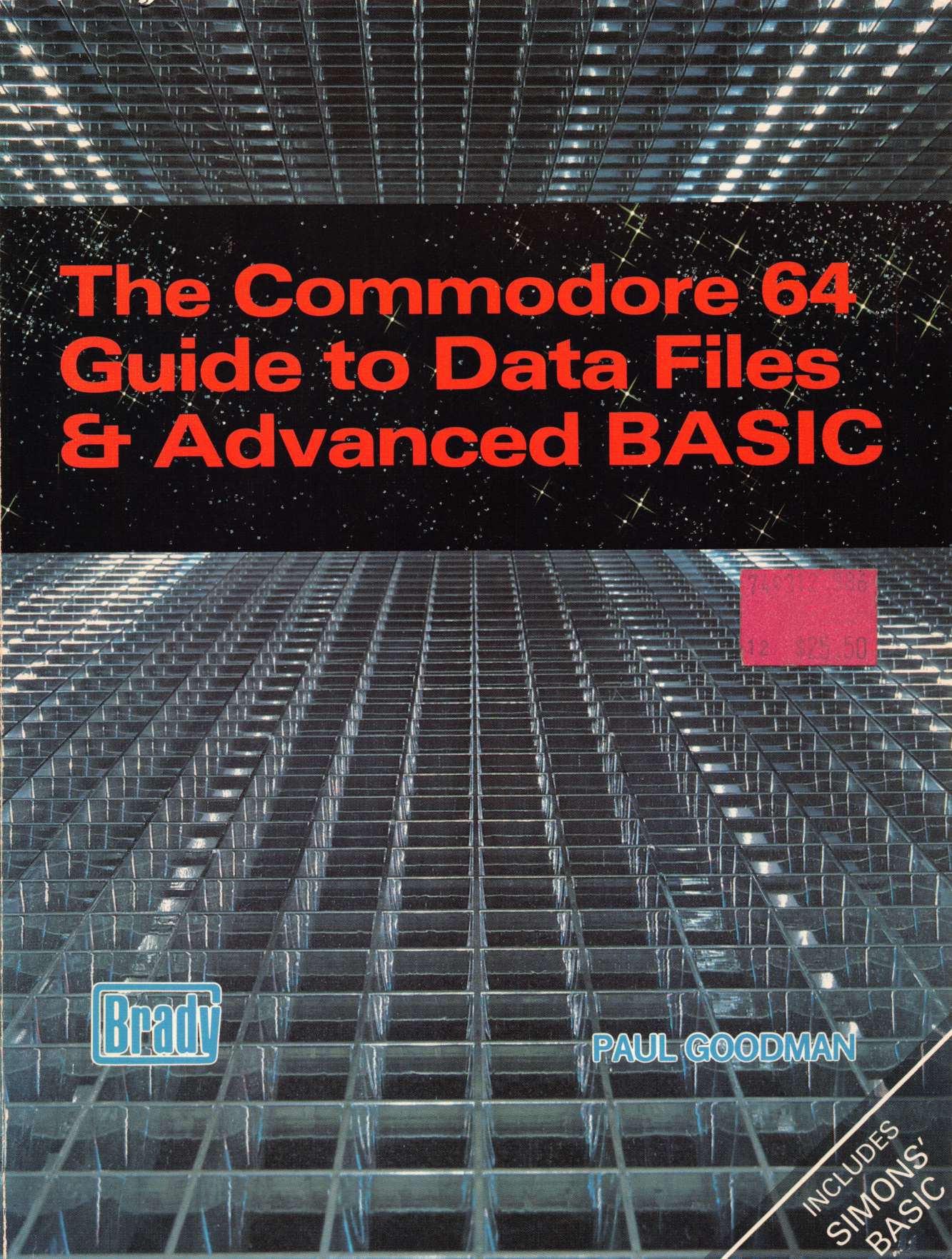

The Commodore 64
Guide to Data Files
& Advanced BASIC
12 $25.50
Brady
PAUL GOODMAN
INCLUDES
SIMONS'
BASIC

There are a few problems with parts of the Goodman text. Listed below are the corrections which I've discovered, but if you know of any others I'd like to hear of them.

Page 53, there should be semicolons in line 140 between the items. ie,
140 PRINT #2,A$;",";B$;",";C$

Page 57, at the bottom of the page line zero should be line six.

Page 58, at the top of the page line five should be line fifteen.

Page 64, IF LO>255 THEN HI=INT(RE/256):LO=RE-HI*256
=====

Page 69, line 170 T=VAL(T5$) should be T=VAL(T$)

Page 76, line 255 PRINT315 should be PRINT#15.

Page 137, page number is missing, and in the example the answer to the question NUMBER OF ITEMS should be 4, not 3.

Page 150, last entry in the right hand column should be ascii value 255 not 225.

SWST — T

MOORE PARAGON PARAFLO RE ORDER REF. 1115/BH

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50

The Commodore 64 Guide to Data Files & Advanced BASIC

The COMMODORE 64

Guide to Data Files & Advanced BASIC

Paul Goodman

Brady Communications Company, Inc.,

A Prentice-Hall Publishing Company
Bowie, Maryland

Library of Congress Cataloging in Publication Data

Goodman, Paul, 1958-
Commodore 64 guide to data files and advanced BASIC.

Includes index.
1. Commodore 64 (Computer)—Programming. 2. Basic (Computer program language) 3. Data structures (Computer science) I. Title. II. Title: Commodore sixty-four guide to data files and advanced BASIC.
QA76.8.C64G665 1984 001.64'2 84-18462

ISBN 0-89303-375-8

Prentice-Hall International, Inc., *London*
Prentice-Hall Canada, Inc., Scarborough, *Ontario*
Prentice-Hall of Australia, Pty., Ltd., *Sydney*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books, Limited, Petone, *New Zealand*
Editoria Prentice-Hall Do Brasil LTDA., *Rio de Janeiro*
Prentice-Hall Hispanoamericana, S.A., *Mexico*

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

Publishing Director: David Culverwell
Acquisitions Editor: Susan Love
Production Editor/Text Design: Barbara Werner
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa

Typesetting: Compolith Graphics, Inc., Indianapolis, Indiana
Printing: Fairfield Graphics, Fairfield, Pennsylvania
Copy Editor: Katherine Adams
Type faces: ITC Avante Garde Gothic (display), Elante (text), OCR-B (program)

Acknowledgements

Few people realize the amount of work required in the preparation of a book. The dedication of the Brady staff to this book easily matched my own.

My sincerest thanks to Sue Love and David Culverwell for their initial interest in the project and their encouragement. Thanks too, to Christi Mangold, and to Barbara Werner for their work in the production of the book. My deepest appreciation to my reviewers, John Bushery, Paul Pavnica, David Graham, Paul Blohm, Robert Darland, and Thomas Moore, for their suggestions and constructive criticism. Special thanks to Paul Pavnica for his reworking of Appendix H and to Alan Zeldin for his ideas.

Finally, belated thanks to the Queens College Department of Computer Science for the chance they took on a young grad student almost five years ago.

As always, "thank you" to my family and my friends for their support.

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs, the text, or the documentation contained in this book. The author and publisher shall not be liable in any event for claims of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the text or the programs. The programs contained in this book are intended for use of the original purchaser-user.

Contents

Introduction

Why should you buy this book instead of one of the five other books about the Commodore 64 sitting next to it on the shelf? Because this is not just another introductory BASIC text that shows you how to program the Commodore 64 to deal blackjack or play Beethoven's "Eroica." The Commodore 64 Guide to Data Files and Advanced BASIC is intended for the serious-minded Commodore 64 owner who wants to learn how to solve the sophisticated, substantial problems that they originally bought a computer for.

The key to writing advanced programs is knowing how to use data files. However, very little instruction is available on their use and on the different programming techniques used with files. This Guide is intended to fill that void.

Three assumptions have been made by the author. First, that you have access to a Commodore 64 with a disk drive. Second, that you have knowledge of BASIC programming: IF, GOTO, strings, how to write, run, and store a program. Finally, that you have a desire to apply the Commodore 64 to tasks you already do manually, from tracking stocks, to tracking horses, to tracking movie stars.

Among the many topics covered in The Commodore 64 Guide to Data Files and Advanced BASIC are:

Files and the Disk Drive—an overview of the disk system.

Sequential Files—when and how to use them and various programming techniques.

Relative Files—when and how to use them and various programming techniques.

Advanced Input/Output—how to get the keyboard and screen to do what you want.

Program Design Techniques—what to consider for designing better programs.

Advanced BASIC statements and structures.

These topics are not just introduced to the reader. They are presented along with programming techniques and program design considerations to develop a sense of how things fit together in the scheme of BASIC programs.

A major goal of The Commodore 64 Guide to Data Files and Advanced BASIC, is to provide and alternative to the Commodore 64 manuals in an exanded and comprehensible manner. No longer will you have to spend sleepless nights deciphering the 1541 disk drive manual in an attempt to understand file handling. All that information (and more) is clearly presented with detailed discussion and examples. To provide a quick reference, all instructions and syntax are laid out in a reference manual style.

Included both as a programming example and as an example of program design, is MAGICFILER. MAGICFILER is a publication-quality database manager for the Commodore 64, which compares in features to many other database managers now being sold. MAGICFILER will help you to easily store and manipulate large amounts of information. Written in BASIC, MAGICFILER is developed and explained on a section basis, showing how a major program is designed, constructed, and programmed. A complete, ready-to-type BASIC listing is also included.

The Commodore 64 Guide to Data Files and Advanced BASIC will provide you with the knowledge necessary to harness the power of your Commodore 64. After reading this book, you will have the ability and confidence to tackle problems that you thought were beyond your grasp.

1 basic review: the important statements

This chapter is intended to review and sharpen your understanding of four of the most powerful structures in BASIC; the IF statement, the FOR loop, subroutines and arrays. Taking a few minutes to review this material will help you get the most out of the rest of the book. Included also are several pertinent Simons' BASIC: extensions to Commodore BASIC. If you are not familiar with Simons' BASIC, see Appendix A.

decisions, decisions

One of the most powerful features of any programming language is its ability to make decisions. By comparing values, the IF statement can decide which statements to execute next. IF expression is true THEN do this statement.

Graphically it could be represented like this

```
100 IF A=7 THEN GOTO 200
```

Here is an example of an IF statement.

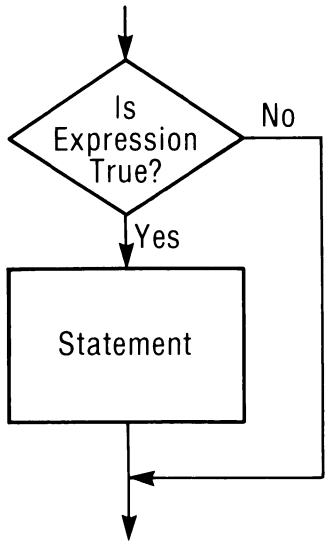


Fig. 1.1

The expression checked in this example is A=7. If it is true, then the GOTO is executed, otherwise it is skipped. The GOTO is optional. The equivalent statement is **100 IF A=7 THEN 200.**

The possible comparisons that can be used in an expression are

=	is equal to
<>	is not equal to
>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to

An expression can also be just a constant or variable. These expressions are all legal

```
IF A >      100 THEN 200
IF A >      B THEN 200
IF A+7 >    B THEN 200
IF A+7 >    B-3*4 THEN 200
```

a pop quiz

```
10 INPUT A
20 IF A > 10 THEN 30
25 PRINT "LESS"
30 PRINT "MORE"
40 END
```

IF 7 was entered, what will the output be?
O.K., time's up.

The output is

LESS
MORE

Note that if a line 27 were added,

```
10 INPUT A
20 IF A > 10 THEN 30
25 PRINT "LESS"
27 GOTO 40
30 PRINT "MORE"
40 END
```

the output would now be only

LESS

Expressions can be combined with logical operators. What's a logical operator, you ask? It is an algebraic notation that combines the answers of two expressions by an operator to produce a single value.

```
100 IF A=3 AND B=4 THEN 400
```

The preceding expression uses the AND operator. For the GOTO 400 to be executed, A must equal 3 and B must equal 4. The table of results for a logical operator, such as AND, is called a truth table. The truth table for AND is

Exp1	AND	Exp2	IS	Result
True		True		True
True		False		False
False		True		False
False		False		False

A second logical operator is OR. With OR, only one side of an expression must be True for the whole expression to be True. The truth table for OR is

Exp1	AND	Exp2	IS	Result
True		True		True
True		False		True
False		True		True
False		False		False

The only logical operator left, and the easiest to understand, is NOT. NOT simply reverses the value of an expression.

NOT(True) is False.
NOT(False) is True.

Let's try a few examples.

Assume A=4, I=4, and J=10. Is the expression (A=I)AND(NOT(J>A)) True or False?.

To evaluate this expression, we must first deal with what is inside the innermost levels of parentheses.

(A=I)AND(NOT(J>A))

We can now rewrite it like this

True AND (NOT(True))

Reducing once more we get

True AND False

Which is of course False.

Try this one on your own.

```
IF (I+4 >=J) OR (NOT(I=3)AND(J <A))
```

The answer is False. If you got it wrong, try again. Remember to start with the innermost set of parentheses.

The colon : is used to separate multiple statements on the same line. Multiple statements may be placed after THEN, but this could be confusing.

```
100 IF A=4 THEN PRINT "YES":GOTO 120
110 PRINT "NO"
120 END
```

What is printed? Will the GOTO be executed regardless of the value of A? If A equals 4, it will print YES, then GOTO 120, otherwise NO will be printed. What is noteworthy is that multiple statements after THEN are either all executed together or not executed at all. They are a matched set. This can lead to an unexpected problem in this situation

```
10 FOR I=1 TO 10:IF K=3 THEN 100:NEXT I
```

What's the problem here? Since NEXT is placed after THEN, it will only be executed when K equals 3. If K is not 3, then the loop is not complete and is exe-

cuted only once. This can be a very confusing bug to find since no error message is created. Instead, the program just doesn't act the way it is expected to. This type of bug is known as a logic error. The program runs fine, but the results are erroneous.

an iffy example

Let's write a program that reads three values and finds the largest. This is trivial, I know, but it's a good demonstration of the concept.

```
10 INPUT A,B,C
20 L=-99 :REM SET LARGEST VALUE TO A SMALL NUMBER
30 IF A>=B THEN L=A: GOTO 50
40 L=B
50 IF C>L THEN L=C
60 PRINT "THE LARGEST WAS",L
70 END
```

Pick a few sets of input and follow them through the program.

IF THEN ELSE

Many programming languages (but not BASIC) have a possible second clause in the IF statement, called ELSE. When ELSE is used, there are two mutually exclusive statements connected to the IF: one which is executed only if the expression is True (THEN), and one which is executed only if the expression is False (ELSE). A typical Pascal statement might look like this:

```
IF A=5 THEN WRITE("FLIP")
       ELSE WRITE("FLOP");
WRITE("FLUP");
```

Either FLIP or FLOP is printed depending upon the value of A. FLUP is written in either case. Since we are not fortunate enough to have an ELSE statement, we must simulate it. There are several good ways to do this.

```
100 IF A=5 THEN PRINT "FLIP":GOTO 120
110 PRINT "FLOP" :REM ELSE CLAUSE
120 PRINT "FLUP"
```

or

```
100 IF A<>5 THEN PRINT "FLOP":GOTO 120
110 PRINT "FLIP" :REM THEN CLAUSE
120 PRINT "FLUP"
```

In both cases one of the two clauses was executed and then control was branched around the other. In cases where a large number of statements are to be executed, an alternate structure is needed, as demonstrated below.

```
100 IF A=5 THEN 300
110 REM ELSE STATEMENTS START HERE
     .
     .
290 GOTO 500
300 REM THEN STATEMENTS START HERE
    .
    .
490 REM THEN STOPS HERE
500 PROGRAM COMES BACK TOGETHER HERE
```

Graphically it could be represented like this.

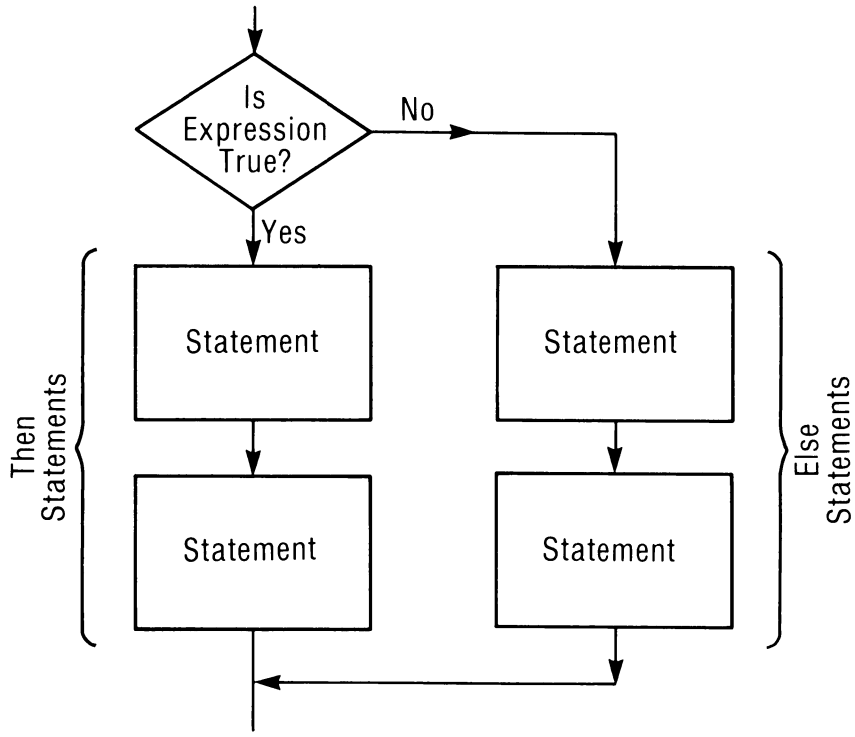


Fig. 1.2

FOR loops

When designing a programming language, programmers look for groups of statements that are commonly executed together with an eye towards combining these statements into a single, more powerful one.

A common example: Let's write a BASIC program to print the numbers 1 through 10 and their squares.

```
10 I =1
20 PRINT I, I*I
30 I=I+1
40 IF I≤10 THEN 20
50 END
```

Here, a loop was formed that will iterate ten times. What are the separate components of this loop?

- The Initialization
- The Body
- The Increment
- The Check and Branch

Here is the same program with its components pointed out.

```
10 I =1 :REM INITIALIZE
20 PRINT I,I*I :REM LOOP BODY
30 I=I+1 :REM INCREMENT
40 IF I≤10 THEN 20 :REM CHECK AND BRANCH
50 END
```

The FOR Loop takes care of most of these components automatically.

```
10 FOR I=1 TO 10
20 PRINT I, I*I
30 NEXT I
```

With the FOR Loop, a variable and its first and last values are given. The variable (in this case I) is first assigned the value 1 and the loop body is executed. When the NEXT statement is hit, the variable is incremented and tested. If it is less than or equal to the final value, the program branches back to do the loop body again. In this example, the loop body is executed ten times, once for each value of I from 1 to 10. When I equals 11 the loop ends and the following statement is executed. To be a little more formal, the actual form of the FOR statement is

FOR variable=initial value TO final value STEP increment value.

We have not seen the STEP option yet because we have always used its default value of one. If we want the variable to be incremented by a value other than one, we specify it after STEP.

```
10 FOR K= 10 TO 20 STEP 2
```

This would add two rather than one to K each time. The values of K would be 10,12,14,16,18,20. A negative STEP can also be used to subtract a number from the variable instead of adding to it.

```
10 FOR P2 = 10 TO 5 STEP -1
20 PRINT P2;
30 NEXT P2
```

This would print: 10 9 8 7 6 5

If line 10 read FOR I= 273 to 371, how many times would the body of the loop be executed? If you subtracted 273 from 371 and got the answer 98, you get an A for effort (shouldn't that be an E for effort?), but the correct answer is 99. Why? Well just remember that there are 10 numbers between 1 and 10 inclusive, so we must subtract.

Final value	371
− Initial value	273
	98
Then add 1	1
To get	99

nested FOR loops

```
5 I=0
10 FOR K=1 TO 3
20 FOR J=1 TO 2
30 I=I+1
40 NEXT J
50 NEXT K
60 PRINT I
```

What would be printed by this example?

The result printed is 6. Why? This situation is described as nested FOR loops. The inner loop is executed for all values of J and for each value of K in the outer loop. This can be better illustrated by adding a line 35.

```
5 I=0
10 FOR K=1 TO 3
20 FOR J=1 TO 2
30 I=I+1
35 PRINT K,J
40 NEXT J
50 NEXT K
60 PRINT I
```

The output would look like this:

```
1  1
1  2
2  1
2  2
3  1
3  2
6
```

more FOR examples

How could we write a program to calculate the factorial ! of a number, e.g., 3!=3*2*1, 4!=4*3*2*1

```
10 INPUT "ENTER NUMBER";N
20 F=1
30 FOR I=1 TO N
40 F=F*I
50 NEXT I
60 PRINT N;" FACTORIAL IS ";F
```

Let's assume N is 4 and track the FOR loop as it executes.

I	F	
-	1	
1	1	
2	2	
3	6	
4	24	The final value of F
Done		

How about a program to raise a number to a power?

```
10 INPUT "BASE NUMBER";B
20 INPUT "POWER";P
30 AN=1
40 FOR I=1 TO P
50 AN=AN * B
60 NEXT I
70 PRINT "ANSWER IS"; AN
```

All that was done here was to use a FOR loop to multiply B by itself, P times. Let's assume B is 4 and P is 3.

I	AN	B	P
-	1	4	3
1	4	4	3
2	16	4	3
3	64	4	3

Done

FOR loops are important to us for three reasons:

1. It is easier to write a program with them than without them.
2. They are handy when using arrays. (Arrays will be reviewed later in this chapter.)
3. They execute faster than a loop written without a FOR. In the example on squaring numbers the FOR loop executed in .6 seconds, the other loop in .66 seconds. That's a 10% improvement which can be vital in time-dependent situations.

subroutines

Question: What can make a program

- More organized?
- Easier to write?
- Easier to read?
- Shorter?

Answer: Subroutines.

If a program must execute the same set of instructions several times, we can place those instructions in a subroutine. A subroutine is a section of a program that is not executed in the same sequence with which we are familiar. The code in a subroutine lies dormant until it is called upon to execute by the program. When the subroutine is called it becomes active and executes. When it is done, it transfers control back to the statement after the statement that called it. This is best illustrated by an example.

```
 10 GOSUB 100
 20 PRINT "BACK"
 30 END
 99 THE SUBROUTINE STARTS HERE
100 PRINT "HERE"
110 RETURN
```

The output is

```
HERE
BACK
```

What happened? The lines were executed in this order: 10,100,110,20,30. The subroutine (lines 100 and 110) is called with the GOSUB statement. When it is encountered, it does the same thing as a GOTO, except the line number of the GOSUB is saved as a mechanism for the subroutine to find its way back to the main program. The RETURN statement (not to be confused with the return key on the keyboard) reverses the process, taking the address saved by GOSUB and branching to the line after that address. In this case it was line 20.

Another example:

```
10 A=4
20 GOSUB 60
30 GOSUB 60
40 PRINT A
50 END
60 A=A+1
70 RETURN
```

What will be the output? The correct answer is 6. The subroutine at line 50 is called twice, by the GOSUB on line 20 and by the one on line 30. Each time it returns to the line following the one that calls it.

Subroutines can also call subroutines, as shown in this example

```
 10 A=4
 20 GOSUB 50
 30 PRINT A
 40 END
 50 A=A+2
 60 GOSUB 100
 70 RETURN
100 A=A-1
110 RETURN
```

What will be printed? The answer is 5. The lines are executed in this order: 10,20,50,60,100,110,70,30,40. We can think of this example in terms of levels. The program is originally on level 1. Each GOSUB drops it down one level and each RETURN raises it up one level

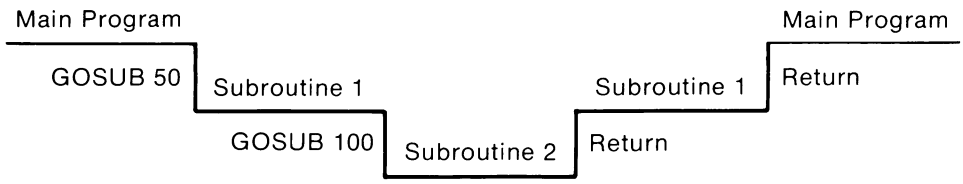


Fig. 1.3

a look inside

Obviously, in the last example, more than one return address must be saved if the program is to find its way back to the original starting point and more than one location will have to be used to save the return addresses. A stack is used for this purpose. A stack is a LIFO (last-in, first-out) structure. Picture a stack of dishes on a table. You always take a dish from the top of the stack and always place a dish back on top of the stack. The action of placing something on top of the stack is called a *push*, and taking something off the stack is called a *pop*. Return addresses are handled in the same way. When a GOSUB is executed, the return address is pushed on the stack. If two GOSUBs have been executed, the address on top is that of the most recent GOSUB. A RETURN pops one off the top. Through this mechanism, it is possible to keep track of the proper address to return to without a lot of fuss.

It is obvious that these examples are trivial, but later on we will use subroutines that contain instructions we wish to execute from different sections of a program. This will save both memory space and the trouble of retyping the same code more than once. For example, we will use subroutines that position the cursor and that accept input from the keyboard.

When we use subroutines, we generally use certain variables to send information to the routine. Certain other variables are used to return information back to the main program. This sending of information back and forth between subroutines and a main program is called *parameter passing.*

arrays

The need for arrays can be demonstrated with three short examples.

Question: Write a program that reads two numbers and finds their average.

Answer:
```
10 INPUT A,B
20 AV=(A+B)/2
30 PRINT AV
```

Question: Write a program that reads four numbers and finds their average.

Answer:
```
10 INPUT A,B,C,D
20 AV=(A+B+C+D)/4
30 PRINT AV
```

Question: Write a program that reads 100 numbers and finds their average.

What is the difficulty here? We would need 100 distinct variables to handle this problem in the same way as the others, but the handling of so many variables would be awkward at best, and sometimes even impossible. Arrays provide a solution to this problem. An array is a set of variables, all called by the same name.

"How can that be?" The answer is that they actually have two names. Each individual variable in an array (called an element) is referred to by two names, the array name and a subscript. Rather like a family name and a first name. The rules for naming arrays are the same as for variables: up to two characters starting with a letter, plus the subscript, which is a number. Unlike regular (scalar) variables, arrays have to be declared in the beginning of a program. The DIMENSION statement is used for this purpose.

```
10 DIM A(10)
```

This creates an array A with 11 elements named A(0), A(1), . . . ,A(10). *Note:* Commodore BASIC differs from many others in this respect. In other BASICs only ten elements would be created, A(1) through A(10). In most examples we will ignore the zero element, since it is usually easier to program without it. Don't forget that the variable A still exists as a variable independent of the array named A. Arrays are usually pictured as lists.

A(0)	0
A(1)	0
A(2)	0
A(3)	0
A(4)	0
A(5)	0
A(6)	0
A(7)	0
A(8)	0
A(9)	0
A(10)	0

Fig. 1.4

Assignment statements for array elements look like you would expect them to. Both the array name and subscript are used.

```
10 A(3)=4
20 A(5)=6
30 A(10)=A(3)
```

The same array after the above statements are executed would appear like this.

A(0)	0
A(1)	0
A(2)	0
A(3)	4
A(4)	0
A(5)	6
A(6)	0
A(7)	0
A(8)	0
A(9)	0
A(10)	4

Fig. 1.5

The real power of arrays comes with using variables as subscripts and then changing the subscripts in a FOR loop. As the old song goes, FOR loops and arrays "go together like horse and carriage."

some examples

```
10 DIM B(4)
20 FOR I= 1 TO 4
30 B(I)=I
40 NEXT I
```

The resulting array looks like this

B(1)	1
B(2)	2
B(3)	3
B(4)	4

Fig. 1.6

pop quiz

Do the same thing as above except reverse the order of the numbers, i.e., end up with an array that looks like this

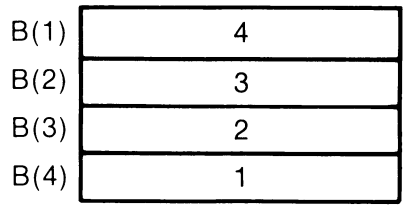


Fig. 1.7

The answer is

```
10 DIM B(4)
20 FOR I= 1 TO 4
30 B(I)=5-I
40 NEXT I
```

Most people would have come up with this code instead

```
20 FOR I= 4 TO 1 STEP -1
30 B(I)=I
40 NEXT I
```

This will produce the same result as the original example B(4)=4 . . . , except that the array will be filled from the bottom up. What was needed was a way to have the values assigned to the array decrease as the subscripts increase. The B(I)=5-I is a cute trick that probably was not obvious to you. A more brute force attack might have produced:

```
15 K=4
20 FOR I= 1 TO 4
30 B(I)=K
40 K=K-1
50 NEXT I
```

Here the value assigned to the array is no longer tied to the subscript.

Now let's see the solution to the averaging problem presented earlier.

```
10 DIM A(100)
19 REM LOOP TO READ IN VALUES
20 FOR I=1 TO 100
30 INPUT A(I)
40 NEXT I
49 REM THIS LOOP FINDS THE AVERAGE
50 S=0
60 FOR I=1 TO 100
```

```
70 S=S+A(I)
80 NEXT I
90 AV=S/100
100 PRINT AV
```

lotto player

Lotto is a game run in many states as a form of legalized gambling. Players pick 6 numbers out of 40, trying to match the official 6 numbers picked. Having the Commodore 64 pick the numbers seems as good a method as any. An array is used to keep track of numbers already generated to eliminate duplication. If you are not familiar with the method of generating random numbers, here is a short review.

The RND function creates a number at random between 0.0 and 1.0. INT(RND(0)*40) creates a number between the integers 0 and 39 inclusive. Since the range we want is 1 to 40, simply add 1.

```
X=INT(RND(0)*40)+1
```

First we set up the array

```
5 DIM A(6)
```

pick a number

```
10 N=INT(40*RND(0))+1
```

check to see if it is a duplicate

```
20 FOR K=1 TO 6
30 IF A(K)=N THEN 10:REM DUPLICATE FOUND
40 NEXT K
```

new number found, assign it the array

```
45 CT=CT+1: REM COUNTS HOW MANY FOUND
50 A(CT)=N
```

check to see if we have six numbers

```
55 IF CT=6 THEN 70
60 GOTO 10
70 PRINT "THE NUMBERS ARE"
80 FOR I=1 TO 6
85 PRINT A(I)
90 NEXT I
```

two-dimensional arrays

Arrays come in multi-dimensional versions as well. Two-dimensional arrays have their elements arranged in a grid of rows and columns.

```
10 DIM B(3,2)
```

This statement dimensions an array of 3 rows and 2 columns. The row is always specified first. Array B can be pictured like this

	Columns		
Rows	0	0	0
	0	0	0
	0	0	0

Fig. 1.8

Each element in the array has to be identified by two subscripts, one for the row and one for the column, such as B(row,col.). The address of each element in this array is

B (1,1)	B (1,2)	B (1,3)
B (2,1)	B (2,2)	B (2,3)
B (3,1)	B (3,2)	B (3,3)

Fig. 1.9

The use of a two-dimensional arrays is not much more difficult than a single dimensional array once you get accustomed to manipulating the two separate dimensions.

Using our array B, let's add up the contents of the elements in the first column. To access each element in a column, we must hold the subscript for that column constant as we vary the rows.

```
10 FOR I=1 TO 3
20 S=S+B(I,1)
30 NEXT I
```

To add up the contents of a row, such as the second row, we hold the subscript for the row constant and vary the columns.

```
10 FOR J=1 TO 2
20 S=S+B(2,J)
30 NEXT J
```

Arrays are one of the most powerful and useful structures in BASIC. They will be used extensively as new concepts are introduced in future chapters.

simons' BASIC extensions

Simons' BASIC includes several new loops and instructions which are very useful in programming. If you are not familiar with Simons' BASIC, turn to Appendix A.

IF expression THEN true statement :ELSE: false statement

Simons' BASIC includes the IF..THEN..ELSE statement. The merits of this instruction have already been discussed, only the syntax needs to be described.

```
IF I=4 THEN PRINT "YES" :ELSE:PRINT "NO"
```

The ELSE is encased by two colons. Unfortunately the entire statement is still limited to two screen lines.

RCOMP:true statement:ELSE:false statement

RCOMP repeats the test done in the most recently used IF..THEN..ELSE statement. Using RCOMP limits the readability of a program and is a bad programming practice. In a program very tight for memory space, RCOMP can be useful, otherwise it should be avoided.

REPEAT..UNTIL condition is true

REPEAT is an extremely powerful loop found in most of the newer programming languages, such as Pascal, C, and Ada. The REPEAT loop operates by executing the statements in the loop's body then checking if the condition is true. If the condition is not true, the loop body is executed again, and so on.

```
110 REPEAT
120 PRINT I,
130 I=I+1
140 UNTIL I=5
```

In this example the output is : 0 1 2 3 4. The REPEAT loop can be graphically represented as shown in Figure 1.10.
Notice the essential difference between the REPEAT and the FOR loops. With the FOR loop, the number of iterations is known before the loop starts to execute. This is called a *bounded loop*. With the REPEAT loop, the number of iterations is

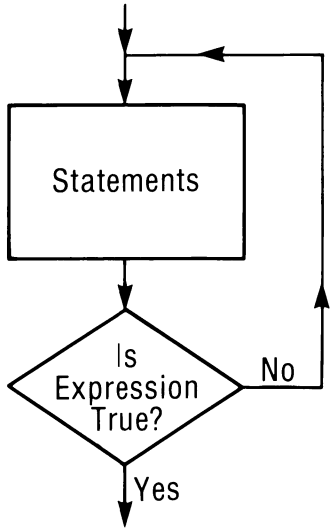


Fig. 1.10

not known, but is dependent upon action in the loop body. This is called a *free loop.*

Two more points about the REPEAT loop:

1. The statements in the loop body are always executed at least once.
2. The condition must be altered in the loop, otherwise an endless loop will occur. For example, the following loop will execute infinitely.

```
100 I=0
110 REPEAT
120 PRINT I,
130 UNTIL I=5
```

REPEAT it again, sam

In the following example, a REPEAT loop is used to remove any leading blanks in the string D$.

```
100 INPUT D$
110 IF D$="" THEN 150
120 REPEAT
130 IF LEFT$(D$,1)=" " THEN D$=RIGHT$(D$,LEN(D$)-1)
140 UNTIL LEFT$(D$,1)<>" " OR LEN(D$)=0
150 REM program continues here
```

faking REPEAT

The REPEAT loop can easily be simulated without any Simons' BASIC statements. Here is the same program except with no REPEAT loop

```
100 INPUT D$
110 IF D$="" THEN 150
120 REM START OF LOOP
130 IF LEFT$(D$,1)=" " THEN D$=RIGHT$(D$,LEN(D$)-1)
140 IF NOT(LEFT$(D$,1)<>" " OR LEN(D$)=0) THEN 130
150 REM program continues here
```

LOOP/EXIT IF condition/END LOOP

Simons' BASIC includes another new loop structure. In this loop the check of the condition can be placed anywhere in the loop body. The condition is checked by the EXIT IF statement. The loop is formed by the LOOP and END LOOP statement bracketing the statements in the body of the loop. Here is the same program segment rewritten with LOOP..END LOOP.

```
100 INPUT D$
120 LOOP
130 EXIT IF D$=""
140 EXIT IF LEFT(D$,1)<>" "
150 D$=RIGHT$(D$,LEN(D$)-1)
170 END LOOP
180 REM program continues here
```

This structure can also be used without the EXIT IF to purposely form an endless loop. This is sometimes done to prevent a user from breaking out of a program.

procedures

Procedures are the same as subroutines. The only difference is that they are named and are called by a different statement. The reason for procedures is to make programs more readable and understandable by giving self-explanatory names to subroutines.

PROC procedure name/END PROC

The PROC statement is used to label a subroutine. The procedure name can be any string of characters. END PROC is used to end a procedure and is equivalent to the RETURN statement. PROC can also be used to label a section of code in a program. When this is done, the END PROC statement is not used.

EXEC procedure name

EXEC is equivalent to the GOSUB statement. The only difference is that a procedure name is used instead of a line number.

CALL procedure name

CALL is equivalent to the GOTO statement. It is used to branch to a labeled section of code rather than to a procedure. This creates a confusing situation, since procedure names are used to label both procedures and sections of a program that are not procedures. When a program is written with CALLs instead of GOTOs, the Simons' BASIC RENUMBER function can be used.

```
10 A=4
20 B=3
30 EXEC ADDEMUP
40 PRINT A,B
50 CALL ENDIT
100 PROC ADDEMUP
110 A=A+B
120 END PROC
140 PROC ENDIT
150 PRINT "ALL DONE"
160 END
```

In this example, the PROC statement was used to label both a procedure ADDEMUP and a program section ENDIT. The output would be 7 and 3. The line were executed in this order: 10,20,30,100,110,120,40,50,140,150,160.

2 screen i/o

Many programs that you may write will require the user of the program to enter data while the program runs. As a programmer, you must choose the method by which data will be accepted. This function of requesting and accepting data from the screen is one of the most important aspects of programming. It is known as the user interface, that part of a program which interacts with humans. A well-designed and easy-to-use user interface makes a program more desirable to use and therefore user-friendly. In this chapter we will develop several user-friendly input routines and techniques as well as some output techniques. Some knowledge of both subroutines and strings is required.

INPUT

Commodore BASIC provides two commands to accept data from the keyboard, INPUT and GET. The INPUT statement in its simplest form is

```
10 INPUT A
```

INPUT halts the running program and displays a question mark on the screen with the blinking cursor next to it. It then waits for the user to enter some data followed by a carriage return. The value that is entered is assigned to the variable A. In the above example, since the variable A is a numeric variable, a number must be entered. If characters were entered instead, the operating system (also called the Kernal) would display the error message **REDO FROM START??** and then wait for the correct data to be entered.

The statement, **10 INPUT A$** accepts for input a string of up to 255 characters and assigns it to the string variable A$. Any of the keyboard characters, including the numbers, the graphics set, and the screen codes, could be entered.

When INPUT is executed, the prompt **?** is displayed. This is not very descriptive and would leave most users wondering what to do next. To let the user know what input is expected and at the same time cut down on some of the frustration of dealing with a computer, BASIC allows us to include a prompt with the INPUT statement.

```
10 INPUT "ENTER YOUR AGE";AG
```

when run will display

```
ENTER YOUR AGE?
```

This is an improvement over just the **?** prompt. Notice the semicolon before the variable AG. This is the only syntax permitted, and omitting it is a common source of errors.

validity checking

Often, in a program, a value to be entered must fall within a certain range. Checking to see if what is entered is valid is one of the most important and, unfortunately, one of the most tedious aspects of developing user-friendly programs.

One approach to validity checking is to display an error message when invalid input is entered and then prompt the user to enter the data again.

```
10 INPUT "ENTER YOUR AGE";AG$
20 AG = VAL(AG$)
30 IF AG > 0 AND AG < 100 THEN 60
35 REM HERE IS THE ERROR MESSAGE
40 PRINT"PLEASE CHECK AND REENTER AGE"
50 GOTO 10
60 REM CONTINUE PROGRAM HERE
```

The IF statement in line 30 checks to see if the input falls in the proper range of 1

to 99. Notice how the INPUT statement accepts a string that is then converted to a number with the VAL function. This is done to prevent the system error message from appearing if a non numeric value is entered.

In a slightly different approach, when an invalid value is entered, the program will beep, display an error message for two seconds, and then clear the screen to start over.

```
10 PRINT CHR$(147) :REM CLEAR THE SCREEN
20 PRINT "ENTER"
30 PRINT "A FOR ADD"
40 PRINT "S FOR SUBTRACT"
50 INPUT "ENTER SELECTION";T$
60 IF T$="A" OR T$="S" THEN 110
69 REM INVALID INPUT
70 PRINT "REENTER SELECTION"
80 GOSUB 500 :REM CALL BEEP ROUTINE
90 FOR I=1 TO 1500:NEXT I:REM DELAY
100 GOTO 10
110 REM CONTINUE PROGRAM HERE
```

Note that line 80 calls a subroutine at line 500 that will cause a beep to sound. This routine, which is a series of POKEs into registers of the SID chip (the Commodore 64's sound synthesizer), will not be explained in this book, but the routine is yours to use. For more information on using the SID, refer to the Commodore 64 User's Guide.

the beep routine

```
500 POKE 54296,15 :POKE 54277,0
501 POKE 54278,240 :POKE 54273,14
502 POKE 54272,37 :POKE 54276,17
503 FOR V=1 TO 50 :NEXT V
504 POKE 54276,0 :POKE 54277,0 :POKE 54278,0
505 RETURN
```

the keyboard buffer

"When your fingers hit a key, a lot of stuff happens inside"

—An eight-year-old philosopher

When a key is pressed on the keyboard, it causes a long sequence of steps to be performed by the computer. The Kernal takes the ASCII code corresponding to that key and places it into the keyboard buffer in first in, first out order (also known as FIFO or a queue). The keyboard buffer is a special set of ten memory locations whose job is to hold characters that are typed on the keyboard until they are requested by an INPUT or GET statement. The buffer allows you to type

faster than the program is processing the input and yet not lose the data. This type of setup is sometimes called a type-ahead buffer. It is common in devices that print more slowly than they process data, such as printing calculators and cash registers. To see the effect of the keyboard buffer, try this short experiment:

```
10 PRINT "ENTER A NUMBER"
20 FOR I=1 TO 1300: NEXT I
30 INPUT A$
```

What happened? If you entered the data at the prompt, it wasn't echoed (printed) until after the loop on line 20 completed. Where was it in the meantime? In the keyboard buffer, of course.

the GET statement

For simple applications, the INPUT command is very useful, but, as noted before, there are several drawbacks to using INPUT:

1. The ? prompt is always there whether you want it or not (just like crabgrass).
2. A carriage return must be entered.
3. The input is always echoed (printed on the screen), which is undesirable in certain applications. There are a few other problems that we will soon see.

A second statement that accepts input from the keyboard, GET, will give us more flexibilty.

GET a$

GET takes a single character from the keyboard buffer and assigns it to the variable specified (in this case A$). The character is not echoed to the screen. Since GET works by taking the first thing it finds in the keyboard buffer, if nothing is there that's what it takes, nothing; or what we call the null string. The null string is a string that has no contents. It is written with two double quotes next to each other like this "". So, when using GET, it is necessary to see if a character was got or not.

```
10 GET T$ :IF T$ ="" THEN 10
```

This will check the character that was picked up. If it is null, it then repeats the GET until a character is entered.

a common use of the GET statement

In programs it is common to "freeze" output on the screen until the user hits a key. This is simple to do.

```
880 PRINT whatever
890 PRINT "<HIT ANY KEY TO CONTINUE>"
900 GET A$:IF A$ ="" THEN 900
910 GOTO 100
```

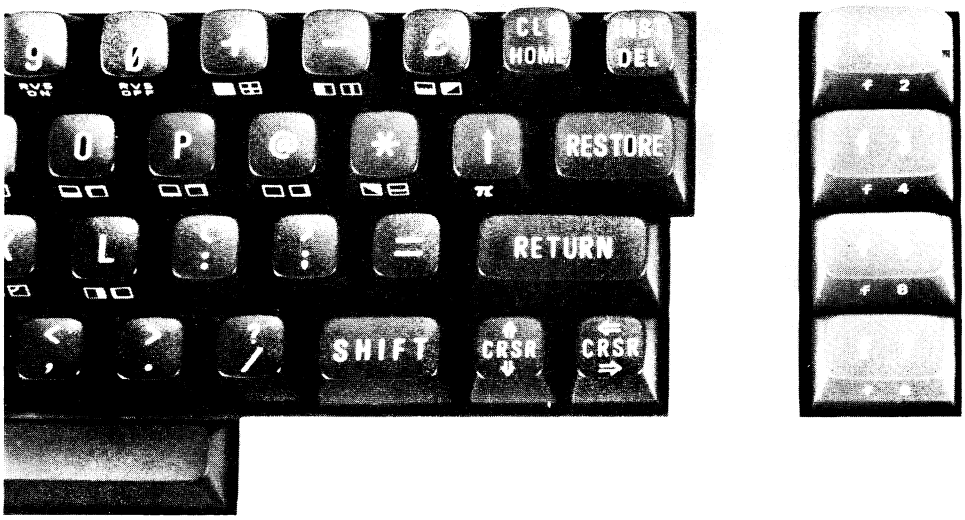
Here the GET on line 900 loops until any key is hit. Notice nothing is done with the variable A$. It is a dummy argument used only because GET requires a variable be used as its argument.

Another common use of the GET is in situations where you want only a single character entered and don't want to have to enter a carriage return. A good example of this is in a menu (a list of possible program options).

```
100 PRINT CHR$(147):REM CLEAR SCREEN
105 PRINT "DINNER CHOICES"
110 PRINT "1) STEAK LA COMMODORE
120 PRINT "2) FISH FOR 64
140 PRINT "ENTER SELECTION ";
150 GET A$ :IF A$="" THEN 150
160 A = VAL(A$)
170 IF A<1 OR A>3 THEN 100
180 PRINT A$ :REM ECHO CHOICE
190 ON A GOTO 200,300,400
```

Notice that on line 180 we have to print A$, since GET didn't. In order to print A$ next to the ENTER SELECTION prompt, the PRINT on line 140 ends in a semicolon, which causes no carriage return to be performed after the PRINT.

function keys



Maybe the most mysterious feature of the Commodore 64's keyboard is those four brown keys sitting snobbishly by themselves on the righthand side of the keyboard. These are the function keys, (labeled f1, f3, f5, f7 on the top, and f2, f4, f6, f8 on their sides). By simply pressing them, you have the power to command programs to perform miraculous tasks. I hate to be the one to bring bad news, but function keys are not what they are cracked up to be. As a matter of fact, if you really want to get down to it, there is no difference between the function keys and the other keys, except that the function keys are provided as sort of generic keys for programmers to define in their program. Like all the other keys, they produce an ASCII code when pressed, but since they are unlabeled, programmers usually use them to allow users to specify options in a program. For example, in a word processor program, f1 might mean delete a word. In machines without function keys, CTRL-D would have been used for this task. Thus the function keys replace the need for complex key strokes. Granted, this is not a great advantage, but some people find it useful.

The ASCII codes for the function keys are listed below.

f1 - 133
f3 - 134
f5 - 135
f7 - 136
f2 - 137
f4 - 138
f6 - 139
f8 - 140

Using function keys is quite simple; for example, to see if f1 has been entered.

```
10 GET A$ :IF A$ = "" THEN 10
20 IF A$=CHR$(133) THEN GOTO 300
```

The ASCII code of the key pressed is examined to see if it is the desired function key. If it is the program branches to the designated line.

a better INPUT routine

As we noted before, the many drawbacks associated with the INPUT statement make it almost too dangerous to use in a program. Some of these problems are:

1. Many applications require a limit be placed on the maximum number of characters that can be entered (e.g. when entering a Social Security number or when trying to make sure that input doesn't wrap around the screen to the next line). The INPUT statement is not able to provide this type of limit.

2. The INPUT statement will "bomb", causing the program to stop and an error message to be displayed if more than 77 characters are entered into a string variable.
3. The INPUT statement, like a sleazy bar, will let just about any character enter, including the screen codes and the graphics characters. Limiting data entry to a range of characters would cut down on potential mistakes by a user.
4. While using the INPUT statement, the delete key can erase not only the typed characters, but the prompt and anything else on the screen as well. A most undesirable attribute.

In order to avoid these drawbacks, we can easily design a subroutine which can be used instead of INPUT. Some of the features we would want to include are:

1. The ability to limit the set of characters that will be accepted as input.
2. The ability to limit the length of the string input.
3. The ability to stop the DELETE key when it gets to the end of the entered string.

A good way of developing a complex program is to write it first in pseudocode. Pseudocode is a cross between English and BASIC in which the instructions are not written, but described in detail. I have found this to be far more valuable and helpful tool than flowcharting. Once a program is written in pseudocode, it is simple to convert it to BASIC.

A. Accept a character from keyboard
   If it is a carriage return, then RETURN
   If it is a delete, GOTO B
   Check for valid character and length of string
   IF OK
   Add character to string
   Print character
   GOTO A

B. If LEN(string) > 0 then
   remove rightmost character from string
   Erase character on screen
   GOTO A

As you might expect, this routine will be built around GET.

```
805 GET T$: IF T$="" THEN 805
```

The only problem with using GET is that no cursor is displayed. Displaying a cursor, preferably blinking, is a prerequisite for any good input routine. Below we will see how to simulate a cursor.

The routine will concatenate (place together in the same string) the individual characters entered into the the string TE$. In line 805, we picked up a character from the buffer. Now let's examine it.

1. If it is a return, then input is complete and RETURN from the subroutine.
2. If it is a DELETE, then delete the last character if there is one and GOTO 805 to get another character.
3. If it is a legal character and the number of characters is under the limit, accept it and GOTO 805.

First let's check the character entered.

```
810 IF T$=CHR$(13) THEN RETURN: REM WE HAVE A CARRIAGE
RETURN
815 IF T$=CHR$(20) THEN 860: REM DELETE KEY
819 REM CHECK IF A VALID CHARACTER
820 IF T$<CHR$(32) OR T$>CHR$(95) THEN 805
```

At this point we have a legal character falling in the range of the capitalized letters and numbers. Before we add it to the string being built, TE$, let's see if TE$ is at the maximum length by comparing its length to the maximum length represented by LI.

```
825 REM WE HAVE A GOOD CHARACTER
830 IF LEN(TE$)>=LI THEN 805
840 TE$=TE$+T$: REM ADD CHARACTER TO STRING
850 PRINT T$; :REM ECHO THE CHAR.
855 GOTO 805: REM GO BACK FOR MORE
```

At 860 we handle the delete key. We have to check TE$ to see if there are any characters to delete and then remove the rightmost character from two places, the string TE$ and the screen. To erase from the screen, we backspace `(CHR$(157))` over the spot, print a space, and then backspace again. Please note that `BS$=CHR$(157).`

```
860 IF LEN(TE$)=0 THEN 805: REM NOTHING TO DELETE
861 REM DELETE IN TWO STEPS
865 TE$=MID$(TE$,1,(LEN(TE$)-1))
870 PRINT " ";BS$;BS$;" ";BS$;
875 GOTO 805
```

Now all that is left to be done is to intialize TE$ and a string containing the backspace character at the start of the routine.

```
800 TE$="":BS$=CHR$(157)
```

We must also do something about simulating the cursor. First you must realize that the cursor has nothing magical about it. It is just an alternating positive and reverse space that is constantly written to the screen by the Kernal in a fashion similiar to what we will now do (except that it is done in machine language).

We can simulate the cursor by calling a small subroutine from inside our GET line.

```
805 GET T$:GOSUB 900:IF T$="" THEN 805
```

Now on every iteration of that loop, the routine at line 900 is called. At 900 we simply alternate the writing of a blank in reverse and normal mode. This will give a blinking effect. To keep the speed of the blinking slow, we will write the blank only every tenth time the routine is called. We will use FL (for flip) to keep track of which mode we will print in reverse (−1) or normal (+1). When the blank is printed FL will be flipped for the next time. CT keeps tracks of how many times the routine has been called since the last time it was printed.

```
900 CT=CT+1
905 IF CT<>10 THEN RETURN: REM DONT PRINT THIS TIME
908 IF FL=1 THEN PRINT " ";BS$;
910 IF FL=-1 THEN PRINT CHR$(18);" ";CHR$(146);BS$; :REM
PRINT IN REVERSE
915 CT=0 :REM RESET COUNTER
920 FL=FL*-1 :REM FLIP FL
925 RETURN
```

If you want to change the speed of the blinking, just adjust the number tested for on line 905.

Here is the input routine all together. If you have the diskette that is available to accompany this book the routine can be found in the file named BETTERINPUT.

```
800 TE$="":BS$=CHR$(157):FL=1
805 GET T$:GOSUB 900:IF T$="" THEN 805
810 IF T$=CHR$(13) THEN RETURN: REM WE HAVE A CARRIAGE
RETURN
815 IF T$=CHR$(20) THEN 860: REM DELETE KEY
820 IF T$<CHR$(32) OR T$>CHR$(95) THEN 805: REM RANGE OF
LEGAL CHARACTERS
825 REM WE HAVE A GOOD CHARACTER
830 IF LEN(TE$)>=LI THEN 805
840 TE$=TE$+T$: REM ADD CHARACTER TO STRING
850 PRINT T$; :REM ECHO THE CHAR.
855 GOTO 805: REM GO BACK FOR MORE
859 REM DELETE KEY
860 IF LEN(TE$)=0 THEN 805
861 REM DELETE IN TWO STEPS
865 TE$=MID$(TE$,1,(LEN(TE$)-1))
870 PRINT " ";BS$;BS$;" ";BS$;
875 GOTO 805
900 CT=CT+1
```

```
905 IF CT<>10 THEN RETURN: REM DONT PRINT THIS TIME
908 IF FL=1 THEN PRINT " ";BS$;
910 IF FL=-1 THEN PRINT CHR$(18);" ";CHR$(146);BS$; :REM
PRINT IN REVERSE
915 CT=0 :REM RESET COUNTER
920 FL=FL*-1 :REM FLIP FL
925 RETURN
```

To use the routine place the length of the input in variable LI in the main program and call it. For example:

```
300 PRINT "ENTER SOCIAL SECURITY NUMBER";
310 LI=9:GOSUB 800 :REM CALL INPUT ROUTINE
315 SS$=TE$ :REM USE STRING RETURNED
```

trapping function keys

Many programs use function keys to allow the user to enter a program option at any point in the program, even while entering other data. Typically, this is done in a word processing program when standard characters are typed to the screen and function keys are used to activate functions like delete, search, etc. This is a simple technique based upon the routine we just developed. After a character is read with the GET statement, a check is made to see if it is a function key. If it is, a subroutine is called to process it. Just one line must be added to our input routine, which tests the ASCII value of the character to see if it one of the function keys.

```
817 IF ASC(T$)>=133 AND ASC(T$)<=140 THEN GOSUB subroutine
```

masked input

In some applications it would be useful to have a flexible input routine that would display as a prompt a mask(or form) for the desired data. The purpose is to give the user a guide while entering data. For example:

```
ENTER SOCIAL SECURITY NUMBER:  . . . -..- . . .
```

The routine to do this is quite interesting and challenging to write. It should be general enough to

- Allow different masks to be used.
- Allow different punctuation symbols—-/, :, −, etc.
- Allow limiting the range of input at each position.

Here is the pseudocode:

- Display mask

- Move cursor over rightmost position
- Set mask pointer to 1
- Examine mask for first position

A. Accept a character (GET)
If it is a carriage return, RETURN
If it is a delete, GOTO B
If it is position for a letter, check validity
If not valid, GOTO A
If it is position for a number, check validity
If not valid, GOTO A
Place character in string
Increment mask pointer
If pointer > LEN(mask), GOTO A
Check next mask position
If position for a letter or number, GOTO A
If punctuation, skip past it & add symbol to string
GOTO A

B. Decrement pointer
Remove deleted character from string
Replace dot or punctuation symbol
GOTO A

writing the routine

- The general form for the mask is "9X-99" where
- a 9 is a position for a number
- an X is a position for a letter
- the - is a punctuation symbol

The variable M$ will hold the mask, M$="9X-99"

Points to be considered in writing the routine are

- Allow only the proper input in each position
- The user should not enter punctuation symbols
- The punctuation symbols should be jumped over by the delete key

The basic strategy is to

- Display the mask with .'s in the input positions
- Backspace to the first position
- Use a pointer (I) and the MID$ function to look at the proper input foreach position as it is being entered

Variables Used
M$ holds the mask
LI length of mask

I	current position in mask
TE$	string being built
T$,C$	scratch strings

If you have the available diskette you can find this routine in a file named **MASKEDIN**.

```
397 REM ********************
398 REM * INPUT WITH MASKS *
399 REM ********************
400 LI = LEN(M$)
405 TE$=""
406 BS$=CHR$(157):REM BACKSPACE
407 FS$=CHR$(29):REM FORWARD SPACE
408 REM
409 REM **DISPLAY MASK**
410 FOR I= 1 TO LI
415 T$=MID$(M$,I,1)
420 IF T$="9" OR T$="X" THEN PRINT ".";:GOTO 430
425 PRINT T$;
430 NEXT I
435 FOR I=1 TO LI:PRINT BS$;:NEXT
436 REM
437 REM
440 I=1 :REM SET POINTER
442 C$=MID$(M$,I,1)
445 GET T$:IF T$="" THEN 445
446 IF T$=CHR$(13) THEN RETURN
447 IF T$=CHR$(20) THEN 600
448 IF I>LI THEN 445
449 REM
450 IF C$="X" THEN 465
454 REM **ACCEPT A NUMBER**
455 IF T$<"0" OR T$>"9" THEN445
460 GOTO 470
461 REM
462 REM
464 REM **ACCEPT A LETTER88
465 IF T$<"A" OR T$>"Z" THEN 445
466 REM
467 REM
470 PRINT T$; :REM PRINT CHAR.
475 TE$=TE$+T$
480 I=I+1 :REM INCREMENT POINTER
490 IF I>LI THEN 445
497 REM
498 REM
```

```
499 REM **CHECK NEXT MASK POSITION**
500 C$=MID$(M$,I,1)
510 IF C$="9" OR C$="X" THEN 442
520 PRINT FS$; :REM SPACE PAST A PUNCTUATION
530 TE$=TE$+C$
540 I=I+1
550 GOTO 442
580 REM
590 REM
595 REM **DELETE**
600 I=I-1 :REM DECREMENT POINTER
604 REM CHECK POSITION ON LEFT
605 C$=MID$(M$,I,1)
606 TE$=LEFT$(TE$,LEN(TE$)-1)
610 IF C$="9" OR C$="X" THEN630
615 PRINT BS$;:I=I-1:GOTO 605 :REM JUMP PUNCTUATION
630 PRINT BS$"."BS$;
650 GOTO 445
```

positioning the cursor for output

Commodore BASIC lacks the facility to position the cursor to a specific screen location prior to a PRINT. This hampers the development of many programs. Fortunately, there is a solution to this problem. The Kernal contains a callable routine which can place the cursor at any XY position on the screen, where X is a horizontal position from 0 to 39 and Y is a vertical position from 0 to 24. This Kernal routine, called Plot, is somewhat difficult to use, requiring a machine language routine to access it. We must use a BASIC subroutine to POKE the machine language routine into an area of memory unused by BASIC (the tape I/O buffer starting at location 828 is used). Then the machine language routine must be executed with a SYS command. Finally, after the machine language routine has executed, we must RETURN from the BASIC subroutine.

```
1000 L0=828
1005 POKE L0, 162
1010 POKE L0+1, Y
1015 POKE L0+2, 160
1020 POKE L0+3, X
1025 POKE L0+4, 24
1030 POKE L0+5, 32
1035 POKE L0+6, 240
1040 POKE L0+7, 255
1045 POKE L0+8, 96
1050 SYS L0
1055 RETURN
```

For example, to start printing at position X=0 Y=4, the code to include in a program is

```
100 X=0:Y=4:GOSUB 1000
110 PRINT "THIS IS AT 0,4"
```

input screens

Input screens display a screen full of prompts for the user and then start accepting data at the first prompt. This gives the user a sense of entering a full page of information at a time. Most systems require a lot of data entry to use them. In writing screens, we will use both the mask and the direct cursor positioning routines which have already been developed in this chapter. We will use five arrays to keep track of the prompts, the masks, the X and Y positions, and the data entered. We will cycle through the arrays displaying the prompts and the masks at their proper positions. Then the mask routine will be called at each prompt in the screen to accept the data. The data that is entered was stored in an array, IN$. There is a problem in that we first want to display the masks after the prompts, but the mask routine accepts input right after displaying the mask. The solution is to display all the masks, without using the mask routine, by using simple PRINT statements. Then, when the mask routine is called, we will position the cursor at the beginning of the mask. When the mask is redisplayed by the routine, it will be superimposed on the one already on the screen. The mask routine has to be altered slightly to use the masks from an array.

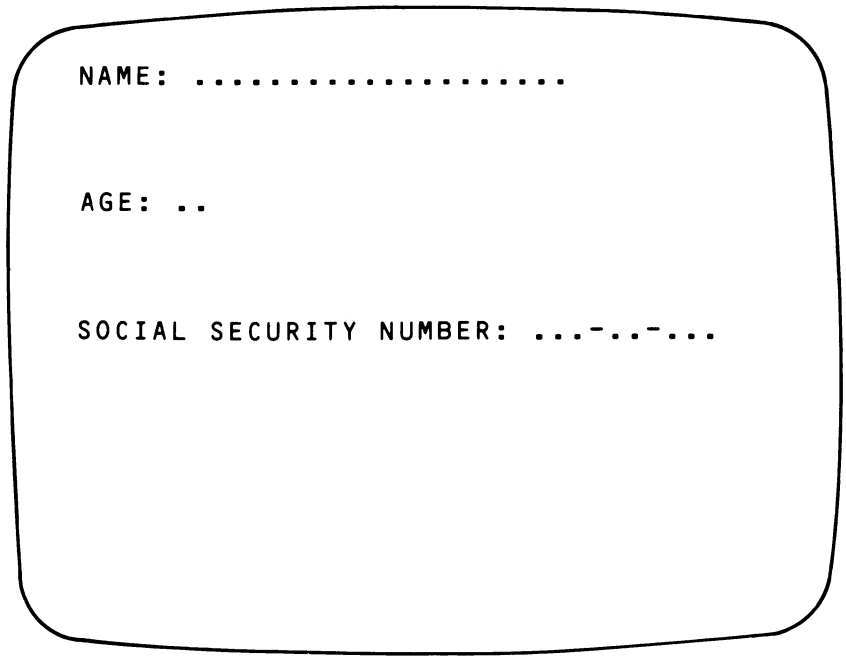
variables used

arrays	
X & Y	holds the position for the prompts
PR$	holds the prompts
M$	holds the masks
IN$	holds the input accepted

```
90 PRINT CHR$(147)
95 REM DISPLAY PROMPTS AND MASKS
100 FOR I = 1 TO 3
110 X=X(I):Y=Y(I):GOSUB 1000:REM CALL POSITION ROUTINE
120 PRINT PR$(I);M$(I);
130 NEXT I
135 REM GET INPUT
140 FOR I= 1 TO 3
150 X=X(I)+LEN(PR$(I)):Y=Y(I):GOSUB 1000: REM POSITION
CURSOR ON MASK
160 GOSUB 900 : REM CALL MASK ROUTINE
170 IF TE$="X" THEN 700 : REM NO MORE INPUT
180 IN$(I)=TE$
```

```
190 NEXT I
200 REM PROCESS DATA
.
.
  GOTO 100
```

Below is the sample screen:



screen memory and ascii

The Commodore 64 screen is composed of a grid of 25 rows (numbered 0 through 24) and 40 columns (numbered 0 through 39). This yields 1000 possible locations on the screen. Each location on the screen has a controlling memory address associated with it. The memory addresses 1024 to 2023 are reserved for this purpose. What is displayed on the screen is dependent on the contents of each memory location. Each possible value from 0 to 255 in that location displays a different character on the screen. The screen display codes are listed in Appendix F. Here is a brief summary.

Letters A..Z	Codes 1..26
Numbers 0..9	" 48..57
Reverse letters A..Z	" 129..154
Reverse numbers 0..9	" 176..185

A character can be displayed on the screen by POKEing its code into the proper screen memory position. For example, to place an A in the upper left-hand corner of the screen, POKE 1024,1; for a reverse A, POKE 1024,129. The memory location of any position on the screen can be determined by the formula Loc=1024+X+Y*40. A character on the screen can easily be changed to reverse video by PEEKing the proper location, adding 128 to the value, and POKEing it back into the location. We will later use this technique to simulate a cursor. Unfortunately, the codes used as screen codes do not coincide with the codes used to represent characters everywhere else in the computer, such as in a string variable or in a file. The other code used is known as ASCII (American Standard Code for Information Interchange). The ASCII codes are listed in Appendix E of this book.

screen reading input

None of the input routines we have explored allows the use of the left and right cursor keys to edit the information being input. A routine that has this facility is more complex and challenging to write than the previous examples. When the cursor is moved, it must display the character that is underneath it. *This was simple when there was no character under the cursor,* but if the cursor is moved over a character, we must find the character underneath the cursor. A second problem involves handling the changes made to the string being built. Using the string functions for this would be very clumsy. The solution hinges on changing our approach to one more like that used by the Kernal for input. This new routine will display the characters on the screen and allow for the use of the cursor keys, but will not build the string until the carriage return is entered. Then the routine will build the input string by scanning the characters on the screen and concatenating them into one string. Here is the psuedocode for the routine. This routine can be found in the available diskette in a file named "SCREENREADER."

A. Accept input (GET) with blinking cursor
   If it is a left arrow GOTO B
   If it is a right arrow GOTO C
   If it is a carriage return GOTO D
   Check the character's validity
   If valid:
      Print on screen
      Increment position counter for next character
      GOTO A

B. Print character under cursor (just in case it is in reverse mode)
   Decrement position counter
   Print cursor left character (CHR$(157))
   GOTO A

C. Print character under cursor
   Increment position counter
   Print cursor right character (CHR$(18))
   GOTO A

D. FOR I= first position TO last position
Convert screen code to ASCII code
Add character to string TE$
NEXT I

cursor routine

- PEEK screen memory for character at cursor position
- Print it by a POKE of either regular or reverse screen code
- RETURN

notes

We will check the character underneath the cursor by PEEKing into the screen memory location for the cursor's position. After finding the screen code, we will have to deal with the translation between the screen codes and the ASCII codes.

1. The translation between ASCII and screen code for character set one is
   Letters : Screen code + 64 = ASCII
   Numbers : Screen code = ASCII
   Symbols : Screen code = ASCII
2. Regular video and reverse video codes differ by 128. To test if a code is reverse video and to convert it to regular, we could use IF P>128 THEN P=P−128. Or we could use the AND operator. Realizing that in binary any number greater than 128 has bit number 8 set to one, we can subtract 128 by masking out the eighth bit by ANDing with 127 (binary 01111111). That is P=P AND 127.
3. The screen code for a space is 32. However, the code for a reverse space 160 also displays a space. We can handle this by testing the code. If it is a space, we can reverse it by using the method previously used.
4. This routine deals with character set one, which contains only the upper case characters. Character set two, which contains both upper and lower case letters, is trickier to use. The lower case codes are the same as the upper case codes in set one. The upper case codes are the same as the ASCII codes.

variables used

CT	Counter for cursor
FL	Mode of cursor (reverse or normal)
LI	Length of input (limit)
T$	Character entered
SP	Screen memory location of first position
PO	Current screen position (from 0)
X,Y	Screen coordinates

TE$ Input string being built
P Used for value of screen memory positions

To use this subroutine, set the X and Y coordinates of the first input position. Then set the maximum length in LI and call the routine. The input is returned in TE$. The returned string is padded on the right with blanks up to the declared length of the input. The extra blanks can be removed with

```
FOR I=1 TO LI
IF RIGHT$(TE$)=" " THEN TE$=LEFT$(TE$,LEN(TE$)-1)
NEXT I
```

tricks used

The routine followed the pseudocode exactly. However, a trick was used to replace the character under the cursor with the same character in normal video mode. This was done just in case the cursor was in reverse video when it was moved. The trick takes advantage of the fact that the cursor routine will display the character in normal video when FL is equal to 1. On lines 600, 630, and 650 the cursor routine was called with FL fixed at 1.

```
497 REM ************************
498 REM * SCREEN READING INPUT *
499 REM ************************
500 CT=1:FL=1:P0=0:LI=10:PRINTCHR$(147);
501 GET T$:GOSUB 700: IF T$="" THEN 501
505 IF T$=CHR$(29) THEN 600
510 IF T$=CHR$(157) THEN 630
515 IF T$=CHR$(13) THEN 650
520 IF T$<CHR$(32) OR T$>CHR$(90) THEN 501
525 REM ACCEPT CHARACTER
530 P0=P0=1:ZZ=P
535 IF P0>LI THEN 501
540 PRINT T$;
545 GOTO 501
599 REM ** RIGHT ARROW **
600 CT=9:FL=1:GOSUB 700:REM REPLACE CHARACTER ON SCREEN
608 P0=P0+1
610 PRINT CHR$(29);
620 GOTO 501
629 REM ** LEFT ARROW **"
630 CT=9:FL=1:GOSUB 700:REM REPLACE CHARACTER ON SCREEN
635 P0=P0-1:PRINT CHR$(157);
640 GOTO 501
645 RETURN
649 REM ** READ SCREEN **
650 CT=9:FL=1:GOSUB 700:REM REPLACE CHARACTER ON SCREEN
```

```
655 FOR I=1024 TO 1024+LI
660 P=PEEK(I):IF P<27 THEN P=P+64
665 TE$=TE$+CHR$(P)
670 NEXT
675 RETURN
700 SP=1024+PO:P=PEEK(SP)
705 CT=CT+1:P=P AND 127
710 IF CT<>10 THEN RETURN
715 IF FL=1 THEN POKE SP,P:GOTO 730
720 POKE SP,P+128
725 IF P+128=160 THEN PRINTCHR$(18)" "CHR$(146)CHR$(157);
730 FL=FL*-1:CT=0
735 RETURN
```

simons' BASIC extensions

Simons' BASIC adds several new I/0 statements that can be used as an adjunct to the techniques described in this chapter. These new instructions can be used in lieu of some of the techniques demonstrated here. However, as has been mentioned, Simons' BASIC takes up almost 8K of memory.

PRINT AT(column,row)

Simons' BASIC includes the useful PRINT AT statement. PRINT AT performs the same function as our cursor positioning routine, placing the cursor at a specific screen location before printing. It is important to note that here the position is given as column then row; the opposite of the routine we developed. Either a string or numeric variable can be printed. The advantage that our positioning routine has over PRINT AT is that we can use our routine to position the cursor for input as well as output.

FETCH "control character",length,string variable

FETCH operates similarly to the masked input routine developed earlier in this chapter. With FETCH you describe the length of the input, the variable to receive the data, and a control character which controls which characters are to be accepted. The control characters used are:

<CLR/HOME>	Unshifted alphabetic characters only
<CRSR DOWN>	Numeric only
<CRSR RIGHT>	All characters

The control character must be in quotes, or the CHR$ function will not work.

```
100 FETCH "<CRSR DOWN>",8,A$
```

This FETCH accepts up to 8 digits for the string A$. The masked input routine is far superior to FETCH. Masked input allows more control over the acceptable input, displays a mask, and uses a blinking cursor (FETCH doesn't).

USE "##text##",string variable

The USE command prints numbers formatted by a mask. The mask is defined with number signs(#) in the positions for digits. Any other characters can be interspersed.

```
10 A$="1122.34"
20 USE "$#,###.##",A$
```

The example prints $1,122.34

USE will not round off a fractional value, works only with a string variable (use the STR$ function on a numeric variable), and does not produce a carriage return. Other than that, it's a useful tool for output.

3

files and the disk drive

No doubt you are already familiar with using the 1541 disk drive for storing BASIC programs. The disk drive also serves another important purpose. Most substantial applications require a disk drive to store information for future use in a data file on a diskette. Data files open up a whole new world of applications, such as mailing lists, word processors, electronic spreadsheets, and budgeting programs.

the disk system

The advantages of using a disk drive rather than a cassette have been spelled out many times before, but it's worth summarizing here:

Ease of use
Speed
Variety of file types
Speed
Accuaracy
Speed
Availability of non-sequential files.

In general, a disk drive is a virtual requirement for serious programming or for the storage of large amounts of data.

physical overview

Together, the Commodore 64, the VIC-1541 disk drive, and the serial interface cable make up the disk system. The disk drive reads and writes information to and from a 5 1/4" mini-floppy diskette in bytes (8 binary digits). One byte is enough storage space to store one character as its ASCII code. During the NEW command the diskette surface is divided up into 684 blocks (areas) that will each hold 256 bytes. These blocks are laid out on 35 concentric rings called tracks. Each track contains 17 to 21 sectors (blocks), with the tracks closest to the center having fewer sectors. This adds up to a total storage space of 174,848 bytes per diskette. Data is accessed from the diskette a block at a time, with all 256 bytes read or written at once.

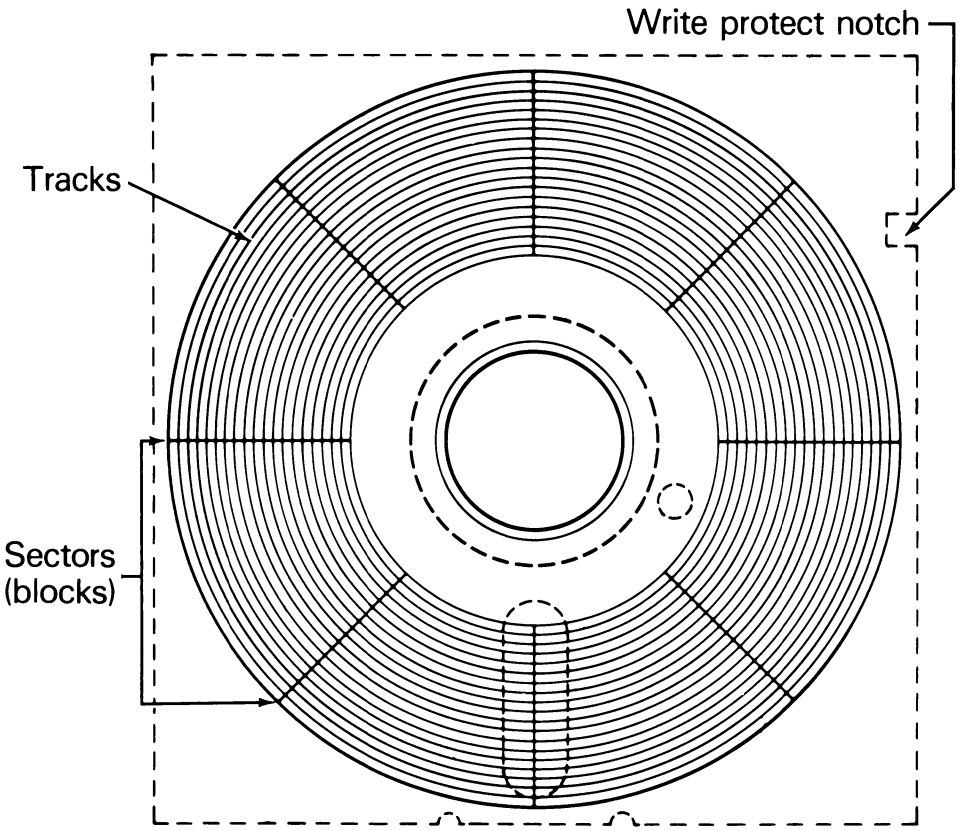


Fig. 3.1

The disk drive is operated by its own 6502 microprocessor (the same one used in the Atari and Apple II computers), and is controlled by a Disk Operating Sys-

tem (DOS) that resides in 8K of Read Only Memory (ROM) in the disk drive. This is one of the unique features of the Commodore 64, and it alleviates the problem of having the DOS stored on every diskette and taking up room in the computer's Random Access Memory (as is the case in the Apple II). The disk drive also contains its own 2K of RAM memory, which can contain a machine language program to be executed by the drive's microprocessor. This is quite a tricky technique, and I have yet to see it done anywhere. A perfect application for this technique would be in a database manager where the drive could be programmed to search a file for certain data, thus eliminating the time needed to transfer data between drive and computer.

The disk drive even sports its own power supply (which is why it is so bulky), so it is important never to cover the louvers on its top. Data is transferred back and forth between the drive and the computer by means of a serial cable. This allows only one bit at a time to be transferred through the cable and is much slower than the byte at a time parallel cables found in many computer systems.

files—an overview

the directory

DOS maintains on Track 18 a list of all the files held on the diskette. This Directory has space for 144 entries. Thus there is a limit of 144 files per diskette. Each Directory entry contains the file's name, its type, its size in blocks, and an indication of where on the diskette the file starts.

the BAM

Just as important as the Directory is the Block Availability Map or BAM. The BAM keeps track of which of the 664 blocks are already holding data. When a file is stored, DOS searches BAM for available blocks in which to place the file, placing the file in the free blocks and marking the BAM accordingly. The structure of BAM is essentially one bit acting as a flag for each block on the diskette. The bit corresponding to a block is set to 1 if the block is occupied or reset to 0 if it is available. Before a file can be saved and in order to prevent files already on the diskette from being written over, the BAM of the diskette in the drive must be in the drive's memory. If the diskette in the drive has been changed, the new BAM must be read in. Theoretically, the drive will detect when a new diskette is being used by checking the two byte disk ID. However, it is always safer to do an INITIALIZE command when a new diskette has been placed in the disk drive.

The Initialize Command:
PRINT#15,"I"

channels

A channel is a pathway for sending information between the disk drive and the Commodore 64. There are 15 channels.

Data Channels

The channels numbered 2 through 14 are used to send data between the disk drive and the Commodore 64.

The Error Channel

The error channel (channel 15), also referred to as the command channel by the disk drive manual, is the pathway between the disk drive and the Commodore 64 for disk instructions and error messages. After certain disk operations (OPEN, position commands) it is necessary to check the error channel for possible error messages. If this is not done, the system may possibly hang.

device numbers

Each system device has a pre-assigned device number, used to refer to that device in a BASIC program.

Disk	- 8
Printer	- 4
Screen	- 3
Keyboard	- 1
RS232 Port	- 2

how data is stored

Character data is stored in a file as the ASCII code of each character. Therefore, the string "GOOD MORNING" would occupy 12 bytes in a file. Numeric values are stored as if the STR$ function has been performed. One byte is used per digit, plus a leftmost byte for the sign (blank or a minus) and a rightmost byte containing the cursor right character (ASCII code 29) added as a separator. Therefore 123 and -123 both occupy 5 bytes.

records and fields

Records and fields are the subdivisions of a file. Files contain one or more records, records contain one or more fields. A file that contained ten names would have ten records, each record containing one field—the name. A file that contained ten names and ages, would still have ten records, but there are two fields in each record—name and age.

file number

A disk file is referred to by both a file name and a file number. The file name which can be from 1 to 16 characters is how the file is listed in the Directory. The file number can be from 1 to 127 and is used to reference the file in several instructions. The file name and number are equated in the OPEN statement.

file types

Before we can write a program using files, it is important to discuss the different types of files available and the commands that are needed to use them. The Commodore 64 supports three types of data files known as sequential, relative, and random. Only the most common and important types, sequential and relative, will be discussed and each has a chapter devoted to it.

In a sequential file, each set of data (known as a record) is stored immediately after the preceding set. The data is usually separated by a comma, a semicolon or a return character (ASCII code 13). These separators must be placed in the file by the BASIC program.

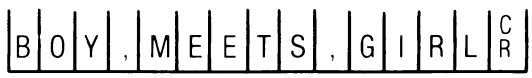


Fig. 3.2

A sequential file must be read into memory starting at the first character (byte). It must also be stored on the diskette starting at the first byte. Therefore it is not possible to replace only a section of the file. Adding data to the end of a file can not be done, except by reading the entire file into memory, appending the data to the file, and then writing it back to the disk.

In relative files, records are stored in areas of predeclared length, called the record length.

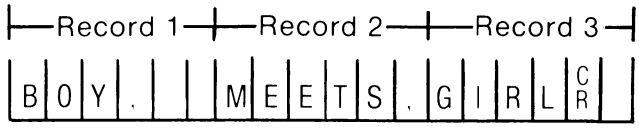


Fig. 3.3

In a relative file any particular byte can be selected, to read or to write to. This direct access to any byte and the predeclared record length allow any section of the file to be changed. Appending to the file (adding data to the end) is also possible.

Here is a comparison of the file types

	RELATIVE	SEQUENTIAL
Access time	slower	faster
Flexibility	more flexible	less flexible
Record size	fixed	not fixed
Use	harder to program	easier to program

the disk commands

the OPEN statement

The OPEN statement is used to allow the computer access to a disk file. When a file has been opened, the red light on the disk drive will go on.

the CLOSE statement

The CLOSE statement is used to terminate a program's access to a file. It is also the signal to the DOS to complete the bookkeeping for the file. When all files are closed the red light on the drive should go out.

PRINT#

The PRINT# statement is used to write data to a disk file. The file number follows the number sign.

INPUT#

The INPUT# statement is used to read a string of up to 80 characters from a file. The file number follows the number sign.

GET#

The GET# statement is used to read a single character from a disk file. The file number follows the number sign.

the status variable ST

ST is a special variable updated by the Kernal with a code reflecting the result of an I/O operation on an OPEN file. It is useful in detecting whether or not any data in a file remains to be read. A value cannot be assigned to the variable ST by a BASIC program.

In the next two chapters, sequential and relative files will be explained in detail along with programming examples you can try.

4 using sequential files

Sequential data files on disk work exactly like they do on cassette tape, only more quickly and accurately. The major drawback of a sequential file is that it must be read or written in its entirety. This means that all processing done with the data from a file must be done with the whole file in memory. For instance, to append data to a file, the whole file must be read into memory, then written back to the disk with the new data added.

The use of a sequential file from a BASIC program is quite simple. Before a sequential file can be used, it must first be OPENed. This creates the file and an entry in the directory if the file doesn't exist or allows access to a file that already exists. Data is written to a file with the PRINT# statement and read with either the INPUT# or GET# statements. After the operations to the file are complete, access to the file must be terminated with a CLOSE statement.

opening a sequential file

The OPEN statement for a sequential file is:

```
OPEN file#,device#,channel#,"0:filename,type,direction"
```

- The device number for the disk drive is 8.
- The file number and the channel number should be the same to avoid confusion. The number can range from 1 to 14.
- The filename is created by the rule you are familiar with, containing from 1 to 16 characters. A string variable can also be used.
- The type is S or SEQ for sequential.
- The direction is R for read
  W for write

The first time a file is OPENed, it must be created by OPENing it for writing. This creates the directory entry for the file. Up to 5 sequential files can be open at once.

Examples

Opening a File for reading

```
OPEN 2,8,2,"Ø:SAMPLE,S,R"
```

or

```
OPEN 2,8,2,"Ø:"+F$+"S,R"
```
where f$ contains the file name.

Opening a file for writing

```
OPEN 2,8,2,"Ø:SAMPLE,S,W"
```

If the file is to replace one that already exists on the disk:

```
OPEN 2,8,2,"@Ø:SAMPLE,S,W"
```

After each OPEN command, the error channel (15) must be read. The channel provides information about the OPEN operation. An error message, error description, and track and sector number are available to be inspected. Normally only the first two are examined. The error channel must be OPENed before any other file.

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

Subsequently, the error channel can be read with

```
INPUT#15,EC,ED$
```

The possible DOS errors that can occur after an OPEN operation are

0 - No problem	60 - File open for write
62 - File not found on disk	63 - File already exists
70 - No channel	64 - File type mismatch
26 - Disk write protected	33 - Invalid file name
34 - No file given	

These error conditions can be checked by a program and appropriate action can then be taken. For example, if error 26 (write protect) occurs, you can prompt the user to check the disk.

closing a sequential file

When a file is no longer needed, or immediately before a program is about to finish, it is MANDATORY that OPENed sequential files be CLOSED. This is necessary for the DOS to complete its bookkeeping for the file.

The form is simple

```
CLOSE file#
```

For the file used in the OPEN example

```
CLOSE 2
```

writing data to a sequential file

When data is written to a sequential file, it is placed directly after the last data that was written—each field sequentially following the last, hence the clever name. When dealing with sequential files it should be noted that the amount of space taken up by a specific record is dependent on the length of the data. This is different from relative files where each record occupies the same amount of storage. Data is written to an open file with the PRINT# statement

```
PRINT#file#,data
```

The following program segment

```
110 A$="GOOD"
120 B$="NIGHT"
130 C$="LADIES"
140 PRINT#2, A$;",";B$;",";C$
```

when run results in a file with the data arranged as follows:

Data written to a sequential file must be separated from the data following it with either a comma, semicolon, or carriage return. This is necessary in order to facilitate reading back the data at a future time with the INPUT# statement. Note that the strings written to the file are separated by a comma in a string ",". This is done because we want the comma actually placed as a character in the file. A carriage return is automatically placed in the file provided by the PRINT statement after the last variable in the list (C$). No other separator is required. Numeric data

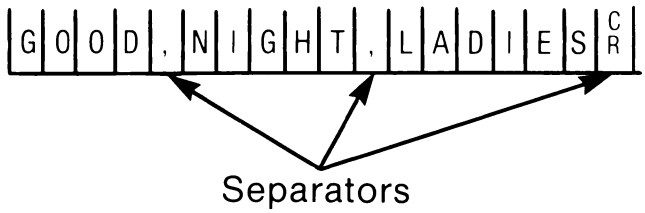


Fig. 4.1

written to a file needs no separator in the file, although a comma must be used in the PRINT# statement to separate the variables.

records and fields in sequential files

Unlike what we will see in relative files, there are no fixed record or field lengths in a sequential file. This makes the concept of records and fields more of a logical than a physical one. Normally, the fields of a record are written (with separators) one after another, with the records written in succession.

reading data from a sequential file

Data is read from a sequential file with either the INPUT# or the GET# commands.

INPUT#file#,variable list

INPUT#, will retrieve from a file that is opened for reading a string of characters (a single field) of up to 80 characters, stopping at the first separator (comma, semicolon, or carriage return) found. The separator is not included as part of the string read. The data written to the file SAMPLE in the previous example can be retrieved with

```
INPUT#2,E$,F$,G$
```

Numeric data placed in a file can be retrieved the same way.

GET#file#,string variable

GET#, will retrieve a single character from a file opened for reading. The field separators have no meaning to GET#, they are treated like any other charac-

ter in the file. GET# is more complicated to use than INPUT# and will be explained later in more detail.

Note: At this time you should test your understanding of these two points.

1. It didn't matter that different variables were used in the PRINT# and INPUT# commands to refer to the same data in the file, SAMPLE. Data in a file is independent of any variable names in a program. This allows for data to be read by a program different from the one that wrote it to the file,
2. The commas used as data separators serve a function distinct from commas used to separate different variables in a PRINT statement.

programming techniques

reading to the end of a file (eof)

In many applications, the number of records in a file is not exactly known. A program will read the contents of a file until no more data is there. this will eventually lead to an error when an attempt is made to read data that is not there. Some mechanism is needed to signal the program when no more data is available. The status variable ST can be used for this purpose. ST is a special system variable whose value is constantly updated by the Kernal to show the result of the latest I/O operation. ST is set to 64 (a coincidence) when the last piece of data has been read from a file. In the following example, data from a file is being read into an array and this operation will stop after the last piece of information is read.

```
100 INPUT#2,D$(I) :REM ST IS SET
110 I=I+1
120 IF ST <> 64 THEN 100
130 REM FILE HAS BEEN READ TIME TO PROCESS DATA
```

GET#

The major drawback of using INPUT#, is that it will read from a file a string no longer than 80 characters. If your application requires reading a longer string, GET# must be used. As described, GET# reads a single character from a file. In using GET#, we will have to read the file character by character and concatenate the characters in a field to form a string. In this process we must check for the field separators ourselves.

```
100 TE$="" :REM CLEAR STRING
110 GET#2,T$
120 IF T$<>"," THEN TE$=TE$+T$: GOTO 110
```

This code assumes the field separator in the file is a comma. There is also a limit to the length of a string read in this fashion. Remember that Commodore BASIC

only allows a string to have a maximum length of 255 characters. In general, this routine should not be used to replace INPUT#, since it will execute more slowly.

GET# is also useful for inspecting a file. This can be very handy in debugging a program since it allows inspection of the field separators. The following routine will dump to the screen the contents of any sequential file already on the diskette. If you have the available diskette, you can find this program in a file named SEQREAD .

```
2 REM ******************************
3 REM * READ ANY SEQUENTIAL FILE   *
4 REM ******************************
5 INPUT "ENTER FILE NAME";F$
10 INPUT "ENTER NUMBER OF BYTES TO READ";B
20 OPEN 15,8,15
30 OPEN 2,8,2,"0:"+F$+",S,R"
40 INPUT#15,EC,ED$
60 FOR I= 1TO B
70 GET #2,A$: PRINT A$
80 NEXT
90 CLOSE 2: CLOSE 15
```

checking if a file exists

Certain applications, like the one we are about to see, require a program to open a file if it exists, or create one if it doesn't. This is important because we don't want to attempt to read from a file with no data in it. This turns out to be trickier than it sounds. The key here is the error channel, which after the OPEN statement will tell us if a file exists or not.

```
10 OPEN 15,8,15
20 OPEN 2,8,2,"0:PHONEBOOK,S,R"
30 INPUT#15,EC,ED$: REM CHECK ERROR CHANNEL
40 IF EC<>62 THEN 90
45 REM FILE DOES NOT EXIST
50 CLOSE 2 :REM CLOSE FILE NUMBER
60 OPEN 2,8,2,"0:PHONEBOOK,S,W" :REM CREATE FILE
70 CLOSE 2
80 GOTO 200
90 REM HERE WE READ IN THE DATA IF THE FILE EXISTED
   .
   .
200 REM
```

example: the phonebook program

This program uses a sequential file to store names and phone numbers. It first reads the file into the arrays N$ and P$(see line 120-180), then allows the user

either to search for a phone number or add a name and number to the list. To end the program the user must use the exit option, which writes the updated file back to the disk. To insure that the user doesn't interrupt the program with the RUN/STOP key, we can disable it with POKE 808,254.

The techniques used in this program resemble those of most programs that use sequential files. All the data from the file is read initially into arrays in memory. Any searching or updating of the information is done in memory. Finally, the data is rewritten to the diskette, replacing the original version of the file. In this example, each record in this program contains two fields, name and phone number. A different array is used for each of the two fields. A single two dimensional array could also have been used for this purpose. In the next chapter, which covers the use of relative files, more sequential file techniques will be developed.

Here is the pseudocode for the program. If you have the available diskette, this program can be found in the file named **SEQPHONEBK**.

```
    Dimension arrays
    Look for file by trying to OPEN it
      If it exists, CLOSE it
      Doesn't exist, OPEN for write, CLOSE, GOTO A
    Read file into arrays
A.  Display main menu
    1. Add a name
       Accept name and phone number
       GOTO A
    2. Search
       Accept name
       Search name array
         if it exists, print it, GOTO A
         Doesn't exist, print message, GOTO A
    3. Exit
       OPEN file with replace option
       Write data into file
       CLOSE all files
```

variables used

N$	array that holds the names
PH$	array that holds the phone numbers
I	Next available position in the arrays
S$	Name to search for
K,J	Scratch variables

```
3 REM ************************************
4 REM * PHONE BOOK WITH SEQUENTIAL FILES *
5 REM ************************************
Ø DIM N$(1ØØ),PH$(1ØØ)
```

S/B 6

S/B 15

```
5 POKE 808,254:REM DISABLE STOP
20 I=1
30 OPEN 15,8,15:REM OPEN COMMAND CHANNEL
98 REM
99 REM
100 PRINT CHR$(147)
110 PRINT "LOADING PHONE BOOK"
120 OPEN 2,8,2,"0:PHONEBOOK,S,R"
130 GOSUB 1000
140 IF EC<>62 THEN 160
145 REM NO FILE
150 CLOSE 2
155 OPEN 2,8,2,"0:PHONEBOOK,S,W"
158 CLOSE 2
159 GOTO 200
160 INPUT#2,N$(I),PH$(I) :REM READ FILE
170 I=I+1
180 IF ST<>64 THEN 160 :REM CHECK STATUS
190 CLOSE 2
194 REM
195 REM *MAIN MENU*
200 PRINT CHR$(147) :REM CLEAR SCREEN
210 PRINT "1)ADD A NAME"
220 PRINT "2)SEARCH FOR A NAME"
230 PRINT "3)EXIT"
235 PRINT "ENTER CHOICE";
240 INPUT C$
250 IF C$<"1" OR C$>"3" THEN 200
260 ON VAL(C$) GOTO 270,300,500
264 REM
265 REM *INPUT SCREEN*
270 INPUT "ENTER NAME";N$(I)
280 INPUT "ENTER PHONE NUMBER";PH$(I)
285 I=I+1
290 GOTO 200
300 INPUT "ENTER NAME TO SEARCH FOR";S$
305 REM SEARCH LOOP
310 FOR K=1 TO I-1
320 IF N$(K)=S$ THEN 370
330 NEXT K
340 PRINT "NAME NOT FOUND"
350 FOR J=1 TO 300:NEXT
360 GOTO 200
368 REM
369 REM *NAME FOUND IN LIST*
370 PRINT "PHONE NUMBER IS ";PH$(K)
380 PRINT"<HIT ANY KEY TO CONTINUE>"
```

```
390 GET T$:IF T$="" THEN 390
400 GOTO 200
498 REM
499 REM *WRITE FILE TO DISK*
500 OPEN 2,8,2,"@0:PHONEBOOK,S,W"
505 GOSUB 1000
510 FOR K= 1 TO I-1
520 PRINT#2,N$(K)",",PH$(K)
530 NEXT K
540 CLOSE 2:CLOSE 15
550 END
999 REM *ROUTINE TO READ ERROR CHANNEL*
1000 INPUT#15,EC,ED$
1010 RETURN
READY.
```

5 using relative files

Relative files are not a way of keeping track of aunts and uncles. They are one of the Commodore 64's major file types. Sequential files, though easy to access, are somewhat limited. Their major drawback is that changes cannot be made to a file without reading the entire file in memory, changing the data in memory, and then replacing the entire file on the diskette. Relative files provide a greater degree of flexibility for the programmer. In a relative file you can change individual records or fields without reading the entire file into memory. Once a file is OPEN, data can be read, written, or added to the end of file. Each of the records in a relative file has a unique record number starting at one. A position command is used to locate particular data in a file before reading or writing. In computers, as in life, you have to give up something to get something. The drawbacks to relative files are that they operate slowly and diskette space is wasted.

when to use a relative file

Typically, a relative file would be used when:

Files larger than the available memory space. If your file is larger than the available memory space, your program could read and process the file in sections rather than as a whole. This is easy to implement with relative files.

When it is not convenient to read the entire file to change the contents of a single record. Since data in relative files can be easily changed or updated, it pays to use them in applications where it is not convenient to read or write an entire sequential file.

as a complement to sequential files

Relative files work best when used in conjunction with a sequential file serving as an index file as part of the Indexed Sequential Access Method (ISAM system). This will be demonstrated later.

records, fields, and record lengths

Data in files is organized into records which are subdivided into fields. For example, in the PHONEBOOK program developed in Chapter 4, each record had two fields, name and phone number. The number of records varies as data is added or deleted. In a relative file, the length of each record is fixed and is declared when the file is OPENed the first time. Fixed lengths do not necessarily have to be set for the fields in a record, but doing so eases the programming process. If there is less than the maximum of characters in a record, the extra space is wasted. In other words, if the record length is ten, ten records would occupy 100 bytes even if each record contained only one character. More space is wasted by side sectors in the DOS's bookkeeping method for relative files, but this is almost transparent to the programmer.

In adapting the phonebook program we saw in the last chapter, some reasonable field lengths might be:

Name	20 bytes
Phone#	12 bytes
record length	32 bytes

Depending upon the programming method used, the record length might differ. If each field in the record is less than 80 bytes long, INPUT# can be used to read the data, but space must be left for a separator. If fields are longer than 80 bytes, which necessitates the use of GET#, the separator can be avoided, thus saving the byte for the separator. Whenever possible, however, INPUT# should be used as it executes much faster than GET#.

Phonebook using separators:

Name	20 bytes
separator	1 byte

Phone#	12 bytes
separator	1 byte
record length	34 bytes

maximum number of records

The number of records in a relative file varies with the size of the records. Records can be stored in up to 658 blocks on the diskette. Each block holds only 254 bytes of data when used by a relative file; the remaining two bytes are used as a pointer to the next block in the file. If the record length is 254 bytes, then there can be 658 records. If the record length is 117 bytes, then 1408 records are possible, and so on.

opening a relative file

to create the file

A relative file is created when it is OPENed for the first time.

```
OPEN file#,device#,channel#,"filename,L,"+CHR$(record
length)
```

Where

- The device number for the disk drive is 8
- The file number and the channel number should be the same (between 1 and 14) to avoid confusion
- The filename is created by the usual rule, from 1 to 16 characters
- The record length is the number of bytes in a record.

subsequent uses of the file

Once a relative file is created, it can be OPENed with a simplified command.

```
OPEN file#,device#,channel#,"filename"
```

Notice that the replace option ' is never used with relative files.

closing a relative file

A relative file is closed the same way as a sequential file:

```
CLOSE file#
```

When data is written to a relative file, it is stored in a buffer in the disk drive until a whole block (256 bytes) is available to be placed on the diskette. Should the buffer be partially filled after the last write to a file has been done, that data is

placed on the diskette by the CLOSE. Therefore, it is mandatory that every relative file that has been OPENed be CLOSEd. Besides, it makes that annoying red light on the drive go out.

the position command

The position command is used to point to a specific byte in a specific record before reading or writing data starting at that byte. The command is sent to the disk drive via the error channel. After the position command is sent, the error channel must be read. Note that the error code 50—Record Not Present—will not always be accurate. It sometimes occurs when positioning to a record in a block already containing a record. In these cases the error code should be ignored. The form of the command is

```
PRINT#15,"P"CHR$(Channel#)CHR$(LO)CHR$(HI)CHR$(byte)
```

Where:

Channel# is the channel number assigned to the file in the OPEN statement. Some versions of the *1541 User's Manual* indicate that 96 should be added to the channel number. Both ways seem to work fine.

Low and Hi are used to represent the number of the record we wish to access. Due to the way the command is sent to the disk drive, it has to be broken up into a low and high component where *record#=256 * hi component + lo component*. The components can easily be computed by

```
HI=INT(rec#/256) : LO=rec# - HI*256
```

In a program the following instructions are handy in calculating the high and low components of a record number (RE).

```
HI=Ø:LO=RE
IF LO>255 THEN HI=INT(RE/256):RE-HI*256
```

Note that in the *1541 User's Manual* this is erroneously shown as IF LO>256. This statement would cause the program to bomb when LO is equal to 256. Try it.

Byte number is the byte in the record where reading or writing will commence. The *1541 User's Manual* indicates that this is optional but that does not seem to be the case. Always include it!

Possible DOS Errors

0	No problem.
26	Diskette write protected.
39	Syntax error. There is an error in a command sent through the error channel.

52 File too large. No space left on the diskette.
62 File not on diskette.

writing data to a relative file

Data is written to a relative file with the PRINT# statement in a manner similar to writing to a sequential file. The characteristics of PRINT# are discussed in Chapter 3. A position command must be executed prior to writing data. The following is an example of writing data to the first record of a relative file.

```
10 OPEN 15,8,15 :REM OPEN ERROR CHANNEL
20 OPEN 2,8,2,"PHONEBOOK,L,"+CHR$(34)
     .
     .
100 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
110 INPUT#15,EC,ED$ :READ ERROR CHANNEL
120 PRINT#2,"PAUL GOODMAN"
130 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(22)
140 INPUT#15,EC,ED$ :READ ERROR CHANNEL
150 PRINT#2,"555-1234"
```

Note that the field separators (carriage returns) are provided by the PRINT# statements.

reading data from a relative file

Data is read from a relative file with either the INPUT# or GET# statements. The characteristics of these statements are discussed in Chapter 4. Position command must always be executed first, and the input will start at the byte indicated in that command. We can read in the data we just wrote with this code

```
100 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
110 INPUT#15,EC,ED$
120 INPUT#2,N$
130 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(22)
140 INPUT#15,EC,ED$
150 INPUT#2,PN$
```

GET#, could have been used to retrieve the same information from the file, but it would be a more complicated program.

```
90 N$="":PN$="" :REM INITIALIZE STRINGS
99 REM ** GET NAME **
100 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
```

```
110 INPUT#15,EC,ED$
120 GET#2,T$
130 IF T$=CHR$(13) GOTO 160 :REM CHECK FOR CARRIAGE
RETURN
140 N$=N$+T$
150 GOTO 120
159 REM ** GET PHONE NUMBER **
160 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(22)
170 INPUT#15,EC,ED$
180 GET#2,T$
190 IF T$=CHR$(13) THEN 220 :REM CHECK FOR CARRIAGE
RETURN
200 PN$=PN$+T$
210 GOTO 180
220 program continues here
```

This code executes several times more slowly than INPUT# because of the larger number of instructions executed.

The following utility program reads the contents of any relative file. It is useful in debugging programs that use relative files. If you have the available diskette you can find this program in a file named READREL .

```
4 REM ************************************
5 REM * READ ANY RECORD IN A RELATIVE FILE *
6 REM ************************************
10 OPEN 15,8,15
20 INPUT "FILENAME";F$
30 INPUT "RECORD LENGTH";RL
40 OPEN 2,8,2,F$
50 GOSUB 500
55 INPUT "RECORD NUMBER?";RN
56 IF RN=-1 THEN 140
60 FOR I=1 TO RL
70 PRINT#15,"P"CHR$(2)CHR$(RN)CHR$(0)CHR$(I)
80 GOSUB 500
90 GET#2,T$
100 PRINT "BYTE ";I;" CONTAINS",T$
110 NEXT
120 GOTO 55
140 CLOSE 2:CLOSE 15
150 STOP
499 REM ** READ ERROR CHANNEL **
500 INPUT#15,EC,ED$
510 PRINT EC,ED$
520 RETURN
```

programming techniques

checking if a file exists

This technique is essentially the same as the one used with sequential files except for the different OPEN statements. We try to OPEN the file with the short form of the OPEN statement. The error channel, when read, will tell us if the file exists or not. If it doesn't exist, we OPEN it and branch around any statements that would read data from the file.

```
10 OPEN 15,8,15
20 OPEN 2,8,2,"PHONEBOOK"
30 INPUT#15,EC,ED$ : REM CHECK ERROR CHANNEL
40 IF EC<>62 THEN 80
45 REM FILE DOES NOT EXIST
50 CLOSE 2
60 OPEN 2,8,2,"PHONEBOOK,L,"+CHR$(34)
70 GOTO 200
80 REM HERE WE PROCESS FILE DATA IF APPROPRIATE
  .
  .
200 REM
```

end of file/number of records in a relative file

There is no way to detect the last record in a relative file (the last record read in a file) as there is in a sequential file. The best way to circumvent this problem is to maintain in the file's first record a count of how many records contain data. This means we will be dealing with two types of records in the file: the first record containing the count of data records that follow and the data records themselves. Whenever a record is added to or deleted from the file, the record count must be updated. In the following example, we read into an array the data in a relative file organized as described above.

```
5 DIM A$(720)
10 OPEN 15,8,15
20 OPEN 2,8,2,"EXAMPLE"
30 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
40 INPUT#2,NO :REM READ RECORD COUNT IN FIRST RECORD
50 FOR RE=1 TO NO
60 HI=0:LO=RE
70 IF LO>255 THEN HI=INT(RE/256):LO=RE-HI*256
80 INPUT#2,A$(I)
90 NEXT RE
 .
 .
```

programming example—the phonebook revisited

Here is the program from the previous chapter rewritten to use relative files. The data is not initially read into memory as before; rather, when we search the records in the file they are read from the file, one by one, in sequential order. Any records added are placed at the end of the file, and the record count in the first record is updated. Note if the file didn't previously exist, a zero is placed in the first record when the file is created. This program can be found in the available diskette in a file named RELPHONEBK.

Here is the pseudocode for the program

```
Try to OPEN file
Doesn't exist, create it, GOTO A
If it exists, read record count from first record
A. Display main menu
    1. Look for a name
         Prompt for name
         Read name field from each record
           If match is found
             Read phone number
             Print it
             GOTO A
           No match found
             Print message
             GOTO A
    2. Add a name
         Prompt for name and number
         Write record
         GOTO A
    3. Exit
         Update first record
```

variable usage

RN	record count
H,L	high and low components of record number
SN$	name to search for
N$	name
PN$	phone number
LI	maximum length of keyboard input
TE$	string with keyboard input
CT,FL	used in simulating the cursor
EC	error code
ED$	error description
BS$	ASCII backspace code

```
4 REM *********************************
5 REM * PHONE BOOK WITH RELATIVE FILES*
6 REM *********************************
8 FL=1:BS$=CHR$(157)
10 OPEN 15,8,15
20 OPEN 2,8,2,"RELPHONEBOOK"
30 GOSUB 1100
40 IF EC<>62 THEN 70
45 REM ** CREATE FILE **
50 CLOSE 2
55 OPEN 2,8,2,"RELPHONEBOOK,L,"CHR$(34)
60 RN=0
62 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
63 GOSUB 1100
64 PRINT#2,0
65 GOTO 100
69 REM ** READ FILE HEADER **
70 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
80 INPUT#2,RN
100 PRINT CHR$(147) : REM CLEAR SCREEN
110 PRINT TAB(10)"P H O N E B O O K"
120 PRINT:PRINT "OPTIONS"
130 PRINT:PRINT "1) LOOK FOR A NAME "
140 PRINT:PRINT "2) ADD A NAME"
145 PRINT:PRINT "3) EXIT"
150 PRINT:PRINT "SELECTION";
160 GET T$:IF T$="" THEN 160
170 T=VAL(T5$):IF T<1 OR T>3 THEN 100
180 ON T GOTO 200,300,400
200 PRINT:PRINT:PRINT"ENTER NAME TO LOOK FOR ";
210 LI=20:GOSUB 800:PRINT" "
215 SN$=TE$
220 FOR I= 2 TO RN+1
222 L=I:H=0
225 IF I>255 THEN H=INT (I/256):L=I-H*256
230 PRINT#15,"P"CHR$(2)CHR$(L)CHR$(H)CHR$(1)
235 GOSUB 1100
240 INPUT#2,N$
245 IF N$=SN$ THEN 270
250 NEXT
255 PRINT "NAME NOT FOUND ":PRINT
260 GOTO 120
270 PRINT " THE PHONE NUMBER IS ";
275 PRINT#15,"PCHR$(2)CHR$(L)CHR$(H)CHR$(22)
280 GOSUB 1100
285 INPUT#2,PN$
290 PRINT PN$
```

```
295 GOTO 120
299 REM ** ADD A NAME **
300 PRINT:PRINT"NAME TO ADD ";:LI=20:GOSUB 800:PRINT " "
305 N$=TE$
310 PRINT"PHONE NUMBER TO ENTER";:LI=12:GOSUB800:PRINT" "
315 PN$=TE$
318 RN=RN+1: REM INCREMENT NUMBER OF RECORDS
319 L=RN+1:H=0
320 IF RN>255 THEN H=INT(RN/256):L=RN-H*256
325 PRINT#15,"P"CHR$(2)CHR$(L)CHR$(H)CHR$(1)
330 GOSUB 1100
335 PRINT#2,N$
340 PRINT#15,"P"CHR$(2)CHR$(L)CHR$(H)CHR$(22)
345 GOSUB 1100
350 PRINT#2,PN$
365 GOTO 120
397 REM ****
399 REM ** EXIT **
400 PRINT#15,"P"CHR$(2)CHR$(1)CHR$(0)CHR$(1)
410 GOSUB 1100
420 PRINT#2,RN
430 CLOSE 2:CLOSE 15
440 END
797 REM *****************
798 REM * INPUT ROUTINE *
799 REM *****************
800 TE$=""
805 GET T$: GOSUB 900:IF T$="" THEN 805
810 IF T$=CHR$(13) THEN RETURN
815 IF T$=CHR$(20) THEN 860
820 IF T$<CHR$(32) OR T$>CHR$(95) THEN 805
830 IF LEN(TE$)>LI THEN 805
840 TE$=TE$+T$
850 PRINT T$;
855 GOTO 805
860 IF LEN(TE$)=0 THEN 805
865 TE$=MID$(TE$,1,(LEN(TE$)-1))
870 PRINT " ";BS$;BS$;" ";BS$;
875 GOTO 805
899 REM ** FLASH CURSOR **
900 CT=CT+1
905 IF CT<>9 THEN RETURN
910 FL=FL*-1:CT=0
915 IF FL=1 THEN PRINT" ";BS$;:GOTO 925
920 PRINT CHR$(18)" "CHR$(146)BS$;
925 RETURN
1097 REM **********************
```

```
1098 REM * READ ERROR CHANNEL *
1099 REM **********************
1100 INPUT#15,EC,ED$
1105 RETURN
```

hashing

If you ran the relative file version of PHONEBOOK, then you noticed how long it takes to search for data on the diskette. Clearly, if a file is very large the time spent in a search of this kind could be significant. In some applications one although not the only way of overcoming this problem is *hashing* (also called hash coding or randomizing). Hashing establishes a relationship between the data and the record number of the record where it is stored. This is usually accomplished by a sophisticated algorithm, but in certain situations the relationship can be very natural and quite simple. A good example is a program that tracks the sales of a store and maintains one record in a relative file for each day of the year. There can be a natural relationship between the day of the year (from 1 to 365) and the record number where that day's data will be stored. A simple program, shown below, converts a date in Month-Day format into the day of the year, assuming D$ holds the date.

```
4 REM ************************************
5 REM * CONVERT DATE INTO RECORD NUMBER *
6 REM ************************************
7 REM
9 DIM M(12)
10 GOSUB 200 : REM INITIALIZE
20 PRINT "ENTER DATE IN MM-DD-YY FORMAT":PRINT
30 INPUT "ENTER DATE";DT$
40 MN=VAL(LEFT$(DT$,2))
50 DY=VAL(MID$(DT$,4,2))
60 YR=VAL(RIGHT$(DT$,2))
70 DT=0
80 IF MN=1 THEN 120
90 FOR I=I TO MN-1
100 DT=DT+M(I)
110 NEXT I
120 DT=DT+DY
130 PRINT DT
140 END
199 REM ** INITIALIZATION SECTION **
200 FOR I=I TO 12 :REM READ IN MONTHS
205 READ M(I)
210 NEXT I
215 DATA 31,28,31,30,31,30,31,31,30,31,30,31
220 RETURN
```

Many other applications can lend themselves to this method quite easily. In determining the suitability of hashing to an application, make sure you can create from each set of data a unique record number. Even when unique record numbers can be created, they may not be consecutive. This leaves empty space in the file without data in it.

access methods reviewed

So far two methods of using relative files in a program have been discussed. Each has its drawbacks. The method used in the PHONEBOOK program to search for a name in the file was to read records from the diskette in sequential order. Compare the name in the record to the one being searched for, and if they match, retrieve the phone number. If they don't, read the next record. Essentially, this is a sequential walk through the file. If the file contained many records, this method would prove to be very slow, the speed being limited by the Commodore 64's slow disk access time. The second method we saw was hashing, which is only applicable in certain situations. The PHONEBOOK program would not easily lend itself to hashing. Another way of handling the task would be to read initially into memory as many records as would fit in the remaining available space. Searching could then be done in memory which is much faster than reading the records one at a time from the diskette. This searching method is the same approach we used with sequential files with one exception, the file will never have to be rewritten to the diskette at the termination of the program. Any new records can be directly appended to the file and any record changed can be individually replaced on the diskette. This method is not a bad approach if the file is small, and reading it into memory does not take an appreciable amount of time. Of course, if the program and the file are large, the whole file can't be stored in memory all at once; 64K is a lot of memory but it only goes so far. In that instance, only part of the file could be maintained in memory, requiring parts of the file to be swapped in and out of memory. This is complicated programming and is very slow, so it should be avoided.

isam—indexed sequential access method

There is another method of accessing relative files which combines the best features of both sequential and relative files with only some of the drawbacks. It is named ISAM, which stands for Indexed Sequential Access Method. This scheme uses both a sequential file and a relative file. The data is stored in the relative file. The sequential file is used as an index to the data in the relative file. In this method the field in the relative file on which searching will occur, called the key, is identified. (In the phonebook program, the key was the name field.) As records are added into the relative file, the key field is also placed into the corresponding position in the sequential file. The first name in the relative file is the same as the first name in the sequential file, the second name in the relative file is the same as the second name in the sequential file, and so on. Since the sequential file contains only one field of the relative file, it is obviously smaller and will occupy less

memory space than the entire relative file. Initially, when the program starts, the sequential (index) file is read into a string array and all searches are done by examining the contents of the array elements. When a match is found in the array, the position in which it is found corresponds to the proper record number in the relative file on the diskette. That record can then be positioned and read. The greatest advantage of using an ISAM is that since the keys are held in memory *searches on the key are done quickly. The drawback is the key field exists in both files,* which wastes some diskette space and requires updating two files; but in any application that requires use of relative files and fast data access, ISAM is a good approach to consider.

isam programming techniques

deleting a record

In a large relative file it is impractical to delete a record by shifting the remaining records into the vacated positions. The easiest way to delete a record in an ISAM system is just to mark the key in the index file as deleted. This is done by placing a rarely used symbol, such as the ampersand (&), in that position of the array. At this point the record is only logically deleted; it still physically exists in the relative file. However, when new records are added, the array can be checked for positions holding deleted records and that space can then be reused. This requires careful programming as in any situation where data is duplicated.

number of records/end of file

Since the index file is a sequential file, it is easily read into an array. The number of records in the relative file can then be determined by examining the array. Every position with data stored in it can be counted as one record. Positions marked as deleted in the index file can not be easily excluded from the count. The counting stops when a position containing the null string is encountered.

```
300 RE=0
310 FOR I= 1 TO 720
320 IF IS$(I)="" THEN 350
330 IF IS$(I)<>"&" THEN RE=RE+1: REM COUNT THIS RECORD
340 NEXT I
350 REM COUNTING IS DONE
```

programming example—isam phonebook

Once again I have the pleasure of presenting the PHONEBOOK program in a third form. This time it is an ISAM system. The delete feature described above has been added. First, the index file is read into an array and then all searches are done on the array. Before the program is terminated, the index file is rewritten to

the diskette just in case of an addition or deletion. This could be avoided by using a variable as a flag. The flag is set on (given a value) if a change is made. When it is time to exit the program, the flag is checked. If it is set, then the array has to replace the index file. If it is not set (no changes were made), the file does not have to be replaced, saving time. This version can be found on the available diskette in a file named **ISAMPHONEBK**.

Here is the pseudocode for the program.

```
    OPEN or create sequential (index) file
    If it exists, read it into an array
A.  Display main menu
    1. Add a name
        Find a position in the file
        Prompt for name and number
        Write record to relative file
        GOTO A
    2. Search for a name
        Prompt for name
        Search array for name
        Name not found, print message, GOTO A
        Name found
          Read record from relative file
          Print phone number
          GOTO A
    3. Delete a record
        Prompt for name to delete
        Search array for name
        Name found, place delete character in array, GOTO A
        Not found, print message, GOTO A
    4. Exit
        Replace sequential file(disk) with array
        CLOSE all files
```

variable usage

IS$	index array
LI	length of keyboard input
TE$	holds keyboard input
ST	I/O status variable
N$	name
PN$	phone number
SN$	name to search for
L,H	high and low component of record number
BS$	backspace character
I	scratch variable

```
1 REM ********************************
2 REM * PHONE BOOK PROGRAM WITH ISAM *
3 REM ********************************
4 BS$=CHR$(157):FL=1
5 DIM IS$(720)
6 REM POKE 808,254:REM DISABLE STOP KEY
8 REM
9 REM ** OPEN OR CREATE FILES **
10 OPEN 15,8,15: REM OPEN ERROR CHANNEL
20 OPEN 2,8,2,"0:PHONEINDEX,S,R"
25 GOSUB 1100
30 IF EC<>62 THEN 80
34 REM
35 REM **FILES DO NOT EXIST**
40 CLOSE 2
45 OPEN 2,8,2,"0:PHONEINDEX,S,W"
50 CLOSE 2
55 OPEN 4,8,4,"ISPHONEBOOK,L,"+CHR$(34)
60 GOTO 110
74 REM
75 REM ** IF FILES EXIST **
80 OPEN 4,8,4,"ISPHONEBOOK"
83 REM **READ INDEX FILE**
84 I=I
85 INPUT#2,IS$(I)
90 I=I+1
95 IF ST<>64 THEN 85 :REM CHECK STATUS
100 CLOSE 2 :REM CLOSE INDEX FILE
108 REM ** MAIN MENU **
110 PRINT CHR$(147): REM CLEAR SCREEN
120 PRINT "OPTIONS"
125 PRINT:PRINT"1)ADD A NAME"
130 PRINT:PRINT"2)SEARCH FOR A NAME"
135 PRINT:PRINT"3)DELETE A RECORD"
138 PRINT:PRINT"4)EXIT"
140 PRINT:PRINT"ENTER SELECTION ";
150 GET T$:GOSUB 900:IF T$="" THEN 150
155 PRINT" "
160 T=VAL(T$)
165 IF T<1 OR T>4 THEN 110
170 ON T GOTO 175,275,350,410
172 REM
173 REM ** ADD A NAME **
174 REM **FIND POSITION FOR NEW NAME**
175 FOR I= 1 TO 720
180 IF IS$(I)="" OR IS$(K)="&" THEN 200
185 NEXT I
```

```
187 PRINT "NO SPACE REMAINING TO ADD "
190 GOTO 110
200 PRINT"NAME TO BE ENTERED ";
205 LI=20:GOSUB 800:PRINT" "
210 N$=TE$
215 PRINT"PHONE NUMBER TO BE ENTERED ";
220 LI=12:GOSUB 800:PN$=TE$:PRINT " "
225 IS$(I)=N$
228 REM **WRITE RECORD TO REL. FILE**
230 L=I:H=0
235 IF L>255 THEN H=INT(I/256):L=I-H*256
240 PRINT#15,"P"CHR$(4)CHR$(L)CHR$(H)CHR$(1)
245 GOSUB 1100
250 PRINT#4,N$
255 PRINT315,"P"CHR$(4)CHR$(L)CHR$(H)CHR$(22)
260 GOSUB 1100
265 PRINT#4,PN$
270 GOTO 110
273 REM ** FIND A NAME **
275 PRINT "ENTER NAME TO SEARCH FOR ";
280 LI=20:GOSUB 800:SN$=TE$:PRINT" "
285 FOR I= 1 TO 720
290 IF SN$=IS$(I) THEN 310
295 NEXT I
300 REM **NAME IN INDEX, GET NUMBER FROM DISK**
310 L=I:H=0
312 IF L>255 THEN H=INT(I/256):L=I-H*256
315 PRINT#15,"P"CHR$(4)CHR$(L)CHR$(H)CHR$(22)
318 GOSUB 1100 :REM READ ERROR CHANNEL
319 INPUT#4,PN$ :REM GET PHONE NUMBER
320 PRINT "THE PHONE NUMBER IS ";PN$
325 GOTO 120
348 REM
349 REM ** DELETE A NAME **
350 PRINT "ENTER NAME TO DELETE ";
355 LI=20:GOSUB 800:SN$-TE$:PRINT " "
360 FOR I= 1 TO 720
365 IF IS$(I)=SN$ THEN 390
370 NEXT I
375 PRINT "NAME IS NOT HERE TO DELETE"
380 GOTO 120
390 IS $(I)="&"
400 GOTO 120
408 REM
409 REM ** EXIT ROUTINE **
410 CLOSE 4 :REM CLOSE REL. FILE
415 OPEN 2,8,2,"@0:PHONEINDEX,S,W"
```

```
420 FOR I= 1 TO 720
425 IF IS$(I)="" THEN 440
430 PRINT#2, IS$(I)
435 NEXT I
440 CLOSE 2
445 END
797 REM ************************
798 REM * SCREEN INPUT ROUTINE *
799 REM ************************
800 TE$=""
805 GET T$:GOSUB 900:IF T$="" THEN 805
810 IF T$=CHR$(13) THEN RETURN
815 IF T$=CHR$(20) THEN 860
820 IF T$<CHR$(32) OR T$>CHR$(95)THEN805
830 IF LEN(TE$)>LI THEN 805
840 TE$=TE$+T$
842 PRINT T$;
855 GOTO 805
860 IF LEN(TE$)=0 THEN 805
865 TE$=MID$(TE$,1,(LEN(TE$)-1))
870 PRINT" ";BS$;BS$;" ";BS$;
875 GOTO 805
899 REM ** FLASH CURSOR **
900 CT=CT+1
905 IF CT<>9 THEN RETURN
910 FL=FL*-1:CT=0
915 IF FL=1 THEN PRINT " ";BS$;;GOTO 925
920 PRINT CHR$(18)" "CHR$(146);BS$;
925 RETURN
1097 REM ***********************
1098 REM * READ ERROR CHANNEL *
1099 REM ***********************
1100 INPUT#15,EC,ED$
1105 RETURN
```

6 program design

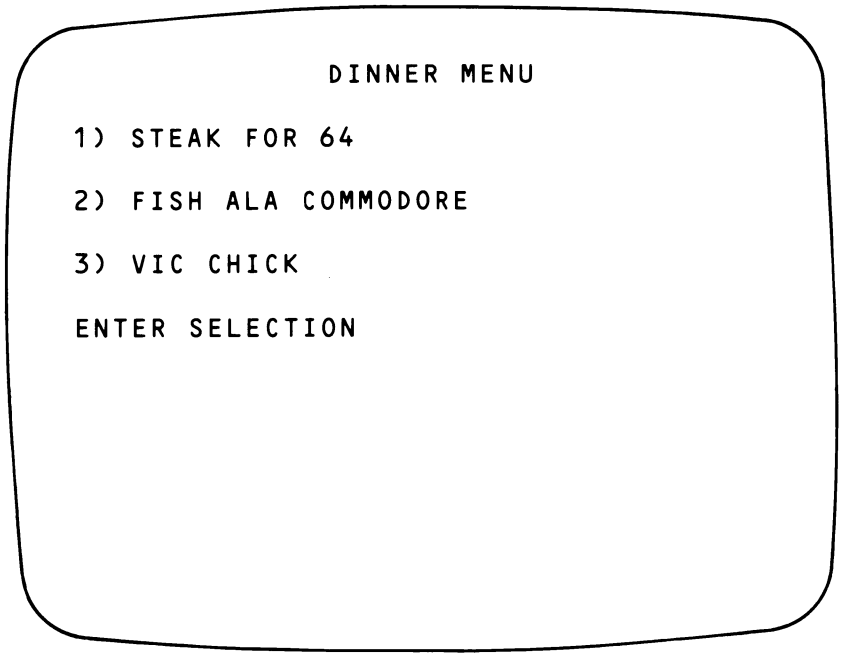
Developing a program requires more than just programming knowledge and skills. Choices have to be made by the programmer that will affect the entire program. These choices range from the type of file structure to be used to deciding what the user will see on the screen and how he will interact with the program. All of these fall into the category of program design. This chapter will discuss several areas to be considered when writing programs.

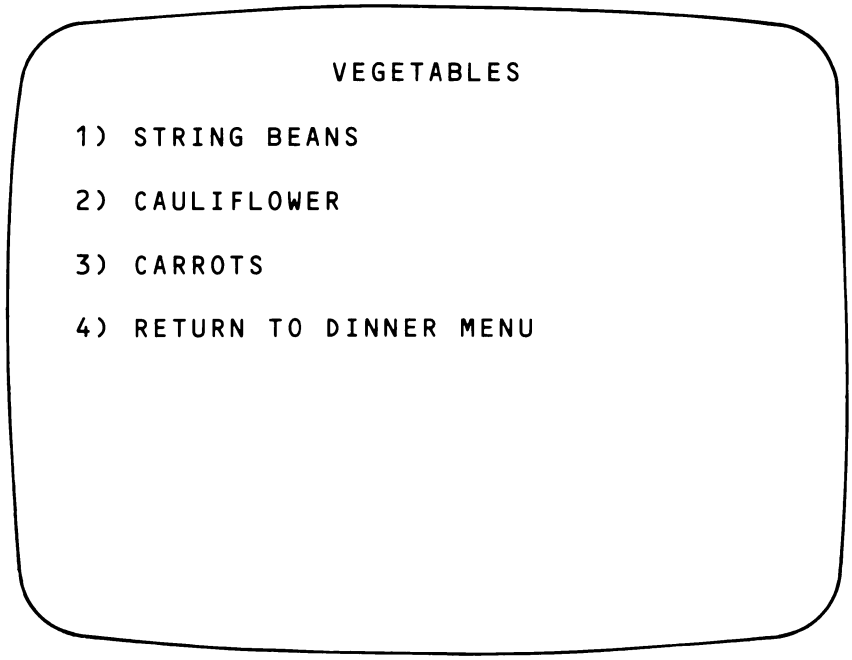
menu vs. command

A major choice that has to be made early in the design process is whether the program will be menu-driven or command-driven. Both have their advantages and drawbacks.

In a menu-driven system, the user is always provided with a list of the possible program options. He is then prompted to enter a code for the option chosen. The PHONEBOOK programs developed in the previous two chapters are all menu-driven. They are, however, very simple systems. Most programs require many levels of menus, starting with a main menu displayed when execution starts.

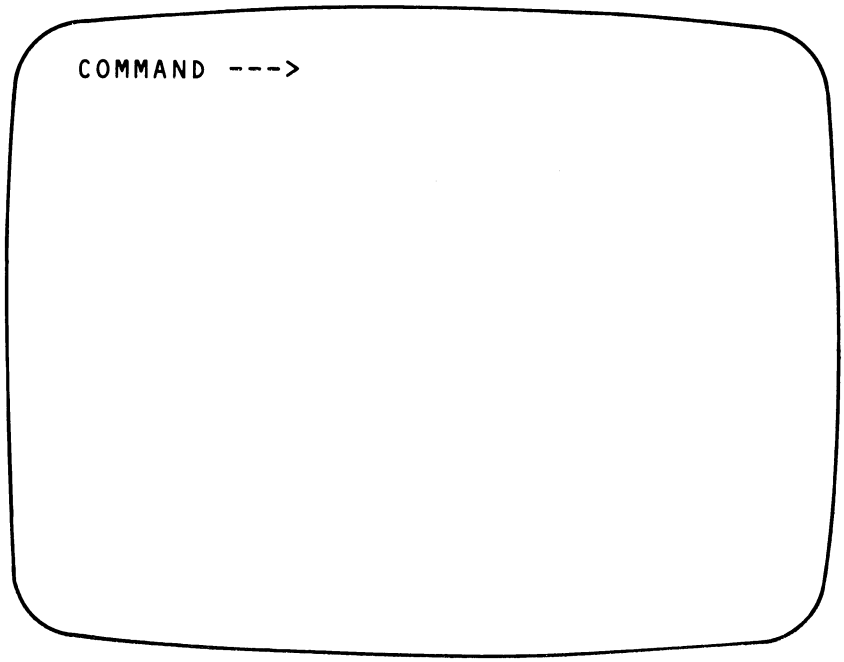
The options chosen from the main menu lead to submenus. Several levels of submenus are frequently present.





The trend in software has been towards menu-driven systems, because they are simpler to use and easy to learn.

In a command system, the user enters a command or command sequence obtained from a manual. One command system you are already familiar with is the Disk Operating System (DOS) commands for the Commodore 64. Any DOS operation can be done by entering one line of instructions. If a menu system were used, you would first have to select an option from the main menu to indicate that you wanted to enter a DOS command. Then from a DOS submenu, you would enter an option such as SAVE or LOAD. Finally, you would be prompted for the file name. This is convenient for the novice user but soon grows tedious.



Command systems are easier to program than menu systems and are faster since a single command can usually replace several options entered into several menus. Which should you implement in your programs? The choice depends on the program. A simple program like the PHONEBOOK example, where there is only one level of options, should be menu-driven. In a more complicated system, a choice has to be made between ease of learning and ease of use. The two systems are not incompatible. Programs can employ both systems in separate parts of the program. They can also be combined. A command system can have a command that would provide a list of all the possible options at that point.

provide editing features

Whenever a program requires input, there should be some opportunity to edit and correct that input. This is a major factor in making a program user

friendly. When data is being entered, the delete key should be operational. All the input routines developed in Chapter 2 demonstrate this technique. It is also convenient to have the right and left cursor keys available for editing. An input routine providing this facility was also developed in Chapter 2.

After the data has been entered, the user should have the chance to inspect the input and if necessary to go back and make corrections. Full screen editing is the preferable way to do this, except it is difficult to implement in BASIC and would execute slowly. An alternative, used in many programs, is to prompt the user with DO YOU WISH TO MAKE ANY CHANGES?(Y/N) or something similar. The user can then be prompted for which input item to change (if more than one exists) and given a chance to re-enter the data. I try to provide this editing feature whenever a program displays data.

double-check before killing anything

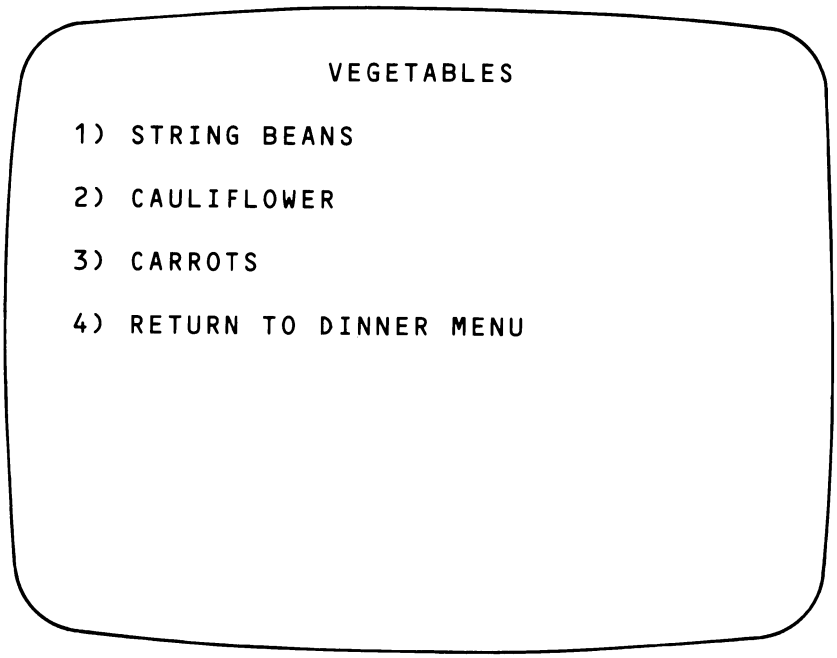
Nothing is more frustrating and aggravating to the user than to have work deleted after accidentally hitting the wrong key. Most well-written programs will prompt the user a second time before doing anything that would destroy a substantial amount of work. A prompt of ARE YOU SURE YOU WANT TO DELETE THIS FILE(Y/N) or something similar will save the user a lot of work, and win a new friend.

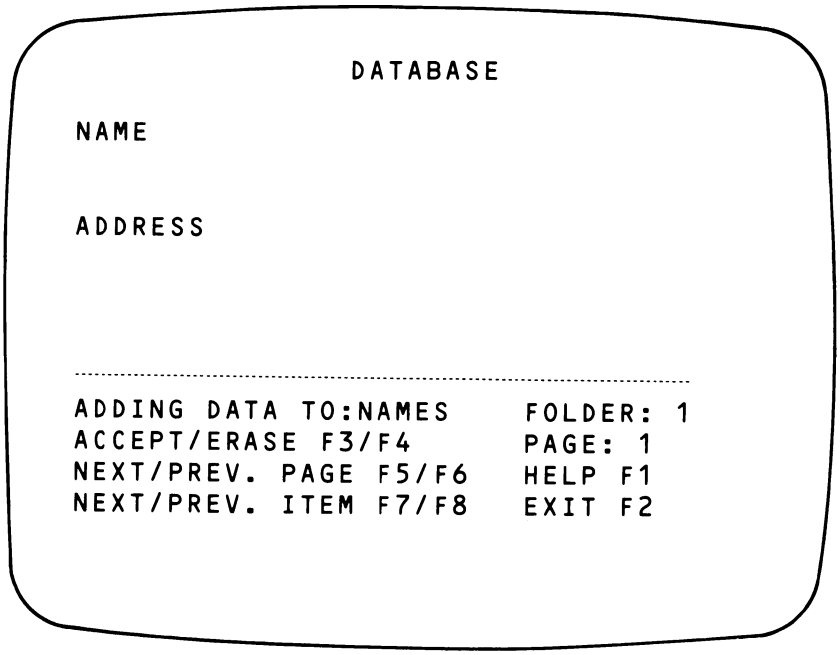
provide a chance to escape

In a menu-driven system, always provide a way to return from a sub-menu to the previous menu. This will make it possible to undo an errant keystroke. I have used too many programs that forced me to turn off the computer and start over because I had selected the wrong menu option, and there was no chance to go back to the previous menu.

use function keys where they make sense

In the last several years there has been a lot of emphasis placed on the use of function keys, and in Chapter 3 we discussed how to use them in your programs. The question still remains, do I include the use of function keys in my program or not? Unfortunately, there is no simple yes or no answer to this question. I only use function keys in situations where a menu would not be appropriate. An example of this type of situation is a program in which a user can either enter data or use a function key to exit, as opposed to a menu, in which only program options are entered. Some programs use function keys to select many options. Letters or numbers are better suited for this purpose. When I use function keys, I always reserve part of the screen to display a reminder of which function key does what.


VEGETABLES
1) STRING BEANS
2) CAULIFLOWER
3) CARROTS
4) RETURN TO DINNER MENU


DATABASE
NAME
ADDRESS
ADDING DATA TO:NAMES
ACCEPT/ERASE F3/F4
NEXT/PREV. PAGE F5/F6
NEXT/PREV. ITEM F7/F8
FOLDER: 1
PAGE: 1
HELP F1
EXIT F2

screen layout

Information should always be well placed on the screen. I have seen many programs where this was not done, and it is a real turn-off to the user or the prospective buyer. The computer screen is as much a communications medium as a newspaper or advertisement, and information displayed on it should have a professional touch. The more appealing the layout of information, the easier and the less threatening it is to use. Some points to consider when designing screens are

use the full screen

The Commodore's screen is 40 columns wide by 25 rows high. Use as much of it as possible. Here is the same menu displayed in two different ways.
The second menu is obviously easier to read and to use.

pick eye-appealing screen, border, and character colors

One of the Commodore 64's unique abilities is the 16 possible colors that can be used for the screen, the screen border, and the characters. They should all be set in easy-to-read, eye-appealing combinations. I have seen programs, including one popular word processor, in which the color combination was practically unreadable. I prefer the screen and the border to be the same color, dark blue with light blue letters or a black screen with white letters. The color combination displayed by the computer when it is turned on is not bad either. Remember that all users do not have color monitors, so the color combination picked has to show up well on a black and white or green screen.

To change the border color - POKE 53280, color code
To change the screen color - POKE 53281, color code

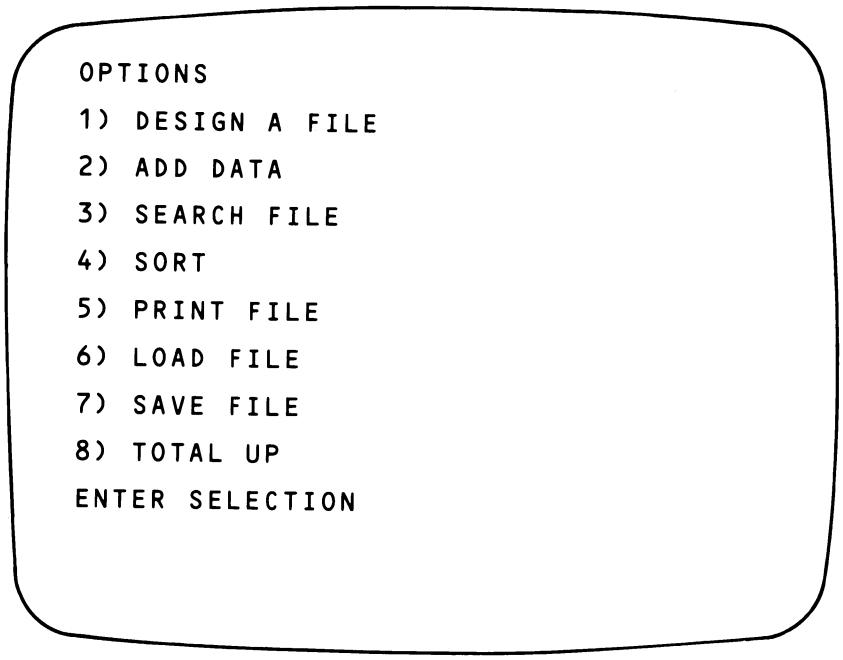
Where the color codes are:

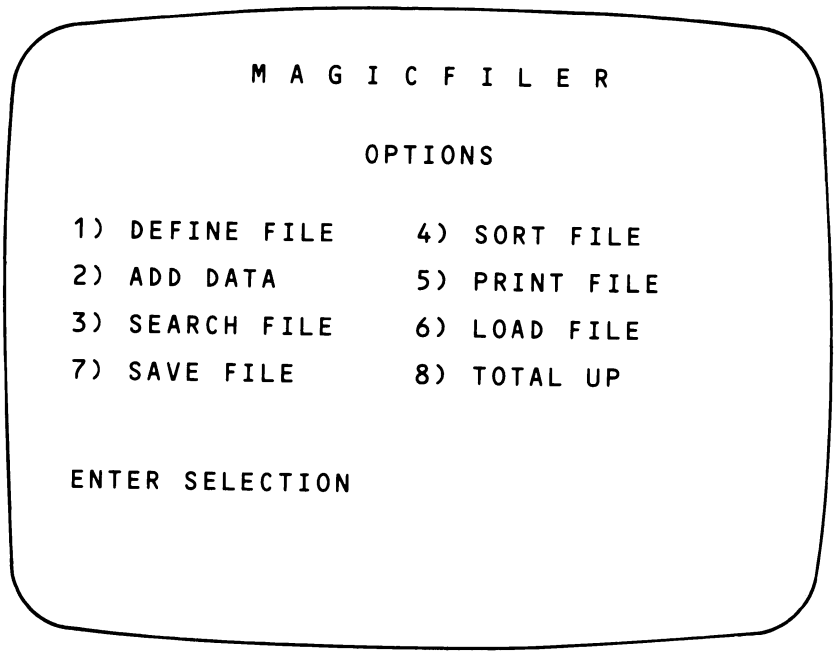
0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Gray 1
4	Purple	12	Gray 2
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Gray 3

The character color is changed by printing an ASCII screen code.

Character color

5	White	151	Gray 1
28	Red	152	Gray 2
30	Green	153	Light Green


OPTIONS
1) DESIGN A FILE
2) ADD DATA
3) SEARCH FILE
4) SORT
5) PRINT FILE
6) LOAD FILE
7) SAVE FILE
8) TOTAL UP
ENTER SELECTION


M A G I C F I L E R
OPTIONS
1) DEFINE FILE
2) ADD DATA
3) SEARCH FILE
7) SAVE FILE
4) SORT FILE
5) PRINT FILE
6) LOAD FILE
8) TOTAL UP
ENTER SELECTION

31	Blue	154	Light Blue
129	Orange	155	Gray 3
144	Black	156	Purple
149	Brown	158	Yellow
150	Light Red	159	Cyan

use reverse video to highlight information

Reverse video mode is handy for making information stand out on the screen. An ASCII code (CHR$(18)) is used in a PRINT statement to switch to reverse video. Anything printed afterwards is in reverse. The reverse mode is automatically cancelled after the PRINT statement, or it can be turned off in the PRINT statement with CHR$(146).

saving space/compacting a program

The Commodore 64 has a lot of memory space (64K bytes) of which 38911 bytes are available to store both a BASIC program and the value of its variables. At any time the amount of remaining memory can be determined with the FRE function. The statement

```
X =FRE(Ø)
```

assigns to X the number of bytes of memory available for use. In programs where memory space is tight, the remaining memory should be checked before placing any more data in memory. In any large program a balance has to be struck between the program size and the amount of data that can be stored. This tension can be eased by using memory space more efficiently. I have compiled two lists on how to save memory. One shows how to compact programs, the other how to store data efficiently.

compacting a program

There are several steps that can be taken to decrease the size of a BASIC program. Many of them, however, affect the readability of the program and limit the ability to make changes to the program. They should only be done after a program is fully debugged. A good practice is to have two versions of a program, the original and a compacted version.

A brief discussion of how BASIC instructions are stored in memory will be useful here. For every program line, two bytes are used to store the line number. No matter how large or small the line number is, it takes up the same amount of memory. Three more bytes are used for internal bookkeeping, two are used to keep track of the next instruction, and one to mark the end of the instruction. This makes the total overhead for each line five bytes. Any BASIC keyword (IF, GOTO, THEN, REM, etc.) appearing in a line occupies one byte regardless of its

length. This is because keywords are stored in memory as a numeric code called a *token*. When they are listed to the screen, they are expanded. Everything else on a line, such as variable names or numeric values, occupies one byte per character.

```
   10  X=4          occupies 8 bytes
10000  X=4          occupies 8 bytes
10000  GOTO 100     occupies 9 bytes
10000  GOTO 9999    occupies 10 bytes
```

remove REM statements

REMs waste a lot of space: the four byte overhead, one byte for the keyword, and one byte for every character afterwards. It makes sense to program using REMs and then remove them in a final version. When programming, be sure not to branch to a REM statement. This will save you a big headache when you remove them.

```
DO THIS              NOT THIS

110 GOTO 200         110 GOTO 199
  .                    .
  .                    .
199 REM SEARCH       199 REM SEARCH
200 X=3              200 X=3
```

remove spaces in the program

BASIC doesn't require that any spaces be placed in a program line. One byte is saved for each space removed.

```
100 IF X=4 THEN 100
```

This line could be written without spaces, saving three bytes.

```
100 IFX=4THEN100
```

Of course the readability of the program is severely hampered.

put more than one statement on a line

Placing more than one statement on a line saves the four bytes for each extra line included.

```
10 X=4
20 Y=3
30 Z=2
```

Eight bytes can be saved by converting these three statements into one.

```
10 X=4:Y=3:Z=2
```

I make it a practice always to group together statements that are logically connected. A tremendous amount of space can be saved by doing this without damaging readability too badly.

stop line number inflation

If line numbers are kept low, space can be saved when lines are referenced in GOTO or GOSUB statements. This turns out to be only a minor space saver.

use variables instead of constants

In the MAGIC FILER program, presented in the next chapter, the word "record" is printed many times. It would occupy six bytes in every PRINT statement, plus two bytes for the quotes. Instead, I assigned it to the string variable R$ which occupies only two bytes in each PRINT statement, although it adds one assignment statement to the program. If the word "record" were printed 20 times, a total of 108 bytes would be saved. This technique can also be done with numeric values. Each digit in a numeric value takes one byte. If the same value is used many times, it can be replaced by a variable.

use single letter variables

When possible use a single letter as a variable name rather than a two-character name. This saves one byte every time that variable is used. In a large program the savings can be considerable.

do not use an END or STOP statement

Some programs do not need an END or STOP statement and excluding it saves five bytes.

use subroutines

Any section of a program that is longer than a line and is repeated several times, can be placed into a subroutine. This saves significant space in a program as well as typing. There is a catch. Execution speed will be adversely affected. A happy medium is to use subroutines only for groups of four or more statements that are repeated more than twice.

use TAB and SPC

In PRINT statements, instead of using cursor characters use the TAB and SPC functions to position the output. The cursor position subroutine we wrote in Chapter 2 does not save space because of the line needed to call it. However it is invaluable in certain applications.

saving variable space

Variables in memory occupy different amounts of storage depending upon their type and, in the case of strings, their contents.

Integers (A%) Occupy 7 bytes, not the 4 Commodore claims—2 for the name, 2 for the value, the remaining 3 are wasted.

Reals (A) Occupy 7 bytes—2 for the name, 5 for the value (1 exponent, 4 mantissa)

Strings (A$) Occupy 7 bytes plus 1 for each character it contains.

Integer Arrays 5 bytes for the name
+ 2 bytes for each dimension
+ 2 bytes for each element

DIM A%(10) would allocate 29 bytes.
5+2+2*11elements (remember the zeroth element)

DIM A%(2,3) would allocate 33 bytes.
5+4+2*12elements

Real Arrays 5 bytes for the name
+ 2 bytes for each dimension
+ 5 bytes for each element.

DIM A(10) would allocate 62 bytes.
5+2+5*11

DIM A(2,3) would allocate 69 bytes.
5+4+5*12

Unlike real and integer arrays, string arrays are not allocated all the memory they need by the dimension statement. Rather, they grow as character strings are assigned to the elements. In a program tight for memory space, the size of strings should be watched to avoid an out-of-memory error.

String Arrays 5 bytes for the name
+ 2 bytes for each dimension
+ 3 bytes for each element
+ 1 byte for each character in each string element

use integer rather than real arrays

Integer variables take up the same amount of space as real variables, but elements in integer arrays save three bytes per element over real arrays. Therefore, it makes sense to use integer arrays whenever possible.

use the zero element in arrays

The zero element in an array is just sitting there waiting to be used. I usually avoid using it, except when absolutely necessary.

reuse scratch variables

Reuse variables whose values are no longer needed rather than declaring new ones. Try to use only one variable as the index in all FOR loops. Of course, in nested FOR loops different index variables must be used. If more than one GET statement is used, try to use the same variable as the argument each time.

clean up after your strings

When the values of string variables are changed, the old string is not erased from memory. Using the FRE function in your program will clean up old strings sitting in memory. There is a drawback to this technique. The FRE function can take a substantial time to work if there are a large number of active and inactive strings.

speeding up a program

In BASIC, the execution speed of a program can depend on how it is written. Here are some ways to speed up a program when execution time is critical. Some of these techniques are the same as those intended to save space.

use variables instead of constants

Besides saving memory space, using variables speeds up execution. A constant in an expression has to be evaluated each time the line is executed. Here is a comparison.

```
                                5 T=1000
10 FOR I=1 TO 1000              10 FOR I=1 TO 1000
20 X=X+1000                     20 X=X+T
30 NEXT I                       30 NEXT I

execution
time        7.5 seconds         4.5 seconds
```

declare frequently used variables first

In BASIC, variables are stored in a list. Variables are added to the end of the list when they are used for the first time in the program. Every time a variable is referenced, the list is searched sequentially for that variable. Therefore, variables that are declared early in a program are found more quickly, saving time.

use NEXT statements without the index variable

```
10 FOR I= 1 TO 10
20 PRINT I
30 NEXT I
```

In the preceding program, the I in NEXT I is not needed. Eliminating it does not affect the operation of the program and provides a slight speedup. The increase in execution speed seems to be 8%.

disable the RUN/STOP key

In most programs it is not desirable to allow the RUN/STOP key to be used to stop the program as it runs. Disabling it will prevent disastrous effects, such as interrupting a calculation or skipping an exit routine. The RUN/STOP and RESTORE keys can be disabled from inside a program with

```
POKE 808,254
```

At the same time, it is wise to lock in whichever character set is being used, upper case or upper and lower case. This is a two step process, first select the character set and then disable the SHIFT/COMMODORE key combination. Strangely enough, this is done with screen codes.

`PRINT CHR$(14)`	Switches the screen to lower and upper case characters.
`PRINT CHR$(142)`	Switches the screen to upper case characters.
`PRINT CHR$(8)`	Disables the SHIFT/COMMODORE key combination
`PRINT CHR$(9)`	Enables the SHIFT/COMMODORE key combination

always let the user know what is happening

Nothing is more distressing to a user than when the computer seems to be doing nothing. *The computer actually may be doing* something like a sort or loading a file from the disk, but it looks like nothing is happening. Quite often panic sets in and irrational things, such as turning off the machine, may be done. When an operation is to take a long time, I always let the user know what is happening. I clear the screen and display the name of the operation, SAVING, SEARCHING, etc. I then display stars across the screen as the program executes. This indicates to the user that the program is still working. This can be done with a simple PRINT statement from inside the loop that is executing.

```
100 INPUT#2,A$(I)
110 PRINT "*";
120 I=I+1
130 IF ST<>64 THEN 100
```

use a boot program

An idea related to the previous one is to use a boot program. The Commodore 64 loads BASIC programs very slowly and can leave the user wondering if anything is happening. Even a relatively short program can take a minute to load. A boot program is a very short program (two or three lines long) which prints a message and then loads the main program.

```
10 PRINT CHR$(147) :REM CLEAR THE SCREEN
20 PRINT "LOADING PLEASE STAND BY"
30 LOAD "program",8
```

When used from inside a program, LOAD both LOADs and RUNs a program held in a file on the diskette. This technique is also known as chaining, and can be used to split up a program too large to fit in memory. Any variables used by the first program are available to the new one being loaded.

group delays together

No matter how wonderful they are, microcomputers are slow. Delays may occur at various points in a program. They occur when a file is read or searched, when extensive calculations are being done, or for numerous other reasons. When possible, it is preferable to combine the delays rather than to scatter them throughout the program. One long delay is less frustrating for the user than several shorter delays scattered in the program. At least the user can go grab a sandwich and a beer.

7 magicfiler: a database manager

Perhaps the most powerful task that a computer can perform is information management. The Commodore 64 can replace manual filing systems with programs that access data stored in files. The operations done in manual systems, such as storing and retrieving information, searching for specific information, making changes, printing the information and maybe some mathematical operations, can all be easily replaced by a program. For example, you can write a program to store income tax expense information.

Since many applications require the same type of information management, a question arises. Do I write a special program for each information storage application, or do I write a single, more generalized program that can be used for many applications? The answer lies in the fact that many applications are so similar that one generalized program would be a better choice. Database managers are such programs. A database is simply a collection of records and fields. Once a database manager is written, no programming is involved in using it to store, retrieve, or manipulate information. This appeals to nontechnical people who want to use computers and don't care about programming them. Database

managers shift the task of information management from writing a program to using a program.

In this chapter we will discuss MAGICFILER. MAGICFILER is a short but powerful database program that compares favorably to many other databases being sold for the Commodore 64. The design choices made prior to programming will be explained and then the program will be developed with section-by-section explanations. Many of the routines and concepts covered in this book will be applied. MAGICFILER is provided not only as a programming example, but also for your for personal recordkeeping, such as tax records and mailing lists. A user's guide to the program is provided in Appendix B.

design considerations

The major design criteria for MAGIGFILER were to make it easy to learn and easy to use. Such attributes are sometimes called user-friendliness. This was accomplished by designing MAGICFILER to be menu-driven and to have a consistent user interface. This means that the user will always see the same screen format whenever he is dealing with data. This allows the user to develop confidence in the program and assures him that no surprises will occur.

Speed of operation was also given importance in the design. I wanted all operations like sorts and searches to be done quickly. This speed requirement meant that all data would have to be kept in memory rather than in relative files, which are slow to use. Because memory will be used to store data, space for the program is at a premium. Because all operations on the data occur in memory, sequential files can be used to store the data on a diskette (no operations have to take place in a file). I prefer to use sequential files over a clumsy and harder-to-program ISAM system. Sequential files also require less programming to support, decreasing some of the demand on memory space.

The data structures for programs of this type are all similar. Records composed of fields are stored in a sequential file on the disk. At the start of the program the data is read into a two-dimensional string array, each row in the array representing one record and each column in that row a field. The array is dimensioned for as many columns as there are fields.

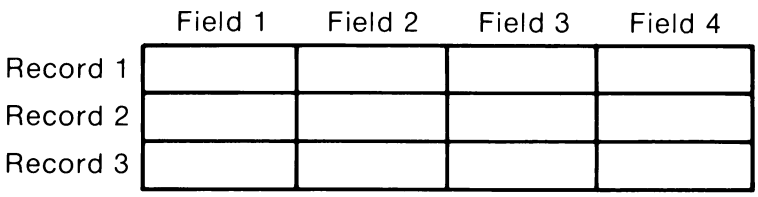


Figure 7.1

The maximum number of fields allowed is ten, the number of fields per record being stored in the first record of the file. Determining the maximum number of records allowed requires a balancing of the available memory space vs. the overhead required for the array. If the array is dimensioned too large there will be no room left for data; if it is dimensioned too small not enough records will be available for use. The solution is to allow 600 records for files of up to five fields and 400 records for files with six or more fields. The maximum number of characters per field chosen was 25. This allows each field to be displayed on one line of the screen. The last field is allowed to hold 255 characters to allow some flexibility. Each field is named and the name can be up to ten characters long. To avoid the overuse of computerese, the word "item" is used instead of "field" in the program. The two terms will be used interchangeably in the discussion of the program.

The functions included in MAGICFILER are

1. *Design a file* select the number of items (fields) and the item names.
2. *Add data* add records to the file.
3. *Search* flip through the file or search for occurrences of specific data.
4. *Sort* sort the file in memory with any field as the sort key.
5. *Print* print the file according to the specifications entered.
6. *Save* save a file on a diskette.
7. *Load* load a file from a diskette.
8. *Total* tally the contents of a field through the whole file.

All the options are available from the main menu.

writing the program

initialization

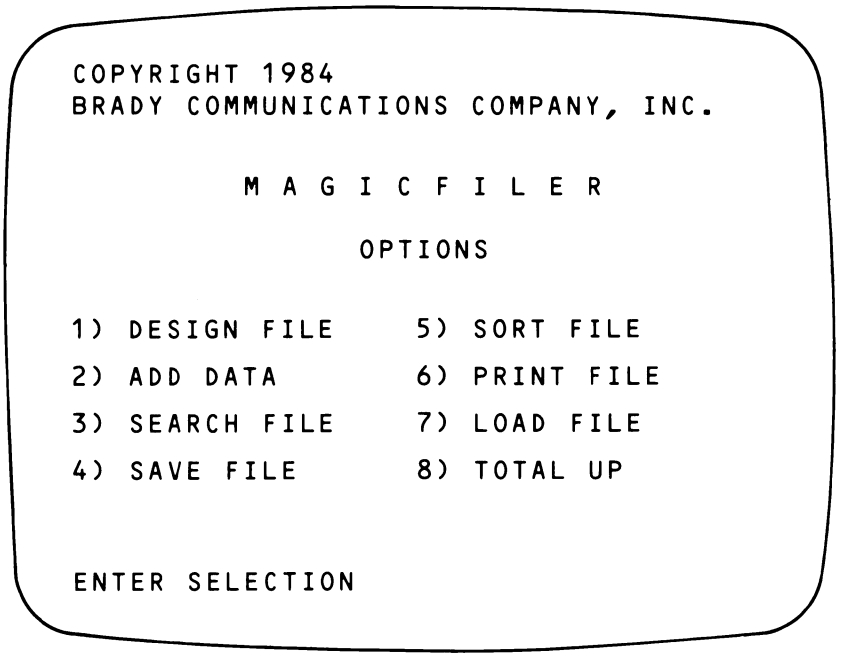
Here frequently-used strings are assigned to string variables to save space.

variables used

RX maximum number of records (see Chapter 6).
FL Used in cursor simulation (see Chapter 2).

```
1 REM ** MAGIC FILER **
2 REM POKE 808,254
3 POKE 53280,6:POKE 53281,6:PRINT CHR$(8):PRINT CHR$(154)
4 RX=600:FL=1
6 RO$=CHR$(18):OF$=CHR$(146):CL$=CHR$(147):BS$=CHR$(157)
8 F$=" FILE":R$="RECORD":SE$="SEARCH":I$=" ITEM"
```

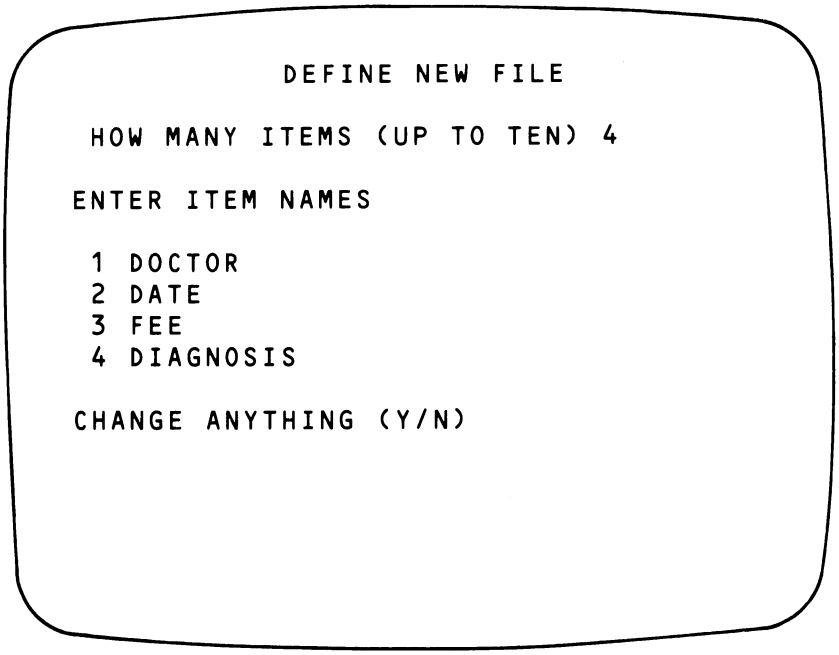
the main menu



The main menu is displayed and the selected option is branched to. All options eventually return to the main menu.

```
200 PRINTCL$"COPYRIGHT 1984 ROBERT J. BRADY CO."
210 PRINT:PRINT:PRINTTAB(8)"M A G I C F I L E R"
220 PRINT:PRINT:PRINTRO$TAB(14)"OPTIONS"
230 PRINT:PRINT:PRINT" 1)DESIGN"F$TAB(22)"5)SORT"F$
250 PRINT:PRINT"  2)ADD DATA"TAB(22)"6)PRINT"F$
265 PRINT:PRINT"  3)"SE$F$TAB(22)"7)LOAD"F$
280 PRINT:PRINT"  4)SAVE"F$TAB(22)"8)TOTAL UP"
290 PRINT:PRINT:PRINT: PRINT"   ENTER SELECTION ";
300 GOSUB1990
301 PRINTT$;
310 K=VAL(T$)
315 IFK<1ORK>8THEN200
320 ON K GOTO500,850,1100,3000,2350,1600,2500,1400
330 GOTO290
```

design file



Here the file is designed by the user. The user indicates how many fields will be used and what they will be called. The data array D$ is then dimensioned. A chance to edit the field names is provided.

variables used

D$() The data array.
IT$() Array that holds the item (field) names.
NO Number of items.

subroutines called

2000 Change anything prompt.
2100 Edit.
4900 Input routine.

We check for a file already in memory by seeing if the array that holds the item names is empty. If a file already exists the user has a chance to exit. Otherwise, the existing array is erased by resetting the array pointer.

```
497 REM ***************
498 REM * DESIGN FILE *
499 REM ***************
```

```
500 IFIT$(1)=""THEN 505
501 PRINTCL$:PRINT" DESTROY PRESENT"F$"(Y/N)";
502 GOSUB1990:IFT$<>"Y"THEN200
```

The user is prompted for the number of items to create and the array is dimensioned.

```
503 POKE 49,PEEK(47):POKE 50,PEEK(48)
505 PRINTCL$:PRINTRO$TAB(10)" DEFINE NEW"F$
510 PRINT:PRINT" HOW MANY"I$"S (UP TO TEN)";
520 GOSUB4900:NO=VAL(TE$):IF NO<1 OR NO>10THEN 505
523 IF NO>5 THEN RX=400
525 DIM D$(RX,NO)
```

The user is prompted for the item names and they are stored in the IT$ array.

```
530 PRINT:PRINT"ENTER"I$" NAMES"
580 FOR I=1TONO
590 PRINTI;:GOSUB4900:IT$(I)=TE$
600 NEXTI
```

Finally a chance to edit the input is provided.

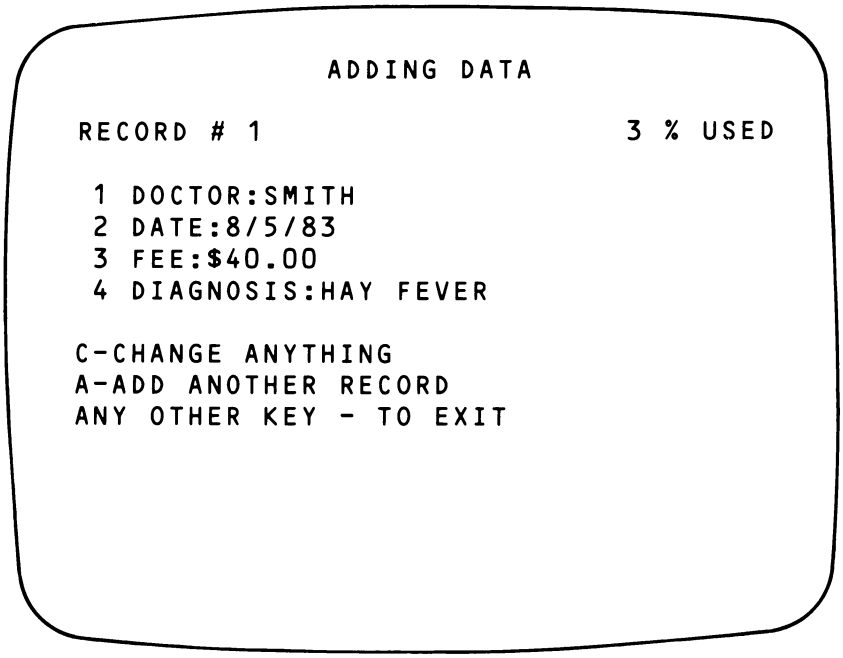
```
610 GOSUB2000 :REM CHANGE ANYTHING?
620 IF T$<>"Y"THEN200
640 GOSUB2100 : REM EDIT PROMPT
650 PRINT"ENTER NEW"I$" NAME"
660 PRINT NU;:GOSUB4900:IT$(NU)=TE$
670 PRINTCL$TAB(9)RO$"DEFINE NEW"F$:PRINT:PRINT
672 PRINTRO$I$" NAMES":PRINT:PRINT
675 FOR J=1TONO:GOSUB3600:PRINT:NEXT
680 GOT0610
```

add data

The add data routine places new records into the array in memory. The record number and the percentage of memory already used is displayed. After the data is entered the user can either add another record, edit the data, or return to the main menu.

variables used

D$() The data array.
IT$() Array that holds the item (field) names.
NO Number of items.
RE Number or records.
I Number of bytes free in memory.



K Length of previous record.
J Scratch variable.

subroutines called

1800 Count the number of records.
2000 Change anything prompt.
2100 Edit.
4900 Input routine.

First check if a file is already in memory.

```
847 REM ************
848 REM * ADD DATA *
849 REM ************
850 IF IT$(1)=""THEN4950
```

The percentage of memory used is computed by finding the amount of memory available (I) and dividing it by the maximum possible free space.

```
855 PRINTCL$:PRINT:PRINTTAB(12)"PLEASE WAIT"
857 P=600:IFNO>5 THEN P=400
858 P=P*NO*3+NO*5*3+500
860 GOSUB1800:RE=I:I=FRE(0):K=0
```

```
862 PRINT CL$TAB(12);RO$"ADDING DATA"
864 IF I<1Ø THEN PRINT"NO SPACE AVAILABLE":GOTO 21Ø
866 I=I-K:K=Ø
867 PRINTSPC(11)INT((1-I/(3Ø85Ø-P))*1ØØ)" % USED":PRINT
```

Prompt the item names and accept input. Remove leading blanks in the input (line 906).

```
88Ø FORJ=1TONO
89Ø GOSUB36ØØ:GOSUB49ØØ :REM PROMPT & GET INPUT
9Ø5 D$(RE,J)=TE$
9Ø6 IFLEFT$(D$(RE,J),1)<>" "THEN91Ø
9Ø7 D$(RE,J)=RIGHT$(D$(RE,J),LEN(D$(RE,J))-1):GOTO9Ø6
91Ø NEXTJ
```

After the data is entered display the menu.

```
92Ø PRINT:PRINT"C-CHANGE ANYTHING":PRINT"A-ADD ANOTHER
"R$
921 PRINT"ANY OTHER KEY - TO EXIT"
923 GOSUB1995
925 IFT$="A"THEN995
928 IFT$<>"C"THEN2ØØ
```

Here the "change a field" option is done.

```
93Ø GOSUB21ØØ :REM EDIT PROMPT
94Ø PRINTNU;RO$IT$(NU)OF$":";:GOSUB49ØØ:D$(RE,NU)=TE$
942 PRINTCL$R$"# ";RE:PRINT
943 FORK=1TONO:PRINTK;RO$IT$(K)OF$;":";D$(RE,K):NEXT
944 GOTO92Ø
```

After a record is entered, its length is determined and the record number is incremented.

```
995 FORP=1TONO:K=LEN(D$(RE,P))+K:NEXT
996 RE=RE+1:IFRE<=RXTHEN862
997 PRINT:PRINT"NO SPACE AVAILABLE":GOTO 21Ø
```

search

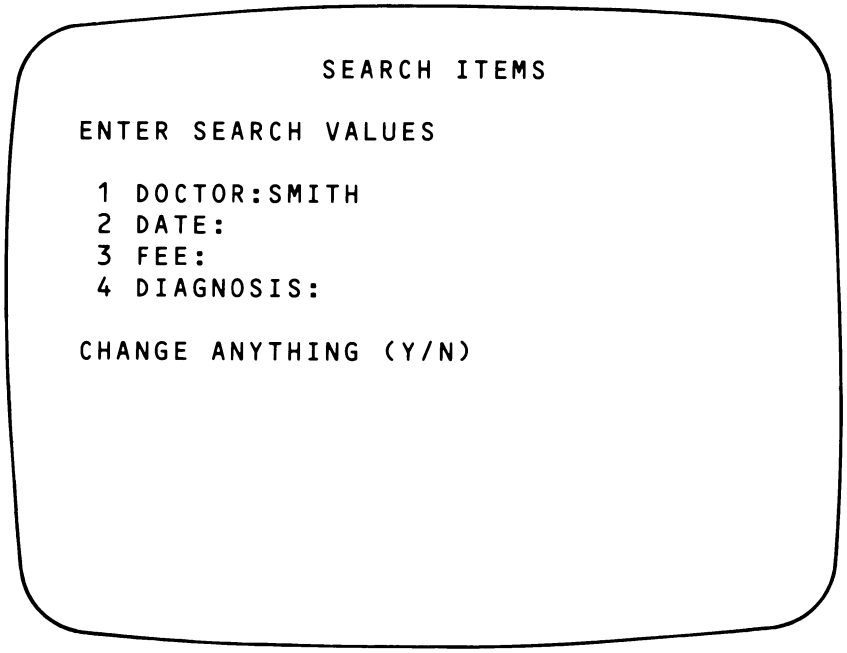
Search is the most complicated function in the program. There are two different search modes. One reviews the file in sequential order. The other mode allows the user to retrieve only those records that match a description that he enters. A menu is used to pick the mode of searching the file to be used.

The review option is simple, just display the records one at a time in sequential order. After displaying a record, provide options to return to the main menu, edit the data displayed, print the record, and display the next or previous record in the file.

The search option allows the user to enter a information into a "match record." The user is prompted with the item names and either data to be matched or a carriage return for "don't check this field" is entered for each item. The file is then search for records that are the same as the match record. There are six different options for entering informaion to be searched for.

ABC	finds all records with ABC in that field
A..	finds all records where that field starts with A
..C	finds all records where that field ends in C
/AB	finds all records where that field does not contain AB. The NOT operator can be combined with any other type of search
>12	finds all records where the value of that field is greater than 12.
<12	finds all records where the value of that field is less than 12.

The first record matching the description is displayed. An option to continue the search is provided along with the others.



variables used

D$() The data array
IT$() Array holding item names
S$() Holds the match record, data to search for is in the same field as it occurs in the records
NO The number of items
MX Number of records in memory
R Search type
1 for NOT search
2 for < search
3 for > search
RE record number
D$ used to hold a specific value of the data array D$()
S$ used to hold a specific value of the search data array S$()
L Length of S$
P Mode switch
0 for search mode
1 for review mode
NU Field being edited
I,J Scratch variables

subroutines called

1800 Count the number of records
2000 Change anything prompt
2100 Edit prompt
4900 Input routine

Display initial options.

```
1000 PRINTCL$TAB(10)RO$SE$" OPTION":PRINT
1101 IF IT$(1)="" THEN PRINT"NO FILE":FOR I=1 TO
300:NEXT:GOTO200
1102 PRINT"S - "SE$F$" FOR SPECIFIC"I$
1103 PRINT"R - REVIEW"F$" SEQUENTIALY":PRINT
1104 PRINT"ENTER OPTION";:GOSUB1990
1105 IF T$="R"THENRE=1:P=1:GOSUB1800:MX=I:GOTO1317
```

File is to be searched, prompt for the data to match.

```
1107 P=0
1108 PRINTCL$RO$TAB(13)SE$I$"S"
1109 IFIT$(1)=""THEN4950
1110 PRINT"ENTER "SE$" VALUES"
1120 FORJ=1TONO
1130 PRINT:GOSUB3600:GOSUB4900
```

```
1177 S$(J)=TE$
1178 IF LEFT$(S$(J),1)<>" " THEN 1180
1179 S$(J)=RIGHT$(S$(J),LEN(S$(J))-1):GOTO 1178
1180 NEXTJ
1190 GOSUB2000:IFT$<>"Y"THEN1250
1200 GOSUB2100
1210 PRINT:PRINT NU;" "RO$IT$(NU)OF$" ";
1212 GOSUB4900:S$(NU)=TE$
1215 PRINTCL$TAB(13)RO$SE$I$"S"
1220 FORI=1TONO:PRINTI;" "RO$IT$(I)OF$" "S$(I):NEXTI
1222 GOTO 1190
```

Search process is started.

```
1250 GOSUB1800:MX=I:REM FIND LAST RECORD
1252 RE=1
1254 PRINTCL$TAB(9)"S E A R C H I N G"
1255 IF RE>MX-1THEN1365
1256 PRINT"*";
```

Each field from a record is isolated and compared to the same field in the match record.

```
1260 FORJ=1TONO
1262 R=0 :REM SEARCH TYPE
1265 S$=S$(J):D$=D$(RE,J):L=LEN(S$)
```

Determine which type of search and remove the search symbol.

```
1268 IFS$=""THEN1310
1269 IFLEFT$(S$,1)="/"THENS$=RIGHT$(S$,L-1):R=1:L=L-1
1270 IFLEFT$(S$,1)=">"THENS$=RIGHT$(S$,L-1):R=3:L=L-1
1271 IFLEFT$(S$,1)="<"THENS$=RIGHT$(S$,L-1):R=2:L=L-1
1272 IFLEFT$(S$,2)<>".."THEN1276
```

Partial match is done along with checking for NOT operator (R=1).

```
1273 IFRIGHT$(S$,L-2)<>RIGHT$(D$,L-2)ANDR=0THEN1360
1274 IFRIGHT$(S$,L-2)=RIGHT$(D$,L-2)ANDR=1THEN1360
1275 GOTO1310
1276 IFRIGHT$(S$,2)<>".."THEN1282
1278 IFLEFT$(S$,L-2)<>LEFT$(D$,L-2)ANDR=0THEN1360
1279 IFLEFT$(S$,L-2)=LEFT$(D$,L-2)ANDR=1THEN1360
1280 GOTO1310
```

The other types of matches are made.

```
1282 IF(S$<>D$)AND(R=Ø)THEN136Ø
1284 IF (S$=D$)AND(R=1)THEN136Ø
1286 IFR=3ANDD$<=S$THEN136Ø
1288 IFR=2ANDD$>=S$THEN136Ø
131Ø NEXTJ
```

If a matching record is found, then display it.

```
1317 IF D$(RE,1)="?"THEN RE=RE+1:GOTO 1254
1318 PRINTCL$R$"# ";RE:PRINT
132Ø FOR K=1 TO NO:PRINTK;RO$IT$(K)OF$":"D$(RE,K):NEXT K
1324 PRINT:PRINTRO$"SELECT OPTION"
1325 IF P=OTHENPRINT"S-CONTINUE TO "SE$
1326 PRINT "C-CHANGE SOMETHING":PRINT"P-PRINT THIS "R$
1327 PRINT"U-UP ONE "R$:PRINT"B-BACK ONE "R$
1328 PRINT"D-DELETE THIS "R$:PRINT"ANY OTHER KEY - TO
EXIT"
1329 GOSUB1995
```

Display menu with options.

```
133Ø IF T$="S" AND P=OTHEN PRINTCL$;"S E A R C H I N G":
GOTO136Ø
1331 IF T$="U" THEN RE=RE+1:GOSUB1372:IF RE<MX THEN1317
1332 IF T$="B" THEN RE=RE-1:GOSUB1374:IF RE>Ø THEN 1317
1335 IF T$="P" THEN GOSUB138Ø:GOTO1317
```

Change/update the fields.

```
1336 IFT$<>"C"THEN1339
1337 GOSUB21ØØ:PRINTNU;RO$IT$(NU)OF$":";
1338 J=NU:GOSUB49ØØ:D$(RE,NU)=TE$:GOTO1317
```

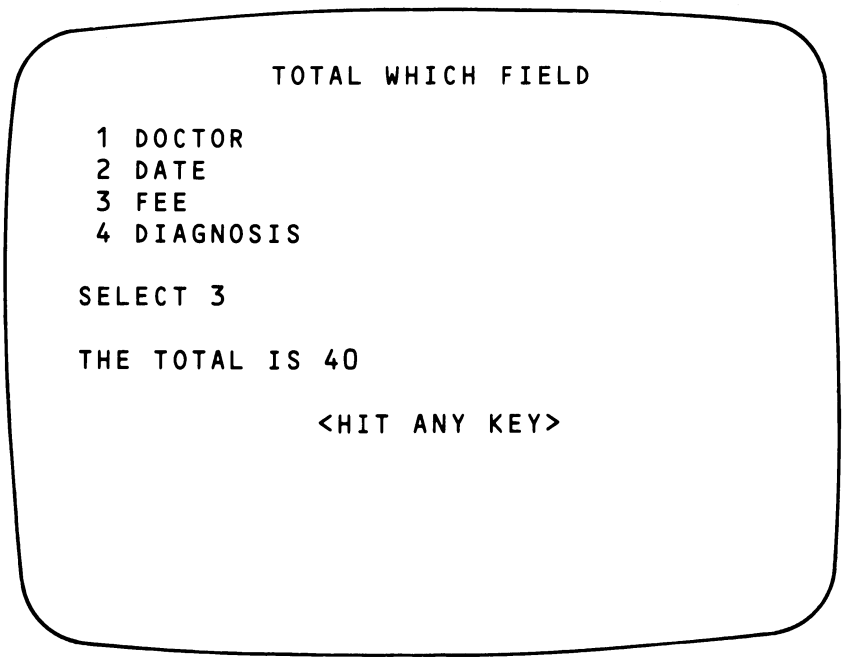
These two routines are used to check for deleted records when examining the next or previous record.

```
1339 IF T$<>"D" THEN 1342
134Ø PRINT"ARE YOU SURE(Y/N)";
1341 GOSUB199Ø:IF T$="Y"THEN D$(RE,1)="?
1342 FOR K=1TO1Ø:S$(K)="":NEXT K
1344 GOTO2ØØ
136Ø RE=RE+1:GOTO1255
1365 GOTO2ØØ
1372 IF D$(RE,1)="?"THEN RE=RE+1:RETURN
1373 RETURN
1374 IF D$(RE,1)="?"THEN RE=RE=1:RETURN
1375 RETURN
```

This is the routine to send a record to the printer.

```
1380 OPEN4,4:FOR K=1TO NO:PRINT#4,IT$(K)":"D$(RE,K)
:NEXT:PRINT#4,
1382 CLOSE 4:RETURN
```

Total



Total cycles through the file, adding together the contents of the indicated field. Since all data is stored as strings, the data to be added has to be converted first to a numeric value with the VAL function. Dollar signs are removed before adding.

variables used

D$()	The data array
MX	Record count
NU	Number of field being tallied
J	Holds total
K	Scratch variable

subroutines called

1800	Count the number of records
4900	Input routine

Prompt for which field is to be totaled.

```
1400 PRINTCL$:PRINT"OPTION"
1401 PRINT" 1) TOTAL":PRINT" 2) RETURN TO MAIN MENU"
1402 PRINT:PRINT"ENTER OPTION";:GOSUB1990:IF T$<>"1" THEN
200
1403 PRINTCL$TAB(10)"TOTAL WHICH FIELD":PRINT:PRINT:PRINT
1405 FORK=1 TO NO:PRINTK;RO$IT$(K):NEXT:IF IT$(1)=""THEN
200
1410 PRINT:PRINT:PRINT"SELECT ";:GOSUB4900:NU=VAL(TE$)
```

Count the number of records.

```
1420 GOSUB 1800:MX=I-1:J=0
```

Loop through the file, convert the field contents to a value.

```
1425 FORK=1TOMX
1426 IFLEFT$(D$(K,NU),1)<>"$"THEN1428
1427 J=J+VAL(RIGHT$(D$(K,NU),LEN(D$(K,NU))-1)):GOTO1429
1428 J=J+VAL(D$(K,NU))
1429 NEXT K
```

Print total.

```
1430 PRINT:PRINT"THE TOTAL IS ";J:PRINT:PRINT
1432 PRINTTAB(13)"<HIT ANY KEY>"
1435 GOSUB 1995:GOTO 200
```

print

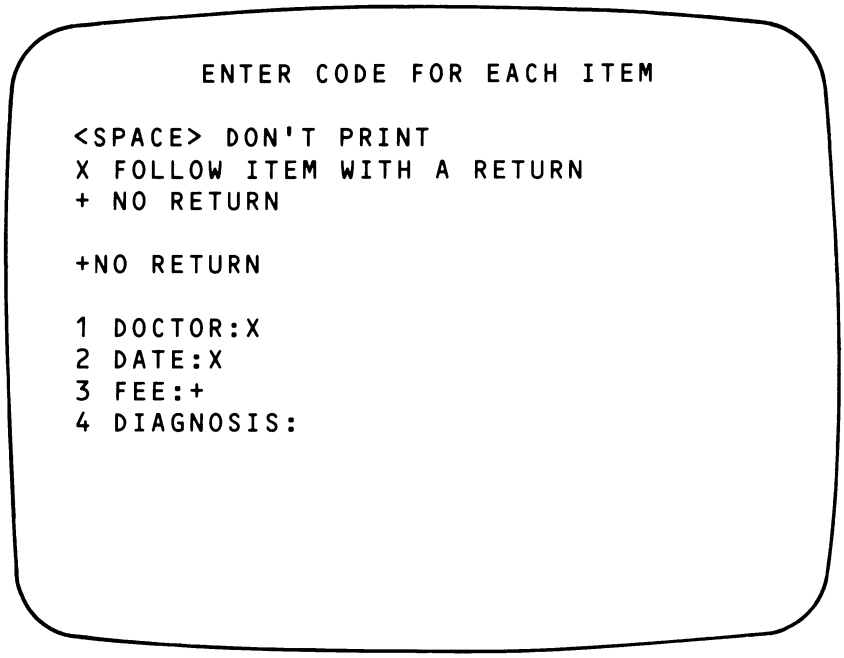
Print is used to dump all the records to the printer. Several options are available to describe the format of the printed output. A code is entered to specify how each item will be printed.

X	Print this item, then advance the printer to the next line
+	Print this item but skip two spaces rather than do a carriage return
space	Do not print this item at all.

The user is then prompted for whether the item names are to be printed and how many records are to be printed on a page.

variables used

IT$()	Array containing item name
D$()	The data array
P$()	Array containing the print codes
R	Records per page



CT Number of records already on the page
TE$ String read by input routine (4900)
J,I Scratch variables

subroutines called

1750 Print routine
1800 Count the number of records
1990 Get a character
1995 Get a character, no cursor
3600 Prompt item names
4900 Input routine

Check if a file exists in memory.

```
1597 REM *********
1598 REM * PRINT *
1599 REM *********
1600 PRINTCL$RO$TAB(14)"PRINT "F$
1601 PRINT "1) PRINT":PRINT"2) RETURN TO MAIN MENU"
1602 PRINT:PRINT"ENTER SELECTION";:GOSUB1990:IF
T$<>"1"THEN 200
1603 IF IT$(1)="" THEN 4950
```

Prompt the item names and accept the print code for each item. Since the

code is only one character long, a GET statement is used in lieu of the input routine.

```
1605 PRINT:PRINTTAB(7)"ENTER CODE FOR EACH"I$:PRINT
1610 PRINT"<SPACE> DON'T PRINT"
1615 PRINT"X  FOLLOW"I$" WITH A RETURN"
1620 PRINT"+  NO RETURN"
1622 OPEN4,4
1625 FOR J=1 TO NO
1630 GOSUB3600
1635 GOSUB1990
1640 IF T$<>" "ANDT$<>"+"ANDT$<>"X"THEN1635
1645 P$(J)=T$:PRINT T$
1650 NEXT
1655 PRINTCL$RO$TAB(13)"PRINT SPECS"
```

Prompt for how many records per page and whether to print item names or not.

```
1660 PRINTR$"S PER PAGE ":GOSUB4900
1665 R=VAL(TE$):IF R<1 OR R>15 THEN 1655
1670 PRINT:PRINT"PRINT"I$" NAMES<Y/N> ";
1675 GOSUB1990
1680 IF T$<>"Y" AND T$<>"N" THEN1675
1685 PRINTT$:TE$=T$
```

Set up printer message.

```
1686 PRINTCL$:PRINTTAB(9)"SET UP PRINTER"
1687 PRINT:PRINT:PRINTTAB(8)"THEN HIT ANY KEY"
1688 GOSUB1995:GOSUB 1800
```

Loop through the array (D$), calling the print routine and then return to the main menu.

```
1690 RE=1
1695 GOSUB1750
1700 FOR K=CT TO 66/R:PRINT#4,:NEXT K
1702 IF CT=R THEN CT=0
1710 RE=RE+1:IF RE<I THEN 1695
1715 CLOSE4
```

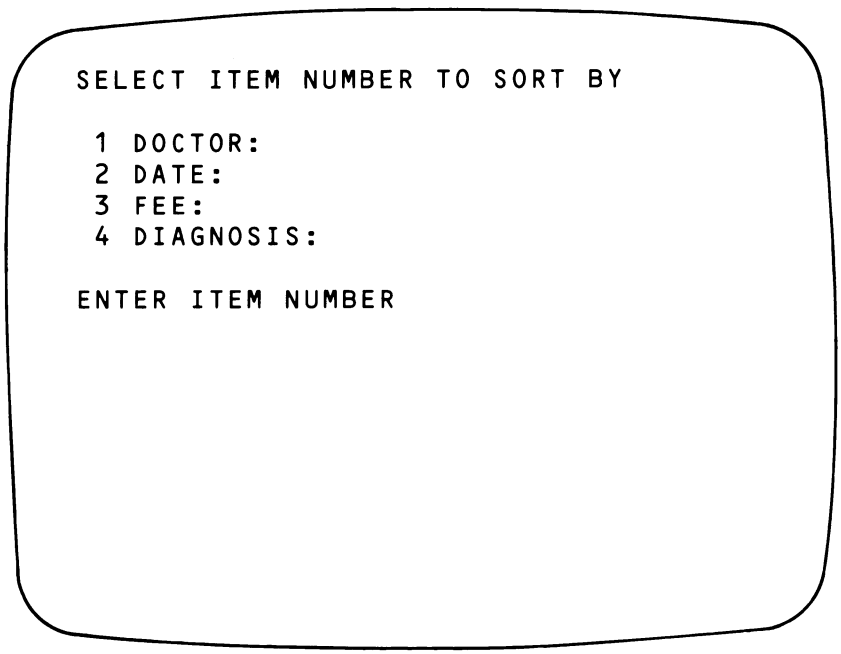
In the print routine, each print code is examined in a loop. One of two similar sets of statements is branched to; one prints the item names and the other doesn't. The print code for a field is looked at, the item is printed if designated, and a carriage return is added if needed.

```
1720 FOR K=1TO10:P$(K)="":NEXTK:GOTO200
```

```
1750 FOR K=1TO NO
1755 IF TE$="Y" THEN 1775
1758 IF P$(K) =" "THEN 1790
1760 PRINT#4," ";D$(RE,K)"  ";
1765 IF P$(K)="X" THEN PRINT#4,:CT=CT+1
1770 GOTO1790
1775 IF P$(K)=" "THEN 1790
1780 PRINT#4,IT$(K)":"D$(RE,K)"  ";
1785 IF P$(K)="X" THEN PRINT#4,:CT=CT+1
1790 NEXTK
1795 RETURN
```

sort

The sort feature will sort the array in memory using whatever field is specified as the sort key. The sort key is that field whose contents are used to place the records in sort order. The records are sorted in descending order. The original order of the data in memory is destroyed, but the file on the diskette is not affected. The insertion sort algorithm is used rather than the bubble sort because of its better performance. Unfortunately, the code is very hard to follow. The headache when sorting a two-dimensional array is the requirement that all the data in a row must be exchanged. A separate loop is needed for that task (see lines 2430, 2450, 2580).



variables used

D$()	The data array
B$()	Array used for temporary storage of a record
MX	Number of records
SF,NU	Item number of sort key
NO	Number of items per record
J,M,P	Scratch variables

subroutines called

1800 Count the number of records

First the item number of the sort key is prompted for.

```
2347 REM ******************
2348 REM * INSERTION SORT *
2349 REM ******************
2350 PRINTCL$
2352 IF IT$(1)="" THEN4950
2353 PRINT"OPTIONS":PRINT"1) SORT":PRINT"2) RETURN TO
MAIN MENU"
2354 PRINT:PRINT"SELECT OPTION";:GOSUB 1990:IF T$<>"1"
THEN 200
2355 PRINTCL$"SELECT"I$" NUMBER TO SORT BY"
2360 FOR J=1TO NO:GOSUB3600:PRINT:NEXT
2370 PRINT:PRINT"ENTER"I$" NUMBER ";
2380 GOSUB4900:NU=VAL(TE$):IF NU<1ORNU>NO THEN2370
```

Then the insertion sort routine is performed.

```
2400 PRINTCL$:PRINTTAB(12)"S O R T I N G"
2405 GOSUB1800:MX=I-1
2406 SF=NU
2410 FOR M=1 TO MX-1
2420 J=M
2430 FOR P=1 TO NO:B$(P)=D$(M+1,P):NEXT P
2435 PRINT"*";
2440 IF B$(SF)>D$(J,SF) THEN 2480
2450 FOR P=1 TO NO:D$(J+1,P)=D$(J,P):NEXT P
2460 J=J-1
2470 IF J>=1THEN2440
2480 FOR P=1 TO NO:D$(J+1,P)=B$(P):NEXT P
2490 NEXTM
2495 GOTO200
```

save file

SAVE FILE stores all the information held in memory in a diskette file. First, the number of items per record (NO) is stored, then the item names, and finally each record. An end of file (EOF) marker is placed after the last record to facilitate easy loading of the file.

variables used

D$()	The data array
IT$()	Array holding the item names
G$,F$	Name of the diskette file
NO	Number of fields per record
MX	Number of records
EC	Error code from error channel
ED$	Error description from the error channel
J,K	Scratch variables

subroutines called

1800	Count the number of records
3300	The disk operations menu
3650	Get file name prompt

Check if the file exists and give the user a chance to escape.

```
2997 REM *************
2998 REM * SAVE FILE *
2999 REM *************
3000 IF IT$(1)=""THEN4950
3005 GOSUB3300:ON VAL(T$)GOTO3100,200
```

Initialize the drive just in case the diskette has been changed and then open the file.

```
3100 OPEN15,8,15
3103 GOSUB 3650:REM GET FILE NAME
3105 PRINTCL$TAB(13)"S A V I N G"
3107 PRINT#15,"I"
3110 OPEN2,8,2,"@0:"+G$+".DATA,S,W"
```

Store the number of items and the item names.

```
3115 GOSUB1800:MX=I-1
3120 INPUT#15,EC,ED$
3130 PRINT#2,NO
3140 FOR J=1 TO NO
3150 PRINT#2,IT$(J)
3160 NEXT J
```

Store the records, skipping any that have been deleted.

```
3165 J=Ø
3170 J=J+1
3175 IF D$(J,1)="?" THEN3170
3180 FOR K=1 TO NO
3185 IF D$(J,K)="" THEN D$(J,K)=" "
3190 PRINT#2,D$(J,K)
3195 PRINT"*";
3200 NEXTK
3210 IF J<MX THEN3170
```

Place the EOF marker and close the file.

```
3220 PRINT#2,"?"
3270 CLOSE2:CLOSE 15
3280 GOTO200
```

load file

LOAD FILE reads the contents of a sequential file stored by the STORE FILE option. The INPUT# statement is used to read all the data except the last field in each record which may contain up to 255 characters and is read with the GET# statement.

variables used

D$()	The data array
IT$()	Array holding the item names
G$,F$	Name of the diskette file
NO	Number of fields per record
MX	Number of records
EC	Error code for error channel
ED$	Error description from the error channel
N,K	Scratch variables

subroutines called

1800	Count the number of records
3300	The disk operations menu
3650	Get file name prompt

Give the user a chance to escape, prompt for the file name, and open the file.

```
2497 REM *************
2498 REM * LOAD FILE *
2499 REM *************
```

```
2500 GOSUB3300:ON VAL(T$)GOTO2580,200
2580 GOSUB3650
2600 PRINTCL$:PRINTTAB(12)"L O A D I N G"
2605 OPEN15,8,15
2610 OPEN2,8,2,G$+".DATA,S,R"
2620 INPUT#15,EC,ED$
```

Check if the file is on the diskette.

```
2624 IF EC<>62 THEN2630
2625 PRINTCL$:PRINTTAB(5)F$" IS NOT ON THIS DISK"
2626 CLOSE15:CLOSE2:FOR K=1 TO 350:NEXT K:GOTO 200
```

Read the number of items.

```
2630 INPUT#2,NO
```

Reset the array.

```
2635 POKE 49,PEEK(47):POKE 50,PEEK(48)
```

Read the item names.

```
2640 FOR J=1 TO NO
2650 INPUT#2,IT$(J)
2660 NEXTJ
```

Dimension the array.

```
2663 IF NO>5 THEN RX=400
2665 DIM D$(RX,NO)
```

Read the first NO-1 fields, check for the eof marker.

```
2670 FOR K=1 TO 999
2680 FOR N=1 TO NO-1
2690 INPUT#2,D$(K,N)
2692 IF D$(K,N)="?"THEN D$(K,N)="":GOTO 2900
2695 PRINT"*";
2700 NEXTN
```

Read in the last field with GET#.

```
2702 GET#2,T$:IF T$<>CHR$(13)THEN
D$(K,N)=D$(K,N)+T$:GOTO2702
```

Close the loop and the file.

```
2710 NEXTK
2900 CLOSE2:CLOSE 15
2910 GOTO200
```

subroutines

1800 count the number of records

This routine counts the number of records in the data array. It is called by several parts of the program which loop through the array, such as SEARCH. Rather than examine the array sequentially, the routine divides and examines it in halves. This saves looking at half the array when one of the first RX/2 records contains data.

```
1799 REM ** COUNT NUMBER OF RECORDS **
1800 I=INT((1+RX)/2)
1801 FOR K=1TONO
1802 IF D$(I,K)<>"" THEN1805
1803 NEXT K
1804 J=1:GOTO1806
1805 J=I
1806 FOR I=J TO RX
1808 IF D$(I,1)=""THEN1830
1810 NEXTI
1830 IF NO=1 THEN1860
1832 FOR K=2TO NO
1835 IF D$(I,K)<>""THEN1850
1840 NEXTK
1845 GOTO1860
1850 J=I+1:GOTO1806
1860 RETURN
```

2000—change anything prompt

```
1999 REM ** CHANGE ANYTHING? **
2000 PRINT"CHANGE ANYTHING (Y/N)";
2010 GOSUB1990:PRINT" ":RETURN
```

2100—which field to change

```
2099 REM ** EDIT PROMPT **
2100 PRINT:PRINT"ENTER"I$" NUMBER TO CHANGE ";:GOSUB4900
2110 NU=VAL(TE$):IF NU<1ORNU>NO THEN2100
2115 RETURN
```

3300—disk options menu

```
3300 PRINTCL$:PRINT"OPTIONS"
3310 PRINT:PRINT"1) USE DISK"
3320 PRINT:PRINT"2) RETURN TO MAIN MENU"
3325 PRINT:PRINT"ENTER SELECTION ";
3340 GOSUB1990
3350 IF T$<>"1"AND T$<>"2" THEN3340
3355 PRINTT$
3356 PRINT
3360 RETURN
```

3600—item name prompt

This short routine is used by all sections of the program that prompt for data.

```
3599 REM ** PROMPT LINE **
3600 PRINTJ;RO$IT$(J)OF$":";:RETURN
3605 IF G$="" THEN3600
3610 RETURN
```

3650—get the file name

```
3650 PRINT "ENTER"F$" NAME ";:GOSUB4900:G$=TE$:RETURN
```

4900—input routine

This routine is similar to the input routine developed in chapter three. The only difference is that the maximum length of the data is not specified. The routine looks at the item number being entered and only allows the maximum number of characters for that field to be entered, either 25 or 255. The cursor routine directly follows the input routine.

```
4899 REM ** INPUT ROUTINE **
4900 TE$=""
4901 GET T$:GOSUB5000:IFT$=""THEN4901
4902 IF T$=CHR$(13)THEN PRINT" ":RETURN
4903 IF T$=CHR$(20)THEN 4915
4904 IF(LEN(TE$)>25 AND J<NO)OR(LEN(TE$)>254 AND
J=NO)THEN4901
4905 IF T$<CHR$(32) OR T$>CHR$(100) THEN 4901
4906 PRINTT$;:TE$=TE$+T$:GOTO 4901
4915 IFLEN(TE$)<=0 THEN 4917
4916 TE$=MID$(TE$,1,(LEN(TE$)-1)):PRINT" "BS$BS$" "BS$
4917 GOTO4901
4950 PRINT CL$"NO"F$" DESIGN":FOR KK=1TO350:NEXT:GOTO200
4999 REM ** CURSOR ROUTINE **
```

```
5000 KT=KT+1
5002 IF KT<>7THEN RETURN
5004 KT=0:FL=FL*-1
5006 IF FL=1 THEN PRINT" "BS$;
5008 IF FL=-1THEN PRINTRO$" "0F$BS$;
5010 RETURN
```

notes on typing magicfiler

It is worthwhile to type in MAGICFILER. This program is comparable in features and performance to many database filing systems sold for the Commodore 64 between $29 and $69. Many features in the program, including the search and print functions, outperform and offer more flexibilty than many other programs for the Commodore 64. Even used only as a mailing list program, it offers more features then many programs written explicitly for that purpose. MAGICFILER will be very handy for any of your recordkeeping tasks, such as tax records, recipes, business expenses, mailing lists, and even keeping track of what is on your diskettes. The program is listed in ready-to-type form. Screen display codes have been excluded from the program to make it easier to read and type. To save memory space for data, comments can be excluded (no REM line is branched to) and blanks can be excluded, but I recommend they be left in until you are sure you have typed the program correctly. Only two slight changes may have to be made in the program. The command on line 3 that disables the STOP key (POKE 808,254) should not be included until the program is fully debugged of typing mistakes. Otherwise, to stop the program you will have to shut off the computer, and this will make the program very hard to correct. The other change is line 867, where the percentage of memory used by the file is calculated. Depending on whether you included comments and spaces, the amount of memory space occupied by the program will vary and the percentage that appears can be erroneous. To find the right value to be used in the expression, finish typing the program, then do a PRINT FRE(0) in direct mode. Use the number returned to replace 30850.

If you own Simons' BASIC, I recommend not using the cartridge when running the MAGICFILER program. MAGICFILER does not use any of Simons' BASIC instructions and the ROM cartridge steals about 8000 bytes from the memory. It may also slow down the execution of some program segments.

If you have the available diskette, the program is in a file called MAGICFILER . There is a second version of the program already compacted in a file called SMALLMAGIC. See Appendix H about making backup copies of the program.

```
   1 REM ** MAGIC FILER
   3 POKE 53280,6:POKE 53281,6:PRINT CHR$(8):PRINT
CHR$(154):REM POKE 808,254
   4 RX=600:FL=1
```

```
  6 RO$=CHR$(18):OF$=CHR$(146):CL$=CHR$(147):BS$=
CHR$(157)
  8 F$=" FILE":R$="RECORD":SE$="SEARCH":I$=" ITEM"
 200 PRINTCL$"COPYRIGHT 1984 ROBERT J. BRADY CO."
 210 PRINT:PRINT:PRINTTAB(8)"M A G I C  F I L E R"
 220 PRINT:PRINT:PRINTRO$TAB(14)"OPTIONS"
 230 PRINT:PRINT:PRINT"  1)DESIGN"F$TAB(22)"5)SORT"F$
 250 PRINT:PRINT"  2)ADD DATA"TAB(22)"6)PRINT"F$
 265 PRINT:PRINT:  3)"SE$F$TAB(22)"7)LOAD"F$
 280 PRINT:PRINT"  4)SAVE"F$TAB(22)"8)TOTAL UP"
 290 PRINT:PRINT:PRINT: PRINT"   ENTER SELECTION ";
 300 GOSUB1990
 301 PRINTT$;
 310 K=VAL(T$)
 315 IFK<1ORK>8THEN200
 320 ON K GOTO500,850,1100,3000,2350,1600,2500,1400
 330 GOTO290
 497 REM ***************
 498 REM * DESIGN FILE *
 499 REM ***************
 500 IFIT$(1)=""THEN505
 501 PRINTCL$:PRINT" DESTROY PRESENT"F$"(Y/N)";
 502 GOSUB1990:IFT$<>"Y"THEN200
 503 POKE 49,PEEK(47):POKE 50,PEEK(48)
 505 PRINTCL$:PRINTRO$TAB(10)" DEFINE NEW"F$
 510 PRINT:PRINT" HOW MANY"I$"S (UP TO TEN) ";
 520 GOSUB4900:NO=VAL(TE$):IF NO<1 OR NO>10THEN 505
 523 IF NO>5 THEN RX=400
 525 DIM D$(RX,NO)
 530 PRINT:PRINT"ENTER"I$" NAMES"
 580 FOR I=1TONO
 590 PRINTI;:GOSUB4900:IT$(I)=TE$
 595 IF IT$(I)=""THEN590
 600 NEXTI
 610 GOSUB2000 :REM CHANGE ANYTHING?
 620 IF T$<>"Y"THEN 200
 640 GOSUB2100 : REM EDIT PROMPT
 650 PRINT"ENTER NEW"I$" NAME"
 660 PRINT NU;:GOSUB4900:IT$(NU)=TE$
 670 PRINTCL$TAB(9)RO$"DEFINE NEW"F$:PRINT:PRINT
 672 PRINTRO$I$" NAMES":PRINT:PRINT
 675 FOR J=1TONO:GOSUB3600:PRINT:NEXT
 680 GOTO610
 847 REM ************
 848 REM * ADD DATA *
 849 REM ************
 850 IF IT$(1)=""THEN4950
```

```
 855 PRINTCL$:PRINT:PRINTTAB(12)"PLEASE WAIT"
 857 P=600:IFNO>5 THEN P=400
 858 P=P*NO*3+NO*5*3+500
 860 GOSUB1800:RE=I:I=FRE(0):K=0
 862 PRINT CL$TAB(12);RO$"ADDING DATA"
 864 IF I<10 THEN PRINT"NO SPACE AVAILABLE":GOTO 210
 866 I=I-K:K=0
 867 PRINTSPC(11)INT((1-I/(30850-P))*100)" % USED":PRINT
 880 FORJ=1TONO
 890 GOSUB3600:GOSUB4900 :REM PROMPT & GET INPUT
 905 D$(RE,J)=TE$
 906 IFLEFT$(D$(RE,J),1)<>" "THEN910
 907 D$(RE,J)=RIGHT$(D$(RE,J),LEN(D$(RE,J))-1):GOTO906
 910 NEXTJ
 920 PRINT:PRINT"C-CHANGE ANYTHING":PRINT"A-ADD ANOTHER
"R$
 921 PRINT"ANY OTHER KEY - TO EXIT"
 923 GOSUB1995
 925 IFT$="A"THEN995
 928 IFT$<>"C"THEN200
 930 GOSUB2100 :REM EDIT PROMPT
 940 PRINTNU;RO$IT$(NU)OF$":";:GOSUB4900:D$(RE,NU)=TE$
 942 PRINTCL$R$"# ";RE:PRINT
 943 FORK=1TONO:PRINTK;RO$IT$(K)OF$;":";D$(RE,K):NEXT
 944 GOTO920
 995 FORP=1TONO:K=LEN(D$(RE,P))+K:NEXT
 996 RE=RE+1:IFRE<=RXTHEN862
 997 PRINT:PRINT"NO SPACE AVAILABLE":GOTO 210
1097 REM **********
1098 REM * SEARCH *
1099 REM **********
1000 PRINTCL$TAB(10)RO$SE$" OPTION":PRINT
1101 IF IT$(1)="" THEN PRINT"NO FILE PRESENT":FOR I=1 TO
300:NEXT:GOTO200
1102 PRINT"S - "SE$F$" FOR SPECIFIC"I$
1103 PRINT"R - REVIEW"F$" SEQUENTIALY":PRINT
1104 PRINT"ENTER OPTION";:GOSUB1990
1105 IF T$="R"THENRE=1:P=1:GOSUB1800:MX=I:GOTO1317
1107 P=0
1108 PRINTCL$RO$TAB(13)SE$I$"S"
1109 IFIT$(1)=""THEN4950
1110 PRINT"ENTER "SE$" VALUES"
1120 FORJ=1TONO
1130 PRINT:GOSUB3600:GOSUB4900
1177 S$(J)=TE$
1178 IF LEFT$(S$(J),1)=" " THEN 1180
1179 S$(J)=RIGHT$(S$(J),LEN(S$(J))-1:TOTO 1178
```

```
1180 NEXTJ
1190 GOSUB2000:IFT$<>"Y"THEN1250
1200 GOSUB2100
1210 PRINT:PRINT NU;" "RO$IT$(NU)OF$" ";
1212 GOSUB4900:S$(NU)=TE$
1215 PRINTCL$TAB(13)RO$SE$I$"S"
1220 FORI=1TONO:PRINTI;" "RO$IT$(I)OF$" "S$(I):NEXTI
1222 GOTO 1190
1250 GOSUB1800:MX=I:REM FIND LAST RECORD
1252 RE=1
1254 PRINTCL$TAB(9)"S E A R C H I N G"
1255 IF RE>MX-1THEN1365
1256 PRINT"*";
1260 FORJ=1TONO
1262 R=0 :REM SEARCH TYPE
1265 S$=S$(J):D$=D$(RE,J):L=LEN(S$)
1268 IFS$=""THEN1310
1269 IFLEFT$(S$,1)="/"THENS$=RIGHT$(S$,L-1):R=1:L=L-1
1270 IFLEFT$(S$,1)=">"THENS$=RIGHT$(S$,L-1):R=3:L=L-1
1271 IFLEFT$(S$,1)="<"THENS$=RIGHT$(S$,L-1):R=2:L=L-1
1272 IFLEFT$(S$,2)<>".."THEN1276
1273 IFRIGHT$(S$,L-2)<>RIGHT$(D$,L-2)ANDR=0THEN1360
1274 IFRIGHT$(S$,L-2)=RIGHT$(D$,L-2)ANDR=1THEN1360
1275 GOTO1310
1276 IFRIGHT$(S$,2)<>".."THEN1282
1278 IFLEFT$(S$,L-2)<>LEFT$(D$,L-2)ANDR=0THEN1360
1279 IFLEFT$(S$,L-2)=LEFT$(D$,L-2)ANDR=1THEN1360
1280 GOTO1310
1282 IF(S$<>D$)AND(R=0)THEN1360
1284 IF (S$=D$)AND(R=1)THEN1360
1286 IFR=3ANDD$<=S$THEN1360
1288 IFR=2ANDD$>=S$THEN1360
1310 NEXTJ
1317 IF D$(RE,1)="?"THEN RE=RE+1:GOTO 1254
1318 PRINTCL$R$"# ";RE:PRINT
1320 FOR K=1 TO NO:PRINTK;RO$IT$(K)OF$":"D$(RE,K):NEXT K
1324 PRINT:PRINTRO$"SELECT OPTION"
1325 IF P=0THENPRINT"S-CONTINUE TO "SE$
1326 PRINT "C-CHANGE SOMETHING":PRINT"P-PRINT THIS "R$
1327 PRINT"U-UP ONE "R$:PRINT"B-BACK ONE "R$
1328 PRINT"D-DELETE THIS "R$:PRINT"ANY OTHER KEY - TO
EXIT"
1329 GOSUB1995
1330 IF T$="S" AND P=0THEN PRINTCL$;"S E A R C H I N G":
GOTO1360
1331 IF T$="U" THEN RE=RE+1:GOSUB1372:IF RE<MX THEN1317
1332 IF T$="B" THEN RE=RE-1:GOSUB1374:IF RE>0 THEN 1317
```

```
1335 IF T$="P" THEN GOSUB1380:GOTO1317
1336 IFT$<>"C"THEN1339
1337 GOSUB2100:PRINTNU;RO$IT$(NU)OF$":";
1338 J=NU:GOSUB4900:D$(RE,NU)=TE$:GOTO1317
1339 IF T$<>"D" THEN 1342
1340 PRINT"ARE YOU SURE(Y/N)";
1341 GOSUB1990:IF T$="Y"THEN D$(RE,1)="?
1342 FOR K=1TO10:S$(K)="":NEXT K
1344 GOTO200
1360 RE=RE+1:GOTO1255
1365 GOTO200
1372 IF D$(RE,1)="?"THEN RE=RE+1:RETURN
1373 RETURN
1374 IF D$(RE,1)="?"THEN RE=RE=1:RETURN
1375 RETURN
1380 OPEN4,4:FOR K=1TO NO:PRINT#4,IT$(K)":"D$(RE,K)
:NEXT:PRINT#4,
1382 CLOSE 4:RETURN
1397 REM ***********
1398 REM * TOTAL UP*
1399 REM ***********
1400 PRINTCL$:PRINT"OPTION"
1401 PRINT" 1) TOTAL":PRINT" 2) RETURN TO MAIN MENU"
1402 PRINT:PRINT"ENTER OPTION";:GOSUB1990:IF T$<>"1" THEN
200
1403 PRINTCL$TAB(10)"TOTAL WHICH FIELD":PRINT:PRINT:PRINT
1405 FORK=1 TO NO:PRINTK;RO$IT$(K):NEXT:IF IT$(1)=""THEN
200
1410 PRINT:PRINT:PRINT"SELECT ";:GOSUB4900:NU=VAL(TE$)
1420 GOSUB 1800:MX=I-1:J=0
1425 FORK=1TOMX
1426 IFLEFT$(D$(K,NU),1)<>"$"THEN1428
1427 J=J+VAL (RIGHT$(D$(K,NU),LEN(D$(K,NU))-1)):GOTO1429
1428 J=J+VAL (D$(K,NU))
1429 NEXT K
1430 PRINT:PRINT"THE TOTAL IS ";J:PRINT:PRINT
1432 PRINTTAB(13)"<HIT ANY KEY>"
1435 GOSUB 1995:GOTO 200
1597 REM *********
1598 REM * PRINT *
1599 REM *********
1600 PRINTCL$RO$TAB(14)"PRINT "F$
1601 PRINT "1) PRINT":PRINT"2) RETURN TO MAIN MENU"
1602 PRINT:PRINT"ENTER SELECTION";:GOSUB1990:IF
T$<>"1"THEN 200
1603 IF IT$(1)="" THEN 4950
1605 PRINT:PRINTTAB(7)"ENTER CODE FOR EACH"I$:PRINT
```

```
1610 PRINT"<SPACE> DON'T PRINT"
1615 PRINT"X  FOLLOW"I$" WITH A RETURN"
1620 PRINT"+  NO RETURN"
1622 OPEN4,4
1625 FOR J=1 TO NO
1630 GOSUB3600
1635 GOSUB1990
1640 IF T$<>" "ANDT$<>"+"ANDT$<>"X"THEN1635
1645 P$(J)=T$:PRINT T$
1650 NEXT
1655 PRINTCL$RO$TAB(13)"PRINT SPECS"
1660 PRINTR$"S PER PAGE ":GOSUB4900
1665 R=VAL(TE$):IF R<1 OR R>15 THEN 1655
1670 PRINT:PRINT"PRINT"I$" NAMES<Y/N> ";
1675 GOSUB1990
1680 IF T$<>"Y" AND T$<>"N" THEN1675
1685 PRINTT$:TE$=T$
1686 PRINTCL$:PRINTTAB(9)"SET UP PRINTER"
1687 PRINT:PRINT:PRINTTAB(8)"THEN HIT ANY KEY"
1688 GOSUB1995:GOSUB 1800
1690 RE=1
1695 GOSUB1750
1700 FOR K=CT TO 66/R:PRINT#4,:NEXT K
1702 IF CT=R THEN CT=0
1710 RE=RE+1:IF RE<I THEN 1695
1715 CLOSE4
1720 FOR K=1TO10:P$(K)="":NEXTK:GOTO200
1750 FOR K=1TO NO
1755 IF TE$="Y" THEN 1775
1758 IF P$(K) =" "THEN 1790
1760 PRINT#4," ";D$(RE,K)"  ";
1765 IF P$(K)="X" THEN PRINT#4,:CT=CT+1
1770 GOTO1790
1775 IF P$(K)=" "THEN 1790
1780 PRINT#4,IT$(K)":"D$(RE,K)"  ";
1785 IF P$(K)="X" THEN PRINT#4,:CT=CT+1
1790 NEXTK
1795 RETURN
1799 REM ** COUNT NUMBER OF RECORDS **
1800 I=INT((1+RX)/2)
1801 FOR K=1TONO
1802 IF D$(I,K)<>"" THEN1805
1803 NEXT K
1804 J=1:GOTO1806
1805 J=1
1806 FOR I=J TO RX
1808 IF D$(I,1)=""THEN1830
```

```
1810 NEXTI
1830 IF NO=1 THEN1860
1832 FOR K-2TO NO
1835 IF D$(I,I)<>""THEN1850
1840 NEXTK
1845 GOTO1860
1850 J=I+1:GOTO1806
1860 RETURN
1990 GETT$:GOSUB5000:IFT$=""THEN1990
1994 RETURN
1995 GETT$:IF T$=""THEN1995
1997 RETURN
1999 REM ** CHANGE ANYTHING? **
2000 PRINT"CHANGE ANYTHING (Y/N)";
2010 GOSUB1990:PRINT" ":RETURN
2099 REM ** EDIT PROMPT **
2100 PRINT:PRINT"ENTER"I$" NUMBER TO CHANGE ";:GOSUB4900
2110 NU=VAL (TE$):IF NU<10RNU>NO THEN 2100
2115 RETURN
2347 REM ******************
2348 REM * INSERTION SORT *
2349 REM ******************
2350 PRINTCL$
2352 IF IT$(1)="" THEN4950
2353 PRINT"OPTIONS":PRINT"1) SORT":PRINT"2) RETURN TO
MAIN MENU"
2354 PRINT:PRINT"SELECT OPTION";:GOSUB 1990:IF T$<>"1"
THEN 200
2355 PRINTCL$"SELECT"I$" NUMBER TO SORT BY"
2360 FOR J=1TO NO:GOSUB3600:PRINT:NEXT
2370 PRINT:PRINT"ENTER"I$" NUMBER ";
2380 GOSUB4900:NU=VAL(TE$):IF NU<10RNU>NO THEN2370
2400 PRINTCL$:PRINTTAB(12)"S O R T I N G"
2405 GOSUB1800:MX=I-1
2406 SF=NU
2410 FOR M=1 TO MX-1
2420 J=M
2430 FOR P=1 TO NO:B$(P)=D$(M+1,P):NEXT P
2435 PRINT"*";
2440 IF B$(SF)>D$(J,SF) THEN 2480
2450 FOR P=1 TO NO:D$(J+1,P)=D$(J,P):NEXT P
2460 J=J-1
2470 IF J>=1THEN2440
2480 FOR P=1 TO NO:D$(J+1,P)=B$(P):NEXT P
2490 NEXTM
2495 GOTO200
```

```
2497 REM *************
2498 REM * LOAD FILE *
2499 REM *************
2500 GOSUB3300:ON VAL(T$)GOTO2580,200
2580 GOSUB3650
2600 PRINTCL$:PRINTTAB(12)"L O A D I N G"
2605 OPEN15,8,15
2610 OPEN2,8,2,G$+".DATA,S,R"
2620 INPUT#15,EC,ED$
2624 IF EC<>62 THEN2630
2625 PRINTCL$:PRINTTAB(5)F$" IS NOT ON THIS DISK"
2626 CLOSE15:CLOSE2:FOR K=1 TO 350:NEXT K:GOTO 200
2630 INPUT#2,NO
2635 POKE 49,PEEK(47):POKE 50,PEEK(48)
2640 FOR J=1 TO NO
2650 INPUT#2,IT$(J)
2660 NEXTJ
2663 IF NO>5 THEN RX=400
2665 DIM D$(RX,NO)
2670 FOR K=1 TO 999
2680 FOR N=1 TO NO-1
2690 INPUT#2,D$(K,N)
2692 IF D$(K,N)="?"THEN D$(K,N)="":GOTO 2900
2695 PRINT"*";
2700 NEXTN
2702 GET#2,T$:IF T$<>CHR$(13)THEN
D$(K,N)=D$(K,N)+T$:GOTO2702
2710 NEXTK
2900 CLOSE2:CLOSE 15
2910 GOTO200
2997 REM *************
2998 REM * SAVE FILE *
2999 REM *************
3000 IF IT$(1)=""THEN4950
3005 GOSUB3300:ON VAL(T$)GOTO3100,200
3100 OPEN15,8,15
3103 GOSUB 3650:REM GET FILE NAME
3105 PRINTCL$TAB(13)"S A V I N G"
3107 PRINT#15,"I"
3110 OPEN2,8,2,"@0:"+G$+".DATA,S,W"
3115 GOSUB1800:MX=I-1
3120 INPUT#15,EC,ED$
3130 PRINT#2,NO
3140 FOR J=1 TO NO
3150 PRINT#2,IT$(J)
3160 NEXT J
```

```
3165 J=Ø
3170 J=J+1
3175 IF D$(J,1)="?" THEN3170
3180 FOR K=1 TO NO
3185 IF D$(J,K)="" THEN D$(J,K)=" "
3190 PRINT#2,D$(J,K)
3195 PRINT"*";
3200 NEXTK
3210 IF J<MX THEN3170
3220 PRINT#2,"?"
3270 CLOSE2:CLOSE 15
3280 GOTO200
3300 PRINTCL$:PRINT"OPTIONS"
3310 PRINT:PRINT"1) USE DISK"
3320 PRINT:PRINT"2) RETURN TO MAIN MENU"
3325 PRINT:PRINT"ENTER SELECTION ";
3340 GOSUB1990
3350 IF T$<>"1"AND T$<>"2" THEN3340
3355 PRINTT$
3356 PRINT
3360 RETURN
3599 REM ** PROMPT LINE **
3600 PRINTJ;RO$IT$(J)OF$":";:RETURN
3605 IF G$="" THEN3600
3610 RETURN
3650 PRINT "ENTER"F$" NAME ";:GOSUB4900:G$=TE$:RETURN
4899 REM ** INPUT ROUTINE **
4900 TE$=""
4901 GET T$:GOSUB5000:IFT$=""THEN4901
4902 IF T$=CHR$(13)THEN PRINT" ":RETURN
4903 IF T$=CHR$(20)THEN 4915
4904 IF(LEN(TE$)>25 AND J<NO)OR(LEN(TE$)>254 AND
J=NO)THEN4901
4905 IF T$<CHR$(32) OR T$>CHR$(100) THEN 4901
4906 PRINTT$;:TE$=TE$+T$:GOTO 4901
4915 IFLEN(TE$)<=Ø THEN 4917
4916 TE$=MID$(TE$,1,(LEN(TE$)-1)):PRINT" "BS$BS$" "BS$
4917 GOTO4901
4950 PRINT CL$"NO"F$" DESIGN":FOR KK=1TO350:NEXT:GOTO200
4999 REM ** CURSOR ROUTINE **
5000 KT=KT+1
5002 IF KT<>7THEN RETURN
5004 KT=Ø:FL=FL*-1
5006 IF FL=1 THEN PRINT: "BS$;
5008 IF FL=-1THEN PRINTRO$" "OF$BS$;
5010 RETURN
```

a last word—structured programming

The concept of structured programming has not been mentioned so far in this book. Structured programming is a programming style that came into vogue in the early 1970s. The idea is that a program should logically flow from top to bottom in an organized fashion that is efficient and easy to read, modify, and debug. The more modern programming languages, such as Pascal and C, embody structured programming as a building block of the language, replacing the use of GOTOs, with new program structures. Because BASIC is an older language, more effort is needed to write structured programs.

All of the examples in this book were written in good, structured code. I felt it was better to present this topic in a subtle form rather than confusing the reader with these concepts before he is ready for them. All the programs flow smoothly from top to bottom, with each section of the program doing a single task, and are highlighted with comments (REMs). The use of psuedocode is another structured programming technique. None of the programs jump around from top to bottom to middle to perform a single task. All programs start with an intialization section and then usually a menu branches off to different program sections. All subroutines follow at the end. If you have followed the examples, you are now on your way to good programming style.

appendix a
simons' BASIC

Compared to other microcomputer BASICs, Commodore BASIC is deficient in several respects. A sixteen-year-old student from England, David Simons, decided to do something about the situation. The fruit of his labor is Simons' BASIC, a ROM cartridge that adds 114 commands to the Commodore 64's BASIC repertoire.

The ROM cartridge is one of the best investments you can make for your Commodore 64. The cartridge slides into the expansion port in the rear of the computer (on the side of the power on LED). Like any other ROM cartridge or piece of circuitry, it must NEVER be inserted or removed when the computer is on. Doing so will probably destroy the chips inside the cartridge. When the computer (with the cartridge in) is turned on, the screen appears with a white backround, blue border, and black letters (on a color monitor, of course), rather than with the standard colors. The system message indicates that 30719 bytes are available. The extra commands need 8192 bytes of memory to work. Programming can then be done using both standard BASIC and the Simons' BASIC extensions. Many of the new instructions were possible previously by using POKEs. Others are totally new.

The 114 commands can be grouped into 7 categories:

1. Programming aids–commands that aid in programming and debugging
2. Additional programming structures–several new loops and statements are included
3. Error trapping–The ability to check for errors before an error message appears
4. Screen I/O commands–Instructions for input validation and output.
5. Extra numeric and string operations
6. Music–Commands to harness the polyphonic music capability
7. Graphics Plotting and sprite commands

Since music and graphics are not part of the scope of this book, they will not be discussed. Some of the extensions are covered in those chapters where they would logically fall.

Simons' BASIC is excellently conceived and written but some drawbacks do exist.

1. Over 8K of memory is lost. Simons' BASIC is impractical for programs like MAGICFILER that need the memory for data storage.
2. Some machine language programs will not load correctly when the Simons' BASIC cartridge is used. The DOS wedge still operates properly.
3. No instructions to aid in file operations are included.
4. A program using Simons' BASIC instructions must be run on a Commodore 64 with a Simons' BASIC cartridge; Therefore, it is not practical to write commercial software using Simons' BASIC until every Commodore user has one.

programming aids

Programming aids are a set of direct commands that help in typing and debugging programs.

AUTO start,increment

AUTO is a direct command which prompts BASIC line numbers for programming. This eliminates the task of typing a line number as you enter program lines.

```
AUTO 10,5
```

Prompts line number starting with 10 and then in increments of 5 (15,20,25,etc.). A carriage return stops the prompts.

DELAY n

One of the most annoying features of the Commodore 64 is the speed with which a program is listed to the screen and the inability to start and stop the listing. This makes reading a program as it is listed difficult. DELAY offers a way to slow down the speed of the listing. The speed is determined by the value of N from 1 to 255. The shift key must be held down for DELAY to work.

In reality, this is not as useful as it sounds. DELAY controls the speed with which characters are sent to the screen rather than whole lines. Most values of N list a program extremely slowly. DELAY 30 provides a reasonable speed to read a program, but I have found the old-fashioned method of holding down the CTRL key more useful.

DUMP

DUMP is a direct command that produces a list of program variables and their values. This is an excellent debugging tool. A DUMP can be sent to the printer by doing a CMD 4. Unfortunately, DUMP does not function properly if a program contains an array.

COLD

COLD reintitializes Simons' BASIC, erasing both programs and data. The initial message is displayed. A program can be recalled with the OLD command, but data is lost forever.

FINDstring

FIND is probably the most useful of the programming aids. FIND searches for occurrences of the specified string in a program and displays the line numbers of the occurrences.

```
10 X$="ABCD"
20 A$="1"
30 Y=12
40 Z=122
```

The following examples of FIND use the program listed above.

The search **FIND1** would return **30,40**

the search **FIND"1"** would return **20**

the search **FINDA** would return **20**

Strings in quotes are treated differently from characters not in quotes. When looking for strings, the whole string must be specified including the quotes. If line 10 reads

10 X$=ABCD (which of course is an error)

the result of **FINDA** would be **10,20**

FIND is very useful for locating lines that are referenced by other lines. FIND1200 would find every line that branches to line 1200. When compacting a program by combining lines together, use FIND in this manner to check if any removed lines are referenced.

MERGE program name,device#

MERGE is a very useful command that acts strangely in some circumstances. MERGE allows different program files to be combined into one program in memory. Frequently used subroutines can be kept in a file and then MERGE can be used to add them to a program. If the following program were in memory

```
10 X=4
20 Y=3
```

and this program was in a diskette file named T,

```
30 Z=5
40 W=4
```

then MERGE"T",8 would combine the two programs together.

```
10 X=4
20 Y=3
30 Z=5
40 W=4
```

The file being MERGEd cannot be interlaced with the existing program in memory. The line numbers in the file must be greater than those used in the program in memory. For example if the program in memory were

```
50 X=10
60 Y=3
```

MERGE"T",8 would result in

```
50 X=10
60 Y=3
30 Z=5
40 W=4
```

This seems like a contradiction, but nevertheless this is what happens. The statement GOTO 30, would produce an error message.

OLD

OLD is a handy instruction which undoes the effect of a NEW statement, thus restoring a program. It works until any new program lines are entered.

PRINT$ and PRINT%

PRINT$ will convert a hexadecimal number to decimal and print the result. PRINT% will convert a binary number to decimal and print the result. These two instructions have been grouped together with the programming aids since these conversions are often made when machine language programming is done from BASIC with POKEs (machine language uses hexadecimal values but POKEs need a decimal value). For some reason PRINT$ needs at least 4 hexadecimal digits. PRINT% needs at least 8 binary digits (bits). A number can be padded with leading zeros to fit the required number of digits.

```
PRINT% 10001010
```

returns **138**

```
PRINT$ A1BC
```

returns **41404**

PAGE n

The PAGE command controls the listing of a program by breaking it up in groups of N lines. After a PAGE command is done, LIST will display the number of the first line of the program. Hitting the return key displays the first N lines, then the next N lines, and so on. The listing cannot be re-directed to the printer.

TRACE n

TRACE is a powerful debugging tool. When the TRACE 10 command is given in direct mode, a light-colored window appears in the upper right hand corner of the screen. When a program is run, the line numbers of statements that have executed scroll through the window. The window is large enough to display the line numbers of the last six statements executed. This is a fantastic debugging tool, allowing the programmer to follow along with the program and knowing exactly which lines have executed. Since the line numbers fly by quickly, execution can be slowed by holding down the Commodore key as the program runs. When execution is finished the window disappears. TRACE 0 turns off the TRACE function. It should be noted that the window will erase any output printed to the same portion of the screen.

RETRACE

As noted, the TRACE window disappears when the program finishes. RETRACE redisplays the window, indicating the last six program lines executed.

screen i/o

Simons' BASIC has several new screen input and output statements in its repertoire. Several of these statements (PRINT AT, USE, FETCH AND INKEY) are covered in Chapter 2. Others are discussed below.

ON KEY

ON KEY is a combination of a GET and IF statement that is useful in menu applications. ON KEY looks at the first character in the keyboard buffer. It then executes a statement depending upon whether the character is in a given list of characters held in a string. The status variable ST contains the ASCII code of the character entered.

```
100 A$="ABCD"
110 ON KEY A$,: GOTO 200
120 GOTO 110
```

replaces

```
100 GET A$:IF A$ ="" THEN 100
110 IF A$>="A" OR A$<="D" THEN 200
120 GOTO 100
```

Note that in the first example, the ON KEY statement is in a loop. The GOTO 200 is executed when a character in the list is entered.

RESUME

RESUME causes the last ON KEY statement to be re-executed.

DISABLE

DISABLE must be used after using an ON KEY statement. DISABLE shuts off the keyboard buffer scan operation.

FLASH, color,speed

FLASH is used to flash all characters of the same color that appear on the screen. Color is the standard 0 to 15 color code. Speed is a value of 1 to 255, with every increment of the value increasing the speed by 1/60 of a second. FLASH is useful for highlighting important messages on the screen.

OFF

OFF is used to cancel the FLASH command.

BFLASH speed,color1,color2

BFLASH continuously switches the color of the screen border between the colors specified. The speed is a value of 1 to 255. In real terms the colors are switched speed*1/60 times per second. BFLASH 0 stops the sequence.

INV row,column,width,length

INV switches any characters in the defined area into reverse video. The defined area is a rectangle with its upper left-hand corner at row, column. Its dimensions are width and length. To invert the characters in the top half of the screen: INV 0,0,40,12.

HRDCPY

HRDCPY copies the exact contents of the screen to the printer. This cannot be used if high resolution graphics are displayed. The command automatically

takes care of opening and closing the printer file. Many programs offer an option to print a copy of the information on the screen. This can easily be implemented by using HRDCPY in the program.

programming statements and structures

Simons' BASIC includes several new BASIC statements and programming structures which are valuable additions to the Commodore BASIC instruction set.

PAUSE "message",s

PAUSE prints the message and waits S seconds before proceeding to the next statement. PAUSE can be used with or without the message. This statement is useful when displaying a message before moving on to a different screen or menu. In Commodore BASIC, a delay in a program can also be accomplished by using a empty FOR loop, for example FOR I=1 TO 300 : NEXT I

CGOTO expression

CGOTO evaluates the expression and uses the answer as a line number to branch to. CGOTO can be used to eliminate the need for many IF statements.

```
100 IF I=1 THEN 200
110 IF I=2 THEN 300
120 IF I=3 THEN 400
```

can be replaced by

```
100 CGOTO (I+1)*100
```

CGOTO is only useful when some relationship exists between the line numbers being branched to and the possible values of the expression. Internally, CGOTO works by placing the value of the expression into the current basic line number (held in locations 58 and 59).

RESET linenumber

RESET is an extension of the Commodore BASIC RESTORE statement. RESTORE resets the data line pointer to the first DATA statement. RESET resets the data line pointer to the DATA statement whose line number is given in the instruction. Internally it works by replacing the current data line number in locations 63 and 64.

In the following example the second DATA statememt is read twice

```
10 READ A
20 READ B
```

```
 30 RESET 110
 40 READ C
 .
 .
110 DATA 10
120 DATA 20
```

The variables B and C will have the same value: 20.

LOCAL variable list/GLOBAL

LOCAL and GLOBAL are used to define sections of a program where the scope of variables is limited. This means that in this section of the program, the value of the designated variables are limited to that program section; outside that section its value is different. LOCAL stores the values of the variables listed and frees them for new values. GLOBAL restores the original values, erasing the temporary values in the process. Typically, this technique is used in subroutines and lets fewer variables be used in a program, saving memory and confusion.

```
 10 I=10
 20 K=20
 30 J=30
 40 PRINT "I="I,"K="K,"J="J
 50 GOSUB 100
 60 PRINT "I="I,"K="K,"J="J
 70 STOP
 99 REM SUBROUTINE
100 LOCAL I,K
110 I=-15
120 K=-17
130 PRINT "I="I,"K="K,"J="J
140 GLOBAL
150 RETURN
```

The output of the preceding example is

```
I= 10   K= 20   J= 30
I=-15   K=-17   J= 30
I= 10   K= 20   J= 30
```

error trapping

Simons' BASIC provides a mechanism to prevent a program from crashing when it hits an error. When error trapping, the Commodore's error routines are shut off and control is passed to an error routine in the program. After appropriate action is taken to handle the error, the program can continue.

The statements used are:

ON ERROR GOTO linenumber-Turns on error trapping. When an error occurs, the program's error routine is branched to via the GOTO statement. The syntax of the statement must be exact.

NO ERROR-Turns off the Commodore's error routine and stops the system error message from being displayed. This must be the first line in an error routine.

OUT-Disables the ON ERROR statement.

When using ON ERROR two new systems variables are used:

ERRLN-Holds the line number of the error.

ERRN-Holds the number of the error. The error numbers are in the following table:

1	Too many files
2	File open
3	File not open
4	File not found
5	Device not present
10	Next without for
11	Syntax error
12	Return without gosub
13	Out of data
14	Illegal quantity
15	Overflow
16	Out of memory
17	Undefined statement
18	Bad subscript
19	Re-dimensioned array
20	Division by zero
21	Illegal direct
22	Type mismatch
23	String too long

```
  5 ON ERROR: GOTO 100
 10 INPUT "ENTER VALUE";N
 20 P=10/N
 30 PRINT P
 40 STOP
 99 REM ERROR ROUTINE
100 NO ERROR
110 PRINT "CAN NOT DIVIDE BY ZERO"
120 GOTO 5
```

In the example above, if a zero is entered a division by zero error would nor-

mally occur. If the error occurs, control is passed to the error routine at line 100 by the ON ERROR statement at line 5. In the program's error routine the Commodore's error routine is shut off by the NO ERROR statement, an error message is printed and control is passed back to the INPUT statement. Without error trapping, the Commodore's message would have been printed and the program would have stopped.

notes on error trapping with simons' BASIC

1. Many of the errors that can be trapped by this technique can be either detected during debugging (syntax, udefined statement, etc.) or by the error channel when using files (file already open, file not on disk, etc.). This technique is most useful for checking user input (divide by zero, overflow) or for checking for rare occurrences in certain programs (bad subscript, out of memory).
2. The NO ERROR statement must be the first line in an error routine.
3. Error trapping is very tricky to use and the slightest mistake will either make the program hang or make it uneditable. Practice the technique until you get it right before using it in a program.

appendix b magicfiler user's guide

MAGICFILER is a powerful, easy to use database program that compares in features and power to many other programs being sold for the Commodore 64. It is totally menu-driven, meaning that no complicated instruction sequences have to be memorized. With MAGICFILER, rather than spending time organizing information, you spend time using information. This User's Guide is meant as a quick reference for MAGICFILER. The complete program is listed and explained in Chapter 8.

design

To use MAGICFILER you must first analyze the information you wish to store to identify what makes up a record. A record is a single set of information. Take for example doctors' bills. Each bill is a separate record that contains several items of information such as the doctor's name, date, fee, and diagnosis. Several records together form a file. Data is stored by MAGICFILER in this order. To design a file, use the design option from the main menu and then indicate how many items there are per record and the item names.

example

NUMBER OF ITEMS? 3 S/B 4

ENTER ITEM NAMES

1. DOCTOR
2. FEE
3. DATE
4. DIAGNOSIS

Each item declared can hold up to 25 characters execpt for the last item which can hold 255. This makes the last item in a record perfect for holding notes.

ADD DATA

ADD DATA is used to place records into memory. Whenever the ADD DATA option is selected, records entered are appended to the end of the file currently in memory. While entering items, the delete key can be used to correct data entry. After all the items in a record are entered, you may edit any of the items. While entering records, the percentage of the file that is filled is displayed. It is a good idea to stop entering records once a file reaches 90%.

SEARCH FILE

SEARCH FILE is used to examine the records stored in memory. There are two ways to examine the file: review mode, which displays the records sequentially; and search mode, which allows you to enter items you wish to match in the file. The mode is selected in an initial menu. After a record is displayed there are several options available

Go to the next record
Go back to the previous record
Print this record
Delete this record
Escape to the main menu
Continue searching (search mode only)

In search mode there are several ways to match items
ABC finds items with ABC
A.. finds items starting with A
..C finds items ending with C
/AB finds items NOT having AB
>10 finds items greater than 10
<10 finds items less than 10

The NOT operation can be combined with any other match. If more than one item is to be matched, all the specified matches must be made for a record to be found.

The search

1. DOCTOR:SMITH
2. FEE:
3. DATE:
4. DIAGNOSIS:

finds records where the doctor's name is Smith.

The search

1. DOCTOR:S..
2. FEE:
3. DATE:
4. DIAGNOSIS:

finds records where the doctor's name begins with an S.

The search

1. DOCTOR:
2. FEE:<$30.00
3. DATE:
4. DIAGNOSIS:

finds all records where the fee is less tham $30.00

The search

1. DOCTOR:SMITH
2. FEE:
3. DATE:..84
4. DIAGNOSIS:

finds all records where the doctor's name is Smith and the visit took place in 1984.

For SEARCH FILE to work properly with numerical figures and dates, the item to match must be in the same format as the items stored; that is, if dollar signs are used in the file, they must be used in the match item.

SORT FILE

SORT FILE is used to place the records in memory in sort order. The user specifies which items will be compared when placing the records in descending order. Only the file in memory is sorted; any files on diskette are not effected. If you SORT and then SAVE, the diskette file is changed.

PRINT FILE

PRINT FILE is used to make a hard copy of all the records in memory. If a copy of only a single record is needed, use the print option in SEARCH. The user has the following options for printing

How many records per page? This can be used to print specialized forms, such as mailing labels.

Print the item names along with the data?

How to print each item? A code is entered next to each item name. The codes follow:

X	follow this item with a return
+	follow this item with two spaces
space	Don't print this item at all

All these options are displayed during PRINT.

LOAD FILE

LOAD FILE is used to retrieve a previously stored file from a diskette and place it into memory. A prompt appears asking for the name of the file to load. Any file already in memory will be destroyed.

SAVE FILE

SAVE FILE will store the records in memory onto a diskette for future use. A file name is prompted. Any file already with that name on the diskette is replaced.

TOTAL

TOTAL goes through the file adding together the contents of the same item in each record. Only numerical values are added. In our doctor bill example, we could easily add together all the doctor's fees that were paid.

appendix c
dos commands and error messages

While not actually a part of BASIC, there is a special set of commands to be used with the disk drive. These commands are sent to the disk drive via the error channel and are interpreted by the Disk Operating System (DOS) that resides in ROM inside the drive. These DOS commands are normally done in direct mode, but can be and are used from within programs. A special set of error messages is associated with the DOS commands and BASIC instructions that access the disk drive. These error messages will be discussed later.

Since the DOS commands are sent via the error channel (channel 15), the channel must first be OPENed

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

Several of the commands require a drive number to identify on which disk drive a file is located. It is zero for a one-drive system. In a two-drive system, the first drive is numbered zero and the second drive is numbered one.

NEW

The NEW command is used to prepare a blank (unused) diskette for use in the disk drive. It is during this operation that a diskette is divided into tracks and sectors by the placing of special timing marks on the diskette's surface. The NEW operation takes several minutes, during which time the computer can be used but not the disk drive. The form of the NEW statement is

```
PRINT#15, "NEW Ø:DISKNAME,ID"
```

or

```
PRINT#15, "NØ:DISKNAME,ID"
```

Where
Diskname is a string of up to 16 characters used to identify the diskette, and
ID is a two-character code which should be unique on every diskette you own.

SCRATCH

SCRATCH is used to erase unwanted files on a diskette. Any type of file (program, sequential, random, relative) can be SCRATCHED. The form is

```
PRINT#15, "SCRATCH Ø:FILENAME"
```

or

```
PRINT#15, "SØ:FILENAME"
```

Many files with similar names can be SCRATCHed with a single SCRATCH command using a *wildcard.*

The command

```
PRINT#15, "SØ:FILE*"
```

will erase all files on a diskette that have a name starting with FILE. Be aware that the wildcard can only be used as a suffix. It cannot be used as a prefix—this would result in erasing all the files on a diskette.

COPY

The COPY command is used to make a second copy of a file on the same diskette. The form is

```
PRINT#15, "COPY Ø:BACKUP=Ø:ORIGINAL"
```

or

```
PRINT#15, "CØ:BACKUP=Ø:ORIGINAL"
```

There must be no previous file with the same name as the new file. The name of the existing file goes on the right of the equal sign. The name for the duplicate goes on the left side of the equal sign. COPY can also be used to combine up to four files together into one big file.

```
PRINT#15, "C0:BIGCOPY=0:FILE1,0:FILE2,0:FILE3"
```

INITIALIZE

INITIALIZE is used to tell the disk drive to read the diskette name and ID of a diskette in the drive. This should be done before placing any information onto a new diskette after the diskette in the drive has been switched. The form is

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

RENAME

RENAME is used to change the name of a file on a diskette. The form is

```
PRINT#15,"RENAME 0:NEWNAME=OLDNAME"
```

or

```
PRINT#15,"R0:NEWNAME=OLDNAME"
```

The file OLDNAME will now be named NEWNAME. If a file already called NEWNAME exists, an error message will appear on the error channel.

VALIDATE

After many programs have been SAVEd and SCRATCHed on a diskette, the directory of the diskette can become disorganized. This results in some of the free space on the diskette being unusable. The VALIDATE command recovers blocks that are scattered all over the diskette, allowing them to be used for storage. After the VALIDATE command is executed, you will notice more available blocks in the directory listing. VALIDATE takes several minutes to operate. The form is

```
PRINT#15,"VALIDATE"
```

or

```
PRINT#15,"V"
```

error messages

Here is a list of some of the DOS error messages that can occur from the use of DOS commands and BASIC statements that access files. These messages appear on the error channel and must be read with INPUT#15, EC, ED$, where EC is the error code, and ED$ is the error description.

0–No error. No error exists after performing a command. This is the message when a command has been executed *without a problem.*

1–File scratched. This is the response to confirm SCRATCHing a file. The error description will include how many files were erased.

26-Write protect on. The diskette in the drive has its write protect notch covered.

30-Syntax error in command.

31-Invalid command. The DOS does not recognize the command.

32-Syntax error. The command is longer than 58 characters.

34-No file given. No file is specified in the command.

50-Record not present in a relative file. Occurs when a record that is not present in a file is specified in a position command. This is not an error, but rather an advisory before reading or writing to a relative file. See the chapter on relative files for comments on the inaccuracy of this message.

51-Overflow in record. Occurs when there is an attempt to write more characters than defined to a record.

52-File too large. Record position in a relative file indicates that a disk overflow will occur.

60-File opened for write. A read has been attempted on a sequential file opened for write.

61-File not open. The file being accessed has not been opened.

62- File not found. The file specified in the OPEN statement is not on the diskette. See the chapters on file access for tips on the use of this message.

63-File exists. A file already exists with the same name as the one being created.

70-No channel available. The requested channel is not available or all chan nels are in use. This error sometimes occurs if a file is not CLOSEd before it is re-OPENed.

72-Disk full. The diskette or its directory is filled up.

appendix d table of programming techniques

input

sequential files

relative files

index sequential access method

appendix e
ascii codes

ASCII/Commodore Values
Uppercase/Graphics Mode

ASCII Value	Effect	Key(s)	Quote Mode Display
0		CTRL-@	
1		CTRL-A	
2		CTRL-B	
3		CTRL-C	
4		CTRL-D	
5	white foreground	CTRL-2 or CTRL-E	
6		CTRL-F	
7		CTRL-G	
8	disable SHIFT-C=	CTRL-H	
9	enable SHIFT-C=	CTRL-I	
10		CTRL-J	
11		CTRL-K	
12		CTRL-L	
13	start new line	RETURN or CTRL-M	
14	switch to lowercase	CTRL-N	
15		CTRL-O	
16		CTRL-P	
17	cursor down	CRSR↓↑ or CTRL-Q	
18	reverse-video on	CTRL-9 or CTRL-R	
19	home cursor	HOME or CTRL-S	
20	delete character	DEL or CTRL-T	
21		CTRL-U	
22		CTRL-V	
23		CTRL-W	
24		CTRL-X	
25		CTRL-Y	
26		CTRL-Z	
27		CTRL-[	
28	red foreground	CTRL-3 or CTRL-£	
29	cursor right	CRSR→← or CTRL-]	
30	green foreground	CTRL-6 or CTRL-↑	
31	blue foreground	CTRL-7 or CTRL-=	

ASCII Value	Character	Key(s)	ASCII Value	Character	Key(s)	ASCII Value	Character	Key(s)
32		space	64	@	@	96		
33	!	SHIFT-1	65	A	A	97		
34	"	SHIFT-2	66	B	B	98		
35	#	SHIFT-3	67	C	C	99		
36	$	SHIFT-4	68	D	D	100		
37	%	SHIFT-5	69	E	E	101		
38	&	SHIFT-6	70	F	F	102		
39	'	SHIFT-7	71	G	G	103		
40	(	SHIFT-8	72	H	H	104		
41	)	SHIFT-9	73	I	I	105		
42	*	*	74	J	J	106		
43	+	+	75	K	K	107		
44	,	,	76	L	L	108		
45	-	-	77	M	M	109		
46	.	.	78	N	N	110		
47	/	/	79	O	O	111		
48	0	0	80	P	P	112		
49	1	1	81	Q	Q	113		
50	2	2	82	R	R	114		
51	3	3	83	S	S	115		
52	4	4	84	T	T	116		
53	5	5	85	U	U	117		
54	6	6	86	V	V	118		
55	7	7	87	W	W	119		
56	8	8	88	X	X	120		
57	9	9	89	Y	Y	121		
58	:	:	90	Z	Z	122		
59	;	;	91	[	SHIFT-:	123		
60	<	SHIFT-,	92	£	£	124		
61	=	=	93	]	SHIFT-;	125		
62	>	SHIFT-.	94	↑	↑	126		
63	?	SHIFT-/	95	←	←	127		

ASCII Value	Effect	Key(s)	Quote Mode Display
128			
129	orange foreground	C= -1	
130			
131			
132			
133	as programmed	F1	
134	as programmed	F3	
135	as programmed	F5	
136	as programmed	F7	
137	as programmed	SHIFT-F1	
138	as programmed	SHIFT-F3	
139	as programmed	SHIFT-F5	
140	as programmed	SHIFT-F7	
141	start new line	SHIFT-RETURN	
142	switch to uppercase		
143			
144	black foreground	CTRL-1	
145	cursor up	SHIFT-CRSR↓↑	
146	reverse-video off	CTRL-0	
147	clear screen	SHIFT-HOME	
148	insert character	SHIFT-DEL	
149	brown foreground	C= -2	
150	light red foreground	C= -3	
151	gray 1 foreground	C= -4	
152	gray 2 foreground	C= -5	
153	lt. green foreground	C= -6	
154	lt. blue foreground	C= -7	
155	gray 3 foreground	C= -8	
156	purple foreground	CTRL-5	
157	cursor left	SHIFT-CRSR→←	
158	yellow foreground	CTRL-8	
159	cyan foreground	CTRL-4	

AV Sa Cl Iu Ie	Character	Key(s)
160		SHIFT-space
161		⌘-K
162		⌘-I
163		⌘-T
164		⌘-@
165		⌘-G
166		⌘-+
167		⌘-M
168		⌘-£
169		SHIFT-£
170		⌘-N
171		⌘-Q
172		⌘-D
173		⌘-Z
174		⌘-S
175		⌘-P
176		⌘-A
177		⌘-E
178		⌘-R
179		⌘-W
180		⌘-H
181		⌘-J
182		⌘-L
183		⌘-Y
184		⌘-U
185		⌘-O
186		SHIFT-@
187		⌘-F
188		⌘-C
189		⌘-X
190		⌘-V
191		⌘-B
192		SHIFT-*
193		SHIFT-A
194		SHIFT-B
195		SHIFT-C
196		SHIFT-D
197		SHIFT-E
198		SHIFT-F
199		SHIFT-G
200		SHIFT-H
201		SHIFT-I
202		SHIFT-J
203		SHIFT-K
204		SHIFT-L
205		SHIFT-M
206		SHIFT-N
207		SHIFT-O
208		SHIFT-P
209		SHIFT-Q
210		SHIFT-R
211		SHIFT-S
212		SHIFT-T
213		SHIFT-U
214		SHIFT-V
215		SHIFT-W
216		SHIFT-X
217		SHIFT-Y
218		SHIFT-Z
219		SHIFT-+
220		⌘-—
221		SHIFT-—
222		SHIFT-↑
223		⌘-*
224		
225		
225		

8/B 255

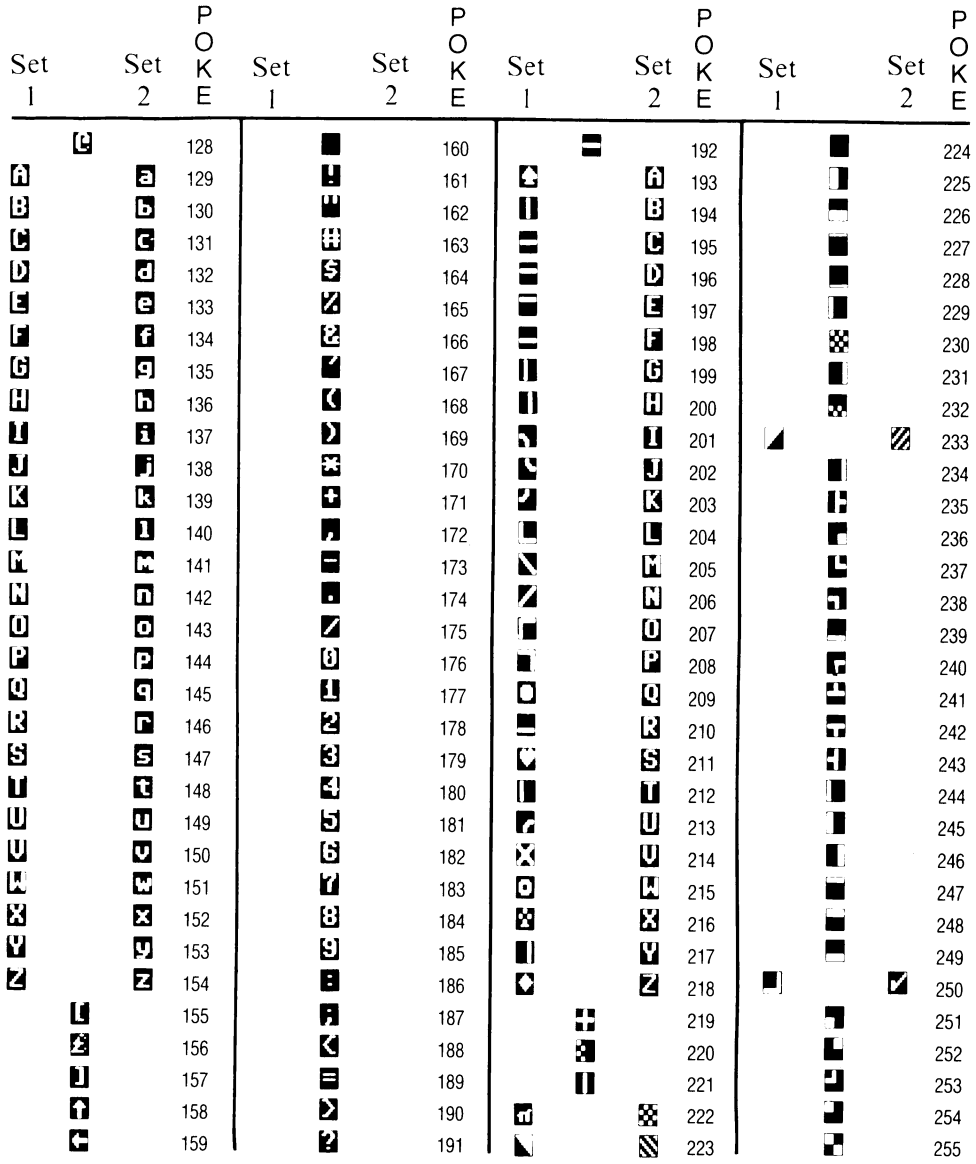
appendix f
screen codes

Set 1	Set 2	POKE
	@	0
A	a	1
B	b	2
C	c	3
D	d	4
E	e	5
F	f	6
G	g	7
H	h	8
I	i	9
J	j	10
K	k	11
L	l	12
M	m	13
N	n	14
O	o	15
P	p	16
Q	q	17
R	r	18
S	s	19
T	t	20
U	u	21
V	v	22
W	w	23
X	x	24
Y	y	25
Z	z	26
[		27
£		28
]		29
↑		30
←		31

Set 1	Set 2	POKE
		32
	!	33
	"	34
	#	35
	$	36
	%	37
	&	38
	'	39
	(	40
	)	41
	*	42
	+	43
	,	44
	-	45
	.	46
	/	47
	0	48
	1	49
	2	50
	3	51
	4	52
	5	53
	6	54
	7	55
	8	56
	9	57
	:	58
	;	59
	<	60
	=	61
	>	62
	?	63

Set 1	Set 2	POKE
		64
	A	65
	B	66
	C	67
	D	68
	E	69
	F	70
	G	71
	H	72
	I	73
	J	74
	K	75
	L	76
	M	77
	N	78
	O	79
	P	80
	Q	81
	R	82
	S	83
	T	84
	U	85
	V	86
	W	87
	X	88
	Y	89
	Z	90
	+	91
		92
		93
		94
		95

Set 1	Set 2	POKE
		96
		97
		98
		99
		100
		101
		102
		103
		104
		105
		106
		107
		108
		109
		110
		111
		112
		113
		114
		115
		116
		117
		118
		119
		120
		121
		122
		123
		124
		125
		126
		127


Set 1
Set 2
POKE
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255

appendix g binary-hex conversion table

BASE CONVERSION TABLE

BIN	*DEC*	*HEX*	*BIN*	*DEC*	*HEX*
0	0	00	110101	53	35
1	1	01	110110	54	36
10	2	02	110111	55	37
11	3	03	111000	56	38
100	4	04	111001	57	39
101	5	05	111010	58	3A
110	6	06	111011	59	3B
111	7	07	111100	60	3C
1000	8	08	111101	61	3D
1001	9	09	111110	62	3E
1010	10	0A	111111	63	3F
1011	11	0B	1000000	64	40
1100	12	0C	1000001	65	41
1101	13	0D	1000010	66	42
1110	14	0E	1000011	67	43
1111	15	0F	1000100	68	44
10000	16	10	1000101	69	45
10001	17	11	1000110	70	46
10010	18	12	1000111	71	47
10011	19	13	1001000	72	48
10100	20	14	1001001	73	49
10101	21	15	1001010	74	4A
10110	22	16	1001011	75	4B
10111	23	17	1001100	76	4C
11000	24	18	1001101	77	4D
11001	25	19	1001110	78	4E
11010	26	1A	1001111	79	4F
11011	27	1B	1010000	80	50
11100	28	1C	1010001	81	51
11101	29	1D	1010010	82	52
11110	30	1E	1010011	83	53
11111	31	1F	1010100	84	54
100000	32	20	1010101	85	55
100001	33	21	1010110	86	56
100010	34	22	1010111	87	57
100011	35	23	1011000	88	58
100100	36	24	1011001	89	59
100101	37	25	1011010	90	5A
100110	38	26	1011011	91	5B
100111	39	27	1011100	92	5C
101000	40	28	1011101	93	5D
101001	41	29	1011110	94	5E
101010	42	2A	1011111	95	5F
101011	43	2B	1100000	96	60
101100	44	2C	1100001	97	61
101101	45	2D	1100010	98	62
101110	46	2E	1100011	99	63
101111	47	2F	1100100	100	64
110000	48	30	1100101	101	65
110001	49	31	1100110	102	66
110010	50	32	1100111	103	67
110011	51	33	1101000	104	68
110100	52	34	1101001	105	69

BASE CONVERSION TABLE (Continued)

BIN	*DEC*	*HEX*	*BIN*	*DEC*	*HEX*
1101010	106	6A	10011111	159	9F
1101011	107	6B	10100000	160	A0
1101100	108	6C	10100001	161	A1
1101101	109	6D	10100010	162	A2
1101110	110	6E	10100011	163	A3
1101111	111	6F	10100100	164	A4
1110000	112	70	10100101	165	A5
1110001	113	71	10100110	166	A6
1110010	114	72	10100111	167	A7
1110011	115	73	10101000	168	A8
1110100	116	74	10101001	169	A9
1110101	117	75	10101010	170	AA
1110110	118	76	10101011	171	AB
1110111	119	77	10101100	172	AC
1111000	120	78	10101101	173	AD
1111001	121	79	10101110	174	AE
1111010	122	7A	10101111	175	AF
1111011	123	7B	10110000	176	B0
1111100	124	7C	10110001	177	B1
1111101	125	7D	10110010	178	B2
1111110	126	7E	10110011	179	B3
1111111	127	7F	10110100	180	B4
10000000	128	80	10110101	181	B5
10000001	129	81	10110110	182	B6
10000010	130	82	10110111	183	B7
10000011	131	83	10111000	184	B8
10000100	132	84	10111001	185	B9
10000101	133	85	10111010	186	BA
10000110	134	86	10111011	187	BB
10000111	135	87	10111100	188	BC
10001000	136	88	10111101	189	BD
10001001	137	89	10111110	190	BE
10001010	138	8A	10111111	191	BF
10001011	139	8B	11000000	192	C0
10001100	140	8C	11000001	193	C1
10001101	141	8D	11000010	194	C2
10001110	142	8E	11000011	195	C3
10001111	143	8F	11000100	196	C4
10010000	144	90	11000101	197	C5
10010001	145	91	11000110	198	C6
10010010	146	92	11000111	199	C7
10010011	147	93	11001000	200	C8
10010100	148	94	11001001	201	C9
10010101	149	95	11001010	202	CA
10010110	150	96	11001011	203	CB
10010111	151	97	11001100	204	CC
10011000	152	98	11001101	205	CD
10011001	153	99	11001110	206	CE
10011010	154	9A	11001111	207	CF
10011011	155	9B	11010000	208	D0
10011100	156	9C	11010001	209	D1
10011101	157	9D	11010010	210	D2
10011110	158	9E	11010011	211	D3

BASE CONVERSION TABLE (Continued)

BIN	*DEC*	*HEX*	*BIN*	*DEC*	*HEX*
11010100	212	D4	11101010	234	EA
11010101	213	D5	11101011	235	EB
11010110	214	D6	11101100	236	EC
11010111	215	D7	11101101	237	ED
11011000	216	D8	11101110	238	EE
11011001	217	D9	11101111	239	EF
11011010	218	DA	11110000	240	F0
11011011	219	DB	11110001	241	F1
11011100	220	DC	11110010	242	F2
11011101	221	DD	11110011	243	F3
11011110	222	DE	11110100	244	F4
11011111	223	DF	11110101	245	F5
11100000	224	E0	11110110	246	F6
11100001	225	E1	11110111	247	F7
11100010	226	E2	11111000	248	F8
11100011	227	E3	11111001	249	F9
11100100	228	E4	11111010	250	FA
11100101	229	E5	11111011	251	FB
11100110	230	E6	11111100	252	FC
11100111	231	E7	11111101	253	FD
11101000	232	E8	11111110	254	FE
11101001	233	E9	11111111	255	FF

appendix h
using the available diskette

You are encouraged to experiment with all the programs provided in this book. To make this easier to do, a diskette is available with many of the programs discussed. You should make a backup copy of this diskette, placing the original away for safe keeping and use the backup copy. Here are a set of steps to aid you in making the backup.

1. Put a write protect tab on the orginal diskette. This is very important and will protect you from making a mistake when copying.
2. Before you proceed check the original diskette by reading the directory.
   a. Place the original diskette in the disk drive.
   b. Type: `LOAD ''$'',8` carriage return
   c. Type: `LIST` carriage return

At this point you should have on the screen:

```
**********************************************
Ø "MAGIC            " DD  2A
6    "MAIN MENU"          PRG
5    "MASKEDIN"           PRG
5    "SCREENREADER"       PRG
2    "SEQREAD"            PRG
6    "SEQPHONEBK"         PRG
3    "READREL"            PRG
8    "RELPHONEBK"         PRG
12   "ISAMPHONEBK"        PRG
3    "BETTERINPUT"        PRG
```

```
34   "MAGICFILER"          PRG
27   "SMALLMAGIC"          PRG
1    "LOTTO"               PRG
551 BLOCKS FREE.
**********************************************
```

If you do not have the proceeding on your screen, try step 2 again. If you still can not read the directory you have a faulty diskette or faulty disk drive and you should see your software or Commodore dealer.

3. Now you need to initialize a diskette to be used as a backup. Be sure to use either a blank diskette or one whose contents you wish to discard.
   a. Remove the original diskette.
   b. Insert the blank diskette.
   c. Type: `OPEN 15,8,15` carriage return
   d. Type: `PRINT#15,"NØ:MF-BACKUP,64"` carriage return

This operation will format the diskette and will take a few minutes. The drive will groan as though it is in pain but this is normal.

4. Check the backup diskette by reading the directory.
   a. Type: `CLOSE 15` carriage return
   b. Type: `LOAD ''$'',8` carriage return
   c. Type: `LIST` carriage return

At this point you should see on the screen:

```
**************************************
Ø "MF-BACKUP        "64    2A
664 BLOCKS FREE.
**************************************
```

If you do not have the above screen, try steps 3 and 4 with a different diskette. If all has gone well then you have initialized the backup diskette and you are now ready to copy the programs from the original diskette to the backup.

5. a. Remove the backup diskette.
   b. Insert the original diskette.
   c. Type: `LOAD ''MAIN MENU'',8` carriage return
   d. Remove the original diskette.
   e. Insert the backup diskette.
   f. Type: `SAVE ''MAIN MENU'',8` carriage return
6. Repeat step 5 for all the following programs. A chart to help keep track of the copying process. Note that the first program has already been copied in step 5.

	Remove Backup Diskette	Insert Original Diskette	Load Program	Remove Original Diskette	Insert Backup Diskette	Save program
"MAIN MENU"	a__	b__	c__	d__	e__	f__
"MASKEDIN"	a__	b__	c__	d__	e__	f__
"SCREENREADER"	a__	b__	c__	d__	e__	f__
"SEQREAD"	a__	b__	c__	d__	e__	f__
"SEQPHONEBK"	a__	b__	c__	d__	e__	f__
"READREL"	a__	b__	c__	d__	e__	f__
"RELPHONEBK"	a__	b__	c__	d__	e__	f__
"ISAMPHONEBK"	a__	b__	c__	d__	e__	f__
"BETTER INPUT"	a__	b__	c__	d__	e__	f__
"MAGICFILER"	a__	b__	c__	d__	e__	f__
"SMALLMAGIC"	a__	b__	c__	d__	e__	f__
"LOTTO"	a__	b__	c__	d__	e__	f__

Any program on the disk can be accessed by loading it directly and by loading the Main Menu.

index

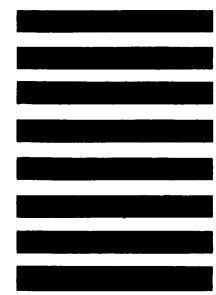
About the Author

Paul Goodman is an Instructor of Computer Science at Queens College of the City University of New York. He has taught over 2,500 students how to program.

Now you can easily access advanced BASIC programs and data filing techniques with the stroke of a few keys!

See over for details . . .

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1976 BOWIE, MD

POSTAGE WILL BE PAID BY ADDRESSEE

Brady Communications Co., Inc.
A Prentice-Hall Publishing Company
Bowie, Maryland 20715

Related Resources Shelf

Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers

Michael Richter

Learn how good programs are written, how to read and use them, how to tell a good one from a bad one, and how to gain knowledge trough the experience of writing advanced software.

☐ 1984/204pp/paper/D3022-3/$12.95

The Commodore 64 Software Buyer's Guide

Gary Phillips

Here's a complete guide to software for the Commodore 64. Includes word processing, games, graphics, data management.

☐ 1984/352pp/paper/D3820-0/$16.95

Handbook of BASIC for the Commodore 64

David I. Schneider, Fred E. Mosher

A complete and easy-to-use BASIC reference manual designed for programmers of all levels. Organized alphabetically by BASIC statement, this handy guide allows the reader to pinpoint information without wading through lengthy descriptions.

☐ 1984/368pp/paper/D505X-7/$14.95

Machine Language for the Commodore 64 and other Commodore computers

Jim Butterfield

This learn-by-doing tutorial explores machine code in the real environment of Commodore personal computers, examining important concepts such as output. . .address modes. . .linking BASIC and machine language. . .memory maps of the interface chips. . .and much more.

☐ 1984/326pp/paper/D6528-6/$12.95

To order, simply clip or photo copy this entire page, check off your selection, and complete the coupon below. Enclose a check or money order for the stated amount.

(Please add $2 postage/handling per book plus local sales tax.)
Or call toll-free 800-638-0220. In Maryland call 301-262-6300.

Mail to:
Brady Communications Co., Inc. • Dept. TS • Bowie, MD 20715

Name _____

Address _____

City/State/Zip _____

Charge my credit card instead: ☐ MasterCard ☐ VISA

Account # _____ Expiration Date _____

Signature _____

Prices subject to change without notice.

". . .nowhere else have I seen file handling explained so well."

"Advanced programmers will be motivated by the fact that they will possess several new skills by the time they finish."

"The book includes the 'right stuff' in the 'right way'."

THE COMMODORE 64 GUIDE TO DATA FILES & ADVANCED BASIC

Paul Goodman

Harness the hidden power of your C64 with advanced programming techniques! Take **The Commodore 64 Guide to Data Files & Advanced BASIC** home with you. . .put it on your shelf for reference or keep it at your side to challenge your programming wits. Don't miss this opportunity for a book that speaks to you, the serious programmer.

Includes tips on file handling and writing data base management programs designed to help you manipulate large amounts of information and learn how to solve sophisticated, substantial problems. Most importantly, this infinitely useful guide doesn't just *tell* you how. . .it *shows* you how with practical examples!

You'll also find information on:

- Advanced Input/Output, how to get the keyboard and the screen to do what you want
- Program design techniques, what to consider for designing better programs
- Advanced BASIC statements and structures
- Simons' BASIC

CONTENTS

ALSO AVAILABLE—an accompanying diskette featuring major programs from the text. Sold separately or as part of a book/diskette package.

ISBN 0-89303-375-8