tracks contains exactly 21 sectors. If your program asks DOS to allocate these blocks on a new disk, all of the requests will be granted. Then your program can calculate the track and sector of any of 357 256-byte records on the disk without maintaining a list of tracks and sectors.

The 91,392 bytes of disk storage reserved using this technique could be divided into any number of fixed-size records. Of course, the mathematics of calculating which disk block(s) a particular record falls in will be much simpler if the record size is an integer power of two. Here are two examples of calculating record locations:

Example 1:

```
1000 REM GIVEN R, CALCULATE T,S,O
1010 REM WHERE R = RECORD NUMBER
1020 REM          T = TRACK
1030 REM          S = SECTOR
1040 REM          O = OFFSET INTO BLOCK
1050 REM RECORD SIZE = 64 BYTES
1060 REM MAX RECORD = 256/64*357
1070 REM            = 1428
1080 T = INT(R/84): REM 84 REC/TRK
1090 S = INT((R-(T*84))/4): REM 4 REC/SEC
1100 O = (R AND 3)*64
1110 T = T + 1 : REM FIRST TRACK = 1
1120 RETURN
```

Example 2:

```
1000 REM GIVEN R, CALCULATE T,S,O
1010 REM WHERE R = RECORD NUMBER
1020 REM          T = TRACK
1030 REM          S = SECTOR
1040 REM          O = OFFSET INTO BLOCK
1050 REM RECORD SIZE = 10 BYTES
1060 REM MAX RECORD = 91392/10
1070 REM            = 9139
1080 BYTE = R*10: REM ABSOLUTE BYTE
1090 T = INT(BYTE/(21*256))
1100 S = INT((BYTE-(T*21*256))/256)
1110 O = BYTE-((T*21*256)+(S*256))
1120 T = T + 1
1130 RETURN
```

For the second example, where record size is not an integer power of two, records will fall across block boundaries. If the value returned for the block offset in variable O was greater than 245, then the calling routine would have to read the part of the record in the calculated block and concatenate that with the rest of the record read from the next block. You can see how this adds a lot of complications to a program. In most cases where you use this technique, the disk space lost by using power-of-two record sizes is unimportant.

Using fixed size records requires some planning during program design. If your program will read a list of variables and strings from each record using INPUT#, there must be RETURN, comma, or semicolon characters between the individual items of the records. Some programmers avoid these extra characters by storing each record as a single string followed by a RETURN character. To write a record, the program must first build a string out of all of the individual elements of the record. Some (or all) of these elements may have to be padded with extra blanks so that each element goes into a fixed location in the record string:

Plan for 128 Byte Record:

Byte Range	Variable	Use
001-019	LNAME$	last name
020-029	FNAME$	first name
030-030	INAME$	middle initial
031-033	AREA$	area code
034-040	PHONE$	phone number
041-080	STREET$	number & street
081-120	CITY$	city & state
121-125	ZIP$	zip code
126-127	none	empty space
128	none	final RETURN

Reading a record:

```
80 OPEN 15,8,15
90 OPEN 5,8,5,"#"
100 GOSUB 1000: REM CALCULATE T,S,O
110 PRINT#15,"U1:"5;0;T;S
120 PRINT#15,"B-P:"5;O
130 INPUT#5,ALL$
140 LNAME$ = LEFT$(ALL$,19)
150 FNAME$ = MID$(ALL$,20,10)
160 INAME$ = MID$(ALL$,30,1)
170 AREA$ = MID$(ALL$,31,3)
180 PHONE$ = MID$(ALL$,34,7)
190 STREET$ = MID$(ALL$,41,40)
200 CITY$ = MID$(ALL$,81,40)
210 ZIP$ = MID$(ALL$,121,5)
```

Padding a string to proper length:

```
300 IF LEN(CITY$)=40 THEN 360
310 IF LEN(CITY$)<40 THEN 330
320 CITY$ = LEFT$(CITY$,40): GOTO 360
330 FOR N = LEN(CITY$) TO 40
340 CITY$ = CITY$ + " "
350 NEXT N
360 REM LEN(CITY$) = 40
```

Writing a record:

```
500 GOSUB 1000: REM CALCULATE T,S,O
510 PRINT#15,"B-P:"5;O
520 ALL$=LNAME$+FNAME$+INAME$+AREA$
520 ALL$=ALL$+PHONE$+CITY$+STREET$+
    ZIP$+" "
530 PRINT#5,ALL$
540 PRINT#15,"U2:"5,0,T,S
```

Instead of combining all of the elements to conserve disk space, it is much simpler to design the record length with plenty of room and then print and input individual elements of the records. The important restriction is that each record must not exceed the maximum record length. Either way, your program will have to check for and possibly adjust record lengths in order to use fixed size records.

Using Variable Length Records

The other method of keeping track of where a particular record is located requires that your program use a list of track and sector numbers. As the random file is allocated blocks, the program builds an array or arrays of these values. To access any particular record, the program finds the track and sector by using the record number as the array index. If the random file is to be permanent, the program must store the array or arrays on the disk as another file. It is most convenient to use a sequential file for this purpose. Note that this method will require a list of offset values if you want to store more than one record per 256 byte block. Reading across block boundaries is very complex, so your program should only write complete records on the blocks and then not try to use the small space remaining.

'I Sam, 'ow you doin'?

This method of maintaining a pointer to each record is sometimes called "Indexed Sequential Access Method" (ISAM). With most computer systems that support ISAM, more of the functions are

automatic, but that doesn't mean that the Commodore 64 is any less capable. By elaborating upon the track and sector list a bit, many wonderful things become possible. For example, if your program kept a track and sector list and a string array of the last names associated with each record (using our name and address program format), then by sorting the string array in RAM, each record could be printed in alphabetical order quite easily. Later, if you decided that you needed a sort by zip code instead, you could tell your program to build a new string array consisting of the zip codes of the records. Sort the zip code array and you can print the records out in zip code order for bulk mailing.

Sorting a key array:

```
1000 REM SORT ARRAYS KEY$, T, S
1010 REM N = NUMBER OF RECORDS
1020 REM RECORD SIZE = 1 BLOCK
1030 REM BUBBLESORT IS THE SLOWEST
1040 REM BUT SIMPLEST METHOD
1050 FOR M = N TO 1 STEP -1
1060 F = -1
1070 FOR P = 1 TO M-1
1080 IF KEY$(P)<KEY$(P+1) THEN 1190
1090 F = 0
1095 REM SWAP KEY STRINGS
1100 TEMP$ = KEY$(P)
1110 KEY$(P) = KEY$(P+1)
1120 KEY$(P+1) = TEMP$
1125 REM SWAP TRACK NUMBERS
1130 TEMP = T(P)
1140 T(P) = T(P+1)
1150 T(P+1) = TEMP
1155 REM SWAP SECTOR NUMBERS
1160 TEMP = S(P)
1170 S(P) = S(P+1)
1180 S(P+1) = TEMP
1190 NEXT P
1200 IF F = -1 THEN M = 1
1210 NEXT M
```

Note: Sort routine inner loops are extremely time-critical. For a real program you wouldn't want REM *statements inside of the* FOR . . . NEXT loops.

Sorting or searching a key array in RAM can be hundreds or even thousands of times faster than performing the same operations on the disk. More than one key file can be created and stored on the disk. You could write a general purpose utility program just for creating new key files. Then you could have it make a key file of names and a key file of zip codes, or anything else that you wanted to sort or search for in the random file data.

To extract the key information from a random file, the program would need a list of tracks and sectors to find each record. Then the program could read and discard items from each record until the key item was reached. The key would be stored in the key array and the program could go on to look at the next record. If the items were in fixed locations within the records, the program could use the DOS B-P command so that the unneeded items could be skipped:

```
300 REM READ KEY$ ARRAY FROM RECORDS
310 REM KEY WANTED IS AT RECORD+25
320 REM N = NUMBER OF RECORDS
330 REM T = TRACKS, S = SECTORS
340 OPEN 15,8,15
350 OPEN 5,8,5,"#"
360 FOR M = 1 TO N
370 PRINT#15,"B-R:"5;0;T(M);S(M)
380 PRINT#15,"B-P:"5;25
390 INPUT#5,KEY$(M)
400 NEXT M
410 CLOSE 5
420 REM NOW SAVE KEY$ ARRAY
430 OPEN 5,8,5,"0:MYKEYS,S,W"
440 FOR M = 1 TO N
450 PRINT#5,KEY$(M)
460 PRINT#5,T(M)
470 PRINT#5,S(M)
480 NEXT M
490 CLOSE 5
500 CLOSE 15
```

Another program could load any desired key file from the disk, sort it, and then write it back to the disk. A third program could add or delete random file records. (Once this program was run, you would have to rebuild and sort the key files.) Finally, a fourth program could print the random file data selectively, using a specified key file.

Each of these programs could be written and tested individually to keep the debugging simple. Once they all worked, you could add a short menu program that would query the user and then

load and run the desired program. If each of the programs ended by loading and running the menu program, you would have a complete "custom database manager." Having such "system programs" available for the user is one of the attractions of large mainframe computers. With your Commodore 64, you can have the same types of program tools, as long as you are willing to get in there and write them.

Format of the Commodore 64 Floppy Disk

Disk blocks each contain 256 bytes of data. The significance of each byte depends upon the way the block is being used. There are three file formats supported by the Commodore 64. These are program, sequential, and random access files. In addition, there are disk blocks that are reserved for the file directory and Block Availability Map (BAM).

Any block on the disk can be read using the B-R DOS command. If you use GET# n,D$ to input the data, you can then examine each byte read from the disk. In random access files, the only differences between bytes within the records are differences assigned by the user's program. If you wish to include links to the last record, data length, or whatever, it is completely up to you.

Format: random access block

0-255	free for data

Sequential file blocks have two bytes of data at the beginning of each block which point to the next block of the file. When a program takes input from the file, linking to the next block is automatically taken care of by BASIC. BASIC knows where the end of the file is by looking at the directory. Your program can do the same thing.

Format: sequential file block

0	next track (binary)
1	next sector (binary)
2-255	free for data

Program files are very much like sequential file blocks, except for the first block. On the first block of a program file the third and fourth bytes serve as a load address for the file. If you load the program with LOAD"name",8 the load address will be ignored and the program will be loaded to the bottom of the BASIC work area. If you use LOAD "name",8,1 then the load address will be used and the program loaded starting at that memory location. This can be very useful for loading machine language routines and utilities from the disk during a program.

Format: program file, first block

0-1	next track and sector
2-3	load address
4-255	image of program

Format: program file, all subsequent blocks

0-1	next track and sector
2-255	more image of program

In the section of this chapter on the use of the LOAD and SAVE commands, there is a short program that you can use to make a program file from any range of memory. As you work with the different file types you will sometimes need to check on the contents of a disk block to debug your program. Here is a handy utility program for reading disk blocks to RAM and then displaying the RAM contents. Several disk blocks can be read into contiguous RAM (try $C000) so that the "MAKE PROG" utility can be used to save them en masse as a single, loadable program.

```
1 REM SECTOR VIEW PROGRAM
5 HEX$ = "0123456789ABCDEF"
10 PRINT CHR$(147);"DISK TO MEMORY
   PROGRAM"
20 PRINT : PRINT
22 PRINT"R . . . READ A SECTOR TO RAM"
24 PRINT"D . . . DISPLAY MEMORY"
26 PRINT"E . . . EXIT PROGRAM"
28 INPUT R$: IF R$="D" THEN 185
29 IF R$="E" THEN 310
```

```
30 OPEN 15,8,15
40 OPEN 5,8,5,"#"
50 INPUT"WHICH TRACK (DECIMAL) ";T
60 INPUT"WHICH SECTOR(DECIMAL) ";S
70 PRINT#15,"U1:"5;0;T;S
80 INPUT"WHAT ADDRESS(HEX)          ";A$
87 GOSUB 1000 : REM CONVERT A$ TO
   DECIMAL A
90 FOR N = A TO A+255
100 GET#5,D$
105 D = ASC(D$+CHR$(0))
110 POKE N,D
120 NEXT N
130 REM FIND OUT IF THERE'S MORE
135 R$ = ""
140 INPUT"ANOTHER (Y/N) "R$
150 IF LEFT$(R$,1)="Y" THEN 50
155 REM NO MORE, CLOSE FILES
160 CLOSE 5
170 CLOSE 15
175 GOTO 20
180 REM DISPLAY MEMORY ROUTINE
185 A$ = ""
190 INPUT"DUMP RAM ADDRESS (HEX) ";A$
195 IF A$ = "" THEN 290
200 GOSUB 1000 : REM CONVERT A$ TO A
220 FOR N = A TO A+20
230 D = PEEK(N)
235 GOSUB 2000: REM N->N$ D->D$ IN HEX
238 B$="": IF D<127 THEN IF D>31 THEN
    B$=CHR$(D)
240 PRINT N$;"  ";D$;"  ";B$
250 NEXT N
260 R$ = ""
270 INPUT "MORE ";R$
280 IF LEFT$(R$,1)="Y" THEN A=A+20:
    GOTO 220
290 GOTO 20
310 END
320 :
999 REM CONVERT A$ TO DECIMAL A
```

```
1000 A = 0
1010 FOR C = 1 TO 4
1020 B = ASC(MID$(A$,C,1))-48
1030 IF B>9 THEN B = B-7
1040 A = A*16+B
1050 NEXT C
1060 RETURN
1070 :
1999 REM CONVERT N & D TO HEX N$ & D$
2000 N$ = "" : D$ = ""
2005 Q = N : R = D
2010 FOR C = 1 TO 4
2020 M = Q/16
2030 P = (M-INT(M))*16+1
2040 N$ = MID$(HEX$,P,1)+N$
2045 Q = INT(M)
2050 NEXT C
2060 :
2070 FOR C = 1 TO 2
2080 M = R/16
2090 P = (M-INT(M))*16+1
2100 D$ = MID$(HEX$,P,1)+D$
2110 R = INT(M)
2120 NEXT C
2130 RETURN
```

The Commodore 1541 disk drive can make your Commodore 64 into a real data processing machine. Not only can you load and store programs much faster than you could with a tape recorder, but your programs can make use of the increased data handling capability as well. You can use sequential disk files to store more data than the computer has memory to hold, and process it a single record at a time. You can store any type of data you like in random access files almost anywhere on the disk. You can store and reload any part of memory by creating your own program-type files.

Almost every application program running on a microcomputer makes use of a file system. Programs running on the Commodore 64 are no exception. In the following chapters we will use some of these techniques for saving graphics and machine code routines on the disk.

Chapter 4

Machine Language Programming

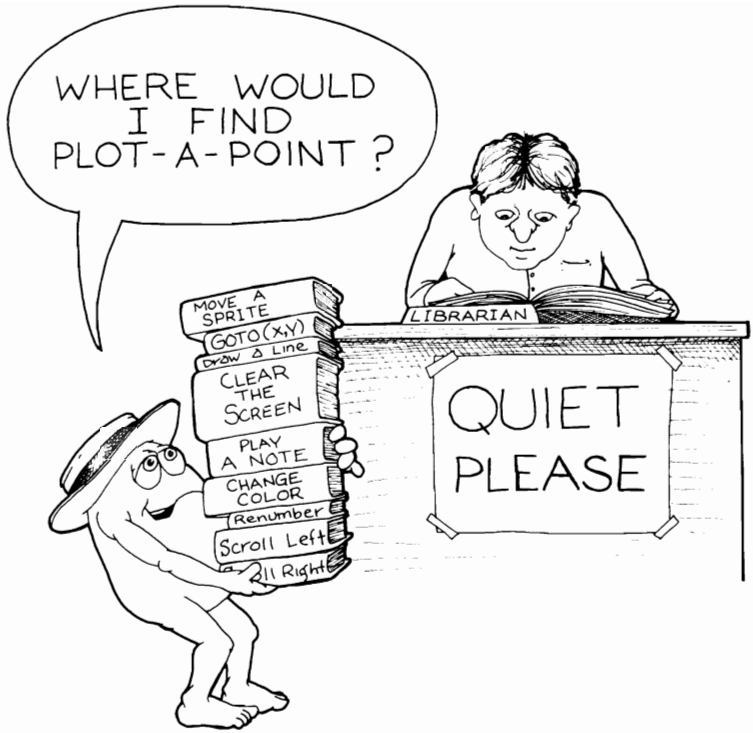
Remember that a few chapters back I told you to start studying a book on machine language programming? Well, if you didn't rush right out and buy one then, maybe you will once you've read this chapter. Many of the program examples used in later chapters will be machine language programs. Machine language programs are often used to make BASIC programs run faster or to add functions that are not included in the BASIC interpreter (i.e., you can't get there from BASIC). Both of these are good reasons, but in the case of the Commodore 64 the latter is extremely important.

Your computer contains some very capable hardware (the "chips") for generating graphics and sound, but the BASIC interpreter doesn't have any commands for using these features. You can buy an extended BASIC cartridge from Commodore to get a few more commands, but these commands are nowhere near as versatile as comparable machine language programs. Machine language programs can also be written to act upon the resident BASIC program, something a BASIC program cannot do directly. Finally, a machine language program which performs a useful task can be loaded by any of several programs as needed. This gives you the option of maintaining a "subroutine library." Interesting possibilities . . . .

Here are just a few ideas for using machine language in your Commodore 64:

- Use machine language subroutine calls from a BASIC program to plot points in either of the high resolution graphics modes.

- Clear an entire 8000 byte graphics bit map in less than a 1/60 of a second!



- Plot lines and fill areas faster than the eye can see.
- Use an interrupt routine to mix the video graphics and text modes on the same screen.
- Use the RAM under the BASIC and Kernal ROMs for enabling graphics or redefined character sets without losing BASIC program space.
- Animate by moving sprites around the screen much faster than BASIC can.
- Move sprites or play music automatically, even while editing your program!
- Renumber your BASIC program when you run out of room between statements.

- Actually add additional commands to BASIC.
- Redefine BASIC's input, output, or several other routines.
- Interface BASIC to your own unsupported hardware.
- Translate time critical portions of games to machine language for real "arcade speed."
- Write routines for collecting data from the joystick connectors and turn your computer into a programmable logic state analyzer or digital oscilloscope.

Here's How You Do It

Machine language programs are usually written in something called "assembly language." Assembly language is a set of mnemonic codes, each of which corresponds to a machine language instruction. True machine language programming consists only of numbers, be they binary, decimal, or hexadecimal (hex). Machine language instructions are what the microprocessor expects, assembly just makes all of this easier for human beings:

Machine Language	Assembly Language
$A9 $01	LDA #$01
$8D $00 $80	STA $8000
$20 $20 $C0	JSR $C020

In this example, "LDA" stands for "LoaD the Accumulator," "STA" stands for "STore the Accumulator," and "JSR" stands for "Jump to SubRoutine." The corresponding machine language opcodes, $A9, $8D and $20, are meaningful only to the 6510 microprocessor. Once a program has been written in assembly, there are several ways of converting it into a useable machine language program or subroutine.

Doing It By Hand: Slow, But Cheap

Let us say that you have written your first assembly language program. It is a very short program:

Mnemonic	Data
JSR	$3001
ORA	#$80
LDY	#$03
RTS	

The task of converting an assembly program to machine opcodes (called assembling) is so simple that you can easily do it with just a pencil and paper, and a bit of time. Note that 6510 opcodes and data are best represented as hex numbers. In the back of this

book you will find complete tables of all of the legal opcodes organized by mnemonic and addressing mode. You can use one of these tables to translate an assembly program into hex numbers. Your program should now look like this:

Hex	Mnemonic	Data
20 01 30	JSR	$3001
09 80	ORA	#$80
A0 03	LDY	#$03
60	RTS	

Note the inverse order of the address $3001 in the first instruction. That is not a typographical error. The 6510 microprocessor expects addresses that way. When more than one byte is used to store a number, different processors use different orders of significance. This difference is called "byte-sex." And you thought your computer was neuter! Suffice it to say that the 6510 has a firm byte-sexual preference.

If you wanted to include this short machine language subroutine in a BASIC program by READing data statements and POKEing memory, the next step would be to convert all of the hex numbers into decimal numbers. You can use a table to assist you, or you can use the following program:

```
5 REM CONVERTS 1 TO 4 DIGIT HEX NUMBERS
10 PRINT "HEX TO DECIMAL CONVERTER"
20 PRINT : PRINT
30 A$ = ""
40 INPUT A$
50 IF A$ = "" THEN 180
60 V = 0
70 FOR C = 1 TO LEN(A$)
80 D = ASC(MID$(A$,C,1))-48
90 IF D > 9 THEN D = D-7
100 V = V*16+D
110 NEXT C
120 IF LEN(A$)<3 THEN H$ = "": GOTO 160
130 E$ = STR$(INT(V/256))
```

```
140 F$ = STR$([V+(65536*(V>32767))]AND
    255)
150 H$ = " HI="+E$+" LO ="+F$
160 PRINT " = ";V;H$
170 GOTO 30
180 END
```

Using either method, convert the hex numbers to decimal and write them next to their hex counterparts. Your program should now look something like this:

Decimal	Hex	Mnemonic	Data
32 1 48	20 01 30	JSR	$3001
9 128	09 80	ORA	#$80
160 3	A0 03	LDY	#$03
96	60	RTS	

Finally, the last step is to include the decimal data in the BASIC program. A loop is also needed to READ and POKE the data into memory. Hex address $C000 or decimal 49152 is the first address of a 4096 byte area of RAM that is reserved for use by your machine code subroutines. Most of the machine code examples in this book will start at $C000, but they could be placed anywhere in this four kilobyte range.

```
10 REM THIS PROGRAM USES A MACHINE
   CODE SUBROUTINE
140 REM POKE THE SUBROUTINE INTO
    MEMORY
150 FOR A = 49152 TO 49159
160 READ D
170 POKE A,D
180 NEXT A
190 REM THIS IS THE SUBROUTINE CODE
200 DATA 32,1,48,9,128,160,3,96
500 REM NOW CALL THE SUBROUTINE
510 SYS 49152
```

This program will READ the 32 from the DATA statement, and then POKE it to 49152, READ the 1, POKE it to 49153, and so on. Once the machine code is in RAM, the BASIC program executes or "calls" it by use of the SYS command in line 510. A BASIC program can place many different code subroutines at different addresses in RAM and may call any of them several times.

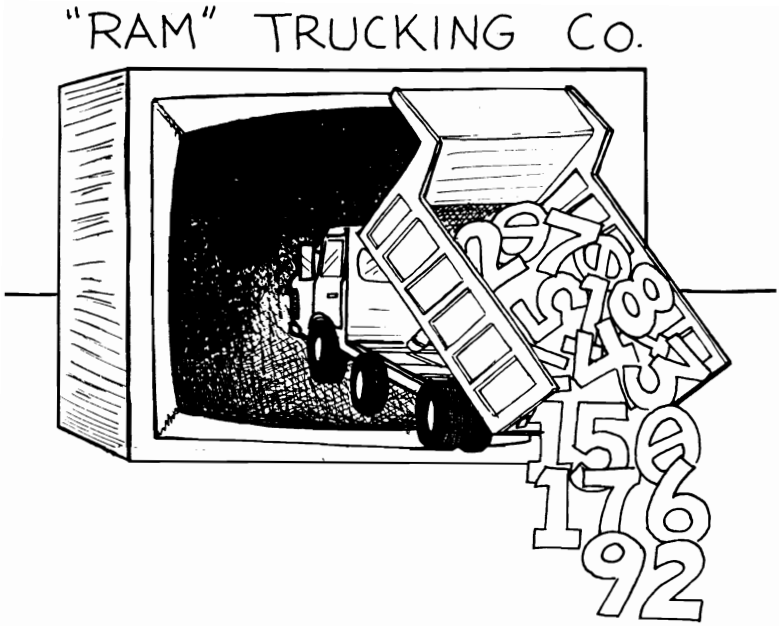
By the way, if you try to actually assemble and run this program you will probably discover another unique property of machine language programs: they can completely disrupt BASIC and cause the normal system-user interaction to stop. The former is called "bombing" and the latter "hanging-up the computer."

The result of a runaway machine language program can be anything from a totally unresponsive machine to something so subtle that you don't discover a problem until you've put in several more hours work on a program and then find that you can't save it! So here is a word to the wise: When you are testing new machine language programs or subroutines, cycle the power off and on once to completely reset the computer before going on to something else. Also, don't forget to make backup copies of important disks.

Using a Monitor Program: Fast, But Fragile

There are several different monitor programs commercially available for the Commodore 64. These may come in the form of programs on disk or as ROM cartridges. A monitor is a program that lets the user interactively inspect and alter the contents of the computer's memory. Complete monitor programs will permit you to type in lines of assembly language for direct assembly into RAM. Another feature is the ability to "disassemble" programs already in RAM. Before you buy a monitor program, make sure that it can both assemble and disassemble. It can also be very convenient if the monitor can load programs and save new programs from areas of memory.

Other ways of doing these have already been discussed, but if you are working with a monitor program it is a bother to have to run a separate BASIC "program maker" utility just to save some of memory as a program. Monitor programs assemble directly into



memory. Once the machine code is in RAM, you can save it as a program file on the disk. A BASIC program can load the machine code back into RAM using the LOAD "filename",8,1 command. If you want to store the machine code within a BASIC program as DATA items, you will need a short program to show you the contents of memory as decimal values:

```
5  REM DUMP MEMORY IN DECIMAL
10 INPUT "FIRST ADDRESS (DECIMAL) ";F
20 INPUT "LAST ADDRESS (DECIMAL) ";L
30 FOR A = F TO L
40 PRINT A;" = ";PEEK(A)
50 NEXT A
60 END
```

First use the monitor to assemble the lines of the machine language subroutine into memory. Hex address $C000 (decimal 49152) is a good place to start, since Commodore has reserved this area for your machine code subroutines. Then leave the monitor and run the program above. The program will display the

machine language code as decimal values. Write them down so that you can incorporate them into your BASIC program in DATA statements.

6510 machine language subroutines may or may not be relocatable, so be careful to POKE the machine code back to exactly where it was assembled.

Other Monitor Features

There are two other useful features that may be included in a monitor. The first is the "go" or "execute" command. Once a subroutine or entire machine code program has been assembled, you can use this command to execute it. If the code ends with an RTS, execution will return to the monitor. The other useful feature that you will find in a good monitor is the "break-point." Break-points are places in your machine code program where you want execution to temporarily return to the monitor. There will be commands for setting and removing the break-points, and for continuing execution just after the break-point. When you jump into the code with the "go" command, the first break-point executed will cause a return to the monitor and a display of all the register contents. You may use monitor commands to change register contents if you like, and then jump back into the machine code. Break-points are often the only possible way of debugging a machine language program.

Using an Assembler: Worth the Expense?

Assembler programs for the Commodore 64 are available from several sources, including Commodore. One of the most complete assemblers is called "Merlin" from Southwestern Data Systems. For a good assembler you can pay anywhere from $30 to $200. Using an assembler is the easiest way to work with programs longer than a few lines. With a typical assembler you can enter and edit source code files, assemble to RAM or directly to disk program files, assemble several different source code files together, and print assembly listings. Assembler output is normally in the form of disk files, but by assembling into RAM and

then running the program above, you can convert the assembler output into decimal DATA statements for a BASIC program.

The main convenience of using an assembler is that you can save source code files on the disk. If there is a bug in the assembled code, it is a trivial exercise to edit the source code file and reassemble. If you use a monitor program, to fix an error you may have to re-enter all of the program lines each time! Since even experienced machine language programmers often write programs with "bugs," I would recommend that anyone interested in learning and using machine language buy an assembler. This does not mean that I don't recommend monitor programs. Both assemblers and monitors are essential tools for the machine language programmer.

Extending the System Routines: Intercepting Vectors

When a system programmer writes service routines for a computer, the decision has to be made to either "hard-wire" or "vector" the routine calls. With the hard-wired approach, calls are made to the absolute addresses of each routine to output characters, fetch the next byte of a BASIC program, handle a hardware interrupt, and so forth. With the vectored technique used in the Commodore 64, a call to a system routine is made to an indirect jump instruction which uses RAM data to point to the routine which is to be used. When you press <RUN STOP> and <RESTORE>, the RAM pointers to the standard routines are all restored. If you wish to change the behavior of a vectored routine, you may (carefully) store the address of your own substitute routine at the vector location. Here is a partial list of system vectors:

Hex	Decimal	Description
0003	3	Convert floating to integer
0005	5	Convert integer to floating
0043	67	BASIC INPUT routine
00F5	245	Keyboard decode table
028F	655	Keyboard table setup

Hex	Decimal	Description
0300	768	Print BASIC error message
0302	770	BASIC warm start
0304	772	Tokenize BASIC text
0306	774	BASIC LIST command
0308	776	BASIC character dispatch
030A	778	BASIC token evaluation
0314	788	Hardware IRQ interrupt
0316	790	BRK instruction interrupt
0318	792	Non-Maskable Interrupt
031A	794	Kernal OPEN
031C	796	Kernal CLOSE
031E	798	Kernal CHKIN
0320	800	Kernal CHKOUT
0322	802	Kernal CLRCHN
0324	804	Kernal CHRIN
0326	806	Kernal CHROUT
0328	808	Kernal STOP
032A	810	Kernal GETIN
032C	812	Kernal CLALL
032E	814	unassigned vector
0330	816	Kernal LOAD
0332	818	Kernal SAVE

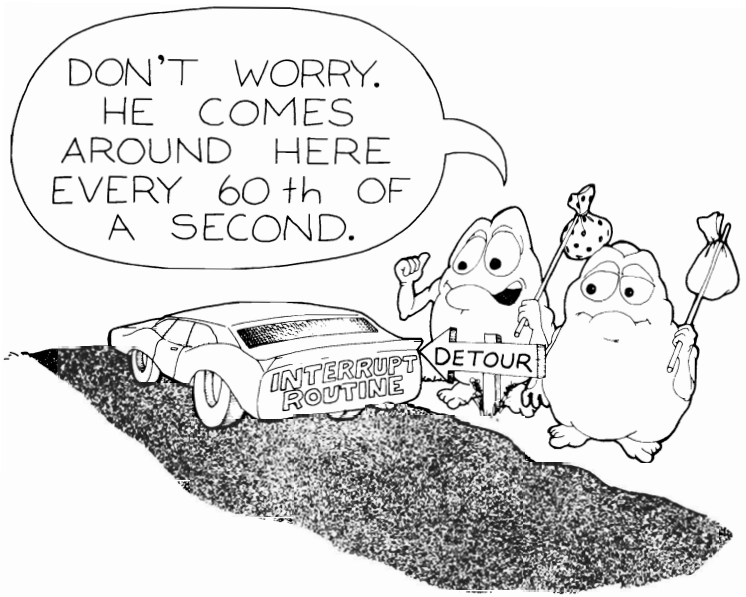
If you intercept any of these system routine calls by storing the address of your own program at the appropriate vector location, it is quite likely that the computer will "hang" unless your program ends with a jump to the original vector address. In general, intercepting vectors is something for very advanced programmers who thoroughly understand the system routine that they are trying to modify. There will be several examples of interception in later chapters. Each will follow these rules:

1. Leave with the registers exactly as they were at the start of the routine.

2. Leave with the stack depth unchanged (or restored).

3. Leave with the processor status flags unchanged (or restored).

4. Exit the interception routine with a jump to the normal routine.



The Clock Is Running: IRQ Interception

The most valuable vector interception in the Commodore 64 is the hardware IRQ interrupt. An interrupt does just what it says: A special signal tells the microprocessor to stop what it is doing in order to attend to something more important. When the task is completed, the microprocessor returns to what it was doing before. In the Commodore 64 there is a regular timer interrupt from one of the CIA chips every sixtieth of a second for keyboard scanning and updating the real time clock. Even though the computer seems to only be running BASIC, these interrupts are always being processed. By intercepting, you can have machine code routines executed automatically! Once a routine is linked in via the vector, it will keep on executing 60 times per second even while you edit and run BASIC programs. Sprites can be moved, music played, or your own machine code routine can be running. This type of interception is also the key to using mixed-mode graphics. In the VIC chip there are registers for enabling "interrupt on raster line x" where "x" is user programmable. If your IRQ interception routine tests for this interrupt, it can change

graphics modes and then update "x" for the next mode switch. You could write a routine that changed the mode at every raster line. In the chapter on graphics there will be an example of this technique.

I Told You So

Using machine language in the form of assembly language, machine language program files, or BASIC "POKE routines" is the key to getting the maximum performance from your Commodore 64. It seems that there are more possibilities for enhancement through machine code calls than there are through writing larger (and slower) BASIC programs. The hardware in the Commodore 64 has fantastic features; things that personal computer buffs could only dream about a few years ago. The BASIC supplied with the machine is merely a default subset of all of the machine's capabilities. Don't let it stop you from using all of the power in the machine. If you aren't already a machine language programmer, start today.

Chapter 5

Character Graphics

The Commodore 64 contains some of the most sophisticated video graphics hardware currently available in a personal computer. The computer was originally called the VIC-40, where VIC stood for Video Interface Computer. Almost every raster graphics technique devised has been incorporated in the Commodore 64 design.

There are three main graphic modes available with the Commodore 64. These are character graphics, bit mapped (high-resolution) graphics, and sprites. Character graphics and bit mapped graphics are mutually exclusive, but sprites can be used with all other graphic modes. Each of the techniques has several options, so that effectively there are six different modes. Commodore can offer this flexibility because, unlike other personal computer manufacturers, they own MOS Technology, their own integrated circuit company. The Commodore 64 does with a single MOS chip what other personal computers can't even match with 10 to 30 chips. Now for the bad news . . . .

Unfortunately, the BASIC in your Commodore 64 only directly supports the character graphics mode. Using bit mapped graphics or sprites require POKEing into machine registers. Bit mapping, sprites, and even use of a redefined character set requires careful management of system memory use. These techniques are complex, but then again, most of them aren't even possible with other personal computers.

Character Graphics: The Default Mode

In any standard Commodore 64 BASIC program, the text screen of 25 lines of 40 characters each is available for the simplest of

graphic displays. The standard character set contains special graphic shapes in addition to the usual numbers, letters, and punctuation symbols. The BASIC program can clear the screen, move the cursor, or display any character in any color by simply using PRINT commands. String assignments and string DATA statements can contain complex combinations of cursor-movement characters, color-shift characters, and printing characters.

Cursor Movement Characters

CHR$(19)	Home cursor
CHR$(147)	Home and clear screen
CHR$(17)	Move down
CHR$(29)	Move right
CHR$(145)	Move up
CHR$(157)	Move left

You may have noticed that the cursor commands are all relative to the current cursor position or the upper left corner. To move to an absolute screen position before printing, you can keep track of the current position and print a series of movement commands, home the cursor and then print down and right commands as needed, or use this simple subroutine:

```
1000 REM MOVE CURSOR TO X,Y
1010 REM SET UP X AND Y BEFORE ENTRY
1020 POKE 211,X
1030 POKE 214,Y
1040 SYS 58732
1050 RETURN
```

Color Shift Characters

CHR$(5)	White
CHR$(28)	Red
CHR$(30)	Green
CHR$(31)	Blue
CHR$(144)	Black
CHR$(156)	Purple
CHR$(158)	Yellow
CHR$(159)	Cyan
CHR$(18)	Reverse on
CHR$(146)	Reverse off

Printing characters can best be observed by running this program:

```
10 FOR D = 32 TO 95 STEP 3
20 PRINT D;CHR$(D),
30 PRINT D+1;CHR$(D+1),
40 PRINT D+2;CHR$(D+2)
50 NEXT D
```

Graphic symbols can best be observed by running this program:

```
10 FOR D = 96 TO 191 STEP 3
20 IF D = 129 THEN D = 161
30 PRINT D;CHR$(D),
40 PRINT D+1;CHR$(D+1),
50 PRINT D+2;CHR$(D+2)
60 NEXT D
```

All of the characters above may be included in a string assignment by typing the corresponding key(s) within the quote marks of the string assignment. For example, you could add to the readability of your program by including lines such as:

```
10 YELLOW$ = "<CTRL-8>"
20 MOVEUP$ = "<SHIFT-CRSR-UP>"
30 HOME$ = "<CLR HOME>"
```

Where the graphic character represented by the <CTRL-8> is generated by holding down the "CTRL" key (left side of keyboard) and then pressing the "8" key; <SHIFT-CRSR-UP> is generated by holding down the <SHIFT> key and then pressing the "CRSR" key with the up and down arrows; <CLR HOME> is generated by pressing the "CLR HOME" key. After a single quote has been typed, each of these procedures will generate a single funny-looking graphics character. Then you can type the final quote.

The quote characters tell the BASIC interpreter that you are defining "run-time action" rather than requesting "immediate action." If you didn't use the quote characters, each command would be executed as you typed in the BASIC line. There is

another way of defining meaningful strings for use in your programs. Using the CHR$ function isn't as easy, but it is helpful if you try to list your programs on a non-Commodore printer:

```
10 YELLOW$ = CHR$(158)
20 MOVEUP$ = CHR$(145)
30 HOME$ = CHR$(19)
40 UPPER$ = CHR$(142)(switch to uppercase)
50 LOWER$ = CHR$(14) (switch to lowercase)
```

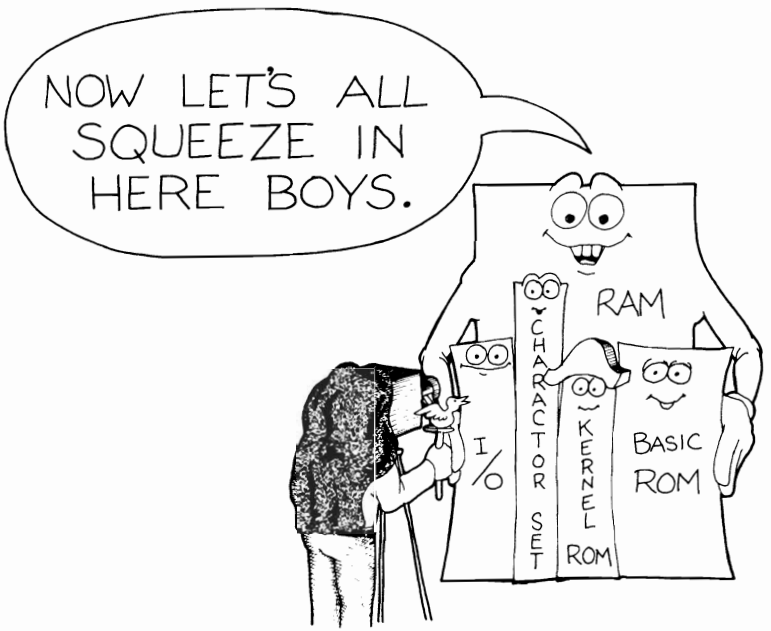
The last two commands CANNOT be typed in directly within quotes. The above example is one of the ways that you can make a program that shifts between the two character sets already in the Commodore 64. To see the second character set, press the <SHIFT> and <COMMODORE> keys together and then run the programs listed above. The two character sets are different only in some of the printing characters:

Character Sets

Range	"Upper" set	"Lower" set
65-90	A through Z	a through z
97-122	graphics	A through Z

Only one of the character sets can be displayed at one time. Each set is a complete 2048 byte area of memory containing the bit patterns needed for generating the characters. The address of the set to be used must be stored in one of the VIC chip registers. The normal BASIC keyboard scanning routine stores an address in this register each time you press the <COMMODORE> and <SHIFT> keys together. If you want to design your own character set, your program can store its address in the VIC chip and all text displays will use it for character generation.

The built-in character sets are hidden "behind" the regular memory at $D000 and $D800 in ROM (Read Only Memory). When the VIC chip accesses $1000 or $9000 to get a character's bit patterns, it sees an image of the $D000 character ROM rather than RAM. If you want to redefine the character set, you will have to do so in another area of memory. This may seem complicated, but there is a very good reason for all of this "phantom memory."



Memory Management: 88K in a 64K Space

The 6510 processor in the Commodore 64 can directly address 65535 bytes (64K) of memory. In your Commodore 64 there are 64K of RAM, 20K of ROM, and 4K of memory mapped input/output devices. How is this magic performed? Several different memory chips and I/O devices share address space by means of "bank switching." The 6510 processor has an I/O port at $0000 and $0001 which controls this bank switching. When the Commodore 64 starts running in BASIC, the processor memory is divided like this:

Default 6510 Processor's Memory Map

Memory Range	Use
$0000-$07FF	BASIC and operating system overhead
$0800-$9FFF	Your BASIC program workspace
$A000-$BFFF	BASIC language in ROM
$C000-$CFFF	Free RAM for your machine code programs
$D000-$DFFF	Input/Output devices and color RAM
$E000-$FFFF	Operating system kernal ROM

The video display in the Commodore 64 is generated entirely by the VIC chip. The VIC can halt the 6510 and take over anytime it needs to read data from memory. Here is how the VIC sees the system memory:

Default VIC Memory Map

Memory Range	Use
$0000-$03FF	Ignored
$0400-$07E7	Text screen (40 by 25)
$07E8-$07F7	Ignored
$07F8-$07FF	Sprite data pointers
$0800-$0FFF	Ignored
$1000-$1FFF	Character generator ROM
$2000-$3FFF	Ignored
$4000-$D7FF	Outside of VIC address space
$D800-$DBE7	Character Color RAM
$DBE8-$FFFF	Outside of VIC address space

This is the default memory map, but many other mappings are possible. For now, we will only consider the default case.

How Characters are Displayed

When a character is displayed, the VIC has to look at 10 different memory locations. First, it looks at the contents of the text screen RAM to find out which character is to be displayed at that screen position. Then, it looks at the corresponding nybble (4 bits) of the character color RAM to see what color the character should be.

Then, for each raster line of the display, it looks up the bit pattern of the character in the character generation ROM. While the 6510 processor sees normal system RAM at $1000, the VIC sees the character generater ROM there instead. The 6510 can see the character generater ROM, but it sees it at $D000 to $DFFF, and then only after some tricky hardware manipulation.

When a character is placed on the screen by a BASIC print statement, BASIC does three things:

1. It stores the character at the cursor location;

2. It sets the corresponding nybble in the character color RAM to the current color; and

3. It advances the cursor and scrolls if necessary.

There is another way to display characters on the screen. You can use the POKE command to store any character code on the text screen. Try this short program:

```
10 REM POKE THE WHOLE SCREEN
20 FOR A = 1024 TO 2023
30 POKE A,(A AND 255)
40 NEXT A
```

The whole screen will be filled with several copies of the character set. The colors of the characters POKEd will depend upon the contents of the color RAM. On the left side of the screen you will see the normal background and foreground colors where BASIC put them to color previous output. Toward the right side of the screen the colors will be those that were last printed at each location. Note that the color memory is scrolled whenever the normal text screen is scrolled so that colors follow "their" characters. Now things get a bit tricky: There are two different versions of the Commodore 64. On my computer, each time you do a screen clear with either the CLR key or by printing CHR$(147), the color memory is set to white. You don't see the white because the text screen is totally cleared. Any new characters such as "READY." are printed in the current color. If you have the other version, neither of these will reset the color memory.

You can determine which version you have by clearing the screen and then running the above short program. If all of the characters are white, then you have the same version that I have. If mostly other colors appear, then beware: In several of the following programs, CHR$(147) is used to clear the text screen AND set the color memory all to white. You will have to add this line to these programs:

```
2 FOR A=55296 TO 56295:POKE A,1:NEXT
```

Whichever version you have, you can set up whole display areas to have certain colors by POKEing the color RAM:

```
1  REM CHARACTER COLOR DEMO
5  PRINT CHR$(147)
10 REM FILL THE TEXT SCREEN
20 FOR A = 1024 TO 2023
30 POKE A,(A AND 255)
40 NEXT A
50 REM FILL THE COLOR RAM
60 FOR A = 0 TO 999
70 POKE A+55296,(A AND 15)
80 NEXT A
```

The screen will fill with blue and white characters and then they will all change color. If you don't like the way BASIC displays via PRINT statements, your program can POKE characters directly to the text screen and colors directly into the color RAM, or any mixture of either with normal PRINT statements. For example, your program prompts could flash from one color to another to get the user's attention.

Another useful function for text display is scrolling. The screen display software used by BASIC will only scroll upward if a RETURN is printed with the cursor on the bottom line, or if the last character position on the screen is printed. You can force an upward scroll by printing several cursor-down characters, or by

repositioning the cursor to the bottom line and then using the command PRINT which automatically issues a carriage return:

```
1000 REM UP-SCROLL SUBROUTINE
1010 X = PEEK(211)
1020 Y = PEEK(214)
1030 POKE 214,24
1040 SYS 58732 : REM POSITION CURSOR
1050 PRINT
1060 POKE 211,X
1070 POKE 214,Y
1080 SYS 58732 : REM OLD POSITION
1090 RETURN
```

Scrolling down, left, or right is feasible only with machine code routines. The algorithms for each are simple, but they would all be agonizingly slow in BASIC:

```
                          ; sample left scroll routine
        * = $C000         ; origin at $C000
        POINT = $FB       ; free page 0 RAM
                          ;
                          ; this routine scrolls the
                          ; display left one column
LEFT    LDA  #$00         ; set up pointer
        STA  POINT
        LDA  #$04         ; high byte of pointer
        STA  POINT+1
LOOP    LDY  #$01         ; index from pointer+1
        LDA  (POINT),Y    ; get character
        DEY               ; index to pointer
        STA  (POINT),Y    ; put character
        INC  POINT        ; advance pointer
        BNE  POKAY
        INC  POINT+1
POKAY   LDA  POINT        ; see if we're done
        CMP  #$E8         ; last low address+1
        BNE  LOOP
        LDA  POINT+1
```

```
CMP #$Ø7        ; last high address
BNE LOOP
RTS             ; or maybe blank the right
                  column?
```

Writing Games with Character Graphics

If you want to write a simple game or graphic display program very quickly, you should use character graphics and BASIC. Note that I said "write . . . quickly", not "run quickly"! In addition to alphanumeric and punctuation characters there are many graphic characters in the first character set. Your program can position the cursor (as in the program above) and then print these characters in any color at any screen position. You can include graphic characters in string assignments or PRINT statements by just typing them in. If you look at the front surface of each key you will see the "<COMMODORE>-key" graphic on the left side and the "<SHIFT>-key" graphic on the right. Remember that these only work correctly with the primary character set, that is, without lower case characters.

Your programs can calculate the absolute memory address of any character or color position so that the data at that address may be read:

```
1ØØØ REM CALCULATE A AND C GIVEN X AND Y
1Ø1Ø REM A = CHARACTER LOCATION
1Ø2Ø REM C = COLOR LOCATION
1Ø3Ø A = (4Ø*Y)+X+1Ø24
1Ø4Ø C = (4Ø*Y)+X+55296
1Ø5Ø RETURN

2ØØØ REM RETURN L AND S GIVEN A AND C
2Ø1Ø REM L = LETTER (CHARACTER CODE)
2Ø2Ø REM S = SHADE (COLOR CODE)
2Ø3Ø L = PEEK(A)
2Ø4Ø S = PEEK(C)
2Ø5Ø RETURN
```

```
3000 REM SET LOCATION (X,Y) TO L AND S
3010 REM ALL FOUR MUST BE SET UP FIRST
3020 GOSUB 1000 : REM CALCULATE A AND C
3030 POKE A,L
3040 POKE C,S
3050 RETURN
```

The three subroutines above were written for clarity rather than speed. In a real game program you would probably want to optimize for speed later by removing comments, defining C as A+54272, and precalculating all constants. More of this type of strategy will be covered in the chapter about the internals of BASIC.

To see the type of speed possible using character graphics from a BASIC program, try running this short program:

```
1  REM BOUNCING BALL
10 PRINT CHR$(147);
20 X = 0 : Y = 0
30 DX = 1 : DY = 1
40 POKE (Y*40)+X+1024,81
50 X = X+DX
60 IF X = 0 OR X = 39 THEN DX = -DX
70 Y = Y+DY
80 IF Y = 0 OR Y = 24 THEN DY = -DY
90 IF Y = 24 AND X = 0 THEN 10
100 GOTO 40
```

This program will POKE the text screen with the graphic "ball" character in a pattern resembling a ball bouncing off walls. This same effect could have been accomplished using PRINT statements for cursor movement and character display, but it would have required more complex logic to handle the changing directions of DX and DY. In most cases, PRINTing cursor movements and display characters is faster than POKEing, especially if you need to POKE both the character and the color. To erase characters, your program can either PRINT or POKE a space character to the text screen, or POKE the corresponding color map address with the code for the background color.

Normally, blanking the character on the text screen is preferable, but if you wanted to temporarily hide a block of characters you could change their color to the background color and then later reveal them by changing their color to something else. The advantage of this would be that your program would not have to store the characters for reprinting, but rather would have to remember the screen area of the hidden text:

```
1000 REM HIDE THE TOP LINE OF TEXT
1010 D = PEEK(53281) : REM BACKGROUND
     COLOR
1020 FOR A = 55296 TO 55335
1030 POKE A,D
1040 NEXT A
1050 RETURN

1200 REM REVEAL THE TOP LINE OF TEXT
1210 REM C = COLOR CODE
1220 FOR A = 55296 TO 55335
1230 POKE A,C
1240 NEXT A
1250 RETURN
```

One important difference to note between PRINTing and POKEing characters is that the actual character codes are different. Commodore chose to use PET-compatible character codes internally, while the BASIC PRINT command uses the ASCII character codes. You could redefine the second character set so that more of the internal display codes had ASCII values, but then BASIC PRINT commands would not work right. In the Commodore User's Guide that came with your computer, there are appendices of the display and ASCII codes. Just remember that you must use the display codes whenever you POKE characters to the video screen.

Using Other Character Sets

The Commodore 64 character sets are nothing more than bit patterns stored in memory. Each screen character is formed from

eight contiguous bytes, one for each row of pixels. An "A" character, for example, can be formed by:

byte 0 --> 00011000 = hex 18 = 24
byte 1 --> 00111100 = hex 3C = 60
byte 2 --> 01100110 = hex 66 = 102
byte 3 --> 01111110 = hex 7E = 126
byte 4 --> 01100110 = hex 66 = 102
byte 5 --> 01100110 = hex 66 = 102
byte 6 --> 01100110 = hex 66 = 102
byte 7 --> 00000000 = hex 00 = 0

A character set consists of 256 character definitions like the "A" above. The order of the character patterns in the 2048 byte character set relates each characters code to a certain pattern. A character set can start on any 2048 byte boundary, but for now let's assume that our new set will start at $3800 or 14336.

$3800 --> pattern for 0 --> was @
$3808 --> pattern for 1 --> was A
$3810 --> pattern for 2 --> was B
$3818 --> pattern for 3 --> was C

$3FF8 --> pattern for 255 --> was a graphic

If you want an "A" displayed when a character code of 1 is stored in part of the screen memory, the pattern for the "A" would be stored at $3808-380F. If you want another shape displayed for a 1 code, the eight bytes defining that shape would be stored at $3808-380F instead. Likewise, each of the 256 character codes can be defined as a familiar character or as any combination of lit pixels in an 8 by 8 array that you care to specify.

There are two important considerations when designing a character set:

1. If your character set will be used with a color display, always turn on pairs of adjacent pixels to avoid strange color effects. A single lit pixel will appear as another color. This is a consequence of the way color television works, not a fault in the Commodore 64.

2. If your character set has only a few differences from one of the normal sets, it is a lot less work to copy and then modify the existing set than it is to completely create your own. This applies both to your time while programming and designing, and to the run-time of a program using a different character set.

Earlier, memory maps of the 6510 and the VIC showed that each "processor" had a different view of system memory. The VIC chip by itself can only address 16K of memory because it only has 14 address lines. The 16K bank used is determined by two lines coming from the output port of one of the CIA chips. Using registers in the VIC and the CIA, you can tell the VIC to use any 2K area of memory as the character set. Several of these areas are used for other purposes by the operating system and BASIC. If you want to use BASIC PRINT commands, you will have to keep the display screen at $0400, so the VIC must operate within the bottom 16K. (Note that this may have profound effects upon the maximum size of a BASIC program. With large programs or programs written in machine code, it is often worthwhile writing your own screen output routines that work with any display area.)

Address	Comment
$0000	Used exclusively by system and BASIC.
$0800	Possible to use with a few POKEs to raise the BASIC work space.
$1000	A character set stored here will be ignored by the VIC, since it sees the regular set here.
$1800	A character set stored here will be ignored by the VIC, since it sees the upper/lower case set here.
$2000	Usable with a 6K BASIC program.
$2800	Usable with an 8K BASIC program.
$3000	Usable with a 10K BASIC program.
$3800	Usable with a 12K BASIC program.

In the chapter on how BASIC works, the use of $0800 by moving the BASIC work area will be discussed. For now we will assume that using the $3800 area is the best we can do. 12K is certainly a lot less than the 38K possible for BASIC work space, but it will do for many programs.

$3800 is in the middle of the BASIC workspace. If your program is small enough that the program lines and variables all fit well below the 12K size limit, it could probably run without overwriting the new character set. However, some string operations write new string variables each time they are performed. If your program uses multiple string commands, you should start your program by lowering the BASIC workspace to protect the new character set:

```
10 REM SET HIGH LIMITS = $3800
20 POKE 52,56 : POKE 56,56
30 REM 56 = $38
```

If you want to copy one of the normal character sets to $3800 for modification, a bit of system programming magic is called for. To let the 6510 processor see the character ROM, you have to turn off the regular clock interrupts and deselect the input/output devices:

```
5000 REM COPY THE ROM TO $3800
5010 REM THE CLOCK WILL LOSE TIME!
5020 REM DISABLE IRQ
5030 POKE 56334,PEEK(56334) AND 254
5040 REM SELECT CHARACTER ROM
5050 POKE 1,PEEK(1) AND 251
5060 REM COPY THE WHOLE SET
5070 FOR A = 0 TO 2047
5080 POKE 14336+A,PEEK(53248+A)
5090 NEXT A
5100 REM DESELECT ROM
5110 POKE 1,PEEK(1) OR 4
5120 REM ENABLE IRQ
5130 POKE 56334,PEEK(56334) OR 1
5140 REM NOW TELL VIC TO USE RAM
     CHARACTERS
5150 POKE 53272,(PEEK(53272) AND 240)+14
5160 RETURN
```

This subroutine will take about two minutes since 2048 bytes are being moved. A machine language routine could do this in less than one tenth of a second. When the subroutine has been run, the display should look exactly as it did before. The difference is

that the VIC will be getting the character patterns from RAM. You can test this by altering one of the patterns:

```
FOR A=14344 TO 14351:POKE A,60:NEXT
```

Type this line in directly and you will see all of the "A" characters on the screen instantly turn into vertical bars (a $3C pattern). If you type in the same line but with a 5 rather than a 60, you will be able to see the color interference if you have a color monitor or television.

The subroutine above copies the normal character set to RAM. If you wanted to copy the upper/lower case set, the only thing to change would be the address in line 5080, from 53248 to 55296.

One of the advantages of using your own character set in a graphics game program is that you can change the shapes of any number of occurences of a character on the screen all at once and very quickly. In a game that needed more than eight moving objects, you might choose to use a redefined character set sacrificing small movement increments for speed of movement and animation. For example, if a character was defined to have a small mushroom shape, your "centipede massacre" game could have hundreds of dancing mushrooms just by redefining the shape every so often. With 256 different character shapes to work with, very elaborate graphics and even multiple-character animation is possible.

```
1  REM DANCING MUSHROOMS DEMO
10 GOSUB 500 : REM PRINT SOME "A"S
20 GOSUB 5000 : REM SUBROUTINE ABOVE
30 RESTORE : REM READ DATA FROM LINE
   130
40 FOR D = 1 TO 4
50 FOR E = 0 TO 7
60 READ C
70 POKE 14344+E,C
80 NEXT E
90 REM DELAY HERE OR DO WHATEVER?
100 NEXT D
110 GOTO 30
120 REM 4 CHARACTER PATTERNS
```

```
130 DATA 24,36,66,66,126,24,24,24
140 DATA 0,24,36,66,66,126,48,24
150 DATA 0,24,36,66,66,126,96,48
160 DATA 0,24,36,66,66,126,48,24

500 REM PRINT SOME MUSHROOMS
510 FOR A = 1 TO 500
520 IF RND(1) > .8 THEN 540
530 PRINT "A"; : GOTO 550
540 PRINT " ";
550 NEXT A
560 RETURN

5000 REM SUBROUTINE FROM ABOVE
```

In a machine language version of this game it would be feasible to define five or 10 different characters as mushrooms. Then the changing display would be so complex that the player would not be able to tell that some of the shapes were changing together. It would look like every mushroom had independent movement. The illusion would be even more complete if mushrooms had random colors.

Extended Background Mode

The VIC chip offers a mode for displaying characters with four different background colors along with the usual 16 different character colors. In the VIC registers there are four locations to store background colors. Each register can be set as any of the normal 16 colors. When characters are displayed, the upper two bits of each character are interpreted as the number of the background register to use for that display position. The extended background mode is turned on by setting bit 6 of the VIC control register at 53265:

```
10 REM TURN ON EXTENDED BACKGROUND
20 POKE 53265,PEEK(53265) OR 64
```

Your program also needs to select the four background colors. Background register 0 color will be used as the general screen

background color in screen areas without characters. HINT: You will find that more colors are clean and readable against white or black backgrounds than against other colors.

```
30 REM NOW SELECT FOUR COLORS.
40 POKE 53281,0 : REM REGISTER 0 = BLACK
50 POKE 53282,2 : REM REGISTER 1 = RED
60 POKE 53283,4 : REM REGISTER 2 = PURPLE
70 POKE 53284,5 : REM REGISTER 3 = GREEN
```

Now if you POKE display character codes to the text screen, the top two bits of each code will determine the background color. The normal color RAM at 55296 will determine the character color. The compromise made in this mode is that with only 6 bits of character code to define the character, only the bottom 64 characters of the character set may be displayed.

The easiest way to use this mode is to POKE display codes directly to the screen. The background register that you use can be selected by ORing the display code with one of four numbers:

```
1200 REM DISPLAY CHARACTER C
1210 REM AT POSITION (X,Y)
1220 REM WITH BACKGROUND B = 0-3
1230 REM AND CHARACTER COLOR S
1240 A = 1024 + X + 40*Y
1250 POKE A,(C + 64*B)
1260 POKE A+54272,S
1270 RETURN
```

For a demonstration of this mode, add the following lines to all of those already given above and run the program. No matter how good your color television or monitor is, you will see combinations of background and character colors that are difficult or impossible to read:

```
100 REM GENERATE SOME CHARACTERS
110 FOR C = 1 TO 63
120 S = C AND 15
130 B = INT(C/16)
140 Y = INT(C/32 + 4)
```

```
150 X = C AND 31
160 GOSUB 1200
170 NEXT C
180 END
```

Displaying characters in this mode is not supported by BASIC. If you print a string, it will be displayed with the background color coming from background register 0, as in the normal text mode. If you print strings containing graphics characters (above display code 63) they will instead be displayed as one of the bottom 64 display characters with a different background color.

The results of printing a particular string or CHR$( ) can be predicted only by comparing the two tables of CHR$ codes and display codes and deducing what the upper two bits will contain. With some experimentation you can type in strings using the <SHIFT> and <COMMODORE> keys to set the background color. Don't forget to turn on the extended background mode before you try this.

Extended background mode will work fine with your own redefined character set. Keep in mind though that you would only have 64 characters to define.

Multi-Colored Character Mode

There is another character graphics mode available that is mutually exclusive of extended background mode. In multi-colored character mode, pairs of character pattern bits are used to specify one of the background color registers or the normal color memory as the source of color information. This is one of the most complicated modes of the VIC, and is not supported well by BASIC. This mode has little value used with the standard character sets. You may want to copy part of a standard set, because your program will still be able to print normal character strings in any of eight colors, but to make use of multi-colored characters you will have to create some new character patterns.

Multi-colored character mode is turned on by setting bit 4 of $D016:

```
10 REM TURN ON MULTI-COLOR MODE
20 POKE 53270, PEEK(53270) OR 16
```

If you type in these lines and run them, the light blue characters on your display will immediately change into strange multi-colored shapes. If you press CTRL-5 to change the text color to purple, you will be able to list the lines in normal text characters. Here's what is happening: This mode is selected or deselected on a character by character basis by bit 3 of the color RAM. By default, the text color is set to light blue when the Commodore 64 is turned on. Light blue is color code 14 or binary 1110. Bit 3 of bytes of color RAM containing the light blue code is on, so light blue characters are displayed in multi-color mode. Conversely, the code for purple is 4 or binary 0100, so purple characters are displayed normally. Characters in any of the first eight colors will be displayed normally. These are the colors that you can select using the control key and one of the number keys.

Normal	Character Colors
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow

In color memory or in VIC registers, these codes are the same. You may have noticed that the colors above are in the same order as the colors on the top row of the keyboard, but offset by one. This makes it easy to remember the codes.

When you print a character in one of the upper eight colors, or POKE color RAM with a color code above 7, the lower three bits are interpreted as one of the colors above and the character is dis-

played in multi-colored mode. In this mode there are only four pixels horizontally by eight pixels vertically per character position. Each pixel can have one of four colors. A pair of bits in the character set selects the source for the color information for each pixel. Remember how character shapes were defined earlier? In the multi-colored mode they are defined with less shape information and more color information:

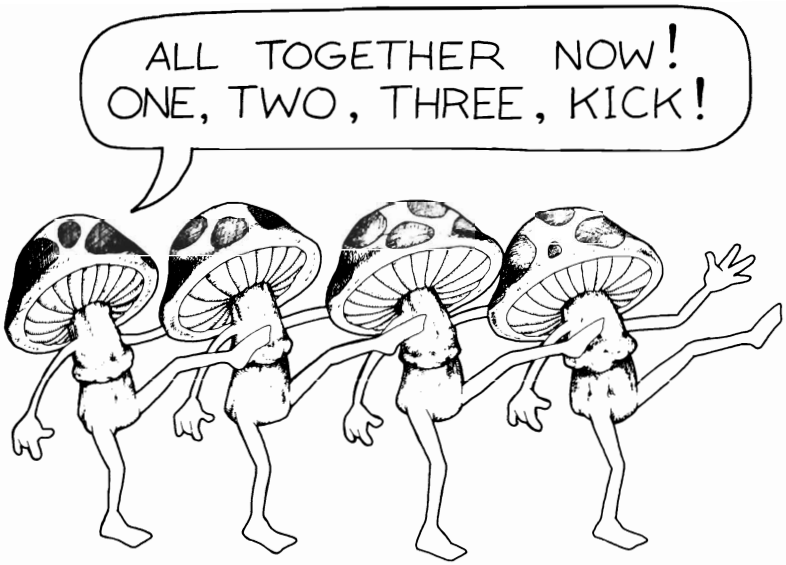
Character Definition

0 --- Use color in background register 0
1 --- Use color in background register 1
2 --- Use color in background register 2
3 --- Use color in lower 3 bits of color RAM

Shape	Binary	Hex	Decimal
0 1 1 0	00 01 01 00	14	20
1 2 2 1	01 10 10 01	69	105
1 2 2 1	01 10 10 01	69	105
0 3 3 0	00 11 11 00	3C	60
0 3 3 0	00 11 11 00	3C	60
0 3 3 0	00 11 11 00	3C	60
0 3 3 0	00 11 11 00	3C	60
0 0 0 0	00 00 00 00	00	0

The mushroom shape above could, for example, have a white cap with yellow in the center and a green stem, all on a black background with the following register settings:

```
10 POKE 53281,0 : REM R0 = BLACK
20 POKE 53282,1 : REM R1 = WHITE
```




```
30 POKE 53283,7 : REM R2 = YELLOW
40 PRINT CHR$(30) : REM GREEN
```

Anytime the character defined above was printed, the three-colored mushroom would be displayed. Notice that the color of the stem is selected by the contents of the color RAM. Mushrooms with different colored stems could be printed by shifting the text color at any time. Beware of trying to print mushrooms with the text color set to one of the lower eight, as the mushroom bit patterns would then be interpreted as normal monochrome character bits. All of the mushroom caps could be changed in color by storing new color codes in background registers 1 and 2. Of course, the screen background color could be changed at any time without disturbing the mushroom colors by storing a new color code in background register 0. Keep in mind that some colors are very difficult to see against other colors, so screen backgrounds (register 0) of black or white are preferable. Also, this mode can generate color dot information with such a high bandwidth that most color monitors (including Commodore's) are unable to

resolve individual colors. This problem can be lessened considerably by only using black or white and one other color per horizontal line of pixels.

Here's an alternate method for translating multi-color shape patterns into decimal numbers for those of us who can't translate binary into hex into decimal in our heads. To translate a row of bit pairs into decimal, multiply each pair's code (0-3) by the less significant bit's decimal equivalent and add the results:

```
0 1 1 0
. . . .
. . . 0.times 1 = 0
. . 1...times 4 = 4
. 1.....times 16 = 16
0.......times 64 = 0
                  ----
        sum       = 20

1 2 2 1
. . . .
. . . 1.times 1 = 1
. . 2...times 4 = 8
. 2.....times 16 = 32
1.......times 64 = 64
                   ----
        sum       = 105

0 3 3 0
. . . .
. . . 0.times 1 = 0
. . 3...times 4 = 12
. 3.....times 16 = 48
0.......times 64 = 0
                   ----
        sum       = 60
```

Here is a complete program that prints some of the multi-colored mushrooms:

```
5 REM MULTI-COLORED CHARACTERS DEMO
10 POKE 53281,0 : REM REG 0 = BLACK
```

```
20 POKE 53282,1 : REM REG 1 = WHITE
30 POKE 53283,7 : REM REG 2 = YELLOW
40 REM TURN ON MULTI-COLOR MODE
50 POKE 53270,PEEK(53270) OR 16
60 REM SELECT $3800 FOR CHARACTER SET
70 POKE 53272,(PEEK(53272) AND 240) OR 14
80 FOR A = 14344 TO 14351
90 READ D
100 POKE A,D : REM DEFINE DISPLAY CODE 1
110 NEXT A
120 FOR A = 14592 TO 14599
130 READ D
140 POKE A,D : REM DEFINE DISPLAY CODE 32
150 NEXT A
160 PRINT CHR$(30) : REM GREEN SHIFT
170 PRINT CHR$(147);"A A A A A"
180 INPUT R$ : REM WAIT FOR RETURN
190 POKE 53272,(PEEK(53272) AND 240) OR 4
200 REM MUSHROOM CHARACTER DATA
210 DATA 20,105,105,60,60,60,60,0
220 REM BLANK CHARACTER DATA
230 DATA 0,0,0,0,0,0,0,0
240 END
```

This graphics mode has the highest potential for very fast, complex, arcade-style video games written in machine language. With sprites for moving objects, multi-colored characters for appearing/disappearing objects, and the excellent sound effects and music of the SID chip, commercial quality games can easily be written for the Commodore 64.

Chapter 6

Bit-Mapped Graphics

Bit-mapped graphics mode is the highest resolution graphics mode available with the VIC chip. The term "bit-mapped" signifies that every pixel of the display has a controlling bit in memory. 64000 bits of memory are displayed on the screen at one time. Bits that contain a zero have the background register 0 color, while bits that contain a one are displayed with a color code from the color RAM. Most personal computers with a high resolution graphics mode permit only one color against black or white. The Commodore 64 hi-res mode permits your program to color the pixels of every 8 by 8 screen area in a different color by getting the color information from another 1000 byte area of RAM called the "color matrix."

Bit-mapped displays are most useful for programs that draw lines or large objects. While character graphics require objects to be placed at one location of a 40 by 25 grid, bit-mapped objects may be drawn anywhere on a 320 by 200 grid. With the proper software, bit-mapped objects can even be drawn partly off screen.

Again that term "proper software" comes up. Commodore's BASIC has no commands for supporting these "high-res" graphics, so everything has to be done with POKEs or machine language routines. Nevertheless, use of the bit-mapped mode is straightforward and can yield images with incredible detail. The drawbacks to using the bit-mapped mode are that it takes a long time to move complex objects around the screen, and a lot of memory is required to store the bits. It takes 8000 bytes of memory to be exact, plus another 1000 bytes for the color information. This 9000 bytes has to be in the 16K area of memory that the VIC is using, so you can see that the biggest problem is figuring out where to put your program. For now let's still work in the

bottom 16K of memory. In any 16K area, there are two possible choices for the bit-map. With almost 8K used for the bits, there are eight possible choices for the 1K color matrix. If we want to use a BASIC program to generate the hi-res displays, it will have to fit in there too. Here is a possible memory map for using the bottom 16K:

Memory Map for Bit-Mapped Graphics

$0000-03FF	BASIC and OS use
$0400-07E7	Text screen and color matrix
$0800-1FFF	BASIC program workspace
$2000-3FFF	Bit-map area

This memory allocation gives us only 6K of RAM for BASIC programs. Use of other areas of memory will be discussed later. In order to ensure that your program doesn't overwrite the graphics area, you will need to tell BASIC that $1FFF is the top of RAM.

```
10 REM SET NEW TOP OF RAM
20 POKE 56,31 : POKE 52,31
30 REM 31 DECIMAL = $1F
```

The next thing that a program has to do is tell the VIC chip that the upper half of the memory bank is to be used for the bit-map. The register in the VIC that controls the bit-map location also controls which of the possible color matrix areas to use, so your program can set both at once:

$D018 VIC Memory control register

bits 7-4	Selects which 1K block of memory is to be used for the video matrix (text or color information)
bit 3	Selects which 8K block to use for the bit-map.

```
bit  7 6 5 4 3 2 1 0
     0 0 0 1 1 0 0 0 = $18 = decimal 24
```

```
40 REM SET VIC MEMORY CONTROL REG
50 POKE 53272,24
```

Next, the program has to actually turn on the bit-mapped mode:

```
60 REM TURN ON BIT-MAPPED MODE
70 POKE 53265,PEEK(53265) OR 32
```

If you run the program lines given so far, your text display will instantly change to several multi-colored blocks on a screen of mostly red and black lines. In the bit-mapped mode, what was the text screen is being interpreted as color information. The text screen normally contains mostly blanks or $20. The upper four bits set the foreground color to 2 (red), while the lower four bits set the background color to 0 (black). Every byte of the former text screen memory that contains a blank specifies that red pixels on a black background are to be used for an 8 by 8 pixel square in the corresponding position on the high-res display. If you figure it out mathematically, you will find that the 320 by 200 hi-res display is made up of 40 by 25 blocks of 8 by 8 pixels each. So each byte of former text display memory will now control that SAME block's color, only the pixel information will now come from the bit-map instead of a character set. Once you understand this, the rest is easy.

The next thing that our program needs to do is to set up the color information. We could use a loop to POKE the color codes into each byte, or several loops if the color scheme were more complex, but for now let's make use of the fact that BASIC can quickly set this entire area of memory to $20 by clearing the screen:

```
80 REM SET COLORS = RED/BLACK
90 PRINT CHR$(147)
```

The last part of setting up a bit-mapped hi-res display is to erase the bit-map to prepare a "clean slate" for the rest of the program. The easiest way to do this is with a BASIC loop:

```
100 REM CLEAR BIT-MAP
110 FOR A = 8192 TO 16191
120 POKE A,0
130 NEXT A
```

If you run the program lines so far, you will find that it takes more than 20 seconds to erase the whole bit-map. If you run it again, it won't seem to do anything because the RAM is already filled with zeros. This is an ideal task for a machine language program. Here is a sample program that will run anywhere in memory that you POKE or LOAD it:

```
FHGR    LDA #$3F     ; MSB OF LAST HGR PAGE
        STA $FC      ; USE $FB,FC AS A POINTER
        LDA #$00     ; POINT TO WHOLE PAGES
        STA $FB
        LDY #$3F     ; HIGHEST ADDRESS IN LAST
                       PAGE
        LDA #$00     ; FILL PATTERN
        LDX #$20     ; # OF PAGES TO FILL
```

```
FILL    STA ($FB),Y
        DEY
        CPY #$FF     ; CROSSED PAGE BOUNDARY?
        BNE FILL
        DEC $FC      ; POINT TO NEXT PAGE
        DEX          ; COUNT PAGES
        BNE FILL     ; LOOP 32 TIMES
        RTS
```

This machine language routine will erase the bit-map in less than a thirtieth of a second. Whichever way you choose to erase the display, your program is now ready to start drawing lines or objects on the screen. We know from the BASIC lines above that 16191 is the address of the last byte of the bit-map. If the program POKEs something into 16191, it will show as red dots:

```
200 REM DISPLAY A SHORT LINE
210 POKE 16191,255
```

A short red line should appear in the lower right corner of the display.

```
220 POKE 8192,255
```

Now, another short red line should appear in the upper left corner of the display. To change the color of one of these lines, you would need to POKE a new pair of color codes into the color matrix:

```
230 POKE 1024,5*16 : REM GREEN/BLACK
```

Sure enough, the upper left line is now green. So much for the simple screen locations, but how do we draw something in the middle? The bit-map memory corresponds to the screen pixels in "chunks" of eight bytes. The upper left corner of the screen is the first byte of the bit-map, with bit 7 toward the left edge of the screen. The next seven bytes of the bit-map are directly below the first byte. The ninth byte is back at the top of the screen, just to the right of the first byte, just the way the character graphics work. The whole top row of 8 by 8 blocks takes 320 bytes, so the 321st byte is displayed directly below the eighth. This may seem

a bit complicated, but actually it is much simpler than some other personal computers that only display seven out of eight bits and scramble the addresses completely. It also makes it quite simple to print text characters on the high-res display. More about this later. For now, what we need is a simple algorithm for calculating the address and bit of any screen location. Fortunately, this is rather easily done:

```
1000 REM GIVEN X,Y PLOT A POINT
1010 ROW = INT(Y/8)
1020 CHAR = INT(X/8)
1030 LINE = Y AND 7
1040 BIT = 7-(X AND 7)
1050 BASE = 8192
1060 BYTE = BASE + ROW*320 + CHAR*8 +
     LINE
1070 POKE BYTE,PEEK(BYTE) OR 2 ^ BIT
1080 RETURN
```

Now we can add a few more lines to our program and we'll get to see some real plotting. A simple straight line plot will do for now:

```
300 FOR X = 0 TO 100
310 Y = X
320 GOSUB 1000
330 NEXT X
```

Finally, these lines can be used to restore normal operating conditions at the end of your program so that you won't have to use RUN-STOP RESTORE to start running again:

```
500 INPUT R$          : REM WAIT FOR RETURN
510 POKE 53272,23 : REM RESTORE MODE
520 POKE 53265,27 : REM RESTORE MEMORY
     CONTROL
530 END
```

If you have not already entered and run the program above, I encourage you to do so now since it will serve as a good starting point for writing your own bit-mapped hi-res programs. Now

that you've really run the program, you may have noticed that the line was drawn about as fast as a snail could race across your screen. What happened to the half a million operations per second speed of the 6510 microprocessor? The problem is that all of this address calculation, point plotting, and even line and shape drawing should be in machine code instead of BASIC. Of course, that would require a much larger subroutine than our bit-map eraser. Here is an assembler version of a routine that will plot (light) or unplot (erase) any point on the screen. It is very fast and could easily be expanded to draw straight lines:

```
; BIT-MAP PLOTTER SUBROUTINE
;
; POKE X VALUE (LOW)
;        X VALUE (HIGH)
;        Y VALUE
;        MSB OF BIT-MAP
; CALL PLOT OR UNPLOT
;
     *=$C000
POINT=$FB
TEMP =$FD

CALCA   LDA #$00        ; <--POKE X(LOW) HERE
        AND#$F8
        STA TEMP
        LDA #$00
        STA POINT+1
YPOKE   LDA #$00        ; <--POKE Y HERE
        AND#$F8
        STA POINT
        ASL POINT       ; = Y AND $F8
        ROL POINT+1
        ASL POINT       ; = (Y AND F8)*2
        ROL POINT+1
        CLC             ; = (Y AND F8)*4
        ADC POINT
        STA POINT
        LDA #$00
        ADC POINT+1
        STA POINT+1     ; = (Y AND F8)*5
        LDY #$03
```

```
LOOP8   ASL POINT       ; POINT*8 LOOP
        ROL POINT+1
        DEY
        BNE LOOP8       ; = (Y AND F8)*40
        LDA YPOKE+1
        AND #$07
        CLC
        ADC TEMP
        ADC POINT
        STA POINT
        LDA #$00
        ADC POINT+1
XPOKE   ADC #$00        ; <-- POKE X(HI) HERE
BPOKE   ORA #$20        ; <-- POKE BASE MSB HERE
        STA POINT+1     ; ADDRESS IS READY
        LDA CALCA+1
        AND #$07
        TAY
        LDA TABLE,Y     ; GET BIT PATTERN
        LDY #$00
        RTS
;
PLOT    JSR CALCA       ; CALL THIS FROM BASIC
        ORA (POINT),Y
        STA (POINT),Y
        RTS
;
UNPLOT  JSR CALCA       ; CALL THIS FROM BASIC
        EOR #$FF        ; REVERSE BIT PATTERN
        AND (POINT),Y
        STA (POINT),Y
        RTS
;
TABLE .BYTE $80,$40,$20,$10
      .BYTE $08,$04,$02,$01
;
      .END
```

Even using this subroutine, you will find that drawing is still slow when BASIC has to calculate Y = f(X) for every point. Straight lines can be drawn very quickly by adding another machine

language routine to calculate each pair of points and repeatedly call the plotting subroutine. Another intermediary subroutine that might be useful would be a "turtle graphics" interpreter. A BASIC program (or programs) using such an interpreter could make SYS calls to move a "turtle" (a single point cursor) around the hi-res display, drawing lines, rotating, or changing color very quickly. Many implementations of BASIC support turtle graphics because they tend to encourage shape drawing (fast) versus function plotting (slow).

Labeling Your Graphics

Due to a lot of foresight on the part of the Commodore engineers, it is very simple to label bit-mapped graphics with normal text characters. The bit patterns of individual characters in a character set occupy eight consecutive bytes. The bit patterns that represent one 8 by 8 block of bit-mapped display also happen to occupy eight consecutive bytes. To place a character on the bit-mapped display, simply move the eight bytes of the character definition from the character set to an address in the bit-map that is divisible by eight.

If your program is using a redefined character set, this will be a straightforward memory-to-memory move. If your program is using the default character set, it will also have to perform the switching given in the section on downloading the standard character set, once for each character displayed. A machine language routine to print characters on the hi-res display would not be difficult to write. It could even keep track of the next location to display to automate printing character sequences.

With a little more ambition, you could even intercept the CHROUT vector at $0326 so that all BASIC output would go to your hi-res character printer.

Note: If you try this, make your interception routine jump to the old screen output routine when it is done putting the character in the bit-map, otherwise BASIC will blow up! Also keep in mind that BASIC output to the text screen can interfere with use of that same memory area for the color matrix.

Multi-Color Bit-Mapped Graphics

If you write a program to plot colored lines, you will eventually find a major problem with bit-mapped mode: Since the color is controlled in 64 pixel blocks, two lines of different colors cannot intersect. When the second line is drawn and the byte of the color matrix set to that line's color, the color of a segment of the first line will change too. This severely limits the use of multiple colors in the bit-mapped mode. To draw a particular high resolution picture in multiple colors, you must painstakingly plan the color layout. To use bit-mapped mode as a general utility without this planning, you must only use one color.

Naturally, we can compromise. By selecting multi-colored bit-map mode, we trade horizontal resolution for individual pixel color control. This is done very much like multi-colored character mode. In fact, there is only one bit in the VIC for multi-color; it selects the color mode for either graphics or text.

When multi-colored bit-map mode is selected (let's call it MCBM, okay?), each pair of bits in the bit-map determine the source of the color information for a single pixel, giving a graphics resolution of 160 horizontal by 200 vertical. All of the address mapping, (x,y) to address conversion, and so forth is the same except that two bits must be set to plot a point instead of one. With two bits to control the color source, there are four possible combinations:

MCBM mode color control

Bit code	Source of color information
00	Background register 0
01	Upper 4 bits of color matrix
10	Lower 4 bits of color matrix
11	Color memory ($D800 RAM)

This mode lets you have all 16 colors on the screen at once with up to three intersecting lines without interference. For use as a general utility the four color sources should all be set in advance and then only those four colors used. For example, with the back-

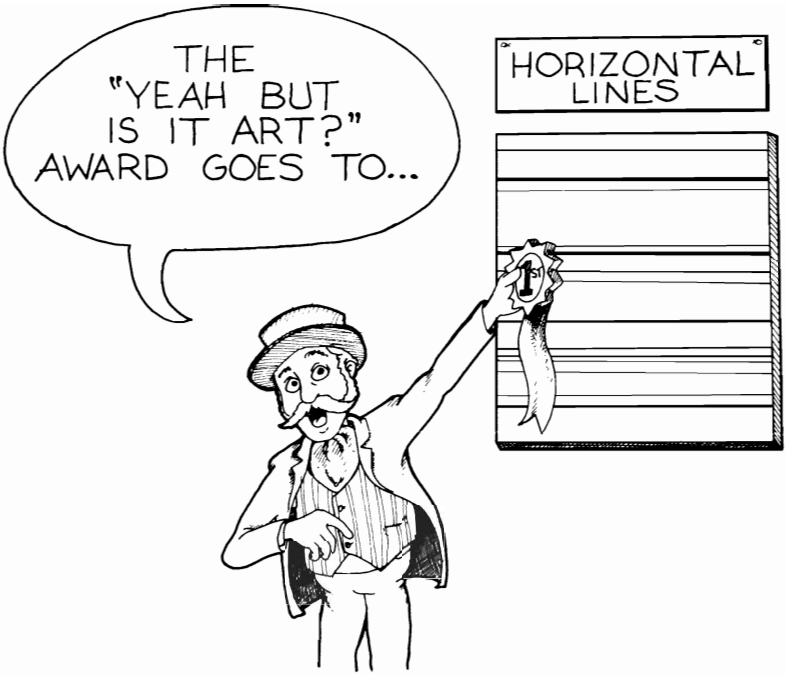
ground 0 register set to 0 (black), the color matrix filled with $25 (red/green), and the color RAM at $D800 filled with $06 (blue), then red, green, or blue points could be plotted on a black background just by selecting the proper bit patterns.

Sacrificing half of the horizontal resolution for this generality is not much of a loss, since most color monitors or television sets can't resolve a single high-res pixel anyway. Another advantage of MCBM mode is that there is little color interference since pairs of pixels always have the same color.

To select MCBM mode, set up the VIC memory register as in the last section, turn on bit-mapped mode by setting bit 5 of $D011 (53265) to 1, and then turn on multi-color mode by setting bit 4 of $D016 (53270) to 1. Set up the background register at $D021 (53281), the color matrix at $400 (1024-2023), and the color RAM at $D800 (55296-56295) with color codes. If you use the machine code subroutine given in the last section, you can just call it twice for each point that you want to plot. In this example, <xlow>, <xhi>, <y>, <unplot>, and <plot> must each be replaced by the actual addresses used by the assembled subroutine.

```
4000 REM PLOT (X,Y) IN COLOR C
4010 REM X = 0 TO 159
4020 REM Y = 0 TO 199
4030 REM C = 0 TO 3
4040 POKE <XLOW>,(X*2) AND 255
4050 POKE <XHI>,INT(X/128)
4060 POKE <Y>,Y
4070 IF C AND 2 = 0 THEN SYS <UNPLOT>
4080 IF C AND 2 = 2 THEN SYS <PLOT>
4090 POKE <XLOW>,PEEK(<XLOW>) OR 1
4100 IF C AND 1 = 0 THEN SYS <UNPLOT>
4110 IF C AND 1 = 1 THEN SYS <PLOT>
4120 RETURN
```

Here is a quick demo of MCBM mode plotting. The bit-map erase is written in BASIC, so be patient. The program turns on MCBM first so that you can see all of the memory erasing. The bit map is erased first, then the color matrix, and finally the color RAM. The plotting is done directly to the bit map without (x,y) translation:




```
10 REM SELECT BIT-MAPPED MODE
20 POKE 53265,PEEK(53265) OR 32
30 REM SELECT MULTI-COLORED MODE
40 POKE 53270,PEEK(53270) OR 16
50 REM SET UP MEMORY REGISTER
60 POKE 53272,24
70 REM SET BACKGROUND REGISTER
80 POKE 53281,0 : REM = BLACK
90 REM SET BORDER REGISTER
100 POKE 53280,0 : REM = BLACK
110 Z = 0 : REM DECLARE Z FOR SPEED
120 FOR A = 8192 TO 16191
130 POKE A,Z
140 NEXT A
150 FOR A = 1024 TO 2023
160 POKE A,37
170 NEXT A
180 REM 37 = $25 = RED/GREEN
190 FOR A = 55296 TO 56295
```

```
200 POKE A,6
210 NEXT A
220 REM 6 = $06 = BLUE
230 REM NOW PLOT THREE COLOR LINES
240 FOR A = 8192 TO 16191
250 POKE A,85
260 POKE A+2,170
270 POKE A+4,255
280 NEXT A
290 GOTO 290
300 REM 85 = 01 01 01 01 BINARY
310 REM 170 = 10 10 10 10 BINARY
330 REM 255 = 11 11 11 11 BINARY
```

Note that this program plots the three colored lines horizontally. If you change the program a bit, you can get it to plot them vertically. If you plot them vertically directly adjacent, your monitor or television will probably display them as a single white band, as mine does. Separate each pair of colored lines with a black line and they will be much easier to see.

```
240 REM ALTERNATE TO PROGRAM ABOVE
250 REM DRAW THREE VERTICAL LINES
260 FOR A = 8192 TO 16191 STEP 320
270 FOR B = A TO A+7
280 POKE B,8 : POKE(B+8),76
290 NEXT B
300 NEXT A
310 GOTO 310
```

Chapter 7

Advanced Graphic Techniques

Sprites

Users of bit-mapped graphics systems have always had one common problem: Moving a large object about the display is too slow because of the processor time required to erase and then redraw all of the bits making up the object. Another problem is that the program must "remember" the background underneath a bit-mapped object so that it can be restored once the object is moved. With a rather slow 8-bit processor like the 6510, these problems make it difficult to write programs moving complex objects over complex backgrounds with acceptable speed. The MOS division of Commodore came up with a solution for the VIC chip: movable object blocks, or "sprites."

A sprite is a graphic object which the VIC chip can display "over" the background image. The background image described by the bit map or text screen remains intact. When the sprite is moved, the background image beneath is displayed as it was before the sprite was there. The single most important feature of a sprite is that it can be moved on the display by changing only it's X or Y position in VIC registers. The VIC chip can display up to eight sprites at once. In the VIC chip, each sprite has an X position register, an X position high bit, a Y position register, a color register, an X expansion bit, a Y expansion bit, and a multi-color enable bit as follows:

Sprite control registers

Enable register	53269	$D015
X expansion	53277	$D01D
Y expansion	53271	$D017

Controlling sprite positions

	High bit of X	53264	$D011

	Sprite	X position	Y position
0	$D000	$D001	
	1	$D002	$D003
	2	$D004	$D005
	3	$D006	$D007
	4	$D008	$D009
	5	$D00A	$D00B
	6	$D00C	$D00D
	7	$D00E	$D00F

Controlling sprite priority

Sprite to sprite	Automatic	
Sprite to background	53275	$D01B

For each of the single registers above, sprites 0-7 are controlled by setting the corresponding bit 0-7 in the register.

Programs can move sprites by altering the contents of the appropriate register, incrementing to move down or right, decrementing to move up or left. Note that with more than 256 possible display positions horizontally, nine bits are required to specify the X position of each sprite.

Controlling sprite colors

Sprite	Color	Register
0	53287	$D027
1	53288	$D028
2	53289	$D029
3	53290	$D02A
4	53291	$D02B
5	53292	$D02C
6	53293	$D02D
7	53294	$D02E

Each sprite has its own color register in the VIC chip, so each can be any of the 16 possible colors. Your program can change a sprite's color anytime you like by just POKEing in a new color code.

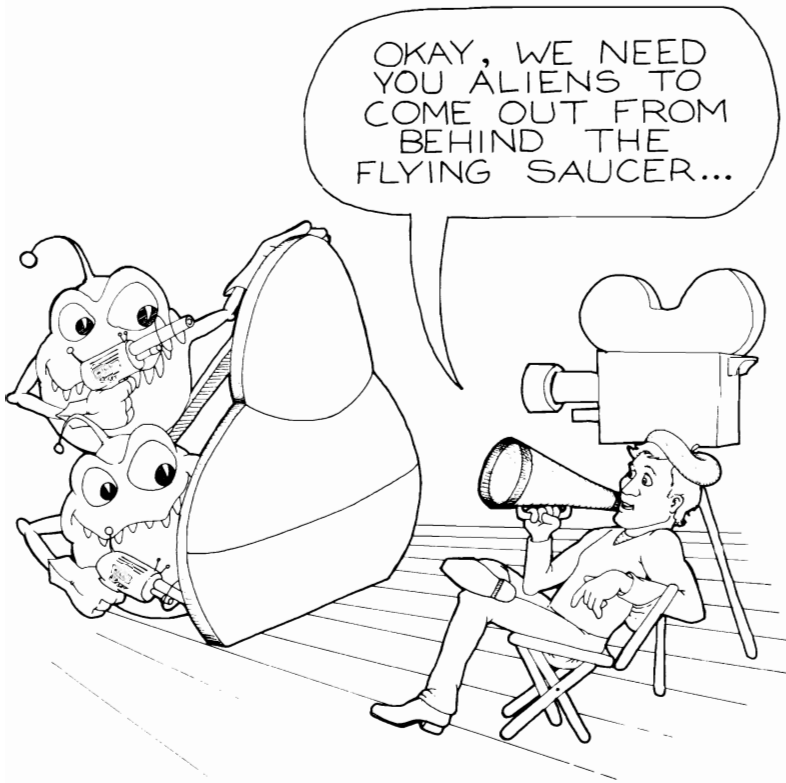
A sprite's shape is described just like every other graphic object in the Commodore 64, as a sequence of bit patterns in memory. Each sprite definition requires 63 consecutive bytes of RAM to make up the 24 by 21 dot shape. Each bit in the description of a regular sprite is either a 1 (colored) or a 0 (transparent). The background image will show through transparent portions of a sprite. The RAM bytes correspond to the sprite image if you lay them out like this:

Sprite definition: Flying Saucer

bit ------> 765432107654321076543210

byte 0 ---> 000000000000000000000000
byte 3 ---> 000000000011110000000000
byte 6 ---> 000000000111111000000000
byte 9 ---> 000000001111111100000000
byte 12 --> 000000011111111110000000
byte 15 --> 000000011001100110000000
byte 18 --> 000000011001100110000000
byte 21 --> 000111111111111111111000
byte 24 --> 111111111111111111111111
byte 27 --> 111111111111111111111111
byte 30 --> 000000001000010000000000
byte 33 --> 000000001000000100000000
byte 36 --> 000000010000000010000000
byte 39 --> 000001111100000111100000
byte 42 --> 000000000000000000000000
byte 45 --> 000000000000000000000000
byte 48 --> 000000000000000000000000
byte 51 --> 000000000000000000000000
byte 54 --> 000000000000000000000000
byte 57 --> 000000000000000000000000
byte 60 --> 000000000000000000000000

Once you have designed a sprite, you must get the pattern into memory somehow. To begin with, you may do this by translating each row into three decimal numbers, and then POKEing the 63 numbers into RAM with READ and DATA statements. For instance, the row describing bytes 3, 4, and 5 above would translate to 0, 60, 0 respectively. Maybe you enjoy this sort of thing, but after a few sprites I started looking for a better way.



One of the best ways of creating sprites is to use one of several commercially available sprite editor programs. These are available from several program authors and companies, including Commodore. Invariably they let you actually edit a sprite on the screen, pixel-by-pixel rather than by fiddling endlessly with numbers. Output from these editors may be in the form of decimal numbers for READ . . . DATA statements, as sequential files

for an INPUT# ... POKE loop, or as "machine code" files for direct memory loading. Some of them will even append lines of DATA statements onto the end of your BASIC programs. If you can't find one of these editors, or if you just prefer to work in assembly language, you can use any assembler to create sprites as loadable files:

```
; THIS IS A SPRITE DEFINITION
* = <SPRITE ADDRESS>
.BYTE %00000000,%00000000,%00000000
.BYTE %00000000,%00011000,%00000000
(and so forth for 63 bytes total)
```

Here the "%" symbol is used as the symbol for a base 2 constant. Your assembler may use a different symbol. Read the manual.

Each sprite takes up 63 bytes of memory somewhere in the VIC chip's 16K bank. For convenience, these must be on even 64 byte boundaries. Therefore, it is possible to have 256 sprite definitions in RAM at once, even though only eight are displayed at a time. This ignores the fact that the VIC needs a text screen area or color matrix and maybe an 8K bit map, not to mention RAM used by BASIC or the operating system.

Sprite Areas in the First Bank

Area number	**Address**	**Use**
0	0-63	BASIC
1	64-127	BASIC
2	128-191	BASIC
.	.	BASIC
10	640-703	BASIC
11	704-767	okay
12	768-831	Operating system
13	832-895	okay
14	896-959	okay
15	1024-1087	Text screen
.	.	
31	1984-2047	Text screen
32	2048-2111	BASIC workspace
.	.	BASIC workspace
255	16320-16383	BASIC workspace

As you can see, only areas 11, 13 and 14 are free for storing sprite patterns. Area 11 is not used by any system routines, while areas 13 and 14 are only used as a tape I/O buffer. More sprite areas can be reserved in the 16K bank by moving the BASIC workspace up or by limiting it to below 16K. If you want to reserve a bit-map, new character set, color matrix, and so forth, you may be left with no room for BASIC workspace. There are other solutions to this problem which involve using other banks of RAM for the VIC. These will be covered in a later section of this chapter. For now, let us just use the four free areas in the bottom 16K bank for examples.

To tell the VIC chip which of the 256 possible sprite areas to use for the eight displayable sprites, you must store the area numbers at the "sprite pointers" at 2040-2047. To display a sprite pattern in area 11 as sprite 0, store an 11 at 2040. To display a sprite from area 14 as sprite 7, store a 14 at 2047. Which sprite is which is important since lower numbered sprites have higher display priority over higher numbered sprites. If your program displays two sprites overlapping, the lower numbered sprite will be "in front" of the higher numbered sprite.

```
10 REM A SIMPLE SPRITE DEMO
20 POKE 2040,11 : REM SPRITE 0 AT 704-767
30 POKE 53269,1 : REM ENABLE SPRITE 0
40 POKE 53287,5 : REM SPRITE 0 COLOR
50 REM NOW DEFINE SPRITE
60 FOR A = 704 TO 766 STEP 2
70 POKE A,0 : POKE A+1,255
80 NEXT A
90 POKE 53264,0 : REM X POSITION HIGH BITS
100 REM NOW MOVE THE SPRITE AROUND
110 FOR C = 0 TO 225
120 POKE 53248,C : REM X POSITION
130 POKE 53249,C : REM Y POSITION
140 NEXT C
150 END
```

If you run this program, a large green object will move across the display from the upper-left corner and stop in the lower-right portion of the screen. You can then list the program, edit, load a

new program from disk, or whatever; the sprite will stay where it is until you move it elsewhere or disable it. Unlike extraneous text characters, it does not interfere with the input of BASIC lines because it is not really in the screen memory. Once it is on the screen, you can get an idea of the way sprite editors work by POKEing new values into the sprite pattern from 704 to 766. Or try changing the color by POKEing 53287. If the movement was too slow, try changing line 110 to:

```
110 FOR C = 0 TO 225 STEP 4
```

Each sprite also has an X expansion bit in the VIC at 53277, and a Y expansion bit at 53271. Try POKEing a 1 (for sprite 0) into each of these registers. When you POKEd the Y expansion register with a 1, the sprite expanded vertically so that part of it was off the screen. In fact there are positions at both sides, the top, and the bottom of the screen where sprites are not visible. This is a convenient way to hide a sprite temporarily.

Reading Sprite Collision Information

Sprite to Sprite	53278	$D01E
Sprite to Background	53279	$D01F

Your program can interrogate these registers to find out if any sprite is colliding with another sprite or with the background image. But be careful here. These registers are only set when the video scan actually displays the sprites of interest. This means that a fast moving sprite could pass all the way through another without the collision being detected! If you really need to detect collisions, you would be better off using arithmetic comparisons of their X and Y position registers, especially if you are moving them quickly.

Sprite to background priority is controlled by the register at 53275 ($D01B). By setting bits to 1 in this register you can make sprites that can hide behind the background image. Try POKEing 53275 with a 1 and then running the last program example. If you have some program lines on the display, you will see the sprite move "behind" them. You can get some interesting effects by combining the use of the automatic priority of lower numbered sprites with individual control of sprite to background priority.

Using Interrupts with Sprites

There are two ways that you can use interrupts with sprites. Sprite to sprite, or sprite to background collisions can generate interrupts if you enable either by setting the proper bit in the VIC IRQ enable register at 53274 ($D01A).

Collision Interrupts

Sprite to Sprite	set bit 2 = 1
Sprite to Background	set bit 1 = 1

Your machine code program must intercept the normal IRQ vector, check the corresponding bits in the VIC IRQ register at 53273 ($D019), do something about the collision, clear the IRQ register, and then jump to the old IRQ routine. The same precautions given above about collision detection apply. Using interrupts implies at least a fast, professional BASIC program, if not a program written entirely in assembly language. Trying to use the VIC chip's hardware collision detection would be a mistake.

The other, more useful, way to use interrupts with sprites is to make use of the regular sixtieth of a second clock interrupts to turn sprites into "missiles" that move automatically. An IRQ interception routine to add or subtract a certain amount from a sprite's position registers each time it is called is not difficult to write. Here is an example of such a routine that will make sprite 0 move up the screen continually, even while you edit or list a program!

```
        ; POINT THE OLD VECTOR TO INT
        * = 49152     ; $C000
PATCH   SEI           ; DISABLE IRQ
        LDA #$10
        STA 314       ; VECTOR ADDRESS
        LDA #$C0
        STA 315
        CLI           ; ENABLE IRQ
        RTS
        ;
```

```
        ; INTERCEPTION ROUTINE
        * = 49168 ; $C010
INT     DEC $D001 ; SPRITE 0 Y POSITION
        JMP 59953 ; OLD IRQ VECTOR
        ;
        ; USE SYS 49152 TO ENABLE THIS ROUTINE
```

You can do the same from BASIC with the following subroutine. It assumes that you have a sprite left on the screen from the last program:

```
1000 REM POKE THE INTERCEPTION ROUTINE
1010 FOR A = 49152 TO 49157
1020 READ D
1030 POKE A,D
1040 NEXT A
1050 DATA 206,1,208,76,49,234
1060 REM DISABLE IRQ
1070 POKE 56334,PEEK(56334) AND 254
1080 POKE 788,0 : POKE 789,192
1090 REM ENABLE IRQ
1100 POKE 56334,PEEK(56334) OR 1
```

If you add these lines to the previous program, you will be able to start the "missile flying" by typing GOTO 1000. The sprite will start moving up the screen. Now if you run the first program, you will see the program and the IRQ routine both moving the sprite! As soon as the program is finished, the sprite will continue moving up the screen. You can list the program or edit it, and the sprite will keep on moving! If you access the disk, the sprite will freeze momentarily while interrupts are disabled, but then move again as soon as the disk operation is over. You can change the sprite color, X position, X or Y expansion, shape, or priority by POKEing the appropriate registers of memory; the sprite will keep moving up the screen.

The implications of this capability should be obvious to anyone who has done any game programming. Several different objects can be set up to automatically move about the screen at different rates and in different directions, all without program control. Moving objects at a constant rate is one of the more difficult

aspects of writing a professional quality game. Different parts of the program require different amounts of time to execute, so object movement tends to be uneven and "jerky." By automating the motion, the game program becomes much simpler. Object direction and speed can be taken from a table in RAM by the IRQ routine. The program can just update the table anytime it needs to change an object's direction or speed. These same problems occur when you try to play music while a program is running. Including a call to a music-playing subroutine within your interrupt interception routine is an excellent solution.

Finally, there is one more technique available for use with sprites. If several different definitions are created for a sprite, the sprite shape can be changed by changing which sprite area number the sprite pointer at 2040-2047 contains. For example, several different sprite images of a walking man could be defined at sprite areas 240-250. Each successive image would portray another fraction of a complete stride. If a program moved the sprite (X position) across the screen while changing the sprite pointer from 240 through 250, the man would appear to walk across the screen. This could easily be incorporated into the IRQ interception routine so that life-like game characters could walk, swim, crawl, or fly across the screen.

Multi-Colored Sprites

Just like the other graphic modes, sprites can trade horizontal resolution for more color information. Unlike the other modes, sprite "multi-coloredness" can be controlled individually, so some sprites can be multi-colored while others are displayed normally. A sprite is specified as multi-colored by setting the appropriate bit in the VIC register at 53276 ($D01C).

```
200 POKE 53276,PEEK(53276) OR (2 ^ N)
210 REM N = THE SPRITE NUMBER 0-7
```

Each pair of bits in the sprite's definition will be interpreted as the source for the color of a single pixel.

Multi-Colored Sprite Color Sources

Bit Pattern	Source
00	Transparent (no color)
01	VIC register 53285 $D025
10	VIC sprite color register
11	VIC register 53286 $D026

Multi-colored sprites consist of 12 horizontal pixels by 21 vertical pixels, but they are not half the width of normal sprites. Instead, the pixels are each twice as wide as normal sprite pixels (making them the same size as normal sprites). In designing a program using multi-colored sprites, you will first have to decide upon the two colors (01 color and 11 color) that all of the multicolored sprites will share. Then decide upon the individual sprite colors for each color register. If you don't have a sprite editor and a good color monitor or television, you should probably design the sprite shapes on quadrille paper with a set of colored pencils. When you're finished, translate the colors into bit pairs, bytes into decimal, and so forth.

Everything else about sprite positioning, expansion, collisions, and priorities applies to the use of multi-colored sprites.

Mixing Graphics Modes

The graphics modes of the VIC chip are controlled by only a few registers. It is possible to change the mode very quickly with a machine language program. The VIC chip can generate a processor interrupt each time the display raster line is equal to a certain register's contents. If the interrupt routine checks for this type of interrupt, and then stores new mode data in the VIC registers, the VIC chip can change modes every time a certain raster line of the display is reached. If the interrupt routine also stores a new interrupt raster line number in the VIC, there can be several mode changes per display scan. Theoretically, this means that part of the screen can be bit-mapped, part normal characters, and/or part multi-colored characters. By changing sprite pointers and registers in the interrupt routine, more than eight sprites can be displayed at once (with certain limitations).

The major limitation with this technique is that the screen division is done horizontally only; entire raster scans must be done in a single mode. A minor limitation is that the normal sixtieth of a second interrupt for the real time clock and keyboard scan must be disabled or else the mode switching will be very erratic. Now the only question is "Why do it at all?" With most other computer systems, one reason was to place text areas in a bit-mapped graphics display. That is quite easy to do by directly copying character data to the bit map in the Commodore 64, and the result is even more versatile. Another classical reason has been so that character graphic objects could be moved quickly over a bit-mapped background. In the Commodore 64 this is done quicker and with much better detail by using sprites over any background desired. Finally, another reason has been so that automatic system scrolling of output text could be used in part of a graphics display.

In the Commodore 64, scrolling is a simple memory-to-memory move, so automatic scrolling is not a great gain. In fact, we shall see in the next section that the normal text output routines may not be usable in many cases. All considered, it may not be worth giving up "missiles" along with the real time clock unless you absolutely must have more than eight sprites displayed at once. If that is the case, here is a sample of the type of interrupt interception routine that you will need:

A Sample IRQ Interception Routine

```
INTER   LDY  #$00      ; USE Y AS AN INDEX
LOOP    LDA  $D012     ; READ VIC RASTER LINE
        CMP  RASTER,Y
        BEQ  FOUND
        INY
        CPY  SPLITS    ; IF NOT OUT OF DATA
        BNE  LOOP      ; TRY THE NEXT ONE
EXIT    PLA            ; RESTORE REGISTERS
        TAY            ; AS THE IRQ ROUTINE
        PLA            ; STACKED THEM
        TAX
        PLA
        RTI            ; RESUME EXECUTION
```

```
FOUND  LDA DATA,Y      ; READ DATA FROM TABLE

       INSERT YOUR OWN CODE HERE TO
       SWITCH SPRITES' X AND Y REGISTERS,
       SPRITE POINTERS, ETC.
       BASED ON THE VALUE READ.

       INY
       LDA RASTER,Y    ; SET UP FOR NEXT LINE
       STA $D012
       LDA #$01
       STA $D019       ; CLEAR INTERRUPT FLAG
       BNE EXIT        ; ALWAYS BRANCH

SPLITS .BYTE 2         ; NUMBER OF SCREEN
                       SEGMENTS
RASTER .BYTE 50,150    ; RASTER LINE NUMBERS
                       ; PUT IN AS MANY AS YOU
                       WANT
       .BYTE 50        ; END LIST WITH FIRST LINE
                       NUMBER
DATA   .BYTE 1,2       ; YOUR CODE INTERPRETS
                       THIS
                       ; TO TELL IT WHAT TO DO
```

To install this interception routine (assuming it was assembled for $C000 and saved on the disk), use something like this:

```
10 REM INSTALL IRQ INTERCEPTION
20 IF A=0 THEN A=1: LOAD "IRQ",8,1
30 POKE 56334,PEEK(56334) AND 254
40 POKE 788,0 : REM LOW ADDRESS BYTE
50 POKE 789,192 : REM HI BYTE
60 POKE 53274,1 : ENABLE RASTER IRQ
```

Memory Allocation

In order to make the examples above easier to understand, I have used only the default memory bank 0 of the VIC chip. In some cases of multiple mode use, it got so crowded that only a tiny

BASIC program could be used. Now for the good news! There are three other 16K memory banks that the VIC chip can use, and selecting any of them will give you more room for BASIC workspace. The bank selection is done through one of the CIA chips:

```
100 REM SELECT BANK N N = 0 TO 3
110 REM 56578 = DATA DIRECTION REGISTER
120 POKE 56578,PEEK(56578) OR 3
130 M = 3 - N
140 POKE 56576,(PEEK(56576) AND 252) OR M
```

Bank	Address Range
0	$0000 - $3FFF
1	$4000 - $7FFF
2	$8000 - $BFFF
3	$C000 - $FFFF

Each bank has it's advantages and disadvantages. Let's examine each bank in turn.

Memory Bank 0

This is the default VIC bank and as such, it has a character generator image at $1000. The built-in system software for screen output puts the screen display memory at $0400. This bank is easy to use with standard character-based graphics programs because no memory needs to be reserved and nothing needs to be "switched." If you want to use a redefined character set, bit-mapped graphics, or more than four sprites, you will need to reserve space so BASIC won't overwrite anything. A word to the wise: Even if your program fits below the area that you need for graphics when you load it, that doesn't mean it always will. As BASIC programs run, additional memory is used for storage of variables, especially string variables. If BASIC hasn't been told that the memory available is restricted, it will eventually overwrite your graphics. You can move the BASIC workspace up in memory so you can use space below it for graphics:

```
POKE 44,n: POKE 642,n: POKE n*256,0: NEW
```

Where n is the first 256 byte page for BASIC

n	Starting Address for BASIC	
8	$0800	default workspace
9	$0900	reserves room for four sprites
16	$1000	reserves 2K for character set
64	$4000	reserves 14K

You may have noticed that the BASIC line to reset the workspace was an immediate command rather than part of a program. If you try to use it in a program, it will work, but execution will stop with a "READY" prompt. You will then have to write or load another program (at the new address), and then type RUN again. There is a very nasty way to write a program that resets the workspace pointers and then starts running another program, but it involves saving the second BASIC program as a machine-code type file for loading by the first and it is not a robust utility (translation; it's a pain in the neck to use).

Using bank 0 memory above BASIC will always severely limit the size of your program. For example, if you just want to reserve 256 bytes at the top of the bank for four more sprite definitions, you will only have 13.75K of workspace instead of the usual 38K. This is because BASIC only understands upper and lower limits, not reserved areas in the middle of memory. To set new upper limits, you can use program lines, but do it early in the program:

```
10 REM SET UPPER LIMIT
20 POKE 52,n : POKE 56,n
```

Where n is the 256 byte page number.

n	Reserve Starting Address	
32	$2000	(i.e., for bit map)
56	$3800	(i.e., for 2K character set)
63	$3F00	(i.e., for four sprites)

There's one more thing to consider when using bank 0. Remember that I said earlier that the VIC chip "sees" an image of the standard character ROM at $1000 to $1FFF? Well, this means that 4K of bank zero is not usable for any other kind of video object because the VIC chip will see only the character generator patterns in that address range. Having the character set there can be useful, IF you want characters. If you don't want to use the standard character sets, try using another bank.

Memory Bank 1

This 16K VIC bank is probably the easiest to use with large BASIC programs that need to POKE or PEEK into a bit-mapped display, character set, or many sprite definitions. Bank 1 is between $4000 and $7FFF. You select this bank by setting the appropriate bits of the CIA port. Then all of the areas selected by VIC chip registers will be relative to $4000. This means that if your program selects $2000 for a bit-map area with bank 1 selected, the bit-map will really be at $6000. Don't forget to change your program's POKE and PEEK addresses. It's a good idea to define variables like "BITMAP" early in your program. Then if you need to change banks, you can take care of the address changes by just changing one line.

There is no character generator image in bank 1, so the whole 16K is available for graphic objects, if you need it. Try to allocate the bank from the top down so that BASIC can have the maximum area possible. If you want a bit-map, put it at $6000 with it's color matrix at $5C00. If you want character sets or sprites, shuffle everything around to compress it toward high memory.

BASIC will still have to know not to overwrite your graphics objects, so don't forget to POKE the page numbers as above. By the way, numbers for BASIC must be real physical addresses, while numbers POKEd into the VIC must be calculated relative to the start of the bank.

If you don't want much in the way of memory for graphics objects, you could have a great deal of the system RAM available to BASIC. For example, if you reserved just 2048 bytes from $7800 to $7FFF for a text screen and 16 sprites, BASIC would have 28K to work with.

Memory Bank 2

Bank 2 falls in the address range of $8000 to $BFFF. There is an image of the character generator ROM from $9000 to $9FFF, so standard characters are easy to use. $9FFF also happens to be the default upper limit of the BASIC workspace. These two functions don't interfere because the VIC chip can only see the characters, while the 6510 can only see the RAM. The memory space from $8000 to $8FFF may be a good choice as the graphics area for a program that needs normal characters and up to 48 sprite definitions displayed on a text screen. BASIC could have 30K to work with, which is enough room for quite a large program.

If you want to define a bit-map in bank 2, or you just want more BASIC workspace, there is another possibility. From $A000 to $BFFF, the 6510 sees the BASIC language ROM but the VIC chip sees unused RAM. If your program can use the bit-map as "write-only" memory, then there's no problem with using this area. Any POKEs or machine code store instructions to $A000-BFFF will be written into the hidden RAM and thus displayed. If

you think about the way bit-mapping works, you will see the limitations: Your program won't be able to just turn on or off a bit by a "read-modify-write" procedure because the "read" will return the ROM contents rather than the bit-map byte from the hidden RAM!

There is a way around this problem, but it is a bit complex. The 6510 processor has an output port at $0000 and 0001 which can deselect the BASIC ROM temporarily. Do not do this from BASIC! To read the contents of a hidden RAM byte, you will need a machine code routine to deselect the BASIC ROM, read the byte, and then select the BASIC ROM again.

READING "HIDDEN" RAM

```
          * = $C000

ADDRL   .BYTE 00        ; LOW BYTE OF ADDRESS
ADDRH   .BYTE 00        ; HI BYTE OF ADDRESS
DATA    .BYTE 00        ; DATA READ

READ    LDA $0001       ; GET THE BIT PATTERN
        AND #$FE        ; BIT 0 FORCED LOW
        STA $0001       ; DESELECT ROM

        LDA ADDRL       ; USED TO PASS ADDRESS
        STA $FB         ; POINTER
        LDA ADDRH       ; HI BYTE OF ADDRESS
        STA $FC
        LDY #$00
        LDA ($FB),Y     ; READ THE BYTE
        STA DATA        ; PASS THE VALUE BACK

        LDA $0001       ; NOW SELECT THE ROM
        ORA #$01        ; BIT 0 FORCED HIGH
        STA $0001
        RTS
```

The routine is simple, but two POKEs for the address plus another PEEK to get the value is a lot more overhead to change just one bit. If you want to use this bank it would be a good idea to

write a general graphics command processor in machine code that could work at a higher level of abstraction. Then your BASIC program could just POKE six bytes of parameters and make a single call to have an entire hi-res line drawn. The command processor could deselect the ROM once upon entry, and then turn it back on just before returning to BASIC.

Memory Bank 3

This is the trickiest bank to use, but it may be worth it. Not only is there system ROM in bank 3, but the input/output devices and character generator are all here as well. Just as in bank 2, write commands to the kernel ROM from $E000 to $FFFF will be stored in the RAM beneath. To read from this area, the code routine must make bit 1 of $0001 low, and then take an additional step. The system interrupt vectors are stored in the kernel ROM at $FFFA-FFFF, so while the ROM is deselected your routines will have to make sure that there are NO interrupts. If there were only maskable interrupts in the Commodore 64, the routine could do this with software, but that is not the case. Both the normal sixtieth of a second clock and the "RUN-STOP/RESTORE" function of the keyboard generate nonmaskable interrupts (NMIs). You can turn off the clock, but you can't disable the key combination. If your program just happens to have the kernel ROM deselected when the user hits the "keyboard reset" combination, your program will probably hang forever. Statistically, this occurence might be rare but do keep it in mind.

The RAM space from $C000 to $CFFF is absolutely free of system use. If you want to store graphics objects there, it's fine with BASIC and the kernel software. BASIC can read and write this area with simple PEEKs and POKEs. This is also a great area for machine code routines, so you will want to plan the use of this memory area carefully.

The address range from $D000 to $DFFF is the most congested area in the whole computer. By default, the input/output devices like the VIC, SID, and CIA chips are selected. If you set bit 2 of $0001 low, the character generator ROM is selected. Finally, if you write to the character generator ROM, the data will be stored

in the hidden RAM below. Under program control, the hidden RAM cannot be read, but the VIC chip can use it to generate a display. Again, to deselect the I/O devices you must first turn off the clock interrupts. Do not try writing to the hidden RAM with the I/O devices selected as the data will also be written to the devices that are enabled. This can hang the system until you cycle the power on and off.

Chapter 8

Inside the Commodore 64's BASIC

One of the recurrent themes in this book has been that you can do a lot of impressive stuff with the Commodore 64 if you go beyond the scope of the normal BASIC commands. Yet almost all of the program examples given have been in BASIC or were designed to be called from BASIC. BASIC may not be able to do everything that the hardware can, but it is the main system language, it is in ROM ready to run when you turn the machine on, and it is in every single Commodore 64 manufactured. By understanding how BASIC works internally, you can get much better performance from your programs and make the system easier to work with.

BASIC is an interpretive language. That means that when you RUN a program in memory, BASIC looks at the first line of your program, decides what to do, does it, looks at the next line, decides what to do, does it, looks at the next line, and so forth. Commodore BASIC is termed a "semi-compiled" BASIC because each command in a line is stored in memory as a single byte "token." The tokens are pointers into a table of subroutine addresses. There is an address in the table for the code that performs each BASIC command. This results in much more compact programs than the older "straight-interpretation" that does not use tokens. Other things that appear in lines of BASIC, such as string literals and numbers, are not tokenized in any way. Each time a line is executed, numbers must be converted from their ASCII representation into the internal binary format. This has a great effect on program speed, which will be discussed later. Here is the format of BASIC lines in RAM:

BASIC Program line:

```
      10 FOR A=1 TO 8 STEP 2

$13   08   0A   00   81   20   41   B2   31
 PL   PH   LL   LH   T1        "A"  T2   "1"

 20   A4   20   38   20   A9   20   32   00
      T3        "8"       T4        "2"
```

PL and PH taken together contain the address (in this case $0813) of the next line's PL. This forms a linked list of BASIC lines for commands like GOTO and GOSUB . . . RETURN.

LL and LH taken together contain the line number as a 16 bit binary integer (000A = 10). This is stored next to the pointer to make line searches as fast as possible.

T1 is the token of the first command in the line. In this case, the token $81 stands for the BASIC command FOR. Two bytes, and a lot of interpretation time, are saved by abbreviating the command. BASIC recognizes tokens by looking at bit 7 of each byte of the line. Only tokens have bit 7 equal to one.

Literal data may or may not be present after a token. For example, a PRINT command will work with a literal string, a number, a variable name, an equation, or nothing at all. The code executed for PRINT reads the data following the PRINT token to decide what to print. When it encounters the end of the line or a colon, it returns to the BASIC interpreter. In our example, the $31 is the ASCII code for "1".

T2, T3 and T4 represent the fact that there may be more than one token in a single line. In this example there are four tokens in the line with data in between.

0 is used as an end of line marker. When any command execution code encounters a zero, control is returned to the BASIC interpreter.

The first line of BASIC normally starts at $0801 (2049). Where it starts is controlled by a pointer stored at $2B,2C. By storing another address at the start of BASIC pointer, you can put the

first line anywhere in the BASIC workspace. The pointer at $2B points to the first line pointer, which points to the next line pointer, and so forth until the last line. The last line's pointer points to a "null" pointer; two bytes of zero just past the last line. This all makes it fairly easy to search for a certain line or, in our case, to write a renumber utility:

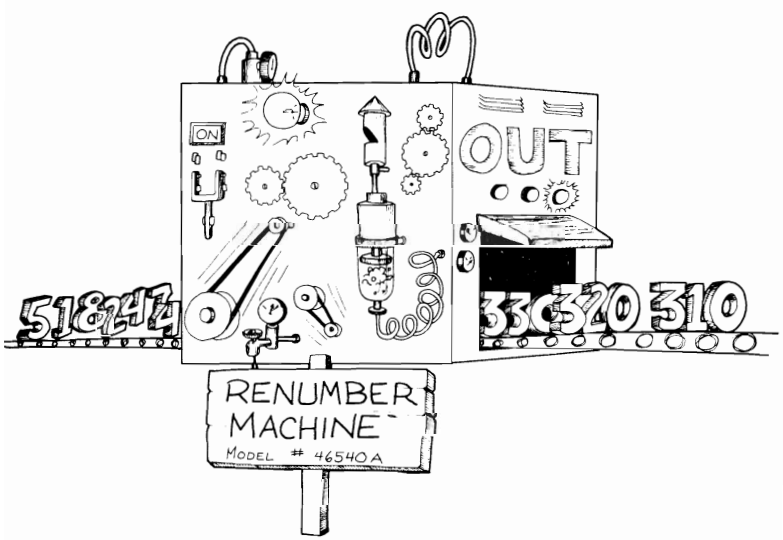
Renumber a BASIC Program:

```
NUMBER = $FB
POINT = $FD
      * = $C000 ; FULLY RELOCATEABLE

AT10    LDA #10         ; START WITH 10
        STA NUMBER
        LDA #00
        STA NUMBER+1
ATNN    LDY #00         ; OPTIONAL ENTRANCE
        LDA $2B         ; FIRST LINE POINTER
        STA POINT
        LDA $2C         ; FIRST LINE POINTER
        STA POINT+1
LOOP    LDA (POINT),Y   ; LOOK FOR NULLS
        INY
        ORA (POINT),Y
        BNE MORE
        RTS             ; FINISHED
MORE    INY             ; NOW POINTS TO LINE
                          NUMBER
        LDA NUMBER      ; GET NEW LINE NUMBER
        STA (POINT),Y   ; RENUMBER
        INY
        LDA NUMBER+1
        STA (POINT),Y
        CLC             ; NOW ADD 10 TO NUMBER
        LDA NUMBER
        ADC #10
        STA NUMBER
        LDA NUMBER+1
        ADC #00         ; FOR CARRY
```

```
STA NUMBER+1
LDY #Ø0          ; RESET Y
LDA (POINT),Y    ; GET NEXT LINE ADDRESS
PHA
INY
LDA (POINT),Y
STA POINT+1      ; UPDATE POINTER
PLA
STA POINT
LDY #Ø0
BEQ LOOP         ; ALWAYS BRANCH
```

This subroutine can be assembled at any convenient address and then saved to the disk. When you need to renumber a BASIC program, it can be loaded, moved anywhere you like, and run with a SYS call. If you call the first instruction, it will renumber all of the current BASIC programs line from 10 with intervals of 10. If you POKE some other starting line number into NUMBER and then call ATNN (start +8), it will begin numbering with that number. If you POKE the parameter in the ADC instruction, it can use any interval between 1 and 255.



Notice that this program only alters line numbers, it does not fix line number references within BASIC lines. If your program contains GOTOs, GOSUBs, or IF . . . THEN <line number>s, they will point to the wrong lines after you renumber! You will have to edit them to point to the new line numbers. An easy way to keep track of line number changes is to change each target line (line jumped to) to a REM statement that includes the old line number. Then after you renumber you will be able to list the program and see the new number for each old number. On the other hand, you may see fit to enhance the renumber routine to look for the appropriate tokens and fix the line numbers following. Be forewarned; that is a much more complex problem, requiring ASCII to binary and binary to ASCII conversions.

BASIC Data Storage

The workspace beyond the program lines is used for storage of all of BASIC's data structures. Variables, number arrays, string pointers, and function pointers are all stored starting directly after the end of program null pointer. Strings built by program actions are stored starting at the top of the workspace and working down. There are two different pointers used by BASIC to keep track of these areas. By understanding the structures involved, you can make machine code routines that obtain data directly from BASIC variables, return data the same way, or perform complex actions like sorting string arrays.

Floating Point Variables

When a program is RUN, data structures are added to the end of the program as they are encountered during program execution. In most programs, the first item defined is a floating point variable. Whatever it is though, the first item's address will be contained in the pointer at $2D,2E. Floating point variables are stored in RAM in this format:

```
SAM = 2

$53 $41 $82 $00 $00 $00 $00
 N1  N2   E  M1  M2  M3  M4
```

N1 and N2 are the first two characters of the variable name. Bit 7 of each ASCII character is low.

E is the exponent offset by 128. This says how many bit positions to shift the mantissa to form the real number. A value of 129 means to shift right one bit, 127 means to shift left one bit.

M1 through M4 are the mantissa, with bit 7 of M1 holding the mantissa sign. The mantissa is always stored fully normalized so that a 1 in bit 7 is understood when performing math operations on the mantissa. Multi-byte binary floating point format numbers are difficult to work with, so machine code routine parameter passing through integer variables is recommended instead.

Integer Variables

Integer variables have little utility in the Commodore 64. In older BASICs, integer operations were faster and integer data structures saved memory space compared to their floating point analogs. In the Commodore 64, all math operations are done in floating point, so use of integer variables actually slows a program because every math operation on an integer variable has to do an extra integer-to-floating point conversion. Integer variables are stored in seven bytes just like floating point variables even though three of those bytes aren't used. This makes the variable search code simpler. Passing parameters with machine code programs may be the only thing that integer variables are good for.

```
BILL% = 7

$C2 $C9 $00 $07 $00 $00 $00
 N1  N2 VH VL  X1  X2  X3
```

N1 and N2 are the first two characters of the integer variable's name. Both bytes have bit 7 set high to signify that it is an integer variable. VH and VL contain the high and low bytes respectively of a 16 bit 2's complement integer. For example, a value of –1 will be represented as $FFFF, while a value of 23 will be represented by a 0 in VH and a 23 in VL.

If you want a machine code program to pass parameters through a variable, the routine could search the seven byte variable structures between the start-of-variables pointer and the start-of-arrays pointer until it found the variable by name, but there is a simpler way: If you declare one or more integer variables as the very first thing in the BASIC program, they will all be in memory in order of declaration, with the start-of-variables pointer pointing to the first one. The code routine can just index off of the pointer contents to read or write any of those variables.

Parameter Passing with Code Routines

BASIC Program:

```
1 REM DECLARE 4 PASSING VARIABLES
2 X% = Ø
3 Y% = Ø
4 COLOR% = Ø
5 FLAG% = Ø
1Ø <REST OF PROGRAM>
```

Code Routine:

```
LDY #Ø3          ; 7 * VARIABLE # + 2 OR 3
LDA ($2D),Y      ; GET LOW BYTE OF X%

LDY #1Ø          ; 7 * 1 + 3
LDA ($2D),Y      ; GET LOW BYTE OF Y%

LDY #16          ; 7 * 2 + 2
LDA ($2D),Y      ; GET HIGH BYTE OF COLOR%

LDY #Ø2          ; 7 * Ø + 2
STA ($2D),Y      ; PUT DATA IN HIGH BYTE OF X%
```

This technique can be much faster than breaking apart variables for POKE statements or rebuilding them from two PEEKs, not to mention that no extra memory is required to temporarily store POKE and PEEK data. Using the pointer as opposed to a fixed

address ensures that the code routine will automatically "find" the variables even if they are moved around because of changes to the BASIC program. As an exercise, you may want to recode one of the BASIC/machine code programs given in an earlier chapter to use integer variable parameter passing.

Functions

BASIC functions are defined before use by a statement like:

```
10 DEF FN AA (X) = X + 7
```

Assuming that the variable X has not already been defined, this DEF statement will add two items to the variable list: A pointer to the function AA (within the BASIC line), and a normal floating point variable X. The format of the floating point dummy variable is exactly as given above for other floating point variables. This is the format used for the function:

```
10 DEF FN AA (X) = X + 7

$C1 $41 $12 $08  $65  $08 $58
 N1  N2  PL  PH PVL PVH  N3
```

N1 and N2 are the first two characters of the function name. N1 has bit 7 set to indicate that it is a function name.

PL and PH are a pointer to the first byte after the equal to sign in the BASIC line that defines the function, for function evaluation at runtime.

PVL and PVH are a pointer to the dummy variable storage area (first byte after the two name characters).

N3 is the first character of the dummy variable name.

String Pointers

When a string variable is first assigned, an entry is added to the variable list. This entry takes up seven bytes just like the other

types, but this entry doesn't actually hold any data. Instead, it contains the length and starting address of the string data. This data will be within the BASIC program in the case of a literal assignment:

```
100 A$ = "HELLO THERE"
```

If the assignment is not from a string literal, then the string data will be stored starting at the top of the BASIC workspace. Each time an operation is performed upon a string variable, the new string is stored further down in memory and the pointer is adjusted. Eventually, the string list may overflow the other variables or arrays, so you may want to include a CLR statement in the outermost loop of any program that does a lot of string manipulation. Here is the format of a string pointer:

```
100 AX$ = "HELLO THERE"

$41 $D8 $0B $0C $08 $00 $00
 N1  N2   L  PL PH  B1  B2
```

N1 and N2 are the first two characters of the string variable's name. Bit 7 of N2 is set, in order to identify the variable as a string-type.

L contains the length of the string, in bytes.

PL and PH are the pointer to the first byte of the string data, whether in the BASIC program or in the string workspace.

B1 and B2 are unused.

Storage of Arrays

Arrays present a convenient method for dealing with long lists or tables of data of similar nature. In terms of storage, a great deal of memory can be saved because the array name is stored only once and the data is kept in the minimum space. The array list starts just after the variable list and is pointed to by $2F,30. Each array entry has only the length required and each array type has

a different format. In all array types, elements are stored in the order derived by incrementing the first dimension, then the second, and so forth. For example, in a two-dimensional array the elements will be stored in this order:

(0,0) (1,0) (2,0) (n,0) (0,1) (1,1) ...

Floating Point Arrays

Floating point arrays are stored with the following format:

```
        10 DIM MANY(10,10)

$4D   41   66   02    2   00   0B        00   0B
 N1   N2   IL   IH    D  LHN  LLN  ...  LH1  LL1

  0    0    0    0    0    0    0    0    0    0
  E    M    M    M  ...    E    M    M    M    M
```

N1 and N2 contain the first two characters of the array name.

IL and IH contain a relative offset to the next array IL byte. This is necessary for runtime searching of the array list, since arrays are all different sizes.

D is the number of dimensions. Dimensionality is strictly enforced by Commodore BASIC.

LLN and LHN contain the length of the LAST dimension. Note that a DIMensioned length parameter of 10 actually creates 11 elements, since 0 is a valid index. The lengths of each dimension are stored in pairs of bytes down to the first pair, represented above by LL1 and LH1. A one-dimensional array will have only one length parameter.

E M M M M represents a number in normal floating point format. Space is reserved for each element of the array.

Integer Arrays

Integer arrays are stored with the following format:

```
10 DIM MA% (10,10)

$CD   C1  FB  00  2    00   0B      00  0B  00 00      00 00
 N1   N2   IL  IH  D  LHN LLN ... LH1 LL1 VH VL ... VH VL
```

N1 and N2 contain the first two characters of the array name. In both bytes, bit 7 is set.

IL and IH contain a relative offset to the next array IL byte.

D is the number of dimensions.

LLN and LHN contain the length of the LAST dimension. (DIM parameter plus one.) The lengths of each dimension are stored in pairs of bytes down to the first pair, represented above by LL1 and LH1. A one-dimensional array will have only one length parameter.

VH and VL represent a number in normal integer variable format. Space is reserved for each element of the array. Since each element of an integer array only requires two bytes, considerable memory savings can be obtained by using integer arrays whenever possible.

String Arrays

String arrays, like string variable entries, hold only length and address information. The actual string characters are stored within the BASIC program or in the string workspace starting at the top of available RAM.

```
10 DIM G$ (5,3)
20 G$(0,0)="HI"

$47   80   51   00    2   00   04
 N1   N2   IL   IH    D LHN LLN  ...
```

00	06	02	1D	08		00	00	00
LH1	LL1	SL	PL	PH	...	SL	PL	PH

N1 and N2 contain the first two characters of the array name. Bit 7 of N2 is set.

IL and IH contain a relative offset to the next array IL byte.

D is the number of dimensions.

LLN and LHN contain the length plus one of the LAST dimension. (DIM parameter plus one.) The lengths of each dimension are stored in pairs of bytes down to the first pair, represented above by LL1 and LH1.

SL holds the length of an element string. An undefined string array element will have a length of zero.

PL and PH contain the address of the first byte of the string characters. There is an SL and PL/PH pair for each element of a string array. Undefined strings have an address of zero.

Maximizing Execution Speed

BASIC program execution can be made faster by keeping in mind some of the program line and data structures. One important thing to consider is that readability should not be sacrificed. Blank characters are skipped by the BASIC interpreter in just a few microseconds each. Don't bother trying to remove them all from your program; the effect is usually undetectable in terms of speed. On the other hand, there are simple practices that can help a great deal:

Line number searches are done from the first line. Anything that you can do to decrease the number of lines between the first line and lines that are the targets of GOTO or GOSUB will increase the search speed. One way to do this is to write a small main program followed by subroutines arranged in order of the number of times each is called. Another important rule is to always use an initialization subroutine located at the end of the program.

When the program is run, this subroutine should be called first. All variables are assigned, even if only with dummy data, then all arrays are dimensioned. This subroutine can have lots of comments so a reader can understand what each variable or array contains. Defining all the variables and the functions before the arrays is important because if you do it the other way around, all of the arrays have to be moved to make room for each new variable or function. Assign the variables and arrays in the order of "most-used first." This will make the searches faster. If your program prints some instructions, you might do all of this initialization while the user reads them.

Another rule for speeding-up a program is to avoid literal constants. Each time a constant is encountered in a program line, it has to be converted from ASCII to floating point. Define all of your constants as named variables in your initialization subroutine.

Avoid any use of integers. Commodore BASIC can only perform floating point math, so you get rid of a lot of type conversion by not using integers.

FOR . . . NEXT loops are faster than incrementing a variable and testing the result.

GOSUB . . . RETURN is faster than GOTO with a GOTO return.

```
10 REM AN OPTIMAL SPEED PROGRAM
20 GOSUB 10000
30 GOSUB 100
40 INPUT N
50 ON N GOSUB 200,300,400
60 GOTO 30

100 PRINT CHR$(147);" MENU"
110 PRINT:PRINT:PRINT
120 PRINT "1 . . . SHOW MEMORY"
130 PRINT "2 . . . CLEAR MEMORY"
140 PRINT "3 . . . SET NEW ADDRESS"
150 RETURN
```

```
200 FOR A = F TO L
210 PRINT A,PEEK(A)
220 NEXT:INPUT N:RETURN

300 FOR A = F TO L
310 POKE A,0
320 NEXT:RETURN

400 INPUT"ENTER FIRST ADDRESS";F
410 INPUT"ENTER LAST ADDRESS";L
420 RETURN

10000 A=0 : REM ADDRESS LOOP COUNTER
10010 N=0 : REM USER MENU CHOICE
10020 F=0 : REM FIRST ADDRESS
10030 L=0 : REM LAST ADDRESS
10040 REM NOW DIM ARRAYS
10050 DIM Z(10,10) : REM EXAMPLE ARRAY
10060 RETURN
```

Minimizing Memory Use

To minimize the amount of memory that a program uses, you must do the opposite of almost everything above. Things that save memory are also things that slow the program execution:

Don't define literals as variables. This will save you seven bytes per literal.

Use integer arrays instead of floating point. This will save you three bytes per element.

Group lines together with colons. This will save you four bytes per line number eliminated, but be careful about IF . . . THEN and REM statements.

Remove spaces and/or remarks.

Best of all, analyze why your program is using so much memory and then rewrite it using better data structures and fewer lines. The Commodore 64 gives you 38K of BASIC workspace, which is very large for a personal computer. If you have a program that really can't fit, you should probably divide it into two or more separate programs.

Customizing BASIC

One of the more interesting possibilities of the Commodore is that by reading and then writing each byte of the BASIC or kernal ROMs, you can copy either into the hidden RAM. Deselect the ROM and you'll be running the system software out of RAM. This makes it possible, in fact quite easy, to customize BASIC or operating system commands.


```
10 REM CUSTOMIZE BASIC A BIT
20 FOR A = 40960 TO 49151 : REM BASIC ROM
30 POKE A,PEEK(A) : REM COPY ROM TO RAM
40 NEXT A
```

```
50 POKE 1,(PEEK(1) AND 254) : REM SELECT
   RAM
60 FOR A = 41848 TO 41853 : REM "READY"
   MESSAGE
70 READ D
80 POKE A,D
90 NEXT A
100 DATA 72,69,76,76,79,33 : REM "HELLO!"
```

If you run the program above, BASIC will be copied into RAM and instead of "READY" the prompt will be "HELLO!". The loop at line 20 that copies each of 8192 bytes will take a while to run, but this is the type of task that makes for a simple machine code routine. You can do the very same thing with the kernel ROM from 57344 to 65535, except that the kernel ROM is selected by bit 1 of the port at address 1. If you store a 0 in bit 1, *both* the kernal ROM *and* the BASIC ROM will be deselected. Don't forget to copy both to RAM before deselecting.

Extending BASIC

There are two different ways to extend BASIC; with system calls or by adding a special character trap. Using system calls has the advantage of being very fast, while the special character extension method lets you write new BASIC-like commands. System calls for any particular program tend to be quite specialized and often only one or two machine code subroutines are needed. In this book several candidate subroutines have been given. Most of these have been very straightforward because they each performed a single function and did not interact with BASIC. The examples of parameter passing and renumbering given above suggest that often a machine code subroutine will be most useful if it does interact with BASIC. Interactive experimentation and development of machine code subroutines is especially difficult because of the time it takes to make a modification.

Another problem is that 6510 opcodes and data tend to be represented in hexadecimal, while BASIC only understands decimal

numbers. Rather than having to use FOR . . . READ . . . POKE . . . NEXT loops with a bunch of incomprehensible decimal data, wouldn't it be great if Commodore BASIC understood hex addresses and data? Here is an extension to the BASIC interpreter itself that can do just that:

```
; CODER SUBROUTINE
;
        * = $C000       ; OR ANYWHERE CONVENIENT
POINT   = $FB           ; USE $FB,FC AS A POINTER
DATA    = $FD           ; TEMPORARY STORAGE
        CLC
        LDA $7A         ; SKIP TO NEXT BASIC LINE
        ADC #5          ; $7A IS THE INTERPRETER
                        POINTER
        STA $7A
        LDA $7B
        ADC #0          ; ADD ANY CARRY
        STA $7B         ; POINTS TO 1ST BYTE OF data
FINDA   JSR GETC        ; LOOK AT BYTE
        CMP #$20        ; SKIP BLANKS
        BEQ FINDA
FOUND   JSR CONA        ; FORM HI BYTE OF ADDRESS
        STA POINT+1
        JSR GETC
        JSR CONA        ; FORM LO BYTE OF ADDRESS
        STA POINT
DODATA  JSR GETC        ; LOOK AT NEXT BYTE
        BNE MORE        ; IS IT END-OF-LINE $00?
        RTS             ; RETURN IF IT IS
MORE    CMP #$30        ; IS IT A NUMBER?
        BCC DODATA      ; IGNORE PUNCTUATION
        JSR CONA        ; FORM A DATA BYTE
        LDY #0
        STA (POINT),Y   ; "POKE" THE DATA
        INC POINT       ; ADVANCE THE POINTER
        BNE DODATA
        INC POINT+1
        BNE DODATA      ; ALWAYS BRANCH FOR MORE
```

```
; CONA CONVERTS THE ACCUMULATOR DATA AND
  THE NEXT
; BYTE INTO A BINARY NUMBER.
CONA    SEC
        SBC  #$30       ; CONVERT ASCII->HEX
        CMP #$0A        ; HANDLE A THRU F
        BCC  OKAY1
        SBC  #$07
OKAY1   ASL             ; SHIFT INTO HI NIBBLE
        ASL
        ASL
        ASL
        STA  DATA       ; SAVE RESULT
        JSR  GETC       ; GET NEXT BYTE
        SEC
        SBC  #$30       ; CONVERT AGAIN
        CMP #$0A
        BCC  OKAY2
        SBC  #$07
OKAY2   ORA DATA        ; COMBINE BOTH
        RTS
; GETC GETS THE NEXT CHARACTER FROM WITHIN
; THE BASIC  DATA STATEMENT.
GETC    INC  $7A
        BNE OKAY3
        INC  $7B
OKAY3   LDY #$00
        LDA [$7A],Y
        RTS
```

When you call this subroutine with a SYS command, it interprets the data on the next line as a hex address followed by some hex numbers to be stored starting at the address. It doesn't mind spaces or most punctuation characters between the numbers, but it does not like colons. The only restrictions are that the SYS call must be the last thing in the line directly before the DATA line, and each hex number must not be shortened: The address must be four characters, the data numbers must be two characters.

You can use this routine to store a machine language program in memory:

```
1000 REM SET UP SUBROUTINE AT $C100
1010 SYS 49152
1020 DATA C100 A9 01 8D 00 04 60
```

You can use it to send data to the screen or color memory:

```
2000 REM $0400 = SCREEN $D800 =
     COLOR RAM
2010 SYS 49152
2020 DATA 0400 01,02,03,04,05,06
2030 SYS 49152
2040 DATA D800 060604040202
```

You can even use it to set up registers in the VIC, SID, or CIA chips:

```
3000 REM SET BACKGROUND COLORS
3010 SYS 49152
3020 DATA D020 00 00 04 08 03
```

Note: The different formats used above are all acceptable to CODER. Use whichever you like or mix them.

As it is written, the CODER subroutine accepts a single line of data each time it is called. It could easily be modified to jump five bytes each time it found an end-of-line zero byte so that it could store many lines of data per call. It would then need to search for a special character like an asterisk to tell it that it should stop. Then the BASIC lines *could* look something like this:

```
5000 REM MODIFIED FOR MULTI-LINES
5010 SYS 49152
5020 DATA C100 A9 02 8D 00 04
5030 DATA      A9 04 8D 01 04
5040 DATA      A9 03 8D 04 04
5040 DATA      A9 05 8D 05 04
5050 DATA      A9 10 8D 10 04
5060 DATA      8D 11 04 60 *
```

Adding New Commands to BASIC

There is another simple way to add new commands to Commodore 64 BASIC. The subroutine that fetches each byte of the BASIC program lines is in RAM rather than the BASIC ROM, so you can modify it to check for special characters. When one of these characters is found in a line of BASIC during program execution, your own machine code subroutines can be executed. The special character must be one that has no meaning to BASIC. The ampersand character is a good choice. The CHRGET routine is copied from ROM into RAM when the power is turned on. It normally looks like this:

Get a Character for BASIC:

```
$0073 --> INC  $7A
          BNE $79
          INC  $7B        ; ADVANCE THE POINTER
$0079 --> LDA ????        ; SELF-MODIFIED VALUE
          CMP#$3A
          BCS $8A         ; BRANCH ON > "9"
          CMP#$20
          BEQ $73         ; SKIP BLANKS
$0084 --> SEC
          SBC #$30        ; THIS TESTS FOR THE RANGE
$0087 --> SEC             ; "0"—"9" = CARRY CLEAR
          SBC #$D0        ; DATA IS UNCHANGED
$008A --> RTS
```

Note that the first three instructions always modify the fourth instruction. This way indirect addressing and clearing the Y register is avoided, so the character fetching is quite fast. The range testing gives us the opportunity to sneak in a test of our own. Let us "patch" the code so that the three bytes at $0084 contain:

```
$0084 --> JMP $CF00 ; JUMP TO OUR CODE
```

Of course, this cannot be done from a BASIC program since the first POKE would make the routine stop working. You could use the CODER subroutine above, or a small code routine:

```
PATCH   LDA #$4C        ; JMP OPCODE
        STA $84
        LDA #$00        ; LOW BYTE OF OUR CODE
        STA $85
        LDA #$CF        ; HI BYTE OF OUR CODE
        STA $86
        RTS
```

Our code (at $CF00 in this example) must check for the ampersand character and if not found, perform the range test patched over before jumping back to CHRGET:

```
OURS    CMP #$26        ; IS IT "&"?
        BEQ  MYCODE
        SEC             ; REPLACE ORIGINAL TEST
        SBC  #$30
        JMP  $0087      ; JUMP BACK JUST AFTER
                          PATCH

MYCODE (here can be anything you like)
         (for example                )
        INC $D020       ; CHANGE BORDER COLOR

         (must end with              )
        JMP #$0073      ; GET NEXT CHARACTER
```

In this example, if the ampersand is found anywhere in a BASIC line, the border color will be changed with no further character processing. This is fine for adding a single extension, but it is more likely that you would want to have code to fetch the next character after the ampersand to tell which of several routines to execute. You could even write a routine so that parameter variable names could be passed following one of these "ampersand commands."

Support &A to &Z

```
MYCODE  TYA             ; SAVE Y REGISTER
        PHA             ; ON THE STACK
        JSR $0073       ; GET NEXT CHARACTER
```

```
            SEC
            SBC  #$41       ; TRANSLATE a-z TO 0-25
            ASL             ; TIMES 2
            TAY             ; USE AS INDEX
            LDA  TABLE,Y    ; GET ROUTINE LO ADDRESS
            STA  JADD+1     ; MODIFY JADD LO BYTE
            INY
            LDA  TABLE,Y    ; GET ROUTINE HI ADDRESS
            STA  JADD+2     ; MODIFY JADD HI BYTE
JADD        JSR  ????       ; JSR TO SOME ROUTINE

EXIT        PLA             ; CLEAN UP STACK
            TAY             ; RESTORE Y
            JMP $0073       ; NORMAL PROCESSING

TABLE .WORD RTNA,RTNB,RTNC . . . RTNZ

(TYPICAL ROUTINE:)

RTNA        INC $D020       ; CHANGE BACKGROUND
            RTS             ; ALL RTNS END THIS WAY
```

RTNA would be called from BASIC by simply inserting "&A" anywhere in a BASIC program line. This could be before other commands or on a line by itself. Many such calls could be on the same line if need be.

```
10 REM USES "AMPERSAND-A" FROM ABOVE
20 SYS 49152 : REM "PATCH"
30 INPUT "DO YOU LIKE THE BORDER
   COLOR";N$
40 IF LEFT$(N$,1) = "Y" THEN 100
50 &A : REM CHANGE BORDER COLOR
60 GOTO 30
100 END
```

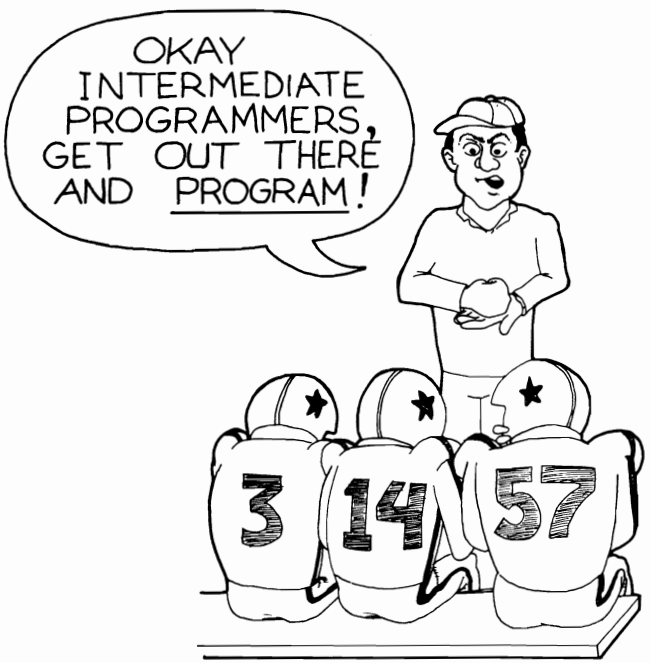
Ampersand commands will also work interactively once PATCH has been executed. In fact, if you then re-enter a BASIC line from a screen listing by placing the cursor on the line and hitting <return>, the ampersand commands in the line will be executed immediately and will be left out of the program line. To edit or

enter additional BASIC lines containing ampersand commands, you will have to "unpatch" CHRGET so that ampersands are ignored:

Unpatch CHRGET for Editing

```
UPATCH LDA  #$38      ; SEC OPCODE
       STA  $84
       LDA  #$E9      ; SBC IMMEDIATE OPCODE
       STA  $85
       LDA  #$30      ; OLD SBC DATA
       STA  $86
       RTS
```

This would be the preferable method for interfacing a machine code graphics program to BASIC. You could define an entire set of commands like &PLOT, &LINE, &COLOR, and so forth, all with literal or variable name parameter passing, just using concepts from this chapter and the previous chapter on video graphics. Other commands could affect the sound generator, read the joysticks or paddles, or whatever you wanted.



Now For The Pep Talk:

Well, this is it. If you haven't skipped any chapters then you now know at least as much about programming the Commodore 64 as I do.

You should be able to specify, design, and write more ambitious programs without getting tangled up in the details by using flow diagrams, variable charts, and top-down design technique.

You should understand the importance of choosing the best (or at least, not the worst) algorithms for your programs. You should also be able to translate equations from reference books into lines of BASIC with relative ease.

You now know how to construct some fairly sophisticated data file structures on the disk. More important, you also should understand what sort of things disks do well, and what sort of things they don't do well.

You have written, or at least entered, short assembler or machine code routines. Longer assembler programs are not inherently different from short ones, so you should be able to write any 6510 code that you need. Be patient! And please, if you don't have an assembler, buy one.

You probably understand more about all of the different graphic modes than you thought possible. There is a lot to learn, but that knowledge is essential since the most outstanding capabilities of the Commodore 64 are tied to the VIC video graphics chip.

Finally, you understand (or at least know where to find) everything about the way BASIC stores program lines and data. With this information you should be able to pass parameters to machine code routines and maybe even add an extension or two.

So, what more can I tell you? Nothing is stopping you now. The only limits on what you can do with your Commodore 64 are those imposed by lack of imagination! Get out there and write the program that captures the world's attention. You'll learn much more by writing real programs than you will by any other means.

Appendix A: Decimal Opcodes

6510 Microprocessor Opcodes in Decimal

OP	IMP	ACC	ABS	OP	#	,X	,Y	(,X)	( ),Y	OP,X	REL	( )	OP,Y
ADC			109	101	105	125	121	97	113	117			
AND			45	37	41	61	57	33	49	53			
ASL		10	14	6		30				22			
BCC											144		
BCS											176		
BEQ											240		
BIT		44	36										
BMI											48		
BNE											208		
BPL											16		
BRK	0												
BVC											80		
BVS											112		
CLC	24												
CLD	216												
CLI	88												
CLV	184												
CMP			205	197	201	221	217	193	209	213			
CPX			236	228	224								
CPY			204	196	192								
DEC			206	198		222				214			
DEX	202												
DEY	136												
EOR			77	69	73	93	89	65	81	85			
INC			238	230		254				246			
INX	232												
INY	200												
JMP			76									108	
JSR			32										
LDA			173	165	169	189	185	161	177	181			
LDX			174	166	162		190						182
LDY			172	164	160	188				180			
LSR		74	78	70		94				86			
NOP	234												
ORA			13	5	9	29	25	1	17	21			
PHA	72												
PHP	8												
PLA	104												
PLP	40												

ROL		42	46	38		62				54			
ROR		106	110	102		126				118			
RTI	64												
RTS	96												
SBC			237	229	233	253	249	225	241	245			
SEC	56												
SED	248												
SEI	120												
STA			141	133		157	153	129	145	149			
STX			142	134									150
STY			140	132							148		
TAX	170												
TAY	168												
TSX	186												
TXA	138												
TXS	154												
TYA	152												

Appendix B: Hexadecimal Opcodes

6510 Microprocessor Opcodes in Hexadecimal

OP	IMP	ACC	ABS	OP	#	,X	,Y	(,X)	( ),Y	OP,X	REL	( )	OP,Y
ADC			6D	65	69	7D	79	61	71	75			
AND			2D	25	29	3D	39	21	31	35			
ASL		0A	0E	06		1E				16			
BCC											90		
BCS											B0		
BEQ											F0		
BIT			2C	24									
BMI											30		
BNE											D0		
BPL											10		
BRK	00												
BVC											50		
BVS											70		
CLC	18												
CLD	D8												
CLI	58												
CLV	B8												
CMP			CD	C5	C9	DD	D9	C1	D1	D5			
CPX			EC	E4	E0								
CPY			CC	C4	C0								
DEC			CE	C6		DE				D6			
DEX	CA												
DEY	88												
EOR			4D	45	49	5D	59	41	51	55			
INC			EE	E6		FE				F6			
INX	E8												
INY	C8												
JMP			4C								6C		
JSR			20										
LDA			AD	A5	A9	BD	B9	A1	B1	B5			
LDX			AE	A6	A2		BE						B6
LDY			AC	A4	A0	BC				B4			
LSR		4A	4E	46		5E				56			
NOP	EA												
ORA			0D	05	09	1D	19	01	11	15			
PHA	48												
PHP	08												

PLA	68										
PLP	28										
ROL		2A	2E	26		3E				36	
ROR		6A	6E	66		7E				76	
RTI	40										
RTS	60										
SBC			ED	E5	E9	FD	F9	E1	F1	F5	
SEC	38										
SED	F8										
SEI	78										
STA			8D	85		9D	99	81	91	95	
STX			8E	86							96
STY			8C	84						94	
TAX	AA										
TAY	A8										
TSX	BA										
TXA	8A										
TXS	9A										
TYA	98										

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

INTERMEDIATE COMMODORE 64

Perfect for the BASIC programmer who is ready to move on.

INTERMEDIATE COMMODORE 64 will take you from being a fledgling BASIC programmer and show you important principles that can help you handle more complicated programming problems. You'll learn how to structure your program "one step at a time," reducing big problems into smaller, more manageable ones. Your programming will become clearer and easier to understand in the process.

The many benefits of structured Commodore BASIC programming include:

- Programs are easily understood.
- Errors are reduced.
- Programs are simple to maintain.
- Coding is faster, speeding up program development.
- Coding large programs is simplified.

You'll also learn about flow diagrams, algorithms, text files, enhanced graphics, special printer techniques, and many more tricks of the trade. So if you're ready to take that intermediate programming step, then INTERMEDIATE COMMODORE 64 is ready for you!

ISBN 0-88190-253-5

DATAMOST INC

20660 Nordhoff Street, Chatsworth, CA 91311-6152
(818) 709-1202