COMMODORE 64 DATA FILE PROGRAMMING

GLENN FISHER
LEROY FINKEL
JERALD R. BROWN

COMMODORE 64™: DATA FILE PROGRAMMING

OVER THREE MILLION PEOPLE HAVE LEARNED TO PROGRAM, USE, AND ENJOY MICROCOMPUTERS WITH WILEY PRESS GUIDES

OTHER COMMODORE BOOKS

BASIC Subroutines for Commodore Computers, Adamis
*__Mastering the Commodore 64™__, Jones & Carpenter
Winning Games on the Commodore 64™, Barrett & Colwill
Winning "Strategy Games" on the Commodore 64™, Matthews & Smith
*__Commodore 64™ Basics: Self-Teaching Guide,__ Harris
Sound and Graphics for the Commodore 64™, Moore
Programming Tips for the Commodore 64™, Highmore & Page

*Book/Disk set available

COMMODORE 64™: DATA FILE PROGRAMMING

GLENN FISHER
Alameda County Office of Education

LEROY FINKEL
San Mateo County Office of Education

JERALD R. BROWN
Educational Consultant

A Wiley Press Book
JOHN WILEY & SONS, INC.
New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy V. Wilson
Editor: Theron R. Shreve
Managing Editor: David Sobel
Composition and Make-Up: Cobb/Dunlop, Inc.

Library of Congress Cataloging in Publication Data
Fisher, Glenn, 1947-
Commodore 64 data file programming.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. File organization (Computer science) I. Finkel, LeRoy.
II. Brown, Jerald, 1940- . III. Title.
QA76.8.C64F57 1985 001.64'2 84-22063
ISBN 0-471-80734-6
Printed in the United States of America

85 86 10 9 8 7 6 5 4 3 2 1

How to Use This Book

When you use the self-instruction format in this book, you will be actively involved in learning data file programming in BASIC. Most of the material is presented in sections called frames, each of which teaches you something new or provides practice. Each frame also gives you questions to answer or asks you to write a program or program segment.

You will learn best if you actually write out the answers and try the programs on your computer. The questions are carefully designed to call your attention to important points in the examples and explanations, and to help you apply what is being explained or demonstrated. We cannot urge you too strongly to really "fill in the blanks" for rapid and accurate learning.

Each chapter begins with a list of objectives—what you will be able to do after completing that chapter. At the end of each chapter is a self-test to provide valuable practice.

The self-tests do triple duty. They can be used as a review of the material covered in the chapter. Or you can read and work through a chapter, take a break, and save the self-test as a review before you begin the next chapter. The self-tests also provide valuable practice, for maximum retention of the material learned. Starting with the Chapter 4 Self-Test, you are asked to write programs that can be used to either create data files or display the contents of data files. These data files are then used by other programs in later chapters, so please don't skip the self-tests! At the end of the book is a final self-test to assess your overall understanding of data file programming. You will find it easy, if you have worked through this self-instruction format without skipping over the practice programs.

Instructors will find this book to be an excellent text for intermediate or advanced courses in BASIC programming at the high school and college levels, as well as for computer center classes, university extension workshops, and in-house instructional settings.

This book is designed to be used with a computer close at hand. What you learn will be theoretical only until you actually sit down at a computer and apply your knowledge "hands-on." We strongly recommend that you and this book get together with a computer! Learning data file programming in BASIC will be easier and clearer if you have regular access to a computer so you can try the examples and exercises, make your own modifications, and invent programs for your own purposes. You are now ready to use data files in BASIC.

Preface

This text will teach you to program data files in BASIC. As a prerequisite to its use, you should have already completed an introductory course or book in BASIC programming and be able to read program listings and write simple programs: This is not a book for the absolute novice in BASIC. You should already be comfortable writing your own programs that use statements including string variables, string functions, and arrays. We do start the book with a review of statements that you already know, though we cover them in more depth and show you new ways to use them.

The book is designed for use by readers who have little or no experience using data files in BASIC (or elsewhere, for that matter). We take you slowly and carefully through experiences that "teach by doing." You will be asked to complete many programs and program segments. By doing so, you will learn the essentials and a lot more. If you already have data file experience, you can use this book to learn about data files in more depth.

The particular data files explained in this text are for the BASIC language used on the Commodore 64, the Commodore PET, and the Commodore CBM, with a cassette recorder or with single or multiple disk drives. Cassette users will be able to use only the files described in Chapters 4, 5, and 6. Owners of Commodore PET 4016, PET 4032, or CBM 8032s will find a special appendix detailing the advantages of BASIC 4.0 for disk operations and data file programming. Data file programming in other versions of BASIC will be similar, but not identical to that taught in this book. You will find this book most useful when used in conjunction with the appropriate reference materials for your computer. For the Commodore 64, these include the *Programmer's Reference Manual* and the *Commodore 64 User's Guide*. References for the other Commodore computers include the *User's Reference Manuals* for Commodore BASIC and the Dual Floppy Disk Drives or Single Floppy Disk Drive, and the *PET/CBM Personal Computer Guide*. For the serious programmer, Raeto West's *Programming the PET/CBM* from Compute! Books is an excellent reference.

Data files are used to store quantities of information that you may want to use now and later; for example, mailing addresses, numeric or statistical information, or tax and bookkeeping data. The examples presented in this book will help you use files for home applications, for home business applications, and for your small business or profession. When you have completed this book, you will be able to write your own programs, modify programs purchased from commercial sources, and adapt programs using data files that you find in magazines and other sources.

Contents

Entering Commodore Cursor and Screen Control Characters

To make the program listings in this book as clear as possible, those Commodore symbols that print as graphics characters have been replaced with an appropriate word surrounded by brackets.

When you type the programs, DO NOT TYPE THE BRACKETS OR THE WORD. INSTEAD, press the appropriate key. If necessary, hold down the SHIFT key while pressing the appropriate key.

For example, when you see

```
[DOWN]
```

simply press the CRSR down key. For

```
[CLR]
```

you would hold down the SHIFT key, and, while holding it down, press the CLR/HOME key.

Note that both a SPACE and a SHIFT-SPACE show a blank space on the screen. For the few programs that call for SHIFT-SPACE, be sure to hold down the SHIFT key, as the computer knows the difference even though you can't see it! Also note that the two keys with arrows (up-arrow and left-arrow) are NOT cursor keys: be sure to use the keys marked CRSR for [LEFT] and [UP].

The following list shows the symbol that will appear on the screen if you press the key indicated.

[HOME] = ▒ (REVERSE-S)
[CLR] = ▒ (REVERSE-SHIFT S (HEART))
[RIGHT]= ▒ (REVERSE-])
[LEFT] = ▒ (REVERSE-SHIFT B (LINE))
[DOWN] = ▒ (REVERSE-Q)
[UP] = ▒ (REVERSE-SHIFT Q (DISC))
[SHSP] = (SHIFT-SPACE)
[PI] = π
[RVS] = ▒ (REVERSE-R)
[OFF] = ▒ (REVERSE-SHIFT R (LINE))

CHAPTER ONE

Writing BASIC Programs for Clarity, Readability, and Logic

Objectives: When you have completed this chapter you will be able to:

1. Describe how a program can be written using a top-to-bottom format.
2. Write an introductory module using REMARK statements.
3. Describe six prettyprinting rules.
4. Describe seven rules to write programs that save memory space.

INTRODUCTION

This text will teach you to use data files in BASIC. You should have already completed an introductory course or book in BASIC programming and be able to read program listings and write simple programs. This book is not for the absolute novice in BASIC but for those who have never used data files in BASIC (or elsewhere, for that matter).

The particular data files explained in this text are for the BASIC language used on the Commodore 64, the Commodore PET, and the Commodore CBM computers, with a cassette recorder, or with single or dual disk drives. Cassette users will be able to use only the files described in Chapters 4, 5, and 6. Owners of Commodore PET 4016, PET 4032, or CBM 8032s will find a special appendix detailing the advantages of BASIC 4.0 for disk operations and data file programming. The programs developed here will work on either version of BASIC.

Data file programming in other versions of BASIC will be similar, but not identical, to those taught in this book. You will find this book most useful when used in conjunction with the appropriate reference materials for your computer.

If you are new to the Commodore 64 (or PET) but not to BASIC, then it is especially important that you review the following manuals:

Commodore 64 User's Guide
Commodore 64 Programmer's Guide
User's Manual for Datasette or Disk Drive(s)

Since it is assumed that you have some knowledge of programming in BASIC and have practiced by writing small programs, the next step is for you to begin thinking about program organization and clarity. Because data file programs can become fairly large and complex, the inevitable debugging process—making the program actually work—can be proportionately complex. Therefore, this chapter is important to you because it provides some program organization methods to help make your future programming easier.

THE BASIC LANGUAGE

The computer language called BASIC was developed at Dartmouth College in the early 1960s. It was intended for use by people with little or no previous computer experience who were not necessarily adept at mathematics. The original language syntax included only those functions that a beginner would need. As other colleges, computer manufacturers, and institutions began to adopt BASIC, they added embellishments to meet their own needs. Soon BASIC grew in syntax to what various sources called Extended BASIC, Expanded BASIC, SUPERBASIC, XBASIC, BASIC PLUS, and so on. Finally, in 1978 an industry standard was developed for BASIC, but that standard was for only a "minimal BASIC," as defined by the American National Standards Institute (ANSI). Despite the ANSI standard, today we have a plethora of different BASIC languages, most of which "look alike," but each has its own special characteristics and quirks.

In the microcomputer field, the most widely used versions of BASIC were developed by the Microsoft Corporation and are generally referred to as MICROSOFT BASIC. These BASICs are available on a variety of microcomputers, but unfortunately the language is implemented differently on each computer system. Microsoft also sells its own versions of BASIC, called BASIC-80 and BASIC-86, useable on many microcomputers.

The programs and runs shown in the main text were actually performed on both a Commodore 64 with 1541 disk drive and on a PET 4032 with 4040 dual disk drive. Cassette programs were performed on the Commodore 64 with a Datasette. For the most part, we used only those language features that appear to be common in *all* versions of Microsoft BASIC. We have also tried to use BASIC language features common to most versions of BASIC, regardless of manufacturer. Rather than attempt to show off the bells and whistles found on the Commodore 64, we will try to present easy-to-understand programs that will run on or be easily adapted to all Commodore computers.

THE BASIC LANGUAGE YOU SHOULD USE: CONSERVATIVE PROGRAMMING

Since you will now be writing longer and more complex programs, *you should adopt conservative programming techniques so that errors will be easier to locate and isolate.* (Yes, you will still make errors. We all do!) This means that

you should not use all the fanciest features available in your version of BASIC until you have tested the features to be sure they work the way you think they work. Even then, you still might decide against using your fancy features, especially those that relate to printing or graphic output and cannot be duplicated on other computers. Some might be special functions that simply do not exist on other computers. Leave them out of your programs unless you feel you must include them.

We have found that not all software (BASIC) features work *exactly* as described in the manufacturer's reference materials, or that the description may be subject to misinterpretation. Thus, *the more conservative your programming techniques, the less change there is of running into a software "glitch."* This chapter discusses a program format that, in itself, is a conservative programming technique.

One reason for conservative programming is that your programs will be more portable or transportable to other computers. "Why should I care about portability?" you ask. Perhaps the most important reason is that you will want to trade programs with friends. But do all of your friends have a computer *identical* to yours? Unless they do, they will probably be unable to use your programs without modifying them. Conservative programming techniques will minimize the number of changes required.

Portability is also important for your own convenience. The computer you use or own today may not be one you will use one year from now, or you may enhance your system. In order to use today's programs on tomorrow's computer be conservative in your programming.

Use conservative programming to:

- Isolate and locate errors more easily.
- Avoid software "glitches."
- Enhance portability.

WRITING READABLE PROGRAMS

Look at the sample programs throughout this book and you will see that they are easy to read and understand because the programs and the individual statements are written in simple, straight-line BASIC code without fancy methodology or language syntax. It is as if the statements are written with the *reader* rather than the computer in mind.

Writing readable BASIC programs requires thinking ahead, planning your program in a logical flow, and using a few special formats that make the program listing easier to read. If you plan to program for a living, you may find yourself bound by your employer's programming style. However, if you program for

pleasure, adding readable style to your programs will make them that much easier to debug or change later, not to mention the pride inherent in trading a clean, readable program to someone else.

A readable programming style provides its own documentation. Such self-documentation is not only pleasing to the eye, it provides the reader/user with sufficient information to understand exactly how the program works. This style is not as precise as "structured programming," though we have borrowed features usually promoted by structured programming enthusiasts. *Our format organizes programs in MODULES, each module containing one major function or program activity.* We also include techniques long accepted as good programming, but for some reason forgotten in recent years. Most of our suggestions do NOT save memory space or speed up the program run. Rather, readability is our primary concern, at the expense of memory space. Later in this chapter, we will show some procedures to shorten and speed up your programs. Modular style programs will usually be better running programs and will effectively communicate your thought processes to a reader.

THE TOP-TO-BOTTOM ORGANIZATION

When planning your program, think in terms of major program functions. These might include some or all of the functions from this list:

```
DATA ENTRY
DATA ANALYSIS
COMPUTATION
FILE UPDATE
EDITING
REPORT GENERATION
```

Using our modular process, divide your program into modules, each containing one of these functions. Your program should flow from module 1 to module 2 and continue to the next higher numbered module. *This "top-to-bottom organization" makes your program easy to follow.* Program modules might be broken up into smaller "blocks," each containing one procedure or computation. The size or scope of a program block within a module is determined by the programmer and the task to be accomplished. Block style will vary from person to person, and perhaps from program to program.

```
USE A MODULAR FORMAT AND TOP-TO-BOTTOM APPROACH
```

REMARK STATEMENTS

Separate program modules and blocks from each other by REMARK statements or blank program lines.

In general, programs designed for readability make liberal use of REMARK statements, but do not be overzealous. A blank line (or nearly blank) can be induced by typing a line number followed by just REM. As a substitute, the line number can be followed by a colon (:). These nearly blank lines will help to set off and highlight program modules or routines.

```
100 REM       DATA ENTRY MODULE
110 REM **** READ DATA FROM LINES 9000-9999
120 :
130 REM
.
.
190 :
200 REM       COMPUTATION MODULE
```

Begin each program module, block, or subroutine with an explanatory REMARK and end it with a blank line REMARK statement indicating the end (see line 190 above).

Consistency in your use of REMARKs enhances readability. Use REM or REMARK, but be consistent. Some writers use the **** shown in line 110 to set off REMARK statements containing comments from other REMARK statements; others use four to six spaces after the REMARK before they add a comment (line 100). Both formats effectively separate REMARK comments from BASIC code.

You can place remarks on the same line as BASIC code using multiple statement lines, but be sure your REMARK is the LAST statement on the line. Such "on-line" remarks can be used to explain what a particular statement is doing. A common practice is to leave considerable space between an on-line remark and the BASIC code, as shown below.

```
220 LET C(X) = C(X) + U :       REM**COUNT UNITS IN C ARRAY
240 LET T(X) = T(X) + C(X):     REM**INCREASE TOTALS ARRAY
```

Liberal use of REMARK statements to separate program modules and blocks is desirable. Using REMARKs to explain what the program is doing is also desirable, but don't be overzealous or simplistic (LET C = A + B does not require a REMARK or explanation!). REM should add information, not merely state an obvious step. "Why" may be more important than "what" in some REMARKs.

Like everything else said in these first chapters, there will be exceptions to what we say here. Keep in mind that we are trying to get you to think through your programming techniques and formats a little more than you are probably accustomed to doing. Thus, our suggested "rules" are just that—suggestions to which there will be exceptions.

GOTO STATEMENTS

Perhaps the most controversial statement in the BASIC language is the unconditional GOTO statement. Its use and abuse causes more controversy than any other statement. Purists say you would *never* use an unconditional GOTO statement such as GOTO 100. A more realistic approach suggests that GOTOs and GOSUBs go *down* the page to a line number larger than the line number where the GOTO or GOSUB appears. This is consistent with the "top-to-bottom" program organization. This same approach, down the page, also applies to using IF . . . THEN statements (there will be obvious exceptions to this rule).

```
140 GOTO 210
150 IF X < Y THEN 800
160 GOSUB 8000
```

A final suggestion: A GOTO, GOSUB, or IF . . . THEN should not go to a statement containing *only* a REMARK. If you or the next user of your program run short of memory space you will delete extra REMARK statements. This, in turn, requires you to change all your GOTOs line numbers, so plan ahead first.

Bad

```
150 GOTO 300
.
.
.
.
300 REM  DATA ENTRY
310 INPUT "NAME? ";N$
```

Good

```
150 GOTO 300
.
.
.
.
.
.
299 REM   DATA ENTRY
300 INPUT "NAME? ";N$
```

A FORMAT FOR THE INTRODUCTORY MODULE

The first module of BASIC code (lines 100 through 199 or 1000 through 1999) should contain a brief description of the program, user instructions when needed, a list of all variables used, and the initialization of constants, variables, and arrays.

The very first program statement should be a REMARK statement containing the program name. Carefully choose a name that tells the reader what the program does, not just a randomly selected name. After the program's name comes the author or programmer's name and the date. For the benefit of someone else who may like to use your program, include a REMARK describing the

computer system and/or software system used when writing the program. Whenever the program is altered or updated, the opening remarks should reflect the change.

```
100 REM  PAYROLL SUBSYSTEM
110 REM  COPYRIGHT CONSUMER PROGRAMMING
         CORP., 1979
120 REM
130 REM  COMMODORE 64 BASIC 2.0
140 REM  MODIFIED BY G. FISHER
150 REM  11/08/83
```

Follow these remarks with a brief explanation of what the program does, contained either in REMARK or PRINT statements. Next add user instructions. For some programs you might offer the user the choice of having instructions printed or not. If instructions are long, place the request for instructions in the introductory module and the actual printed instructions in a subroutine toward the end of your program. That way, the long instructions will not be listed each time you LIST your program.

```
170 REM THIS PROGRAM WILL COMPUTE PAY
       AND PRODUCE PRINTED PAYROLL
180 REM REGISTER USING DATA ENTERED
       BY OPERATOR
190 REM
200 INPUT "DO YOU NEED INSTRUCTIONS ";R$
210 IF R$ = "YES" THEN GOSUB 800
220 REM
```

A handy little trick used by some programmers is to place a SAVE command in a program line near the beginning of the program. For example:

```
 1 GOTO 10
 2 PRINT "[CLR]SAVE" CHR$(34) "@0:PAYROLL" CHR
      $(34) ",8":STOP
READY.
```

This allows the programmer to simply type RUN 2, home the cursor, and press RETURN to save the current version of the program. The CHR$(34) prints the quotes, and the @ allows the program to replace the previous version of that program. Line 1 is necessary to prevent printing the SAVE line at the top of your screen every time you run the program. Of course, there must be a line 10 in your program.

With the 1540, 1541, or 2040 drives, there are occasionally problems using the "SAVE with replace" as shown here. With those drives, it is preferable to SCRATCH (erase) the file and then simply SAVE it. Refer to the section on SCRATCHing files in Chapter 5, to Appendix 5, and to your reference manuals.

This technique saves the program with the assigned name, and can help you avoid the problems created by saving a program with the wrong name and accidentally destroying another program. Notice that this technique keeps only the most current (hopefully corrected) version of your program on the disk, and destroys each previous version. If you want to keep earlier versions, then adding a number or some other unique designation, such as the date, to your program's name will be necessary.

Follow the description/instructions with a series of statements to identify the variables, string variables, arrays, constants, and files used in the program. Again, these statements communicate information to a *reader*, making it that much easier for you or someone else to modify the program later. We usually complete this section *after* we have completed the program so we don't forget to include anything. Sequential and random access files are also identified by name.

Assign a variable name to all "constants" used. Even though a constant will not change during the run of the program, a constant may change values between runs. By assigning it a variable name, you make it that much easier to change the value, that is, by merely changing one statement in the program. It is a good idea to jot down notes while writing the program so important details do not slip your mind or escape notice. When the program has been written and tested (debugged), go back through it, bring your notes up to date, and polish the descriptions in the REMARKs.

```
220 REM VARIABLES
230 REM    G = GROSS PAY
240 REM    N = NET PAY
250 REM    FT = FEDERAL INCOME TAX
260 REM    SI = STATE INCOME TAX
270 REM    F = SOC. SEC. TAX
280 REM    D = DISABILITY TAX
290 REM    X,Y,Z = LOOP VARIABLES
300 REM    H(X) = HOURS ARRAY
310 REM    N$ = NAME (20 CHARACTERS)
320 REM    PN$ = EMPLOYEE NUMBER (5 CHR)
330 REM
340 REM
350 REM CONSTANTS
360 REM    FR = .0613:REM SOC. SEC. RATE
370 REM    SD = .01         :REM SDI RATE
380 REM
390 REM FILES USED
400 REM    ITM = FEDL. TAX MASTER FILE
410 REM    STM = STATE TAX MASTER FILE
```

(Notice the method used to indicate string length in lines 310 and 320 and the use of on-line remarks in lines 360 and 370.)

The final part of the introductory module is the initialization section. In this section, dimension the size of all single and double arrays. Any variables that need to be initialized to zero can be done here for clear communication, even

though your computer initializes all variables to zero automatically. This section also includes any user-defined functions *before* they are used in the program. Data files are usually initialized and OPENed in this program module, as you will see in Chapter 4.

```
420 REM     INITIALIZE
430 DIM H(7), R(10,3), N$(30)
440 :
450 REM
```

THE MODULES THAT FOLLOW THE INTRODUCTION

The remainder of your program consists of major function modules, subroutines, DATA statements, and PRINT routines, when they are needed. Remember to separate each module from others by a blank line REMARK statement and/or a remark identifying the module. These modules can be further divided into user-defined program blocks, each separated by a blank line or a REMARK statement.

A typical second module would be for data entry. Data can be operator-entered from the keyboard or entered directly from DATA statements, a file, or other device. Chapter 3 discusses in detail how to write data entry routines with extensive error-checking procedures to ensure the accuracy and integrity of each data item entering the computer.

For now, we suggest that you write data entry routines so that even a completely inexperienced operator would have no trouble entering data to your program. This means the operator should *always* be prompted as to what to enter and provided with an example when necessary.

```
240 INPUT "TODAY'S DATE (MM/DD/YY)";D$
```

If data are entered from DATA statements, place the DATA statements near the end of your program (some suggest even past an END statement) using REMARK statements to clearly identify the type of data and the order of placement of items within the DATA statements.

```
9400 REM     DATA FOR CORRECT ANSWER ARRAY IN QUESTION ORDER
9410 REM     10 ANSWERS, RANGE 1-5
9420 REM
9430 DATA  4,5,1,3,2,1,1,1,4,4,5
9440 REM
9460 REM     RESPONDENTS ANSWERS TO QUESTIONS
9470 REM     RESP. ID # FOLLOWED BY 10 RESPONSES, 1-5
9480 REM
9490 DATA  17642, 4,5,1,3,2,2,1,4,4,4
9500 DATA  98126, 3,5,2,3,2,1,5,4,5,2
.
.
.
```

You can think of DATA statements as a separate program module. The "in-between" program modules might do computations, data handling, file reading and writing, and report writing. (Some BASIC programming stylists insist that DATA statements in a program be located near the READ statements to which they apply.)

Modular programming style dictates that all printing and report generation, *except error messages*, be done in one program module labeled as such. This limits the use of PRINT statements to one easy-to-find location within your program. (There might be more than one print module.) This makes it that much easier for you to make subsequent changes on reports when paper forms change or new reports are designed.

In the print module your program should NOT perform any computations except trivial ones. Make important computations *before* the program executes the print module(s). This may require greater use of variables and/or arrays to "hold" data pending report printing, but your programs will be much cleaner and easier to debug, since everything will be easy to find in its own "right" place. We have more to say about DATA in the next chapter.

SUBROUTINES

Program control flows smoothly from one module to the next. The ideal program module has one entry point at its beginning and *one* exit point at its end. The exception to this is a mid-module exit, as could happen in line 130 below. (This program segment checks each character in a user's numeric INPUT entry to see if illegal nonnumeric characters have been entered.)

```
100 INPUT "ENTER A VALUE "; V$
110 FOR X = 1 TO LEN(V$)
120 M = ASC( MID$( V$, X, 1)):    REM VAL OF CHA
    RACTER
130 IF M >= 48 AND M <= 57 OR M = 46 THEN 150
140 PRINT "INVALID ENTRY. ENTER NUMBERS AND DEC
    IMAL POINT ONLY." :GOTO 100
150 NEXT
160 :
```

A subroutine exit from a module always RETURNs to the next statement in the module. The use of subroutines is desirable provided you don't overdo it. Some program stylists recommend that the entire main program consist of nothing but GOSUB statements "calling up" a series of subroutines located later in the program. Such a technique is probably guilty of overkill. Strive for a happy medium between the two extremes of no subroutines and nothing but subroutines.

Technically, you need use a subroutine only to avoid duplicating the same program statements in two or more places in your program. A subroutine should be called from *more* than one place in your program. Otherwise, why use a

formal subroutine? Program stylists now agree that subroutines enhance readability and clarity and can be used at the convenience of the programmer (you!). However, again the caution—don't overdo it. Use subroutines to enhance the flow and readability of your program. Stylists also agree that subroutines should be clearly identified using REMARK statements and set off from other program sections with blank REMARK statements. Program stylists disagree, however, on where to place the subroutines. There are two schools of thought. Placement of subroutines can be either immediately past the end of the module that calls the subroutine or in one common module toward the end of the program.

Either

```
300 REM      COMPUTATION MODULE
310....
320....
330 GOSUB 410
340 GOSUB 460
.
.
.
.
400 REM      NUMBER CONVERSION SUBROUTINE
410 ....
.
.
.
.
450 REM      COMPUTATION SUBROUTINE
460 ....
.
.
.
```

Or

```
330 GOSUB 810
340 GOSUB 910
.
.
.
.
800 REM      NUMBER CONVERSION SUBROUTINE
810...
.
.
.
900 REM      COMPUTATION SUBROUTINE
910....
```

It is a good idea to include a REM statement after any GOTOs or GOSUBs which skip more than a few lines. In the second example, lines 330 and 340 don't make much sense without wading clear through to lines 810 and 910. The preferred version is

```
330 GOSUB 810 : REM NUM CONVERSION
340 GOSUB 910 : REM COMPUTATION
```

JUST FOR LOOKS: SPACING

You can do a host of things to your programs to enhance looks and clarity. These techniques are generally called "prettyprinting." Some of these techniques may not be possible on your computer; others may be done by it automatically. Try them out. If they work, use them to make your program look nicer.

One way to make your programs look nicer is to use line numbers of equal length throughout the program. If your program is small, use line numbers 100 through 999. If long, start the program at 1000 and continue to 9999. When your program is listed, it will be aligned neatly. If you have a programming utility (such as Power, SysRes, or Toolkit, or an extended BASIC such as Simon's BASIC), you may use it to renumber a program so the entire program is incremented in steps of 10. While it isn't necessary, it looks pretty and leaves room for future revisions.

Prettyprinting includes adding spaces in statements for clarity. Add spaces so the statement is easy to read. Between the line number and the first word, Commodore BASIC only keeps one space: if you type five, only one will show when the program is listed. If you leave no space after the line number, Commodore BASIC will insert one when you list the program. To indent program lines, you can type a colon (:) after the line number; spaces you type will then be kept. Commodore BASIC will also keep any spaces you type within a program line.

The following rules for spacing are suggested:

1. Use spaces to show the difference between REMARK comments and BASIC code.

Good

```
100 REM   SUBROUTINE TO TEST DATA ENTRY
110 REM
120 LET D$ = N$
130 LET Z = LEN(D$)
```

Better

```
100 REM  SUBROUTINE TO TEST DATA ENTRY
110 REM
120 :          LET D$ = N$
130 :          LET Z = LEN(D$)
```

2. Space before and after arithmetic operators and relational operators.

```
140 LET C = (A * B) / D
150 IF D$ <> C$ THEN PRINT "ENTRY ERROR"
160 IF C <= D THEN 700
```

3. Space after each comma in a DATA list. A space after the keyword DATA is not required, but helps. However, *do not* add extra spaces inside quotes in a data statement.

```
910 DATA 19.95, 26.5, 55
920 DATA PAPER CLIPS, DISKS
930 DATA PENS, "ENVELOPES, MANILA"
```

4. Space between BASIC statement keywords and variables. While this is not required, hidden errors can occur if the space is omitted. Do *not* space before the first parenthesis in TAB(and SPC(; nor before the # in INPUT# and PRINT#; nor before the $ in MID$, LEFT$, RIGHT$, CHR$, and STR$. In these words, the symbol is part of the keyword.

```
150 LET X = Y
160 FOR N = 1 TO X
170 IF R$ = "STOP" THEN 999
```

5. FOR NEXT loop spacing:
   a. Indent the body of a FOR NEXT loop two or three spaces.

```
100 FOR X = 1 TO 40
110 :  LET Y = 2 * D
120 :  PRINT X, Y
130 NEXT X
```

   b. Indent nested FOR NEXT loops two to three spaces, or use colons to indicate nesting.

```
100 FOR X = 1 TO 10
110 : FOR Y = 1 TO 5
120 :  LET Z(X,Y) = 0
130 : NEXT Y
140 NEXT X
```

or

```
100 :FOR X = 1 TO 10
110 :: FOR Y = 1 TO 5
120 :    LET Z(X,Y) = 0
130 :: NEXT Y
140 :NEXT X
```

6. Indent nested IF . . . THEN statements two to three spaces.

```
100 IF A$ = "STOP" THEN 999
110 :   IF B$ = "END" THEN 999
120 :      IF C = 0 THEN 999
130 :
```

7. Indent the body if IF . . . THEN loops.

```
140 IF X <> Y THEN 200
150 :    PRINT "INVALID ENTRY" :GOTO 240
```

OTHER TECHNIQUES TO ENHANCE LOOKS AND READABILITY

You can do still more to make your program clearer to you and another reader. These few ideas are the "finishing touch."

Using LET, even though unnecessary, is very readable. The absence of LET can be confusing, especially in a multiple statement line.

Confusing

```
260 X=Y : C = X*Y: IF X=N THEN X=C
```

Better

```
140 LET X = 0 : Y = 0 : C = 0
```

Best

```
260 LET X = Y : LET C = X * Y : IF X = N THEN LET X = C
```

If you do use LET, include a space between LET and the variable.

Arrange BASIC statements so that they read smoothly from left to right, just as the readers' eyes flow across the paper. This includes placing A before B and 1 before 2.

```
150 READ A, B, C
```

If your typed statement is long, it is probably confusing, especially if it is a mathematical equation. Break it into two or more pieces so it is easy to read. Read the statements aloud to test their readability.

Confusing

```
250 LET T = (N * 3.75) + ((N - 40) * 3.25) + ((N - 60) / 3)/
    ((D * N) * A)
```

Clearer

```
250 LET T = (N * 3.75) + ((N - 40) * 3.25)
255 LET T = T + ((N - 60) / 3) / ((D * N) * A)
```

If you have a Commodore printer, you can maximize the "prettiness" of any printed report by sending special formatting commands to the printer. Be aware, however, that this limits the use of your program to Commodore printers. SPC can also be used to format printed reports; TAB does not work on printers. The best way to provide "prettyprinting" is to set up strings of the proper length and print them, avoiding any formatting commands. Commodore's unique cursor control characters (CLR, HOME, CRSR right, left, up, and down) can be used to format screen presentations. However, these commands are not available on other computers or printers, and your program will not be portable. One possible solution is to place all display routines in clearly identified subroutines so a reader is directed to the right section of your program to make changes.

UNDOING IT ALL TO SAVE SPACE AND SPEED UP RUN TIME

After reading all these rules and ways to enhance readability, you are probablay wondering how you will possibly remember them all. You probably won't, but we hope we have at least sensitized you to the need for writing clear, readable programs. You will adopt your own coding style based on some of these techniques, plus others that you devise for convenience.

Nearly every technique illustrated in this chapter uses what some would consider to be unnecessary memory space. You may in fact find that your computer memory is filled before you have completely entered a long program. When this happens, either rethink your entire problem-solving technique or look for ways to save memory space by making changes to your program. A well-written, readable program takes up more memory space than a poorly written, less readable program. Thus, to save memory space, you may have to undo some of the things you did to enhance readability.

To save large numbers of memory "bytes":

1. Use multiple statements per line.
2. Delete all REMARK statements, beginning with the introductory module.

For further space saving (on the order of 10%):

1. Delete all spaces between characters and words (including BASIC keywords).
2. Omit LET from assignment statements.
3. Dimension arrays sparingly (as close as possible to the size needed).
4. Reuse variables when possible (normally a terrible technique).
5. Delete unnecessary parentheses and semicolons.

If you are concerned about the speed of your program, you can use some techniques to shave microseconds, or even seconds, off the run time. Some of the space-saving techniques above improve the performance of the program. Careful attention to the craft of programming itself may often save time, in providing optimized code. In addition, the following techniques will save time:

1. Define the most commonly used variables first; define all constants before defining any arrays.
2. Define all constants as floating-point variables (C1=3.1415: C2=1.0: C3=0, etc.).
3. Only use floating-point variables.
4. Use FOR NEXT loops instead of IF . . . THEN loops.
5. GOTOs and GOSUBs should either go to the start of the program, or to a line number more than 256 greater.

Remember, these techniques may speed up the run time, but some are considered poor programming style and contrary to the philosophy expressed in this chapter. In the final analysis, the most important way to achieve good program performance is to use the best algorithm, created by the insightful art of translating the steps of solving a problem or performing a job into instructions for the computer.

To save space and lessen distraction we have not followed *all* the rules suggested in this chapter in the rest of this book. However, you will still find our programs easy to read and self-documenting.

Chapter 1 Self-Test

1. Will a useful program written in BASIC on one computer system also run on a different brand of computer that uses BASIC? Why or why not?

_____

_____

_____

2. How can you be most certain that a program you write will also run on another person's computer?

3. What is meant by the portability of a computer program?

4. Name at least three types of information to include in REMARK statements in a program's introductory module.

5. Describe the "top-to-bottom format" for organizing programs.

6. When branching statements such as GOTO and GOSUB are used, what statements should not be branched to and why?

7. Define "initializing."

8. What is the most important reason for designating a segment of a program as a subroutine accessed by GOSUB?

9. When writing a self-documenting, easy-to-read program, what sacrifices are made?

10. In the following multiple-statement line, how many statements will be executed?

```
10 REM CALCULATE : LET C = 12 : LET A = C * QT
```

11. How does example (b) demonstrate good programming style?

(a)
```
10 FORI=1TO10:FORJ=1TO10
20 LETA(I,J)=I*J
30 NEXTJ:NEXTI
```

(b)
```
10 FOR I = 1 TO 10 : REM INITIALIZE ARRAY
20 :   FOR J = 1 TO 10
30 :      LET A(I,J) = I * J
40 :   NEXTJ
50 NEXT I
```

Answer Key

1. The program might not run on a different brand of computer, because different computers use different versions of BASIC.
2. Use conservative programming techniques and the least fancy statements in your version of BASIC.
3. Portability means that the program is likely to run on many computers with few or no modifications.
4. Variables used and what they stand for, files used, descriptive name for program, description of program (if necessary), author of program, last revision of program, version of BASIC and/or system used (any three answers).
5. To the extent possible, the program is written so that it begins execution at the smallest line number and proceeds toward the largest, with a minimum of confusing branching within the program.

6. REMARK statements, in case they are removed from a program to save computer memory space, or speed up program execution time.
7. The first time in a program that value(s) are assigned to variables or elements in an array (often means assignment of zeros); dimensioning where needed.
8. The segment would otherwise have to repeated because it is used more than once in executing the program.
9. Amount of memory used, and speed of program execution.
10. None. The computer goes on to next line-numbered statement if it sees that the first statement in the line is a REMARK.
11. It shows spacing, nesting of loops, single statement per line, and use of REM.

CHAPTER TWO

An Important Review of BASIC Statements

Objectives: To review important aspects of BASIC. When you finish this chapter, you will be able to write BASIC statements using LET, READ, DATA, INPUT, GET, IF . . . THEN, FOR NEXT, GOSUB, RETURN, ON . . . GOTO, LEN, ASC, MID$, LEFT$, and RIGHT$.

INTRODUCTION

We assume you have used BASIC to write programs and that you can read and understand a listing of a BASIC program (are you BASICly literate?); this information serves as a review. Many of the programming techniques in this and the next chapter will be used over and over again in your programming of data files. Even masters at programming in BASIC should give the material a quick run-through. This is important information and skill to have under your belt so that you can give your fullest attention to learning file-handling BASIC statements and techniques in Chapter 4.

VARIABLE NAMES

In early versions of BASIC, you were very limited in the names you could choose for a variable. Commodore BASIC allows you to choose names of two characters, such as CO, T3, R+, or A. The first character must be a letter; the other can be any unshifted character on the keyboard. While you can use longer names, Commodore BASIC only uses the first two to identify variables; thus COUNT and COFFEE are both seen as CO and identical variables to your computer. In addition, you must be careful not to include BASIC keywords within your variable names. WORD and STOLE will give error messages, because the computer will see the BASIC keywords OR and TO in them. The list of BASIC keywords

in your manual tells you which letter combinations you cannot use within your variable names. Appendices 2 and 5 also contain lists of BASIC keywords.

Examples of keywords which are often accidentally included in variable names are IF, OR, TO, AND, FOR, OPEN, DATA.

Use of simple variable names (A, T1, Y$) precludes having to debug a program when the problem is inclusion of a BASIC keyword in a variable name. Notice in our examples that even with simple variables we have selected names that are more likely to be remembered and make sense to someone reading the program. We encourage you to do the same. Within the two-character limitation, you can often choose variable names that are meaningful. Use CO for count, T for total, GT for grand total, S for salary, N$ for name, etc., unless you feel that longer variable names really help keep your programming efforts organized. A table of variables is another useful way to keep your names straight.

The mnemonic advantage of longer variable names is offset by the danger of misspelling them—a debugging headache—and by the danger of including BASIC keywords. And there is also the problem of using two variables with the same first two characters (for example, RAD and RAL), which are identical to the computer.

The letters O and I are poor variable names since they are easily confused with the number 0 (zero), the number 1 (one), or the lowercase letter l (el). Some experienced programmers reserve a few variables and use them the same way in all programs they write. X, Y, and Z are popular as control variables in FOR NEXT loops. K and C are popular for counting in statements like LET C = C + 1.

Variables, also called variable names or labels, identify for the computer a particular place in its memory where information is stored. The information may be numeric (a value) or alphanumeric (a string, discussed more fully later). A value or string is first stored by an assignment statement (LET, READ, INPUT), and subsequent references to the variable tell the computer to use the value or string assigned to (and identified by) that variable. Assignment statements are included in this review of BASIC.

(a) Give two reasons for using simple variable names such as A, X3, and Y$.

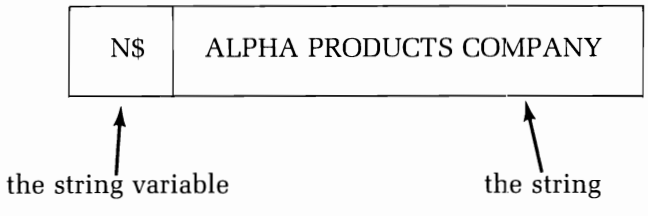
(a) 1. Conserves computer memory space.
2. No reserved words are accidentally used as a variable name.
3. Portability of programs between different versions of BASIC. (Any two answers.)

STRING VARIABLES

The rules for constructing names for string variables are the same as for numeric variables except that a string variable always has a dollar sign ($) as its last character. A is a numeric variable, whereas A$ is a string variable. A string is one or more letters, symbols, or numbers that can be used as information in a BASIC program. Strings are stored in the computer's memory with an assignment statement such as LET B$ = "EXAMPLE OF A STRING." The string variable B$ acts as a label in the computer's memory for the place where the string assigned to B$ is stored. A reference to B$ elsewhere in the program automatically tells the computer to use the string assigned to B$. The string assigned to a string variable is often referred to as the "value" of the string variable.

String variables act much like numeric variables and can generally be manipulated just like numeric variables. The crucial difference is that you cannot use string variables in arithmetic expressions and calculations, even if numeric information is assigned to the string variable. For example, LET F$ = "8.99" does not let you use F$ in numeric calculations, even though the string is comprised of numbers.

String variables and the strings assigned to them take up space in your computer's memory. You can visualize this as a box or compartment that contains alphanumeric information identified by a string variable. For example, the assignment statement LET N$ = "ALPHA PRODUCTS COMPANY" can be thought of as creating a storage compartment in the computer's memory like this:



Remember that a string assigned to a string variable in this way has the string enclosed in **quotation marks.** Only the information between the quotation marks comprises the string; the quotes themselves are not part of the string.

Many, if not most, business and personal applications of data files make much greater use of alphanumeric data (strings) than numeric data (numbers or values), so we are taking this opportunity to reinforce and extend your understanding of the use of string variables. Notice the word "alphanumeric." This term comes from the data processing industry and refers to data that may consist

of alphabetic characters, numeric characters, and/or special characters. For example, the product identification number FC1372 appearing in a catalog is alphanumeric data consisting of two alphabetic characters followed by four numeric characters. An address or hyphenated phone number is also alphanumeric data. To use and store such information in BASIC, assign it to a string variable (LET P$ = "FC1372") because a simple numeric variable would not accept the two alphabetic characters. If an identification number is mostly numeric, but includes a hyphen, asterisk, or even a space (e.g., 84992*, where the "*" denotes a special location, price, etc.), then it too requires the use of a string variable.

One string assigned to a string variable can have zero to 255 characters, including all spaces, punctuation, and special characters. A string with no characters (zero characters) is called a *null string* or empty string. An assignment statement for a null string would be: LET Z$ = "".

There is a crucial difference between the *maximum* length of a string (255 characters) and its *actual* length. The actual length is the number of alphanumeric characters presently assigned to the string variable and stored in the computer's memory. Remember, spaces count as characters. Consider the lengths of the following strings assigned to string variables.

N$	ALPHA PRODUCTS

Actual length: Fourteen characters

C$	MENLO PARK, CA. 94025

Actual length: Twenty-one characters (includes comma, period, and spaces)

Now you do this one:

A$	161 DAWN ST. SUITE 3

(a) What is the maximum length for a string assigned to A$?

_____

(b) What is the actual length of the string shown as assigned to A$ above?

_____

(a) 255 characters.
(b) Twenty characters.

Since Commodore BASIC automatically assumes that a string variable can be assigned a string with up to 255 characters, there is no need to dimension string variables. However, we recommend that you show a person using your program what the string size (maximum *actual* size) is for all string variables listed in the program. Do this by including REMARK statements in the introductory module as shown:

```
140 REM   STRING VARIABLES
150 REM     N$ = CUSTOMER NAME(20)
160 REM     A$ = CUST. ADDRESS(25)
170 REM     C$ = CUST. CITY(15),STATE(2),ZIP(5)
180 REM             (26 TOTAL INCLUDING SPACES)
190 REM
```

(a) How many characters are contained in a null string assigned to a string variable?

_____

(b) In the actual length of a string, how many characters does a space use?

_____

(a) Zero (none).
(b) One.

As noted earlier, you can *assign a string to a string variable using the LET statement.* Remember to place the string inside quotation marks, or the computer will reject the statement; it will tell you that an error has been made. Example:

```
240 LET N$ = "TYPE A POSITIVE"
```

Almost all versions of BASIC allow omitting the word LET from an assignment statement. For this reason, LET statements are sometimes called *direct assignment statements* to distinguish them from INPUT and READ assignment statements. A variable (numeric or string) followed by an equal sign (=) implies LET to BASIC; thus, the "implied LET" direct assignment statement can save a bit of typing and a little memory space. We generally include LET for clarity in reading a program listing. This statement:

```
240 N$ = "TYPE A POSITIVE"
```

means the same in BASIC as the example before this paragraph.

READ–DATA ASSIGNMENT STATEMENTS

DATA statements are like data files in that they hold data to be assigned to variables and are then used in a program. The difference is that a DATA statement holds data that can only be used by the program in which the DATA statement appears, whereas a data file can be created and the data used by a variety of different programs, since it is separate from the program itself. This will be explained in greater detail later.

The READ statement, which must have one or more DATA statements in the same program to READ from, is an assignment statement. One or more data items from a DATA statement are assigned to one or more variables by a READ statement.

```
10 READ A
20 DATA 15, 76.5, 1892, -999
```

The statement READ A assigns a single numeric value from the DATA statement to variable A.

```
10 READ A, B
20 DATA 15, 76.5, 1892, -999
```

The statement READ A, B assigns two consecutive values from the DATA statement; the first to variable A, the second to B.

A program can also use the READ and DATA statements to assign strings to string variables. A DATA statement can contain strings as data items, and these strings are assigned to string variables by a READ statement using the same procedure as for reading numeric values.

```
220 READ A$, B$, C$
  .
  .
  .
910 DATA BLUE, GREEN, GOLD
```

The individual string items in DATA statements do not have to be enclosed in quotation marks, unless the string data include:

COMMAS
SEMICOLONS
LEADING SPACES
FOLLOWING SPACES
SHIFTED CHARACTERS

Leading or following spaces are blank spaces that are to be included and considered part of the string. Shifted characters are capitals in upper/lowercase mode, or graphics in uppercase/graphics mode. If one or more of these is in-

cluded in the string, it *must* be enclosed in quotation marks, just as for a direct assignment statement. Note that the actual length of the data item includes these leading or following spaces, even though they seem invisible.

In line 900, the quotes are necessary because shifted characters are used; in lines 920 and 930 they are necessary around each data item because the punctuation is part of the string data items themselves.

```
220 read n$
230 :
.
.
.
900 data "Glenn Fisher"
910 data faucet, sink, drain
920 data "chains, bicycle"
930 data "Brown, Jerald R.", "Finkel, LeRoy P."
```

Try this test program to see how the "leading and trailing space" rule works.

```
220 READ N$, A$
230 ? N$; A$
910 DATA "TEST     ","     ITEMS"
```

(a) How many spaces should now appear between the strings when the program is run? _____

(a) Six spaces:

```
RUN
TEST      ITEMS
READY.
```

The computer uses an internal "pointer" system to keep track of items in a DATA statement that are "used up" or already assigned to variables in a program RUN. When executing READ–DATA statements, each time a data item is read and assigned to a variable the internal pointer advances one position in the DATA statement to the next data item. If the pointer is pointed at alphanumeric data (a string) and the READ statement is looking for numeric information to assign to a numeric variable, the program will terminate in an error condition. For example:

```
210 READ A
910 DATA ALPHA, NUMERIC
```

An error condition would result from executing this program segment because the statement READ A is "looking" for numeric data to assign to the numeric variable A, but the pointer is pointing at alphanumeric information.

What will happen if this program is RUN?

```
210 READ A$, B$
220 PRINT A$; B$
910 DATA 17926, NUMERIC
```

(a) Will the program RUN without an error condition? _____

(b) What will be assigned to A$ and why? _____

_____

(a) Yes.
(b) A$ = 17926, since a number can be assigned as a string to a string variable (but not vice versa).

Errors with READ and DATA can be very difficult to find. While mistakes in the DATA line usually result in an error message pointing to that line, some (such as misspelling "DATA") show up as an error in the READ line. If you get an error in a READ line, be sure to also check your DATA statements. Commas are critical; extra commas or commas at the end of a line result in an empty string (""). That is, your computer assumes there has been an empty string (i.e., nothing) between two adjacent commas, or after a comma at the end of a DATA statement line.

(a) Find the errors in this DATA line:

```
999 DAT ORANGES,,GRAPES,PICKLES,APPLES,
```

(a) The word DATA is misspelled, there are two commas after ORANGES, and there is a comma at the end of the line (after APPLES).

UNDERSTANDING INPUT, AN IMPORTANT ASSIGNMENT STATEMENT

You can enter numeric or alphanumeric information to be assigned to a numeric variable or a string variable using the INPUT statement.

When using INPUT statements, make certain that the data entry person using your program at a computer terminal knows exactly what kind of information to enter for assignment to a variable by the INPUT statement. To do so, you must fully understand how INPUT works in your version of BASIC.

_____

The INPUT statement should always include a prompting string (a message that appears on the screen) to tell the user exactly what sort of information is to be entered. A typical format for an INPUT statement is:

```
160 INPUT "WHAT IS YOUR AGE"; A
```

An INPUT statement *without* a prompting message (the part enclosed by quotes) causes the computer to display a question mark; the computer then waits for a response from the keyboard.

```
10 INPUT A
20 PRINT A
RUN
?
```

There is nothing more frustrating to a computer user than an INPUT question mark with no hint as to what sort of response is requested. *Always use a prompting string in an INPUT statement.* If necessary, use PRINT statements preceding the INPUT statement to explain to the user what information to enter.

INPUT always displays a question mark (?) on the screen. If you include a prompt in the INPUT statement, your prompt will be followed by the question mark. Of course, you can include punctuation in the prompt string, and a trailing space too, making your prompt easier to read:

```
360 INPUT "WEIGHT "; W
```

This displays the following (assuming the user has typed in "185"):

```
WEIGHT ? 185
```

Another source of user frustration is the funny responses the computer can make when incorrect data are entered. Consider the following example:

```
360 INPUT "WEIGHT ";W
RUN
WEIGHT ? 137 LBS.
?REDO FROM START
WEIGHT ?
```

The user entered 137 LBS. after the prompting message, then pressed the RETURN key. The computer responded with an error message, then repeated the prompt. The input called for a numeric quantity, but the user entered non-numeric information.

If a program asks for multiple entries to one INPUT statement and the user doesn't enter enough items separated from each other by commas, the computer will display a double question mark (??) when RETURN is pressed. The computer continues to display double question marks (??) until an entry has been made

for each variable in the INPUT statement. Since the prompt message is not displayed with the double question marks, this can be dismaying to the user.

```
360 INPUT "WEIGHT AND QUANTITY ";W,Q
RUN
WEIGHT AND QUANTITY ? 137
??
```

For an INPUT statement, here are the possible errors, and their messages:

1.	Too few items	displays double question marks (??)
2.	Too many items	?EXTRA IGNORED (error message)
3.	Wrong data type (string instead of numeric)	?REDO FROM START

If RETURN is pressed with no other entry, the Commodore 64 will use the previous value of that variable (if one exists); if not it will simply redisplay the prompt. However, Commodore PETs and CBMs will exit the program and display READY. There are several ways to avoid this—see Appendix 4 (differences between PET and C-64 input). These problems are serious and must be considered in developing programs.

Our insistence on the importance of understanding INPUT should now be hitting home. So what do you do for the accidental null string entry and other eccentricities of the INPUT statement?

Two programming techniques can help eliminate errors. First, ask the user to enter only *one* value or string per INPUT statement, period! This makes data entry (and data checking, as we will discuss in the next chapter) nice and clean. For example:

```
RUN
CUSTOMER NUMBER? 1025
CUSTOMER NAME? BISHOP BROTHERS
PRODUCT NUMBER? 18625
QUANTITY ORDERED? 144
```

Second, have *all* input entries, whether string or numeric, assigned to string variables. This eliminates error messages for numeric variables that cannot accept alphanumeric information for assignment. In the next chapter you will learn to test for null strings (no entry made) and appropriately advise the user with explicit messages as to the proper entry to be made.

(a) Write an INPUT statement that will result in the following RUN:

```
RUN
YOUR HOME ADDRESS?

100 _____
```

(a)
```
100 INPUT "YOUR HOME ADDRESS"; A$
```

CONCATENATION

Strings can be joined to form longer strings, a process called *concatenation*. Strings are concatenated in BASIC using the plus (+) sign. The process, however, is one of joining, not of arithmetic addition. For example, the strings assigned to F$ and L$ can be concatenated and the new, longer string assigned to another variable N$ in an assignment statement like this:

```
110 LET N$ = F$ + L$
```

Strings assigned to variables can be concatenated with string constants, like this:

```
120 LET G$ = N$ + "CUSTOMER"
```

or

```
150 LET N$ = F$ + " " + L$
```

The statement in line 150 above concatenates the strings associated with F$ and L$ and assigns them to N$, but it also places a space in the new N$ string between the parts of N$ that were (and still are) assigned to F$ and L$.

Look at the following program and show what will be printed when it is RUN.

(a)
```
10 LET F$ = "JANET"
20 LET L$ = "BARRINGTON"
30 LET N$ = F$ + " " + L$
40 PRINT N$

RUN
```

(a) JANET BARRINGTON

THE IF . . . THEN STATEMENTS

The IF . . . THEN statement in BASIC gives the language real power. Here's what it can look like.

```
140 IF X < Y THEN GOTO 800
```

The GOTO can be, and usually is, omitted. However, Commodore BASIC supports IF . . . GOTO, which has advantages if you are using a programming utility. We recommend you use the form that is clearest and least confusing to you.

The simplest form of IF . . . THEN is a *comparison* between two numeric values or expressions. IF the comparison is true, THEN (GOTO) a given line number and continue executing the program with the statement at that line number. Since GOTO is usually omitted, just the line number follows THEN. The possible comparisons are:

=	equals
<	less than
>	greater than
<=	less than or equal to*
>=	greater than or equal to*
<>	not equal to

**Note*: You can put the equal sign either before or after the greater than or less than symbol.

Commodore BASIC also includes in the IF . . . THEN family of statements:

IF . . . THEN LET . . .	Follow rules for regular LET statements. LET can be omitted.
IF . . . THEN GOSUB . . .	Line number follows GOSUB.
IF . . . THEN RETURN . . .	Possible but unusual (usually used in subroutines).
IF . . . THEN PRINT . . .	Follow all the rules for regular PRINT statements.
IF . . . THEN INPUT . . .	These two are possible, but are not recommended because of confusion and debugging complications.
IF . . . THEN READ . . .	
IF . . . THEN STOP . . .	STOP should be used only for debugging.
IF . . . THEN END . . .	
IF . . . THEN IF . . . THEN . . .	Possible; can be confusing, but useful with logical operations (IF A AND NOT B THEN . . .).

(a) What statement is implied after the THEN in the simplest form of the IF . . . THEN Statement? _____

(b) List at least five BASIC statements that can be part of IF . . . THEN statement and that will be executed if the condition (comparison) is true.

_____

_____

_____

_____

_____

(a) GOTO

(b) PRINT,GOTO (assuming a line number appears after THEN), LET (direct assignment statement, with the option of omitting the word LET), READ, INPUT, another IF . . . THEN statement (not recommended), GOSUB, RETURN (any five answers)

IF . . . AND . . . THEN . . . and IF . . . OR . . . THEN . . . are called the logical AND and logical OR. They allow you to put more than one comparison in a single IF . . . THEN statement. The comparisons on both sides of AND must be true for the entire IF . . . THEN comparison to be true. Only one comparison on either side of an OR need be true for the comparison to be true. You can use more than one AND and more than one OR between IF and THEN, and you may use both AND and OR in the same IF . . . THEN statement, which allows three or more comparisons in one IF . . . THEN statement!

Be certain you understand how to use the logical AND and OR to produce the results you want. We find they are useful for certain checks on user INPUT entries. If an INPUT value should be between five and twenty, inclusive, then the following statement would check that the value was within these parameters.

```
150 IF F <= 5 OR F >= 20 THEN PRINT "ENTRY IS
    INCORRECT."
```

Alternately, the following line would check for "within bounds" parameters for the value assigned to F, instead of "out of bounds" values.

```
160 IF F > 5 AND F < 20 THEN PRINT "ENTRY IN
   BOUNDS."
```

Note: Be very careful to have your logic straight or such comparison statements will not do what you want. Also, carefully check any parentheses you may have used; incorrect placement can wreak havoc. For some, flow charts help visualize the alternatives so you can properly construct your comparison statements. We

often insert a PRINT statement in the IF . . . THEN statement and in the following line, and test the program segment with many different values to be sure it branches correctly. Thorough testing of programs and program segments with every conceivable mistake you could enter is a must.

(a) Write two IF . . . THEN statements, one using a logical AND and another using a logical OR. The statement should test to see if the value assigned to variable Y is greater than, but not equal to, zero, and less than, but not equal to, one. When the comparison is true, one statement should print the message BETWEEN ZERO AND ONE, and the other should print NOT BETWEEN ZERO AND ONE.

- - - - - - - - - - - - - - - - - - -

(a)
```
IF Y > 0 AND Y < 1 THEN PRINT "BETWEEN ZERO AND ONE"
IF Y <= 0 OR Y >= 1 THEN PRINT "NOT BETWEEN ZERO AND ONE"
```

Having seen how more than one comparison can be made within a single IF . . . THEN statement, now consider the other end of the comparison statement and how to have more than one instruction executed in the case of a true IF . . . THEN comparison.

Commodore BASIC permits you to do nearly anything after an IF . . . THEN expression, frequently encouraging you to place multiple statements on one line.

```
150 IF X < Y THEN PRINT "TOO LOW": LET C = C + 1 : GOTO 10
160 IF X > Y THEN LET C = C + 1: LET G = 0 : GOTO 10
```

When you use multiple statement program lines, keep in mind that you may be hindering the portability of your program. If this doesn't concern you, forget it! We do urge you to complete your entire "activity" on one line after an IF . . . THEN statement, otherwise the program is extremely awkward to follow. If you cannot complete your activity on one line, then GOTO a section (or GOSUB) where all of the activity can be done together. Follow the acceptable example:

Harder to follow

```
150 IF X < Y THEN LET X = X + D: LET Y = Y/N: GOTO 200
160 IF X > Y THEN LET X = X - D: Y = Y/N: GOTO 10

200 LET C = C + 1 : PRINT "TOO LOW": GOTO 10
```

Acceptable

```
150 IF X < Y THEN 200
160 IF X > Y THEN 250

200 LET X = X + D
210 LET Y = Y/N
220 LET C = C + 1      . . . or all on one line
230 PRINT "TOO LOW"
240 GOTO 10
```

Most of us who program for fun ignore what is going on inside the computer because we don't have to pay attention. However, on occasion, little "bugs," inconsistencies, and our own ignorance can cause some interesting (and frustrating) problems. BASIC software sometimes does funny things, barely detectable because the problem exists at the seventh or eighth decimal location, which may be invisible to the BASIC user. We once spent hours trying to fix a "money changing" program that kept giving us 4.9999 pennies change instead of a nickel. (This points out a very important lesson: Your BASIC language interpreter does not always do things with the accuracy and consistency you might expect.) Therefore when you are comparing floating-point numeric values, especially numbers that have been computed by your computer, try to compare using less than (<), greater than (>), or not equal (<>). The inaccuracies are most likely to occur from division or exponentiation, but can even occur with addition and subtraction of dollars and cents!

Good

```
IF X < 1125.75 THEN...
IF X > 1125.75 THEN...
IF X <> 1125.75 THEN.....
```

Not wise

```
IF X = 1125.75 THEN....
```

The method illustrated below eliminates an incorrect or false decision because of tiny floating-point computational errors:

```
100 IF ABS (X - 1125.75) < .001 THEN ...
```

(a) Why should you avoid IF . . . THEN comparisons for equality?

_____

_____

_____

_____

(a) Internal round-off errors may produce very slightly inaccurate values in calculations. Therefore, a comparison for equality might fail (be false) where you would expect the comparison to be true.

IF . . . THEN STRING COMPARISONS AND THE ASCII CODE

So far the only comparisons used in IF . . . THEN examples have been between two numeric expressions or values. Comparing strings in IF . . . THEN statements begins to get a little tricky. However, comparisons for equality or inequality are fairly straightforward. Examine these statements:

```
220 INPUT "YOUR LEGAL NAME:"; N$
230 IF N$ = "STOP" THEN 999
```

Notice that in line 230 a string variable (N$) is compared with a string constant ("STOP"). A string constant in a comparison must be enclosed in quotation marks. For two strings to be equal, each and every character in the two strings must be identical (upper- and lowercase are different), and the length of the strings and any leading or trailing spaces must be the same. Any difference whatsoever, and the equality comparison will be false. Because the INPUT statement in Commodore BASIC removes leading and trailing spaces from the input string, be sure you don't put unnecessary spaces inside quotes for the comparison string. The following example will illustrate this:

```
10 INPUT "YOUR NAME "; N$
20 IF N$ = " GLENN" THEN PRINT "MATCH!"
30 GOTO 10
```

Try entering the name GLENN with leading or trailing spaces; you won't get a match. Then remove the space before the G in line 20, and enter the name GLENN with several spaces before the G, or after the final N; now you'll always get a match, because the spaces are removed by the INPUT statement.

In line 230, the string assigned to a string variable was compared to a string constant. Likewise, the contents of two string variables can be compared.

```
310 INPUT "THE OLD TITLE:"; T$
320 IF T$ <> D$ THEN PRINT "WRONG TITLE. TRY ANOTHER": GOTO 310
```

The difficulty in string comparisons comes with the "less than" or "greater than" comparisons. These have application in sorting strings, alphabetizing data, or inserting new information into an alphabetically organized data file. In IF . . . THEN comparisons, BASIC compares the two strings one character at a time, from left to right.

Rather than comparing within the construct of a twenty-six-character alphabet, *BASIC uses a standard code that represents every possible signal a terminal keyboard can send to the computer (and vice versa).* Each key and each permitted combination of keys, such as the shift or CONTROL key along with another key, sends a unique electronic *code* pattern to the computer. *These patterns are represented by the decimal numbers 0 through 255 in the ASCII Code chart.*

Although the ASCII set has been standardized, Commodore chose to use a nonstandard code for the alphabet. ASCII stands for American Standard Code for Information Interchange. The ASCII code's 128-character set includes the upper- and lowercase letters of the alphabet, numbers, punctuation, and other special characters and function keys. The Commodore ASCII set also includes graphics characters and a reverse mode character set. These are given ASCII numbers 129 thru 255. Refer to the ASCII chart in Appendix 1 to understand the following information.

Commodore does not use a standard ASCII code. The numbers are in the same location (48–57), but where the alphabet is depends upon whether you are in upper/lowercase mode or in uppercase/graphics mode. In graphics mode, capital A is ASCII 65, and there is no lowercase. In upper/lowercase mode, Commodore ASCII is reversed from standard ASCII. Lowercase letters are 65 to 90, and uppercase starts at 97. The ASCII code for uppercase letters is thus 32 greater than the lowercase ASCII code.

a = 65, so A = 65 + 32 = 97

This fact will be of use later on.

What actually happens in an IF . . . THEN string comparison? BASIC compares the ASCII code number for each character in the two strings, comparing just one character at a time. As soon as an inequality exists between characters, the string with the character that has the lower ASCII code number will be considered "less than" the other string. BASIC *does not* add up the ASCII code values for the two strings being compared to determine "less than" or "greater than."

The following chart shows the results of comparing a series of strings assigned to A$ and B$. Notice that upper and lowercase letters have different ASCII codes.

```
A$      B$

ABC     ABD     A$ is less than B$
MN      MNO     A$ is less than B$
STOP    STO     B$ is less than A$   (A$ is greater than B$)
123A    123a    B$ is less than A$
YES     yes     B$ is less than A$
NO      No      B$ is less than A$
5       15      B$ is less than A$
11      2       A$ is less than B$
```

In the comparison process, if one string ends before the other and no other difference has been found, then the shorter string is said to be "less than" the longer one. One result is that a null string is always "less than" a non-null string, since a null string is empty and has no ASCII code at all. Here are some more examples of string comparisons:

```
A$              B$

SMITH           SMITHE      A$ IS LESS THAN B$
ALCOTJONES      ALCOT       A$ IS GREATER THAN B$ (B$ is less than A$)
JOHNSEN         JOHNSON     A$ IS LESS THAN B$
KELLOG          KELLOGG     A$ IS LESS THAN B$
EQ-8            EQ 8        B$ IS LESS THAN A$
```

Now it's your turn to familiarize yourself with ASCII code comparisons. Fill in the blanks with the appropriate string variable.

```
    C$              D$
(a) JACOB           JACOBS          _____is greater than_____
(b) LOREN           LORAN           _____is less than_____
(c) SMITH-HILL      SMITH HILL      _____is less than_____
(d) ABLE12          ABLE-12         _____is less than_____
(e) Theater         THEATER         _____is less than_____
(f) 95.2            95-2            _____is less than_____
(g) STOP            stop            _____is less than_____
```

(a) D$.C$ — D$ has more characters, others being equal.

(b) D$.C$ — Letter A is less than letter E.

(c) D$,C$ — A space is less than a hyphen.

(d) D$,C$ — A hyphen is less than the number 1.

(e) C$,D$ — Lowercase letters are less than uppercase letters.

(f) D$,C$ — A hyphen is less than a decimal point.

(g) D$,C$ — Lowercase letters are less than uppercase letters.

Two string functions are frequently used in conjunction with the ASCII code. The ASC ( ) function gives the ASCII code number for the first character of the string contained in the parentheses or for the first character of the string assigned to the string variable contained in the parentheses. The ASCII number produced by ASC ( ) may be assigned to a variable, displayed by a PRINT statement, used in arithmetic expressions, and used as a value in an IF . . . THEN comparison.

```
310  LET X = ASC(A$)

190  LET X = ASC("ANTWERP")

480  PRINT ASC(A$)

150  IF ASC(N$) = 0 THEN 110
```

Give the ASCII number or value that will be printed for each of these program segments. Refer to the ASCII chart in Appendix 1. We suggest you place a tab on the chart pages, or fold down the corner of that page.

(a)
```
10 LET D$ = "DOLLAR"
20 PRINT ASC(D$)
RUN
```

_____

(b)
```
10 PRINT ASC("YES")
RUN
```

_____

(c)
```
10 LET F$ = "FRANK"
20 LET L$ = "JONES"
30 LET N$ = L$ + "," + F$
40 PRINT ASC(F$)
50 PRINT ASC(L$)
60 PRINT ASC(N$)
RUN
```

_____

_____

_____

(d)
```
10 PRINT ASC(" ")
RUN
```

_____

(a) 68
(b) 89
(c) 70
    74
    74
(d) 32

Describe the string that must be assigned to A$ in order for the following IF . . . THEN comparisons to be true.

(a) `IF ASC(A$) = 53 THEN 510` _____

(b) `IF ASC(A$) < > 48 THEN 810` _____

(a) First character in A$ is 5.
(b) First character in A$ is not zero.

A string to which no characters have been assigned has no ASCII code numbers associated with it. Therefore, the ASC function would not be able to find a character to convert into an ASCII code number in an empty string.

If the computer tried to execute the function ASC$(A$) when A$ had not been assigned to a string, or when A$ was an empty string (" "), then the error message "?ILLEGAL QUANTITY ERROR" will result. For this reason, in a program you will often see ASC(A$ + CHR$(0)), which returns a value of zero even if A$ is empty or undefined.

The opposite of the ASC( ) function is the CHR$( ) function. An ASCII number is placed in the parentheses. You can also use CHR$( ) in a PRINT statement to print a character corresponding to the ASCII number in each set of CHR$( ) parentheses.

```
840 PRINT CHR$(69); CHR$(78); CHR$(68)
```

(a) By running this program or by reference to the ASCII chart, what will this program line print? _____

(a) END

CHR$(7) in a PRINT statement sounds the beeper on CBM 8032s, PET 4016s, and PET 4032s. It is the standard "bell" code in ASCII, so it will sound a beep on many other computers. To sound a beep on the Commodore 64 or the PET 2001 requires a different technique; see the appropriate reference manuals.

CHR$(34) produces a quotation mark. Since a pair of quotation marks defines a string, you can't include quotation marks directly within a string. Instead, you can use CHR$(34). This allows you to display quotation marks on the screen or printer, or to include them within a string. Here is an example:

```
240 PRINT  "PRESS " CHR$(34) "C" CHR$(34) " TO
     CONTINUE."
READY.
```

It is also possible to place characters directly on the screen without a PRINT statement, using POKE. Likewise, the character at a specific location can be read with PEEK. These are not covered in this book; refer to books on Commodore graphics or your Commodore reference manuals. Keep your Commodore reference materials at hand for details on other features of the Commodore 64 not covered in detail in this book (particularly sound and graphics). Check the ASCII code chart, particularly 0–31, 128–192, and 219–255, to explore other interesting capabilities of this computer, and its graphics character set.

Two other extremely thorough and helpful references are Raeto Collin West's *Reference Guide to the PET/CBM* (Compute! Books), and the Strasma's *PET/CBM Personal Computer Guide*, third edition, (Osborne/McGraw-Hill). While specifically for the PET/CBM, much of the material is relevant to the Commodore 64.

THE LEN FUNCTION

Recall that while the maximum length of string that can be assigned to a string variable is 255 characters, the *actual* length of the string is the number of characters *currently* assigned to a string variable. BASIC provides a function to "count" and report the actual length of a string, or of a string assigned to a particular variable; a function appropriately called the LEN (for LENgth) function. LEN can be used in a print statement to print the number of characters in the string in question. Since the execution of LEN results in a numeric value, it can be assigned as a value to a numeric variable, used as a value in an IF . . . THEN comparison, or used in calculations.

For example:

```
10 LET G$ = "WHAT A GAS"
20 PRINT LEN(G$)
RUN
 10

100 PRINT LEN("NORTHERN MUSIC")
RUN
 14
```

```
10 LET H$ = "1582 ANCHORAGE DRIVE"
20 LET A = LEN(H$)
30 PRINT A
RUN
 20

150 LET R$ = "YES"
160 IF LEN(R$) = 3 THEN PRINT "GO ON TO THE NEXT QUESTION"
RUN
GO ON TO THE NEXT QUESTION

10 LET M$ = "AMERICAN"
20 LET N$ = "FOREIGN"
30 PRINT LEN(M$) + LEN(N$)
RUN
 15
```

A string to which no characters have been assigned is an empty string. Since an empty string has no characters, the LEN function would find a length of zero. If the computer tried to execute LEN(A$) when A$ had not been assigned to a string, then A$ string would be considered empty and the value returned by the LEN function would be zero.

At the top of the ASCII Code Chart (Appendix One), you will find a null character with the ASCII code number zero. The string length of a null character is zero (0).

The confusion can come from the term null string often being used to indicate an empty string, one to which no characters have been assigned, not even the null character with an ASCII code number of zero and a string length of zero.

Show the results of executing each of the following program segments:

(a)
```
10 LET C$ = " "
20 PRINT LEN(C$)

RUN
```

_____

(b)
```
10 LET F$ = "FRANK"
20 LET L$ = "JONES"
30 LET N$ = L$ + ". " +F$
40 PRINT N$
50 PRINT LEN(N$)

RUN
```

_____

_____

(a) 1

(b) JONES, FRANK
12

SUBSTRING FUNCTIONS: VERSATILE TOOLS TO MANIPULATE STRING DATA

Three Commodore BASIC string functions (MID$, RIGHT$, LEFT$) allow you to manipulate the parts of a string, called substrings. The MID$ function is by far the most useful substring manipulating function. It works for two different processes. It allows you to *select* substrings from within a larger string. The MID$ *selection* function has the following forms:

Example 1

```
100 PRINT MID$("CHARGE IT", 1, 6)

RUN
CHARGE
READY.
```

In Example 1, the MID$ function starts at position one (the C), and includes six characters total, thereby returning the substring CHARGE. The MID$ function thus selects characters 1 through 6 inclusive from the string CHARGE IT.

Example 2

```
200 LET TS = "NO SWIMMING ALLOWED"
210 PRINT MID$(T$, 4, 8)

RUN
SWIMMING
READY.
```

Example 2 has a string assigned to T$, and the substring starts at the fourth character and continues to the twelfth character (total of eight characters in the substring).

Example 3

```
10 LET D$ = "JERALD R. BROWN"
20 PRINT "DEAR MR. "; MID$(D$, 11)

RUN
DEAR MR. BROWN
READY.
```

Example 3 uses the abbreviated form of MID$, where only the start position for the substring is indicated. The substring therefore goes from the starting character and includes the rest of D$ to the end of the string. Notice how the substring is used as part of additional output to form the salutation for a letter.

Example 4

```
340 LET A = 7
350 LET C = 2
360 LET D = 4
370 LET W$ = "DON'T LOOK HERE"
380 LET Y$ = MID$(W$, A, C*D+1)
390 PRINT Y$

RUN
LOOK HERE
READY.
```

Example 4 shows that the starting position for the substring, as well as the number of characters to be included in the substring, can be represented by variables or expressions that evaluate to a numeric value. Of course, these variables must have been previously assigned values, just as the string variable must have been previously assigned a string. In this case, the substring has been assigned to a new string variable (Y$) which is used to display the substring selected by MID$.

So, in general, the MID$ function has the form:

> MID$(string variable or constant, substring starting position, how many characters in the substring from the start position).

Note that the three parameters in the MID$ function are separated by commas. The first is usually a string variable to which a string has previously been assigned. The second parameter is the starting position for the substring. The

third parameter *does not* tell the last character position number in the substring, but rather tells *how many characters total to include in the substring*—a point that sometimes confuses people.

The MID$ function can be used to replace part of an existing string. You must first determine which part of the string is to be replaced. This is the advantage of using a standard format, since you will always know the position of a certain data field within a longer string. For example, if you always coded your product string to contain the identification number as characters 1 through 6, the price as characters 7 through 11, and the description as characaters 11 through 40, you could change the price of any product by replacing characters 7 to 11. You first must break down the original string into parts preceding and following the place to insert the new data. This program shows one way to do it:

```
100 INPUT "NEW PRICE "; NP$
110 NP = VAL(NP$) : IF NP = 0 THEN PRINT "ENTER A
    NUMBER." : GOTO 100
120 IF NP > 99.99 THEN PRINT "MUST BE LESS THAN
    $100.00" : GOTO 100
130 A$ = "12345612.49STAINLESS STEEL BAGEL HOLDER"
140 LET L$ = MID$(A$,1,6) : LET R$ = MID$(A$,12)
150 NP$ = LEFT$("    ",5 - LEN(NP$) ) + NP$ : REM
    ADJUST TO 5 CHARACTERS
160 LET A$ = L$ + NP$ + R$
170 PRINT A$
180 END

RUN
NEW PRICE ? $3.27
ENTER A NUMBER.

NEW PRICE ? 327
MUST BE LESS THAN $100.00

NEW PRICE ? 3.27
12345 3.27STAINLESS STEEL BAGEL HOLDER
```

First, we get the new price and check the entry for type and value. Line 130 contains the old string, storing product number 123456, price $12.49, and product description. Line 140 splits the old string into a part preceding and a part following the section to be changed. Line 150 pads the price value with preceding spaces so it will take exactly five characters. Note that we are using the string version of the price, NP$. Finally, we put the string back together in line 160, and print the result.

This material will be covered in depth in the next chapter.

Notice the use of MID$ selection function in PRINT statements in the program below. Remember, it allows you to select and print any part or substring of the string assigned to the string variable in the MID$ parentheses. The other two values or parameters inside the parentheses still indicate where the substring to be printed starts and how many characters it includes. The string from which the substring is selected remains unchanged.

```
150 LET N$ = "FOGHORNE WHILDEFLOWER"
160 PRINT MID$(N$,1,8)
170 PRINT MID$(N$,10,12)
180 PRINT N$

RUN
FOGHORNE
WHILDEFLOWER
FOGHORNE WHILDEFLOWER
```

Notice the use MID$ as a *selection* function in lines 160 and 170 above. The selection function in no way changes the string assigned to N$. This is demonstrated by the output of line 180, even after substrings from N$ have been selected and printed by lines 160 and 170. This same selection function can be used to assign a substring from a string assigned to a string variable, *without* changing the original string from which the substring was selected.

Notice in the program segment below that a substring from an existing string can be assigned to a new variable without changing the string from which it was selected. F$ (for first name) and L$ (for last name) are selected from the entire name (N$) without changing N$.

```
150 LET N$ = "FOGHORNE WHILDEFLOWER"
160 LET F$ = MID$(N$,1,8)
170 LET L$ = MID$(N$,10,12)
180 PRINT N$
190 PRINT "FIRST NAME IS "; F$
200 PRINT "LAST NAME IS "; L$
```

(a) Show the RUN for the program segment above.
RUN

(b) Which character in N$ is not selected for inclusion in either F$ or L$?

(a)
```
RUN
FOGHORNE WHILDEFLOWER
FIRST NAME IS FOGHORNE
LAST NAME IS WHILDEFLOWER
```
(b) The space at character position 9 of N$.

The LEFT$ and RIGHT$ string functions are not as versatile as MID$ and are not used as much in our programming. However, they both work the same way as shown in these program segments:

```
160 PRINT LEFT$(A$,8)
```
means print the leftmost eight characters of A$ (the *first* eight characters in the string assigned to A$);

```
170 LET R = 12
180 LET B$ = RIGHT$(A$,R)
```
means assign to B$ the twelve rightmost characters of A$ (the *last* twelve characters in the string assigned to A$).

These examples demonstrate the substring selection capabilities of LEFT$ and RIGHT$. They are strictly *selection* functions, selecting one or more characters from one end or the other of an existing string to treat as a substring.

We often use LEFT$ for convenience to check for a user's YES or NO response to an INPUT prompting question. Using an IF . . . THEN statement, we have the computer look at the first character of the response string to determine whether or not the answer was YES, as shown in the following program segment:

```
240 INPUT "DO YOU NEED INSTRUCTIONS (YES OR NO)?"; R$
250 IF LEFT$(R$,1) = "Y" THEN 600
```

(a) What responses could a user make to the INPUT prompt above in order for the IF . . . THEN comparisons to be true?

(a) Could type YES or Y or any string that started with the uppercase letter Y.

Since you never know whether the person using your program on the Commodore 64 has the Caps Lock key in uppercase or lowercase mode, we try to cover the possibilities in statements that test the user's entries for specific responses. The following segment illustrates this.

```
240 input "Do you have more entries (y or n) "; r$
250 let r$ = left$(r$,1)
260 if r$ = "Y" or r = "y" then print "Continue
      data entry." : goto 280
270 if r$ <> "n" and r$ <> "N" then print "Enter
      yes or no." : goto 240
280 rem continue

run
Do you have more entries (y or n) ? Sure
Enter yes or no!
Do you have more entries (Y or n) ? Yes
Continue data entry.
Ready.

run
Do you have more entries (Y or N) ? N
Program ends
Ready.
```

Because of the wraparound effect when entering long BASIC statement lines, a multiple statement line such as lines 260 and 270 above is not particularly easy to "disassemble" into its parts (a long IF . . . THEN statement, a couple of PRINT statements, and a GOTO) for easy understanding. When used often, as this routine might be, such one-line modules may be more tempting than an alternative multiline program segment that performs the same job. The segment below is simpler and more easily interpreted by someone trying to understand a program's functioning from a LISTing.

```
240 input "Do you have more entries (Y or N) "; a$
250 let a$ = left$(a$,1)
260 if a$ = "Y" or a$ = "y" then 410 : rem continue
270 if a$ = "N" or a$ = "n" then 310 : rem stop
      entry
280 print "Enter 'Y' for yes or 'N' for no!" : goto
      240
300 :
310 print "Program ends." : end
410 print "Continue data entry."
RUN
DO YOU HAVE MORE ENTRIES (Y OR N) ? SOME
ENTER 'Y' FOR YES OR 'N' FOR NO!
DO YOU HAVE MORE ENTRIES (Y OR N) ? YES
CONTINUE DATA ENTRY
READY.

RUN
DO YOU HAVE MORE ENTRIES (Y OR N) ? N
PROGRAM ENDS
READY.
```

(Chapter 3 deals with this kind of entry checking in detail.)

We have found less use for the RIGHT$ function than for MID$ or for LEFT$, but here is an example. Remember, the numeric value inside the RIGHT$ function's parentheses means to start counting the characters for the substring at the rightmost end of the string from which the substring is being selected, counting toward the beginning of the string.

```
240 INPUT "WHICH HIGH SCHOOL CLASS DID YOU GRADUATE FROM?"; Y$
250 PRINT "YOU GRADUATED IN 19"; RIGHT$(Y$,2)
```

Assume that several people responded to the input prompting question when the above program segment was run. Show what the computer will print for each user's response.

(a) User responds: CLASS OF 1938

Line 250 prints: _____

(b) User responds: CLASS OF '64

Line 250 prints: _____

(c) User responds: 1958

Line 250 prints: _____

(d) User responds: FORTY EIGHT

Line 250 prints: _____

(a) YOU GRADUATED IN 1938
(b) YOU GRADUATED IN 1964
(c) YOU GRADUATED IN 1958
(d) YOU GRADUATED IN 19HT

FOR NEXT STATEMENTS

It is preferable to use a FOR NEXT loop when you have a controlled, repeating sequence of instructions.

Preferred

```
100 FOR X = 1 TO N
110 PRINT X, X^2
120 NEXT X
```

Undesirable

```
100 LET X = 1
110 PRINT X, X^2
120 LET X = X + 1
130 IF X > N THEN 200
140 GOTO 110
```

As you can see, the FOR NEXT loop is more space efficient (it could even have been done in one line), it looks better, and it is easier to read.

A general rule when using FOR NEXT loops is: *do not exit* from the middle of a FOR NEXT loop, except to GOSUB to a subroutine. Leaving the controlled loop makes the program difficult to read and hard to understand. In some circumstances where the loop is encountered many times, exiting from it repeatedly may result in an "?OUT OF MEMORY ERROR." This is almost only found when using GET loops.

An exit to a subroutine is acceptable because a subroutine will RETURN the program to the *inside* of the FOR NEXT loop to continue in sequence, as if there were no exit at all.

You can usually write your program to include everything you need to do inside the loop, rather than leaving the loop. (There will be exceptions.) For more information, look for a book on *Structured Programming* in BASIC. There is a big advantage in reading and debugging a program where this practice has been followed.

Undesirable

```
100 FOR X = 1 TO N
110 IF A(X) = B(X) THEN 130
120 NEXT X
130 LET S = S + 1
140 GOTO 120
```

Preferred

```
100 FOR X = 1 TO N
110 IF A(X) <> B(X) THEN 130
120 LET S = S + 1
130 NEXT X
```

(a) Write a program segment using nested FOR NEXT loops that will print the word HELLO three times, but will print the word GOODBY four times after each appearance of the word HELLO.

(a)
```
10 FOR X = 1 TO 3
20 PRINT "HELLO"
30 FOR Y = 1 TO 4
40 PRINT "GOODBY"
50 NEXT Y
60 NEXT X
```

STRING SEARCHES WITH MID$

MID$ can be used to search within a string for the occurrence of a particular substring, and to pinpoint the location of the substring. In effect, the string is checked character by character until the substring is found. The counter used in the FOR NEXT loop indicates the character position of the *first* character of the substring within the string being searched. Here is an example:

```
11 REM STRING SEARCHES
200 LET S$ = "SAMPLE"
210 LET F$ = "PL"
220 FOR I = 1 TO 6
230 IF MID$( S$, I , 2 ) = F$ THEN 250
240 NEXT I : END
250 PRINT I, MID$( S$, I, 2 )
```

(a) In the program above, the substring is two characters long. How can the value of I in line 250 be 4?

(a) The value of I corresponds to the character position of the first character in the substring. P is in position 4 in "SAMPLE," thus the result of the search is 4.

This simple program will search through the word "SAMPLE" for the letters PL. The FOR NEXT loop sets the starting position, so we start by looking at the first two characters of S$, then the second and third, then the third and fourth, etc. The 2 in the MID$ lets us look at the same number of characters in S$ as there are in F$. Here, F$ has two characters, so the MID$ function looks at the characters of S$ two at a time.

We can generalize this program so S$ and F$ could be of any length, by replacing the numbers we used in the example above with LEN functions, as shown in lines 230 and 240 of the example below. Here is the more general version:

```
200 REM GENERAL SEARCH FOR SUBSTRING
210 LET S$ = "PRACTICE"
220 LET F$ = "ACT"
230 FOR I = 1 TO LEN(S$) - LEN(F$) + 1
240 IF MID$( S$, I , LEN(F$) ) = F$ THEN 270
250 NEXT I : END
260 PRINT "SEARCH STRING NOT FOUND" : END
270 PRINT "SEARCH STRING FOUND AT " I
```

Line 230 needs explaining. The string search can stop before the end of S$, because we are looking at several characters at a time. The stopping point before the end of S$ is the number of characters in F$ [LEN(F$)] plus one. LEN(S$) – LEN(F$) + 1 causes the search to stop at the last set of F$ characters in S$. Since we are looking for the characters of F$, we must tell the MID$ function to look, all at once, at the number of characters in F$. The MID$ function we set up looks at the number of characters in F$ [LEN(F$)], starting at the Ith character of S$. The FOR NEXT loop moves I through the word, with the result that we look at sets of three characters [the LEN(F$)]. First MID$ gives us PRA, then RAC, then ACT.

Here's another sample program that prints out the results of the MID$, so you can see it working. Type this program in, and run it with different search words or letters, to see what happens.

```
300 REM DEMONSTRATION SEARCH
310 LET S$ = "COMPUTER"
320 INPUT "TYPE A SEARCH STRING "; F$
330 FOR I = 1 TO LEN(S$) - LEN(F$) + 1
340 PRINT MID$( S$, I , LEN(F$) )
350 IF MID$( S$, I , LEN(F$) ) = F$ THEN 380
360 NEXT I
370 PRINT "SEARCH STRING NOT FOUND." : GOTO 320

380 PRINT "SEARCH STRING FOUND AT " I
390 GOTO 320
```

Experiment with shorter and longer words, with strings which don't occur in "COMPUTER," and with other ideas that occur to you. The ability to search for substrings is critical to data manipulation, and it's worth your time to play with this program until you are comfortable with the ideas.

(a) Why do we skip to line 380 if we get a match?

(a) To avoid printing the message "SEARCH STRING NOT FOUND." If no match is made, the program will continue after the NEXT I, taking us to line 370.

Here's an example of a use of the MID$ search to return the number of a month, when given the name.

```
410 LET M$ = "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOV
    DEC"
420 INPUT "NAME OF MONTH "; M1$
430 LET M1$ = LEFT$( M1$, 3 )
440 FOR M = 1 TO LEN(M$) - 2
450 IF MID$( M$, M, 3 ) = M1$ THEN 480
460 NEXT M
470 PRINT "MONTH NOT FOUND, TRY AGAIN." : GOTO
    420
480 N = (M + 2) / 3 :REM CALC. MONTH FROM POSIT
    ION
490 PRINT M1$ " IS MONTH NUMBER " N
500 INPUT "AGAIN "; Z$
510 IF LEFT$( Z$, 1 ) = "Y" THEN 420
520 END

RUN
TYPE NAME OF MONTH ? APRIL
APR IS MONTH NUMBER 4
AGAIN ? YES
TYPE NAME OF MONTH ? NO
NO IS MONTH NUMBER 11
AGAIN ? YES
TYPE NAME OF MONTH ? QUIT
MONTH NOT FOUND, TRY AGAIN.
TYPE NAME OF MONTH ? A
A IS MONTH NUMBER 1.3333333
AGAIN ? N
READY.
```

Note that even if you enter NO, the program finds NOVEMBER and returns 11. If you enter just A, the program responds with the first occurrence, which is at position two of M$, in JAN, and we get a nonsense result. We could have checked for this in our program.

(a) What does line 430 do in this program?

_____

_____

_____

(a) Prevents us from entering more than three characters. Since our months are stored as three characters, searching for more than three will result in no match. Limiting the search to three guarantees a match if the proper data (name of a month) is entered.

Here's a program that looks for the first space in a string to divide it into substrings:

```
740 LET F$ = "JANE FONDA"
750 FOR S = 1 TO LEN(F$)
760 IF MID$(F$,S,1) = " " THEN 780 : REM MATCH
     SPACE
770 NEXT S : END
780 PRINT MID$(F$,1,S-1)
```

The REM in line 760 is helpful to tell a reader what really is in that " ". Here's another way, using ASCII as discussed earlier:

```
20 REM ALTERNATE WAY USING CHR$
740 LET F$ = "JANE FONDA"
750 FOR S = 1 TO LEN(F$)
760 IF MID$(F$,S,1) = CHR$(32) THEN 780 : REM M
     ATCH SPACE
770 NEXT S : END
780 PRINT MID$(F$,1,S-1)
```

(a) What is the upper limit for S (the FOR control variable)? _____

(b) What is the length of the substring selected by the MID$ function in line 760? _____

(c) What is the length of the first name substring in F$? _____

(d) In what character position in F$ is the space? _____

(e) Why doesn't S-1 in the MID$ function in line 780 cause one character in the name to be lost? _____

_____

(f) What is printed by line 780? _____

_____

(g) Why is the "END" in line 770? _____

_____

_____

(a) LEN(F$) = 10.
(b) One character.
(c) Four characters.
(d) Character position 5.
(e) Because S gives the character position of the first space, not the last letter, in the F$ "first name" substring.
(f) JANE
(g) The END prevents the program from printing a result when no match is found.

Write a string search program to look for a decimal point (.) entered in a numeric string. Assume the string is N$, and may vary in length. Here's what a run of your program should produce:

```
YOUR NUMBER IS ? $123
PLEASE ENTER YOUR NUMBER WITH A DECIMAL POINT.

YOUR NUMBER IS ? $123.00
YOUR NUMBER INCLUDES A DECIMAL POINT.

YOUR NUMBER IS ?
ENTER A NUMBER, PLEASE.

YOUR NUMBER IS ? $.02
YOUR NUMBER INCLUDES A DECIMAL POINT.

YOUR NUMBER IS ? 2.601
YOUR NUMBER INCLUDES A DECIMAL POINT.
```

```
100 REM CHECK FOR DECIMAL POINT
110 INPUT "YOUR NUMBER IS ";N$
120 IF N$ = "" THEN PRINT "ENTER A NUMBER,
      PLEASE.": GOTO 110
130 FOR I = 1 TO LEN(N$)
140 IF MID$( N$, I, 1 ) = "." THEN 170
150 NEXT I
160 PRINT "PLEASE ENTER YOUR NUMBER WITH A DECIMAL
      POINT." : GOTO 110
170 PRINT "YOUR NUMBER INCLUDES A DECIMAL POINT." :
      GOTO 110
```

MULTI-BRANCHING WITH ON . . . GOTO

The ON . . . GOTO statement allows the computer to branch to a number of different statements depending upon a test or value. The format for the statement is a list of line numbers:

```
ON X GOTO 310, 450, 660, 660, 660, 720, 830, 510
```

Note: X is any variable or expression from which a VALUE will result.

If the value of X is 1 when the ON . . . GOTO statement is encountered and executed, the computer branches (goes to) the first line number in the list of line numbers (in our example, line 310). If the value of X is 2, the program branches to the second line number in the list. As many line numbers can follow GOTO as will fit in a statement line (80 characters maximum). Notice in our example that if X is 3, 4, or 5, the program will branch to the same place (line 660). The line numbers following the GOTO do not have to be in ascending order.

If the value of X is zero, or larger than the number of line numbers in the list, then the ON . . . GOTO statement will be skipped and the next statement in normal line number order will be executed. (Note that if you have multiple statements on one line, the statement following the ON . . . GOTO will be executed, as usual in BASIC.) A negative value, or a value greater than 255, will cause an "ILLEGAL QUANTITY ERROR."

Here is a technique using ON . . . GOTO to determine which choice has been made by a user. The situation could be a "menu" of choices from which the user must select one, or a multiple choice question where the user selects one response. In order to determine where we are in the string "WACPS" (line 420), we need a counter, which is provided by the FOR NEXT loop (lines 410, 430). When a match is found in line 420, the counter I has a value of 1 through 5 depending upon the position in the string "WACPS". This value is used by the ON . . . GOTO in line 440.

```
250 REM MENU PROGRAM DEMO
260 INPUT "DO YOU NEED INSTRUCTIONS ";R$
270 IF R$ <> "YES" THEN 500
280 :
290 :
300 REM MENU SET
310 PRINT "[CLR]":  REM CLEAR SCREEN
320 PRINT "THESE ARE YOUR CHOICES: "
330 PRINT "W    BEGIN WRITING TO NEW FILE"
340 PRINT "A    ADD TO EXISTING FILE"
350 PRINT "C    CHANGE EXISTING FILE"
360 PRINT "P    PRINT CONTENTS OF EXISTING FILE"

370 PRINT "S    SELECT DATA FROM FILE
380 :
390 INPUT "ENTER YOUR CHOICE "; R$
400 LET R$ = LEFT$( R$, 1)
410 FOR I = 1 TO 5
420 IF R$ = MID$( "WACPS", I,1) THEN 440
430 NEXT I : PRINT "ENTRY ERROR--TYPE W,A,C,P,
     OR S" : GOTO 390
440 ON I GOTO 500, 600, 700, 800, 900
450 :
460 :
```

Here is an alternate method to arrive at an ON . . . GOTO value in a "menu" selection situation. In the following program segment, the ASC( ) function is used to convert a letter entered by the user to an ASCII value that is used to determine the value for an ON . . . GOTO statement. The ON . . . GOTO is a multi-branching instruction. In line 260, if the value of R is 1, then the program goes to the first line number given after GOTO. If R = 2, then the program branches to the second line number given, and so on. The value of R must be greater than 1 and no higher than the number of line numbers that follow GOTO. This requires using A, B, C, etc., for the menu choices, rather than the perhaps more helpful mnemonic letters used in the example above.

```
250 REM MENU PROGRAM DEMO
260 INPUT "DO YOU NEED INSTRUCTIONS ";R$
270 IF R$ <> "YES" THEN 500
280 :
290 :
300 REM MENU SET
310 PRINT "[CLR]":  REM CLEAR SCREEN
320 PRINT "THESE ARE YOUR CHOICES: "
330 PRINT "A    BEGIN WRITING TO NEW FILE"
340 PRINT "B    ADD TO EXISTING FILE"
350 PRINT "C    CHANGE EXISTING FILE"
360 PRINT "D    PRINT CONTENTS OF EXISTING FILE"

370 PRINT "E    SELECT DATA FROM FILE
380 :
390 INPUT "ENTER YOUR CHOICE, A-E"; R$
400 LET R = ASC(R$ + CHR$(0)) - 64
410 IF R < 1 OR R > 5 THEN 430
420 ON R GOTO 500, 600, 700, 800, 900
430 PRINT "ENTRY ERROR.  PLEASE ENTER A,B,C,D,O
     R E" : GOTO 390
440 ON I GOTO 500, 600, 700, 800, 900
450 :
460 :
```

(a) In the program above, why is line 410 included:

_____

_____

(a) If R evaluates to less than 0 due to a data entry error, or larger than 5, an error would occur; so the checking is done by line 410.

Note that line 410 isn't necessary if the R processes to a value of 0 or 6 through 255. However, if line 410 were omitted and the entry was "6," with an ASCII value of 54, R would be less than 0, giving an "ILLEGAL QUANTITY ERROR" in line 420 and stopping the program. The "+ CHR$(0)" is necessary in case RETURN is pressed without any entry on the Commodore 64, in which case R$ will be an empty string. If the "+ CHR$(0)" weren't included, the program would stop with an error message. Careful programmers anticipate such unlikely events and provide for them in their programs.

DATA ENTRY USING GET

The Commodore BASIC command GET can be used to get entries from the keyboard that might include commas, quotation marks, and other punctuation which cannot be accomodated with the INPUT statement. GET assigns the first character in the keyboard buffer to a variable. If a key has been pressed prior to the GET statement, the character represented by that key will be assigned to the variable. If no key has been pressed, the empty string ("") will be assigned. Keys pressed along with the SHIFT (or Commodore key) are seen as only one character. Since GET works "on the fly" it is often included within a loop to wait until an appropriate key has been pressed.

In its simplest form, the GET statement is used to act on the basis of a single keystroke. It is most commonly used to pause until the user is ready to continue. Here's how it is used:

```
100 print "Press RETURN to continue."
110 get z$ : if z$ <> chr$(13) then 110
```

The GET in line 110 checks the keyboard. The ASCII value of the last key pressed is compared with chr$(13) (RETURN). If RETURN hasn't been pressed, the program returns to line 110. When RETURN is pressed, the comparison is false, and the program continues.

GET is also commonly used for Y/N responses to yes or no questions, or for a single number or letter response to a series of choices, such as a menu. Here are some examples:

```
100 PRINT "DO YOU WANT INSTRUCTIONS ? "
110 PRINT "PRESS 'Y' FOR YES OR 'N' FOR NO."
120 GET Z$ : IF Z$ = "Y" THEN 900
130 IF Z$ <> "N" THEN 120
140 REM PROGRAM CONTINUES
```

The GET in line 120 picks up the value of any key pressed. If Y was pressed, the program branches to line 900; if not, it continues to line 130. If there is no keypress, Z$ contains an empty string, which is not equal to N, and the program returns to line 120.

Here's another example:

```
100 PRINT "A  ADD NEW FILE"
110 PRINT "C  CHANGE A FILE"
120 PRINT "D  DELETE A FILE"
130 PRINT "L  LOOK FOR A FILE"
140 PRINT "S  SAVE DATA"
150 PRINT "Q  QUIT PROGRAM"
160 GET Z$ : IF Z$  = "A" THEN 300
170 IF Z$  = "C" THEN 400
180 IF Z$  = "D" THEN 500
190 IF Z$  = "L" THEN 600
200 IF Z$  = "S" THEN 700
210 IF Z$  = "Q" THEN 800
220 GOTO 160
```

Here is the routine to accept input. It provides for the use of the DELete key, and will accept any character until a RETURN is pressed. In that regard, it mimics the INPUT statement. Since GET does not display a cursor, this routine provides for a flashing cursor.

```
10000 REM GET FOR INPUT
10010 ZY$ = "" : FZ = 166
10020 PRINT CHR$(FZ)"[LEFT]"; :  FOR Z1 = 1 TO 30
      : NEXT Z1
10030 GET ZZ$ : IF  ZZ$ = "" THEN  FZ = 198 - FZ
     : GOTO 10020
10040 ZZ = ASC(ZZ$) : IF ZZ = 13 AND LEN(ZY$) > 0
       THEN 10090
10050 IFZZ=20ANDLEN(ZY$)>0THENZY$=LEFT$(ZY$,LEN(Z
     Y$)-1):PRINT" [LEFT][LEFT] [LEFT]";:GOTO1002
     0
10055 IF ZZ =  34 THEN ZZ = 39 : REM CHANGE QUOTE
     S TO SINGLE QUOTE
10060 IF ZZ < 32 OR ZZ > 223 THEN 10020
10070 IF ZZ > 95 AND ZZ < 193 THEN 10020
10080 PRINT CHR$(ZZ); : ZY$ = ZY$  + CHR$(ZZ) : G
     OTO 10020
10090 PRINT : RETURN
```

Somewhat arcane variable names were used to allow this to be used as a subroutine without modification. Line 10010 initializes the string ZY$ which will contain the input to an empty string. FZ is used to flash the "cursor" by alternately printing a checkerboard and a normal space. CHR$(166) is the checkerboard, and 198−166 = 32, ASCII for space. This is followed by a CURSOR LEFT, to move back to the same space. This is not the backarrow key, but the SHIFTed CRSR key. A timing loop flashes the "cursor."

Then GET checks the keyboard. If no key has been pressed, we repeat. If a key has been pressed, we get the ASCII value, and if it's a RETURN (ASCII 13), we go to the end.

Line 10050 checks for a DELETE (ASCII 20), uses LEFT$ to take the last character off of the string, and then backs up to print a space over it on the screen. Lines 10060 and 10070 check that the input is just letters, numbers, or punctuation. (These lines can be modified to accept graphics characters, or to not accept SHIFTed characters.)

Finally, line 10080 prints the character typed and adds it to the string. Note that line 10050 must be typed without spaces, and you must use keyword abbreviations (? for PRINT—see Appendix 2) to fit it into an eighty-character line.

Here's how this would typically be used:

```
100 REM SAMPLE
110 PRINT "ENTER NAME AS LAST, COMMA, FIRST: "
120 GOSUB 10000 : REM GET
130 N$ = ZY$ : REM USE VALUE RETURNED
140 PRINT "ENTRY WAS: " N$
```

Note that there will be no question mark displayed (as with INPUT) unless you put it in your PRINT statement prompt. Also, since the input is returned as ZY$, you may wish to equate your variable to ZY$ to use the input, as in lines 130 and 140.

USING THE UNIQUE FEATURES OF COMMODORE COMPUTERS

EIGHTY-CHARACTER LINES

Commodore allows program lines to be up to eighty characters long—two screen lines. This is done by checking for a character in the thirty-ninth location of the first line. If anything is there, the two lines are connected. This can play havoc with logic in two circumstances:

1. editing a line and
2. printing prompts in INPUT statements.

When editing, the solution is simple—list ONLY the line you want to edit (for example, LIST 210). Connected lines give unexpected ?SYNTAX ERRORs, since the line number in the second line mysteriously appears (for the computer) in the middle of a program line, even though it looks like two separate lines on the screen. However, if you LIST the line with the ?SYNTAX ERROR, you will see two lines appear on the screen. You will need to edit or retype them as two separate lines.

The problem also occurs when using prompts with INPUT statements. On the Commodore 64, if a prompt continues past the thirty-ninth character, it links the two lines together. Suddenly, the computer connects your prompt and the user's answer. As a result, your program may choke or quit. Here is an example:

```
400 print : input "Did you really mean a zero value
      (yes or no)"; r$
410 let r$ = left$(r$,1)
420 if r$ = "Y" or r$ = "y" then 340
430 if r$ = "N" or r$ = "n" then print "Reenter
      correct value." : goto 310
440 print "Please enter 'Y' for YES or 'N' for NO."
      : goto 400
```

Type in the program, RUN it, and try different answers. It seems no matter what you type, you keep getting the message "Please enter 'Y' for YES or 'N' for NO." To see what is happening, add a line to print the contents of r$:

```
405 print : print r$
```

You will see the contents of r$ include the entire prompt, the question mark, and whatever you have typed. Since the prompt begins with D, line 410 always results in r$ = "D," line 440 is printed, and the program is stuck in an endless loop.

There are two solutions. One is to just shorten the prompt to less than thirty-eight characters. This problem occurs ONLY when the prompt is longer than thirty-eight characters; the length of the user's response is not a factor. In the example above, we could change the prompt to:

```
400 print : input "Is ZERO o.k. (Yes or No) "; r$
```

Since it is only twenty-seven characters long, it works fine. The other solution is to place the prompt in a PRINT statement and the INPUT on a separate line. Then there is no problem, no matter what length the prompt is.

```
400 print "Did you really mean a zero value (yes or no)"
405 input r$
```

While this leaves the '?' from INPUT on a line by itself, it avoids the thirty-ninth character problem, and allows the user to enter up to forty characters without having the response wrap from one line onto the next.

One final comment on the matter of screen lines. We have printed lines in this book to be easy to read on paper; however, many of them will split words at the edge of your screen, which will make your programs hard to read and make them look very unprofessional. As you use and revise these programs, or write your own, consider keeping PRINT statements to forty characters, so everything is printed neatly to the screen and no words are split at the edge of the screen. You may also have to insert extra spaces, or leave out spaces to achieve the best effect. Believe us, it is worth the effort if others will be using your programs.

REVERSE FIELD (RVS)

Reverse field allows you to highlight information, to call your user's attention to specific information, to an error condition, or simply to emphasize a word. Because the characters for REVERSE FIELD ON (RVS) and REVERSE FIELD OFF (OFF—SHIFT-RVS) are difficult to read, in the programs in this book we have instead used the apostrophe (') to set off information (PRESS 'RETURN' TO CONTINUE.). Here are some examples of where you might use reverse field:

```
120 IF C <> 5 THEN PRINT "[RVS]USE EXACTLY FIVE
      CHARACTERS."
150 :
200 PRINT "PRESS [RVS]RETURN[OFF] TO CONTINUE."

340 :
350 PRINT "YOU ENTERED [RVS]" A$
750 :
760 PRINT "PRESS [RVS]D[OFF]ELETE, [RVS]C[OFF]H
     ANGE, OR  [RVS]I[OFF]NSERT."
```

Here is how the examples would look with apostrophes, as we have done in the rest of the book:

```
200 PRINT "PRESS 'RETURN' TO CONTINUE."

350 PRINT "YOU ENTERED '" A$

760 PRINT "PRESS 'D'ELETE, 'C'HANGE OR 'I'NSERT."
```

The last form is particularly useful with menus, when you want your user to press one key corresponding to the choice—here, a D to delete, a C to change, or an I to insert. Using REVERSE FIELD to highlight the letters helps remind the user, but the entire word is there, so the user is not confronted with "Press A, B, C, D, or E" without any clue as to what the letters mean.

Highlighting should be used sparingly, but can help make your programs easier to use and read.

UPPER/LOWERCASE

Most of the examples in this book are in uppercase only (all capitals), since that is the mode when the machine powers up, and it is easier to read. However, for serious business programs, you will want to use the upper/lowercase mode. There are two ways to switch from one to the other. The easiest is to hold down the SHIFT key and press the COMMODORE (C<) key. Repeating this will return you to the uppercase/graphics mode.

In a program, it is more reliable to change the machine, rather than relying upon the user to press the correct keys. How this is accomplished depends upon whether you are using a PET/CBM or a Commodore 64. The first one would be used at the beginning of a program, the other one at the end. Which order you use them in depends on which mode you want to use for your program.

For the Commodore 64, the statements are

```
20 PRINT CHR$(14) : REM UPPER/LOWER
25 PRINT CHR$(8) : REM DISABLES SHIFT/COMMODOR
   E
920 PRINT CHR$(142) : REM UPPER/GRAPHICS
925 PRINT CHR$(9) : REM ENBLES SHIFT/COMMODORE
```

Of course, both statements could be combined as PRINT CHR$(14) CHR$(8).

For PETs and CBMs, the statements are

```
20 POKE 59468, 14 : REM UPPER/LOWER
920 POKE 59468, 12 : REM UPPER/GRAPHICS
```

See Appendix 4 and your Commodore reference manuals for more specific information. While the following is sometimes used for the Commodore 64, it is not as reliable as the CHR$ method given above. We have used it in this book because of its parallel with the PET/CMB version:

```
20 POKE 53272, 23 : REM  UPPER/LOWER
920 POKE 53272, 21 : REM UPPER/GRAPHICS
```

When you type in a program, it doesn't matter to the computer which mode is being used. However, when you type BASIC keywords (PRINT, INPUT, etc.), you *must* type them WITHOUT USING THE SHIFT KEY. If you are in upper/graphics mode, they will be in uppercase; if you are in upper/lower mode, they will be in lowercase. If you use the shift key while typing BASIC keywords, strange things will happen! You should only use the SHIFT key when typing characters INSIDE QUOTES—those messages that will be printed on the screen or printer, where you wish to use capital letters (or the graphics symbols).

The only exception is the use of the SHIFT key for various punctuation symbols on the keyboard (", #, $, etc.), and when using the screen editing keys (CLR/HOME, CRSR, INST/DEL). You can always check by listing your program.

Remember—if you want a message in a PRINT or INPUT statement to have capitals, you must use the shift key (in upper/lower mode). *Never* use the shift key when typing BASIC keywords. For that reason, we recommend that you don't use the SHIFT-LOCK key; it's too easy to forget and type the wrong things in SHIFTed mode.

Additional valuable information specific to your Commodore computer can be found by reading the appendices.

MULTIPLE-STATEMENT LINES

Some language features in Commodore BASIC are *not* available on other computer systems. Some of these features speed up the program's run time, others save memory space, and some do both. Some features enhance program readability while others confuse the reader. A popular feature is the ability to place multiple BASIC statements on one line separated by a colon, as we showed earlier in discussing IF . . . THEN.

```
140 FOR X = 1 TO 10: PRINT X, X↑2: NEXT X
```

or

```
200 IF X=Y THEN PRINT "YOU WON!": GOTO 10
210 PRINT "SORRY, WRONG NUMBER": GOTO 60
```

A few cautions and suggestions are applicable as you use multiple-statement lines:

1. Multiple-statement lines are hard to read and sometimes hard to understand. If you later change a program, readability may be a problem. It is more clear to use one statement to a line.
2. If you must use multiple-statement lines, carry out a complete procedure or action on *one* line, whenever possible. Carryover to other lines makes reading more difficult and less clear.
3. Finding program errors buried in multiple-statement lines is difficult.
4. Understand completely how IF . . . THEN statements work in a multiple-statement line. In line 200 above, if X *does* equal Y, then "You won" will be printed and the program will branch to line 10. If the X = Y condition is false, line 210 will be executed next. Some people incorrectly presume that GOTO 10 will be executed whether the condition is true or false.
5. REMARK statements must be the *last* statement on a multiple-statement line. Any executable statement after a REMARK will *not* be executed.

Special consideration of the GOSUB statement in multiple-statement lines is warranted. Remember that each GOSUB statement must have a corresponding RETURN statement that appears as the last statement in the subroutine the GOSUB branches to.

Say, for example, a GOSUB is executed when an IF . . . THEN condition is true. After completing the subroutine, the computer must always be instructed to RETURN. The statement it returns to will be:

1. the next statement after GOSUB if it is a multiple-statement line, or
2. the next line numbered statement in normal line number order.

(a) Assume that the comparison in line 120 below is true and the GOSUB statement is executed. Which statement will be executed next after the RETURN from subroutine execution? _____

```
120 IF X = 2 THEN GOSUB 510 : GOTO 360
130 PRINT "X IS LESS THAN TWO"
```

(a) GOTO 360 is the next statement executed after the RETURN.

GET TO KNOW YOUR COMMODORE 64

To stop a program run, simply press the RUN/STOP key. If that doesn't work, or when the program is at an INPUT statement, holding down the RUN/STOP key then pressing the RESTORE key will stop the program and clear the screen. You may need to repeat this several times. (For the PET, hold the SHIFT key and press the RUN key.)

We suggest you review the section on the keyboard in the Commodore User's Guide and learn the uses of the Control (CTRL) and Commodore (C<) keys. Review the section on Editing Tips. You will also find it worth your time to learn how to use the extremely powerful Commodore screen editor. It will save you time and effort, and make entering and correcting programs easier.

This is also a good time to become familiar with the operation of your Datasette or Disk Drive. Review LOADing and SAVing. If you have a disk drive, learn how to NEW the disks to prepare them, and to INITIALIZE the disk drive. Always practice with a "scratch" tape or disk you can afford to erase and reuse in case you make a fatal mistake or your computer doesn't do what you expected.

Chapter 2 Self-Test

1. Why do the authors recommend using "greater than" and "less than" comparisons in IF . . . THEN numeric comparisons, rather than comparisons for equality?

   _____

   _____

   _____

2. When must quotation marks be placed around string data items in a DATA statement?

   _____

   _____

_____

3. How can an empty string be assigned to an INPUT string variable?

4. Show the results of a RUN of the following program:

```
10 LET A$ = "ALFRED"
20 LET B$ = "CONTRACT"
30 LET C$ = "32C"
40 PRINT ASC(A$), ASC(B$), ASC(C$)
RUN
```

5. Describe the string that must have been assigned to D$ for each of these comparisons to be true:
   (a) `IF ASC(D$) < 48 OR ASC(D$) > 57 THEN 660`
   (b) `IF ASC(D$) > 64 AND ASC(D$) < 91 THEN GOSUB 1520`

   (a)

   (b)

6. What value will the LEN function show for a string to which fifteen spaces have been assigned?

7. Show the RUN for the following program:

```
10 LET M$ = "STAR TREK"
20 LET N$ = "WARS"
30 LET G$ = MID$(M$,1,5) + N$
40 PRINT G$
50 PRINT M$
RUN
```

8. Give an example of a simple numeric variable and a simple string variable.

9. Give a reason for avoiding multiple statements in one program line.

_____

_____

_____

10. Examine the following statement:

```
120 IF X > 10 THEN GOSUB 810 : GOTO 110
```

After executing the subroutine starting at line 810, to which statement will the computer return?

_____

Answer Key

1. Round-off error in the computer's computational process may introduce tiny errors that make expected values slightly more or less. Therefore, an equality comparison may fail where you would expect it to succeed.
2. When the string data item includes a comma as part of the string or leading spaces are to be included as part of the string.
3. By pressing the RETURN key without entering anything else from the keyboard.
4. 65 67 51
5. (a) First character of D$ must *not* be a number (0 to 9).
   (b) First character of D$ must be a capital letter (A to Z).
6. Fifteen (spaces count as characters in a string).
7.
```
RUN
STAR WARS
STAR TREK
```
8. Numeric variable: A (or any letter of the alphabet); string variable: A$ or any letter of the alphabet followed by a dollar sign.
9. May make it harder to read the program; may make errors in programming harder to detect (either answer).
10. GOTO 110

CHAPTER THREE

Building Data Entry and Error Checking Routines

Objectives: When you finish this chapter, you will be able to write statements in a data entry program module to check the following aspects of data items:

Proper length
Non-response
Type of data (numeric or alphanumeric)
Inadvertant inclusion of wrong characters
Parameters for numeric data

In addition, you will be able to write data entry modules that:

Have clearly stated prompts.
Use reasonable data fields.
Concatenate data items into a single field.
Check and "pad" entries, as necessary, for proper field length.
Remove excess spaces from data taken from data fields.
Replace data items contained in a data field.
Provide complete explanations of a data entry error to the user.

INTRODUCTION

If you are wondering when you are going to get into data files themselves, be patient. Experience has shown that you need a good background in some special techniques associated with data file programming that use BASIC statements you already know. This will make it much easier and faster to learn the new BASIC statements and functions specifically applied to data file handling. You shouldn't have to struggle to understand a new use for a familiar BASIC statement while trying to absorb the data file statements and techniques, so please don't gloss over this material.

Concern for data entry procedures was introduced in the section on INPUT in the previous chapter. For our purposes *data* are defined as any information that is or will be stored in a data file on disk or cassette. Common examples of data include mailing, subscription, or billing lists; inventories of retail merchandise; accounting information; files of books, recordings, journal articles, or notes for a book; statistical information. *Data entry* includes the process of getting such information into the computer so that it can be stored in a data file. *Data files* usually contain large amounts of data that, to be useful, must be accurate, valid, and error-free in content and format. The accuracy and usefulness of your program output depends entirely on the accuracy of the data in these files. Furthermore, inaccurate or invalid data in a data file (or any place in a program) can cause your program to interrupt, halt, or abort in an error condition in the middle of its run. If your program terminates unexpectedly, there may be no telling what is happening inside the computer. Printed reports can be only partially completed, entered data can be lost or destroyed, data in files can be half processed; the list goes on.

The result of an unexpected program interruption can be catastrophic, though it may not always be so. It is almost impossible to predict exactly what will happen. Therefore, always do everything you can in your programming to avoid errors that can precipitate program interruptions.

Unfortunately, most errors occur at data entry time. This is why we emphasize the use of data entry checking procedures in this chapter—procedures to guarantee that data are entered as clean, valid, and accurate in content and format as your ingenuity and knowledge of programming techniques can make them. Throughout the remainder of this book "error-traps" and places where programming errors are likely to occur are illustrated.

This chapter focuses on constructing the data entry module of a program. This is where, usually with INPUT statements, the computer user is instructed to type in information that is going to be placed in a new data file, or to tell the computer to locate information in an already existing data file. After each response to an INPUT statement, we will use one or more statements to check the response for possible errors. These error-checking statements comprise the largest part of a data entry program module.

DATA FIELD LENGTH

In the last chapter, we discussed using MID$ to replace parts of one string with another. If the new name was longer or shorter than the old one, the length of the entire string changed, and data no longer appeared at the correct positions. We ended up with garbled information.

This problem, and many like it, is avoided by establishing a certain amount of space, a certain number of character positions into which a given element of data or data item is placed.

For the discussion here, we define a data field as a specified section of a string of character positions. This string of character positions could be divided into

several data fields. Data items (pieces of data) composed of real characters such as numbers or words are placed into the data fields.

This makes data fields something like substrings within a string. The fields define the substring positions—the beginning and end and number of characters—which you the programmer tailor to fit the length of the data items which are going into the fields. The examples that follow will help clarify this.

A number of related data items is called a data set, and the various items in a data set could all be stored in one fielded string. Since this is analogous to the way random access data files can store information, and since the programming techniques are similar for data entry error-checking, let's look at this concept more closely.

A data field can be thought of as one of a number of substrings which allows one long string to contain a number of separate data items. The concept of a record in a random access data file is similar, and is discussed later.

Data items always fit between two defined character positions within a fielded string. A simple example would be having one string variable to which both a customer's first and last names are assigned as one string, like this:

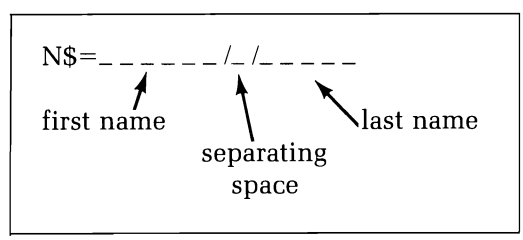
```
N$ = "VIVIAN VANCE"
```

The first name field is a six-character field in N$, occupying the first six character positions of that string (1 through 6). The separator field is a one-character field, located at character position 7. To be sure you understand, fill in the blanks in this sentence.

The last name field has (a) _____ characters and occupies character positions (b) _____ in the string assigned to (c) _____.

(a) Five.
(b) 8 to 12.
(c) N$

Below is a graphic look at the field in N$ with a slash (/) marking the field designations:



This particular data field works for the name in the example. However, the goal is to establish *reasonable* data fields. In this case, a reasonable data field

should hold ANY first or last name that might be assigned to N$. Certainly, many names contain more than six letters for the first name and five letters for the last. On one hand, you want to provide reasonably sized fields for the data. On the other hand, much storage space will be wasted if you try to cover all possibilities. There really may be someone named *John Jacobjingleheimerschmidt*, but reserving twenty-four character positions for a last name data field could be wasteful of storage space; if 95 percent of the last names in a data file have twelve letters or less, then half or more of the last name data field goes unused 95 percent of the time. In a file of 1000, 10,000, or 100,000 names, such as a mailing list, this can amount to a vast amount of unused disk storage space.

Data field lengths must be adequate and reasonable. If all the catalog numbers in an inventory data file are five characters, then obviously a five-character data field is sufficient.

To review, use a slash (/) to mark off the fields in a twenty-six character string assigned to A$, where the data fields hold the city, state, and zip code (the last line in a mailing address). Place a number in each field indicating which of the following data items are to occupy that field.

1. City name (fifteen characters maximum).
2. Two separator spaces.
3. State code (two-letter standard postal abbreviation).
4. Two separator spaces.
5. Zip code (five characters).

(a) A$=_ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

(a) A$=_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ ① /_ _ ② /_ _ ③ /_ _ ④ /_ _ _ _ _ ⑤

Next, consider the following data entry module to enter the city, state, and zip code. These data items are to be placed into the data fields you just defined.

```
100 INPUT "NAME OF CITY:"; C$
110 INPUT "STATE CODE:"; S$
120 INPUT "ZIP CODE:"; Z$
130 LET A$ = C$ + "  " + S$ + "  " + Z$
140 PRINT A$
```

Notice the concatenating statement in line 130—an attempt to get the data items into data fields. But these two RUNs demonstrate a serious problem that relates to the length of the city name.

(a)
```
RUN
CITY NAME?IOWA CITY
STATE CODE?IA
ZIP CODE?52240
IOWA CITY  IA  52240
```

(b)
```
RUN
CITY NAME?SOUTH SAN FRANCISCO
STATE CODE?CA
ZIP CODE?94080
SOUTH SAN FRANCISCO  CA  94080
```

Fill in the spaces to show the results of line 130 in the program for each of the sample RUNs:

(a) A$=_ _ _ _ _ _ _ _ _ _ _ _ _ _ _/_ _/_ _/_ _/_ _ _ _ _

(b) A$=_ _ _ _ _ _ _ _ _ _ _ _ _ _ _/_ _/_ _/_ _/_ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

(a) A$=IOWA_CITY__IA__/52/24/0_/_ _ _ _ _

(b) A$=SOUTH_SAN_FRANC/IS/CO/__/CA__94080

The fact that all cities don't have fifteen letters means that simple concatenation of this data does not place it into the defined character positions for the data fields. One approach is to assign a string of twenty-six spaces to A$ and then use the MID$ function to place each data item into its data field in the string. But SOUTH SAN FRANCISCO would end up being truncated to SOUTH SAN FRANC, and even with the fifteen-character limit, a more intelligent and preferable city name entry would be S. SAN FRANCISCO, SO. SAN FRAN., or even SOUTH S.F.

CHECKING DATA ENTRIES FOR ACCEPTABLE LENGTH

One programming technique to check data entries for acceptable length uses the LEN function in an IF . . . THEN comparison. If the data requested always have a defined number of characters, then an important check for mistakes in data entry would be to see whether the entry has the exact length it should. A U.S. zip code always has five characters, so a check for that data item would look like line 170:

```
160 INPUT "ZIP CODE "; Z$
170 IF LEN(Z$) <> 5 THEN PRINT "ENTER A 5 DIGIT
      CODE" : GOTO 160

RUN
ZIP CODE? 7542
ENTER A 5 DIGIT CODE.

ZIP CODE? 100010
ENTER A 5 DIGIT CODE.

ENTER ZIP CODE:
```

If the entry for the zip code does not have exactly five characters, then a mistake has been made, the user is so advised, and the computer repeats the prompting message and waits for another entry. With the new nine-digit zip code formats, a bit of reprogramming will be necessary.

Now you write a statement to check for proper length of the entry for the INPUT statement below:

```
(a)  140 INPUT "STATE CODE: "; S$
     150 _____________________________________________________
     _____________________________________________________
```

```
(a)  150 IF LEN (S$) <> 2 THEN PRINT "REENTER AS STANDARD
        2 LETTER CODE." : PRINT : GOTO 140
```

How can you check something like a city name, which is allowed fifteen characters or less? The city name could have less than fifteen characters, exactly fifteen, or more than fifteen. If it has more, you could settle for having the data entry truncated (cut off) after fifteen characters. Remember SOUTH SAN FRANC? A better option is to advise the user that a shorter entry is needed and allow the user to reenter the data item with an intelligent abbreviation.

```
120 INPUT "CITY NAME "; C$
130 IF LEN(C$) > 15 THEN PRINT "ENTER 15 CHARAC
     TERS OR LESS " : GOTO 120

RUN
CITY NAME? SOUTH SAN FRANCISCO
ENTER 15 CHARACTERS OR LESS.

CITY NAME?
```

Write a statement (similar to line 130) to check the entry for the INPUT statement below, where the data field for the entry is twenty characters maximum.

```
(a)  310 INPUT "STREET ADDRESS:"; S$
     320 _____________________________________________________
     _____________________________________________________
```

```
(a)  320 IF LEN(S$) > 20 THEN PRINT "ENTER AS 20 CHARACTERS
        OR LESS." : PRINT: GOTO 310
```

_____

"PADDING" ENTRIES WITH SPACES TO CORRECT FIELD LENGTHS

You are probably wondering how to *increase* the length of a data entry that has fewer characters than its data field. The solution involves automating the addition of spaces to "pad" the short entry (say, a short city name) with trailing spaces, so that the resulting city name *string*, which includes the padding spaces, exactly fits the data field. Remember, spaces occupy character positions and count as characters in the length of the string. Line 140 shows how to pad with spaces:

```
120 INPUT "CITY NAME "; C$
130 IF LEN(C$) > 15 THEN PRINT "ENTER 15 CHARAC
    TERS OR LESS " : GOTO 120
140 IF LEN(C$) < 15 THEN LET C$ = C$ + " " : GO
    TO 140
150 PRINT C$, LEN(C$)

RUN
CITY NAME? SEBASTOPOL
SEBASTOPOL
 15
READY.
```

In line 140, if the city name entered and assigned to C$ has less than fifteen characters, then a space is concatenated onto the end of the string. The new string assigned to C$ is the old string plus a space. The statement "goes back to itself" (GOTO 140) and keeps adding another space to the end of the C$ string until the string contains exactly fifteen characters including the spaces.

An alternative and better technique for padding string data to proper field length is possible. This method results in (probably unnoticeably) faster program execution because no iteration (loops) are necessary.

```
120 INPUT "ENTER CITY NAME "; C$
130 IF LEN(C$) > 15 THEN PRINT "ENTER 15 CHARAC
    TERS OR LESS " : GOTO 120
140 IF LEN(C$) < 15 THEN LET C$ = LEFT$(C$ + "
                    ",15 )
150 PRINT C$, LEN(C$)
```

In line 140, if the city name entered and assigned to C$ has less than fifteen characters, fifteen spaces are added. (This way, even a single character will be at least fifteen characters long.) Then the LEFT$ chops off the first fifteen characters. This results in our entry followed by enough spaces to give a total of fifteen. While there are other ways to accomplish this task, this is the simplest.

Let's look at this in more detail. Suppose the city name is DENVER, six spaces. C$+" " results in a string of the letters DENVER followed by fifteen spaces. Now, LEFT$(C$+" ",15) takes the first fifteen

characters of this new string—which will be the letters DENVER followed by nine spaces. We now have a string fifteen characters long consisting of our word followed by spaces.

Here's a short program you can type and run to try it out:

```
10 INPUT "NAME"; N$
20 N$ = LEFT$( N$ + "                ",15)
30 PRINT N$"<"
40 GOTO 10
```

The "<" in line 30 lets you see the end of the spaces. Notice you always get exactly fifteen characters back, whether you enter one or twenty.

This program can be shortened considerably by setting a string equal to fifteen (or whatever number you need) spaces at the beginning of your program:

```
5 B$ = "               " : REM 15 SPACES
```

Now line 20 can be replaced by

```
20 N$ = LEFT$( N$ + B$, 15)
```

For this simple example, that's not much better, but if you use this function twice in your program, you will save fifteen characters of space plus a chance of making typing mistakes.

(a) Describe the output produced by a RUN of this program:

```
10 B$ = "               " : REM 15 SPACES
20 N$ = "GLENN"
30 N$ = LEFT$( N$ + B$, 15)
40 PRINT N$
```

(a) "GLENN   "—a string, fifteen characters long, of the letters GLENN followed by ten spaces.

Now apply the techniques you have been using in a data entry module.

(a) Write a program routine to request that a user enter an alphanumeric product identification code with three characters, plus a product description with up to twenty characters maximum, followed by a two-character code identifying the person making the entries, using their first and last name initials. Once these three data items have been entered and tested,

combine the data into one string of twenty-five characters assigned to a single string variable. The sample RUN showing some error messages also shows the concatenated data for this user's entries.

```
RUN
THREE CHARACTER CODE? 12
ENTRY MUST BE 3 CHARACTERS.

THREE CHARACTER CODE? 123
DESCRIPTION? AUTOMOBILE
YOUR TWO INITIALS? HJR
JUST FIRST AND LAST INITIALS, PLEASE

YOUR TWO INITIALS? JB
123AUTOMOBILE    JB

READY.
```

(a)
```
120 INPUT "A THREE CHARACTER CODE "; C$
130 IF LEN(C$) <> 3 THEN PRINT "ENTRY MUST BE 3
    CHARACTERS" : PRINT : GOTO 120
140 INPUT "DESCRIPTION "; D$
150 IF LEN(D$) > 20 THEN PRINT "TOO LONG: 20 OR
    LESS CHARACTERS" :GOTO 140
160 IF LEN(D$) < 20 THEN LET D$ = LEFT$(D$ + "
                          ",20 )
170 INPUT "YOUR TWO INITIALS "; I$
180 IF LEN(I$) <> 2 THEN PRINT "JUST FIRST AND
    LAST INITIALS, PLEASE" : GOTO170
190 LET R$ = C$ + D$ + I$
200 REM  FOR DEMO ONLY, PRINT R$
210 PRINT R$
```

What's the advantage in setting up data fields in a single string and putting more than one data item into it? The reasons will become clear in later chapters. For now, the answer has to do with how data files can store information using some automated data entry procedures and equipment and with the ease with which BASIC allows the manipulation of substrings using MID$ for particular applications.

Examine the program below and answer the questions that follow it.

```
120 INPUT "THE CITY NAME "; C$
130 IF LEN(C$) > 15 THEN PRINT "ENTER 15 CHARAC
    TERS OR LESS" : PRINT : GOTO 120
140 IF LEN(C$) < 15 THEN LET C$ = LEFT$(C$ + "
                      ",15 )
150 INPUT "STATE CODE "; S$
160 IF LEN(S$) <> 2 THEN PRINT "JUST 2 LETTER A
    BBREVIATION, PLEASE" : GOTO150
170 INPUT "ZIP CODE "; Z$
180 IF LEN(Z$) <> 5 THEN PRINT "ENTER EXACTLY 5
    DIGITS FOR ZIP CODE" : GOTO170
190 LET T$ = C$ + S$ + Z$
200 REM  FOR DEMO ONLY, PRINT T$
210 PRINT T$
```

(a) What is the purpose of line 130?

(b) What does LEFT$(C$+“ ”,15) in line 140 do?

- - - - - - - - - - - - - - - - - - - -

(a) Tests to be sure user has not entered more than the acceptable number of characters (fifteen) for the city name field.
(b) Fills in, adds on, or concatenates spaces from the last character of the C$ string up to and including character field position 15. Changes C$ to a fifteen-character string if there were fewer than fifteen characters in the string entered for C$.

STRIPPING THE PADDING SPACES FROM SUBSTRINGS IN FIELDS

You know how to pad a string with extra spaces to arrive at the proper field length for that data item. Now let's explore a way to eliminate the extra blank spaces when you extract data packed into a string. In the example where we wanted to change a person's last name, it was necessary to pad names with spaces to the proper field length so that corrections could be made, if necessary, and so the first and last names could be found separately. But for name printing purposes, you want to eliminate all the extra blank spaces. The method shown below uses the MID$ function. In our example, N$ really consists of the eight characters for F$, one space separating the two fields, and twelve characters for L$. If the name concatenated in N$ is Jenny Smiles, then

```
N$ = "JENNY     SMILES       "
```

including the field-separating space at character position 9, would result.

The example program below shows how to use first and last names separately, without extra spaces, in a computer-printed "thank you" letter.

```
20 LET F$ = "ROSEMARY"
40 LET L$ = "ROBERTS       "
200 LET N$ = F$ + " " + L$
730 LET F$ = MID$( N$, 1, 9)
740 LET L$ = MID$( N$, 10, 12)
750 Z = LEN(F$) : FOR S = 1 TO Z : IF MID$(F$,S
    ,1) <> " " THEN NEXT S : S = Z
760 PRINT "DEAR " MID$(F$,1,S-1) ","
770 Z=LEN(L$)
780 FOR S1 = 1 TO Z :IF MID$(L$,S1,1) <> " " TH
    EN NEXT S1 :S1 = 13
790 PRINT "IT SURE WAS NICE TO SEE YOU AND"
800 PRINT "MR. " MID$( L$, 1, S1 - 1 );
810 PRINT " AT THE GET-TOGETHER"
820 PRINT "THE OTHER EVENING."
```

```
DEAR ROSEMARY,
IT SURE WAS NICE TO SEE YOU AND
MR. ROBERTS AT THE GET-TOGETHER
THE OTHER EVENING.

READY.
```

The procedure used in this example is called "parsing." It means searching through something one step at a time until you find what you are seeking. Here the FOR NEXT loops in lines 750 and 780 step the MID$ function through each character of the string checking for spaces. If no space is found, then the NEXT statement increments the counter, and the next character is checked. If the character is a space, then we leave the FOR NEXT loop with the value of S (or S1) equal to the position of the first space in the string. When we print, we need to use S – 1, because the position found has a space, and we are looking for the last letter of the word, which will be the position before the space (lines 760 and 800).

(a) In lines 750 and 780, what does the MID$ function search for?

_____

(b) What value is assigned to S and S1 in the same lines? _____

_____

(a) Looks for first space in string.
(b) Assigns character position number of first space to S and S1.

CHECKING ENTRIES FOR NO RESPONSE

One idiosyncracy of the INPUT statement already pointed out is that if the user merely presses the RETURN key when the computer is waiting for a response to an INPUT statement, no string is assigned to the string variable (it consists of "", the empty string). (On the PET and CBM, this can't happen, since the program is terminated by RETURN without input, and you'll get the READY. message.) If the computer then encounters a checking statement that pads the entry with spaces to the proper field length, the entire entry would end up as a string of spaces and be duly included in the data field for that entry. So checking the data entries for no string assignments (empty strings) is a must, and should be part of your data entry program modules. See Appendix 4; also, refer to your computer's reference manuals.

For the Commodore 64, you can use two different techniques to test whether a string variable has been assigned a string. They work equally well.

```
IF A$ = "" THEN ...
```

or

```
IF LEN(A$) = 0 THEN ...
```

On the PET and CBM, the program is terminated if no entry is made before RETURN is pressed for an INPUT statement. See Appendix 4 for some solutions.

The decision the programmer must make (and it will vary with each situation) is what to do after the THEN when the IF . . . THEN condition is true and a null assignment has been mistakenly made. Whatever you do, do *not* have the computer merely repeat the INPUT prompt, as in the "what-not-to-do" example below.

```
170 INPUT "CUSTOMER NUMBER"; C$
180 IF LEN(C$) = 0 THEN 170

RUN
CUSTOMER NUMBER?
CUSTOMER NUMBER?
CUSTOMER NUMBER?
```

A user who persists in not entering the customer number gets no information as to what is wrong. Always provide a helpful error message, perhaps even a beep, so the user knows something is amiss with the present response or entry.

```
170 INPUT "CUSTOMER NUMBER"; C$
180 IF LEN(C$) = 0 THEN GOSUB 1010: GOTO 170
.
.
.
1010 PRINT "PLEASE, WE MUST HAVE THE CUSTOMER
    NUMBER TO CONTINUE." : PRINT
1020 RETURN
.
.
.
```

(a) With this information in mind, write the data entry routine that will produce the prompts shown below. Test each data item for no response, immediately after the item is entered.

```
RUN
CUSTOMER NUMBER?
ENTRY ERROR. PLEASE REENTER.

CUSTOMER NUMBER? 12345
CUSTOMER NAME?
NO ENTRY WAS MADE. PLEASE RESPOND AS REQUESTED.

CUSTOMER NAME? WINDOW SUPPLY CO.
PRODUCT NUMBER?
WE CANNOT CONTINUE WITHOUT THIS DATA.

PRODUCT NUMBER? 0008
QUANTITY ORDERED?
PLEASE ENTER THE CORRECT VALUE.

QUANTITY ORDERED? 12
READY.
```

(a) *

```
200 INPUT "CUSTOMER NUMBER "; C$
210 IF LEN(C$) = 0 THEN PRINT "ENTRY ERROR. PLE
     ASE REENTER." : GOTO 200
220 INPUT "CUSTOMER NAME "; N$
230 IF LEN(N$) = 0 THEN PRINT "NO ENTRY MADE. P
     LEASE TYPE RESPONSE" : GOTO 220
240 INPUT "PRODUCT NUMBER "; P$
250 IF LEN(P$) = 0 THEN PRINT "CAN'T CONTINUE W
     ITHOUT THIS DATA." : GOTO 240
260 INPUT "QUANTITY ORDERED "; Q$
270 IF LEN(Q$) = 0 THEN PRINT" PLEASE ENTER COR
     RECT VALUE." : GOTO 260
```

(or some similar messages).

*You have undoubtedly noticed that the page width of this book does not allow us to use the same wraparound for program lines as is common on the forty-column displays used by Commodore computers. When program lines are broken in two, notice that the breaks occur at different places in the statement line in this book than where the wraparound break appears on your display when you type the same program line. Do not press RETURN at the end of the first line, but simply continue typing until the end of the program line, then press RETURN.

We have chosen the break points to make it easy to read the lines, and to allow us to use a readable typeface. You should have no difficulty in typing in the programs from the listings in this text. About fifty of the longer programs in this book are available on a floppy disk from the publisher, ready to load. We encourage you to type the programs yourself, so you have to read through them and to work the problems yourself.

Depending on the program user's sophistication, even more detailed error messages for problems like the no string entry and others may be necessary. Our examples have given minimum messages to keep the examples short, uncluttered, and easy to understand, but they may not be adequate to ensure a proper response. Return to this example:

```
170 INPUT "CUSTOMER NUMBER:"; C$
180 IF LEN(C$) = 0 THEN GOSUB 1010 : GOTO 170
 .
 .
 .
1010 PRINT "YOU APPARENTLY PRESSED THE 'RETURN' KEY WITHOUT
    MAKING AN ENTRY."
1020 PRINT "WE NEED A CUSTOMER NUMBER THAT LOOKS LIKE
    THIS: A-121." : PRINT
1030 RETURN
```

Another example:

```
230 INPUT "COMPANY NAME: "; C$
240 IF LEN(C$) > 12 THEN GOSUB 1010 : GOTO 230
 .
 .
 .
1010 PRINT "YOU ENTERED: "; C$
1020 PRINT "PLEASE ABBREVIATE THE COMPANY NAME TO 12 CHARACTERS
     OR LESS."
1030 PRINT "EXAMPLE: ALPHA PRODUCTS COMPANY COULD BE SHORTENED
     TO 'ALPHA PRO CO'" : PRINT
1040 RETURN
```

Subroutines need to be protected from the main program that calls or branches to them. Depending on how a program is constructed, a subroutine could be encountered and executed as if it were part of the main program, especially if the subroutine section is one of the program's last modules. *Use a STOP or END statement between the main program and the module(s) containing the subroutines.* This protects the first subroutine after the main program from being executed in normal line order. If you don't do this, the first subroutine will be executed, and when the RETURN statement is reached, the program will stop with a "?RETURN WITHOUT GOSUB" error.

(a) Write an error message subroutine, accessed by a GOSUB statement executed after a true IF . . . THEN comparison, that displays an INPUT entry and describes how to comply with the limit of twenty characters (because of data field length) for entries to the following statement:

```
320 INPUT "PRODUCT DESCRIPTION: "; P$
```

Sample entry to above statement:

```
RUN
PRODUCT DESCRIPTION ? LEFT-HANDED MONKEY WRENCH

YOU ENTERED LEFT-HANDED MONKEY WRENCH
FOR A PRODUCT DESCRIPTION. PLEASE
REENTER, BUT SHORTEN YOUR ENTRY BY
USING ABBREVIATIONS SO THAT THE PRODUCT
DESCRIPTION IS LESS THAN 20 CHARACTERS
LONG, INCLUDING SPACES AND PUNCTUATION.

PRODUCT DESCRIPTION ?
```

(a) Your solution should be similar to this:

```
320 INPUT "PRODUCT DESCRIPTION "; P$
330 IF LEN(P$) > 20 THEN GOSUB 1120 : GOTO 320
1110 STOP : REM PROTECT SUBROUTINE
1120 PRINT : PRINT "YOU ENTERED " P$
1130 PRINT "FOR A PRODUCT DESCRIPTION.  PLEASE"
1140 PRINT "REENTER, BUT SHORTEN YOUR ENTRY BY
1150 PRINT "USING ABBREVIATIONS SO THAT THE PROD
     UCT
1160 PRINT "DESCRIPTION IS LESS THAN 20 CHARACTE
     RS
1170 PRINT "LONG, INCLUDING SPACES AND PUNCTUATI
     ON." : PRINT
1180 RETURN
```

REPLACEMENT OF DATA ITEMS IN A STRING WITH DEFINED DATA FIELDS

The first example program at the beginning of this chapter illustrated the changing of part of an existing string. In the example, the five characters denoting price were replaced by the new price, represented as a string of five characters. Replacing a substring of one length by a substring of another length makes finding information in substrings a nightmare. The example program demonstrated the most practical solution: always use data fields of predefined length for each data item in a string. That way, changes or replacements will be complete, and data will always be in the same character positions.

Now you will practice designing program modules to accomplish the assignment and the replacement of fields within strings, using first and last names as examples.

Step 1. Define the field for the first name to have eight characters and that for the last name, twelve characters, with a space separating the name fields.

Step 2. Create the data entry routine.

```
100 PRINT : INPUT "YOUR FIRST NAME "; F$
110 IF LEN(F$) = 0 THEN PRINT "PLEASE, YOU MUST
      HAVE A NAME." : GOTO 100
120 IF LEN(F$) > 8 THEN PRINT "ABBREVIATE TO LE
     SS THAN 8 LETTERS" : GOTO 100
130 IF LEN(F$) < 8 THEN LET F$ = LEFT$(F$ + "
           ",8)
140 PRINT : INPUT "YOUR LAST NAME ";L$
150 IF LEN(L$)  = 0 THEN PRINT "PLEASE COOPERAT
     E." : GOTO 140
160 IF LEN(L$) > 12 THEN PRINT "ABBREVIATE TO 1
     2 OR LESS LETTERS." : GOTO 140
170 IF  LEN(L$) < 12 THEN LET L$  = LEFT$(L$ +
     "              ",12)
180 LET N$ = F$ + " " + L$
190 PRINT N$ : REM  OUTPUT FOR THIS MODULE
```

Step 3. Replacement routine for last name field.

```
400 INPUT "NEW LAST NAME "; L1$
410 IF LEN(L1$) > 12 THEN PRINT"ABBREVIATE TO 1
     2 OR LESS LETTERS" : GOTO 400
420 IF LEN(L1$) < 12 THEN LET L1$ = LEFT$(L1$ +
      "             ", 12 )
430 :
440 N$ = MID$(N$,1,9) + L1$
450 PRINT N$ : REM THIS MODULE'S OUTPUT
```

Step 4. Name printing routines.

```
600 REM*** TO PRINT FIRST NAME ONLY
610 PRINT MID$(N$,1,8)
620 :
630 REM*** TO PRINT LAST NAME ONLY
640 PRINT MID$(N$,10,12)
650 :
660 REM*** TO PRINT COMPLETE NAME
670 PRINT N$
```

Check your understanding of the routines above by answering the following questions.

(a) In line 170, what is the purpose of this section of the statement?

```
LET L$ = LEFT$(L$ = "            ",12)
```

(b) What does line 180 do?

(c) In line 440, what does the MID$ function do?

(d) Assume all three program modules were typed into the computer. In the data entry module, the first name entered was VAL. The last name entered was JEANS. No changes were made in the name. Show how the name would be displayed by line 670.

(a) Fills in unused character positions with blanks to the correct field length (same technique used in lines 130 and 420).
(b) Packs first and last names into N$, separated by a space.
(c) Concatenates the first nine characters of the original N$ with the new last name (L1$), creating a new N$.
(d) `VAL JEANS`
(All "padding" spaces are included when N$ is printed.)

THE VAL FUNCTION IN DATA ENTRY CHECKS

If the product number and quantity ordered in a program must be numeric quantities, VAL can easily convert these numbers stored as strings to numeric values.

```
330 A$ = "128.95"
340 PRINT VAL(A$)
350 A = VAL(A$)
360 PRINT A

RUN
 128.95
 128.95
```

In the conversion, either a leading space is added for the implied plus sign, or a minus sign is provided if the quantities are negative.

But the VAL function does not completely solve the problem of converting string numbers to numeric values. For example, alphabetic information included in a string you wish to convert to a numeric value presents a very real problem that can range from accidentally using the letter O (oh) for a zero, to a quantity that includes the units that measure that quantity (12 quarts). Therefore, always test to be sure that if numeric values are needed, that is what was entered.

Following are some sample values:

```
5 REM*** VAL TEST #1
10 LET A$ = "ABC"
20 PRINT A$, VAL(A$)
25 REM*** VAL TEST #2 - NULL STRING
30 LET A$ = ""
40 PRINT A$, VAL(A$)
45 REM*** VAL TEST #3
50 LET A$ = "123ABC"
60 PRINT A$, VAL(A$)
65 REM*** VAL TEST #4
70 LET A$ = "ABC123"
80 PRINT A$, VAL(A$)

RUN
ABC         0
            0
123ABC      123
ABC123      0
```

Notice in the RUN above that alphabetic characters result in a value of 0, as do a null string and the mixed alphanumeric data where the alphabetic information *precedes* the numeric (ABC123). Notice also that the mixed data 123ABC results in a value of 123. The VAL function disregards the alphabetic information that *follows* numeric information in the same string. This is convenient if you wish to enter the quantity and the units, such as 14 gallons, but inconvenient if you wish to check for the validity of the data entered. Here, you want to ascertain that the data entered are numeric, so when the VAL function is used you get valid numeric values. At this point, for mixed numbers and letters, assume that the user did enter the correct value.

A simple test to validate numeric information would be

```
IF VAL(A$) = 0 THEN PRINT "ENTER NUMERIC VALUES ONLY."
```

However, if the value zero is an acceptable entry, it will be rejected by this test. A better test includes a check to see if the string entered was zero:

```
IF VAL(A$) = 0 AND LEFT$(A$,1) <> "0" THEN PRINT "NUMBERS ONLY."
```

Which test you use depends upon the possibility of zero being an input.

For program portability, we suggest you place this data test AFTER the statement that tests for an empty string. On some computers, the VAL function of an empty string stops the program with an error condition.

(a) Now do some programming. For the data entry problem on page 80, you wrote a program to produce a data entry sequence with "no entry" checks added. Now add data checks that ensure that the product number and the quantity ordered are numeric values. Also include a data check to be certain that the product number is a four-digit number.

```
200 PRINT : INPUT "CUSTOMER NUMBER "; C$
210 IF LEN(C$) = 0 THEN PRINT "ENTRY ERROR, PLE
    ASE REENTER." : GOTO 200
214 IF VAL(C$) = 0 THEN PRINT "PLEASE ENTER NUM
    BERS ONLY." : GOTO 200
220 PRINT : INPUT "CUSTOMER'S NAME "; N$
230 IF LEN(N$) = 0 THEN PRINT "NO ENTRY MADE--P
    LEASE TYPE A NAME." :GOTO 220
240 PRINT : INPUT "PRODUCT NUMBER "; P$
250 IF LEN(P$) = 0 THEN PRINT "CAN'T CONTINUE W
    ITHOUT PROD. NUMBER." :GOTO 240
254 ----------------------------------------
256 ----------------------------------------
260 PRINT : INPUT "QUANTITY ORDERED "; Q$
270 IF LEN(Q$) = 0 THEN PRINT "PLEASE ENTER COR
    RECT VALUE." : GOTO 290
280 ----------------------------------------
```

```
254 IF VAL(P$) = 0 THEN PRINT "PLEASE ENTER NUM
    BERS ONLY." : GOTO 240
256 IF LEN(P$) <> 4 THEN PRINT "PLEASE ENTER ON
    LY A 4 DIGIT NUMBER." : GOTO 240
280 IF VAL(Q$) = 0 AND Q$ <> "0" THEN PRINT "NU
    MBERS ONLY" : GOTO 290
```

USING STR$ TO CONVERT VALUES TO STRINGS

The STR$ function serves the opposite purpose of the VAL function. It converts numeric values into strings. This allows you to manipulate numbers with string functions. You can use it to convert numeric values to strings assigned to variables in concatenating several small strings into a string variable, as done earlier in this chapter. For example, you may have combined product number, product description, and quantity in inventory into one long string. You may then need the quantity in inventory for an accounting procedure or another calculation. Such operations require a *numeric* value. You would convert the string to a numeric value by using the VAL( ) of the entry string. When the quantity is stored, you can convert back to a string by taking the STR$( ) of the numeric value to place it into the P$ string.

P$ 17633 BOOK TITLE 144

```
P$ = P$ + STR$(Q)
```

or

```
Q$ = STR$(Q)
P$ = P$ + Q$
```

When the computer converts a numeric value to a string with STR$, a leading space is included if the numeric value is positive. A minus sign is included in the string if the value is negative.

Try this demonstration program:

```
140 LET X = 847.25
150 LET X$ = STR$(X)
160 PRINT "X = " X
170 PRINT "X$ = " X$
180 PRINT "LEN(X$) = " LEN(X$)

RUN
X = 847.25
X$ = 847.25
LEN(X$) = 7
```

Notice that the leading space is still there after the value has been converted to a string.

In this example, the LEN(X$) is seven—five numeric characters, the decimal point, and the leading space in a positive value converted to a string. (Remember, blank spaces, decimal points, and other punctuation marks are characters.) If you fail to provide enough string length or field space, you will inadvertently lose significant digits or characters due to computer truncation. A six-digit number with a leading space does not fit in a six-character field.

How many characters will the following data items have if they are converted from values to strings with the STR$ function?

(a) 171.83 _____

(b) 2001 _____

(c) –999 _____

(a) 7
(b) 5
(c) 4

CHECKING FOR ILLEGAL CHARACTERS

Using the ASC function in a data entry checking statement is a powerful tool to determine whether illegal or unlikely characters have been included in an INPUT string. Checking is done by a combination of the the ASC function, the MID$ function, an IF . . . THEN statement, and a FOR NEXT loop. First the length of the entry is determined by the LEN function, which is used as the upper limit of the FOR control variable, like this:

```
350 INPUT "SIX-CHARACTER CATALOG CODE:"; C$
360 FOR X = 1 TO LEN(C$)
```

Then the MID$ function, using the FOR control variable (value of X for any iteration) to determine which character to examine, selects each character in the string for comparison to an ASCII number, like this:

```
370 IF ASC(MID$(C$,X,1)) = 32 THEN PRINT "REENTER BUT DO NOT USE
   SPACES.": GOTO 350
380 NEXT X
```

(*Note:* Here is one of those exceptions when you "leave" a FOR NEXT loop.) Notice that any character that can be entered as part of a string can be checked to see that legal characters that should be there are there, or that illegal characters are not included. Notice, too, that the error message could be located in a subroutine outside of the FOR NEXT loop. In addition, you can use the logical AND and OR, to check for more than one character or group of characters in the same IF . . . THEN statement.

What if a user made the following response to line 350 in the above example? Answer the questions based on this response and this program segment:

```
RUN
SIX-CHARACTER CATALOG CODE?A - 1314
```

(a) What is the length of the substring selected by the MID$ function in line 370?

_____

(b) What ASCII value is compared to 32 the first time through the FOR NEXT loop? _____

(c) The second time through? _____

_____

(d) On which iteration (time through) of the FOR NEXT loop is the comparison in line 370 true? _____

(e) What value does the FOR statement control variable have as an upper limit for this user's response? _____

(a) 1.
(b) 65 (for A).
(c) 32 (for a space).
(d) Second iteration.
(e) LEN(C$)= 8.

(a) Write a data entry checking routine similar to the one above that prints an error message if an illegal character is encountered. Use more than one IF . . . THEN statement with the ASC function in the comparison, or a single IF . . . THEN statement that uses the logical AND and OR. The only *legal* characters for the entry are the digits 0 (zero) through 9 inclusive and the decimal point, such as would be entered for a dollar and cents entry without a dollar sign.

(a)
```
100 INPUT "A VALUE "; V$
110 FOR X = 1 TO LEN(V$)
120 IFASC(MID$(V$,X,1))>=48ANDASC(MID$(V$,X,1))
     <=57ORASC(MID$(V$,X,1))=46THEN140
130 PRINT "INVALID ENTRY. ENTER NUMBERS AND DEC
     IMAL POINT ONLY." : GOTO 100
140 NEXT X
```

An alternative solution for less trauma and typing time:

```
100 INPUT "A VALUE ";V$
110 FOR X = 1 TO LEN(V$)
120 LET V = ASC( MID$( V$, X, 1) )
130 IF V >= 48 AND V <= 57 OR V = 46 THEN 150
140 PRINT "INVALID ENTRY. ENTER NUMBERS AND DEC
     IMAL POINT ONLY." : GOTO 100
150 NEXT X
```

SCREEN FORMATTING

Commodore has provided you with a unique capability to use the CURSOR and screen control keys to format the display of information on the screen. We have not made much use of these capabilities in this book, because they make program listings hard to read and type in. However, they allow you to save program space, and to place information where you want it on the screen.

For example, when displaying address data for a user to change, you may wish to display the data centered on the screen. At the bottom, you can place a "prompt" or "status" line giving the choices or commands. You can also clear the screen, printing information from the top down, rather than starting at the bottom and "scrolling" the screen, which looks cluttered and is hard to read. Here is a short program to illustrate. It reads address data, displays it (always in the same place on the screen), then asks for changes near the bottom. Notice the use of cursor-down instead of multiple print statements (print : print).

```
10 rem format
100 let d$ = "[home][down][down][down][down][do
    wn][down][down][down][down][down][down][down
    ][down][down][down][down][down][down][down][
    down][down][down][down]" : rem home & 23 crs
    r downs
110 for i  = 1 to 10
120 print "[clr][down][down][down][down][down]N
    umber " i
130 print left$(d$,24)"    Press RETURN to cont
    inue."
140 get z$ : if z$ <> chr$(13) then 140
150 next i
160 print "[clr][down][down][down]Done."
```

This sample used a string defined as home (to always start at the top of the screen) followed by twenty-three cursor downs. To place something on line 10, you would PRINT LEFT$(D$,11);. Note that we used the first eleven characters, since HOME is the first. If you leave off the semicolon, then the next PRINT statement will be on the next line and you could use 10 in the LEFT$ to print on the tenth line. If you use formatting statements frequently, it saves a lot of time, space, and hassle to define a string like 'd$' and use LEFT$ to place information on the screen, as the two format examples to follow will demonstrate.

FORMATTING DATA ENTRY

Commodore's ability to use cursor keys within print statements allows extremely powerful formatting of the screen. This is one simple example. In this case, we display old data on the screen and ask the user if they want to accept or change it. To accept it, they merely press RETURN; to change it, then type over the old data.

Here is how it looks:

```
PRESS 'RETURN' TO ACCEPT, OR RETYPE DATA
? THE TURNKEY COMPANY
```

The trick, as shown in the following program, is to print the old name on the screen starting two spaces over, then do a CRSR UP, followed by an INPUT, which then is back on the same line. Pressing RETURN will therefore enter the entire line, and you have your old data back. Since pressing RETURN captures whatever is on the line, any changes that were typed will be picked up.

```
10 rem display old data in input line
100 :
110 n$ = "Old Farming Company
200 print "Retype, or press [rvs]RETURN[off] to
      accept."
210 print "[down]  " n$
220 print "[up][up]" : input n$
230 :
240 print "Data entered was: [rvs]" n$
250 rem program continues
```

Pressing RETURN:

```
Retype or press RETURN to accept.
?Old Farming Company
Data entered was: Old Farming Company

READY.
```

Typing "NEW" over "OLD" and pressing RETURN:

```
Retype or press RETURN to accept.
?New Farming Company
Data entered was: New Farming Company

READY.
```

The only disadvantage to this approach is that you can't see the old name when typing over it, so the method used in the programs in this book may be preferred.

The other use of cursor keys for formatting is as a response to error messages. This allows you to control placement on the screen so repeated incorrect answers don't scroll information, or leave the screen cluttered with many incorrect responses and their error messages. For this, it is often helpful to define a variable as a string of thirty-eight spaces (remember that thirty-ninth character?):

```
B$ = "                                      " : REM 38 SPACES
```

This can be used to print over (and erase) information that is no longer needed. That technique will be used in these sample programs to erase error messages after the user types the next response.

The first example uses cursor keystrokes imbedded in PRINT statements to place information on the screen, and to place the blank lines to erase unneeded information. Notice how much neater the second program looks. What you can't see was how much faster the second program was debugged! As mentioned earlier, the use of a string defined as HOME (*not* CLEAR!) plus twenty-three cursor down characters is a neat way to place information on your screen.

```
10 rem error2
90 let b$ = "
          " : rem 38 sp
100 print "[clr][down][down][down][down][down][
    down]" : rem clear and 6 crsr downs
110 input "Product name[shsp][shsp][shsp][shsp]
    [left][left][left]"; n$
115 print b$ : print b$ : printb$ "[up][up]"
120 if n$ = "[shsp]" then print"[up][up][up]" :
     goto 110
125 let n = len(n$)
130 if n < 12 then n$ = right$("            " +
     n$,12) : goto 145
140 print "[down]Too long--12 characters max."
141 print "[home][down][down][down][down][down]
    [down][down]" b$ "[up]" : goto 110
145 print b$ : print b$ : print b$
150 print "[down]Result is: " n$
160 :
170 rem program continues
```

```
10 rem error1
20 rem places each statement
80 d$ = "[home][down][down][down][down][down][
     down][down][down][down][down][down][down][do
     wn][down][down][down][down][down][down][down
     ][down][down][down]" : rem home and 23 crsr
     downs
90 let b$ = "
          " : rem 38 sp
100 print "[clr]"
105 print left$(d$,6);
109 rem 4 shift-spaces, 3 crsr-left in prompt
110 print b$ "[up]" : input "Quantity ordered[s
     hsp][shsp][shsp][shsp][left][left][left]"; q
     $
120 n = 4 : gosub 500 : rem clear
130 if q$ = "[shsp]" then print left$(d$,8) "Pl
     ease enter a number." : goto 105
140 if val(q$) = 0 then print left$(d$,8) "Digi
     ts 0 - 9 only." : goto 105
150 if len(q$) > 3 then  print left$(d$,8) "Thr
     ee or less digits." : goto 105
160 if val(q$) > 299 then print left$(d$,8) "Ma
     x. 299  units." : goto 105
170 print left$(d$,10) "Quantity is: " q$
180 :
190 rem program continues
200 end
500 rem clear lines routine
510 for  i = 1 to n
520 print b$
530 next i
540 return
```

A DISCUSSION OF DATA ENTRY AND CHECKING PROCEDURES

This chapter has included recommendations, hints, and techniques for dealing with and checking data. This section describes and summarizes procedures used to check and validate all data entries.

There are two schools of thought regarding at what point incoming data should be checked for errors. One states that since the data entry operator's time is costly, the operator should merely enter data using the fastest possible procedures, with no checks for accuracy at the time data are entered. This position requires that more time be spent training the data entry operator in fast, accurate computer entry techniques. Then later, another program does the error checking on the data at fast computer speeds. Whenever a data error is encountered, the

computer "kicks out" or rejects the entire data entry transaction for that set of data and prints the rejected information in a special report. The rejected data set is then reprocessed or reentered by the data entry staff. This procedure works well if the number of rejects is low.

In contrast, we prefer the second approach—checking data on the way in. As each item is entered, it is error-checked immediately. If an error is detected, the computer operator is advised to reenter the data.

One advantage is that the person making the entry error is responsible for correcting it. This method also gives management a better measure of an operator's work flow since only accurate, accepted information is completed during a work day. In the alternate method, data entry rates may seem high, but so may be the reject rate, and special procedures are needed to verify who is making the entry errors. A less subtle technique is to signal an entry error with a terminal beeper or bell. Each time faulty data are detected, the sound signals the operator (and the manager, if present) that an error was made and draws attention to the "culprit." These are concerns in a business environment. Notwithstanding, the beep is helpful because you don't have to watch the screen when typing; your attention is called to problems.

The immediate error check is more in keeping with the small business or personal nature of most programming applications presented here. And since all the error-checking routines follow the data entry immediately, you can easily read the program to see what kinds of error checks are being made.

Two general data entry techniques are universally accepted. One uses a graphic reproduction on the video screen of the paper form from which data are entered. It makes sense to reproduce that form on the screen and have the computer prompt the operator to "fill in the blanks" just as they appear on the paper form or data source sheet.

In many computer applications in small businesses, science, education, administration, and even the home, many kinds of data that you would want stored in a computer would appear on some kind of standardized form. Of course this would not be true if you were entering a year's accumulation of cash register tapes and retail store receipts from a shoe box into a computer program that helps you figure your deductible expenses for tax purposes.

When data to be entered are always on standardized forms, the data entry person sees the form reproduced on the screen, and the cursor, blinking characters, or some other indication tells where on the form the computer expects an entry to be made.

With the capabilities of Commodore BASIC, each keystroke of an entry can be checked for validity of character type, or even for all the acceptable characters for that character position in the data field for that section of the displayed form. Such routines for display, entry, and error-checking from forms would considerably enlarge the fifty or so programs in the rest of this book, so we use simple data entry modules that do not distract from the instructional points being made in the example programs.

Another error-checking technique has the computer redisplay one or more sets of data entered. The operator is then given the chance to reenter any incorrect items, even after the entry checking has been performed by the compu-

ter. This is the "last chance" to pick up spelling errors, number transpositions, typographical errors, and anything else for which entry error checks cannot be designed into the program itself. An example of such a post-data entry display appears below:

```
THANK YOU. HERE IS THE DATA YOU ENTERED.

    CUST. #     PROD. #     QUANTITY

1 - 98213       17892       18
2 - 98213       24618       12
3 - 98213       81811       144

ARE THERE ANY CHANGES (YES OR NO)? YES
NUMBER OF THE LINE IN WHICH A CHANGE IS NECESSARY?
```

Before a summary report such as this one is displayed, clear the screen of previously displayed information. In fact, clearing the screen before each new entry or after the entry of a data set is important in the entire concept of avoiding errors. If the graphic display of a data source form is used, then the screen should be cleared and the form redisplayed with the dataset in which correction is needed. The operator can make any corrections directly on the new form.

Let's design a simple program module to provide entry and error-checking for a mailing list of names and addresses, using a teaching technique employed throughout this text. Part of the program will be provided, and you will be asked to write key statements and sections for the program.

First, consider the data. The data will have the format shown below, with each dataset containing five items within one string with five data fields:

```
                                                        55
/1 ____________20/21 ____________40/41 _____50/12/53 _____57/
     name              address          city  state   zipcode
```

The introductory module, explaining this program's dataset and variables, looks like this:

```
100 REM
110 :
120 REM     VARIABLES USED
130 REM     N$ = NAME (20 CHAR. MAX.)
140 REM     A$ = ADDRESS (20 CHAR. MAX.)
150 REM     C$ = CITY (20 CHAR. MAX.)
160 REM     S$ = STATE (2 CHAR.)
170 REM     Z$ = ZIP CODE (5 CHAR.)
180 REM     D$ = ENTIRE DATASET CONCATENATED (67 CHAR.)
190 :
```

The data entry section of the program should ask for the entry of the five items that make up the dataset, and one of the authors has responded to the RUN of the program you are going to help complete.

```
RUN
NAME: JERALD R. BROWN
STREET ADDRESS: 13140 BAMBI LANE
CITY: SEBASTOPOL
2-LETTER STATE ABBREVIATION CODE: CC
ZIP CODE (5 DIGITS): 95472
```

Next, the computer clears the screen and then displays the dataset just entered in the form of a menu of choices. Notice that the state code for California is CA, not CC as was mistakenly entered. (This is the same RUN continued.)

```
YOU HAVE ENTERED THE FOLLOWING DATA:

- 1 - JERALD R. BROWN
- 2 - 13140 BAMBI LANE
- 3 - SEBASTOPOL
- 4 - CC
- 5 - 95472
- 6 - NO CHANGES

CHECK FOR ERRORS. ENTER THE NUMBER OF THE ITEM THAT NEEDS CORRECTING.
IF NO CHANGES ARE NEEDED. ENTER THE NUMBER '6'.
YOUR SELECTION (1 TO 6)? 4
2-LETTER STATE ABBREVIATION CODE? CA
```

Once again the screen is cleared and the dataset redisplayed, in case more corrections are needed. Because the data in the example are now correct, the user entered a 6 from the menu choices. The program then proceeds to concatenate the data and display it, including the field padding spaces. (This is still the same RUN continued.)

```
YOU HAVE ENTERED THE FOLLOWING DATA:

- 1 - JERALD R. BROWN
- 2 - 13140 BAMBI LANE
- 3 - SEBASTOPOL
- 4 - CA
- 5 - 95472
- 6 - NO CHANGES
```

```
CHECK FOR ERRORS. ENTER THE NUMBER OF THE ITEM THAT NEEDS CORRECTING.
IF NO CHANGES ARE NEEDED, ENTER THE NUMBER '6'.
ENTER YOUR SELECTION (1 TO 6): 6

JERALD R. BROWN      13140 BAMBI LANE      SEBASTOPOLCA95472

DO YOU HAVE ANOTHER NAME AND ADDRESS TO ENTER? N
READY.
```

You will want to refer to these displays from the program RUN to help you see how to write various parts of the program.

This program will use the same statements both for entering and error checking the five data items in each dataset, and also for correcting any incorrect entries. Therefore, you shouldn't be at all surprised that the data entry module is a series of GOSUB statements, one for each data item to be entered.

```
300 REM   MAIN DATA ENTRY MODULE AND ERROR TESTS
310 PRINT "[CLR]":REM CLEAR SCREEN
320 GOSUB 620
330 GOSUB 670
340 GOSUB 720
350 GOSUB 770
360 GOSUB 810
```

Now you know where in the program each subroutine begins. Practice what you have learned by writing the first subroutine for entering a name. Test for no entry and entry too long. Include another statement to pad the name with spaces if necessary for a proper fit into the name field.

620 _____

630 _____

640 _____

650 _____

660 _____

```
620 PRINT : INPUT "ENTER NAME "; N$
630 IF N$ = "" THEN PRINT "NO ENTRY MADE. PLEAS
    E ENTER THE NAME.": GOTO 620
640 IF LEN(N$) > 20 THEN PRINT "ABBREV. TO 20 C
    HARACTERS OR LESS." : GOTO 620
650 IF LEN(N$) < 20 THEN LET N$ = LEFT$(N$ + "
                              ", 20 )
660 RETURN
```

Now write the subroutine for entry of the street address. Use the same tests and a "padding" statement.

Many people will find it faster to use the Commodore screen editor to change line numbers and variable names in otherwise identical statements in the subroutines, instead of retyping the new lines. Again, we encourage you to consult the Commodore manuals for more information about editing BASIC program lines.

(a) 670 _____

680 _____

690 _____

700 _____

710 _____

(a)
```
670 PRINT : INPUT "STREET ADDRESS "; A$
680 IF A$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
    ER THE STREET." : GOTO 670
690 IF LEN(A$) > 20 THEN PRINT "ADDRESS TOO LON
    G-20 LETTERS OR LESS" : GOTO 670
700 IF LEN(A$) < 20 THEN LET A$ = LEFT$(A$ + "
                              ", 20 )
710 RETURN
```

The next subroutine will be for entering the city name. Remember to check the introductory module for testing and padding parameters.

720 _____

730 _____

740 _____

750 _____

760 _____

```
720 PRINT : INPUT "CITY NAME "; C$
730 IF C$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
    ER THE CITY." : GOTO 720
740 IF LEN(C$) > 20 THEN PRINT "CITY TOO LONG--
    20 LETTERS OR LESS" : GOTO 720
750 IF LEN(C$) < 20 THEN LET C$ = LEFT$(C$ + "
                          ", 20 )
760 RETURN
```

Now for the state abbreviation, which must have exactly two characters. Test for no entry and length of entry.

770 _____

780 _____

790 _____

800 _____

```
770 PRINT : INPUT "2-LETTER STATE ABBREVIATION
    "; S$
780 IF S$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
    ER THE CITY." : GOTO 770
790 IF LEN(S$) <> 2 THEN PRINT "PLEASE USE STAN
    DARD 2 LETTER CODE" : GOTO 770
800 RETURN
```

The last data entry subroutine if for—you guessed it—the zip code. Check for no entry and proper number of characters.

810 _____

820 _____

830 _____

840 _____

- - - - - - - - - - - - - - - - - - -

```
810 PRINT : INPUT "ZIP CODE (5 DIGITS) "; Z$
820 IF Z$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER THE ZIP." : GOTO 810
830 IF LEN(Z$) <> 5 THEN PRINT "PLEASE ENTER EX
     ACTLY 5 DIGITS." : GOTO 810
840 RETURN
```

If you have learned to use the BASIC program line editor, you can speed up your program entry and debugging time considerably. You should be getting a feel for writing data entry and data testing statements by now.

Next comes another main section of the program, a module to display the entered dataset so that corrections may be made if needed.

Complete the data display in the form of a menu of choices. Check the sample RUN for the output requirements.

```
370 REM DISPLAY DATA FOR CORRECTIONS
380 PRINT "[CLR]" : PRINT "YOU HAVE ENTERED THE
      FOLLOWING DATA:" : PRINT
```

390 _____

400 _____

410 _____

420 _____

430 _____

440 _____

- - - - - - - - - - - - - - - - - - -

```
370 REM DISPLAY DATA FOR CORRECTIONS
380 PRINT "[CLR]" : PRINT "YOU HAVE ENTERED THE
      FOLLOWING DATA:" : PRINT
390 PRINT "- 1 - "; N$
400 PRINT "- 2 - "; A$
410 PRINT "- 3 - "; C$
420 PRINT "- 4 - "; S$
430 PRINT "- 5 - "; Z$
440 PRINT "- 6 - NO CHANGES"
```

Next, the user is asked to select a choice from the menu.

```
450 PRINT : PRINT "CHECK FOR ERRORS. ENTER NUMB
     ER OF ITEM"
460 PRINT "THAT NEEDS CORRECTING. IF NO CHANGES
     ARE NEEDED, ENTER NUMBER 6."
470 INPUT "ENTER YOUR SELECTION (1 TO 6) "; R$
```

Because an entry of other than 1 to 6 would cause a program error, write a statement to detect and avoid this problem.

480 _____

_____

```
480 IF VAL(R$) <1 OR VAL(R$) >6 THEN PRINT"ENTE
     R A NUMBER FROM 1 TO 6.":GOTO450
```

For any choice from 1 to 5, the program should branch to the appropriate data entry subroutine for reentry of the incorrect data item. Complete this statement:

500 ON VAL(R$) GOSUB _____ : GOTO 370

---

500 ON VAL(R$) GOSUB 620, 670, 720, 770, 810 : GOTO 370

For choice 6 from the menu (no changes in the data set), the computer should be directed to concatenate the data set into one (fielded) string, and display that string. Note that the statement detecting an entry of 6 must precede the ON . . . GOSUB . . . statement, or an error in line 500 could result.

```
490 IF VAL(R$) = 6 THEN 520
500 ON VAL(R$) GOSUB 620, 670, 720, 770, 810 :
     GOTO 370
510 :
520 LET D$ = N$ + A$ + C$ + S$ + Z$
530 :
540 PRINT  "[DOWN][DOWN]" D$ "[DOWN][DOWN]"
```

To complete this program, a section to allow the user a choice to enter more data or not is needed. Check the last segment of the sample RUN, and complete lines 560 and 570.

560 _____

570 _____

```
560 PRINT : INPUT "DO YOU HAVE ANOTHER NAME TO ENTER "; R$
570 IF LEFT$(R$,1) = "Y" THEN PRINT : GOTO 320
```

If you are using lowercase, you want to check for the entry of the shifted character (capital letter):

```
560 input "[down]Do you have more data to enter ";
      r$
570 if left$(r$,1) = "y" or left$(r$,1) = "Y" then
      print : goto 320
580 :
```

Because the subroutines are placed at the end of the program, a statement to separate the rest of the program from the subroutines is necessary. We placed an END statement at line 610, just before the subroutines start.

Enter, RUN, and debug (correct) the program. When you are satisfied that it is correct, SAVE it for use in Chapter 4.

While we omit this re-display feature in most of the example programs for the sake of brevity for typing in the programs, we highly recommend such a double-checking routine in the final version of any programs you develop from examples in this book. This is especially the case when someone other than yourself is to use the programs.

```
100 REM ENTER DATA; THEN REDISPLAY FOR CORRECTI
     ONS
110 :
120 REM  VARIABLES USED
130 REM  N$ = NAME (20 CHAR. MAX.)
140 REM  A$ = ADDRESS (20 CHAR. MAX.)
150 REM  C$ = CITY (20 CHAR. MAX.)
160 REM  S$ = STATE (2 CHAR. )
170 REM  Z$ = ZIP CODE (5 CHAR. )
180 REM  D$ = ENTIRE DATASET CONCATENATED (57 C
     HAR.)
```

```
190 :
200 :
210 :
220 :
230 :
240 :
250 :
260 :
300 REM  MAIN DATA ENTRY & ERROR TESTS
310 PRINT "[CLR]" : REM CLEAR SCREEN
320 GOSUB 620
330 GOSUB 670
340 GOSUB 720
350 GOSUB 770
360 GOSUB 810
370 REM DISPLAY DATA FOR CORRECTIONS
380 PRINT "[CLR][DOWN]YOU HAVE ENTERED THE FOLL
     OWING DATA:[DOWN]"
390 PRINT "- 1 - "; N$
400 PRINT "- 2 - "; A$
410 PRINT "- 3 - "; C$
420 PRINT "- 4 - "; S$
430 PRINT "- 5 - "; Z$
440 PRINT "- 6 - NO CHANGES"
450 PRINT "[DOWN]CHECK FOR ERRORS. ENTER NUMBER
      OF ITEM"
460 PRINT "THAT NEEDS CORRECTING. IF NO CHANGES
      ARE NEEDED, ENTER NUMBER 6."
470 INPUT "[DOWN]YOUR SELECTION (1 TO 6) "; R$
480 IF VAL(R$) <1 OR VAL(R$) >6 THEN PRINT"ENTE
     R A NUMBER FROM 1 TO 6.":GOTO450
490 IF VAL(R$) = 6 THEN 520
500 ON VAL(R$) GOSUB 620, 670, 720, 770, 810 :
     GOTO 370
510 :
520 LET D$ = N$ + A$ + C$ + S$ + Z$
530 :
540 PRINT "[DOWN][DOWN]" D$ "[DOWN][DOWN]"
550 :
560 INPUT "[DOWN]DO YOU HAVE ANOTHER NAME TO EN
     TER "; R$
570 IF LEFT$(R$,1) = "Y" THEN PRINT : GOTO 320
580 IF LEFT$(R$,1) <> "N" THEN PRINT "YES OR NO
     ":GOTO560
590 :
600 :
610 END
```

```
620 INPUT "[DOWN]NAME "; N$
630 IF N$ = "" THEN PRINT "NO ENTRY MADE. PLEAS
     E ENTER THE NAME.": GOTO 620
640 IF LEN(N$) > 20 THEN PRINT "ABBREV. TO 20 C
     HARACTERS OR LESS." : GOTO 620
650 IF LEN(N$) < 20 THEN LET N$ = LEFT$(N$ + "
                          ", 20 )
660 RETURN
670 INPUT "[DOWN]STREET ADDRESS "; A$
680 IF A$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER THE STREET." : GOTO 670
690 IF LEN(A$) > 20 THEN PRINT "ADDRESS TOO LON
     G--20 LETTERS OR LESS" :GOTO 670
700 IF LEN(A$) < 20 THEN LET A$ = LEFT$(A$ + "
                          ", 20 )
710 RETURN
720 INPUT "[DOWN]CITY NAME "; C$
730 IF C$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER THE CITY." : GOTO 720
740 IF LEN(C$) > 20 THEN PRINT "CITY TOO LONG--
     20 LETTERS OR LESS" : GOTO 720
750 IF LEN(C$) < 20 THEN LET C$ = LEFT$(C$ + "
                          ", 20 )
760 RETURN
770 INPUT "[DOWN]2-LETTER STATE ABBREVIATION ";
      S$
780 IF S$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER STATE." : GOTO 770
790 IF LEN(S$) <> 2 THEN PRINT "PLEASE USE STAN
     DARD 2 LETTER CODE" : GOTO 770
800 RETURN
810 INPUT "[DOWN]ZIP CODE (5 DIGITS) "; Z$
820 IF Z$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER THE ZIP." : GOTO 810
830 IF LEN(Z$) <> 5 THEN PRINT "PLEASE ENTER EX
     ACTLY 5 DIGITS." : GOTO 810
840 RETURN
```

Many error-checking procedures depend on personal preference or company policy. Either way, plan ahead. Look carefully at the complete problem or job for which you are using your computer. In what form and format should the data be entered? Are there subtle limits or tests that you can apply to data to detect operator errors? For instance, if you are entering addresses with zip codes and a large percentage of your business is in California, then you know that most zip codes should start with the number 9. It would be appropriate to test whether the entered zip code value begins with a 9, and if not, to inform the operator of a *possible* error.

```
140 INPUT "[DOWN]ZIP CODE "; Z$
150 IF LEN(Z$) <> 5 THEN PRINT "PLEASE ENTER EX
     ACTLY 5 DIGITS." : GOTO 140
160 IF LEFT$(Z$,1) = "9" THEN 230
170 PRINT "THE ZIP CODE "Z$" IS NOT IN CALIFORN
     IA."
180 INPUT "IS IT CORRECT "; R$
190 LET R1$ = LEFT$(R$,1)
200 IF R1$ = "Y" THEN 230
210 IF R1$ = "N" THEN PRINT "PLEASE REENTER." :
      PRINT : GOTO 140
220 PRINT "PLEASE TYPE 'Y' FOR YES OR 'N' FOR N
     O" : LET R$ = "" : GOTO 140
230 REM PROGRAM CONTINUES

RUN
ZIP CODE ? 234567
PLEASE ENTER EXACTLY 5 DIGITS.

ZIP CODE ? 23456
THE ZIP CODE 23456 IS NOT IN CALIFORNIA.
IS IT CORRECT ? N
PLEASE REENTER.

ZIP CODE ? 93456
READY.
```

Since you don't know if a person using your program on a Commodore 64 may have pressed the SHIFT-Commodore keys to change case (or have the SHIFT-LOCK key down on a CBM or PET), it is a good idea to force the mode you want (see Appendix 4). Even so, it is worth considering that the user may use the SHIFT key when entering data. The routine in lines 190–220 illustrate one way to handle this.

We also strongly recommend consistency in your data entry formats, especially for such things as data field lengths. Don't confuse yourself or others who use your programs. If you write several programs that use personal names, use the same size delimiters or data fields. This also allows you to have compatible data

files for various uses. The same goes for address sizes and formats, product descriptions, and other alphanumeric data. Remember, your company may have already made the decision for you, so be sure you know the policies!

For numeric values, quantities, and entries involving monetary values, you may have to dig a little to discover the limits for which the data should be tested. Company policy, common sense, and actual experience may give you the logical limits for a "not less than" or "not to exceed" data entry check. And you can always use the technique to redisplay the data for user corrections after initial data entry, as shown earlier.

Let's review the general data entry error-checking procedures for alphabetic and numeric information.

1. Enter all data into string variables after a clearly stated prompt request from the computer.
2. Enter only one data item per prompt.
3. If you are going to pack a number of data items (a dataset) into one string, enter the data into separate string variables and then concatenate after all checking has been accomplished. Do not enter data directly into a substring position.
4. Checking should include a test for non-response of the type IF LEN(R$)=0 . . . or IF R$ ="".
5. When an error is discovered, include a message not only to tell the operator that an error was made, but also to describe as completely as possible what the error was. Do not merely request a reentry.
6. Check alphabetic data for field length using the LEN function.
7. It may be necessary to pad the entry with spaces to the proper field length, especially for alphabetic data.
8. Thoroughly test numeric data (which we recommend be entered into a string variable) in this order:
   (a) For non-response.
   (b) For excess string length, if applicable.
   (c) For the inadvertent inclusion of alphabetic characters in numeric values, using ASC.
   (d) For any company policy tests or size limit.
   (e) If the datum is an integer value, use the integer declaration (a percent sign following a variable in many BASICs) or test the value to see if it is an integer with a statement like IF X < > INT(X). . . .
   (f) Test for negative values if they are not acceptable. If this sounds like a lot of work, remember that your otherwise excellent program must have valid and accurate data to do its job. Don't skimp. Be complete. For example, the capability of the IF . . . THEN statement to PRINT a message may lull you into trying to oversimplify an error message in order to fit into the same programming line as the IF . . . THEN statement. Don't fall into this trap. Use GOSUBs and provide complete, clear messages to the operator.

You may want to place all error tests and messages in subroutines. This gives your program neatness and clarity. Various entries may be put to the same tests, allowing the check statements to work for various entries if variables and other factors are compatable.

Be alert to other occasions throughout your programs where data errors may occur. While we encourage sensitivity to errors at data entry time, always check for data errors later in your program, especially if the data are subject to various manipulations after the entry routines.

Watch for strange results from functions such as VAL. Get to know the version of BASIC you are using inside and out by thoroughly exploring the reactions of statements and functions in various circumstances.

Have your Commodore reference manuals at hand. The error conditions you encounter will depend largely on your programming skills and the kinds of applications you program. Be alert to the errors that occur and include tests for them. Don't get psychologically locked into your first, second, or third version of a program or programming technique.

Finally, be aware that many programmers test their programs with only sensible data, neglecting the ridiculous mistakes that can, and without a doubt will, be made. When you think you have covered every possibility, let someone with no computer experience try it out. If the program survives, you've checked it all out!

Chapter 3 Self-Test

1. Write an IF . . . THEN comparison that will be true if:
   (a) The entry has exactly seven characters.
   (b) The entry does not have exactly seven characters.
   (c) The first character in an entry is not a number.
   (d) The first character in an entry is a number other than zero.
   (e) The entry is not a null string.

   (a) _____

   (b) _____

   (c) _____

   (d) _____

   (e) _____

2. Write a statement line that checks to see if an entry has fewer than twelve characters, and if so, pads the entry with spaces so that the resulting string has exactly twelve characters.

   _____

   _____

3. Write a data entry checking routine that checks to see that no numbers have been included in a string entry. Write an accompanying subroutine to be called when a number is found that tells the user what was entered, and to reenter without including numbers in the entry.

4. You now have the background to write a data entry module for most kinds of data to be later placed into a data file (covered in the next chapter). Write the data entry module and complete it with data entry error checks, as described below:
   (a) Write a data entry routine that prompts the user to enter:
      1. a five-character alphanumeric product code (must always have five characters),
      2. a product name with a twelve-character maximum length,
      3. the quantity ordered into a three-digit field with a limit of 288 per order,
      4. the price, into a five-character field, with no price exceeding $99.99.

```
RUN
PRODUCT CODE (5 CHARACTERS)? 34567
PRODUCT NAME (12 CHARACTERS MAXIMUM)? HEATER
QUANTITY ORDERED? 65
PRICE PER UNIT (99.99 MAX.)? 98
```

(b) Pack the information entered into one long string (M$) with the following fields:

```
MS = _ _ _ _ _ / _ _ _ _ _ _ _ _ _ _ _ _ _ / _ _ _ / _ _ _ _ _
       C$                N$                Q$       P$
```

Note: do not include slashes in the data field string.

(c) Print parts of M$ in a "report" with the format shown below. (Only three of the four data items in the data set are to be displayed.)

```
PRICE        QUANTITY     PROD. CODE
$98          65           34567
```

Refer back through this chapter for ideas, and try debugging your solution program before looking at our way of doing it. Our solutions are not the only ones possible. The real test is whether the program works, and how foolproof it is. The introductory module to our solution is shown below. You complete the program.

```
100 REM   SOLUTION FOR CHAPTER 3 SELF TEST PROBLEM 4
110 :
120 REM  VARIABLES USED
130 REM    C$ = PRODUCT CODE (5 CHARS.)
140 REM    N$ = PRODUCT NAME (12 CHARS. MAX.)
150 REM    Q$ = QUANTITY ORDERED (3 DIGITS MAX.)
160 REM    P$ = PRICE (99.99 MAX.)
170 REM    M$ = CONCATENATED DATASET
180 :
190 REM  DATA ENTRY AND TESTING MODULE
200 :
```

Answer Key

1. (a) `IF LEN(A$) = 7 THEN . . .`
   (b) `IF LEN(A$) <!> 7 THEN . . .`
   (c) `IF ASC(A$) < 48 AND ASC(A$) > 57 THEN . . .`
   (d) `IF VAL(A$) <!> 0 THEN . . .`
   (e) `IF LEN(A$) <!> 0 THEN . . .`

2. `120 IF LEN(A$) < 12 THEN LET A$ = A$ + LEFT$(A$+"                    ",12)`

   (Your string variable and line number may be different, of course.)

3.
```
100 REM PROBLEM 3-3
310 INPUT "[DOWN]YOUR NAME "; A$
320 FOR X = 1 TO LEN(A$)
330 A = ASC(MID$(A$,X,1))
340 IF A >= 47 AND A < 58 THEN GOSUB 1100 : GOT
    O 310
350 NEXT X
1000 END
1100 PRINT "[DOWN]YOU ENTERED: [RVS]" A$
1120 PRINT "PLEASE REENTER BUT DON'T INCLUDE NUM
    BERS"
1130 RETURN
```

```
100 REM PROB 3-3/ALTERNATE
120 REM  ALTERNATE VERSION USING STRINGS
130 REM  INSTEAD OF ASCII
310 INPUT "[DOWN]YOUR NAME "; A$
320 FOR X = 1 TO LEN(A$)
330 IF MID$(A$,X,1) >= "0" AND MID$(A$,X,1) <=
    "9" THEN GOSUB 1100 : GOTO310
340 NEXT X
1000 END
1100 PRINT "[DOWN]YOU ENTERED: [RVS]" A$
1120 PRINT "PLEASE REENTER BUT DON'T INCLUDE NUM
    BERS"
1130 RETURN
```

4. *Note:* Our solution does not cover all the bases in checking for input errors. Here are some other checks you may have thought to include: for negative quantities, for fractional quantities (using the INT function), for numbers instead of letters entered for the state abbreviation characters, and for a zip code of 00000 or for nine-digit codes. You could also set up a fifty-element string array or a 100-character string containing the legal two-letter combinations for state abbreviations and check to see that the state code entered by the user matched one of the allowed two-character combinations.

```
100 REM PROBLEM 3-4
110 :
120 REM VARIABLES USED
130 REM    C$= PRODUCT CODE (5 CHRS.)
140 REM    N$= PRODUCT NAME (12 CHRS MAX)
150 REM    Q$= # ORDERED (3 DIGITS MAX)
160 REM    P$= PRICE (99.99 MAX)
170 REM    M$= CONCATENATED DATASET
180 :
190 REM DATA ENTRY AND TESTING MODULE
200 :
210 INPUT "[DOWN]PRODUCT CODE (5 CHARACTERS) ";
      C$
220 IF C$ = "" THEN PRINT "NO ENTRY. PLEASE ENT
     ER THE CODE." : GOTO 210
230 IF LEN(C$) <> 5 THEN PRINT "CODE MUST HAVE
     5 CHARACTERS" : GOTO 210
240 INPUT "[DOWN]PRODUCT NAME (12 CHARS. MAX) "
     ; N$
250 IF N$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER THE NAME." : GOTO 240
260 IF LEN(N$) > 12 THEN PRINT"ENTRY TOO LONG--
     12 CHARACTERS OR LESS" : GOTO240
270 IF LEN(N$) < 12 THEN LET N$ = LEFT$(N$ + "
                ", 12 )
280 INPUT "[DOWN]QUANTITY ORDERED "; Q$
290 IF Q$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER QUANTITY." : GOTO 280
300 IF VAL(Q$) = 0 THEN PRINT "ENTRY ERROR--NUM
     BERS ONLY!" : GOTO 280
310 IF LEN(Q$) > 3 THEN PRINT "TOO MANY DIGITS-
     -3 MAX" : GOTO 280
320 IF VAL(Q$) > 288 THEN PRINT "ORDER TOO LARG
     E. 288 MAX." : GOTO 280
330 IF LEN(Q$) < 3 THEN LET Q$ = LEFT$(Q$ + "
      ", 3 )
340 INPUT "[DOWN]UNIT PRICE (99.99 MAX) "; P$
350 IF P$ = "" THEN PRINT "NO ENTRY--PLEASE ENT
     ER PRICE." : GOTO 340
360 IF VAL(P$) = 0 THEN PRINT "ENTRY ERROR--NUM
     BERS ONLY!" : GOTO 340
370 IF VAL(P$) => 100 THEN PRINT"PRICING ERROR!
      99.99 MAX." : GOTO 340
380 IF LEN(P$) < 5 THEN LET P$ = LEFT$(P$ + "
        ", 5 )
390 :
400 LET M$ = C$ + N$ + Q$ + P$
410 :
420 PRINT "[CLR]" : REM CLEAR SCREEN
430 PRINT "PRICE" , "QUANTITY" , "PROD. CODE"
440 PRINT "$" RIGHT$(M$, 5),
450 PRINT MID$(M$, 18, 3),
460 PRINT LEFT$(M$, 5)
```

CHAPTER FOUR

Cassette Tape Data Files

Objectives: When you complete this chapter, you will be able to store and retrieve information from sequential cassette tape files. You will use the following BASIC data file statements in their special formats: OPEN, CLOSE, INPUT#, GET#, and PRINT#.

INTRODUCTION

If you have a disk drive, *do not* read this chapter! Proceed to Chapter 5, which covers the same material, tailored for disk drive files.

This chapter teaches you to manipulate sequential data files on cassettes. It is a different operation than using SAVE or LOAD with BASIC programs on cassette. This chapter will provide all the information you need to successfully write and use sequential cassette data files. At the end of this chapter we will show you how to write a data file program using either disk or tape.

CASSETTE VERSUS DISKS

If you are going to be creating and using large files, or files which need constant revision, consider buying a disk drive. The increase in speed and ease of manipulation of files, and the ability to use random access files, are the big advantages of disk systems. If you don't need the speed, and your files are small enough to fit in the computer's memory, then a cassette system will suffice. Commodore's cassette, while slow, is extremely reliable under normal use with proper care.

WHAT IS A DATA FILE?

A data file is stored alphanumeric information that is separate and distinct from any particular BASIC program. It is located (recorded) on either a magnetic disk, diskette, or cassette tape. This chapter discusses using sequential (also called serial) data files on cassette tape.

In your previous BASIC programming experiences you probably hand-entered all data needed by your programs using INPUT statements. You did this each time you ran your programs. Or, if you had larger amounts of data, you might have entered the data with DATA statements and used the READ statement to access and manipulate the data. In either case, the data were program-dependent; that is, they were part of that one program and not useable by other programs.

A data file is *program-independent*. It is *separate* from any one program and can be accessed and used by many different programs. In most cases, you will use only one program to load a data file with information. But once your data file is loaded (entered and recorded) on disk or cassette tape, you can read the information from that file using many different programs, each performing a different activity with that file's data.

For example, perhaps you have computerized your personal telephone and address directory using data files stored on a disk. You may need just one program to originally load information into that file and add names to it. (This chapter will show you how.) Another program allows you to select phone numbers from the file using NAME as the selection criterion. You can use still another program to change addresses or phone numbers for entries previously made in the file. Another program could print gummed mailing labels in zip code order using the same data file. You could design yet another program to print names and phone numbers by phone number area code. The possibilities go on and on.

Notice that one data file can be accessed by many different computer programs. The data file is located separately on the disk or cassette tape in a defined place. Each program mentioned copies the information from the disk or cassette tape into the electronic memory of the computer as it is needed by that particular program. Alternatively, the program could transfer information from the computer's memory to be recorded onto the disk or cassette tape.

If you already use your tape to SAVE and/or LOAD BASIC programs, then you have some experience with tape files. When you SAVE a BASIC program, it is recorded on this tape in a file. Such files containing BASIC programs are called *program files*. In contrast, the files discussed in this chapter contain data and are therefore called *data files*. The two types of files are different and are used differently. A BASIC program file contains a copy of a BASIC program that you can LOAD, RUN, LIST, and SAVE. A data file contains information only. You access this information using a BASIC program that includes special BASIC statements that access data files; that is, transfer all or part of the data from the magnetic recording on disk or cassette into the computer's electronic memory so the program can use it. You *cannot* LOAD, RUN, LIST, or SAVE a data file. You access the information using a BASIC program instead.

(a) Describe in general terms how you can access data in a data file.

_____

_____

_____

(a) By using a BASIC program that includes special file-accessing BASIC statements.

DATA STORAGE ON CASSETTE TAPE

The format of data storage on a cassette tape, overall, is similar to the storage of a BASIC program. Commodore first writes a 10-second leader, to make sure the tape has wound past its leader, and to provide a significant separation between programs (or files). It then writes header information, consisting of the title and some other information. The header is followed by a short leader tone, then the first block of data. Each block of data is 191 characters long, the length of the cassette buffer in the computer's memory. The header and each block of data is recorded twice onto the tape. When reading the tape, the computer reads the first section, then compares it with the duplicate. While this process makes Commodore's cassette slow compared with others, it also makes it extremely reliable, as errors in the first section can be corrected by the second. At the end of the data, a block is written containing a marker to indicate the end of data.

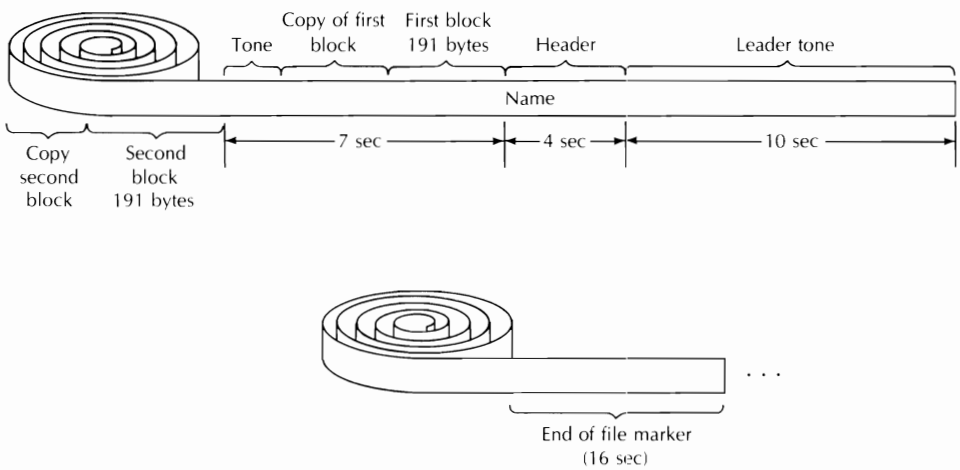


Figure 4–1

How much data can a cassette tape hold? One side of a C-10 cassette is 5 minutes long. One block of 191 bytes takes about 7 seconds to write. This is about 40 seconds per thousand bytes (K), or 6 minutes per 8K. The header and closing block each take about 20 seconds. While you could store two files of 4K on one side of a C-10 cassette, we recommend you store each file on one side of a cassette. It is easier to find, and reduces the chance of accidentally erasing the wrong file (for example, when you forget to rewind the tape after reading the file). Using the approximation of 6 minutes = 8K, we can construct a table of length and storage:

Data	Tape Length
4K	3 min. (C-10)
8K	6 min. (C-15 or C-20)
16K	12 min. (C-30)
24K	18 min. (C-46 or C-60)

It is a good idea to use a tape that is at least as long as the values given here; for example, to store a 24K data file, you should use at least a C-46 to be sure there is enough room on the tape.

For a personal telephone and address directory application, let's see how much storage space is required for each person on file. Each data item has a defined field length.

Name	20	characters
Address (street)	25	
City	10	
State	2	
Zip code	5	
Phone(xxx-xxx-xxxx)	12	
Age	2	(Entered as an integer number)
Birthdate(xx/xx/xx)	8	
Subtotal	84	
Overhead	8	
Total	92	

Note that string character values are used for the zip code.

(a) How many bytes would be required to store the zip code as an integer value instead of a string? _____

(b) Why was a twelve-character string rather than a numeric value used for the phone number? _____

_____

(c) How long a tape would you need to store 150 entries in the address and phone directory? _____

(a) Seven plus overhead. In fact, Commodore stores numeric information as if it were a string.
(b) To include the hyphen(s) or space(s).
(c) 92 × 150 = 13,800 bytes. You would need a C-30 cassette.

CASSETTE CONSIDERATIONS

Low-noise tape seems to work best; buy the best quality you can. Saving 50 cents by buying poor quality tape is silly when compared with the value of the time you spend in creating and saving data on tape: don't skimp! You do not need to buy leaderless tape; most tape leaders are about 4 seconds long, and Commodore waits for 10 seconds before writing the header.

Once you have written data to a cassette, you can protect it from accidental erasure by poking out the tab at the back of the cassette. When you lift the cassette out of the recorder, the tab to remove is at the *left* side.

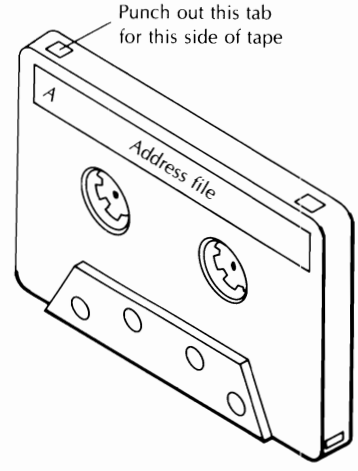


Figure 4–2

If you later want to reuse the tape, you can put a piece of Scotch tape over the hole. Keeping your tapes clean and cool will also extend their life; use the plastic boxes supplied, and keep them away from sunlight, heaters, and off of computers and other sources of heat.

We recommend you *immediately* label and punch out the tab of every tape you use, so you'll never wonder what is on the tape.

Before you first use a cassette tape, some people recommend you fast-forward then rewind it to wind it uniformly, and to eliminate any possible stretching as you use it. Since reading data depends upon correct timing, any stretching may make your data unreadable.

It is also worth keeping your tape recorder in good condition. Buy, and use regularly, a cassette cleaner. Also, buy and use demagnitizer; it does make a difference. When you use the demagnitizer, follow the instructions carefully, and make sure your tapes are at least 3 feet away. We take our tape recorder into

another room to demagnitize it, since the demagnitizer will also demagnitize—erase—tapes. If you begin to have ?LOAD ERROR or ?VERIFY ERROR problems, you may need to have your tape recorder realigned. With heavy use, we had our tape recorder realigned every four to six months; with less use, we recommend yearly realignment as preventive maintenance.

There is one last factor in using cassettes, which will be repeated later. To start sending data to the recorder, Commodore simply checks to see if *any* key on the Datasette has been pressed. If you press PLAY and RECORD to read your data, or FAST FORWARD to save them, the computer and cassette recorder will go their merry ways, but your data will be destroyed. Always include *very explicit* cassette operation instructions to the user in your programs, and do be careful yourself! We have more than once pressed PLAY instead of PLAY and RECORD to "save" data, only to find nothing on the tape when we later went to use it. This is probably the biggest problem with cassette files—they are very susceptible to operator error!

Here are some sample messages; you'll see them in the example programs:

```
INSERT CORRECT CASSETTE TAPE.
REWIND THE TAPE AND PRESS 'STOP' KEY ON
RECORDER.
PRESS 'RETURN' TO CONTINUE.
                    .
                    .
                    .
PRESS 'STOP' KEY ON RECORDER.
REWIND TAPE TO BEGINNING.
PRESS 'STOP' KEY ON RECORDER.
PRESS 'RETURN' KEY TO CONTINUE.
```

Remember that there is a STOP key on the computer: you must remind your users to press the key on the tape recorder, not the one on the computer, or they will stop your program! You can insert the file name in the first prompt, if cassettes are labeled, to further decrease the chance that the user will insert the incorrect cassette.

If an OPEN, INPUT#, PRINT#, or CLOSE statement is encountered and no key on the tape recorder is depressed, an appropriate message will be displayed, depending on the last OPEN to that file number. For a read file, the message will be

```
PRESS PLAY ON TAPE #1   (PET)

PRESS PLAY ON TAPE      (C-64)
```

For a write file, the message will be

```
PRESS PLAY AND RECORD ON TAPE #1   (PET)

PRESS RECORD & PLAY ON TAPE        (C-64)
```

Since pressing the keys at the wrong time will cause the tape to wind endlessly, take advantage of these prompts for the actual writing and reading of data. It is dangerous to provide the "PRESS PLAY ON TAPE" prompt yourself, as the motor in the cassette recorder may not be under computer control. This does not lessen the importance of the 'rewind' prompts discussed earlier.

OPENING CASSETTE FILES

To use a data file, you must first OPEN the file and indicate its name. This can be done as part of the program's initialization module.

The file name can be up to about 190 characters, but it is reasonable to use short names that mean something, so you can keep your files straight. Disk files are limited to names sixteen characters long; for future compatability, you might want to consider that a limit.

The OPEN statement tells the computer which data file this particular program will use. It also provides the computer with other information. The OPEN statement creates a new file if none exists with the name specified, and sets the access mode.

There are two modes for accessing sequential data files—READ and WRITE. They are set by the SECONDARY ADDRESS used in the OPEN statement. An OPEN sequential file will be either READ or WRITE; it can be open for only *one* of these modes at a time.

To change from WRITE to READ, you must CLOSE the file, rewind the tape, and then REOPEN the file. To change from READ to WRITE, you must again CLOSE the file; you can either WRITE to a new tape, saving both files, or you can rewind and WRITE to the same tape, permanently erasing the old file.

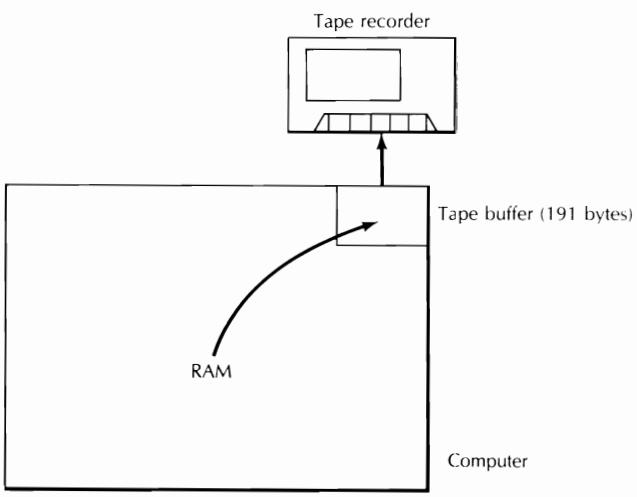


Figure 4–3 Data flow through buffers.

OPEN also assigns space in the computer's memory, called a buffer, to the file. The tape file buffer is 191 characters long. There is only one tape buffer in the C-64; you can only access *one* file at a time. This buffer is permanently assigned to *device one*.

In PETs and CBMs, there are two tape buffers, and two separate tape connectors; each buffer is permanently assigned to a specific connector. Tape unit (connector) one is the accessable one; the unit two connector is inside the case.

All information moving between the tape recorder and the computer goes through the buffers. When you write information to a file, it is accumulated in the buffer until the buffer is full; only then is it written to the tape. Therefore, you may type for a while without any tape activity! The same happens when reading: 191 characters are read into the buffer, and used as requested by the program. When they are all used, then another block of data is read. The consequences of this will be apparent when the CLOSE statement is discussed.

THE OPEN STATEMENT

Now let's look at the OPEN statement, and practice writing statements that include the information needed by the computer to deal with sequential data files in a program.

OPEN has the following format:

```
OPEN 2, 1, 1, "NAME"
```

The first number following the OPEN (2) is the *file number*. Any reference to the file later in the program will use this number. Since you can only open one file to tape, it is convenient to *always* use *one* for the file number. The second number specifies the *device;* 1 is the device number for the first tape recorder. For PETs and CBMs, the second tape recorder is device number two. The third number is called the *secondary address*. It sends specific information to the computer regarding how the file is to be used. Secondary addresses are

Secondary Address Number	Explanation
0	READ
1	WRITE, with END OF FILE mark
2	WRITE, with END OF TAPE mark

If the file does not exist, or if the file name doesn't exactly match the name on the tape, the tape will merrily wind to its end.

Commodore allows default values in the cassette file OPEN statement. We recommend you specify the explicit values, since you may forget what the default values are and destroy your files. The default values are device number one, and read (secondary address 0). Therefore, you could open a file with

```
OPEN 1
```

With tape, the file name is optional, since the computer will simply load the first file it comes to, unless a name is given. Again, we recommend you explicitly supply a name, both when reading and writing. Also, recording just one file on a tape, and labeling the tape, will help you keep your files straight.

END OF FILE/END OF TAPE MARKS

The end of file/end of tape marks have the function of telling your computer when to stop looking for data. When searching for a file name, if the *end of tape* mark is encountered before the file name is found, the computer will stop with the message "?FILE NOT FOUND ERROR." If you consistently put just one file on a tape, you should use this mark (secondary address 2). If your recorder is having trouble reading the tape, it usually will stop with a "?FILE NOT FOUND ERROR" when it encounters the END OF TAPE mark, saving you hours of watching the tape go around.

The *end of file* mark simply tells the computer that it has reached the end of a file. If the file has been properly recognized, the computer will stop reading data from the tape when it reaches this mark. When searching for a file, if the name doesn't match, the computer will read past the end of file marker to look for another file on the tape. If you have three files on a tape (you wouldn't do that, would you?), to read data from the third file, the first two must end with end of file markers, since an end of tape marker will cause the computer to stop reading.

The file name can be assigned as a string (F$):

```
100 INPUT "FILE NAME ";F$
110 OPEN 1, 1, 1, F$
```

You can even allow the user to enter all the file information, by assigning all the parts of an open statement as strings or variables:

```
100 INPUT "FILE NAME ";F$
110 INPUT "FILE NUMBER ";N
120 INPUT "MODE (R OR W) ";M$
130 M = 1 : IF M$ = "R" THEN M = 0
140 OPEN N, 1, M, F$
```

Line 130 produces a secondary address of 1 (end of file mark) for writing, and if an R is entered, sets the secondary address to 0 for reading. In a real program, you would undoubtedly include error-checking statements for all user inputs.

(a) Open a read (or input) file named PHONES, assigned to file number 1.

(b) Open a write (or output) file with end of file marker, with a user-designated name stored in F$, with file number 3.

(c) Open an input (read) file with a user-designated name and file number.

(a)
```
110 OPEN 1, 1, 0, "PHONES"
```

or

```
110 OPEN 1, "PHONES"
```

(b)
```
140 INPUT "FILE NAME "; F$
150 OPEN 3, 1, 1, F$
```

(c)
```
140 INPUT "FILE NAME "; F$
150 INPUT "FILE NUMBER "; N
160 OPEN N, 1, 0, F$
```

THE BUFFER PROBLEM: CLOSING THE FILE

Every file that is opened with an OPEN statement must be closed with a CLOSE statement before the program finishes. As soon as the program is through using a file, and always before the program terminates, include a CLOSE statement to transfer information from the buffer to the tape, close the file, and unassign the buffer(s).

Once a file has been closed, the same file number can be reused for any other file you open (although there are advantages to using unique file numbers for each transaction).

Here are sample CLOSE statements:

```
600 CLOSE 1
900 CLOSE 2
```

A separate CLOSE statement is needed for each file that is opened. A file must be CLOSEd and reOPENed to change from one access mode to another—for example, after writing, before opening the same or a different file for reading. Remember that the tape must also either be rewound or changed when changing access modes.

CLOSE is a vitally important statement. CLOSE, used properly, maintains the integrity and accuracy of your data files. Recall that when you OPEN a file, you move data into a buffer. When the buffer is full, data are moved to the tape; the file is UPDATED. Since a buffer contains up to 191 characters, it may be only partially full when a program terminates.

What happens if the buffer is only partly full of data and there are no more data to finish filling it? You might expect the half-full buffer to simply transfer its contents to the tape when the program finishes execution. But it won't. The data in the half-filled buffer will not necessarily be recorded into your file. Your file may not contain all the information that you expected.

One important function of the CLOSE statement is to transfer the contents of the buffer to the tape file, even though the buffer is not full. As a rule of thumb, any program with an OPEN statement should have a CLOSE statement that is always executed before the program terminates (preferably as soon as the program is through reading or writing data).

If you get trapped with a program that aborts (for example, with a "?SYNTAX ERROR") or terminates and the buffers contain some data, CLOSE can be executed in direct mode, forcing transfer of the buffer contents to the tape file. However, to have to do so indicates poor programming technique and would be completely unacceptable in a work environment. Further instructions on writing your programs to include a CLOSE statement that is always executed are given later in this chapter. They should be read *very* carefully, as you *will* run into these problems during program development!

This is a good time to review the techniques for dealing with the RETURN key being pressed without any other entry, discussed in Appendix 4.

(a) What are two purposes of the CLOSE statement?

- - - - - - - - - - - - - - - - - - -

(a) To unassign the file from the buffer, and to force the buffer to transfer its contents to the tape data file.

Under some conditions, BASIC will automatically "flush" the computer's buffers and close the files. ENDing or LISTing a program, or termination with an error condition ("?SYNTAX ERROR"), will *not* affect files: you must still type CLOSE (in direct mode) to properly close them.

NEW and RUN will close files. However, *EDITing a program will close files.* FILE ERROR conditions will also close files. To be sure, you can always type CLOSE in immediate mode; if the files have been closed, you will get the message "?FILE NOT OPEN ERROR," but no harm will be done.

You must specify the file number in the CLOSE statement. This is another reason for always using the number one (1) for the file number; you never have to try to figure out which file number your program was using when it stopped.

To repeat: Always include a CLOSE statement that is executed before the program terminates, so that buffer flushing is automatic. You should only force buffer flushing under emergency conditions, and then you should use the CLOSE statement in direct mode.

The buffer problem—and it is a real problem—makes it imperative that you *never* remove a tape from the tape recorder if the tape contains an open file. Not only will you lose some data from your un-closed file, but you may find data from a half-filled buffer placed in the wrong file on the wrong tape, which can create some nasty errors.

Be cautious, and remember that data go first to the buffer. They are transferred to the tape only when the buffer is full. If the buffer is not full, force it to transfer the data to the tape file with the CLOSE statement.

(a) If you are outputting data in a program to a data file and the program accidentally terminates without executing a CLOSE statement, what should you do?

(b) What is wrong with this statement?

```
CLOSE
```

(a) Close the file with a CLOSE statement in direct mode.
(b) CLOSE must always include a file number: CLOSE 1

PRINTING DATA TO CASSETTE FILES

Just as PRINT is used to print data to the screen, PRINT# is used to print data to files. PRINT# has the following format:

```
180 PRINT#1, A$
190 PRINT#1, "ADDRESS DATA FILES"
200 PRINT#1, N
```

Each variable should be in a separate PRINT# statement (more on that later), and must be set off by a comma. The number sign (#) must follow the word PRINT, *without a space*. Including a space will result either in a "?SYNTAX ERROR" or in data appearing on your screen instead of on your tape. The file number follows the number sign.

PRINT# sends a carriage return [CHR$(13)] after the last variable. A carriage return character should separate each data item; put only one variable in each PRINT# statement.

Several PRINT# statements can be combined in one program line by using colons:

```
180 PRINT#1, N$ : PRINT#1, B$ : PRINT#1, Q
```

While you can use other formats, there are problems inherent in their use. The two methods displayed here work consistently, for both numeric and string data. The second version has the minor advantage of placing an entire dataset on one line, visually representing the associations of data.

(a) Now it is your turn to practice writing statements to place data in a file. In each of the four program segments below, write a PRINT# statement appropriate for filing the data from that program segment's INPUT statements. Use three as the file number.

```
300 INPUT "HOW MANY SAMPLES "; S
310 INPUT "HOW MANY WERE GREEN "; G
320 PRINT# _____

500 INPUT "TODAY'S DATE "; D$
510 INPUT "CITY NAME "; C$
520 INPUT "STATE CODE "; S$
530 PRINT# _____

900 INPUT "TITLE OF BOOK "; T$
910 INPUT "FIRST LINE OF TEXT "; F$
920 INPUT "NUMBER OF PAGES "; P
930 PRINT# _____
```

(a)
```
320 PRINT#3, S : PRINT#3, G
```

or

```
320 PRINT#3, S
330 PRINT#3, G
```

(Either method can be used. Only the first is shown for the next two answers.)

```
530 PRINT#3, D$ : PRINT#3, C$ : PRINT#3, S$
930 PRINT#3, T$ : PRINT#3, F$ : PRINT#3, P
```

PUNCTUATION IN STRINGS

The hard-to-remember part of printing strings to a file with PRINT# is when you want your strings to include commas, colons, or semicolons, or to purposely include carriage returns. For example, given

```
LET B$ = "PUBLIC, JOHN Q."
```

you would expect the statement

```
PRINT#1, B$
```

executed after the B$ assignment statement to take the entire string enclosed by quotation marks and place it into the file as one string. But the quotation marks are essentially ignored. The comma actually separates PUBLIC from JOHN Q. in the file, breaking the one name into two separate data items. With the PRINT#statement, the solution is to "force" quotation marks on either side of the entire name string by using the CHR$( ) function. CHR$(34) is the ASCII code for the quote (") symbol. Notice how it is used in this program segment:

```
240   LET B$ = "PUBLIC, JOHN Q."
250   PRINT#1, CHR$(34) B$ CHR$(34)
```

As will be discussed in the INPUT# section to follow, there are still problems with this solution. You only need to worry about forcing quotation marks in a PRINT# statement when your string will also include commas, colons, semicolons, or carriage returns. That should not happen very often, and with careful planning it can be avoided entirely.

WRITING A DATA FILE

So far, we have looked at the formats for OPEN and PRINT# statements. Now let's move a step closer to the real world with two programs in BASIC: one to write to a file, and a second, separate BASIC program to read back data from the tape file to the computer's electronic memory and then to display the data.

As noted earlier, using files requires planning. Your plan should consider:

1. What to include in each dataset.
2. How large each data item or dataset will be.
3. Whether technical points, such as embedded commas in strings, must be handled with special techniques.
4. How to test each data item in the dataset as completely as possible for accuracy and validity.

With these considerations in mind, here is a program to help you place a simple inventory from your home or business into a cassette file. The introductory module and possible checks for data validity are included. The file name is PROPERTY.

For simplicity in the example program, we have arbitrarily limited the dataset to three items of data. Other information, such as brand name, model number, and serial number are other possible data for a file like this.

The following program is not complete. Read it, then do the following exercises to fill in the missing statements.

Location 50003 contains a 0 in the Commodore 64, and a 1 or 160 for most PETs (it is also 0 for BASIC 1.0 PETs). The first few lines use this value to set the lowercase; see Appendix 4 for an explanation.

```
100 rem   property inventory cassette file program
101 if peek(50003) <> 0 then poke 59468,14 : re
     m lower case for pets
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem     d$ = description (20)
140 rem     n = number of items
150 rem     v = dollar value
160 rem     cr$ = carriage return
170 rem   files used
180 rem     property = sequential file
190 :
200 rem   initialize
210 _________________________  : rem open file
220 :
230 rem   data entry routines
240 print : input "Item description "; d$
250 if len(d$) > 20 then print "Abbrev. to less
     than 20 characters" : goto 240
260 if len(d$) = 0 then print "Please enter a d
     escription." : goto 240
270 print : input "Number of items "; n$
280 n = val(n$) : if n = 0 and n$ <> "0" then p
     rint "Enter a number" : goto270
290 if n <> int(n) then print  "Enter whole num
     bers only." : goto 270
300 if n <= 0 then print "There must be some un
     its; enter number" : goto 270
310 print : input "What is the dollar value of
     each "; v$
320 v = val(v$) : if v = 0 and v$ <> "0" then p
     rint "Enter a number" : goto310
330 if v <= 0 then 400
340 _________________________ : rem print to fil
     e
350 :
360 print "Writing: " d$; n; v
370 print
380 goto 240
390 :
400 print : print "Did you really mean a zero v
     alue--"
410 input "(yes or no) "; r$
420 let r$ = left$(r$,1)
430 if r$ = "n" or r$ = "N" then print "Reenter
      correct value." : goto 310
440 if r$ = "y" or r$ = "Y" then 340
450 print "Please enter 'Y' for YES or 'N' for
     NO." : print : goto 400
460 :
470 rem   file close routine
480 _________________________ : rem close file
```

Practice using cassette file statements by filling in the missing lines in the Property Inventory program:

(a) Open the file

```
210 open _____
```

(b) Given three variables—d$ for the description, n for number of items, and v for value—write a statement to print them to the file opened in (a).

```
340 print# _____
_____
```

(c) Write the statement in line 480 to close the file opened in (a):

```
480 _____
```

(a) `210 open 1, 1, 2, "property"`

(Also, open 1, 1, 1, "property", or open 1, 1, 1.) The file number can be any number 1–127 of your choice. You could also capitalize "Property" if you choose, and do it consistently!

(b) `340 print#1, d$ : print#1, n : print#1, v`

or

```
340 print#1, d$
342 print#1, n
344 print#1, v
```

The file number must match that in the OPEN statement.

(c) `480 close 1`

Again, the file number must match that in the OPEN statement.

Here's the complete program:

```
100 rem   property inventory cassette file prog
    ram
101 if peek(50003) <> 0 then poke 59468,14 : re
    m lower case for pets
102 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
110 :
120 rem   variables used
130 rem     d$ = description (20)
140 rem     n = number of items
150 rem     v = dollar value
160 rem     cr$ = carriage return
170 rem   files used
180 rem     property = sequential file
190 :
200 rem   initialize
210 open 1, 1, 2, "property"
220 :
230 rem   data entry routines
240 print : input "Item description "; d$
250 if len(d$) > 20 then print "Abbrev. to less
     than 20 characters" : goto 240
260 if len(d$) = 0 then print "Please enter a d
    escription." : goto 240
270 print : input "Number of items "; n$
280 n = val(n$) : if n = 0 and n$ <> "0" then p
    rint "Enter a number" : goto270
290 if n <> int(n) then print  "Enter whole num
    bers only." : goto 270
300 if n <= 0 then print "There must be some un
    its; enter number" : goto 270
310 print : input "What is the dollar value of
    each "; v$
320 v = val(v$) : if v = 0 and v$ <> "0" then p
    rint "Enter a number" : goto310
330 if v <= 0 then 400
340 print#1, d$ : print#1, n : print#1, v
350 :
360 print "Writing: " d$; n; v
370 print
380 goto 240
390 :
400 print : print "Did you really mean a zero v
    alue--"
410 input "(yes or no) "; r$
420 let r$ = left$(r$,1)
430 if r$ = "n" or r$ = "N" then print "Reenter
     correct value." : goto 310
440 if r$ = "y" or r$ = "Y" then 340
450 print "Please enter 'Y' for YES or 'N' for
    NO." : print : goto 400
460 :
470 rem   file close routine
480 close 1
```

(a) The above program has one small but important "bug." Find and describe the error. _____

_____

_____

(a) The program never executes the file closing routine at line 400; the CLOSE statement is needed to assure flushing the last data items from the buffer to the file on the diskette.

The problem of how to indicate to the program when to close the file is part of preplanning. The program should include a way for the user to indicate to the computer that the user is done with the program for now, or that all data have been entered. Either of the two procedures shown below could be included in the previous program for this purpose. The choice is yours, but we prefer the second.

```
238 print "Type 'STOP' if finished."
245 if d$ = "STOP" or d$ = "stop" then 470
```

or

```
370 print
372 print "Is there any more data ?"
374 print "Press 'Y' for YES or 'N' for NO."
376 get z$ : if z$ = "Y" or z$ = "y" then 240 : rem
      next entry
378 if z$ = "N" or z$ = "n" then 470
380 goto 376
390 :
```

Two reminders: First, when you write a file to tape, it records over any previous information on the tape, permanently destroying it! Either use a new tape, or be certain you no longer need whatever was on the tape. Second, when you write file programs (or any program), prepare some written documentation for yourself and other users. Even you may have trouble seeing how the program works six months down the line. At the least, some description of the file format is needed. A good habit is to include such information in REMark statements in the program itself (as we did in the Property File program), and on the cassette tape label.

(a) In a program, why do you need to check that all data to be included in the data file have been entered?

_____

_____

(a) So that a CLOSE statement can be executed.

READING DATA FROM A FILE

Now that you can write to a file, let's learn how to read data from an existing file. To do this, you must know what data were placed in the file, and how they were arranged. After that, reading from the file is straightforward.

To read from a file, you first OPEN the file for READ, then use the INPUT# statement:

```
350 OPEN 2, 1, 0, "TEMPFILE"
360 INPUT#2, A, B$, C
```

or

```
350 OPEN 2, "TEMPFILE"
360 INPUT#2, A, B$, C
```

Notice the use of commas in line 360: one after the file number, and others to separate the variables, just like an INPUT statement. Line 360 will read three data items from the cassette tape file and assign them to variables A, B$, and C.

You must have the correct type of variable (numeric or string) in the INPUT# statement to match the data that are being read from the data file. If you try to read string data into a numeric variable, your program will terminate with "?FILE DATA ERROR."

To avoid such problems, be sure you know how the data were initially placed into the file, whether string or numeric, and in what order. To be safe, you may wish to read all the data as strings, and do the conversion to numeric data in the program, where you can control the display of messages.

Here's an example of this situation:

```
100 OPEN 2, 1, 1, "TEMPFILE"
                  .
                  .
                  .
170 PRINT#2, A ;CHR$(13); B$ ;CHR$(13); C
```

The data stored by line 170 can *only* be read by the following sequence of variables:

```
400 OPEN 2, 1, 0, "TEMPFILE"
410 INPUT#2, A, B$, C
```

A statement like INPUT#2, A$, B, C will result in "?FILE DATA ERROR."

Of course, as long as you have the correct sequence of variable *types* (in this example, numeric, string, numeric) you could read the data with any combination of INPUT# statements:

```
410 INPUT#2, A
420 INPUT#2, B$, C
```

or

```
410 INPUT#2, A
420 INPUT#2, B$
430 INPUT#2, C
```

The INPUT# statement with multiple variables works just like the corresponding INPUT statement: successive values must be separated by commas or carriage returns. Separating file data with commas is fraught with problems, so you will usually only deal with carriage returns as separators.

Remember our insistence on printing each value followed by a carriage return? It is just like our earlier recommendation that each INPUT statement be used to get one, and just one value or string! If you place each variable in a separate PRINT# statement, or at least separate them with carriage returns, you won't have to worry about data formats and lost or unreadable data.

We'll repeat that: Use a separate PRINT# statement to write each variable; then INPUT# can take whatever format is convenient.

The program below is the companion to the Inventory file creation program to read the PROPERTY file. Again, lines 200, 270, and 320 are for you to fill in following the program.

```
100 rem   read data from property file
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem     n$ = description (20)
140 rem     q = quantity
150 rem     d = dollar value
160 rem     r$ = user response
170 rem   files used
180 rem     property, seq.;  dataset format: n$
     , q, d
190 :
199 :
200 _____________________________ : rem ope
     n file
210 :
220 rem   print headings
230 print"[clr]" : rem clear screen
240 print "Description" tab(22) "Quan." tab(30)
      "Value Ea."
250 :
260 rem   file read/report routine
270 _____________________________ : rem rea
     d data
280 print n$ tab(22) q tab(30) d
290 goto 270
300 :
310 rem   close file
320 _____________________________ : rem clo
     se file
```

Write the missing statements:

(a) Write line 200 to OPEN the "Property" file for reading:

200 _____

(b) Write the INPUT# statement to read the data (refer back to the Inventory writing program if you don't remember the format)—or look in the introductory module (see why we insist on it?).

270 _____

(c) Close the file: 320 _____

_____

(a) `200 open 1, 1, 0, "property"`

or

`200 open 1, "property"`

(The file number can be any number 1–127 of your choosing.)

(b) `270 input#1, d$, n, v`

The file number must match that in the OPEN statement.

(c) `320 close 1`

Here's the complete Inventory file reading program, followed by a sample run.

```
100 rem   read data from property file
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
110 :
120 rem   variables used
130 rem     n$ = description (20)
140 rem     q = quantity
150 rem     d = dollar value
160 rem     r$ = user response
170 rem   files used
180 rem     property, seq.;  dataset format: n$
    , q, d
190 :
199 :
200 open 3, 1, 0, "property"
210 :
220 rem   print headings
230 print"[clr]" : rem clear screen
240 print "Description" tab(22) "Quan." tab(30)
     "Value Ea."
250 :
260 rem   file read/report routine
270 input#3, n$, q, d
280 print n$ tab(22) q tab(30) d
290 goto 270
300 :
310 rem   close file
320 close 3
```

```
run

Description          Quan.    Value Ea.
Television           2          500
Cassette Recorder    2          125
Video Recorder       1          750
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
break in 280
ready.
```

This RUN was terminated by a hasty press of the RUN/STOP key, an obvious error condition, leaving an open file. What if you wanted to do more with the data, and wanted the program to continue beyond reading and printing the data? A technique exists that allows the program to read to the end of the file and continue properly.

DETECTING THE END OF A FILE

When your program uses PRINT# to add data to a file, each PRINT# statement moves an internal pointer ahead. The computer uses this pointer to keep track of where it is within the buffer. When all data have been entered, the end of file marker is located just past the last data item. The end of file marker is automatically put in place by the computer when the file is closed.

When you INPUT# data from the file, the file pointer is always looking at the next data item available in the file (or in the buffer, to be more exact). How can we detect the end of a file? Commodore has an answer in the STATUS variable.

STATUS

ST is the STATUS variable. Its value is set after each input or output operation to indicate the status of that operation. For tape, a zero indicates a proper load (for programs), values between 4 and 60 indicate read errors, 64 indicates end of file, and –128 indicates end of tape.

ST is most commonly used to check for the end of a file. This example assumes file 1 has been opened for reading:

```
250 INPUT#1, A$
260 PRINT A$
270 IF ST = 0 THEN 250
280 IF ST < 64 THEN PRINT "READ ERROR"
290 CLOSE 1
```

This could also be programmed as

```
250 INPUT#1, A$
260 PRINT A$
270 IF ST = 64 THEN 290
280 GOTO 250
290 CLOSE 1
```

There is a problem with this second approach: tape errors will not stop the program from trying to read, and the program may go on forever.

We'll use the first approach to modify the previous program so it does not terminate with an END OF FILE error. Modify line 290 as shown here:

```
290 IF ST = 0 THEN 270
```

USING ST WITH A PRINTER

Because the STatus is set by each input/output operation, printing data to a printer as it is read from a file requires special handling. Otherwise, the ST will always be zero to reflect the status of the printer operation, rather than the file-reading operation. Assuming file 4 has been opened to the printer with an OPEN 4, 4 :

```
250 INPUT#1, A$
260 PRINT#4, A$
270 IF ST = 0 THEN 290
280 IF ST < 64 THEN PRINT "READ ERROR"
290 CLOSE 1 : PRINT#4 : CLOSE 4
```

ST will always be zero after the PRINT# (unless your printer is disconnected or turned off), so the program will go on forever. Instead, the program must store the value of ST immediately after the INPUT#, but not check it until after the PRINT#:

```
250 INPUT#1, A$
255 SV = ST
260 PRINT#4, A$
270 IF SV = 0 THEN 290
280 IF SV < 64 THEN PRINT "READ ERROR"
290 CLOSE 1 : PRINT#4 : CLOSE 4
```

Reference manuals for various computer systems show considerable variation in explanations of how the pointer and end of file marker work. However, the ideas expressed here are the same for any system; the keywords and programming statements will differ.

(a) What does ST stand for? _____

(b) What does the value of 64 for ST indicate?

_____

(c) Why is it necessary to store ST in another variable when reading from tape then printing to a printer?

_____

_____

(a) STATUS

(b) End of file.

(c) Any input or output operation resets the value of ST, so it will no longer properly reflect the file status.

FILE-READING UTILITY PROGRAMS

When you start writing files, it is handy to have a utility program to read back your files for you, so you can see how the data look. This is particularly useful when you start getting *file data errors*. Here are two short programs which will read and display files. Try these on your file data programs. You'll find the GET version particularly useful if you want to experiment with the way data are actually written to the disk by PRINT#, or if you start getting "?STRING TOO LONG" errors.

The first program uses INPUT# to read the data. The disadvantage of this program is the usual disadvantage of INPUT#; if you have written data to the file with something other than carriage returns separating the data items, this program will not read all of your data. However, it is fast and simple; when disaster strikes, it can be typed in at the keyboard if necessary.

```
10 REM TAPE FILE READER
100 OPEN 1 : REM DEFAULT READ--FIRST FILE ON TA
    PE
110 INPUT#1, A$
120 PRINT A$
130 IF ST = 0 THEN 110
140 CLOSE 1 : END
```

This version uses GET# to read a file. The advantage of GET# is that it is not subject to limitations of punctuation or character number. A file containing commas, colons, or semicolons may be read with this routine, allowing you to examine how the data were written and thus to fix the problem. Line 130 checks for a carriage return at the end of the data item.

```
10 REM TAPE READ WITH GET
100 OPEN 1 : REM DEFAULT READ
110 LET S$ = ""
120 GET#1, Z$
130 IF Z$ = CHR$(13) THEN 160
140 LET S$ = S$ + Z$
150 GOTO 120
160 PRINT S$
170 IF ST = 0 THEN 110
180 CLOSE 1
```

This particular version concatenates each character read until it encounters a carriage return, then it prints the concatenated string and starts building the next string. This allows you to see commas, spaces, and other punctuation which would not show up if read by an INPUT# statement. However, it will also die from "?STRING TOO LONG" problems.

Here's another version which simply prints each character as it is read without concatenating it into a string. (Of course, you could avoid the "?STRING TOO LONG" problem by revising the previous program to print S$ whenever it had more than 255 characters, then reset it to an empty string). While the preceding program could actually be used to read file data in a program, this one will only display data. It prints the PETASCII value of each character, so you can even see invisible characters. Note the "+ chr$(0)" in the ASC function—it's necessary in case the GET picks up a null character (""), which will return an error message. Also added is "Press RETURN to continue," to let you step through the file. STatus has been saved in the variable SV, because the GET in line 180 is seen as an input/output operation, and resets the status to 0. (Read that section on STATUS very carefully—it's a tricky beast!)

```
100 rem read any file with get
120 open 1 : rem default read
140 get#1, c$ : let sv = st
150 print asc(c$ + chr$(0));
160 if c$ <> chr$(13) and sv = 0 then 140
170 print
180 print "[down] Press [rvs]RETURN[off] to conti
    nue."
190 get z$ : if z$ <> chr$(13) then 190
200 if sv = 0 then 130
210 close 1
220 print "File read and closed."
```

MULTIPLE FILE OPERATIONS IN ONE PROGRAM

We have used the word "copy" to describe how the INPUT# statement works when data are transferred from the disk data file into the computer's memory. "Copy" implies that the data in the file do not change when they are input into the buffer.

You can think of the process described above as like listening to a cassette tape recording of music: playing the tape doesn't erase it, and it can be played back again and again. The data in a file are similarly unaffected and unchanged by "playing back" the data into the computer, and the data remain in the file for another use. In BASIC, the only way to change data in a sequential file is with a PRINT# statement.

You can WRITE data to a file, then READ them back from the same file in the same program. But it is crucial that you CLOSE the file after writing (recording information into the file), *and that you rewind the tape*, before you can reopen the file to READ the data (copy it into the computer's memory). You must OPEN 1, 1, 1 (or OPEN 1, 1, 2) for writing, CLOSE the file, *rewind the tape*, then OPEN 1, 1, 0 for READing.

The program you will see next illustrates the procedure to open and close files at appropriate times. Quality assurance (QA) data from a manufacturing process are entered into a file. The data are a long list of numbers that represent, let's say, the clarity of glass from various meltings of recycled bottles, using a scale of 1 to 6.

After the data are all entered and the file is closed, the program will read the QA values from the file, accumulate the number of occurrences of each category of clarity (1 to 6) in an array, and then display a summary of the data. The program is self-documented by REMark statements. Use QCONTROL as the name for the file, as in the sample RUN that follows:

```
run
File name?qcontrol
Enter numbers 1-6 only. Enter 99 to stop
qa number:? 1
qa number:? 1
qa number:? 4
qa number:? 6
qa number:? 2
qa number:? 3
qa number:? 5
qa number:? 4
qa number:? 2
qa number:? 1
qa number:? 2
qa number:? 3
qa number:? 42
Use numbers 1-6 only!

qa number:? 4
qa number:? 2
qa number:? 5
qa number:? 5
qa number:? 6
qa number:? 99

Results of quality control data

QA Number     Quantity
1              3
2              4
3              2
4              3
5              3
6              2
ready.
```

Here's the program used to produce the RUN shown above.

```
140 rem file input/output demo
141 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
142 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
150 :
160 rem  user enters quality control results fo
    r file
170 rem   program prepares summary report from
    file
180 :
190 rem      variables used
200 rem        f$ = file name
210 rem        n  = qa measure
220 rem        v  = qa measure
230 rem        c() = counting array
240 rem      seq file = qcontrol (user entered):
     dataset format = n
250 :
260 rem      initialize array (not necessary in
    commodore basic)
270 for x = 1 to 6 : let c(x) = 0 : next x : re
    m basic sets all vars. to 0
280 :
290 rem      initialize file
300 input "File name "; f$
301 print "Insert cassette and rewind."
302 print "Press 'STOP' key on recorder when do
    ne."
303 print "Press 'RETURN' to continue."
304 get z$ : if z$ <> chr$(13) then 304
310 open 1, 1, 2, f$
320 :
330 rem      data entry routine
340 print "Enter integers 1 - 6 only.  Enter '9
    9' to stop."
350 :
360 input "QA number "; n
370 if n = 99 then 430
380 if n < 1 or n > 6 then print "Reenter using
     1 to 6 only." : goto 360
390 print#1, n
400 goto 360
410 :
420 rem    close file
430 close 1
440 :
450 rem    open file to read
452 print : print "Insert cassette with file "
    f$ " and rewind."
454 print "Press 'STOP' key on recorder when do
    ne."
456 print "Press 'RETURN' to continue."
457 get z$ : if z$ <> chr$(13) then 457
```

```
460 open 1, 1, 0, f$
470 :
480 rem    read file and accumulate data in arr
    ay
490 print "Reading data..."
500 input#1, v
510 let c(v) = c(v) + 1
520 if st = 0 then 500
530 :
540 rem    print report from array
550 print "[clr]" : rem clear screen
560 print "Results of quality control data"
570 print
580 print "QA number","Quantity"
590 for v = 1 to 6
600 print v, c(v)
610 next v
620 :
630 rem    close file
640 close 1
650 print "File closed, job complete."
```

Refer to the previous program to answer the following questions:

(a) In which line does the computer obtain the name of the data file?

_____

(b) Which statement checks the parameters for the quality control numbers?

_____

(c) How does the computer know that all data have been entered? _____

_____

(d) Why are two CLOSE statements used in the same program?

_____

_____

(e) What does line 520 do? _____

_____

(f) In line 510, how many different values can V have?

_____

(g) What would happen if line 500 read "500 INPUT#2, V" ?

_____

_____

(h) What would happen if the user didn't rewind the tape as instructed in lines 452–456? _____

_____

(a) Line 300.
(b) Line 380.
(c) User enters 99 as input value.
(d) The data file must be closed after READing and after WRITing.
(e) Checks for end of file. If ST is 0, there is more data, so the program returns to line 500.
(f) Six (1 to 6).
(g) No data would be read, and the program would terminate with a "?FILE NOT OPEN ERROR" because the file numbers don't match.
(h) The tape is positioned beyond the data written; the program would try to find a file named "qcontrol," searching to the end of the tape. No data would be read or displayed.

One unique feature of file programs is that sometimes nothing appears to be happening when the program is RUN. There may be no printed report or anything happening on the screen other than RUN and READY. To the novice, this seeming lack of activity may be alarming. Of course, you will see and hear the cassette motor go on and off as files are opened and closed, and as data are written.

To minimize the feeling that "nothing is happening," we recommend that you print appropriate messages on the screen as the program runs, to inform the operator about what is happening. (This may be an exercise in futility for the C-64, which clears the screen when reading or writing files—or maybe you will want to print a message warning the user that the screen will clear.) For example:

```
210  PRINT "WRITING DATA TO FILE."
220  PRINT "WRITING " A$ : PRINT#1, A$
  .
  .
  .
780  PRINT "FINISHED WRITING, FILE CLOSED."
```

As the last activity in Chapter 3 before the Self-Test, you (should have) constructed a program for *entering* names and addresses, and *displaying* the entered data for corrections. Now you can add the statements to *create a file* of names and addresses.

If you SAVEd the program, LOAD it and LIST it now for reference, or turn back to the end of Chapter 3 and enter it. Then complete line 250 to create the ADDRESS file.

```
(a)  200 REM     SEQUENTIAL FILE NAME: ADDRESS
     210 REM     DATASET FORMAT: D$
     220 :
     230 REM     INITIALIZE
     240 :
     250 ____________________________________________________________
     260 :
     300 REM     MAIN DATA ENTRY MODULE AND ERROR TESTS
```

```
(a)  250 OPEN 1, 1, 2, "ADDRESS"
```

Only two more file-handling statements are needed for a functioning program to create a name and address file. Complete line 540 to transfer the concatenated dataset to the file, and take a wild guess at what is needed at line 590.

```
(a)     490 if r = 6 then 520
        500 on r gosub 620, 670, 720, 770, 810 : goto 3
            70
        510 :
        520 let d$ = n$ + a$ + c$ + s$ + z$
        530 :
        540 ____________________ : rem write data
        550 :
        560 print : print "Do you have another name to
            enter (Press 'Y' or 'N') ?"
        570 get r$ : if  r$ = "" then 570
        580 if r$ = "y" or r$ = "Y" then 310
        585 if r$ <> "n" and r$ <> "N" then 570
        590 ____________________ : rem close file
        600 print "File closed."
```

```
(a)  540 print#1, d$
     590 close 1
```

When you run this program a second time, you will need to insert a new tape in your recorder, or you will record over your old file. If the old data were important, we hope you punched out the tabs on the back of your cassette, so you can't record over it. See how critical proper labeling and handling of cassette tapes is?

The complete program follows:

```
100 rem   address file creating program
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
110 :
120 rem  variables used
130 rem  n$ = name (20 char. max.)
140 rem  a$ = address (20 char. max.)
150 rem  c$ = city (10 char. max.)
160 rem  s$ = state (2 char. )
170 rem  z$ = zip code (5 char. )
180 rem  d$ = entire dataset concatenated (67 c
    har.)
190 :
200 rem  seq. file name = address
210 rem  dataset format = d$
220 :
230 rem  initialize
240 :
242 print "Insert cassette for 'ADDRESS' file n
    ow."
243 print "REWIND, then press 'STOP' on recorde
    r."
244 print "Press 'RETURN' when completed."
246 get r$ : if r$ <> chr$(13) then 246
250 open 1, 1, 2, "address"
260 :
300 rem  main data entry & error tests
310 print "[clr]" : rem clear screen
320 gosub 620
330 gosub 670
340 gosub 720
350 gosub 770
360 gosub 810
370 rem display data for corrections
380 print "[clr]" : print "You have entered the
     following data:" : print
390 print "- 1 - "; n$
400 print "- 2 - "; a$
410 print "- 3 - "; c$
420 print "- 4 - "; s$
430 print "- 5 - "; z$
440 print "- 6 - no changes"
450 print : print "Check for errors. Enter numb
    er of item"
460 print "that needs correcting. If no changes
     are needed, enter number 6."
470 print : input "Enter your selection (1 to 6
    ) "; r$
```

```
475 r = val(r$)
480 if r < 1 or r > 6 then print "Enter a numbe
    r from 1 to 6." : goto 450
490 if r = 6 then 520
500 on r gosub 620, 670, 720, 770, 810 : goto 3
    70
510 :
520 let d$ = n$ + a$ + c$ + s$ + z$
530 :
540 print : print "Writing " d$ : print#1, d$
550 :
560 print : print "Do you have another name to
    enter (Press 'Y' or 'N') ?"
570 get r$ : if  r$ = "" then 570
580 if r$ = "y" or r$ = "Y" then 310
585 if r$ <> "n" and r$ <> "N" then 570
590 close 1
600 print "File closed."
610 end
620 print : input "Name "; n$
630 if n$ = "" then print "No entry made. Pleas
    e enter the name." : goto 620
640 if len(n$) > 20 then print "Abbrev. to 20 c
    haracters or less." : goto 620
650 if len(n$) < 20 then let n$ = left$(n$ + "
                          ", 20 )
660 return
670 print : input "Street address "; a$ : let a
     = len(a$)
680 if a$ = "" then print "No entry--please ent
    er the street." : goto 670
690 if a > 20 then print "Address too long--20
    letters or less." : goto 670
700 if len(a$) < 20 then let a$ = left$(a$ + "
                          ", 20 )
710 return
720 print : input "City name "; c$
730 if c$ = "" then print "No entry--please ent
    er the city." : goto 720
740 if len(c$) > 20 then print "City too long--
    20 letters or less" : goto 720
750 if len(c$) < 20 then let c$ = left$(c$ + "
                          ", 20 )
760 return
770 print : input "Two-letter state abbreviatio
    n "; s$
780 if s$ = "" then print "No entry--please ent
    er the state." : goto 770
790 if len(s$) <> 2 then print "Please use stan
    dard 2 letter code" : goto 770
800 return
810 print : input "Zip code (5 digits) "; z$
820 if z$ = "" then print "No entry--please ent
    er the zip." : goto 810
830 if len(z$) <> 5 then print "Please enter ex
    actly 5 digits." : goto 810
840 return
```

DISPLAYING ONE DATASET AT A TIME FROM A FILE

Whenever we work extensively with files, we write a small utility program that lets us read through the file, one item at a time, to verify that everything is as it should be. A properly written data file editing program also lets you make changes in the file data as it reads through the file (such a program is coming up!). Our example will use the previous application, the ADDRESS file.

The program shown below allows you to look at each dataset, one item at a time, with the prompt, "PRESS RETURN FOR NEXT ADDRESS." The "PRESS RETURN TO CONTINUE" technique is very popular for screen-oriented systems (as opposed to printer systems). It allows the computer user to review the data displayed for as long as necessary, and then move on to the next dataset. The most frequently used keys are RETURN and SPACE. We prefer RETURN, as users are accustomed to pressing it when done; beware the "PRESS ANY KEY TO CONTINUE"—users have been known to spend hours trying to decide which key to press—only to press the RUN/STOP key and seriously damage the program!

The program clears the screen to reduce "screen clutter" (previously displayed information) before each dataset is displayed, using the CLR (SHIFT-CLR/HOME) within quotes in a print statement. See line 330. Notice how the GET statement is used with the "PRESS RETURN TO CONTINUE" technique. As mentioned earlier, we prefer using GET for single-keystroke entries, or when waiting for a keypress, because on the PET, pressing RETURN in response to INPUT ends the program. (See Appendix 4 for some solutions.)

Help us to complete this companion program to the ADDRESS file-creating program. This "file read and display" program should display each dataset in ADDRESS, one at a time. Have each name and address displayed in mailing label format, using string functions to "de-concatenate" the data items for display (lines 280, 290, and 300). Also complete the data file-handling statements in lines 250 and 260.

(a)

```
100 rem  read and display address file one data
     set at a time
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem  variables used
130 rem  n$ = name (20 char. max.)
140 rem  a$ = address (20 char. max.)
150 rem  c$ = city (10 char. max.)
160 rem  s$ = state (2 char. )
170 rem  z$ = zip code (5 char. )
180 rem  d$ = entire dataset concatenated (57 c
     har.)
190 :
200 rem  seq. file name = address
210 rem  dataset format = d$
```

```
220 :
230 rem  initialize
240 :
242 print "Insert 'ADDRESS' cassette and REWIND
    ."
244 print "When rewound, press 'STOP' key on re
    corder."
246 print "Press RETURN key after stopping tape
    ."
248 get z$ : if z$ <> chr$(13) then 248
250 _____________________________ : rem o
    pen
260 _____________________________ : rem i
    nput
270 :
275 print "[clr][down][down][down][down][down]"
     : rem clear scrn & 5 down
280 _____________________________ : rem p
    rint name
290 _____________________________ : rem p
    rint address
300 _____________________________ : rem p
    rint city/state/zip
310 if st <> 0 then 350
320 print : print "Press 'RETURN' for next addr
    ess."
325 get z$ : if z$ <> chr$ (13) then 325 : rem
    wait for return
330 print "[clr]" : goto 260
340 :
350 close 1 : print " File read."
360 print "Rewind and remove cassette."
```

(a)
```
230 rem  initialize
240 :
242 print "Insert 'ADDRESS' cassette and REWIND
    ."
244 print "When rewound, press 'STOP' key on re
    corder."
246 print "Press RETURN key after stopping tape
    ."
248 get z$ : if z$ <> chr$(13) then 248
250 open 1, 1, 0, "address"
260 input#1, d$
270 :
280 print left$(d$,20)
290 print mid$(d$,21,20)
300 print mid$(d$,41,10) + " " + mid$(d$,51,2)
    + " " + right$(d$,5)
310 if st <> 0 then close 1 : end
320 print : print "Press 'RETURN' for next addr
    ess."
325 get z$ : if z$ <> chr$ (13) then 325 : rem
    wait for return
330 print "[clr]" : goto 260 : rem clear screen
```

Refer to the program above to answer these questions.

(a) What is assigned to z$ in line 325? _____

_____

(b) Since z$ acts as a dummy variable in the program above, what is the purpose of line 325?

_____

_____

(c) How often was the screen "refreshed" in the program above?

_____

(a) Whatever key is pressed.
(b) Keeps the data items on the screen until the user presses the RETURN key to continue. The program waits at the GET statement until the RETURN key is pressed.
(c) Before (or after) each complete dataset of three items was displayed.

COPYING AND EDITING CASSETTE DATA FILES

Data file utility programs can be written for use with a single cassette recorder, but they are cumbersome. In order to copy or edit data in a file with a cassette recorder, you must first read the entire file into the computer's memory, storing it in an array. You then make the necessary changes, insert a new tape, and write the entire file at once to the new tape. This limits the size of your file—not so much by the size of your computer memory, but by the size of a file that can be read reliably from a tape. Up to 16K is no problem, but files longer than 16K are susceptible to reading errors in proportion to their length. If you will be using large files, now is the time to get a disk drive!

Now let's develop a model utility program to copy an existing cassette data file onto a new cassette tape. But first, write a program to enter the data for the original cassette data file, the one that will later be copied. Here are the specifications for that program and data file:

1. The data items are numeric values in the range from 1 to 100, representing statistical information (such as age, driving speed, or the like). Include a data entry check for these parameters.
2. The values are copied onto the file with a PRINT# statement using one variable (one value at a time).
3. You may enter as few values as you wish. We suggest a minimum of a couple of dozen values and as many as 350 to 400 values. Using a large number of values in the file gives you a feel for the amount of time it takes to manipulate cassette data files.
4. Place –999 as the ending data flag at the end of the file data.

(a) Now write the program and create the data file on your computer.

```
100 REM  STAT1 DATA ENTRY
110 REM  FILES: STAT1 CASSETTE TAPE
120 :
130 REM  OPEN FILE
140 PRINT "INSERT AND REWIND CASSETTE."
150 PRINT "BE SURE TO PRESS 'STOP' KEY ON RECOR
    DER WHEN DONE."
160 PRINT "WHEN DONE, PRESS 'RETURN' KEY."
170 GET Z$ : IF Z$ <> CHR$(13) THEN 170
180 OPEN 1, 1, 2, "STAT1"
190 :
200 REM DATA ENTRY WITH TESTS
210 PRINT : INPUT "VALUE (OR -999 TO END) "; V
220 IF V = -999 THEN 299 : REM STOP
230 IF V < 1 OR V > 100 THEN PRINT "DATA OUT OF
      RANGE: REENTER." : GOTO 210
240 PRINT#1, V
250 GOTO 210
299 CLOSE 1
300 PRINT "FILE CLOSED--REWIND TAPE." : END
```

Now write a simple cassette file copy routine. The key to the technique involved is to think of the array(s) as the temporary file. You have a cassette file filled with numeric values originally recorded onto the file with a program that had a PRINT# statement like this:

```
PRINT#1, V
```

Remember, you need to know that information in order to read the file correctly. Assume that the values in the file represent statistical data you want to send to a friend in another state, so you need a cassette copy. The file has between 350 and 400 values. You need to know how many data items are in the file so you can correctly dimension the array. It is possible to write your program so all data are stored in an array until data entry is terminated, then to count the number of data and write this as the first datum in the file. (We'll show you this method later.) However, simply DIMensioning an appropriately large array works fine. Here is the introductory module:

```
100 REM   NUMERIC FILE COPY
110 :
120 REM   VARIABLES USED
130 REM     V = VALUE FROM FILE
140 REM     K = COUNTER
150 REM     T = ARRAY
160 REM     X = ARRAY DIMENSION (VARIABLE)
170 :
180 REM   FILES USED
190 REM     STAT1 = CASSETTE TAPE FILE
```

```
200 REM      STAT1COPY = OUTPUT FILE COPY
210 REM      END OF DATA FLAG IS -999
220 :
230 REM    INITIALIZE
240 PRINT "[CLR]" : REM CLEAR SCREEN
250 PRINT "INSERT AND REWIND CASSETTE."
260 PRINT "BE SURE TO PRESS 'STOP' KEY ON RECOR
     DER WHEN DONE."
270 PRINT "WHEN DONE, PRESS 'RETURN' KEY."
280 GET Z$ : IF Z$ <> CHR$(13) THEN 280
290 :
300 PRINT "ABOUT HOW MANY ITEMS IN THIS FILE "
     : INPUT X
310 DIM T(X)
320 LET K = 1
330 :
```

Look at line 300. At the moment, you can't know exactly how many data items are in the file, so you ask "ABOUT HOW MANY ?" The program uses this value to dimension the array. If you overstate this value by too much, you could run out of computer memory, so be realistic. On the other hand, if you underestimate, you will run out of storage space and lose some of your data. Line 320 initializes a variable, K, used to count the data items as they are read from the cassette data file into the array. Now, fill in the blanks in the next program segment:

(a)

```
320 LET K = 1
330 :
340 REM OPEN FILE AND READ INTO ARRAY
350 _____ : REM OPE
     N FILE
360 _____ : REM REA
     D DATA
370 _____ : REM INC
     REMENT COUNTER
380 _____ : REM TES
     T STATUS
390 _____ : REM CLO
     SE FILE
400 PRINT "FILE CLOSED.  REWIND TAPE AND REMOVE
     ."
410 :
```

(a)
```
350 OPEN 1, 1, 0, "STAT1"
360 INPUT#1, T(K)
370 LET K = K + 1
380 IF ST = 0 THEN 360
390 CLOSE 1
400 PRINT "FILE CLOSED.  REWIND TAPE AND REMOVE
    ."
410 :
```

In line 360 you may have assigned the data item to a temporary variable (for example, INPUT#1, T), then placed it in the array after incrementing the array counter; however, there is no problem in assigning it directly to the array element, as done here.

(a) If the statement at line 390 closes the file, what routines will start at line 420 ? _____

_____

_____

(a) Instructions to change or rewind the tape cassette, followed by the array printing routine.

Now, complete the crucial statement in the next program segment (line 500);

(a)
```
410 :
420 REM   INITIALIZE FILE COPY
430 PRINT "[CLR]" : REM CLEAR SCREEN
440 PRINT  "PLACE TAPE FOR COPY IN RECORDER, RE
    WIND, THEN PRESS 'STOP' KEY."
450 PRINT "BE SURE TAPE IS READY BEFORE PRESSIN
    G  'RETURN'"
460 GET Z$ : IF Z$ <> CHR$(13) THEN 460
470 :
480 OPEN 1, 1, 2, "STAT1COPY"
490 FOR Y = 1 TO K
500 ___________________________________ : REM PRI
    NT VALUE TO CASSETTE FILE
510 NEXT Y
520 :
530 REM CLOSE FILE
540 CLOSE 1
550 PRINT "FILE CLOSED, COPY COMPLETE."
560 PRINT "REWIND AND REMOVE CASSETTE."
570 :
```

(b) Why is it necessary to use a counter (K) to count data?

(a) 500 PRINT#1, T(Y)
(b) It keeps track of how many items in the array are actual data; originally we just guessed at the size: K was incremented in the reading part of the program as each data item was read, giving us an accurate count. Now we use it to print just those data items read into the array.

For this application, the program concludes by CLOSEing the file, thus writing an end of tape or end of file marker, depending on whether the secondary address was 1 or 2 when the file was OPENed.

```
510 :
520 REM  CLOSE FILE
530 CLOSE 1
540 PRINT "JOB COMPLETED."
550 :
```

Use part of the copy program to prepare an add-to-file program. Can you see what changes have to be made?

```
100 REM   NUMERIC FILE COPY
110 :
120 REM   VARIABLES USED
130 REM     V = VALUE FROM FILE
140 REM     K = COUNTER
150 REM     T = ARRAY
160 REM     X = ARRAY DIMENSION (VARIABLE)
170 :
180 REM   FILES USED
190 REM     STAT1 = CASSETTE TAPE FILE
200 REM     STAT1COPY = OUTPUT FILE COPY
210 REM     END OF DATA FLAG IS -999
220 :
230 REM   INITIALIZE
240 PRINT "[CLR]" : REM CLEAR SCREEN
250 PRINT "INSERT AND REWIND CASSETTE."
260 PRINT "BE SURE TO PRESS 'STOP' KEY ON RECOR
    DER WHEN DONE."
270 PRINT "WHEN DONE, PRESS 'RETURN' KEY."
280 GET Z$ : IF Z$ <> CHR$(13) THEN 280
290 :
300 PRINT "ABOUT HOW MANY ITEMS IN THIS FILE "
    : INPUT X
310 DIM T(X)
320 LET K = 1
330 :
340 REM OPEN FILE AND READ INTO ARRAY
350 OPEN 1, 1, 0, "STAT1"
360 INPUT#1, T(K)
370 LET K = K + 1
380 IF ST = 0 THEN 360
390 CLOSE 1
400 PRINT "FILE CLOSED.  REWIND TAPE AND REMOVE
    ."
410 :
420 REM   INITIALIZE FILE COPY
430 PRINT "[CLR]" : REM CLEAR SCREEN
```

```
440 PRINT  "PLACE TAPE FOR COPY IN RECORDER, RE
    WIND, THEN PRESS 'STOP' KEY."
450 PRINT "BE SURE TAPE IS READY BEFORE PRESSIN
    G  'RETURN'"
460 GET Z$ : IF Z$ <> CHR$(13) THEN 460
470 :
480 OPEN 1, 1, 2, "STAT1COPY"
490 FOR Y = 1 TO K
500 PRINT#1, T(Y) : PRINT "WRITING " T(Y) : REM
     PRINT VALUE TO CASSETTE FILE
510 NEXT Y
520 :
530 REM CLOSE FILE
540 CLOSE 1
550 PRINT "FILE CLOSED, COPY COMPLETE."
560 :
```

Surprise! You can use all of the copy program up to line 520.

Up to now the program can copy the original file into an array and then onto a new cassette file. You can just reuse the original tape if you're confident that you don't need a back up copy.

(a) Describe what the program to add-to-file should do in the routine beginning at line 560.

(b) Fill in the blanks for the routine you described in (a), completing lines 550, 560, 570, and 590:

```
520 :
530 REM   ENTER NEW DATA ITEMS (-999 TO STOP)
540 REM   PRINT TO FILE
550 _____________________________ : REM ENTER
    DATA
560 _____________________________ : REM TEST F
    OR END OF DATA
570 _____________________________ : REM WRITE
    TO FILE
580 GOTO 550
590 _____________________________ : REM CLOSE
    FILE
600 :
```

(c) How does the program know to stop asking for more new data?

(a) Provide for entry of new data and add it to the new file.

(b)
```
520 :
530 REM   ENTER NEW DATA ITEMS (-999 TO STOP)
540 REM   PRINT TO FILE
550 PRINT : INPUT "DATA ITEM (-999 TO END) "; V

560 IF V = -999 THEN 590 : REM QUIT
570 PRINT#1, V : PRINT "WRITING " V
580 GOTO 550
590 CLOSE 1 : PRINT "FILE CLOSED, REWIND AND RE
     MOVE THE TAPE."
600 :
```

(c) By checking to see whether the user has entered the end of file marker (–999).

We have followed the same procedures you would use for disk files, except an array was used instead of a temporary disk file.

With the use of arrays as temporary files, it is important to know how many data items are stored in a file so you can properly dimension the arrays. Before you estimated, but here is a variation you might want to consider. You could, as a policy, always keep track of the number of data items in a file and place this figure as the first data item in the cassette tape file. Then use this value to dimension the array. Some changes in procedures are required.

The new procedures are:

1. Read first data item (number of datasets or data items) = X.
2. Ask how many new datasets are to be added = Y.
3. Dimension the array(s) with X + Y.
4. Read the file into the array.
5. Add the new data for the file into the array rather than the new file.
6. When a user signal indicates all new data have been entered, then print the new total number of data items into the new file.
7. Copy the array data items into the new file.

As an example of this procedure, consider the grocery list program (Problem 2 for this chapter, or disk version in Chapter 5). Here the data entered are item descriptions and quantity (how much of that grocery item to buy). The first data item in the data file tells how many datasets are already contained in the file. The program follows the seven steps outlined above.

```
100 REM  ADD TO CASSETTE GROCERY FILE
110 :
120 REM  VARIABLES USED
130 REM    D$ = ITEM DESCRIPTION
140 REM    N  = NUMBER NEEDED
150 REM    G$,N = ARRAYS
160 REM    R$ = RESPONSE STRING
170 REM    K  = DATASET COUNT
180 REM    A  = # OF ADDED DATASETS
190 :
200 REM  FILES USED
210 REM    CASSETTE FILE = GROCERY
220 REM    DATASET FORMAT: D$, N
230 :
240 REM  FILE INITIALIZATION
250 PRINT "INSERT 'GROCERY' CASSETTE, REWIND, A
    ND PRESS RECORDER'S 'STOP' KEY."
260 PRINT "WHEN COMPLETED, PRESS 'RETURN' KEY."

270 GET R$ : IF R$ <> CHR$(13) THEN 270
280 OPEN 1, 1, 0, "GROCERY"
290 REM   READ NUMBER OF DATASETS
300 INPUT#1, K
310 PRINT: INPUT "HOW MANY DATASETS WILL BE ADD
    ED "; A
320 DIM G$(K+A), N(K+A)
330 :
```

(a) What does line 300 do? _____

_____

(b) Explain what line 320 does. _____

_____

(c) Complete lines 360 and 380:

```
330 :
340 REM  LOAD FILE INTO ARRAY
350 FOR X = 1 TO K
360 _____ : REM INPUT
      DATASET FROM FILE
365 PRINT "DATA #" X ;G$(X),N(X)
370 NEXT X
380 _____ : REM CLOSE
      READ FILE
390 :
```

(d) Examine the FOR NEXT loop in the next program segment and complete lines 460 and 470. Line 410 (and 480) control how many new datasets are entered. Line 410 gives the number of the next empty element in the arrays G$ and N.

```
380 :
390 REM   DATA ENTRY ROUTINE
400 FOR X = 1 TO A
410 LET K = K + 1
420 INPUT "ITEM DESCRIPTION "; D$
430 INPUT "NUMBER NEEDED "; N
440 :
450 REM   PLACE DATA INTO ARRAY
460 _____________________________ : REM LOAD
     D$ INTO  ARRAY
470 _____________________________ : REM LOAD
     N INTO ARRAY
480 NEXT X
490 :
```

(a) Reads the first data item in the file—the number of datasets in the file.
(b) Dimensions arrays G$ and N to size K (the number of datasets currently in the file) plus A (the number of datasets to be added).

(c)
```
360 INPUT#1, G$(X), N(X)
```

(d)
```
470 LET G$(K) = D$
480 LET N(K) = N
```

(a) And now to copy the data-filled arrays onto the new cassette data file (or over the old one). Complete lines 570, 630, and 660:

```
500 :
510 REM   RECORDER INSTRUCTIONS
520 PRINT "[CLR]" : REM CLEAR SCREEN
530 PRINT "PLACE REWOUND TAPE IN RECORDER; IF N
     ECESSARY, REWIND."
540 PRINT "MAKE SURE NO RECORDER KEY IS DOWN, T
     HEN PRESS 'RETURN' TO CONTINUE."
550 GET R$ : IF R$ <> CHR$(13) THEN 550
560 REM   OPEN FILE
570 _____________________________ : REM OPEN
     FILE
580 :
590 REM  PRINT NUMBER OF DATASETS TO FILE
600 _____________________________ : REM # DAT
     ASETS
610 REM  PRINT ARRAY CONTENTS INTO FILE
620 FOR X = 1 TO K
630 _____________________________ : REM PRINT
     ARRAY TO FILE
640 NEXT X
650 :
660 _____________________________ : REM CLOSE
     FILE
670 PRINT "FILE CLOSED, JOB COMPLETE."
680 PRINT "REWIND TAPE AND REMOVE."
690 :
```

(a) Here is the complete program.

```
100 REM  ADD TO CASSETTE GROCERY FILE
110 :
120 REM  VARIABLES USED
130 REM    D$ = ITEM DESCRIPTION
140 REM    N  = NUMBER NEEDED
150 REM    G$,N = ARRAYS
160 REM    R$ = RESPONSE STRING
170 REM    K  = DATASET COUNT
180 REM    A  = # OF ADDED DATASETS
190 :
200 REM  FILES USED
210 REM    CASSETTE FILE = GROCERY
220 REM    DATASET FORMAT: D$, N
230 :
240 REM  FILE INITIALIZATION
250 PRINT "INSERT 'GROCERY' CASSETTE, REWIND, A
     ND PRESS RECORDER'S 'STOP' KEY."
260 PRINT "WHEN COMPLETED, PRESS 'RETURN' KEY."

270 GET R$ : IF R$ <> CHR$(13) THEN 270
280 OPEN 1, 1, 0, "GROCERY"
290 REM   READ NUMBER OF DATASETS
300 INPUT#1, K
310 PRINT: INPUT "HOW MANY DATASETS WILL BE ADD
     ED "; A
320 DIM G$(K+A), N(K+A)
330 :
340 REM  LOAD FILE INTO ARRAY
350 FOR X = 1 TO K
360 INPUT#1, G$(X), N(X) : REM INPUT DATASET FR
     OM FILE
365 PRINT "DATA #" X ;G$(X),N(X)
370 NEXT X
380 CLOSE 1 :REM CLOSE READ FILE
390 :
400 REM   DATA ENTRY ROUTINE
410 FOR X = 1 TO A
420 LET K = K + 1
430 INPUT "ITEM DESCRIPTION "; D$
440 INPUT "NUMBER NEEDED "; N
450 :
460 REM   PLACE DATA INTO ARRAY
470 LET G$(K) = D$    : REM LOAD D$ INTO  ARRAY

480 LET N(K) = N : REM LOAD N INTO ARRAY
490 NEXT X
500 :
```

```
510 REM   RECORDER INSTRUCTIONS
520 PRINT "[CLR]" : REM CLEAR SCREEN
530 PRINT "PLACE REWOUND TAPE IN RECORDER; IF N
     ECESSARY, REWIND."
540 PRINT "MAKE SURE NO RECORDER KEY IS DOWN, T
     HEN PRESS 'RETURN' TO CONTINUE."
550 GET R$ : IF R$ <> CHR$(13) THEN 550
560 REM   OPEN FILE
570 OPEN  2, 1, 2, "GROCERY" : REM OPEN FILE
580 :
590 REM  PRINT NUMBER OF DATASETS TO FILE
600 PRINT#2, K            : REM # DATASETS
610 REM  PRINT ARRAY CONTENTS INTO FILE
620 FOR X = 1 TO K
630 PRINT#2, G$(X) : PRINT#2, N(X) : REM PRINT
     ARRAY TO FILE
640 NEXT X
650 :
660 CLOSE 2               : REM CLOSE FILE
670 PRINT "FILE CLOSED, JOB COMPLETE."
680 PRINT "REWIND TAPE AND REMOVE."
690 :
```

CONVERTING DISK FILE PROGRAMS TO CASSETTE FILES

You now know the basics of cassette file operation, and will be able to not only write programs using cassette files, but also to convert programs using disk files to the use of cassette files. There are three major areas of change:

1. the OPEN statement;
2. the procedures for copying and editing files;
3. the procedures for checking for errors.

DIFFERENCES IN THE OPEN STATEMENT

Here are some typical disk file OPEN statements:

```
OPEN 1, 8, 8, "TEMPFILE,S,R"
OPEN 2, 8, 5, "TEMPFILE,S,W"
OPEN 2, 8, 8, "@0:TEMPFILE,S,W"
```

As for cassettes, the first three numbers are the file number, the device number, and the secondary address. For disks, the file number can be any number from 1 to 127, the device number is 8, and the secondary address can be any number from 2 to 127. The secondary address is *not* significant for disk files, unlike for cassette files.

(Actually, as with cassettes, file numbers can be any number from 1 to 255, and file numbers over 127 send linefeed plus carriage return, whereas those less than 127 send only carriage return. The device number can be any number from 8 to 15, but 8 is common; a second drive is sometimes used as device 9. The secondary address assigns a buffer in the disk drive, which is important for disk file programs, but need not concern you in converting for cassette file use.)

The information in double quotes requires some interpretation. The "S" stands for "sequential"; disks support another file type, called *relative*. Only *sequential* file programs can be converted to cassette. The R and W indicate READ or WRITE. For cassette files, these are designated by secondary addresses of 0 for READ, and 1 or 2 for WRITE. The "@0:" in the last example allows the file to replace an existing file (like copying over the same tape with cassette files), and is simply deleted when translating for cassette files.

Here's your chance to practice converting disk sequential file OPEN statements for cassette files:

(a) `100 OPEN 1, 8, 8, "GROCERY,S,R"`

(b) `200 OPEN 5, 8, 5, "@0:QUALITY CONTROL,S,W"`

(c) `500 OPEN 2, 8, 3, "MAIL LIST,S,R"`

(d) `100 OPEN 8, 8, 8, "PHONE LIST,S,W"`

(a) `100 OPEN 1, 1, 0, "GROCERY"`

(b) `200 OPEN 5, 1, 1, "QUALITY CONTROL"`

or

`200 OPEN 5, 1, 2, "QUALITY CONTROL"`

(c) `500 OPEN 2, 1, 0, "MAIL LIST"`

(d) `100 OPEN 8, 1, 1, "PHONE LIST"`

or

`100 OPEN 8, 1, 2, "PHONE LIST"`

CHANGES IN PROCEDURES FOR COPYING AND EDITING FILES

With a disk drive, it is possible to have two (or more) files open at the same time. Copying and editing of files is done by opening the existing file for READ and a temporary file for WRITE, then READing a dataset, and either copying it by WRITing it to the temporary file, or displaying it for editing, then WRITing the edited version to the temporary file. Later, the temporary file is renamed. Obviously, the procedure is different for cassette files—but you have already learned it! Recall how an array was used to hold data temporarily in the computer's memory, then the data was written to a new cassette. Here's an example of a disk copy program, followed by the version for cassette:

```
100 REM   SEQUENTIAL FILE COPYING PROGRAM
110 :
120 REM   VARIABLES USED
130 REM    D1$ = CONCATENATED DATASET
140 REM    F$, FA$ = USER ENTERED FILE NAMES
150 :
160 REM   FILES USED
170 REM      SEQ. FILE = TRANSAC1, TRANSAC1.CPY
180 REM      DATASET FORMAT: D1$ (CONCATENATED F
    ROM A$ + T$ + C$)
190 :
200 REM   FILE INITIALIZATION
210 :
220 INPUT "NAME OF INPUT FILE "; F$
230 INPUT "NAME OF WRITE FILE "; FA$
240 OPEN 1, 8, 8, F$ + ",S,R"
250 OPEN 2, 8, 8, FA$ + ",S,W"
260 :
270 REM   READ INPUT FILE, WRITE TO OUTPUT FILE

280 :
290 INPUT#1, D1$
300 LET SV = ST
310 PRINT#2, D1$
320 IF SV = 0 THEN 290
330 :
340 REM   CLOSE FILE ROUTINE
350 CLOSE 1 : CLOSE 2
390 :
```

```
100 REM   CASSETTE FILE COPYING PROGRAM
110 :
120 REM   VARIABLES USED
130 REM    D$ = CONCATENATED DATASET
140 REM    F$, FA$ = USER ENTERED FILE NAMES
145 REM    N = NUMBER DATASETS FOR ARRAY
150 :
160 REM   FILES USED
170 REM     SEQ. FILE = TRANSAC1, TRANSAC1.CPY
180 REM     DATASET FORMAT: D$ (CONCATENATED FR
    OM A$ + T$ + C$)
190 :
200 REM   FILE INITIALIZATION
210 :
220 INPUT "NAME OF INPUT FILE "; F$
225 INPUT "NAME OF OUTPUT FILE "; FA$
230 PRINT : INPUT "ABOUT HOW MANY DATASETS IN F
    ILE "; N
231 DIM D$(N)
232 PRINT "INSERT CASSETTE " F$ ", REWIND, AND
    PRESS 'STOP' KEY ON RECORDER."
234 PRINT "WHEN CASSETTE REWOUND, PRESS 'RETURN
    ' KEY."
236 GET Z$ : IF Z$ <> CHR$(13) THEN 236
240 OPEN 1, 1, 0, F$
260 :
270 REM   READ INPUT FILE, STORE AS ARRAY
280 :
285 FOR C = 1 TO N
290 INPUT#1, D$(C) : PRINT C; D$(C)
300 IF ST = 0 THEN NEXT C
305 LET N = C - 1 : REM SET ACTUAL NUMBER
310 CLOSE 1 : PRINT "FILE CLOSED.  INSERT TAPE
    FOR COPY."
320 PRINT "REWIND, AND PRESS 'STOP' KEY ON RECO
    RDER."
324 PRINT "WHEN CASSETTE REWOUND, PRESS 'RETURN
    ' KEY."
326 GET Z$ : IF Z$ <> CHR$(13) THEN 326
330 OPEN 2, 1, 2, FA$
340 FOR C = 1 TO N
350 PRINT#2, D$(C) : PRINT "WRITING #" C "--" D
    $(C)
360 NEXT C
370 CLOSE 2
380 PRINT "FILE CLOSED. REWIND COPY TAPE AND RE
    MOVE"
390 :
```

Now to try your hand at a bigger task. Here are parts of the file-editing program from Chapter 6. Make the revisions necessary for the program to work with cassette files.

EDIT SEQUENTIAL FILE—DISK VERSION

```
100 rem   credit file editor (1)
101 rem   disk files version
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     c$ = cust. #
150 rem     c1$ = cust. #
160 rem     n$ = cust. name
170 rem     r$ = user response
180 rem     r, r1 = credit rating value
190 :
200 rem   files used
210 rem     seq. files: credit (user entered),
     tempfile
220 rem     dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "File name "; f$
265 open 15, 8, 15 : rem disk error channel
270 open 1, 8, 8, f$ + ",s,r" : gosub 810
280 open 2, 8, 2, "tempfile,s,w" : gosub 810
290 :
300 rem   data entry routine
310 print "Enter 'STOP' for customer number if
     no more changes."
320 :
330 print : input "Customer number "; c$
340 if c$ = "stop" or c$ = "STOP" then 780
350 if c$ = "" then print "Enter a number or ty
     pe 'STOP'." : goto 330
360 if len(c$) <> 5 then print "Entry error.  U
     se 5 digits." : goto 330
370 if val(c$) = 0 then print "Entry error; num
     bers only." : goto 330
380 :
390 rem   file search routine
400 input#1, c1$, n$, r
410 let sv = st
420 if c$ = c1$ then 500 : rem match
430 print#2, c1$, n$, r
440 if sv = 0 then 400
450 print "Error: Cust. # " c$
```

```
455 print "is not in the file.  Check your numb
    er  and re-enter."
460 close 1 : close 2 : rem reset file  pointer
    s
470 print : goto 270
480 :
490 rem   cust # found, proceed with data entry

500 print
510 print n$ "  Current credit rating: " r
520 print : input "New credit rating "; r$
530 let r1 = val(r$)
540 if len(r$) <> 1 then print "Enter one digit
     only." : goto 520
550 if r1 < 1 or r1 > 5 then print "Numbers fro
    m 1 - 5 only." : goto 520
560 :
570 rem   print new info to tempfile
580 print#2, c$, n$, r1
590 :
600 rem   print remainder of credit file to tem
    pfile
610 input#1, c$, n$, r
620 let sv = st
630 print#2, c$, n$, r
640 if sv = 0 then  610
650 :
660 rem   close files
670 close 1 : close 2
680 :
690 rem   delete original and rename tempfile
700 print#15, "s0:" + f$ : gosub 810
710 print#15, "r0:" + f$ + "=0:tempfile" : gosu
    b 810
720 :
730 rem   continue request
740 print "[clr]" : rem clear screen
750 print "More changes ('Y' for YES or 'N' for
     NO) ?"
760 get r$ : if r$ = "Y" or r$ = "y" then print
     "[clr]" : goto 270
770 if r$ <> "n" and r$ <> "N" then 760
780 print "Job completed."
790 close 1 : close 2 : close 15
799 end
800 :
810 rem   disk error subroutine
820 input#15, x, x$, y, z
830 if x < 20 then return : rem no error
840 print  "disk error # " x ", " x$
850 close 15 : rem close all files and quit
860 end
```

EDIT SEQUENTIAL FILE—CASSETTE VERSION

```
100 rem   credit cassette file editor (1)
101 if peek(50003) <> 0 then poke59468,14 : rem
     pet lower case
102 if peek(50003) = 0 then poke53272,23 : rem
     c-64 lower case
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     c$ = cust. #
150 rem     c$(x,1) = cust. #
160 rem     c$(x,2) = cust. name
170 rem     r$ = user response
180 rem     c$(x,3),r,r1 = credit rating value
185 rem     c$ = array              **
186 rem     k = number of datasets **
187 rem  ** indicates lines changed from disk v
     ersion
190 :
200 rem   files used
210 rem     seq. files: credit (user entered),
     tempfile
220 rem     dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "File name "; f$
270 open 1, 1, 0, f$   : rem **
280 input#1, k : dim c$(k,3) : for x = 1 to k :
     rem **
290 input#1, c$(x,1), c$(x,2), c$(x,3) : rem **

295 next x : rem read into array  **
300 rem   data entry routine
310 print : print "Enter 'STOP' for customer nu
     mber if no more changes."
320 :
330 print : input "Customer number "; c$
340 if c$ = "stop" or c$ = "STOP" then 610 : re
     m done
350 if c$ = "" then print "Enter a number or ty
     pe 'STOP'." : goto 330
360 if len(c$) <> 5 then print "Entry error.  U
     se 5 digits." : goto 330
370 if val(c$) = 0 then print "Entry error; num
     bers only." : goto 330
380 :
390 rem   file search routine
400 for x = 1 to k : rem **
```

```
410 if val(c$) = val(c$(x,1)) then 500 : rem ma
    tch  **
420 next x : rem **
430 : rem **
440 : rem **
450 print "Error: Cust. # " c$
455 print "is not in the file.  Check your numb
    er and re-enter."
460 : rem **
470 print : goto 300 : rem **
480 :
490 rem   cust # found, proceed with data entry
500 print
510 print c$(x,2) "  Current credit rating: " c
    $(x,3)
520 print : input "New credit rating ";r$
530 let r1 = val(r$) : let l = len(r$)
540 if l <> 1 then print "Only a one digit numb
    er is acceptable." : goto 520
550 if r1 < 1 or r1 > 5 then print "Numbers fro
    m 1 - 5 only." : goto 520
560 :
570 rem   print new info to array **
580 let c$(x,3) = r$
590 goto 300 : rem **
600 rem   print remainder of credit file to tem
    pfile--delete 600-770 **
610 rem   write array to tape
620 close 1 : print "Rewind and remove old file
    ."
630 print "Insert and rewind cassette for new f
    ile."
640 print "Be sure all keys are up on cassette,
    "
650 print "then press 'RETURN'"
660 get z$ : if z$ <> chr$(13) then 660
670 open 2, 1, 2, f$
680 print#2, k : rem number of datasets
690 for c = 1 to k
700 print#2, c$(c,1) : print#2, c$(c,2) : print
    #2, c$(c,3)
710 next c
720 close 2
730 print "Rewind and remove your cassette."
740 print "Job completed."
```

In the preceding cassette version, changes from the disk version have been marked with REM**.

DEALING WITH ERROR AND OPERATOR MESSAGES

Disk systems have a different error system than cassettes. Fortunately, the lines necessary for disk error detection are easy to spot. The secret is a secondary address of 15. Here's one:

```
120  OPEN 1, 8, 15
```

For cassette files you will need to delete the OPEN and the corresponding CLOSE statements from the program.

In addition, a well-written disk program should read the error channel after each disk access to detect and deal with errors. This is usually done as a subroutine, which commonly looks like this:

```
2000 REM DISK ERROR READ
2010 INPUT#1, X, X$, Y, Z
2020 IF X < 20 THEN RETURN
2030 PRINT "DISK ERROR--" X$
2040 END
```

or, more simply, as

```
2000 REM DISK ERROR READ
2010 INPUT#1, X, X$, Y, Z
2020 PRINT X$ : RETURN
```

If you find such a subroutine, you will need to delete it and all GOSUBs which reference it. You should then carefully check the program for any INPUT# statements with the file number opened to the disk error channel, and remove them. At worst, when you run the program it will stop with a FILE NOT OPEN ERROR because you deleted the OPEN statement, but overlooked one of the INPUT# statements.

(a) Which lines must be deleted from program "EDIT SEQUENTIAL FILE—DISK VERSION on page ? _____

_____

_____

(a) 270 and 280 : gosub 810
700 and 710 (delete entire lines)
790 : close 15
810–890 (delete entire lines)

ST is the same for both disk and tape, so no change is necessary there.

The other change necessary is in PRINT statements to the user of your program. Disk files take care of much of the handling internally that must be done by the user for a tape recorder—such things as rewinding the tape, pressing the appropriate keys on the recorder, etc. In translating from disk to tape, you will have to add appropriate PRINT statements such as

```
250 PRINT "INSERT CASSETTE AND REWIND"
260 PRINT "THEN PRESS 'STOP' KEY ON RECORDER."
270 PRINT "PRESS 'RETURN' WHEN COMPETED"
280 GET Z$ : IF Z$ <> CHR$(13) THEN 280
290 PRINT "PRESS BOTH THE 'PLAY' AND 'RECORD' KEYS"
300 PRINT "ON THE RECORDER NOW"
```

WRITING PROGRAMS FOR DISK OR TAPE FILES

It is not difficult to write a program which allows the user to select either tape or disk files. After the usual initialization routine, we ask "Cassette or Disk?" using a GET routine to pick up either the C or D. Likewise, we can ask "Read or Write?" using GET again to determine which choice. The program so far looks like this:

```
200 print "[rvs]C[off]assette or [rvs]D[off]isk ? "
210 get zd$ : if zd$ <> "c" and zd$ <> "d" then 210
220 print "[rvs]W[off]rite or [rvs]R[off]ead ? "
230 get zm$ : if zm$ <> "w" and zm$ <> "r" then 230
240 input "[down]File name "; f$
250 if zm$ = "r" then 500 : rem read
290 :
300 rem    open files for write
```

Now we need to write the OPEN statements for both cassette and disk files. While it would be possible to write it as one statement, it is easier to use IF . . . THEN statements and write separate OPEN statements for disk and tape. You write the two open statements:

(a) 310 IF _____

(b) 320 _____

(a) `310 if zd$ = "c" then open 1, 1, 2, f$ : goto 330`
(b) `320 open 1, 8, 8, f$ + ",s,w"`

(You might have written line 320 in the same format as 310, without the GOTO.)

Here's the rest of the WRITE part of the program; complete the missing lines (350 and 390):

```
330 input "Name "; n$
340 input "Age "; a
350 _____________________________ :  rem wr
     ite data to file
360 print "[down][rvs]C[off]ontinue or [rvs]S[o
     ff]top ?"
370 get z$ : if z$  <> "c" and  z$ <> "s" then
     370
380 if z$  = "c" then 330
390 _____________________________ :  rem cl
     ose file
400 rem end program
410 print "[down][rvs]R[off]ead file or [rvs]S[
     off]top ?"
420 get z$ : if z$ <> "r" and z$ <> "s" then 42
     0
430 if z$ = "s" then end
490 :
```

```
350 print#1, n$ : print#1, a

390 close 1
```

The same format can be used for READing either a cassette or disk file. You should be able to easily complete the program. Again, fill in lines 510, 520, 530, 560, and 570.

```
500 rem   open file for read
510 _____________________________ : rem  ope
    n tape file
520 _____________________________ : rem  ope
    n disk file
530 _____________________________ : rem  rea
    d data
540 print "Name: " left$(n$ + "        ",6);
550 print "Age: " right$("    " + str$(a),3)
560 _____________________________ : rem chec
    k for end of file
570 _____________________________ : rem clos
    e file
580 print "[down]File closed, all data displaye
    d."
590 goto 400 : rem end program
```

```
510 if zd$ = "c" then open 2, 1, 0, f$ : rem open tape file
520 if zd$ = "d" then open 2, 8, 2, f$ + ",s,r" : rem open disk file
530 input#2, n$, a          : rem read data
560 if st = 0 then 530      : rem check for end of file
570 close 2                 : rem close file
```

That wasn't so painful, was it? Now you can write file programs that will use whichever system is available—tape or disk. Of course, we haven't put in all the prompts that are helpful with a tape system. This could be done with a statement of the form IF zd$ = "c" THEN GOSUB xxx, which checks for device "c", as above, and sends the program to a subroutine of the appropriate instructions.

You are now ready to try your hand at writing your own programs for cassette data files, in the Self-Test following. With the information in this last section, you will also be able to rewrite any of the programs in Chapters 5 and 6 from disk to cassette tape files.

If you get a disk drive at some future date, you will find the knowledge you have gained about cassette files will help you to quickly learn how to write disk sequential files.

Chapter 4 Self-Test

1. (a) Write a program to enter datasets into a cassette file. Each dataset contains two string data items, followed by two numeric data items. Include a user response for "MORE DATA?" after each dataset entry and close the file after the last data item.

```
100 :
120 REM     VARIABLE LIST
130 REM       A$,B$ = ALPHA DATA
140 REM       C$(C), D$(D) = NUMERIC DATA
150 :
```

1. (b) Write a companion program to Problem 1(a) to display the contents of the cassette file. Include the end of file test.

```
110 :
120 REM     VARIABLE LIST
130 REM       A$,B$ = ALPHA DATA
140 REM       C,D = NUMERIC DATA
150 :
```

2. (a) Write a grocery list program that allows you to enter the following into a cassette file:

   Item description - twenty characters maximum
   Quantity to buy

   The data entry tests should include a section that displays the entry if the quantity is less than one or more than ten and allows the user to reenter that quantity if needed.

```
120 :
130 REM     VARIABLES USED
140 REM       D$ = ITEM DESCRIPTION
150 REM       Q = QUANTITY TO BUY
160 REM       R$ = USER RESPONSE
170 :
```

2. (b) Write a companion program to the one written in 2(a) to display the grocery file contents.

```
110 :
120 REM     INTRODUCTORY MODULE
130 REM      VARIABLES USED
140 REM        D$ = ITEM DESCRIPTION
150 REM        Q = QUANTITY TO ORDER
160 :
```

3. (a) Write a program to create a cassette file mailing list, as indicated in the variable list below. Concatenate the data items into one dataset per person.

```
110 :
120 REM    VARIABLES USED
130 REM    N$(20) = NAME
140 REM    A$(20) = ADDRESS
150 REM    C$(10) = CITY
160 REM    S$(2) = STATE
170 REM    Z$(5) = ZIP CODE
180 REM    D$(57) = ENTIRE DATASET
190 REM    R$ = USER RESPONSE VARIABLE
200 :
```

3. (b) Write a companion program to count each data item as it is displayed and display the number of the total datasets in the file (1,2,3. . . . Total is 14).

```
110 :
120 REM     VARIABLES USED
130 REM     D$ = ONE ENTIRE DATASET (57 CHAR.)
140 REM     K = DATASET COUNTING VARIABLE
150 :
```

3. (c) Write a program, including recorder user instructions, to make a copy of the mailing list on a separate tape.

```
110 :
120 REM VARIABLES USED
130 REM     D$ = ONE COMPLETE DATASET (57 CHARS.)
140 REM     K = NO. OF DATASETS IN FILE
141 REM     X = FOR-NEXT LOOP CONTROL VARIABLE
142 REM     R$ = 'PRESS ENTER TO CONTINUE' RESPONSE VARIABLE
150 :
```

Answer Key

These programs are the minimum program necessary. We've tried to illustrate some of the different possible solutions—you'll note several ways of checking for a null response, for example.

Pay particular attention to the difference between the first and second solution to Problem 1(b). The first is the minimal solution—it runs perfectly. The second solution has some added features that make the program much easier to use. The first is a title. It's nice to know which program you're using! The second feature is the use of clearing the screen to prevent scrolling of data from the bottom of the screen. It makes things neater and easier to read. The last feature is specific instructions regarding the cassette. We find the reminder "Rewind and remove cassette" personally helpful, and more useful than "File read and closed."

```
100 REM   PROB  4-1A SOLUTION
110 :
120 REM   VARIABLE LIST
130 REM      A$,B$ = ALPHA DATA
140 REM      C$(C), D$(D) = NUMERIC DATA
150 :
160 OPEN 1, 1, 2, "FILE DATA"
170 :
180 REM   DATA ENTRY ROUTINE
190 PRINT : INPUT "STRING DATA #1 "; A$
200 IF A$ = "" THEN PRINT "PLEASE ENTER SOMETHI
    NG." : GOTO 190
210 PRINT : INPUT "STRING DATA #2 "; B$
220 IF B$ = "" THEN PRINT "PLEASE ENTER SOMETHI
    NG." : GOTO 210
230 PRINT : INPUT "NUMERIC DATA "; C$ : LET C =
     VAL(C$)
240 IF C$ = "" THEN PRINT "PLEASE ENTER A NUMBE
    R." : GOTO 230
250 IF C = 0 THEN PRINT "ENTER NUMBERS ONLY, PL
    EASE." : GOTO 230
260 :
270 PRINT : INPUT "NUMERIC DATA #2 "; D$ : LET
    D = VAL(D$)
280 IF LEN(D$) = 0 THEN PRINT "PLEASE ENTER A N
    UMBER." : GOTO 270
290 IF D = 0 THEN PRINT "ENTER NUMBERS ONLY." :
     GOTO 270
300 :
310 PRINT#1, A$ : PRINT#1, B$ : PRINT#1, C : PR
    INT#1, D
320 :
330 PRINT "MORE DATA (PRESS 'Y' OR 'N') ?"
340 GET R$ : IF R$ = "" THEN 340
350 IF R$ = "Y" THEN 190
360 IF R$ <> "N" THEN 340
370 :
380 REM   END OF DATA--CLOSE FILE
390 CLOSE 1
400 END
```

```
100 REM   PROB 4-1B SOLUTION
110 :
120 REM   VARIABLE LIST
130 REM     A$,B$ = ALPHA DATA
140 REM     C, D = NUMERIC DATA
150 :
160 REM   OPEN FILE
170 OPEN 1, 1, 0, "FILE DATA"
180 :
190 REM   READ FILE
200 PRINT "STRG. 1", "STRG. 2", "NUM. 1", "NUM.
    2"
210 INPUT#1, A$, B$, C, D
220 PRINT A$, B$, C, D
230 IF ST = 0 THEN 210
240 :
250 REM   CLOSE FILE
260 CLOSE 1
270 END
```

ENHANCED VERSION

```
20 REM   'IMPROVED' VERSION
100 REM   PROB 4-1B SOLUTION
110 :
120 REM   VARIABLE LIST
130 REM     A$,B$ = ALPHA DATA
140 REM     C, D = NUMERIC DATA
150 :
160 REM   OPEN FILE
162 PRINT "[CLR][DOWN][DOWN]   READ FILE DATA F
    ILE "
163 PRINT : PRINT "INSERT 'FILE DATA' TAPE AND
    REWIND."
164 PRINT : PRINT "THEN PRESS 'RETURN'."
166 GET R$ : IF R$ <> CHR$(13) THEN 166
170 OPEN 1, 1, 0, "FILE DATA"
180 :
190 REM   READ FILE
195 PRINT "[CLR][DOWN][DOWN]" : REM CLEAR, CRSR
    DOWN 2
200 PRINT "STRG. 1", "STRG. 2", "NUM. 1", "NUM.
    2"
210 INPUT#1, A$, B$, C, D
220 PRINT A$, B$, C, D
230 IF ST = 0 THEN 210
240 :
250 REM   CLOSE FILE
260 CLOSE 1
270 PRINT : PRINT "FILE READ AND CLOSED."
280 PRINT : PRINT "REWIND AND REMOVE CASSETTE."

290 END
```

```
100 REM   PROBLEM 4-2A SOLUTION
110 :
120 REM   INTRODUCTORY MODULE
130 REM   VARIABLES USED
140 REM     D$ = ITEM DESCR. (20 CHRS)
150 REM     Q = QUANTITY TO BUY
160 REM     R$ = USER RESPONSE
170 :
180 REM   INITIALIZE FILE
190 OPEN 1, 1, 2, "ORDER"
200 :
210 REM   DATA ENTRY ROUTINE
220 PRINT : PRINT "TYPE '-999' TO STOP."
230 PRINT : INPUT "ITEM DESCRIPTION "; D$
240 IF VAL(D$) = -999 THEN 410
250 IF LEN(D$) = 0 THEN PRINT "ENTER A DESCRIPT
    ION OR '-999'." : GOTO 230
260 IF LEN(D$) > 20 THEN PRINT "LIMIT DESCRIPTI
    ON TO 20 CHARACTERS." : GOTO 230
270 :
280 PRINT : INPUT "QUANTITY "; Q$ : LET Q = VAL
    (Q$)
290 IF Q = 0 THEN PRINT "NUMBERS ONLY!" : GOTO
    280
300 IF Q >= 1 AND Q < 10 THEN 370
310 PRINT : PRINT "YOU ENTERED A QUANTITY OF: "
    Q
320 PRINT "IS THAT WHAT YOU WANTED ('Y' OR 'N')

330 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 33
    0
340 IF R$ = "N" THEN 280
350 :
360 REM   WRITE TO FILE
370 PRINT#1, D$ : PRINT#1, Q
380 GOTO 220
390 :
400 REM   CLOSE FILE
410 CLOSE 1
420 PRINT "FILE WRITTEN. REWIND & REMOVE CASSET
    TE."
430 END
```

```
100 REM   PROB 4-2B SOLUTION
110 :
120 REM   INTRODUCTORY MODULE
130 REM   VARIABLES USED
140 REM     D$ = ITEM DESCRIPTION
150 REM     Q  = QUANTITY TO  ORDER
160 :
170 REM   OPEN FILE
180 OPEN 1
190 :
200 REM   READ AND PRINT FILE
210 PRINT "ITEM" TAB(24) "QUANTITY"
220 INPUT#1, D$, Q
230 PRINT D$ TAB(24) Q
240 IF ST = 0 THEN 220
250 :
260 REM   CLOSE ROUTINE
270 CLOSE 1
280 PRINT : PRINT "FILE READ AND CLOSED."
290 END


100 REM   PROB 4-3A SOLUTION
110 :
120 REM   INTRODUCTORY MODULE
130 REM   VARIABLES USED
140 REM     N$ = NAME (20)
150 REM     A$ = ADDRESS (20)
160 REM     C$ = CITY (10)
170 REM     S$ = STATE (2)
180 REM     Z$ = ZIP (5)
190 REM     D$ = ENTIRE DATASET (67)
200 REM     R$ = USER RESPONSE
210 REM     F$ = FILE NAME (USER)
220 :
230 REM   INITIALIZE
240 PRINT "[CLR][DOWN][DOWN]" : REM CLEAR & CRS
    R DOWN 2
250 PRINT "ADDRESS FILE WRITER" : PRINT : PRINT

260 INPUT "NAME FOR FILE "; F$
270 OPEN  1, 1, 2, F$
280 :
290 REM   DATA ENTRY/TESTS
300 PRINT : INPUT "NAME "; N$
310 REM   DATA TESTS
320 LET N$ = LEFT$(N$ + "                    ",
    20)
330 :
340 PRINT : INPUT "ADDRESS "; A$
350 REM   DATA TESTS
360 LET A$ = LEFT$(A$ + "                    ",
    20)
```

```
370 :
380 PRINT : INPUT "CITY "; C$
390 REM   DATA TESTS
400 LET C$ = LEFT$(C$ + "                    ",
    20)
410 :
420 PRINT : INPUT "STATE "; S$
430 REM   DATA TESTS
440 IF LEN(S$) <> 2 THEN PRINT "ENTER 2 LETTER
    CODE." : GOTO 420
450 :
460 PRINT : INPUT "ZIP CODE "; Z$
470 IF LEN(Z$) <> 5 THEN PRINT "ENTER 5 DIGIT Z
    IP CODE." : GOTO 460
480 :
490 LET D$ = N$ + A$ + C$ + S$ + Z$
500 :
510 PRINT#1, D$
520 :
530 PRINT : PRINT "MORE ENTRIES ('Y' OR 'N')
540 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 54
    0
550 IF R$ = "Y" THEN 300
560 :
570 REM   CLOSE FILE
580 CLOSE 1
590 PRINT "FILE '" F$ "' WRITTEN AND CLOSED."
600 END


100 REM   PROB 4-3B SOLUTION
110 REM   READ ADDRESS FILE
120 :
130 REM   VARIABLES USED
140 REM      D$ = ENTIRE DATASET (57)
150 REM      K  = DATASET COUNTER
160 :
170 REM   INITIALIZE
180 OPEN 1 : REM DEFAULT READ
190 LET K = 0
200 :
210 REM   READ, COUNT, & DISPLAY DATA
220 INPUT#1, D$
230 K = K + 1
240 PRINT K "  " D$
250 IF ST = 0 THEN 220
260 :
270 REM   CLOSE ROUTINE
280 PRINT "TOTAL DATASETS =" K
290 CLOSE 1
300 PRINT "FILE READ AND CLOSED."
310 END
```

```
100 REM PROB 4-3C SOLUTION
110 REM COPY ADDRESS FILE
120 :
130 REM     VARIABLES USED
140 REM       D$ = ENTIRE DATASET (57)
150 REM       K  = NUMBER OF DATASETS
160 REM       X  = FOR/NEXT LOOP VAR.
170 REM       R$ = USER RESPONSE
180 :
190 REM     INITIALIZE
200 PRINT "[CLR][DOWN][DOWN]ADDRESS FILE COPIER
    "
210 PRINT : INPUT "ABOUT HOW MANY ITEMS IN THIS
     FILE "; K
220 REM 'K' COULD BE READ AS FIRST VARIABLE ON
    TAPE
230 DIM A$(K)
240 OPEN 1
250 :
260 REM     READ DATASETS INTO ARRAY
270 FOR X = 1 TO K
280 INPUT#1, A$(X) : REM READ INTO ARRAY
290 IF ST = 0 THEN NEXT X
300 LET K = X : REM SET K TO NUMBER ACTUALLY RE
    AD
310 CLOSE 1
320 :
330 PRINT : PRINT "STOP RECORDER, REWIND & REMO
    VE TAPE."
340 PRINT "INSERT NEW TAPE, REWIND, THEN PRESS"

350 PRINT "'STOP' KEY ON RECORDER"
360 PRINT "PRESS 'RETURN' WHEN FINISHED."
370 GET R$ : IF R$ <> CHR$(13) THEN 370
380 :
390 REM    OPEN COPY FILE
400 OPEN 2, 1, 2, "ADDRESS.COPY"
410 REM    WRITE # DATASETS
420 PRINT#2, K
430 :
440 REM    COPY ARRAY TO TAPE
450 FOR X = 1 TO K
460 PRINT#2, A$(X)
470 NEXT X
480 :
490 REM    CLOSE ROUTINE
500 CLOSE 2
510 PRINT "COPY COMPLETE.  REWIND AND REMOVE TA
    PE."
520 END
```

CHAPTER FIVE

Creating and Reading Back Sequential Data Files (Disk)

Objectives: When you complete this chapter, you will be able to store and retrieve numeric and/or alphanumeric data in sequential disk data files, using the following BASIC data file statements with the correct formats: OPEN, CLOSE, INPUT#, GET#, and PRINT#. You will be able to appropriately use the DISK commands COPY, INITIALIZE, SCRATCH, and VALIDATE. You will know how to read and use the disk error channel.

INTRODUCTION

A data file is stored alphanumeric information that is separate and distinct from any particular BASIC program. It is located (recorded) on either a magnetic disk, diskette, or cassette tape. This chapter discusses using sequential (also called serial) data files on disks and diskettes. Such files are very similar to cassette tape files, and the working concepts are the same.

In your previous BASIC programming experiences you probably hand-entered all data needed by your programs using INPUT statements. You did this each time you ran your programs. Or, if you had larger amounts of data, you might have entered the data with DATA statements and used the READ statement to access and manipulate the data. In either case, the data were program-dependent; that is, they were part of that one program and not useable by other programs.

A data file is *program-independent*. It is *separate* from any one program and can be accessed and used by many different programs. In most cases, you will use only one program to load a data file with information. But once your data file is loaded (entered and recorded) on disk or cassette tape, you can read the information from that file using many different programs, each performing a different activity with that file's data.

For example, perhaps you have computerized your personal telephone and address directory using data files stored on a disk. You may need just one

program to originally load information into that file and add names to it. (This chapter will show you how.) Another program allows you to select phone numbers from the file using NAME as the selection criterion. You can use still another program to change addresses or phone numbers for entries previously made in the file. Another program could print gummed mailing labels in zip code order using the same data file. You could design yet another program to print names and phone numbers by phone number area code. The possibilities go on and on.

Notice that one data file can be accessed by many different computer programs. The data file is located separately on the disk in a defined place. Each program mentioned copies the information from the disk into the electronic memory of the computer as it is needed by that particular program. Alternatively, the program could transfer information from the computer's memory to be recorded onto the disk.

If you already use your disk to SAVE and/or LOAD BASIC programs, then you have some experience with disk files. When you SAVE a BASIC program, it is recorded on this disk in a file. Such files containing BASIC programs are called *program files*. In contrast, the files discussed in this chapter contain data and are therefore called *data files*. The two types of files are different and are used differently. A BASIC program file contains a copy of a BASIC program that you can LOAD, RUN, LIST, and SAVE. A data file contains information only. You access this information using a BASIC program that includes special BASIC statements that access data files; that is, transfer all or part of the data from the magnetic recording on disk or cassette into the computer's electronic memory so the program can use it. You *cannot* LOAD, RUN, LIST, or SAVE a data file. You access the information using a BASIC program instead.

(a) Describe in general terms how you can access data in a data file.

- - - - - - - - - - - - - - - - - - -

(a) By using a BASIC program that includes special file-accessing BASIC statements.

DATA STORAGE ON DISKS

A magnetic disk has limited data storage capacity that varies from one computer to another, from one size disk to another, and from one recording system to another. In general, a disk file written on one computer cannot be read by another computer. For Commodore disk drives, the file capacity of a diskette is approximately 170,000 bytes. This is equivalent to a fifty-page book! For the 8050 (double-density) disk drive, the capacity is about 527,000 bytes. The term "byte" will be explained shortly.

A "disk" refers to several styles of magnetic storage. Floppy disks are made of a flexible, magnetic-coated plastic, and come in two sizes—8 inch and 5¼ inch. The smaller is often called a diskette. Hard disks are also available for microcomputers. Although more expensive, they have larger data storage capacities. Fortunately, these physical variations do not affect the BASIC statements used to store and access data files.

Other variations occur in the way data are recorded on disks. A disk can be recorded on one or both sides and in more or less space depending upon the drive system. A double-density system records twice as much data in the same space as a single-density system. Commodore's 1540, 1541, 2031, and 4040 drives are single-density drives. The 8050 drive is double density, and the 8250 is double sided, double density. Double-sided means that the disk drive has two recording heads, one on each side of the disk. It does not mean that you turn the disk over.

A closer look at the way in which Commodore single-density drives format a 5 ¼-inch diskette will provide an example. As you see in Figure 5–1, the disk is divided into thirty-five concentric circles, called tracks. Each track is in turn divided into sectors. Table 5–1 shows the number of sectors per track. Most manufacturers use the same number of sectors for each track. Since the outer tracks are longer, information is less crowded. Commodore disk drives record the same information density on each track; sectors are the same length, so more sectors fit on the larger, outer tracks, as Figure 5–1 illustrates.

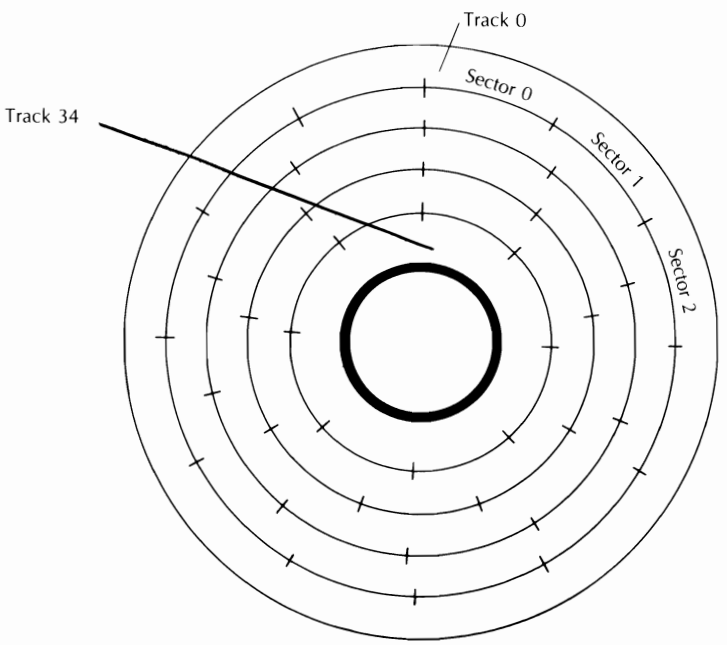


Figure 5–1 *Note:* Sector length remains nearly equal over the disk surface, from one track to another.

TABLE 5–1
SECTORS PER TRACK

Tracks	Sectors
0–16	21
17–23	19
24–29	18
30–34	17

BYTES PER SECTOR: 256
TOTAL SECTORS: 683
RESERVED TRACK: 18—Directory and BAM (Block Availability Map)

What is this thing called a byte? A *byte* is computer jargon for both a unit of computer memory and a unit of disk storage. Each byte has an electronic pattern that corresponds to one alphanumeric character of information. One letter of the alphabet, one special character, or one numeric character entered as a string (such as LET B$="3") takes up one byte of storage space. A twenty-character name takes twenty bytes of disk storage space.

The general rule for storing strings in data files is that the amount of storage needed for each string is equal to the actual length of the string.

(a) How many bytes of disk storage are required by the string assigned to N$?

```
N$ = "BASIC DATA FILES ARE FUN"
```

(a) Twenty-four. (Spaces also take one byte.)

Keeping track of disk storage space requirements for string data is easy, since one character equals one byte. Numeric values are stored on the disk as if they were strings, with a leading place for a sign (space for plus, or – for a minus), the digits, then a CRSR right. Thus a three-digit number takes 5 bytes, and –99.99 takes 7 bytes.

For a personal telephone and address directory application, let's see how much disk storage space is required for each person on file. If each data item has a defined field length, here is how to estimate the byte count for disk storage.

Name	20	characters
Address (street)	25	
City	10	
State	2	
Zip code	5	
Phone (xxx–xxx–xxxx)	12	
Age	2	(Entered as an integer number)
Birthdate (xx/xx/xx)	8	
Subtotal	84	
Overhead	8	(estimate)
Total	92	

Note that string character values are used for the zip code. Since integer values can only go up to 32767, a zip code such as 95472 would cause an overflow error.

(a) How many bytes would be required to store the zip code as an integer value instead of a string? _____

(b) Why was a twelve-character string rather than a numeric value used for the phone number? _____

_____

(c) How many sectors would 150 entries in the address and phone directory take up in storage? _____

_____

(d) What is the maximum number of people you could file in your directory on one disk with a capacity of 168,000 bytes? _____

(a) Seven, plus "overhead."

(b) It could not have included hyphens, which make the number easier to read. (Note that if the telephone number had been entered without hyphens as a numeric value, the precision would have had to be at least ten in order not to lose significant digits from the number.)

(c) 92 × 150 = 13,800 bytes. 13,800 divided by 256 = 54.92, or fifty-five sectors.

(d) 168,000 divided by 92 = 1826.

We call the eight items in each entry in the personal directory a *record* or a *dataset*. A record or dataset consists of all data that are included in one complete transaction or entry into a data file. Grouping information by dataset and then accessing or otherwise manipulating the dataset as a group of data items makes programming and reading programs much easier.

Sequential data files can be visualized as one long, continuous stream of information, with datasets recorded one after the other. Imagine datasets recorded continuously on a magnetic tape cassette (a single, long ribbon of tape) and you have a fairly accurate image of how a sequential file looks in theory. That is how you as a file user should think of it. The truth is, a file can be partially located on one track or one sector, and partially on another, depending on the computer system and how the file was filled. Fortunately, the physical location of the file on a disk is "invisible" to the user. All you need remember is the long, continuous stream of information.

SEQUENTIAL VERSUS RELATIVE DATA FILES

Data filing systems can use sequential data files or relative data files. The latter are explained fully in Chapters 7 and 8. (Commodore uses the term *relative file* for a file type called a *random access file* by most other computers. Commodore uses the term "random access file" to refer to a direct disk access file. These files are difficult to use, and are *not* like random access files on other computers. To avoid confusion, we will consistently use the term *relative* files.)

Sequential data files use disk storage space more efficiently than relative data files. It will quickly become clear to you that a disk is very easy to fill to capacity, despite the seemingly large number of bytes that can be stored on it. Thus, sequential files are *space-efficient*. However, it is somewhat difficult to change data stored in a sequential file. *Sequential files are designed for "permanent" information* that changes infrequently. You can change data in sequential files, but it is not as easy or efficient as in relative files. Thus, another criterion for choosing between sequential and relative data files is how often changes in data can be expected.

Another consideration is the time it takes to access information stored on a disk. When you have a large data file with loads of information, it takes more computer time to find or access a particular dataset at the end of a sequential file than it would in a relative file. To access the 450th data set in a sequential file of 475 data sets, the computer must sequentially search through 449 datasets before coming upon the 450th dataset. Using relative files, the computer can immediately access the 450th record without having to search through the other 449 records. Therefore access time is another factor in selection of sequential or relative data files.

(a) What are three factors to consider when choosing to use sequential or relative data files? _____

_____

(a) Storage space efficiency, changing data, and time for accessing data.

OPENING SEQUENTIAL DATA FILES

To use a data file, you must first OPEN the file and indicate its name. This can be done as part of the program's initialization module.

The file name can be up to sixteen characters long. If you use more than sixteen characters for the file name, you will get either "?SYNTAX ERROR" or "?STRING TOO LONG ERROR."

It is common to end the file name with a few letters to indicate the type of file; this is called an "extension." Usually, the extension indicates the type of file: .FIL might be used for data files, .BAS for BASIC programs, and .WPF for a word processor file. Since Commodore indicates the file type on the disk directory, extensions are unnecessary, but may be convenient.

DIRECTORY

To see the DIRECTORY of a disk, type LOAD "$0",8; when READY. appears, type LIST. You can hold the CTRL key to slow the listing, or press RUN/STOP to stop it. LOADing the directory will erase any program currently in memory—beware!

(The DOS Wedge program allows you to view the directory without erasing your program, one of its many advantages. It is on the Test/Demo disk packed with your drive, or available from local users groups.)

(For PETs and 8032s, you can type DIRECTORY or CATALOG; you will be able to see the directory, but any programs in memory will not be disturbed. Pressing the SPACE bar will stop the display; SPACE again to continue, press RUN/STOP to terminate it.)

The DIRECTORY of the disk will look something like this:

```
0 "DATA FILES DISK" ID 2A
11    "FILE MAKER"      PRG
3     "ADDRESSES"       SEQ
21    "LIBRARY"         REL
629 BLOCKS FREE
```

The first line is called the header; it is the disk's title. The 0 indicates drive 0; with a dual disk drive, you could also display the directory of a disk in drive 1. The disk name can be up to sixteen characters long. ID is a two-character ID code that you specify when you NEW or HEADER the disk. It should be a different two characters for each disk (see Appendices 4 and 5). 2A specifies the DOS version of the ROMs in the disk drive; 2A, also known as DOS 2, is used in the 1540, 1541, 2031, and 4040.

However, be warned that there does seem to be a format difference between the 1540/1541 and the 2031/4040 drives; disks NEWed on the former cannot be BACKUPed on the latter, and there are reports of disks written to on one becoming unreadable on the other.

Following the header, information about the programs on the disk is shown. The number before each name tells how many sectors, or blocks, the program takes up. "FILE MAKER" takes eleven sectors, so is approximately $11 \times 256 = 2816$ bytes long. PRG means it is a BASIC program. Sequential files are labeled with SEQ, as for "ADDRESSES," and relative files are labeled REL, as for "LIBRARY." Finally, the number of sectors (blocks) left on the disk is given. An unused disk has 664 blocks free.

Refer to your manuals for more information about how to load, print, and interpret the directory.

THE OPEN STATEMENT

The OPEN statement tells the computer which data file (or files) this particular program will use. It also provides the computer with other information. The OPEN statement creates a new file if none exists with the name specified, and sets the access mode.

There are two modes for manipulating sequential and relative (random) data files—READ and WRITE. They are usually abbreviated to R or W.

For files which will be read INTO the computer, use R; the computer (you) will be READing information from the disk. To save information on the disk, you WRITE it from the computer. The words mean exactly what you would expect. In BASIC 4.0, there is also an APPEND command for adding information to a sequential file; see Appendix 5.

An OPEN sequential file will be either READ, WRITE, or APPEND (if available); it can be open for only *one* of these modes at a time. To change from WRITE to READ, you must CLOSE and then reOPEN the file.

OPEN also assigns space in the computer memory and disk drive memory, called a buffer, to the file. Disk file buffers are 256 characters. The disk drive has ten file buffers. Two buffers are used for a sequential file, and three for a relative file; this permits a maximum of five sequential files, or three relative files, or some combination of these, to be OPEN at a time. Fortunately, you don't need to know any of the details about buffers, only that you can only open a limited number of files.

All information moving between the disk and the computer goes through the buffers. When you write information to a file, it is accumulated in the buffer in that computer until that buffer is full; only then is the information transferred to the buffer in the disk drive and then recorded on the disk. Because data goes to the buffer, rather than to the diskette, you may type for a while without any disk activity!

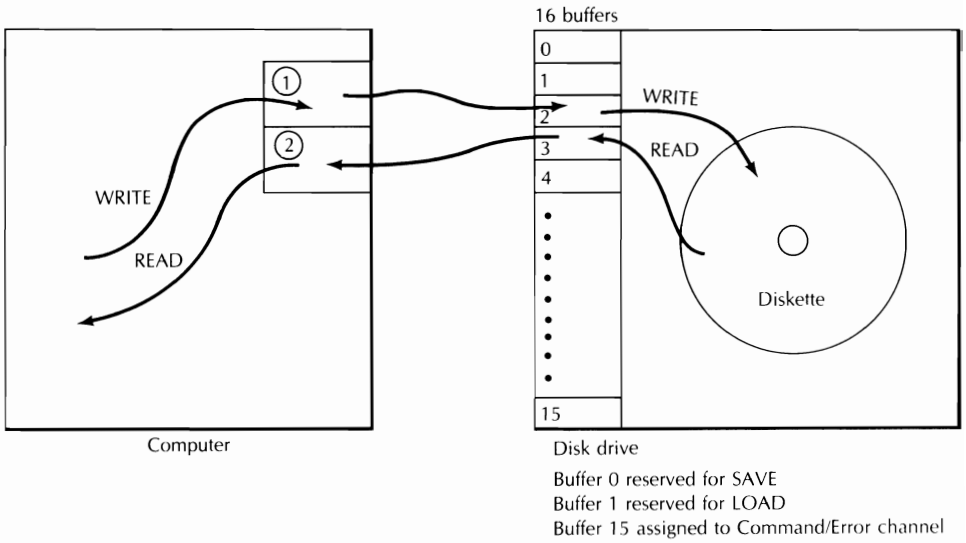


Figure 5–2 Data flow through buffers.

The same happens when reading; 256 characters are read into the buffers, and used as requested by the program. When the data in the buffer are all used, then another 256 bytes are read. The consequences of this will be apparent when the CLOSE statement is discussed.

Commodore disk drives write with verify: after writing, the data are read back and checked against the data in the buffer. While this slows disk access a little, it helps assure the integrity of your data.

INITIALIZING YOUR DISK DRIVE

Commodore uses terminology regarding disk operations that is different than other computers, and can be very confusing. One case in point is "INITIALIZing." On some computers, this means to format the disk; Commodore uses the word "NEW" ("HEADER" in BASIC 4.0). Commodore uses the word "INITIALIZE" for a very specific action—reading the Block Availability Map (BAM) into the drive controller, so the drive knows what space is free for the placement of new data. The BAM is updated every time new programs or files are stored on the disk, so that it always contains a correct map of the still-unused disk space.

We recommend you include an INITIALIZATION command as the first disk command after opening the disk command/error channel in *every* program. Why is this so important? The drive assumes that if the disk ID hasn't changed, the disk hasn't changed; it only reads the BAM when it detects a new ID. If you read from one disk, then replace it and write to another with the same ID, data will possibly be written over existing data, since the drive is using the BAM of the first disk to decide where to write on the second.

Initializing the drive forces the drive to read the BAM, so you can be sure the disk drive will write new data only to unused space on the disk. Assuming file 15 has been opened as the disk error/command channel, the initialization statement is the letter I (for Initialize) followed by the drive number (0, zero):

```
210 PRINT#15, "I0"
```

(a) Determine where the initialization statement should be placed in this sample introductory module, and write the line:

```
100 REM  SAMPLE PROGRAM
110 :
120 REM  VARIABLES USED
130 REM      N$ = NAME (20)
140 REM      D$ = BIRTHDATE (9)
150 :
160 OPEN 15, 8, 15
170 OPEN 1, 8, 8, "SAMPLE,S,W"
180 :
```

(a) `165 PRINT#15, "I0"`

While we have not used INITIALIZE in our example programs (to focus on the essential file-handling elements), you should use it in all of your programs!

OPENING FOR READ

Now let's look at the OPEN statement, and practice writing statements that include the information needed by the computer to deal with sequential data files in a program.

OPEN has the following format:

```
OPEN 2, 8, 5, "0:NAME,S,R"
```

The first number following the OPEN (2) is the *file number*. Any reference to the file later in the program will use this number. The second number specifies the *device*; 8 is the device number for disk drives. The third number (5) is called the *secondary address*; it is sometimes referred to as the *channel*. It is common to use the same number for the *file number* and the *secondary address* to minimize confusion:

```
OPEN 2, 8, 2, "0:NAME,S,R".
```

The file number can be any number from 1 to 255. However, file numbers over 127 send linefeed plus carriage return, whereas those under 127 send only carriage return. (However, see note in Appendix 4; this isn't true for BASIC 1.0 and 2.0 PETs, where *all* files send linefeeds.) Since extra linefeeds are disasters in a file, you will only use numbers 1 through 127. The device number can be any number from 8 to 15, but 8 is the default value. When a second drive is added, it may be reset to device number 9.

Commodore reserves some secondary addresses for disk operations: 0 is used for program SAVE, 1 for LOAD, and 15 for the command or error channel. Therefore the user can *only* use secondary addresses 2 through 14.

Inside quotes, the drive number is followed by a colon, then the name of the file. The S indicates a sequential file. R means open for READ; information will be READ from the disk into the computer. The commas must precede the S and R! There cannot be a space before the R (or W for a WRITE file); to avoid problems, we recommend you do not insert extra spaces in the OPEN statement. The order of the S and R may be reversed.

If the file does not exist when you OPEN for read, you will get the error message "?FILE NOT FOUND."

OPENING FOR WRITE

Here's a file opened for WRITE:

```
OPEN 6, 8, 6, "TEMPFILE,S,W"
```

This time, we used the same file and secondary address numbers. If TEMPFILE doesn't exist, the file will be created as it is open. If it does exist, you'll get a "?FILE EXISTS." Nothing will be written and your program will terminate. Later on, we'll show you how to write a new file using an existing name—but remember, you will erase *all* the old data!

```
100 INPUT "FILE NAME ";F$
110 OPEN 3,8,3, F$ + ",S,R"
```

The file name can be assigned as a string (F$). To use it, you must use the + to concatenate the file name with the other file information as one string. Note the comma *before* the S, inside the quotes, without it you will get an error message.

You can even allow the user to enter all the file information, by assigning all the parts of an OPEN statement as strings or variables:

```
100 INPUT "FILE NAME ";F$
110 INPUT "FILE NUMBER ";N
120 INPUT "MODE (R OR W) ";M$
130 OPEN N, 8, N, F$+ ",S," + M$
```

Again, note the commas inside the quotes and the + for concatenation. In a real program, you would undoubtedly include error-checking statements for all user inputs.

DISK ERROR CHANNEL

Commodore disk drives have a special channel (secondary address) which is used to communicate error and disk drive status information to the computer. The secondary address is 15, and no file name is used. Generally, the error channel is opened at the beginning of a program and closed at the end. Closing the error channel will *close all open files in the disk drive* (but not the computer), leading to an error message if a file is accessed. Therefore, *close all open files before you close the error channel!*

Here is the format for opening and reading the error channel:

```
10   OPEN 1, 8, 15 : REM OPEN CHANNEL
1010 INPUT#1, X, X$, Y, Z
1020 PRINT X; X$; Y; Z
```

The channel is opened at the beginning of the program; later, after disk access, we read the information and print it. For a list of disk error codes and messages, see Appendix 3 or refer to your Commodore manuals.

Check your understanding by writing three statements or program segments according to the following specifications:

(a) Open a file named PHONES for reading (input), assigned to file number 1.

(b) Open a file with a user-designated name for writing (output), to file number 2.

(c) Open a read (input) file with a user-designated name and file number.

(a)
```
110  OPEN 1, 8, 6, "PHONES,S,R"
```
(*Note:* the 6 could be replaced with any number between 2 and 14.)

(b)
```
140 INPUT "FILE NAME ";F$
150 OPEN 2, 8, 2, F$ + ",S,W"
```

(c)
```
140 INPUT "FILE NAME ";F$
150 INPUT "FILE # ";N
160 OPEN N, 8, 4, F$ + ",S,R"
              or
160 OPEN N, 8, N,F$ + ",S,R"
```

The advantage of the former is clear if you ever use a file number greater than 14!

THE BUFFER PROBLEM: CLOSING THE FILE

Every file that is opened with an OPEN statement must be closed with a CLOSE statement before the program finishes. The CLOSE statement *must* have a file number. As soon as the program is through using a file, and always before the program terminates, include a CLOSE statement to transfer information from the buffers to the disk, close the file, and unassign the buffers.

Once a file has been closed, the same file number can be reused for any other file you open (although there are advantages to using unique file numbers for each transaction).

Here are some CLOSE statements:

```
600 CLOSE 1
800 CLOSE 1 : CLOSE 2
```

A separate CLOSE statement is needed for each file that is opened. A file must be CLOSEd and reOPENed to change from one access mode to another—for example, after writing, before reopening the file for reading.

CLOSE is a vitally important statement. CLOSE, used properly, maintains the integrity and accuracy of your data files. Recall that when you OPEN a file, you move data into a buffer. When the buffer is full, data are moved to the disk; the file is updated. Since a buffer contains up to 256 characters, it may be only partially full when a program terminates.

What happens if the buffer is only partly full of data and there is no more data to finish filling it? You might expect the half-full buffer to simply transfer its contents to the disk for recording when the program finishes execution. But it won't do that. The data in the half-filled buffer will not necessarily be recorded into the file. Your file may not contain all the information you expected.

One important purpose of the CLOSE statement is to force the buffer to transfer its contents to the data file even though the buffer is not full. As a rule of thumb, any program with an OPEN statement should have a CLOSE statement that is always executed before the program terminates.

If you get trapped with a program that aborts (for example, with a "?SYNTAX ERROR"), or terminates and the buffers contain some data, CLOSE can be executed in direct mode, forcing transfer of the buffer contents to the disk file. Undoubtedly this will happen often while you are debugging and developing your programs, so learn how to deal with this problem! This is a good time to review ways of dealing with RETURN being pressed without data entry (see Appendix 4 and Chapter 3).

If you end up with unclosed files on your disk, they will show in the directory with a size of 0 and an asterisk preceding the program type. To remove such files from your disk, you must use VALIDATE (COLLECT in BASIC 4.0). Never SCRATCH an unclosed file. This topic is covered in more detail in Appendix 3 and later in this chapter.

To have to CLOSE files in direct mode in a finished program indicates poor programming technique and would be completely unacceptable in a work environment. Further instructions on writing your programs to include a CLOSE statement that is always executed are given later in the chapter.

(a) What are two purposes of the CLOSE statement:

_____

_____

(a) To unassign the buffer and to force the buffer to transfer its contents to the disk data file.

Under some conditions, BASIC will "flush" the computer's buffers and close the buffers in the computer. However, only a CLOSE statement will close the buffers assigned in the disk drive; reopening a file not properly closed on the disk drive will, at best, result in an error message; at worst, you'll lose that and other files, too.

ENDing or LISTing a program, or termination with an error condition ("?SYNTAX ERROR"), will *not* affect files.

The following BASIC statements will close files in the computer; NEW, RUN.

EDITing a program will close files in the computer. FILE ERROR conditions will also close files in the computer. None of these conditions close buffers in the disk drive! Remember, closing files in the computer does *not* transfer contents of the buffers to disk; to actually move data to the disk, you must CLOSE the files *in the disk drive*. Thus, if you unintentionally close files in the computer, you must

reopen them and close them properly again to close your disk buffers and transfer all data to the diskette.

Both the computer and the disk drive open buffers when a file is assigned (OPEN); both buffers must be "flushed" for your data to end up on your disk, and this requires using the CLOSE statement. If you are unsure, the easiest way to be safe is to OPEN then CLOSE the disk command/error channel by typing

```
9999 OPEN 1, 8, 15 : INPUT#1, A, B$ : PRINT B$ :CLOSE 1
RUN 9999
```

This will close any open files in both the computer and the disk drive, ensuring that your data will be written to the disk.

If the error light on your drive is on (4040, 2031) or flashing (1540, 1541), you must type the above to close your files! Of course, you could use any other line number larger than any used in your program.

If the computer tells you "?FILE OPEN ERROR" when you run the line above, run it again; the error will close the channel in the computer, so you can reopen it and close the channel in the disk drive.

If you have a PET, you can simply type

```
OPEN 1, 8, 15 : CLOSE 1
```

in direct mode. If you have loaded the DOS Wedge (also called C-64 Wedge or Universal Wedge) into either your PET or C-64, you can type < to open the error channel, read it, and close it.

(a) If you are outputting data in a program to a data file and the program accidentally terminates without executing a CLOSE statement, what should you do?

- - - - - - - - - - - - - - - - - - -

(a) Close the file with a CLOSE statement in direct mode.

To repeat: Always include a CLOSE statement that is executed before the program terminates, so that buffer flushing is automatic. You should only force buffer flushing under emergency conditions, and then you should use the CLOSE statement in direct mode. Remember that you *must* type a file number with CLOSE: CLOSE 15.

The buffer problem—and it is a real problem—makes it imperative that you *never* remove a disk from the disk drive if the disk contains an open file. Not only will you be unable to access data in the open file, but you may find data from a half-filled buffer placed in the wrong file on the wrong disk, which can create some nasty errors. See Appendix 3 or refer to your disk drive manual for a solution to the problem of open files on a disk. These are indicated by an asterisk (*) before the type designation (SEQ), and by a sector length of 0 (zero). It is a problem which can corrupt all data on a disk!

Be cautious, and remember that data go first to the buffer. They are transferred to the diskette only when the buffer(s) is full. If the buffer is not full, force it to transfer the data to the disk file with the CLOSE statement.

(b) What statement will close all open files on a disk, and should be typed in *direct mode* whenever a file program stops unexpectedly?

(b)
```
OPEN 1, 8, 15 : CLOSE 1
```

```
9999 OPEN1,8,15 : INPUT#1, X,X$ : PRINT X, X$ :
     CLOSE 1
RUN 9999
```

PLACING DATA INTO A SEQUENTIAL DATA FILE USING PRINT#

You have learned to set up communication between the computer and the disk system with the OPEN statement and CLOSE statement. Now you will learn how to place data into a file; that is, to actually record onto the disk. BASIC does this with a form of the PRINT statement, distinguished from the regular PRINT statement by using a # sign followed by the number of the file into which the data are to be recorded. PRINT# will place new data into the file. *Note:* you cannot abbreviate PRINT# by ?#; while it will look alright in a listing, it will give an error message. You must type "PRINT#". In addition, there must *not* be a space between the T and the # sign. See Appendix 2 (Keywords and Abbreviations).

```
240 PRINT#1, A
```

This statement tells the computer to print to file 1 the value assigned to the numeric variable A. The file number must always be followed by a comma.

```
270 LET X = 3
280 PRINT# X, A
```

The buffer number can be a variable, but be sure the variable is assigned a value before the print statement is executed! You can space before the file number.

Look at the next example carefully. The program prints data files starting at the very beginning of the file.

```
100 REM FILE WRITE DEMO
110 OPEN 2, 8, 2, "@0:DEMO,S,W"
120 LET A$ = "A LONG STRING"
130 LET N = 1234.56
140 PRINT#2, A$
150 PRINT#2, N
160 CLOSE 2
170 END
```

(The use of the @ in the OPEN statement allows you to run and rerun this program. It will be explained in the section REPLACING SEQUENTIAL FILES on page 222 of this chapter.)

PRINTING STRING DATA

Numeric and alphanumeric (string) data are PRINTed into a file in the same way. You want to have a carriage return character separate each data item. If you use a separate PRINT# statement for *each* variable that you place into the file, you will have no trouble later reading the data back from the file. PRINT# sends a carriage return after the last variable; if there is just one variable, it will be followed by the carriage return.

Consider the following example (an OPEN statement is assumed to occur earlier in the program from which this segment is taken):

```
220 INPUT "YOUR NAME "; N$
230 PRINT#1, N$
240 INPUT "YOUR PHONE NUMBER "; P$
250 PRINT#1, P$
260 INPUT "YOUR AGE "; A
270 PRINT#1, A
```

It is usually preferable to print an entire dataset together. Here are two ways to keep the dataset items together visually:

```
220 INPUT "YOUR NAME "; N$
230 INPUT "YOUR PHONE NUMBER "; P$
240 INPUT "YOUR AGE "; A
250 PRINT#1, N$ : PRINTT#1, P$ : PRINT#1, A
```

Note that each PRINT# statement *must* be separated by colons. Here is the other way to do it:

```
220 INPUT "YOUR NAME "; N$
230 INPUT "YOUR PHONE NUMBER "; P$
240 INPUT "YOUR AGE "; A
250 CR$ = CHR$(13)
260 PRINT#1, N$ ;CR$; P$ ;CR$; A
```

The semicolons *are not optional*—if they are not present, you may lose some of your data. For some peculiar reason, omitting the semicolons causes only the first and last variables on a line to be written to disk.

PRINT# is slightly complicated in that BASIC 1.0 and 2.0—small keyboard and 2001 series PETs—send a linefeed [CHR$(10)] with the carriage return. For your data to be readable and useful, you *must* suppress the linefeed by ending each PRINT# with a carriage return [CHR$(13)] followed by a *semicolon* (;). We have not done so in this book; if you want to write truly universal file programs, or if you use a computer with one of those versions, you must use this alternative form. See also Appendix 4 on the differences between versions of Commodore BASIC.

Here's how the above example would look for those machines:

```
220 INPUT "YOUR NAME "; N$
230 INPUT "YOUR PHONE NUMBER "; P$
240 INPUT "YOUR AGE "; A
250 CR$ = CHR$(13)
260 PRINT#1, N$ ;CR$; P$ ;CR$; A ;CR$;
```

Notice the last carriage return followed by a semicolon to suppress the unwanted linefeed character.

It seems tempting to use commas or semicolons as separators between variables in a PRINT# statement, as you would in a PRINT statement. In fact, they don't work! When we study the INPUT# statement, some of the reason for these distinctions will become clear. If in doubt, place each variable in a *separate* PRINT# statement!

(a) Write a program segment that will READ all of the strings from line 920 and then output the strings to file buffer 2.

```
240 _____________________________________________________
250 _____________________________________________________
920 DATA DISKETTES, PAPER CLIPS, ENVELOPES
```

(a)
```
240 READ N1$, N2$, N3$
250 PRINT#2, N1$ ;CR$; N2$ ;CR$; N3$
```

or

```
250 PRINT#2, N1$ : PRINT#2, N2$, : PRINT#2, N3$
```

PUNCTUATION IN STRINGS

The hard-to-remember part of printing strings to a file with PRINT# is when you want your strings to include commas, colons, or semicolons, or to purposely include carriage returns. For example, given

```
LET B$ = "PUBLIC, JOHN Q."
```

you would expect the statement

```
PRINT#1, B$
```

executed after the B$ assignment statement to take the entire string enclosed by quotation marks and place it into the file as one string. But the quotation marks are essentially ignored. The comma actually separates PUBLIC from JOHN Q. in the file, breaking the one name into two separate data items. With the PRINT# statement, the solution is to "force" quotation marks on either side of the entire name string by using the CHR$( ) function. CHR$(34) is the ASCII code for the quote (") symbol. Notice how it is used in this program segment:

```
240  LET B$ = "PUBLIC, JOHN Q."
250  PRINT#1, CHR$(34) B$ CHR$(34)
```

As will be discussed in the INPUT# section to follow, there are still problems with this solution. You only need to worry about forcing quotation marks in a PRINT# statement when your string will also include commas, colons, semicolons, or carriage returns. That should not happen very often, and with careful planning it can be avoided entirely.

The following two programs are brief samples to illustrate how numeric and string data are written to a file. You can type them in and run them, then use one of the utility file reader programs given in the next section to read back the data you have written to the disk. If you have time and are curious, you can experiment with different formats in the PRINT# statement, such as forced commas between data items, presence and absence of semicolons around CR$, etc. We have given the version of each program using separate PRINT# statements for each variable, but included the alternate versions using carriage returns as well.

```
100 REM FILE WRITE DEMO #1-"DEMOPROG"
110 :
120 OPEN 1, 8, 2, "@0:DEMO1,S,W"
130 :
180 LET CR$ = CHR$(13)
190 :
200 REM       READ DATA & PRINT TO FILE
210 :
220 READ A, B, C
230 IF A = -1 THEN 990 : REM ** CHECK FOR END-O
     F-DATA FLAG
240 PRINT#1, A :PRINT#1, B :PRINT#1, C
250 GOTO 220
260 :
900 REM      DATA FOR DEMO
910 DATA 23, 26, 18, 19, 22, 20
920 :
930 REM      END OF DATA FLAG
940 DATA -1, -1, -1
950 :
980 REM      CLOSE FILE
990 CLOSE 1
```

or

```
240 PRINT#1, A ;CR$; B ;CR$; C
```

```
100 REM   FILE WRITE DEMO FOR STRING DATA (STRG
     DEMO)
110 :
120 OPEN 2, 8, 2, "STRGDEMO,S,W"
130 :
200 REM   READ DATA AND WRITE TO FILE
210 :
220 READ A$, B$, C$
230 IF  A$ = "END" THEN 990 : REM *** CHECK FOR
      END-OF-DATA FLAG
240 PRINT#2, A$ : PRINT#2, B$ : PRINT#2, C$
250 GOTO 220
260 :
900 REM   DATA FOR DEMO
910 DATA THUMB TACKS, PENCILS
920 DATA BATTERIES, FELT TIP PENS
```

```
930 DATA TAPE, MANILA FOLDERS
940 :
950 REM   END OF DATA FLAG
960 DATA END, END, END
970 :
980 REM   CLOSE FILE
990 CLOSE 2
```

or

```
125 LET CR$ =  CHR$(13) : REM CARRIAGE RETURN
240 PRINT#2, A$ ;CR$; B$ ;CR$; C$
```

(The use of the @ in the OPEN statement allows you to run and rerun this program. It will be explained in the section REPLACING SEQUENTIAL FILES on page 222 of this chapter.)

Now it is your turn to practice writing statements to place data in a file. In each of the four program segments below, write a PRINT# statement appropriate for filing the data from that program segment's INPUT statements. Use three as the file number.

(a)
```
300 INPUT "HOW MANY SAMPLES "; S
310 INPUT "HOW MANY WERE GREEN "; G

320 PRINT# _____
```

(b)
```
500 INPUT "TODAY'S DATE "; D$
510 INPUT "CITY NAME "; C$
520 INPUT "STATE CODE "; S$

530 PRINT# _____
```

(c)
```
900 INPUT "TITLE OF BOOK "; T$
910 INPUT "FIRST LINE OF TEXT "; F$
920 INPUT "NUMBER OF PAGES "; P

930 PRINT# _____
```

(a)
```
320 PRINT#3, S : PRINT#3, G
```
or
```
320 PRINT#3, S ;CHR$(13); G
```
or
```
320 PRINT#3, S
330 PRINT#3, G
```

(Either method can be used. Only the first is shown for the next two answers.)

(b)
```
530 PRINT#3, D$ : PRINT#3, C$ : PRINT#3, S$
```

(c)
```
930 PRINT#3, T$ : PRINT#3, F$ : PRINT#3, P
```

You may occasionally see a program using the following format:

```
PRINT#1, A "," B "," C
```

However, there are so many problems with this format that we recommend you forget you've ever seen this! Methods of separating variables other than the two shown in this book *will not work* consistently, and are full of problems.

WRITING A FILE

So far, we have concentrated on the formats for the OPEN and PRINT# statements for creating and writing to a sequential file.

Now let's move a step closer to the real world with a set of two programs in BASIC: one to output or write to a file, and a second, separate BASIC program to read back (input) data from the file to the computer's electronic memory and then display the data.

As noted earlier, using files requires planning. Your plan should consider:

1. What to include in each dataset.
2. How large each data item or dataset will be.
3. Whether technical points, such as embedded commas in strings, must be handled with special techniques.
4. How to test each data item in the dataset as completely as possible for accuracy and validity.

With these considerations in mind, here is a program to help you place a simple inventory from your home or business into a disk file. The introductory module and possible checks for data validity are included. The file name is PROPERTY.

For simplicity in the example program, we have arbitrarily limited the dataset to three items of data. Other information, such as brand name, model number, and serial number are other possible data for a file like this.

The following program is not complete. Read it, then do the following exercises to fill in the missing statements.

Location 50003 contains a 0 in the Commodore 64, and a 1 or 160 for most PETs (it is also 0 for BASIC 1.0 PETs). The first few lines use this value to set the lowercase; see Appendix 4 for an explanation.

```
100 rem   property inventory file load program
101 if peek(50003) <> 0 then poke 59468,14 : re
     m lower case for pets
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
```

```
 120 rem   variables used
 130 rem     d$ = description (20)
 140 rem     n = number of items
 150 rem     v = dollar value
 160 rem     cr$ = carriage return
 170 rem   files used
 180 rem     property = sequential file
 190 :
 200 rem   initialize
 205 cr$ = chr$(13)
 210 open 1, 8, 8, "@0:property,s,w"
 220 :
 230 rem   data entry routines
 240 print : input "Item description "; d$ : let
       d = len(d$)
 250 if d > 20 then print "Abbrev. to less than
      20 characters." : goto 240
 260 if d = 0 then print "Please enter a descrip
      tion." : goto 240
 270 print : input "Number of items "; n$ : let
      n = val(n$)
 280 if n = 0 and n$ <> "0" then print "Enter a
      number." : goto270
 290 if n <> int(n) then print  "Enter whole num
      bers only." : goto 270
 300 if n <= 0 then print "There must be some un
      its; enter number" : goto 270
 310 print : input "What is the dollar value of
      each "; v$ : let v = val(v$)
 320 if v = 0 and v$ <> "0" then print "Enter a
      number" : goto310
 330 if v <= 0 then 400
 340 print#1, d$ ;cr$; n ;cr$; v
 350 :
 360 print "Writing: "d$; n; v
 370 print
 380 goto 240
 390 :
 400 print : print "Did you really mean a zero v
      alue ?"
 410 input "(yes or no) "; r$
 420 let r$ = left$(r$,1)
 430 if r$ = "n" or r$ = "N" then print "Reenter
       correct value." : goto 310
 440 if r$ = "y" or r$ = "Y" then 340
 450 print "Please enter 'Y' for yes or 'N' for
      no." : print : goto 400
 460 :
 470 rem   file close routine
 480 close 1
ready.
```

(The use of the @ in the OPEN statement allows you to run and rerun this program. It will be explained in the section REPLACING SEQUENTIAL FILES on page 222 of this chapter.)

(a) The above program has one small but important "bug." Find and describe the error.

(a) The program never executes the file closing routine at line 400; the CLOSE statement is needed to assure flushing the last data items from the buffer to the file on the diskette.

The problem of how to indicate to the program when to close the file is part of preplanning. The program should include a way for the user to indicate to the computer that the user is done with the program for now, or that all data have been entered. Either of the two procedures shown below could be included in the previous program for this purpose. The choice is yours, but we prefer the second.

```
100 :
110 :
238 print "Type 'STOP' if finished"
245 if d$ = "stop" or d$ = "STOP" then 410
246 :


371 :
373 print
374 print "Is there more data to enter ?"
375 input "(yes or no) "; r$
376 let r$ = left$(r$,1)
377 if r$ = "n" or r$ = "N" then 510
378 if r$ = "y" or r$ = "Y" then print"[clr]" :
      goto 240 : rem clear screen
379 print "Enter 'Y' for yes or 'N' for no." :
     goto 373
```

Here's another version of the second procedure, using GET instead of INPUT. As mentioned earlier, there are advantages to using GET when the response is a single keystroke.

```
372 print
373 print "Is there more data to enter ?"
374 print "Enter 'Y' for YES or 'N' for NO."
375 get r$ : if r$ = "" then 375
376 if r$ = "n" or r$ = "N" then 510
377 if r$ <> "y" and r$ <> "Y" then 375
379 print "[clr]" : goto 240 : rem clear screen
```

This procedure works for terminating a program and closing files containing discrete datasets, as has been described.

When you write file programs (or any program), prepare some written documentation for yourself and other users. Even you may have trouble seeing how the program works without some time and effort, six months down the line. At the least, some description of the file format is needed. A good habit is to include such information in REMark statements in the program itself.

When you OPEN and write to an existing file, you must use "write with replace" or no data will be written to that file, or you must erase (SCRATCH) the file before you OPEN to write the new data. (See the section REPLACING SEQUENTIAL FILES)

(a) In a program, why do you need to check that all data to be included in the data file have been entered?

(a) So that a CLOSE statement can be executed.

READING DATA FROM A FILE

Now that you can write to a file, let's learn how to read data from an existing file. To do this, you must know what data were place in the file, and how they were arranged. After that, reading from the file is straightforward.

To read from a file, you first OPEN the file for READ, then use the INPUT# statement:

```
350 OPEN 2, 8, 2, "0:TEMPFILE"
360 INPUT#2, A, B$, C
```

or

```
350 OPEN 2, 8, 2, "TEMPFILE"
360 INPUT#2, A, B$, C
```

Notice the use of commas in line 360, one after the file number and others to separate the variables, just like an INPUT statement. Line 360 will read three data items from the diskette file and assign them to variables A, B$, and C.

You must have the correct type of variable (numeric or string) in the INPUT# statement to match the data that are being read from the data file. If you try to read string data into a numeric variable, your program will terminate with "?FILE DATA ERROR."

To avoid such problems, be sure you know how the data were initially placed into the file, whether string or numeric, and in what order. To be safe, you may wish to read all data as strings, and do the conversion to numeric data in the program, where you can control the display of messages.

Here's an example of this situation:

```
100 OPEN 3, 8, 3, "TEMPFILE"
  .
  .
  .
170 PRINT#3, A ;CHR$(13); B$ ;CHR$(13); C
```

The data stored by line 170 can ONLY be read by the following sequence of variables:

```
400 OPEN 4, 8, 4, "TEMPFILE"
410 INPUT#4, A, B$, C
```

A statement like INPUT#2, A$, B, C will result in "?FILE DATA ERROR."

Of course, as long as you have the correct sequence of variable *types* (in this example, numeric, string numeric) you could read the data with any combination of INPUT# statements:

```
410 INPUT#4, A
420 INPUT#4, B$, C
```

or

```
410 INPUT#4, A
420 INPUT#4, B$
430 INPUT#4, C
```

The INPUT# statement with multiple variables works just like the corresponding INPUT statement: successive values must be separated by commas or carriage returns. Separating file data with commas is fraught with problems, so you will usually only deal with carriage returns as separators.

Since INPUT (and INPUT#) only reads to a comma, colon, semicolon, or RETURN in data, any string data including one of those characters *cannot* be read by INPUT#—you must use GET# instead (see sample file-reading programs). INPUT# will ignore the remainder of the string after the offending punctuation, just as INPUT ignores "DAVE" if you type "BARRY, DAVE".

Remember our insistence on printing each value followed by a carriage return? It is just like our earlier recommendation that each INPUT statement be

used to get one, and just one value or string! If you place each variable in a separate PRINT# statement, or at least separate them with carriage returns, you won't have to worry about data formats and lost or unreadable data.

We'll repeat that: Use a separate PRINT# statement to write each variable; then INPUT# can take whatever format is convenient.

Going back to the simple inventory program described earlier in this chapter, recall that the alphanumeric description (D$), followed by quantity (N), followed by dollar value (V) were placed into the file in that order. The variable names D$, N, and V were used in the program when the data were printed to the file. The variable names themselves were separate from the data items. Therefore, you can use any appropriate string or numeric variable name in INPUT # statements when data are read out of the file into the buffer.

(a) Which of the following statements is appropriate to input data from the PROPERTY data file?

(1)
```
270 INPUT #1, A, B, C
```

(2)
```
270 #1, A$, B, C
```

(3)
```
270 INPUT #1, A, B$, C$
```

(a) Statement (2).

The program below is the companion to the Inventory file creation program. Again, lines 200, 270, and 320 are for you to fill in following the program.

Here's the complete Property file-reading program, followed by a sample run.

```
100 rem   read data from property file
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem     n$ = description (20)
140 rem     q = quantity
150 rem     d = dollar value
160 rem     r$ = user response
170 rem   files used
180 rem     property, seq.;  dataset format: n$
     ,q,d
190 :
199 :
200 open 3, 8, 3, "property,s,r"
210 :
220 rem   print headings
230 print"[clr]" : rem clear screen
240 print "Description" tab(22) "Quan." tab(30)
      "Value Ea."
250 :
260 rem   file read/report routine
270 input#3, n$, q, d
280 print n$ tab(22) q tab(30) d
290 if  st = 0 then  270 : rem get next
300 :
310 rem   close file
320 close 3
```

```
run

Description          Quan.    Value Ea.
Television           2          500
Cassette Recorder    2          125
Video Recorder       1          750
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
Speakers             2          150
break in 280
ready.
```

This RUN was terminated by a hasty press of the RUN/STOP key, an obvious error condition, leaving an open disk file. What if you wanted to do more with the data, and wanted the program to continue beyond reading and printing the data? A technique exists that allows the program to read to the end of the file and continue properly.

Just as with regular READ and DATA statements in BASIC, the data file uses a pointer to point "at" the next data item available in the buffer holding data from the disk file. When a file is opened for input, the pointer is positioned at the beginning of the file and points at the first data item. Each execution of INPUT# pushes the pointer forward as many places as there are variables in the statement's variable list.

`PRINT#1, A$`	moves the pointer one position to the second data item position.
`INPUT#1, N, N$`	moves the pointer past data items 1 and 2 to item 3. The pointer is always looking at the next position.
`INPUT#1, W, X, Y, Z$`	moves the pointer four places so the next item read will be at position 5.
`PRINT$1, A ;CR$; B$`	moves the pointer ahead two places, so the next data item added by a PRINT# statement will be at position 3.

DETECTING THE END OF A FILE: STATUS

When your program uses PRINT# to add data to a file, each PRINT# statement moves an internal pointer ahead. The computer uses this pointer to keep track of where it is within the buffer. When all data have been entered, the end of file marker is located just past the last data item. The end of file marker is automatically put in place by the computer when the file is closed.

When you INPUT# data from the file, the file pointer is always looking at the next data item available in the file (or in the buffer, to be more exact). An attempt to read past the end of the file results in a change in the status variable (ST), which can be used to signal the end of the file. Commodore gives the following in its manuals:

```
IF (ST AND 64) = 64 THEN CLOSE 1 : END
```

The value of ST is set to 64 at the end of the file. The AND allows us to see the end of file value even if other errors are being signaled by ST. It is actually preferable to simply check that ST is nonzero to signal the end of a file, because ST is set to other values by a variety of problems, all of which should terminate the reading

of data. If you remove the disk, the version given above will forever take the last datum from the buffer, since ST will be 2, not 64. While this is unlikely, it argues for a simpler version:

```
IF ST <> 0 THEN CLOSE 1 :END
```

In most sample programs in this book, you'll see the version:

```
750 IF ST = 0 THEN 300
760 CLOSE 1 : END
```

In this case, the program will loop back to read more data until the end of file marker sets ST to 64; then it will continue with the next line.

We'll use this to modify the previous program so that it does not terminate with an end of file error. Modify line 290 as shown below:

```
290  IF ST = 0 THEN 270
```

ST is affected by *any* input/output operations from the computer. This includes GET and INPUT as well as more obvious input/output operations. If you follow a READ by printing to the printer (with a PRINT#), the ST will reflect the PRINT# operation, not the READ. In that case, you must store the value of ST in another variable. Assuming file 4 has been opened to the printer with an OPEN 4, 4:

```
330 INPUT#1, A$
340 SV = ST : REM STORE ST FROM READ
350 PRINT#4, A$ : REM TO PRINTER
360 IF SV = 0 THEN 330
```

If you left out line 340, and changed line 360 to "IF ST . . .", the program would never stop, because ST will always be 0 following the PRINT# to the printer. This once cost me several dozen sheets of paper, so be warned!

Reference manuals for various computer systems show considerable variation in explanation of how the pointer and end of file marker work. However, the ideas expressed here are the same for any system; the keywords and programming statements will differ.

(a) What does ST stand for? _____

(b) What does the value of 64 for ST indicate?

_____

(c) Why is it necessary to store ST in another variable when reading from disk then printing to a printer?

_____

_____

(a) STATUS
(b) End of file.
(c) Any input or output operation resets the value of ST.

FILE-READING UTILITY PROGRAMS

When you start writing files, it is handy to have a utility program to read back your files for you, so you can see how the data look. This is particularly useful when you start getting *file data errors*. Here are two short programs which will read and display files. Try these on your file data programs. You'll find the GET version particularly useful if you want to experiment with the way data are actually written to the disk by PRINT#, or if you start getting "?STRING TOO LONG" errors.

The first program uses INPUT# to read the data. The disadvantage to this program is the usual disadvantage of INPUT#: if you have written data to the file with something other than carriage returns separating the data items, this program will not read all of your data. However, it is fast and simple; when disaster strikes, it can be typed in at the keyboard if necessary.

```
20 input "File name"; f$
500 open1, 8, 8, f$ + ",s,r"
510 input#1, n$ : print n$
520 if st = 0 then 510
530 close1 : end
```

This version uses GET# to read a file. The advantage of GET# is that it is not subject to limitations of punctuation or character number. A file containing commas, colons, or semicolons may be read with this routine, allowing you to examine how the data were written and to fix the problem. Line 130 checks for a carriage return at the end of the data item.

```
100 rem   read any file with get
110 input "File name "; f$
120 open1, 8, 8, f$ + ",s,r"
125 s$ = "" : rem initialize to null
130 get#1, c$
140 if c$ <> chr$(13) and st = 0 then s$ = s$ +
      c$ : goto 130
150 print s$
160 if st = 0 then 125
170 close 1
180 print "File read and closed."
```

This particular version concatenates each character read until it encounters a carriage return, then it prints the concatenated string and starts building the next string. This allows you to see commas, spaces, and other punctuation which would not show up if read by an INPUT# statement. However, it will also die from "?STRING TOO LONG" problems.

Here's another version which simply prints each character as it is read without concatenating it into a string. (Of course, you could avoid the "?STRING TOO LONG" problem by revising the previous program to print S$ whenever it had more than 255 characters, then reset it to an empty string). While the preceeding program could actually be used to read file data in a program, this one only will display data. It prints the PETASCII value of each character, so you can even see invisible characters. Note the "+ chr$(0)" in the ASC function—it's necessary in case the GET picks up a null character (""), which will return an error message. Also added is "Press RETURN to continue" to let you step through the file. STatus has been saved in the variable SV, because the GET in line 180 is seen as an input/output operation, and resets the status to 0. (Read that section on STATUS very carefully—it's a tricky beast!)

```
100 rem   read any file with get
110 input "File name "; f$
120 open1, 8, 8, f$ + ",s,r"
130 get#1, c$  : let sv = st
140 print asc(c$ + chr$(0));
150 if c$ <> chr$(13) and sv = 0 then 130
160 print
170 print "[down]Press 'RETURN' to continue."
180 get z$ : if z$ <> chr$(13) then 180
190 if sv = 0 then 130
200 close 1
210 print "File read and closed."
```

PERMANENTLY REMOVING FILES FROM DISKS

Situations will arise when you will want to erase a file from a disk. It may be a temporary file, such as those created for demonstration or testing programs in this book, or a file that is of no further use to you for other reasons. The SCRATCH command can be used to delete files as well as programs from a disk. Using this command deletes the file from the disk, destroying the file's contents by deleting all references to the file from the disk directory. SCRATCH can be executed in direct mode, like RUN or LIST. It can also be used within a program, but we discourage this use except for very temporary files. Here is the form:

```
OPEN 1, 8, 15, "S0:TEMPFILE" : CLOSE 1
```

SCRATCH is abbreviated as "S". The 0 refers to drive 0; on dual drive systems, you can replace it with 1 to SCRATCH files on drive 1. On single drive systems (1540, 1541), the 0 may be omitted if the program name starts with a letter.

Within a program, SCRATCH can be used by PRINTing to the disk command channel. Assuming there has been an OPEN 15, 8, 15 earlier,

```
450 PRINT#15, "S0:TEMPFILE"
```

would carry out the same action as the direct command above.

Use SCRATCH very carefully, as the action is irreversible. Once a file has been SCRATCHed, there is no going back. Accidentally destroying the wrong file, especially if you haven't made a backup copy, can mean that you wasted hours or days entering data into a file. Think carefully before SCRATCHing!

Be sure you understand the difference between SCRATCH and CLOSE. CLOSE merely disassociates a buffer from the file and flushes the buffer contents onto the disk if you are writing data. After a CLOSE statement, the data file is still recorded on the disk. SCRATCH eliminates the file from the disk by erasing all reference to it in the directory.

After a file has been SCRATCHed, the space or sectors on the disk that the file occupied can be reused. A file created later can (and probably will) overwrite, that is, re-record, the same sectors that were occupied by the SCRATCHed file.

(a) Write a statement to scratch (erase) the file "TEMPADDRESS" on drive 0.

(a) `OPEN 1, 8, 15, "S0:TEMPADDRESS" : CLOSE 1`

You could use any number from 1 to 255 for the file number, but you must use 8 for the device and 15 for the channel.

REMOVING OPEN (*) FILES FROM A DISK

Occasionally, a program will terminate unexpectedly, leaving an open file on a disk. When you look at the DIRECTORY of a disk, you can tell open files by two indicators: the number of blocks (file size) is zero (0), and there will be an asterisk (*) before the program type. *Do not ever SCRATCH* one of these files; to do so will eventually disasterously scramble data on your disk! To remove these files, you must VALIDATE your disk. The VALIDATE command, like other disk commands, requires OPENing a file to the disk error channel:

```
OPEN 15, 8, 15, "V0" : CLOSE 15
```

VALIDATE takes about two minutes. It reads through each file in the directory and recreates the Block Availability Map (BAM) indicating which sectors on the disk are free and which are taken. After a VALIDATE, you will notice any * files have disappeared from the directory.

(a) Tell which file in this directory is open.

```
0     SAMPLE DISK     SD 2A
15  "ADDRESSES"       SEQ
7   "PHONE BOOK"      SEQ
0   "PHONE BOK"      *SEQ
1   "PHONE BOKE"      SEQ
```

(b) Write the statement to remove the offending file from the disk: _____

(a) PHONE BOK—the size is zero, and there is an asterisk (*) before the file type.

(b) `OPEN 1, 8, 15, "V0" : CLOSE 1`

REPLACING SEQUENTIAL FILES

Unlike most other Disk Operating Systems (DOSs), Commodore disk files are protected against accidental rewriting. On some computer systems, this is called "locking" files. If you try to run a program creating a sequential file a second time, the program will stop with the message "?FILE EXISTS." How can you set up the program to create a sequential file with new data under an existing name?

There are two methods. The first is to use the SCRATCH command within the program to SCRATCH any existing file with the name of your sequential file. Then you will be writing your file with a name which is not on the disk, and there is no problem. However, if you forget to SCRATCH the old file first, perhaps while you are modifying your program, you will be unable to save your data. There is another way.

Commodore has a built-in REPLACE function in its DOS. To replace a sequential file named "ADDRESS" you would use the following format:

```
OPEN 2, 8, 2, "@0:ADDRESS,S,W"
```

The only difference from a normal OPEN command is in the at sign (@) in front of the drive number.

There is an obvious problem with wholesale use of the REPLACE feature. If you have the wrong disk in the drive when you run your program, you may destroy an important file. However, with some caution, you can use it for all of your OPENs when writing. Do *not* use the replace function in an OPEN for READ!

In all of the following examples, files for writing will be OPENed with the replacement function, allowing you to run them as often as you wish. Use of the REPLACE function is handy during debugging and testing; you probably don't want the REPLACE function in finished programs.

However, remember that using the REPLACE function will *destroy* any previous file data.

(a) Write an OPEN statement using file 5 to replace the data in your "GROCERY" file. _____

_____

(a)
```
OPEN 5, 8, 5, "@0:GROCERY,S,W"
```

MULTIPLE FILE OPERATIONS IN ONE PROGRAM

We have used the word "copy" to describe how the INPUT# statement works when data are transferred from the disk data file into the computer's memory. "Copy" implies that the data in the file do not change when they are input into the part of the computer's electronic memory designated as the buffer.

You can think of the process described above as like listening to a cassette tape recording of music: playing the tape doesn't erase it, and it can be played back again and again. The data in a file are similarly unaffected and unchanged by "playing back" the data into the computer, and the data remain in the file for another use. In BASIC, the only way to change data in a sequential file is with a PRINT# statement.

You can WRITE data to a file, then READ them back from the same file in the same program. But it is crucial that you CLOSE the file after writing (recording information into the file), before you can reOPEN the file to READ the data (copy it into the computer's memory). You must OPEN . . . ",S,W" for writing, CLOSE the file, then OPEN . . . ",S,R" for READing.

The program you will see next illustrates the procedure to open and close files at appropriate times. Quality assurance (QA) data from a manufacturing process are entered into a file. The data are a long list of numbers that represent, let's say, the clarity of glass from various meltings of recycled bottles, using a scale of 1 to 6.

After the data are all entered and the file is closed, the program will read the QA values from the file, accumulate the number of occurrences of each catagory of clarity (1 to 6) in an array, and then display a summary of a data. The program

is self-documented by REMark statements. Use QCONTROL as the name for the file, as in the sample RUN that follows:

```
run
file name?qcontrol
enter numbers 1-6 only.  enter 99 to stop.
qa number:? 1
qa number:? 1
qa number:? 4
qa number:? 6
qa number:? 2
qa number:? 3
qa number:? 5
qa number:? 4
qa number:? 2
qa number:? 1
qa number:? 2
qa number:? 3
qa number:? 42
use numbers 1-6 only!

qa number:? 4
qa number:? 2
qa number:? 5
qa number:? 5
qa number:? 6
qa number:? 99

results of quality control data

qa number quantity
1          3
2          4
3          2
4          3
5          3
6          2
ready.
```

```
140 rem   file input/output demo
150 :
160 rem   user enters quality control results f
     or file
170 rem   program prepares summary report from
     file
180 :
190 rem   variables used
```

```
200 rem      f$ = file name
210 rem      n% = qa measure
220 rem      v = qa measure
230 rem      c() = counting array
240 rem    seq file = qcontrol (user entered)
250 rem    dataset format: n%
260 :
270 rem    initialize array (not necessary in co
     mmodore basic)
280 for x = 1 to 6 : let c(x) = 0 : next x : re
     m basic zeros all variables
290 :
300 rem    initialize file
310 input "file name "; f$
320 open 1, 8, 8, f$ + ",s,w"
330 :
340 rem    data entry routine
350 print "Enter numbers 1 to 6 only.  Enter 99
      to stop."
360 :
370 input "QA Number "; n%
380 if n% = 99 then 430
390 if n% < 1 or n% > 6 then print "Use numbers
      1 to 6 only!" : goto 370
400 print#1, n%
410 goto 370
420 :
430 rem file close
440 close 1
450 :
460 rem    open file to read
470 open1, 8, 8, f$ + ",s,r"
480 :
490 rem    read file and put data in array
500 input#1, v
510 let c(v) = c(v) + 1
520 if st  = 0 then 500
530 :
540 rem     close file routine
550 close 1
560 :
570 rem print report from array
580 print "[clr]" : rem clear screen
590 print "Results of Quality Control Data"
600 print
610 print "QA Number", "Quantity"
620 for v = 1 to 6
630 print v, c(v)
640 next v
650 :
```

Refer to the previous program to answer the following questions:

(a) Through which statement does the computer obtain the name of the data file? _____

(b) Which statement checks the parameters for the quality control numbers?

_____

(c) How does the computer know that all data have been entered? _____

_____

(d) Why are two CLOSE 1 statements used in the same program? _____

_____

(e) What does line 520 do? _____

_____

(f) In line 510, how many different values can V have? _____

(a) Line 310.
(b) Line 390.
(c) User enters 99 as input value.
(d) The data file must be closed after output *and* after input.
(e) Checks each file input data item to see if it is the end of file marker.
(f) Six (1 to 6).

Help us write another program that first creates a data file call DEMO2 and then displays the contents of that data file. Complete lines 200, 240, 270, 310, 330, 360, and 380. (Read the REMs and comments.)

```
100 REM  DATA FILE DEMO: CREATE, READ, DISPLAY
     FILE
110 REM
120 REM  VARIABLES USED:
130 REM    D$ = DATA ITEM WRITE VAR.
140 REM    R$ = DATA ITEM READ VAR.
150 REM    X  = FOR-NEXT LOOP VAR.
160 REM
170 REM  FILE NAME: DEMO2
180 REM  FILE DATA FORMAT: ONE STRING DATA ITEM

190 REM
200 _____ : REM OPEN
     THE FILE FOR WRITE
210 REM ** USING A FOR-NEXT LOOP, PLACE 8 STRIN
     GS (DATA ITEMS) IN THE DATAFILE
220 FOR X = 1 TO 8
230 LET D$ = "TEST" + STR$(X)
```

```
240 ________________________________ : REM PRINT
     TO THE FILE
250 NEXT X
260 REM ** NOW CLOSE THE FILE
270 ________________________________
280 REM ** A PRINT STATEMENT TELLS US SO FAR, S
    O GOOD...
290 PRINT "FILE WRITTEN AND CLOSED."
300 REM ** RE-OPEN FILE FOR INPUT; CAN USE A DI
    FFERENT FILE NUMBER
310 ________________________________
320 REM ** NOW READ FROM FILE AND PRINT DATA IT
    EMS
330 ________________________________
340 PRINT R$
350 REM ** USE 'ST' TO CHECK FOR END OF FILE
360 ________________________________
370 GOTO 340
380 ________________________________
390 PRINT "FILE CLOSED."
```

```
200 OPEN 1, 8, 8, "@0:DEMO2,S,W"
240 PRINT#1, D$
250 :
270 CLOSE 1
280 :
310 OPEN 2, 8, 8, "DEMO2,S,R"
320 :
340 INPUT#2, R$
350 :
360 IF ST <> 0 THEN 380
370 :
380 CLOSE 2
```

(a) Now show everything that will be printed or displayed when this program is RUN. _____

(a)
```
RUN
FILE WRITTEN AND CLOSED
TEST 1
TEST 2
TEST 3
TEST 4
TEST 5
TEST 6
TEST 7
TEST 8
FILE CLOSED
READY.
```

One unique feature of file programs is that sometimes nothing appears to be happening when the program is RUN. There may be no printed report or anything happening on the screen other than RUN and READY. To the novice, this seeming lack of activity may be alarming. Of course, you will see the light on the disk drive go on and off as files are opened and closed, and hear the drive spin as data are written.

To minimize the feeling that "nothing is happening," we recommend that you print appropriate messages on the screen as the program runs, to inform the operator about what is happening. For example:

```
210  PRINT "WRITING DATA TO FILE."
  .
  .
  .
780  PRINT "FINISHED WRITING, FILE CLOSED."
```

Again, beware of reopening (or replacing) an existing sequential data file for WRITE. You will destroy all the data in that file; data cannot be recovered easily, if at all. Erased disk files are gone. Having made that tragic mistake ourselves, we feel obligated to keep warning you.

(a) Which statements in the previous program help assure the user that "invisible" data file activity has taken place?

(a) Lines 290 and 390.

As the last activity in Chapter 3 before the Self-Test, you (should have) constructed a program for *entering* names and addresses, and *displaying* the entered data for corrections. Now you can add the statements to *create a file* of names and addresses.

If you SAVEd the program, LOAD it and LIST it now for reference, or turn back to the end of Chapter 3 and enter it. Then complete line 250 to create the ADDRESS file.

(a)
```
200 REM   SEQUENTIAL FILE NAME: ADDRESS
210 REM   DATASET FORMAT: D$
220 :
230 REM   INITIALIZE
240 :
250 _____________________________________________________
260 :
300 REM  MAIN DATA ENTRY MODULE AND ERROR TESTS
```

(a)
```
250 OPEN 1, 8, 4, "@0:ADDRESS,S,W"
```

Only two more file-handling statements are needed for a functioning program to create a name and address file. Complete line 540 to transfer the concatenated dataset to the file, and take a wild guess at what is needed at line 590.

```
490 if r = 6 then 520
500 on r gosub 620, 670, 720, 770, 810 : goto 3
    70
510 :
520 let d$ = n$ + a$ + c$ + s$ + z$
530 :
540 _____________________________________ : rem pr
    int to file
550 :
560 print : print "Do you have another name to
    enter (Press 'Y' or 'N') ?"
570 get r$ : if  r$ = "" then 570
580 if r$ = "y" or r$ = "Y" then 310
585 if r$ <> "n" and r$ <> "N" then 570
590 _____________________________________ : rem cl
    ose file
600 print "File closed."
610 end
```

```
540  print#1, d$
590  close 1
```

Remember, if you RUN this program a second time, all the data entered during the first RUN will be erased. Adding data to a file and changing existing data in a sequential file are topics covered in the next chapter.

The complete program appears on the next pages.

```
100 rem   address file creating program
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
110 :
120 rem  variables used
130 rem  n$ = name (20 char. max.)
140 rem  a$ = address (20 char. max.)
150 rem  c$ = city (20 char. max.)
160 rem  s$ = state (2 char. )
170 rem  z$ = zip code (5 char. )
180 rem  d$ = entire dataset concatenated (67 c
    har.)
190 :
200 rem  seq. file name = address
210 rem  dataset format = d$
220 :
```

```
230 rem  initialize
240 :
250 open 1, 8, 4, "@0:address,s,w"
260 :
300 rem  main data entry & error tests
310 print "[clr]" : rem clear screen
320 gosub 620
330 gosub 670
340 gosub 720
350 gosub 770
360 gosub 810
370 rem display data for corrections
380 print "[clr]" : print "You have entered the
      following data:" : print
390 print "- 1 - "; n$
400 print "- 2 - "; a$
410 print "- 3 - "; c$
420 print "- 4 - "; s$
430 print "- 5 - "; z$
440 print "- 6 - no changes"
450 print : print "Check for errors. Enter numb
     er of item"
460 print "that needs correcting. If no changes
      are needed, enter number 6."
470 print : input "Enter your selection (1 to 6
     ) "; r$
475 r = val(r$)
480 if r < 1 or r > 6 then print "Enter a numbe
     r from 1 to 6." : goto 450
490 if r = 6 then 520
500 on r gosub 620, 670, 720, 770, 810 : goto 3
     70
510 :
520 let d$ = n$ + a$ + c$ + s$ + z$
530 :
540 print#1, d$ : print "Writing " d$
550 :
560 print : print "Do you have another name to
     enter (Press 'Y' or 'N') ?"
570 get r$ : if  r$ = "" then 570
580 if r$ = "y" or r$ = "Y" then 310
585 if r$ <> "n" and r$ <> "N" then 570
590 close 1
600 print "File closed."
610 end
620 print : input "Name "; n$
630 if n$ = "" then print "No entry made. Pleas
     e enter the name." : goto 620
640 if len(n$) > 20 then print "Abbrev. to 20 c
     haracters or less." : goto 620
650 if len(n$) < 20 then let n$ = left$(n$ + "
                                ", 20 )
```

```
660 return
670 print : input "Street address "; a$ : let a
    = len(a$)
680 if a$ = "" then print "No entry--please ent
    er the street." : goto 670
690 if a > 20 then print "Address too long--20
    letters or less." : goto 670
700 if a < 20 then let a$ = left$(a$ + "
                    ", 20 )
710 return
720 print : input "City name "; c$
730 if c$ = "" then print "No entry--please ent
    er the city." : goto 720
740 if len(c$) > 20 then print "City too long--
    20 letters or less" : goto 720
750 if len(c$) < 10 then let c$ = left$(c$ + "
                          ", 10 )
760 return
770 print : input "Enter 2-letter state abbrevi
    ation "; s$
780 if s$ = "" then print "No entry--please ent
    er the state." : goto 770
790 if len(s$) <> 2 then print "Please use stan
    dard 2 letter code" : goto 770
800 return
810 print : input "Enter the zip code (5 digits
    ) "; z$
820 if z$ = "" then print "No entry--please ent
    er the zip." : goto 810
830 if len(z$) <> 5 then print "Please enter ex
    actly 5 digits." : goto 810
840 return
```

DISPLAYING ONE DATASET AT A TIME FROM A FILE

Whenever we work extensively with files, we write a small utility program that lets us read through the file, one item at a time, to verify that everything is as it should be. A properly written data file-editing program also lets you make changes in the file data as it reads through the file (as shown in Chapter 6). Our example will use the previous application, the ADDRESS file.

The program shown below allows you to look at each dataset, one item at a time, with the prompt "PRESS RETURN FOR NEXT ADDRESS." The "PRESS _____ TO CONTINUE" technique is very popular for screen-oriented systems (as opposed to printer systems). It allows the computer user to review the data displayed for as long as necessary, and then move on to the next dataset. The most frequently used keys are RETURN and SPACE. We prefer RETURN, as users are accustomed to pressing it when done; beware the "PRESS ANY KEY TO CONTINUE"—users have been known to spend hours trying to decide which key to press—only to press the RUN/STOP key and seriously damage the program!

The program clears the screen to reduce "screen clutter" (previously displayed information) before each dataset is displayed, using the CLR (SHIFT-CLR/HOME) within quotes in a print statement. See line 330. Notice how the GET statement is used with the "PRESS RETURN TO CONTINUE" technique. As mentioned earlier, we prefer using GET for single-keystroke entries, or when waiting for a keypress, because on the PET, pressing RETURN in response to INPUT ends the program.

Help us to complete this companion program to the ADDRESS file-creating program. This "file read and display" program should display each dataset in ADDRESS, one at a time. Have each name and address displayed in mailing label format, using string functions to "de-concatenate" the data items for display (lines 280, 290, and 300). Also complete the data file-handling statements in lines 250 and 260.

```
100 rem  read and display address file one data
     set at a time
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem  variables used
130 rem  n$ = name (20 char. max.)
140 rem  a$ = address (20 char. max.)
150 rem  c$ = city (10 char. max.)
160 rem  s$ = state (2 char. )
170 rem  z$ = zip code (5 char. )
180 rem  d$ = entire dataset concatenated (57 c
     har.)
190 :
200 rem  seq. file name = address
210 rem  dataset format = d$
220 :
230 rem  initialize
240 :
250 _____________________________ : rem ope
     n
260 _____________________________ : rem inp
     ut
270 :
275 print "[clr][down][down][down][down][down]"
      : rem clear screen and down 5 lines
280 _____________________________ : rem pri
     nt name
290 _____________________________ : rem pri
     nt addr
300 _____________________________ : rem pri
     nt city etc.
310 if st <> 0 then close 1 : end
320 print : print "Press 'RETURN' for next addr
     ess."
325 get z$ : if z$ <> chr$ (13) then 325 : rem
     wait for return
330 print "[clr]" : goto 260 : rem clear & rest
     art
```

```
230 rem  initialize
240 :
250 open 1, 8, 8, "address,s,r"
260 input#1, d$
270 :
275 print "[clr][down][down][down][down][down]"
      : rem clear screen and down 5 lines
280 print left$(d$,20)
290 print mid$(d$,21,20)
300 print mid$(d$,41,10) + " " + mid$(d$,51,2)
     + " " + right$(d$,5)
310 if st <> 0 then close 1 : end
320 print "[down]Press [rvs]RETURN[off] for n
     ext address."
325 get r$ : if r$ <> chr$(13) then 325 : rem
     wait for return
330 goto 260
```

Refer to the program above to answer these questions.

(a) What is assigned to R$ in line 325? _____

_____

(b) Since R$ acts as a dummy variable in the program above, what is the purpose of line 325? _____

_____

_____

(c) How often was the screen "refreshed" in the program above?

_____

(a) Nothing (an empty string).
(b) Keeps the data items on the screen until the user presses the RETURN key to continue. The program returns to the GET statement until the RETURN key is pressed.
(c) Before (or after) each complete dataset of three items was displayed.

THE SUPER DISK ERROR ROUTINE

The routine developed below will give you and those using your program almost total control over disk errors, so the user will never be surprised by a nasty disk error message, resulting in termination of the program.

_____

The routine is used by calling it with a GOSUB after *every* disk access—OPEN, INPUT#, PRINT#, and CLOSE, as well as uses of PRINT# to access the disk command channel to rename, scratch, or concatenate files.

The simplest routine is

```
900 REM     DISK ERROR ROUTINE
910 INPUT#15, X, X$, Y, Z
920 PRINT X, X$
990 RETURN
```

This prints whatever message results from disk access, then returns. Since disk errors divide into several types, we need to add lines to handle each of them. The first addition we can make is to ignore errors of less than 20:

```
920 IF X < 20 THEN RETURN
925 PRINT "[RVS]ERROR #" X "," X$
```

Refer to Appendix 3 to verify that several types of disk error will never show up in a properly debugged file-writing program [numbers 30–39 (Syntax Errors), 60–64 (File open/closed, file exists, file not found), and 70 (No channel)]. For sequential files, we can ignore errors 50–52. The number left are manageable. We'll develop some for you, then have you help finish the task.

Let's start with an easy one: error 26, WRITE PROTECTION.

```
930 IF X = 26 THEN PRINT " REMOVE TAB AND REINS
      ERT DISK." : GOTO 980
980 PRINT "PRESS [RVS]RETURN[OFF] TO CONTINUE."
985 GET Z$ : IF Z$ <> CHR$ (13) THEN 985
```

Errors 20–27 and 74 are all caused either by no disk in the drive, or by problems reading the disk, so we can provide one error message:

```
940 IF X = 74 OR X > 28 THEN 960
950 PRINT "REMOVE, REINSERT YOUR DISK AND CLOSE
      DRIVE DOOR."
955 PRINT "IF ERROR RECURS, END PROGRAM AND USE
      ANOTHER DATA DISK."
```

(a) Since error 74 may also result from the drive not being on, provide an error message for it:

960 _____

_____

```
--------------------
```

(a)
```
960 IF X = 74 THEN PRINT "IS YOUR DRIVE TURNED
    ON?"
```

Here's the message for error 29, which is avoidable if you initialize disks before you open each file or perform file management routines to that drive:

```
970 IF X = 29 THEN PRINT#15, "I0"
```

(a) Now try your hand at error 62—FILE NOT FOUND; the user probably inserted the wrong disk:

975 _____

_____

```
--------------------
```

(a)
```
975 IF X = 62 THEN PRINT "INSERT CORRECT DISK I
    N DRIVE."
```

The last error is 72, DISK FULL. It can be handled by a message similar to the last, so let's modify line 975:

```
977 IF X = 72 OR X = 73  THEN PRINT "INSERT ANO
    THER DISK IN DRIVE."
```

Now we have the major parts, some refinements can be made. The first involves lines 940–955. Ideally, when such problems arise, you want to close your files, remove the disk, and restart the program. We can do that by changing lines 950 and 955:

```
950 CLOSE 15 : PRINT "[RVS]FATAL ERROR. REMOVE
      DISK AND INSERT ANOTHER ONE."
955 PRINT "PROGRAM RESTARTED: PLEASE REENTER DA
      TA." : GOTO 100
```

While this is a bit more abrupt, it leaves the operator free of worry about what to do. The last refinement involves giving the user the choice of whether to continue upon an error. This requires modifying lines 980 and 985:

```
980 PRINT "PRESS [RVS]RETURN[OFF] TO CONTINUE,
    OR [RVS]S[OFF] TO STOP."
985 GET Z$ : IF Z$ <> "S" AND Z$ <> CHR$(13) TH
    EN 985
988 IF Z$ = "S" THEN CLOSE15 : END
```

When error conditions exist, you may be better off terminating the program, rather than allowing the user to continue.

While other improvements are doubtless possible, this will cover all common errors, and most of the uncommon ones as well. Since it is used as a subroutine, you only need to type it once, at the end of your program. For short, personal programs, it's overkill, but for longer programs, or those to be used by others, you'll find it essential. Here's the entire routine:

```
900 REM   DISK ERROR ROUTINE
910 INPUT#15, X, X$, Y, Z
920 IF X < 20 THEN RETURN
925 PRINT "[RVS]ERROR #" X "," X$
930 IF X = 26  THEN PRINT "REMOVE TAB AND REINS
    ERT DISK." : GOTO 980
940 IF X = 74 OR X > 28 THEN 960
950 CLOSE15 : PRINT "[RVS]FATAL ERROR!  REMOVE
    DISK AND INSERT ANOTHER ONE."
955 PRINT "PROGRAM RESTARTED: PLEASE REENTER DA
    TA." : GOTO 100
960 IF X = 74 THEN PRINT "IS YOUR DRIVE TURNED
    ON?"
970 IF X = 29 THEN PRINT#15, "IO"
975 IF X = 62 THEN PRINT "INSERT CORRECT DISK I
    N DRIVE."
977 IF X = 72 OR X = 73  THEN PRINT "INSERT AND
    THER DISK IN DRIVE."
980 PRINT "PRESS [RVS]RETURN[OFF] TO CONTINUE,
    OR [RVS]S[OFF] TO STOP."
985 GET Z$ : IF Z$ <> "S" AND Z$ <> CHR$(13) TH
    EN 985
988 IF Z$ = "S" THEN CLOSE 15 : END
990 RETURN
```

USING A DISK ERROR SUBROUTINE

Now that we have written this wonderful disk error subroutine, how is it used? On some of these errors, when we return to the program we want to repeat the operation that lead to the error. This requires a line following the GOSUB to check for a specific condition and return to the beginning of that line, or to a previous line.

For example, look at the introductory module below. Assume we have run the program with the wrong disk in the drive, so that when the OPEN statement is executed, an error ("?FILE NOT FOUND") is signaled. If the user chooses to continue following the message "Insert correct disk in drive," we need to re-execute the OPEN command, or the program will get stuck trying to INPUT from an unOPENed file.

```
200  REM   INITIALIZATION
210  OPEN 15, 8, 15
220  OPEN 1, 8, 8, "SAMPLE,S,R"
230  GOSUB 800
240 :
250  REM  READ DATA
260  INPUT#1, N$
```

(a) Modify line 230 above so the OPEN statement will be correctly executed should the user choose to continue:

_____

_____

(a) `230 GOSUB 800 : IF X > 20 THEN CLOSE 15 : GOTO 210`

In this example, since file 1 is open, we must close it before re-executing the OPEN statement. Line 230 also could have been written as

```
230 GOSUB 800 : IF X > 20 THEN CLOSE 1 : GOTO 220
```

The level of control over disk errors you should write into your programs depends upon who will be using them. We've given you a sample disk error routine, and one possible way of dealing with disk errors within your program. If others will be using your programs for important data (or if you are), it is worth the time to carefully consider all possible disk disasters, and to prepare your program to cope with them. It is particularly important to remember to close disk files when there is an error that stops the program.

Again, considering the possible level of knowledge of the user, you may wish to write your error routine to stop on any error other than perhaps "FILE NOT FOUND," "DISK NOT READY," or "FILE EXISTS."

(a) What is the best way to assure closing any open disk files if a program stops because of an error condition?

(b) Write the appropriate BASIC statements to do that.

(a) Open a file to the disk error/command channel, read the channel, and close the file.

(b) PET: `Open 15, 8, 15 : CLOSE 15`
C-64: Using the DOS Wedge, type ">".
C-64:
```
9999 OPEN 15, 8, 15 : INPUT#15, X, X$: PRINT X$ : CLOSE 15
     RUN 9999
```
(You could use any line number greater than the highest used in your program. You need to open the disk channel in case it was closed. If you get a "?FILE OPEN" error, just repeat the statement.)

Chapter 5 Self-Test

The problems in this self-test require you to write programs to store data in data files and then to write companion programs to display the data in those data files. All data files that you create in this self-test will be used in Chapter 6, *so don't skip this section*. You will need a system with a disk drive to actually store the programs. If you have only a cassette data storage system, write the programs anyway, as only minor modification will be needed in most cases to adapt the programs to cassette data file application. The introductory module is given so your solutions will look something like the solution provided. Save the programs and files for later use, modification, and reference. Try out your solutions and try to debug the programs before looking at the solutions provided. Believe me, our "first draft" solutions had to be debugged, too! Good luck and keep on hackin'.

1. (a) Write a program to make a data file called GROCERY that stores your grocery shopping list. Include the description or name of each grocery item (maximum of twenty characters) and a numeric value telling the quantity of that item to buy. Store at least four datasets in the file.

Here is a sample RUN, followed by the introductory module and workspace to write the program:

```
run
name of input file? grocery
enter 'stop' when finished

item description? bread
quantity? 2

item description? butter
quantity? 2

item description? smoked turkey
quantity? 1

item description? swiss cheese
quantity? 3

item description? stop
files closed.
ready.
```

```
100 rem   prob 5-1a solution (grocery list)
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   introductory module
130 rem   variables used
140 rem       d$ = item description
150 rem       q  = quantity to order
160 rem       f$ = user entered file name
170 :
180 rem   files used
190 rem       seq. file = grocery (user entered
     )
200 rem       dataset format: d$,q
210 :
220 rem   file initialization
230 :
```

1. (b) Write a companion program to display the contents of GROCERY. Here is a sample RUN:

```
Run
Name of input file: grocery

Item            Quantity

Bread           2
Butter          2
Smoked turkey   1
Swiss cheese    3
Ready.
```

2. (a) Write a program to enter the following data in a data file for a customer credit file maintained by a small business. Each dataset consists of three items:

   1. five-digit customer number (must have exactly five digits);
   2. customer name (twenty characters maximum);
   3. customer credit rating (a single digit number 1, 2, 3, 4, or 5).

   Include data entry checks for no entry and for the parameters set forth in the list above. Enter at least two datasets in the data file. Remember, the customer numbers must be different for each customer and should be in *ascending* order, i.e., each larger than the previous one, such as 19652, 19653, 19654, etc.

   A sample RUN is followed by the program's introductory module.

```
run
name of writefile? credit

customer number? 12345
customer name? jrb, inc.
credit rating? 3

do you have more data to enter? y

customer number? 123456
customer name? lp corp
credit rating? 5

do you have more data to enter? n
job completed.
ready.
```

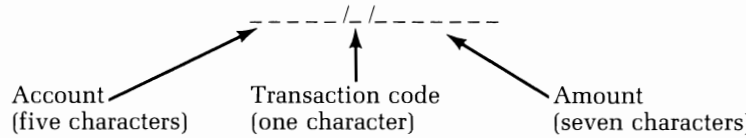
```
100 rem   self-test prob. 5-2a solution
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
110 rem  credit file loader
120 :
130 rem   variables used
140 rem      f$ = file name (user entered)
150 rem      c$ = customer number
160 rem      n$ = customer name
170 rem      r$ and r = credit rating value
180 rem      q$ = user response to continue dat
    a entry
190 :
200 rem   files used
210 rem      seq. file = credit (user entered)
220 rem      dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "Write-file name "; f$
270 open 1, 8, 8, "@0:" + f$ + ",s,w"
280 :
290 rem   data entry routine
300 print "[clr]" : rem clear screen
```

2. (b) Write a companion program to display the contents of the file. Here is a sample RUN:

```
Run
File name? Credit
Cust #        Cust name       Credit rating
12345         JRB, INC.       3
12346         LP CORP         5

All data displayed and file closed.
Ready.
```

3. (a) Write a program to enter data into a transaction file. A transaction file is the data on a business transaction, such as that of a bank, a retail store, or a mail-order business. For our example, each transaction produces a dataset stored as one thirteen-character string with three fields, as shown below:



Account number = five-character field

Transaction code = one-characater field (for a bank, 1 = check, 2 = deposit, etc.)

Amount = seven-character field

Your program should include data entry checks for no entry and for the parameters set forth above. Check cash amount entries for nonnumeric characters, except the decimal point. Your program should allow the user to select (input) a name for the data file.

Create two different data files with your program, with seven datasets (seven transactions) in each data file. Name file #1 TRANSAC1, and name file #2 TRANSAC2. Use the account numbers given below for the two files. For duplicate account numbers in the same file, make a complete dataset entry for each account number, so that each of the two files contains seven datasets.

File #1	File #2
10762	10761
18102	18203
43611	43611
43611	80111
43611	80772
80223	80772
98702	89012

Note: Only the account numbers are shown here; the complete datasets also include transaction codes and amounts, to be supplied from your imagination.

The sample RUN is followed by the program's introductory module.

```
Run
Write-file name: transac2

Enter '-1' instead of account number when finished.

Account number (5 digits): 10761
Transaction code: 1
Amount: $255.88

Enter '-1' instead of account number when finished.

Account number (5 digits): 18203
Transaction code: 2
Amount: $12.98

Enter '-1' instead of account number when finished.

Account number (5 digits): 43611
Transaction code: 1
Amount: $9482.08

Enter '-1' instead of account number when finished.

Account number (5 digits):
Transaction code: 2
Amount: $178.67

Enter '-1' instead of account number when finished.

Account number (5 digits): 80772
Transaction code: 2
Amount: $10.25

Enter '-1' instead of account number when finished.

Account number (5 digits): 80772
Transaction code: 1
Amount: $18.85

Enter '-1' instead of account number when finished.

Account number (5 digits): 89012
Transaction code: 1
Amount: $225.75
```

```
Enter '-1' instead of account number when finished.

Account number (5 digits): -1
File closed.
Ready.

100 rem  prob. 5-3a solution--create transac# f
     iles
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem    f1$ = file name
140 rem    d1$ = datasets from file 1,2
150 rem     a$ = account # (5 char.)
160 rem     t$ = transaction code (1)
170 rem     c$ = cash amt. (9999.99 max)
180 rem     x  = for-next loop variable
190 :
200 rem   files used
210 rem     seq. file = transac1 or transac2 (u
     ser entered)
220 rem     dataset format: d1$ (concatenated f
     rom a$ + t$ + c$)
230 :
240 :
250 rem   file initialization
```

3. (b) Write a companion program to display the contents of a data file with the above dataset format. Again, the file name should be user entered. A sample RUN for TRANSAC1 is shown.

```
run
File name to read? transac1
Account #       Code       Amount
10762           1          152.33
18102           1           11.95
43611           1           55.25
43611           2          145.01
43611           1          589.45
80223           2           75.98
98702           2           23.88

all data displayed and file transac2 closed.
ready.
```

4. (a) Write one program and use it to create three different data files called LETTER1, LETTER2, and LETTER3. Each file should contain the text of a form letter. Each line of text in the letters is to be entered and stored as one dataset. A sample RUN is followed by the program's introductory module.

```
run
Letter file number: 1
Enter text line or 'stop'
We regret to inform you that electricl service to your area
Enter text line or 'stop'
will be discontinued beginning next month.
Enter text line or 'stop'
We hope this will not inconvenience you.
Enter text line or 'stop'
stop
File letterl closed.
ready.
```

```
100 rem   prob. 5-4a solution--creates letter#
     files
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem    r$ = text line
140 rem    f$ = user entered file number concat
     enated with 'letter'
150 :
160 rem   files used
170 rem     seq. file = letter (plus f$, a user
     -entered number)
180 rem     dataset format: r$ (text of letter#
     )
190 :
200 rem   file initialization
210 :
220 print : input "Letter file number "; f$
```

4. (b) Write a companion program to display the data file above selected by the user. Only the number for LETTER# need be entered. Our sample RUN looks like this for file LETTER1:

```
run
Letter file number: 1
We regret to inform you that electrical service to your area
will be discontinued beginning next month.
We hope this will not inconvenience you.

Letter1 file closed.
ready.
```

Answer Key

1. (a)
```
100 rem   prob 5-1a solution (grocery list)
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   introductory module
130 rem   variables used
140 rem       d$ = item description
150 rem       q  = quantity to order
160 rem       f$ = user entered file name
170 :
180 rem   files used
190 rem       seq. file = grocery (user entered
     )
200 rem       dataset format: d$,q
210 :
220 rem   file initialization
230 :
240 print : input "Name of write file "; f$
250 open 1, 8, 8, "@0:" + f$ + ",s,w"
260 :
270 rem   data entry routine
280 :
290 print "Type 'STOP' instead of description w
     hen finished with data entry."
300 print
310 print : input "Item description "; d$
320 if d$ = "stop" or d$ = "STOP" then 470
330 if d$ = "" then print "Enter a description
     or 'STOP'." : goto 310
340 if len(d$) > 20 then print "Limit descripti
     on to 20 characters." : goto 310
350 print : input "Quantity "; q$ : q = val(q$)

355 if q = 0 and q$ <> "0" then print "Enter a
     number." : goto 350
360 if q >= 1 and q < 10 then 430
370 print "You entered a quantity of " q
380 print "Is that what you wanted ? (Press 'y'
      for yes or 'n' for no.)
390 get r$ : if r$ <> "y" and r$ <> "n" then 39
     0
400 if r$ = "n" then 350
410 rem   write to file routine
420 :
430 print#1, d$ : print#1, q
440 print "Written to file: " d$; q : print : g
     oto 290
450 :
460 rem   close file
470 close 1
480 print "File closed."
```

1. (b)
```
100 rem  self-test prob. 5-1b solution--read &
     display grocery datafile
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem  variables used
130 rem    d$ = item description
140 rem    q  = quantity to order
150 rem    f$ = file name (user entered)
160 :
170 rem  files used
180 rem    seq. file = grocery (user entered)
190 rem    dataset format: d$, q
200 :
210 rem  file initialization
220 input "Name of input file "; f$
230 open 1, 8, 8, f$ + ",s,r"
240 :
250 rem  read and print file
260 :
270 print "[clr][down][down]" : print "Item" ta
     b(22) "Quantity" : print
280 input#1, d$, q
290 print d$ tab(22) q
300 if st = 0 then 280
310 :
320 rem   close routine
330 close 1
340 print : print "File closed."
```

2. (a)
```
100 rem   self-test prob. 5-2a solution
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 rem  credit file loader
120 :
130 rem   variables used
140 rem      f$ = file name (user entered)
150 rem      c$ = customer number
160 rem      n$ = customer name
170 rem      r$ and r = credit rating value
180 rem      q$ = user response to continue dat
     a entry
190 :
200 rem   files used
210 rem      seq. file = credit (user entered)
220 rem      dataset format: c$, n$, r
230 :
```

```
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "Write-file name "; f$
270 open 1, 8, 8, "@0:" + f$ + ",s,w"
280 :
290 rem   data entry routine
300 print "[clr]" : rem clear screen
310 print
320 print : input "Customer number "; c$
330 if c$ = "" then print "Please enter a numbe
    r." : goto 320
340 if val(c$) = 0 then print "Entry error--num
    bers only." : goto 320
350 if len(c$) <> 5 then print "Entry error: us
    e 5 digits." : goto 320
360 :
370 print : input "Customer name "; n$
380 if n$ = "" then print "Please enter a name,
     now." : goto 370
390 if len(n$) > 20 then print "Please limit na
    me to 20 characters." : goto 370
400 :
410 print : input "Credit rating "; r$
420 let r = val(r$)
430 if len(r$) <> 1 or r = 0 then print "Only a
     one-digit number!" : goto 410
440 if r < 1 or r > 5 then print "Numbers 1 - 5
     only!" : goto 410
450 :
460 rem   print new data to file
470 print#1, c$ : print#1, n$ : print#1, r
480 print "Writing " c$, n$, r
490 :
500 rem   request to continue data entry
510 print
520 print "Do you have more data to enter ? (Pr
    ess 'y' or 'n'.)
530 get q$ : if q$ <> "y" and q$ <> "n" then 53
    0
540 if q$ = "y" then 300
550 rem   close files
560 close 1
570 print "Job completed."
```

2. (b)
```
100 rem   self-test prob. 5-2b solution
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 rem  credit file display
120 :
130 rem   variables used
140 rem      f$ = file name (user entered)
150 rem      c$ = customer number
160 rem      n$ = customer name
170 rem      r  = credit rating value
180 :
190 :
200 rem   files used
210 rem      seq. file = credit (user entered)
220 rem      dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "File name to read "; f$
270 open 1, 8, 8, f$ + ",s,r"
280 :
290 rem   read/print file
300 print : print "Cust. #" tab(8) "Cust. Name"
      tab( 30) "Credit"
310 print tab(30) "Rating"
320 input#1, c$, n$, r
330 print c$ tab(8) n$ tab(32) r
340 if st = 0 then 320 : rem not end of file
350 :
360 rem   close file
370 close 1
380 print : print "All data displayed and file
     closed."
```

3. (a)
```
100 rem  prob. 5-3a solution--create transac# f
     iles
101 if peek(50003) <> 0 then poke 59468,14 : re
     m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem    f1$ = file name
140 rem    d1$ = datasets from file 1,2
150 rem     a$ = account # (5 char.)
160 rem     t$ = transaction code (1)
170 rem     c$ = cash amt. (9999.99 max)
180 rem     x  = for-next loop variable
190 :
200 rem   files used
```

```
210 rem     seq. file = transac1 or transac2 (u
    ser entered)
220 rem     dataset format: d1$ (concatenated f
    rom a$ + t$ + c$)
230 :
240 :
250 rem   file initialization
260 print : input "Write-file name "; f1$
270 open 1, 8, 7, "@0:" + f1$ + ",s,w"
280 print "[clr]" : rem clear screen
290 :
300 rem   data entry/tests
310 print "Enter '-1' to end data entry."
320 print
330 input "Account number (5 digits) "; a$
340 if a$ = "-1" then 590
350 if val(a$) = 0 then print "Please make a nu
    meric entry." : goto 310
360 if len(a$) = 5 then 380
365 print "You entered [rvs]" a$ "[off].  Only
    5 digits." : goto 310
370 :
380 print : input "Transaction code (1 digit) "
    ; t$
390 rem   data test
400 if t$ = "" then print "Please make a one-di
    git entry." : goto 380
410 if len(t$) <> 1 then print "Enter one-digit
     code.": goto 380
420 :
430 print : input "Amount (no '$') "; c$
440 if c$ = "" then print "Please make an entry
    " : goto 430
450 if val(c$) > 9999.99 then print "Max. is 99
    99.99; reenter." : goto 430
460 for x = 1 to len(c$)
470 let c = asc(mid$(c$,x,1))
480 if c >= 48 and c <= 57 or c = 46 then 500
490 print "Invalid entry. Use only numbers and
    dec-imal point." : goto 430
500 next x
510 if len(c$) < 7 then let c$ = right$ ("
    " + c$,7)
520 let d1$ = a$ + t$ + c$
530 :
540 print#1, d1$
550 print "[clr]" : rem clear screen
560 goto 310
570 :
580 rem   close file
590 close 1
600 print "File closed."
```

3. (b)
```
100 rem   prob. 5-3b solution--read and display
      transac# files
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem    f1$ = file name
140 rem    d1$ = datasets from file 1 or 2
150 :
160 rem   files used
170 rem     seq. file = transac1 or transac2 (u
    ser entered)
180 rem     dataset format: d1$ (a)concatenated
      dataset--see line 280)
190 :
200 rem   file initialization
210 print : input "File name to read "; f1$
220 open 1, 8, 7, f1$ + ",s,r"
230 :
240 rem   file read and print
245 print "[clr][down][down]" : rem clear and 2
      lines down
250 print "Acct. #", "Code", "Amount"
260 input#1, d1$
270 print left$(d1$,5), mid$(d1$,6,1), right$(d
    1$,7)
280 if st = 0 then 260
290 :
300 rem   close
310 close 1
320 print : print "All data displayed."
330 print "File " f1$ " closed."
```

4. (a)
```
100 rem   prob. 5-4a solution--creates letter#
    files
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
      c-64 lower case
110 :
120 rem   variables used
130 rem    r$ = text line
140 rem    f$ = user entered file number concat
    enated with 'letter'
150 :
160 rem   files used
170 rem     seq. file = letter (plus f$, a user
    -entered number)
180 rem     dataset format: r$ (text of letter#
    )
```

```
190 :
200 rem   file initialization
210 :
220 print : input "Letter file number "; f$
230 let f$ = "letter" + f$
240 open 2, 8, 8, f$ + ",s,w"
250 :
260 print : print "Type line of text or 'STOP'.
    "
265 print "Start line with quotes to include co
    mma, colon, or semicolon."
270 print : input r$
280 if r$ = "stop" or r$ = "STOP" then 330
290 print#2, chr$(34) r$ chr$(34)
300 goto 260
310 :
320 rem   close file
330 close 2
340 print "File " f$ " closed."
```

4. (b)

```
100 rem   prob. 5-4b solution--read and display
    letter# files
101 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
102 if peek(50003) = 0 then poke 53272,23 : rem
    c-64 lower case
110 :
120 rem   variables used
130 rem    r$ = text line
140 rem    f$ = user entered file number concat
    enated with 'letter'
150 :
160 rem   files used
170 rem     seq. file = letter (plus f$, a user
    -entered number)
180 rem     dataset format: r$ (text of letter#
    )
190 :
200 rem   file initialization
210 :
220 print : input "Letter file number "; f$
230 let f$ = "letter" + f$
240 open 5, 8, 8, f$ + ",s,r"
250 :
260 rem   read/print file contents
270 input#5, r$
280 print r$
290 if st = 0 then 270
300 :
310 rem   close files
320 close 5
330 print : print "File " f$ " closed."
```

CHAPTER SIX

Sequential Data File Utility Programs

Objectives: When you finish this chapter you will be able to:

1. Write a program to make a copy of a sequential data file.
2. Write a program to add data to an existing sequential file.
3. Write a program to change the data in an existing sequential file.
4. Write a program to examine the contents in a sequential file and to change, add, or delete data.
5. Write a program to merge the contents of two sequential files into one file, maintaining the numeric or alphabetic order of the data.
6. Write a program that uses or combines selected data from more than one sequential file.

Now that you understand the BASIC statements to create and use sequential data files, let's build on this with more advanced techniques, including writing some file utility programs that help in your overall programming using data files. You will also develop embryonic file applications to practice what you have learned and provide a basis from which to develop personally useful programs. Most of the data files used in this chapter are created with programs you should have written for the Chapter 5 Self-Test, so if you skipped that, go back and write those programs before starting this chapter.

MAKING A DATA FILE COPY

A very useful file utility program is one that makes a duplicate copy of your data file. COPY is a Commodore disk command with several functions. Check your disk drive manual (see also Appendices 2 and 5) for information on its use. Using COPY or a utility program, you can make backup copies of data files, or copy a file from one disk drive to another, or copy the contents of a file onto the same disk with another name.

If you have two drives, refer to your disk manual for instructions on setting device numbers and on including device numbers in disk commands; you will have to set one of your drives to device 9. For the dual drives, you can copy from one drive to another by simply specifying the drive numbers.

A file copy utility program in BASIC not only allows you to make backup copies of data files, it can also be incorporated into programs to add data to existing data files.

You have the background to write a file copying program. Follow these steps:

1. Open the READ file (the one you want to copy).
2. Open the WRITE file (the one you are copying into).
3. Input the next dataset from the READ file.
4. Print the dataset to the new file (the copy).
5. Check for end of file.
6. If not found, repeat steps 3–5.
7. Close both files.

Assume you are going to copy a file that contains an unknown number of datasets, each dataset being a single string concatenated from three data items. This is the format for TRANSAC1 and TRANSAC2 from Problem 3(a) in the Chapter 5 Self-Test. (You did create those files, didn't you?)

(a) Complete the program below to make a copy of TRANSAC1, and name the file copy TRANSAC1.CPY. Fill in lines 240, 250, 290, 310, 320, and 350.

```
100 REM   SEQUENTIAL FILE COPYING PROGRAM
110 :
120 REM   VARIABLES USED
130 REM    D1$ = CONCATENATED DATASET
140 REM    F$, FA$ = USER ENTERED FILE NAMES
150 :
160 REM   FILES USED
170 REM     SEQ. FILE = TRANSAC1, TRANSAC1.CPY
180 REM     DATASET FORMAT: D1$ (CONCATENATED F
    ROM A$ + T$ + C$)
190 :
200 REM   FILE INITIALIZATION
210 OPEN 15, 8, 15 : REM DISK ERROR CHANNEL
220 INPUT "NAME OF INPUT FILE "; F$
230 INPUT "NAME OF WRITE FILE "; FA$
240 _____
250 _____
260 :
270 REM   READ INPUT FILE, WRITE TO OUTPUT FILE

280 :
290 _____
300 LET SV = ST : REM SAVE ST FROM INPUT
310 _____
320 _____
330 :
340 REM   CLOSE FILE ROUTINE
350 _____
360 GOSUB 400 : REM DISK ERROR CHECK
370 END
390 :
400 REM DISK ERROR ROUTINE
410 INPUT#15, X, X$, Y, Z
420 IF X < 20 THEN RETURN
430 PRINT "[DOWN][DOWN][RVS]DISK ERROR #" X ":
    " X$
440 CLOSE15 : END
```

(b) To review the steps for this kind of program, write the corresponding line number(s) for each step in the outline.

1. Open the READ file _____
2. Open the WRITE file _____
3. Input the next dataset from the READ file _____
4. Print the dataset to the new file _____
5. Check for end of file _____

6. If not found, repeat steps 3–5

7. Close both files

(c) When you RUN this program, what appears on your screen? Show the RUN here.

(a)
```
240 OPEN 1, 8, 8, F$ + ",S,R"
250 OPEN 2, 8, 8, FA$ + ",S,W"
290 INPUT#1, D1$
310 PRINT#2, D1$
320 IF SV = 0 THEN 290
350 CLOSE 1 : CLOSE 2
```

(b)
1. 240
2. 250
3. 290
4. 310
5. 300 or 320
6. 320
7. 350

(c)
```
RUN
NAME OF INPUT FILE? TRANSAC1
NAME OF WRITE FILE? TRANSAC2
READY.
```

It can be unsettling to get no more than a "READY." from a program when so much internal activity is supposed to be taking place. The final "READY." is your only clue that your program has completed its task. But you don't know for sure. We have a few suggestions. Add a statement at line 360 that prints a message indicating the job is complete—for example, 360 PRINT "COPY COMPLETED." A statement such as that lets you know the program did execute past the CLOSE statement at line 350. (That's why we chose line 360.)

A second suggestion is to OPEN the copied data file for input, read the datasets from it, and display them. This will verify that the copy was made. We often do this because our first versions of these example programs always had errors in them, and displaying the file can provide debugging clues.

```
430 OPEN 2, 8, 8, F1$ + ",S,R"
440 INPUT#2, D1$
450 PRINT "READING " D1$ : REM DISPLAY DATASET
460 IF ST = 0 THEN 440
470 CLOSE 2
```

You now have a complete file-copying utility program. You can use it to copy any data file by simply changing the INPUT# and PRINT# statements in lines 290 and 310 to conform to the data format or datasets in the particular data file

you want to copy. Again, if a file program uses a separate PRINT# statement for each variable, and writes all data as strings, this program as written will copy the files produced. Uniformity has its virtues.

This program also illustrates another problem that may haunt you. We have used F1$ and D1$ for variables. Some of you may have typed the letter "i" or "l" instead of the number "1." Wherever possible, use variable names that cannot be mistaken. FA$ would be preferable to F1$. Zero and the letter "O" is the other pair that causes mistakes; we recommend that you not use them in variable names.

A more universal copy program would use GET# instead of PRINT#. GET# reads the file character by character, allowing the programmer to decide how to respond to punctuation in the string. Because it reads character by character, it is noticeably slower! The most usual routine simply concatenates characters until a carriage return is found, then prints the concatenated string. (You may run into garbage collection problems on older PETs or C-64s; see Appendix 4.) Here is the file reader using GET# (see the File Reader Utility programs in Chapter 5, page 219–220):

```
430 OPEN 2, 8, 8, F1$ + "S,R"
440 S$ = "" : REM EMPTY STRING
450 GET#2, C$
460 IF C$ <> CHR$(13) THEN S$ = S$ + C$ : GOTO 450
470 PRINT S$ : IF ST = 0 THEN 440
480 CLOSE 2
```

If you are printing to another file or the printer, don't forget to save the ST after the GET# in line 450; you will also have to change IF ST . . . to IF (your variable name) . . ., as discussed in DETECTING THE END OF A FILE, Chapter 5, page 217–218).

ADDING DATA TO THE END OF A SEQUENTIAL FILE

In Commodore BASIC it is easy to add data to the end of a sequential file. There are actually two ways to do this. The most consistent is to use the command "APPEND" (see Appendix 5 for the BASIC 4.0 version). The other method is to use the disk command CONCATENATE, which simply connects two data files on the disk. Examples of both methods will follow.

We recommend APPEND rather than CONCATENATE for adding data to the end of a file. Since CONCATENATE physically links files on the disk, there is no guarantee that datasets in the concatenated files will match. With APPEND, you have to use a program to do the adding, and it is likely to use the same dataset format.

APPENDING TO A FILE

To append to a file, you must OPEN the file as an APPEND file:

```
OPEN 1, 8, 8, "ADDRESS,A"
```

An APPEND file is assumed to be a sequential WRITE file, so you only need to specify the "A". The comma is essential, as it is for the S and R or W in the usual sequential file OPEN command.

(a) Write an OPEN statement to OPEN a file to the name stored in F$. _____

- - - - - - - - - - - - - - - - - - -

(a) `OPEN 1, 8, 8, F$ + ",A"`

The comma *must* be included inside the quotes with the designation A; it is part of the OPEN format. Other than the different letter, an APPEND OPEN statement is like any other OPEN statement; you can specify drive number (0 or 1) if you have dual drives.

You *cannot* OPEN an APPEND file for replacement; that is, the following statement is *not* allowed:

```
OPEN 1, 8, 8, "@0:FILENAME,A"
```

When you OPEN a file for APPEND, the file pointer is moved to the end of the file data, so that subsequent PRINT# operations take place starting after the last piece of existing data, and new data are added or appended beyond the previous end of the file.

This means that to change the file entry programs you have written so far so that they add (APPEND) data to the end of an existing file, you only have to make one small change to the OPEN statement—changing the file mode from ",S,W" to ",A". There are two simple ways to accomplish this. The first is to write two OPEN statements in your program, and use an IF . . . THEN statement to select the appropriate one. The other is to set ",S,W" and ",A" as variables, and concatenate the appropriate variable in the OPEN statement.

Here's how it's done with IF . . . THEN statements:

```
100 print : input "Name of file "; f$
110 print "[rvs]N[off]ew or [rvs]O[off]ld file
      (Press [rvs]N[off] or [rvs]O[off]).
120 get r$ : if r$ <> "n" and r$ <> "o" then 120
130 if r$ = "o" then open 1, 8, 8, f$ + ",a"
140 if r$ = "n" then open 1, 8, 8, f$ + ",s,w"
```

Using variables to store the file mode works like this:

```
100 print : input "Name of file "; f$
110 print "[rvs]N[off]ew or [rvs]O[off]ld file
      (Press [rvs]N[off] or [rvs]O[off]).
120 get r$ : if r$ <> "n" and r$ <> "o" then 120
130 a$ = ",s,w" : if r$ = "o" then a$ = ",a"
140 open 1, 8, 8, f$ + a$
```

A more sophisticated method would be to use the disk error channel to indicate that a file exists, and if it does, to OPEN it as an APPEND file. You'll help us write this one.

Here are the steps:

1. Open the file for WRITE.
2. Check the disk error channel.
3. If no error, go to step 5.
4. If the error is 63, "FILE EXISTS," then
   a. close the file;
   b. open the file for APPEND.
5. Write to the file.

(a) Assuming, as usual, the file named is stored in F$, write the OPEN statement matching step 1:

120 _____

(a) `120 OPEN 1, 8, 8, F$ + ",S,W"`

Now, we read the disk error channel:

```
130 INPUT#15, X, X$
```

and check for error 63 (you could also check for X$ = "FILE EXISTS":)

```
140 IF X <> 63 THEN 170
```

Now help us finish:

(a) Write the line to close the WRITE file:

150 _____

(b) Write the line to reopen the file for APPEND:

160 _____

(a) `150 CLOSE 1`
(b) `160 OPEN 1, 8, 8, F$ + ",A"`

APPEND BY CONCATENATION

To APPEND two files with CONCATENATE requires that you first create both files, then issue the disk command to link the two files. The steps are

1. Write the original file.
2. Open and write data to be added to a temporary file (TEMPFILE).
3. Close TEMPFILE.
4. CONCATENATE the two files.
5. SCRATCH TEMPFILE.

The program to carry out step 2 can be based on the original program, or on many of the examples in the last chapter. The only new step is the last. Assuming file #15 is opened to the disk error/command channel, here they are:

```
800 PRINT#15, "C0:NEWFILE=0:OLDFILE,0:TEMPFILE"
810 PRINT#15, "S0:TEMPFILE"
```

(C is the abbreviation for CONCATENATE, and S for SCRATCH).

CONCATENATE has a few complications. *Don't* put *any* extra spaces inside the quotes; they may be read as part of the file name, causing "FILE NOT FOUND" errors. CONCATENATE works fine when you assign a new name to the concatenated file, as we did in this example. When you want to assign the name of an existing file, then that file name *must* be the first file in the list after the equals sign (=):

```
800 PRINT#15, "C0:OLDFILE=0:OLDFILE,0:TEMPFILE"
```

If you try to assign a concatenated file to any file name other than the first in the list, you will get a disk error, and the CONCATENATION won't work. Here's how *not* to do it:

```
800 PRINT#15, "C0:TEMPFILE=0:OLDFILE,0:TEMPFILE"
```

You can concatenate up to four files in one CONCATENATE statement. Be sure to read the disk error channel after a CONCATENATE statement to be sure it worked properly! For more information on CONCATENATE, see your disk manual.

A GENERAL APPEND PROCEDURE

Here's a more general procedure you can use to add new data to the end of an existing sequential data file. It is useful on computers without APPEND or CONCATENATE functions, and as a precursor to a program allowing you to edit a sequential file.

1. Open the READ file.
2. Open the temporary file for WRITing.
3. READ the next dataset.
4. Save the STatus (SV = ST).
5. Print the dataset to the new (temporary) file.
6. Check the saved status for end of file.
7. If not end of file, repeat steps 3–6.
8. Enter and test the new data to be added to the end of the temporary file. Include a test for the user to end data entry.
9. Print the new dataset to the temporary file.
10. Repeat steps 8 and 9 until all new data are entered.
11. Close both files.
12. Scratch the old (READ) file.
13. Rename the temporary file.

CHANGING DATA IN A FILE

Remember, you cannot write data into a READ file, and you cannot read from a WRITE file. This means that data already placed in a file cannot be changed easily, but they can be changed. The procedure is straightforward: READ the data, change it, store it in a temporary file, then RENAME the temporary file with the original file's name. A few tricks will be explained as you are guided in writing this program.

The example uses a customer credit file for a small business (Chapter 5 Self-Test question 2). Each dataset consists of three items:

1. five-digit customer number (must have exactly five digits);
2. customer name (twenty characters maximum);
3. customer credit rating (a single digit number 1, 2, 3, 4, or 5).

The program task is to change the credit rating of selected customers, with the user entering the customer number and new credit rating. Below is a typical RUN sequence.

```
run
File name? creditl

Type 'STOP' for customer number if no more changes.
```

```
Customer number ? 13762
Error:  Cust.  # 13762
is not in this file. Check your number and reenter.

Customer number ? 13726

Paleo Mechanics Current credit rating:  3
New credit rating? 4

More changes ? (Press 'Y' for YES or 'N' for NO.)

Customer number  ? 11123

ABC Diaper Service  Current credit rating: 3
New credit rating ? 1

More changes ? (Press 'Y' for YES or 'N' for NO.)

Job completed.

ready.
```

While the procedure outline below is tailored to the particular data structure of the example program, the basic idea is adaptable to data files with different data structures (more or less data items in a dataset, or data in field strings).

1. Open the disk error channel.
2. Open the input (READ) file—the one needing changes.
3. Open the temporary WRITE file.
4. Enter the data item to search for, with data entry checks. Include user option for "no more searches."
5. Read a complete dataset from the file.
6. Test the customer number entered by user against customer number in the dataset read from file in step 5; write rejected datasets to the temporary file.
7. Check for end of file in input file. If not found, return to step 5. If found:
   a. display a message that data search was unsuccessful;
   b. close both files (to reset pointers to beginning of files);
   c. go back to step 2.
8. Display the data item found for the user. Ask user to enter changes. Include data entry checks.
9. Print dataset with new data to temporary (WRITE) file.
10. Print remainder of read file to temporary (WRITE) file.
11. Close both files.

12. Delete the original name from the disk with SCRATCH (umless you wish to keep this file for backup or future reference, in which case either this file or the updated file must be given a different name).
13. RENAME the temporary file to the original name (but see step 12).
14. Give the user the option to repeat the procedure starting at step 2.
15. Close the disk error channel.

The program will be developed a segment at a time, with blanks for you to fill in according to the on-line REMARKS. Below is the introductory module, which you understand by now, followed by the data entry routine with entry checks. Fill in the blanks in lines 350, 360, and 370, to test for RETURN without an entry, correct customer number length (five digits), and for nonnumeric entry.

```
100 rem   credit file editor (1)
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     c$ = cust. #
150 rem     c1$ = cust. #
160 rem     n$ = cust. name
170 rem     r$ = user response
180 rem     r, r1 = credit rating value
190 :
200 rem   files used
210 rem     seq. files: credit (user entered),
     tempfile
220 rem     dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "File name "; f$
265 open 15, 8, 15 : rem disk error channel
270 open 1, 8, 8, f$ + ",s,r" : gosub 810
280 open 2, 8, 2, "tempfile,s,w" : gosub 810
290 :
300 rem   data entry routine
310 print "Enter 'STOP' for customer number if
     no more changes."
320 :
330 print : input "Customer number "; c$
340 if c$ = "stop" or c$ = "STOP" then 780

350 _____: rem no entry

360 _____: rem length

370 _____: rem non-numeric
380:
```

```
350 if c$ = "" then print "Enter a number or ty
    pe 'STOP'." : goto 330
360 if len(c$) <> 5 then print "Entry error.  U
    se 5 digits." : goto 330
370 if val(c$) = 0 then print "Entry error; num
    bers only." : goto 330
```

Now for the interesting part. The program searches through the data file for the customer the user requested.

(a) When searching the data file for the customer and encountering the end of file *without* finding the customer, what should the program do?

(b) Before *another* search is made for a customer in the file, what must be done to the file?

(c) Fill in lines 400 and 430 below.

```
380 :
390 rem   file search routine
400 _____________________________
410 let sv = st
420 if c$ = c1$ then 500 : rem match
430 _____________________________
440 if sv = 0 then 400
450 print "Error: Cust. # " c$
455 print "is not in the file.  Check your numb
    er  and re-enter."
460 close 1 : close 2 : rem reset file  pointer
    s
470 print : goto 270
480 :
```

(d) Why was variable C1$ used in line 400 instead of C$?

(e) If you delete line 460 and then RUN the program, what will happen if an incorrect customer number is entered and then a correct customer number?

(a) Print an error message indicating that the customer was not in the file (see the sample RUN presented earlier). (See lines 400, and 450–455).

(b) Close and reopen the files to reset the file pointers to the beginning of the data (very important!). (See line 460.)

(c)
```
400 INPUT#1, C1$, N$, R
430 PRINT#2, C1$ : PRINT#2, N$ : PRINT#2, R
```

(d) Two values would have been assigned to C$, creating a program error. Note the error message at lines 450–455. Also notice the CLOSE and OPEN to reset the data file pointer to the beginning of the file (line 460).

(e) The end of file check in line 440 will detect nonzero status (end of file mark) in both cases, and the error message will be printed both times, since the customer number cannot be found. You will still be at the end of the file when you start the search for the next customer number, since the file wasn't closed and reopened.

When the file has been searched and the correct customer found, the program prints the customer name on the screen (line 510) as a doublecheck to the computer operator that the correction is being made for the right customer.

```
480 :
490 rem   cust # found, proceed with data entry
500 print
510 print n$ " Current credit rating: " r
520 print : input "New credit rating "; r$
530 let r1 = val(r$)
540 if len(r$) <> 1 then print "Enter one digit
      only." : goto 520
550 if r1 < 1 or r1 > 5 then print "Numbers fro
      m 1 - 5 only." : goto 520
560 :
570 rem  print new info to tempfile
580 print#2, c$ : print$2, n$ : print#2, r1
590 :
```

In line 580, the new customer rating is written into the TEMPFILE, along with the accompanying customer number and name. You have now completed routines to search the original file, and to place old and new data into TEMPFILE.

(a) Considering the location of the file pointer in the READ (input) file, what should the program do next?

(a) Write the remaining data in the input file to the WRITE (output) file (TEMPFILE).

(a) Fill in the blanks in this segment (610, 620, and 630):

```
590 :
600 rem   print remainder of credit file to tem
     pfile
610 _____________________________________
620 _____________________________________
630 _____________________________________
640 if sv = 0 then  610
650 :
660 rem   close files
670 close 1 : close 2
680 :
```

(a)
```
610 input#1, c$, n$, r
620 let sv = st
630 print#2, c$ : print#2, n$ : print#2, r
```

The next program section requires a decision regarding the original file in which corrections were made. If that file is no longer needed, then the program module shown below will do the job, replacing the original file with the temporary file containing the changes. If the original file is to be kept for archives or other purposes, then a different file name must be assigned to the original file and/or to TEMPFILE.

The BASIC instruction for changing the name of a file is RENAME, and has this format:

```
PRINT#15, "RENAME0:newname=oldname"
```

or

```
PRINT#15, "R0:newname=oldname"
```

where the R stands for RENAME and the 0 is the drive number. As with CONCATENATE, you must *not* include *any* extra spaces! Of course, a file must be open to the disk command/error channel, with secondary address 15 prior to using the above:

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

It is convenient to open this as file 15, so it doesn't get confused with other files. If you are typing RENAME in direct mode, the OPEN can be combined with the instruction like this:

```
OPEN 15, 8, 15, "R0:newname=oldname"
```

Again, be sure not to put in extra spaces—they may get read as part of the file name. The use of RENAME for PETs or CBMs with 4.0 BASIC is shown in Appendix 5.

If the file name has been assigned to a string variable, then the string variable may be concatenated with the other information.

(a) Write the PRINT#15 statement to RENAME file "TEMPFILE" to a file name stored in the variable F$.

- - - - - - - - - - - - - - - - - - -

(a) `PRINT#15, "R0:" + F$ + "=tempfile"`

If the new file name is the same as that of a file which already exists on the disk, then an error results, and no changes are made in file names.

An error also results if the old file name does not exist on the disk. Your only indication of this may be a brief period of flashing indicator light (1540, 1541 drives), or a brief flash of the error light (4040, 8050, 2031 drives).

Now consider the next section of the Credit File Editor (1) program.

```
680 :
690 rem   delete original and rename tempfile
700 print#15, "s0:" + f$ : gosub 810
710 print#15, "r0:" + f$ + "=0:tempfile" : gosu
    b 810
720 :
```

(a) What does line 700 do?

(b) Line 710 renames "TEMPFILE" to what?

(c) Why must the instructions in line 700 come before those in line 710?

(a) Line 700 SCRATCHes (erases) the file named in the variable F$.
(b) "TEMPFILE" is renamed to the name stored in F$.
(c) The old file name must be deleted before "TEMPFILE" is RENAMEd, or an error will result (?FILE EXISTS).

The possibility of errors reinforces the necessity of providing a routine to read the disk error channel, display any error messages, and allow the user to decide whether to continue. A sophisticated disk error routine was developed at the end of the last chapter. For now, we will write a short subroutine at line 800 to read the error channel, display the error, and stop if an error is detected.

We cannot emphasize too strongly that you *must* check the disk error channel after every disk operation—especially OPEN, CLOSE, SCRATCH, RENAME, and other file manipulation commands. It is conservative practice to check the error channel after *every* (yes!) disk operation—PRINT#, INPUT#, and GET#, as well as the commands listed above.

In the Credit File Editor program, we've only used the subroutine after the OPEN statements (lines 270 and 280, page 272), and after the disk commands here.

```
800 REM     DISK ERROR CHECK
810 INPUT#15, X, X$, Y, Z
820 IF X < 20 THEN RETURN : REM NO ERROR
830 PRINT "[DOWN][RVS]DISK ERROR: "X$
840 CLOSE15 : END
```

Line 810 reads the information from the error channel. You can use whatever variables you wish; X is the error number, X$ is the error message, and Y and Z contain track and sector numbers. Line 820 returns if there is no error. If there is an error, line 830 prints the message, and line 840 closes all files in the disk drive and stops the program. Note that this does not guarantee that your files will be OK; if you have a WRITE ERROR, closing your file still won't put the information from the buffer(s) onto your diskette. However, this routine works for most common errors.

The remainder of the program is a module that informs the computer whether or not more changes are needed in the file. It could replace lines 310 and 340.

```
720 :
730 rem   continue request
740 print "[clr]" : rem clear screen
750 print "More changes ('Y' for YES or 'N' for
    NO) ?"
760 get r$ : if r$ = "Y" or r$ = "y" then print
    "[clr]" : goto 270
770 if r$ <> "n" and r$ <> "N" then 760
780 print "Job completed."
790 close 1 : close 2 : close 15
799 end
```

If you RUN this program with large files, each change will take considerable disk accessing time. If you enter the data in the original file in customer number order, and also enter the changes in customer number order, the need to repeatedly finish copying, close, and reopen the files after each change is eliminated, reducing computer time.

Again, it is helpful to display some message to your user explaining what is happening, so they don't worry that the program or computer has suddenly "gone to sleep." A minimum message is "Please wait." "Resetting file pointers, please wait a few seconds." may be more informative.

Now that you have worked through this example program, let us make some important observations about sequential data files.

1. They are best used for data that do not change often, since changing data in a sequential file is very time consuming as compared to relative files.
2. Sequential files use disk space efficiently. The exception is grouping data items in strings of predefined length, a technique that defeats the efficient use of disk space by sequential files. The "padding" spaces within the "fielded" string are essentially wasted space.

Here is a complete listing of the credit file change program so that you can see it all at once:

```
100 rem   credit file editor (1)
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     c$ = cust. #
150 rem     c1$ = cust. #
160 rem     n$ = cust. name
170 rem     r$ = user response
180 rem     r, r1 = credit rating value
190 :
```

```
200 rem   files used
210 rem     seq. files: credit (user entered),
    tempfile
220 rem     dataset format: c$, n$, r
230 :
240 rem   initialize files
250 print "[clr]" : rem clear screen
260 print : input "File name "; f$
265 open 15, 8, 15 : rem disk error channel
270 open 1, 8, 8, f$ + ",s,r" : gosub 810
280 open 2, 8, 2, "tempfile,s,w" : gosub 810
290 :
300 rem   data entry routine
310 print "Enter 'STOP' for customer number if
    no more changes."
320 :
330 print : input "Customer number "; c$
340 if c$ = "stop" or c$ = "STOP" then 780
350 if c$ = "" then print "Enter a number or ty
    pe 'STOP'." : goto 330
360 if len(c$) <> 5 then print "Entry error.  U
    se 5 digits." : goto 330
370 if val(c$) = 0 then print "Entry error; num
    bers only." : goto 330
380 :
390 rem   file search routine
400 input#1, c1$, n$, r
410 let sv = st
420 if c$ = c1$ then 500 : rem match
430 print#2, c1$ : print#2, n$ : print#2, r
440 if sv = 0 then 400
450 print "Error: Cust. # " c$
455 print "is not in the file.  Check your numb
    er  and re-enter."
460 close 1 : close 2 : rem reset file  pointer
    s
470 print : goto 270
480 :
490 rem   cust # found, proceed with data entry

500 print
510 print n$ "  Current credit rating: " r
520 print : input "New credit rating "; r$
530 let r1 = val(r$)
540 if len(r$) <> 1 then print "Enter one digit
     only." : goto 520
550 if r1 < 1 or r1 > 5 then print "Numbers fro
    m 1 - 5 only." : goto 520
560 :
570 rem   print new info to tempfile
580 print#2, c$ : print#2, n$ : print#2, r1
590 :
```

```
600 rem   print remainder of credit file to tem
     pfile
610 input#1, c$, n$, r
620 let sv = st
630 print#2, c$ : print#2, n$ : print#2, r
640 if sv = 0 then  610
650 :
660 rem   close files
670 close 1 : close 2
680 :
690 rem   delete original and rename tempfile
700 print#15, "s0:" + f$ : gosub 810
710 print#15, "r0:" + f$ + "=0:tempfile" : gosu
     b 810
720 :
730 rem   continue request
740 print "[clr]" : rem clear screen
750 print "More changes ('Y' for YES or 'N' for
     NO) ?"
760 get r$ : if r$ = "Y" or r$ = "y" then print
     "[clr]" : goto 270
770 if r$ <> "n" and r$ <> "N" then 760
780 print "Job completed."
790 close 1 : close 2 : close 15
791 end
799 :
800 rem    disk error check
810 input#15, x, x$, y, z
820 if x < 20 then return : rem no error
830 print "[down][rvs]Disk error: "x$
840 close15 : rem close all files and quit
850 end
```

(a) Write the corresponding program line number(s) for each step in the outline.

1. Open the disk error channel. _____
2. Open the input (READ) file—the one needing changes. _____
3. Open the temporary WRITE file. _____
4. Enter the data item to search for, with data entry checks. Include user option for "no more searches." _____
5. Read a complete dataset from the file. _____
6. Test the customer number entered by user against customer number in the dataset read from file in step 5. _____

   Write rejected datasets to the temporary file. _____

7. Check for end of file in input file. If not found, return to step 5. ___

   If found:

   a. Display a message that data serach was unsuccessful. _____

   b. Close both files (to reset pointers to beginning of files). _____

   c. Go back to step 2. _____

8. Display the data item found for the user. Ask user to enter changes.

   Include data entry checks. _____

9. Print dataset with new data to temporary (WRITE) file. _____

10. Print remainder of read file to temporary (WRITE) file. _____

11. Close both files. _____

12. Delete the original name from the disk. _____

13. RENAME the temporary file to the original name. _____

14. Give the user the option to repeat the procedure starting at step 2.

    _____

15. Close the disk error channel. _____

(a)
1. 265
2. 270
3. 280
4. 330–370
5. 400
6. 420–430
7. 440 (410)
   a. 450–455
   b. 460
   c. 470 (270)
8. 510, 520–550
9. 580
10. 610–640
11. 670
12. 700
13. 710
14. 740–770
15. 790

Modify the Credit Rating program to change customer names instead of credit rating (companies do change names). Only four program lines are involved. Show them below.

```
490 rem  cust # found, proceed with data entry
500 print
510 print c$ " Current name: " n$
520 print : input "New customer name "; n$
530 let n = len(n$)
540 if n = 0 then print "Enter a name." : goto
      520
550 if n > 20 then print "20 chars. max--reent
      er." : goto 520
560 :
570 rem  print new info to tempfile
580 print#2, c$ : print#2, n$ : print#2, r1
590 :
```

Only four changes were necessary in this modularly designed program. A factor that minimized changes was that the entire data set was dealt with all at once instead of reading one data item at a time. Remember this when writing future programs.

EDITING, DELETING, AND INSERTING SEQUENTIAL FILE DATA

Whenever we work extensively with files, we write a small utility program that lets us read through the file, one item at a time, to verify that everything is as it should be. A properly written sequential data file-editing program also lets you make changes in the file data as it reads through the file. Our example will use the previous application, the CREDIT file. Remember that the dataset consists of:

1. five-digit customer number stored as a string;
2. a twenty-character customer name;
3. a credit rating, stored as a numeric value from 1 to 5.

```
100 rem   press 'return' (credit) file reader
110 rem   using 'press return to continue'
111 :
115 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
116 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
120 :
130 rem   variables used
140 rem     c$ = cust. # (5)
150 rem     n$ = cust name  (20)
160 rem     r  = credit rating
170 rem     r$ = user response
180 rem     f$ = file name
190 :
200 rem   files used
210 rem     seq. file = credit (user entered)
220 rem     dataset format: c$, n$, r
230 :
240 rem   file initialization
250 print : input "File to read "; f$
260 open 1, 8, 8, f$ + ",s,r"
270 :
300 rem   read file and display
310 print "[clr][down][down][down][down]" : rem
     clear scrn & down 4
320 input#1, c$, n$, r
330 print : print "Customer Number: [rvs]" c$
340 print : print "Press 'RETURN' to continue."

350 get r$ : if r$ <> chr$(13) then 350
360 :
370 print : print "  Customer Name: [rvs]" n$
380 print : print "Press 'RETURN' to continue."

390 get r$ : if r$ <> chr$(13) then 350
400 :
410 print : print "  Credit Rating: [rvs]" r
420 print : print "Press 'RETURN' to continue."

430 get r$ : if r$ <> chr$(13) then 350
440 :
450 if st = 0 then 320
460 :
470 rem   close file
480 close 1
490 print : print "File read and closed."
```

(a) What is assigned to R$ in lines 350, 390, and 430?

(b) What is the purpose of lines 330, 360, and 390?

(a) The ASCII value of whatever key is pressed on the keyboard; if no key is pressed, the empty string (""), ASCII 0 is assigned to R$ by the GET.
(b) Wait until the RETURN key is pressed; keep items displayed on the screen until the user chooses to continue.

The next version of the Credit File Editor program allows the user to change any data item displayed on the screen, or to accept it "as is" by pressing the RETURN key to continue. The procedure includes copying the CREDIT data file to a temporary file, "TEMPFILE," as you read through the file making changes. Here is the first part of the program with the capability of changing the customer number.

```
100 rem   credit file editor (2)
110 :
120 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
130 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
140 :
150 rem   type 'c' to change item
160 :
170 rem   variables used
180 rem     c$ = cust # (5)
190 rem     n$ = cust name (20)
200 rem     r  = credit rating
210 rem     r$ = user response
220 rem     f$ = file name
230 :
240 rem   files used
250 rem     seq. files: credit (user entered),
    tempfile
260 rem     dataset format: c$, n$, r
270 :
300 rem   file initialization
310 print "[clr]" : input "File to read "; f$
320 open 15, 8, 15 : rem error channel
330 open 1, 8, 8, f$ + ",s,r" : gosub 2000
340 open 2, 8, 2, "@0:tempfile,s,w" : gosub  20
    00
350 :
```

```
400 rem   read file and display
410 print "[clr][down][down]" : rem clear and d
    own 2
420 print "Press 'C' to change item"
430 print "Press 'RETURN' to accept with no cha
    nge."
440 input#1,  c$,  n$, r
450 let sv = st : rem save status
460 :
500 rem   display and query
510 print : print "Customer number: " c$
515 print "'C' or 'RETURN'"
520 get r$ : if r$ = "" then 520
530 if r$ = "C" or r$ = "c" then gosub 760: got
    o 570
555 if r$ <> chr$(13) then 520
560 :
570 print : print "Customer Name: " n$
575 print "'C' or 'RETURN'"
580 get r$ : if r$ = "" then 580
590 if r$ = "C" or r$ = "c" then gosub 830 : go
    to 610
595 if r$ <> chr$(13) then 580
600 :
610 print : print "Credit Rating: " r
615 print "'C' or 'RETURN'"
620 get r$ : if r$ = "" then 620
630 if r$ = "C" or r$ = "c" then gosub 890 : go
    to 660
635 if r$ <> chr$(13) then 620
640 :
650 rem   print to tempfile
660 print#2, c$ : print#2, n$ : print#2, r : go
    sub 2000
670 if sv = 0 then 410
680 :
750 rem   change cust # routine
760 print : input "New Customer Number "; c$
770 if len(c$) = 0 then print "Enter 5 digit nu
    mber." : goto 760
780 if len(c$) <> 5 then print "Enter exactly 5
     digits." : goto 760
790 if val(c$) = 0 then print "Entry error: num
    bers only." : goto 760
800 return
810 :
2000 rem   disk error channel
2010 input#15, x, x$, y, z    : rem read
2020 if x < 20 then return    : rem check and ret
    urn
2030 print "Disk error " x$ : close 15 : end : r
    em stop on error
```

Notice the few additions to the Credit File Editor program: the output (WRITE) file (lines 250, 340); the instruction changes (lines 150, 420); and the entry test (line 530).

Although we have first displayed the choices, then the dataset, we prefer to display the choices *below* the dataset on the screen. This way, the user first sees the dataset, then is given the choices. Incorrect data entry may scroll the screen, leaving no visible instructions. When using INPUT for the choices, the statement of choices and the INPUT statement should not be separated, so the question mark and flashing cursor of the INPUT statement will be associated with a question. A question mark and cursor by themselves are often confusing to program users, so this is a situation to be avoided. Here, we've used a GET routine, so there is no cursor, and the instructions can be placed where we wish. The change could be made by swapping lines 410–430 with lines 440–510. Note that line 515 is one solution.

For reasons that will become apparent, a subroutine (lines 750–800) is used for entering the change in the customer number. The same data entry checks are used that were originally used in the CREDIT file-creating program.

A caution: this subroutine does *not* print the new customer number to TEMPFILE. In order to maintain identical files, use one statement to print *the entire dataset* into TEMPFILE, as was originally done with the credit rating file-creation program.

You may have noticed that the new customer number was assigned to C$, replacing the old customer number stored there. Can you look ahead and see why?

EDITING DATA (CREDIT FILE EDITOR)

We now present a routine that will allow a change in the customer name, using the format of the subroutine above. Notice lines 830–850 for data entry, and for tests for null string and length of entry. Lines 570–595 are repeated here for your reference.

```
560 :
570 print : print "Customer Name: " n$
575 print "Press 'C' or 'RETURN'."
580 get r$ : if r$ = "" then 580
590 if r$ = "C" or r$ = "c" then gosub 830 : go
    to 610
595 if r$ <> chr$(13) then 580
600 :
810 :
820 rem   change cust name routine
830 print : input "New Customer Name "; n$
840 if len(n$) = 0 then print "Please enter a n
    ame." : goto 830
850 if len(n$) > 20 then print "Too long: 20 ch
    aracters or less." : goto 830
860 return
870 :
```

Next comes a program segment that allows a change to be entered for the credit rating. Upon returning from the subroutine, the program writes the entire dataset, including changes (if any), to TEMPFILE. Examine lines 660 and 890–910. Lines 610–635 are repeated here for your reference.

```
595 if r$ <> chr$(13) then 580
600 :
610 print : print "Credit Rating: " r
615 print "Press 'C' or 'RETURN'."
620 get r$ : if r$ = "" then 620
630 if r$ = "C" or r$ = "c" then gosub 890 : go
    to 660
635 if r$ <> chr$(13) then 620
640 :
650 rem   print to tempfile
660 print#2, c$ : print#2, n$ : print#2, r : go
    sub 2000
670 if sv = 0 then 410
680 :
870 :
880 rem   change credit rating routine
890 print : input "New Credit Rating "; r$
900 if len(r$) <> 1 then print "Enter only a 1
    digit number." : goto 890
910 if val(r$) <1 or val(r$) > 5 then  print "N
    umbers 1 to 5 ONLY." : goto 890
920 return
930 :
```

Did you notice line 660? Carefully planned, the routine that prints to the file uses the same variables C$, N$, and R, that can contain either new data, or old, unchanged data items.

(a) Describe the last routines needed to complete this program.

(a) Routines to close the files, to SCRATCH the original file, to RENAME TEMPFILE to the original file name, and a subroutine to check the disk error channel.

The end of file test in line 670 to branch to the next program segment is already set up:

```
670  if sv = 0 then 410
```

While experiencing a bit of *dejà vu*, complete the final section to rename TEMPFILE to the original file name by filling in lines 710, 720, and 730, and add a disk error channel check subroutine starting at line 2000.

(a)
```
680:
690 rem close routine
700 close 1 : close 2 : gosub 2000
710 _____
720 _____
730 _____
740 print "Job completed." : end
 .
 .
 .
2000 rem disk error check
2010 _____ : rem read
2020 _____ : rem check & return
2030 _____ : rem stop on error
```

(b) What file needs to be closed? Why is it closed in line 730 instead of line 700?

_____

_____

(a)
```
670 if sv = 0 then 410
680 :
690 rem    close routine
700 close  1 : close 2 : gosub 2000
705 print : print "Erasing " f$
710 print#15, "s0:" + f$ : gosub 2000
715 print : print "Renaming 'TEMPFILE' to" f$
720 print#15, "r0:" + f$ + "=tempfile" : gosub
    2000
730 close 15
740 print "Job completed." : end
741 :
749 :
1990 :
2000 rem   disk error channel
2010 input#15, x, x$, y, z    : rem read
2020 if x < 20 then return    : rem check and ret
    urn
2030 print "[down][rvs]Disk error " x$ : close 1
    5 : end : rem stop on error
```

(b) The disk error channel, file #15. It can't be closed until the SCRATCH and RENAME operations are completed, so the disk error subroutine can be used.

DELETING DATA (CREDIT FILE EDITOR)

Yet another desirable editing feature is the ability to delete a complete dataset from a data file. This is in addition to the program's ability to make changes in an existing dataset. To delete a dataset, have the program read the dataset from the input file, but *not* copy it into TEMPFILE. Thus, the dataset "disappears." This editing option can be integrated into the existing program you have seen developing. First, enter a statement to inform the user of the option to delete a dataset:

```
430 print "Press 'D' to delete dataset."
```

Complete the other change (line 540) in a multiple-statement line that tests for the user input 'D' and, if present, branches to line 670, *without* writing this dataset to the file (review the logic behind this part of the program carefully—it's not straightforward—we kept losing data here and there!). Check the context (statements around line 540) before writing line 540. [Refer back to the program "Credit File Editor (2)."]

(a) 530 if r$ = "C" or r$ = "c" then gosub 760 : goto 570

540 _____

(a) 540 if r$ = "D" or r$ = "d" then 670

This skips line 660, so the dataset is *not* printed to the file; since it won't show up in the new file, it is deleted. Line 670 checks for end of file to see if there are more data. If there are, the next dataset is read.

By the way, it's a good idea to be redundant about deletion—to redisplay the entire dataset and ask a second time, "Delete this dataset?" Alternatively, you might ask "OK to delete ('Y' or 'N')?" after the user has pressed 'D'. That way, the user isn't stuck deleting a dataset because of an incorrect keystroke or a misread customer number. We haven't done that in this program. However, it's easy to do. Write the necessary program lines (assuming line 540 is changed to "... then 1000"):

(a) 1000 rem check delete
1010 print " _____"
1020 _____
1030 _____
1040 _____

(a)
```
1000 rem   check delete
1010 print "Delete this dataset (Press [rvs]Y[off] or
      [rvs]N[off]) ?"
1020 get r$ : if r$ <> "y" and r$ <> "n" then 1020
1030 if r$ = "n" then 500 : rem return to original
1040 goto 670 : rem delete by not printing
```

You now have an example of a sequential file-editing program that allows changes in data items and deletion of sets of data. As you will see in the next chapters, these operations are much simpler and much faster with relative files.

INSERTING NEW DATA (CREDIT FILE EDITOR)

Another useful sequential file-editing feature allows you to insert a new dataset *partway through* an existing data file to keep data in numerical or alphabetical order. After locating a certain dataset, the new dataset is then inserted immediately after it.

To insert a new dataset into the file, you can enter the new data with the subroutines used previously to make changes in the file. How's that for program

efficiency! Following are some of the new statements needed, with blanks for you to complete (lines 550, 970, and 980). Check back to "Credit File Editor (2)" for correct line numbers to branch to.

(a)
```
440 print "Press 'I' to insert new dataset after
      this one."

550 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ then 950

940 rem   file insert routine
950 :
960 gosub 760
970 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
980 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
990 goto 660 : rem print to tempfile
```

(a)
```
550 if r$ = "I" or r$ = "i" then 950
940 rem   file insert routine
950 ::::::
960 gosub 760
970 gosub 820
980 gosub 880
990 goto 660 : rem print to tempfile
1000 :
```

Our "policy" is to insert the new dataset *after* the one just displayed. This means you cannot insert data before the first dataset! It also means you must add a new statement to the program in line 950.

(a) What will it do?

_____

_____

(b) Write line 950.

950 _____

(a) Line 950 will print the current dataset to TEMPFILE before a new dataset is entered.

(b)
```
950 print#2, c$ : print#2, n$ : print#2, r
```

The following outline for the final version of the program allows for insertion, deletion, or changes of data in the file.

1. Open the disk error channel.
2. Open the input file.
3. Open the temporary file for writing.
4. Display a "menu" for the user to select changes to be made, including a "no changes" option.
5. Read the entire dataset from the file and display the first data item (not dataset) in the current dataset.
6. Allow the user to enter a selection from the "menu" and test for the selection possibilities.
7. If the user entered "C" for change:
   a. allow the user to enter change with data entry checks;
   b. display the next data item from data file for current dataset (if any remain in dataset);
   c. present options and test user selection;
   d. repeat steps a, b, and c until all items in a dataset have been presented;
   e. print the dataset (with any changes) to the temporary file;
   f. go to step 11.
8. If user entered "I" for insert:
   a. print dataset to the temporary file;
   b. user enters new dataset with data entry checks;
   c. print the newly entered data to the temporary file;
   d. go to step 11.
9. If user entered "D" for delete, go to step 11.
10. If the user just entered 'RETURN,' go to steps 7b–7d.
11. If not end of file, go to step 4.
12. Close both files
13. Delete the original file from the disk.
14. Rename the temporary file with the name of the original program.
15. Close the disk error channel.

Now gather together this data file-editing utility program for the CREDIT file. It allows you to change, delete, insert, or add to the CREDIT data file.

We have also "souped it up" a bit to take advantage of Commodore's screen display features. Notice the use of REVERSE to call attention to displays, and to highlight choices.

```
100 rem   credit file editor (3)
110 :
120 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
130 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
140 :
150 rem   type 'c' to change item
```

```
160 rem   type 'd' to delete item
170 rem   type 'i' to insert item
180 :
190 rem   variables used
200 rem     c$ = cust # (5)
210 rem     n$ = cust name (20)
220 rem     r  = credit rating
230 rem     r$ = user response
240 rem     f$ = file name
250 :
260 rem   files used
270 rem     seq. files: credit (user entered),
    tempfile
280 rem     dataset format: c$, n$, r
290 :
300 rem   file initialization
310 print "[clr][down][down]" : input "File to
    read "; f$
320 open 15, 8, 15 : rem error channel
330 open 1, 8, 8, f$ + ",s,r" : gosub 2000
340 open 2, 8, 2, "@0:tempfile,s,w" : gosub  20
    00
350 :
400 rem   read file and display
410 print "[clr][down][down]" : rem clear and d
    own 2
420 print "Press [rvs]C[off] to change item"
430 print "Press [rvs]D[off] to delete dataset"

440 print "Press [rvs]I[off] to insert new data
    set after   this one."
450 print "Press [rvs]RETURN[off] to accept wit
    h no change."
460 input#1, c$, n$, r
470 let sv = st : rem save status
480 :
500 rem display and query
510 print "Customer number: [rvs]" c$
520 gosub 1110
530 if r$ = "C" or r$ = "c" then gosub 760: got
    o 570
540 if r$ = "D" or r$ = "d" then 670 : rem chec
    k e-o-f
550 if r$ = "I" or r$ = "i" then 950
555 if r$ <> chr$(13) then 520
560 :
570 print "[down]Customer Name: [rvs]" n$
580 gosub 1160
590 if r$ = "C" or r$ = "c" then gosub 830  : g
    oto 610
595 if r$ <> chr$(13) then 580
600 :
```

```
610 print "[down]Credit Rating: [rvs]" r
620 gosub 1160
630 if r$ = "C" or r$ = "c" then gosub 890 : go
    to 660
635 if r$ <> chr$(13) then 620
640 :
650 rem   print to tempfile
660 print#2, c$ : print#2, n$ : print#2, r : go
    sub 2000
670 if sv = 0 then 410
680 :
690 rem    close routine
700 close  1 : close 2 : gosub 2000
705 print "[clr][down][down][down][rvs]Erasing
    file: " f$
710 print#15, "s0:" + f$ : x = 0 : gosub 2000
711 if x > 20 then 710
715 print "[down]Renaming [rvs]TEMPFILE[off] to
     [rvs]" f$
720 print#15, "r0:" + f$ + "=tempfile" : x = 0
    : gosub 2000
721 if x > 20 then 710
730 close 15
740 print "Job completed." : end
741 :
749 :
750 rem   change cust # routine
760 print : input "New Customer Number "; c$
770 if len(c$) = 0 then print "Enter 5 digit nu
    mber." : goto 760
780 if len(c$) <> 5 then print "Enter exactly 5
     digits." : goto 760
790 if val(c$) = 0 then print "Entry error: num
    bers only." : goto 760
800 return
810 :
820 rem   change cust name routine
830 print : input "New Customer Name "; n$
840 if len(n$) = 0 then print "Please enter a n
    ame." : goto 830
850 if len(n$) > 20 then print "Too long: 20 ch
    aracters or less." : goto 830
860 return
870 :
880 rem   change credit rating routine
890 print : input "New Credit Rating "; r$
900 if len(r$) <> 1 then print "Enter only a 1
    digit number." : goto 890
910 if val(r$) <1 or val(r$) > 5 then  print "N
    umbers 1 to 5 ONLY." : goto 890
920 return
930 :
```

```
940 rem   file insert routine
950 print#2, c$ : print#2, n$ : print#2, r
960 gosub 760
970 gosub 820
980 gosub 880
990 goto 660 : rem print to tempfile
1000 :
1100 rem   prompt
1110 print "[rvs]C[off], [rvs]D[off], [rvs]I[off
     ]";
1120 print " or [rvs]RETURN[off]."
1130 get r$ : if r$ = "" then 1130
1140 return
1150 :
1160 print "[rvs]C[off]"; : goto 1120
1990 :
2000 rem   disk error channel
2010 input#15, x, x$, y, z    : rem read
2020 if x < 20 then return    : rem check and ret
     urn
2030 print "[down][rvs]Disk error " x$
2040 close 15 : end
```

Write the corresponding program line number(s) for each step in the outline below, except for item 10, where you are to fill in the blanks inside parentheses.

(a)
1. Open the disk error channel. _____
2. Open the input file. _____
3. Open the temporary file for writing. _____
4. Display a "menu" for the user to select changes to be made, including a "no changes" option. _____
5. Read the entire dataset from the file. _____

   Display the first data item (not dataset in the current dataset. _____
6. Allow the user to enter a selection from the "menu" and test for the selection possibilities. _____
7. If the user entered "C" for change:
   a. allow the user to enter change with data entry checks; _____
   b. display the next data item from data file for current dataset (if any remain in dataset); _____
   c. present options and test user selection; _____
   d. repeat steps a, b, and c until all items in a dataset have been presented;
   e. print the dataset (with any changes) to the temporary file; _____
   f. go to step 11. _____

8. If user entered "I" for insert:
   a. print dataset to the temporary file; _____
   b. user enters new dataset with data entry checks; _____
   c. print the newly entered data to the temporary file; _____
   d. go to step 11. _____
9. If user entered "D" for delete, go to step 11. _____.
10. If the user just pressed 'RETURN' to display the next data item, go to steps ____ to ____.
11. If not end of file (ST = 0) go to step 4. _____
12. Close both files. _____
13. Delete the original file from the disk. _____
14. Rename the temporary file with the name of the original program.
15. Close the disk error channel. _____

(a)
1. 320
2. 330
3. 340
4. 410–450
5. 460–510
6. 520–555
7. a. 760–800
   b. 570
   c. 580–595
   d. (no answer required)
   e. 660
   f. 670 (EOF)
8. a. 950
   b. 960–980
   c. 660
9. 540
10. 7b, 7d
11. 670 (470)
12. 700
13. 710
14. 720
15. 730

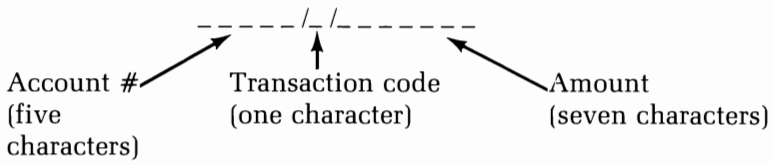
MERGING THE CONTENTS OF TWO SEQUENTIAL FILES

In many business applications of computers, information in data files is maintained in alphabetic or numeric order. This can be done by customer number, customer name, product number, or some other key to filing. It is often necessary or desirable to merge the contents of two data files, both already in some order, to make a third data file with the same order or sequence. A utility program to merge files also allows you to learn some new file programming techniques with wider applications.

Follow these steps to merge two data files into one.

(a)
1. Open the disk error channel.
2. Open the two files that will be merged (1 and 2) as read files. If files don't exist, go to step 14.
3. Open the temporary file (3) for writing the merged data.
4. Read the first dataset from file 1 and store the STatus.
5. Test file 1 for end of file; if found, read a dataset from file 2 and go to step 12.
6. Read first dataset from file 2 and store the STatus.
7. Test file 2 for end of file; if found, go to step 12.
8. Test datasets to see which file dataset (1 or 2) is to be printed to the merge file.
9. Print selected dataset to file 3. This requires two separate routines;
   a. one if file 1 dataset is selected, or
   b. another if file 2 dataset is selected.
10. Check for end of file (nonzero STatus), and read another dataset from whichever file's dataset was printed to file 3 in step 9. Again, the two routines are used:
    a. check for end of file 1; if not, read another dataset from file 1, or
    b. check for end of file 2; if not, read another dataset from file 2.
11. Go to step 8.
12. Copy the remainder of the data in file 1 or 2 to file 3. Again, two routines are needed:
    a. if file 1 is empty, copy the remaining datasets in file 2 to file 3, or
    b. if file 2 is empty, copy the remaining datasets in file 1 to file 3.
13. Close all files.
14. Optional routine to display merged data files for confirmation of a successful merge.
15. Close the disk error channel.

The model program merges two transaction files into a third larger file that combines the other two. In the example, each transaction produces a dataset stored as one thirteen-character string with three fields, as shown below.



Account number = five character field
Transaction code = one-character field (for a bank, 1 = check, 2 = deposit, etc)
Amount = seven-character field.

Assume that the datasets are stored in two data files (TRANSAC1 and TRANSAC2), each in ascending numerical order by account number (Problem 3 in the Chapter 5 Self-Test). The goal is to produce a third file that combines the data in the first two files, but maintains the numerical order when the file merging is complete. Also assume that more than one dataset can have the same account number in either or both data files.

This last assumption requires a decision. When merging, if two datasets have the same account number, the program will copy the dataset from file 1 first, then the dataset with the same number from file 2.

FILE 1	FILE 2
10762	10761
18102	18203
43611	43611
43611	80111
43611	80772
80223	80772
98702	89012

FILE 3 (files 1 and 2 merged into one)

10761
10762
18102
18203
43611
43611
43611
43611
80111
80223
80772
80772
89102
98702

(*Note:* Only the account numbers are shown here; the complete datasets also include transaction codes and amounts.)

This program is called FILE MERGE. It gets tricky, so read the text and program segments carefully. The initializing process is familiar.

```
100 rem   file merge
110 :
111 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
112 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
120 rem   variables used
130 rem     f1$, f2$, f3$ = file names
140 rem     d1$, d2$ = datasets from file 1, 2
150 rem     d1, d2 = account numbers
160 rem     r$, z$ = user response
170 rem   files used
180 rem     seq. read files: transac1, transac2

190 rem     seq. write file: transmerge
200 rem     dataset format: d1$ (concatenated d
    ata)
210 :
220 rem   file initialization
230 open 15, 8, 15 : rem disk channel
240 input "Read file #1 "; f1$
250 input "Read file #2 "; f2$
260 input "Name for merged file "; f3$
270 :
280 open 1, 8, 8, f1$ + ",s,r" : gosub 2000
290 open 2, 8, 2, f2$ + ",s,r" : gosub 2000
300 open 3, 8, 3, "@0:" + f3$ + ",s,w" : gosub
    2000
310 :
```

For program readability, file F1$ uses file 1, file F2$ uses file 2, etc. Remember that you cannot use secondary addresses of 0 and 1, which are reserved for program SAVE and LOAD. Notice that we use secondary address 8 for file 1 to avoid this, but use secondary addresses matching the file number for other files.

```
310 :
320 rem   read file #1
330 input#1, d1$
340 let s1 = st
350 let d1 = val(left$(d1$,5))
360 if s1 <> 0 then input#2, d2$ : let s2 = st
    : goto 640
370 :
```

If file 1 is empty to begin with, we read one item from file 2 and go to 640, storing the status after each INPUT#. Because ST is set to 64 when you read the last datum in a file, the check for end of file must follow PRINTing the datum to the merge file, or you won't be able to copy your last datum.

Line 350 converts the part of the dataset field string that continues the account number into a numeric value.

```
350 let dl = val(left$(dl$,5))
```

You write the next segment. It should read the first data item from file 2, convert the account number part of the dataset string into a numeric value, and then check for end of file, branching to line 690 if the end has been reached.

(a)
```
370 :
380 rem    read file #2
390 _____________________________________________________
400 _____________________________________________________
410 _____________________________________________________
420 _____________________________________________________
430 _____________________________________________________
```

(a)
```
370 :
380 rem   read file #2
390 input#2, d2$
400 let s2 = st
410 let d2 = val(left$(d2$,5))
420 if s2 <> 0 then 690
430 :
```

The next decision is which dataset, that from file 1 or from file 2, will be copied into file 3 first?

```
430 :
440 rem   test for merge
450 if d1 = d2 then 500
460 if d1 < d2 then 500
470 goto 570
480 :
```

The program so far

```
100 rem   file merge
110 :
111 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
112 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
120 rem   variables used
130 rem     f1$, f2$, f3$ = file names
140 rem     d1$, d2$ = datasets from file 1, 2
150 rem     d1, d2 = account numbers
160 rem     r$, z$ = user response
170 rem   files used
180 rem     seq. read files: transac1, transac2

190 rem     seq. write file: transmerge
200 rem     dataset format: d1$ (concatenated d
    ata)
210 :
220 rem   file initialization
230 open 15, 8, 15 : rem disk channel
240 input "Read file #1 "; f1$
250 input "Read file #2 "; f2$
260 input "Name for merged file "; f3$
270 :
280 open 1, 8, 8, f1$ + ",s,r" : gosub 2000
290 open 2, 8, 2, f2$ + ",s,r" : gosub 2000
300 open 3, 8, 3, "@0:" + f3$ + ",s,w" : gosub
    2000
310 :
320 rem   read file #1
330 input#1, d1$
340 let s1 = st
350 let d1 = val(left$(d1$,5))
360 if s1 <> 0 then input#2, d2$ : let s2 = st
    : goto 640
370 :
380 rem   read file #2
390 input#2, d2$
400 let s2 = st
410 let d2 = val(left$(d2$,5))
420 if s2 <> 0 then 690
430 :
440 rem   test for merge
450 if d1 = d2 then 500
460 if d1 < d2 then 500
470 goto 570
480 :
```

Examine lines 450 and 460.

(a) What would happen in the program at line 500?.

(b) The program tests for equality in line 450. In line 460, the test was for D1 less than D2. If both tests are false, then what is the relationship between D1 and D2?

(c) What should happen in the program routine at line 570 (the GOTO from line 470)?

(a) Copy the contents of D1$ to file 3.
(b) D1 is greater than D2.
(c) Copy the contents of D2$ into file 3.

Continue with the file-copying segment for copying a dataset from file 1 to file 3.

```
480 :
490 rem   copy #1 to merge file
500 print#3, d1$
```

(a) After executing the above segment, the program should now read another dataset from file 1. You might want to have the program branch back to the routine at line 320 and continue executing from there. Why would this result in a program error?

(a) The routine at 320 reads from file 1, but then goes right on to read another dataset from file 2, replacing the dataset already assigned to D2$ without it having been copied to file 3.

Additional statements for this program segment are needed to read the next data item from file 1:

```
480 :
490 rem   copy #1 to merge
500 print#3, d1$ : print d1$ : gosub 2000
510 if s1 <> 0 then 640
520 input#1, d1$ : let s1 = st
530 let d1 = val(left$(d1$,5))
540 goto 450
550 :
```

(a) When the program finds the end of file 1, it branches to line 640. Think ahead. What should happen in the routine at line 640?

(a) Since all datasets have been read from file 1 and copied to file 3, all the remaining data in file 2 should be copied into file 3. You'll see this routine soon.

Here is the routine we need to copy a dataset from file 2 to file 3:

```
550 :
560 rem   copy #2 to merge
570 print#3, d2$ : print d2$ : gosub 2000
580 if s2 <> 0 then 690
590 input#2, d2$ : let s2 = st
600 let d2 = val(left$(d2$,5))
610 goto 450
620 :
```

Notice how carefully you must think through these file utility programs. You are nearing the end with a few more "cleanup" routines needed. Two similar routines are needed to copy or dump the remainders of file 2 to 3, and 1 to 3.

```
630 rem   write remainder # 2 to merge
640
```

The program branches to the routine begun just above from line 510, where the program had just finished copying a file 1 dataset into file 3. When the end of file is detected, it shows that all data in that file have now been copied to file 3. The program branches to line 640, and all datasets remaining in file 2 must be copied into file 3.

(a) Since a dataset has been assigned to D2$, what must happen at line 640?

- - - - - - - - - - - - - - - - - - -

(a) Copy the contents of D2$ to file 3.

The rest is easy. Check file 2 for end of file, and dump any remaining datasets to file 3.

```
620 :
630 rem   write remainder #2 to merge
640 print#3, d2$ : print d2$ : gosub 2000
650 if s2 <> 0 then 760
660 input#2, d2$ : let s2 = st
670 goto 640
680 :
```

Write the corresponding routine to copy the remainder of file 1 to file 3. The check for end of file should branch to line 760.

(a) 690 rem write remainder #1 to merge

700 _____

710 _____

720 _____

730 _____

740 :

(b) The end of file statements in lines 650 and 710 branch to line 760. What final routine should appear there?

- - - - - - - - - - - - - - - - - - -

(a)
```
680 :
690 rem   write remainder #1 to merge
700 print#3, d1$ : print d1$ : gosub 2000
710 if s1 <> 0 then 760
720 input#1, d1$ : let s1 = st
730 goto 700
740 :
```

(b) Close all files, since all data have been copied and merged.

Once the files are closed, the program gives the user the option to display the contents of the merged files to verify that it did happen and to judge whether the program works properly. In FILE MERGE all the activity takes place between the computer memory and the disk, with no evidence of the action appearing on the screen. You see only RUN and READY., so did it really happen? The routine included at the end of the complete listing of FILE MERGE lets you be sure (see lines 790 through 900).

In writing the final listing, we have added print statements to display the data being merged onto the screen, so the user will know that something is happening. A "print d$" was simply added to each line with a PRINT#3 (print to merge file) statement. Also note the exemplary use of the disk error subroutine after *every* disk access.

```
100 rem   file merge
110 :
111 if peek(50003) <> 0 then poke 59468,14 : re
    m pet lower case
112 if peek(50003) = 0 then poke 53272,23 : rem
     c-64 lower case
120 rem   variables used
130 rem     f1$, f2$, f3$ = file names
140 rem     d1$, d2$ = datasets from file 1, 2
150 rem     d1, d2 = account numbers
160 rem     r$, z$ = user response
170 rem   files used
180 rem     seq. read files: transac1, transac2

190 rem     seq. write file: transmerge
200 rem     dataset format: d1$ (concatenated d
    ata)
210 :
220 rem   file initialization
230 open 15, 8, 15 : rem disk channel
240 input "Read file #1 "; f1$
250 input "Read file #2 "; f2$
260 input "Name for merged file "; f3$
270 :
280 open 1, 8, 8, f1$ + ",s,r" : gosub 2000
290 open 2, 8, 2, f2$ + ",s,r" : gosub 2000
300 open 3, 8, 3, "@0:" + f3$ + ",s,w" : gosub
    2000
310 :
320 rem   read file #1
330 input#1, d1$ : gosub 2000
340 let s1 = st
350 let d1 = val(left$(d1$,5))
360 if s1 <> 0 then input#2, d2$ : let s2 = st
    : gosub 2000 : goto 640
370 :
380 rem   read file #2
390 input#2, d2$ : gosub 2000
```

```
400 let s2 = st
410 let d2 = val(left$(d2$,5))
420 if s2 <> 0 then 690
430 :
440 rem   test for merge
450 if d1 = d2 then 500
460 if d1 < d2 then 500
470 goto 570
480 :
490 rem   copy #1 to merge
500 print#3, d1$ : print d1$ : gosub 2000
510 if s1 <> 0 then 640
520 input#1, d1$ : let s1 = st : gosub 2000
530 let d1 = val(left$(d1$,5))
540 goto 450
550 :
560 rem   copy #2 to merge
570 print#3, d2$ : print d2$ : gosub 2000
580 if s2 <> 0 then 690
590 input#2, d2$ : let s2 = st : gosub 2000
600 let d2 = val(left$(d2$,5))
610 goto 450
620 :
630 rem   write remainder #2 to merge
640 print#3, d2$ : print d2$ : gosub 2000
650 if s2 <> 0 then 760
660 input#2, d2$ : let s2 = st : gosub 2000
670 goto 640
680 :
690 rem   write remainder #1 to merge
700 print#3, d1$ : print d1$ : gosub 2000
710 if s1 <> 0 then 760
720 input#1, d1$ : let s1 = st : gosub 2000
730 goto 700
740 :
750 rem   close files
760 close 1 : close 2 : close 3 : gosub 2000
770 print : print "Merge completed."
780 print : print
790 print "Would you like to see contents of th
     e   merged file ? ('Y' or 'N')"
800 get r$ : if r$ <> "y" and r$ <> "n" then 80
     0
810 if r$ = "n" then end
820 :
830 rem   print contents of merged file
840 open 3, 8, 3, f3$ + ",s,r" : gosub 2000
850 input#3, d3$ : gosub 2000
860 print left$(d3$,5), mid$(d3$,6,1), right$(d
     3$,7)
870 if st = 0 then 850
880 :
890 rem   close file
```

```
900 close 3 : gosub 2000
910 close 15 : print "Done."
920 end
930 :
2000 rem   disk error channel
2010 input#15, x, x$, y, z    : rem read
2020 if x < 20 then return    : rem check and ret
     urn
2030 print "[down][rvs]Disk error " x$ : close 1
     5 : end : rem stop on error
```

The following routine can be substituted for the routine to display the merged file to allow the user to print the contents of the merged file to a printer. Note the necessity of setting a variable (sv) equal to the status immediately following the INPUT# (line 850), so output to the printer doesn't keep the program from detecting the end of the file. (If you forget that step, your printer will print the last data item in the file until you shut it off.)

```
780 print : print
790 print "Would you like to see contents of th
     e   merged file ? ('Y' or 'N')"
800 get r$ : if r$ <> "y" and r$ <> "n" then 80
     0
810 if r$ = "n" then end
820 :
830 rem   print contents of merged file to prin
     ter
835 open4,4
837 print#4, f1$, f2$, f3$
840 open 3, 8, 3, f3$ + ",s,r" : gosub 2000
850 input#3, d3$ : let sv = st
860 print#4, left$(d3$,5), mid$(d3$,6,1), right
     $(d3$,7)
870 if sv = 0 then 850
875 print#4 : close4
880 :
890 rem   close file
900 close 3 : gosub 2000
910 close 15 : print "Done."
920 end
```

Write the corresponding program line number(s) for each step of the following outline:

(a) 1. Open the disk error channel. _____

2. Open the two files that will be merged (1 and 2) as read files. If files don't exist, go to step 14. _____

3. Open the temporary file (3) for writing the merged data. _____

4. Read the first dataset from file 1. _____

5. Test file 1 for end of file; if found, read a dataset from file 2 and go to step 12. _____
6. Read first dataset from file 2. _____
7. Test file 2 for end of file; if found, go to step 12. _____
8. Test datasets to see which file dataset (1 or 2) is to be printed to the merge file. _____
9. Print selected dataset to file 3. This requires two separate routines:
   a. one if file 1 dataset is selected _____, or
   b. another if file 2 dataset is selected. _____
10. Check for end of file (nonzero status), and read another dataset from whichever file's dataset was printed to file 3 in step 9. Again, the two routines are used:
    a. check for end of file 1; if not, read another dataset from file 1. _____, or
    b. check for end of file 2; if not, read another dataset from file 2. _____
11. Go to step 8. _____
12. Copy the remainder of the remaining data in file 1 or 2 to file 3. Again, two routines are needed:
    a. if file 1 is empty, copy the remaining datasets in file 2 to file 3 _____, or
    b. if file 2 is empty, copy the remaining datasets in file 1 to file 3. _____
13. Close all files. _____
14. Optional routine to display merged data files for confirmation of a successful merge. _____
15. Close the disk error channel. _____

(a)
1. 230
2. 280, 290
3. 300
4. 330
5. 360
6. 390
7. 420
8. 450–470
9. a. 500–540
   b. 570–610

10. a. 510–520
    b. 580–590
11. 540 or 610 (410)
12. a. 640–670
    b. 700–730
13. 760
14. 790–900
15. 910

PROBLEMS WITH SEQUENTIAL DATA FILES

You should be aware of some frequent errors made in using sequential files and some programming techniques used for successful programs accessing data files.

The most frequent programming error is failing to keep track of the file pointers. With Commodore computers, where the computer distinguishes between sequential input and output files at the time the file is opened, the problem is usually greater with input files. Each time you use an INPUT# statement in a program, ask yourself how the file pointer is affected and where it is located before and after executing the statement.

(a) How can you reset the data file pointer to the beginning of a file? _____

(a) Close the file. Pointer is at beginning of file when file is reopened.

Another frequent error occurs when a program sequentially searches through a data file for a particular dataset or data item. Let's say you have a data file of names arranged alphabetically by last name. After you enter the name to be searched, the program searches through the file until it finds the name and then prints the information on your printer for that person. Then you enter a second name. When writing the program, ask yourself where the file pointer will be located after the first search.

Assume the first name searched and located is DORIAN SCHMIDT and the second name is HAMILTON ANDERSON. The data file search for the second name takes up where the search for the first name left off. The second name obviously will not be found before you reach the end of file. If the data file pointer was not reset to the beginning of the file after the first search, ANDERSON will never be found because the file was in alphabetical order and the search for the second name started at SCHMIDT.

The solution, of course, is to make sure the program resets the pointer to the beginning of the file after every search, by using a CLOSE followed by an OPEN statement.

(a) When a file has been partially read through during a data search, why must the file pointer be reset to the beginning of the file before a new search of the file commences?

(a) Because if the pointer is midway in the file and the new datum searched for is near the beginning of the file, the search would not find the datum.

Another program situation to watch out for is a file application program containing routines that sequence through a data file using one INPUT# statement in a program loop. Think through this situation carefully when writing a program. Go through the loop in your mind or on paper two or three times before running the program to make certain the program will perform as expected. You may find (as we did in the MERGE program) that the first INPUT# statement cannot be part of a file reading loop without data being lost. Your program may need two different INPUT# statements, in separate routines. One may read only the first dataset or data item, while the other reads the rest of the file, or as far as necessary, in a program loop routine. This is common.

(a) How could data be lost when all data file inputs are included in data file reading loops? _____

(a) In cases where one file or the other is empty, or where one file contains more data than another.

Errors also occur when the contents of arrays are copied into a data file, a topic not covered earlier. The contents of a one- or two-dimensional array can be copied into a file or read from a file back into an array, provided you use the correct programming techniques. Such data manipulation has many uses. There is a tendency to think of array data as something that is used up or transient, but storing array data in a file gives it permanence.

To load data into a data file from a one-dimensional array:

P (1)	1761	The wrong way:
(2)	18	`PRINT#1, P`
(3)	1942	The correct procedure:
(4)	24	`FOR X = 1 TO 6`
(5)	8209	`PRINT#1, P(X)`
(6)	2	`NEXT X`

The correct procedure could also be done as one multiple-statement line. Similarly, to load array data into a data file from a two-dimensional array:

C

	(1,1)	(1,2)	(1,3)
(1,1)	A	C	P
(2,1)	N	M	S
(3,1)	G	H	T
(4,1)	B	D	E

The wrong way:

```
PRINT#1, C(X,Y)
```

The correct procedure:

```
FOR X = 1 TO 4
  FOR Y = 1 TO 3
    PRINT#1, C(X,Y)
  NEXT Y
NEXT X
```

(a) To read data into (or out of) an array from (or to) a data file, what programming technique is used? _____

(a) FOR NEXT loop.

The last source of errors is the use (or misuse) of Commodore's STatus marker. As noted earlier, ST is changed to 64 when the last datum is read from the file. If you check the STatus *before* printing or otherwise using the value read, you will lose the last datum from your file. With complicated file management programs, you have to think very carefully about when to check the STatus to detect the end of file mark. In addition, because the status is reset after *every* input or output operation, you generally will have to save the status value in another variable. Obviously, you must then test that variable, not ST. Alas, it's a point that is easy to overlook as we have!

(a) In the following example, determine when to check for end of file (and for the end of which file!):

> This program reads addresses from file 1 and replaces addresses in a concatenated dataset from file 2. If there are fewer addresses in file 1, then the remainder of file 2 is copied to the new file.

a. read data from file 1;
b. read data from file 2;
c. insert address from file 1 into dataset from file 2;
d. print new dataset to file 3;
e. copy datasets from file 2 to file 3.

(b) How many variables will be needed to check the status? For which files?

_____

_____

_____

(a) Between d and e. If file 1 is empty, then proceed with e. If file 2 is empty, then stop.
(b) You would need two separate end of file checks, and therefore two separate variables storing the status: one for the status of file 1 and the other for the status of file 2.

A LETTER-WRITING PROGRAM

The next sequential file application example is a letter-writing program you may find useful in your home or business. It illustrates one way in which form letters can be generated. This application presents some new techniques, and reviews others.

Assume that you wrote the Address File-Creating Program in Chapter 5, and that you did the Chapter 5 Self-Test, Problem 4, and have three form letters stored in data files called LETTER1, LETTER2, and LETTER3. When these letters are printed, you want the program to put the inside address and salutation in the letter from data located in yet another sequential data file called ADDRESS. The file ADDRESS contains the names and addresses in the mailing list. The data have the format shown below, with each dataset containing five items in fields within one string.

```
                                                    55
/1          20/21          40/41          50/ 12 /53          57/
 name        address        city          state   zip code
```

The salutation for each letter will be:

Dear resident of *(name of city)*

To print the letters on your printer, you must change the PRINT statements to PRINT# statements. (CMD doesn't work well, because any input or output operation, including INPUT, INPUT#, GET, and disk operations, terminates it; you would have to add many CMD statements to this program to make it work.) This gives us a chance to demonstrate another trick: writing a program that allows the user to select display on the CRT or on the printer.

The program uses the screen to enter which form letter (1, 2, or 3) you want to send to each name on the mailing list. This program, then, uses four data files (only two data files at a time), a printer, and a screen. If you don't have a printer, the program is easily adapted to have all the program output displayed on a screen. Some interesting techniques can be learned from this example.

Follow these steps for this particular program:

1. Open disk error channel
2. Open the address data file for reading.
3. Input the address dataset and display the name.

4. Allow user to select the form letter to be used with this name, with data entry checks.
5. Open selected form letter file for reading.
6. Print the inside address (dataset from address file).
7. Print salutation with addressee's city of residence.
8. Read one dataset (one line of text from form letter file) and print it.
9. Check for end of letter file; if not found repeat steps 8 and 9.
10. Check for end of address file. If not found, go back to step 3.
11. Close both files and the disk error channel.

Look at the introductory module of the program. The ADDRESS file is opened and, as indicated in the line 250 remark, the LETTER files are user selected and opened when selected.

```
100 rem   letter writing program
110 :
120 :
130 rem   variables used
140 rem     n$ = name, address, etc.
150 rem     r$ = response string
160 rem     t$ = letter text string
170 rem     f$ = letter file name
180 rem   files used
190 rem     seq. files: letter1, letter2, lette
    r3, address
200 rem     dataset formats: letter#-r$; addres
    s-n$ (concatenated data)
210 :
220 rem   file initialization
230 open 1, 8, 8, "address,s,r"
240 :
250 rem   form letter file is user selected and
      opened in line 380
260 :
270 rem   read name & address
280 input#1, n$ :  let s1 = st
290 :
```

The program assigns the first name and address dataset string to variable N$ in line 280.

Now it's your turn. Have the program display the party's name on the screen, and then ask the user to select the letter to be printed, to this party, like this:

```
RUN
JERALD R. BROWN

LETTER NUMBER FOR THIS NAME? 1
```

Fill in lines 320, 330 and 340. Line 340 should test the input value for 1 to 3, corresponding to LETTER1, LETTER2, and LETTER3.

(a)
```
290 :
300 REM    DISPLAY NAME/ LETTER REQUEST
310 PRINT "#" : REM CLEAR SCREEN
320 _____________________________________________
330 _____________________________________________
340 _____________________________________________
350 :
```

```
290 :
300 rem   display name/letter request
310 print "[clr][down][down]" : rem clear & crs
     r down 2
320 print left$(n$,20) : print
330 print : input "Letter number for this name
     "; f$
340 if val(f$) < 1 or val(f$) > 3 then print "L
     etters 1 - 3 only." : goto 330
350 :
```

Examine the following routine for creating the name of an existing data file:

```
350 :
360 rem   initialize letter file
370 let f$ = "letter" + f$
380 open 2, 8, 2, f$ + ",s,r"
390 :
```

(a) If the user enters 2, in response to "WHICH LETTER?", what file name is created and assigned to F$?

_____

(a) LETTER2 (note the string concatentation in line 370).

Write the inside address printing statements (to be printed by the printer OPENed as file number 4). Fill in lines 430, 440, and 450.

```
390 :
400 rem   print inside address
410 open 4,4 : rem printer
420 print#4 : print#4 : print#4 : rem blank lin
     es
```

_____

```
430 ------------------------------------
440 ------------------------------------
450 ------------------------------------
460 :
```

```
390 :
400 rem   print inside address
410 open 4,4 : rem printer
420 print#4 : print#4 : print#4 : rem blank lin
     es
430 print#4, left$(n$,20)
440 print#4, mid$(n$,21,20)
450 print#4, mid$(n$,41,10) "," mid$(n$,51,2) "
     ," right$(n$,5)
460 :
```

This next routine prints the salutation. Notice how the city name is extracted from N$ in line 490.

```
460 :
470 rem   print salutation
480 print#4 : print#4 : rem blank lines
490 print#4, "Dear Resident of " mid$(n$,41,10)
      ","
500 :
```

(a) For practice, write a BASIC statement that would print this alternate salutation: Hello, all you folks at *(street address)*

(a) `print#4, "Hello, all you folks at "; mid$(n$,21,20)`

The next routine to print the text of the letter is fairly straightforward. The data input loop continues until that file's data are exhausted.

```
510 rem   print letter text
520 input#2, t$
530 let s2 = st
540 print#4, t$
550 if s2 = 0 then 520
560 :
570 rem   close letter file, get next address
580 print#4 : close 4  : rem close printer
590 close 2
600 if s1 = 0 then 280 : rem next addr
610 :
```

Unless you have an unusual situation, the carriage returns at the end of print#4 statements to your printer will match the carriage returns written as end of data items to your disk file, and the file will print just as they were typed. [If, for some reason, your printer insists on printing everything on the same line, change the file number OPENed to the printer to a number greater than 128, which will send linefeeds with the carriage return. If, on the other hand, your printer insists on double spacing, you may have been sending linefeed characters along with carriage returns at the end of each PRINT#. In this case, you'll need to go back and add ;CHR$(13); to the end of every PRINT# statement (refer to Appendix 4—"Files in BASIC 1.0/2.0").]

(a) Give two reasons for closing the letter file in line 580.

_____

_____

(b) Without checking back, what happens in line 280, which is branched to from line 600 (600 if st = 0 then 280)?

_____

_____

(a) Resets the pointer so that the letter can be used again, and only one OPEN statement is needed for all letter files.
(b) The next name and address dataset is read.

And now, you write the last routine necessary to properly complete this program by completing line 620.

(a)
```
610 :
620 rem   close address file
630 _____________________________________________
640 print "Job completed."
```

(a)
```
610 :
620 rem   close address file
630 close 1
640 print "Job completed."
```

If you do not have a printer, you can change all PRINT#4 statements to PRINT, and delete the OPEN 4 and CLOSE 4 statements. You will need to add the following lines so each dataset can be read from the screen:

```
585 print "[down]Press [rvs]RETURN[off] for next
      name."
586 get r$ : if r$ <> chr$(13) then 586
```

Better yet, you can allow the user to select output to the screen or printer. Here's the routine allowing you your choice of screen or printer:

```
100 print "Printer or Screen (p/s) ?"
110 get z$ : if z$ <> "s" and z$ <> "p" then 11
    0
120 d = 3 : if z$ = "p" then d = 4
130 open 4,d : rem open file
140 for i = 1 to 50
150 print#4, i, i*i
155 if d = 4 then 200
160 rem wait and clear screen
170 print "Press 'RETURN' to continue."
180 get z$ : if z$ <> chr$(13) then 180
190 print "[clr]" : rem clear
200 next i
210 print#4 : close 4
220 end
230 print "[clr]" : return : rem clear
```

Following is a complete listing of the letter-writing program:

```
100 rem   letter writing program
110 :
120 :
130 rem   variables used
140 rem     n$ = name, address, etc.
150 rem     r$ = response string
160 rem     t$ = letter text string
170 rem     f$ = letter file name
180 rem   files used
190 rem     seq. files: letter1, letter2, lette
    r3, address
200 rem     dataset formats: letter#-r$; addres
    s-n$ (concatenated data)
210 :
220 rem   file initialization
230 open 1, 8, 8, "address,s,r"
240 :
250 rem   form letter file is user selected and
      opened in line 380
260 :
270 rem   read name & address
280 input#1, n$ :  let s1 = st
290 :
300 rem   display name/letter request
310 print "[clr][down][down]" : rem clear & crs
    r down 2
320 print left$(n$,20) : print
330 print : input "Letter number for this name
    "; f$
```

```
340 if val(f$) < 1 or val(f$) > 3 then print "L
    etters 1 - 3 only." : goto 330
350 :
360 rem   initialize letter file
370 let f$ = "letter" + f$
380 open 2, 8, 2, f$ + ",s,r"
390 :
400 rem   print inside address
401 print "[rvs]S[off]creen or [rvs]P[off]rinte
    r ?"
402 get r$ : if r$ <> "s" and r$ <> "p" then 40
    2
403 dn = 3 : if r$ = "p" then dn = 4
410 open 4,dn : rem printer
420 print#4 : print#4 : print#4 : rem blank lin
    es
430 print#4, left$(n$,20)
440 print#4, mid$(n$,21,20)
450 print#4, mid$(n$,41,10) "," mid$(n$,51,2) "
    ," right$(n$,5)
460 :
470 rem   print salutation
480 print#4 : print#4 : rem blank lines
490 print#4, "Dear Resident of " mid$(n$,41,10)
      ","
500 :
510 rem   print letter text
520 input#2, t$
530 let s2 = st
540 print#4, t$
550 if s2 = 0 then 520
560 :
570 rem   close letter file, get next address
580 print#4, chr$(140) : close 4  : rem form fe
    ed & close printer
590 close 2
600 if s1 = 0 then 280 : rem next addr
610 :
620 rem   close address file
630 close 1
640 print "Job completed."
```

Chapter 6 Self-Test

1. Write a program to make a copy of the ADDRESS file that you created at the end of Chapter 4 (before the self-test), and used in the letter-writing program. Name the copy file ADDRESS.CPY. A RUN of the program should look like this:

```
run

Name for copy file ? address.copy

Address file copy completed with name
address.copy.

ready.
```

Use the program from the end of Chapter 5 (just before the Self-Test) to read and display ADDRESS.COPY for verification of an accurate copy. The introductory module for the copying program is shown below.

```
100 REM   PROB 6-1
110 :
120 REM   VARIABLES USED
130 REM   N$ = NAME (20)
140 REM   A$ = ADDRESS (20)
150 REM   C$ = CITY (10)
160 REM   S$ = STATE (2)
170 REM   Z$ = ZIP CODE (5)
180 REM   D$ = ENTIRE DATASET (57)
190 REM   F$ = COPY FILE NAME
200 :
210 REM   SEQUENTIAL FILE NAMES: ADDRESS, ADDRESS.CPY (USER ENTERED)
220 REM   DATASET FORMAT: D$ (CONCATENATED DATASET)
230 :
240 REM   INITIALIZE
```

2. (a) Write a program that can create a sequential data file whose data items are the titles of computer magazines. Create the files shown below, and enter the items in each file in alphabetical order.

```
File 1                File 2

Byte                  Business Computer Systems
Compute!              Creative Computing
Dr. Dobbs Journal     Datamation
Infoworld             Interface Age
Microcomputing        Microsystems
PC World              Personal Computing
Popular Computing     Recreational Computing
```

Use the program to create two separate files. Use MAGLIST1 and MAGLIST2 as the file names. Maintain alphabetical order of the data items within each file. The introductory module of the program follows the beginning of a program RUN.

```
run

File name ? maglistl

If done, type 'STOP'.

Magazine name ? stop

File 'maglistl' closed.
ready.
```

```
100 REM   PROB 6-2A
110 :
120 REM   VARIABLES USED
130 REM     F$ = FILE NAME
140 REM     M$ = DATA (MAGAZINE)
150 REM     R$ = USER RESPONSE
160 :
170 REM   FILES USED
180 REM    SEQ. FILE NAMES: MAGLIST1, MAGLIST2
190 REM    DATASET FORMAT: M$ (MAGAZINE TITLES)
200 :
210 REM   INITIALIZATION
```

2. (b) Write a program that can display the contents of the user-selected file of magazine titles, including either MAGLIST1 or MAGLIST2. Use the program to verify the contents of the files mentioned. A sample RUN looks like this:

```
run

File name ? maglist2

Business Computer Systems
Creative Computing
Datamation
Interface Age
Microsystems
Personal Computing
Recreational Computing

File 'maglist2' read & closed.
ready.
```

```
100 rem   prob 6-2b solution--read & display ma
    glist# files
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     m$ = data (magazine)
150 rem     r$ = user response
160 :
170 rem   files used
180 rem     seq: maglist1, maglist2
190 rem     dataset: m$ (mag. titles)
200 :
210 rem   initialization
220 print : input "File name "; f$
```

2. (c) Write a program to merge into one alphabetically organized sequential data file the contents of MAGLIST1 and MAGLIST2. These two files should have their own data organized alphabetically within each file. Name the merged file MAGMERGE.

Include a routine at the end of this program that gives the user the option to automatically display MAGMERGE to verify a successful and complete merge. Refer back to this chapter for guidelines to organizing your program.

Here is a RUN of the magazine title merging program.

```
run

Input file #1 ? maglist1

Input file #2 ? maglist2

Working...merge in progress

Merge completed.

Would you like to see the merged file ?
(Press 'Y' or 'N')
Business Computer Systems
Byte
Compute!
Creative Computing
Datamation
Dr. Dobbs Journal
Infoworld
Interface Age
Microcomputing
Microsystems
PC World
Personal Computing
Popular Computing
Recreational Computing

File displayed and closed.
ready.
```

Write your program following the introductory module.

```
100 REM    PROB 6-2C
110 :
120 REM   VARIABLES USED
130 REM     F1$,F2$,F3$ = FILE NAMES
140 REM     D1$,D2$ = DATA FROM FILES 1,2
150 REM     R$ = USER RESPONSE
160 :
170 REM   FILES USED
180 REM     SEQ. FILE NAMES: MAGLIST1, MAGLIST2, MAGMERGE
190 REM     DATASET FORMAT: D1$ (ONE STRING)
200 :
210 REM   INITIALIZATION
220 :
```

3. Finish writing a program that allows you to enter into a data file named WORKLIST a list of household maintenance tasks to be done. This program is designed so that you can create a new WORKLIST file if none exists, and either add new tasks or delete tasks already on file. Examine the two sample RUNs and the program segments provided.

   You are to write the following modules to complete the program:

   1. Add data items to file;
   2. Delete data items from the file;
   3. Rename TEMPFILE (with the changes) to WORKLIST.

   The first sample RUN shows the file being created for the first time.

```
run

Create or edit 'WORKLIST'
File name ? worklist
'N'ew or 'O'ld file (Press 'N' or 'O')
'A'dd or 'D'elete data (Press 'A' or 'D')
New task or 'STOP' ? Wash windows

New task or 'STOP' ? Fix back steps
New task or 'STOP' ? Change oil in car
New task of 'STOP' ? Fix leak under kitchen sink
New task or 'STOP' ? Proofread Chapter 7 in Data
Files book
New task or 'STOP' ? STOP
File updated and closed.
ready.
```

This second sample RUN shows the task you have finished taken out of the file (apparently you proofread the chapter).

(Since the keystrokes for choices are picked up by a GET routine, they won't be displayed on the screen unless you add a PRINT statement to print them; here they are shown in [ ].)

```
run


Create or edit 'WORKLIST'

File name ? worklist

'N'ew or 'O'ld file (Press 'N' or 'O') [O]

'A'dd or 'D'elete data (Press 'A' or 'D') [D]

Press 'RETURN' to accept data displayed.Press 'D' to delete.

Wash windows
'RETURN' or 'D'elete [RETURN]

Fix back steps
'RETURN' or 'D'elete [RETURN]

Change oil in car
'RETURN' or 'D'elete [RETURN]

Fix leak under kitchen sink
'RETURN' or 'D'elete [RETURN]

Proofread Chapter 7 in Data Files book
'RETURN' or 'D'elete [D]

File updated and closed.

ready.
```

Now you finishing writing the program after the modules given below.

```
100 rem   prob 6-3 solution
110 rem   create or edit worklist file
120 :
130 rem   variables used
140 rem     f$ = file name
150 rem     r$, c$ = response string
160 rem     m$ = task description
170 :
180 rem   files used
190 rem     seq.: worklist (user entered)
200 rem           tempfile
210 rem     dataset: m$ (one string)
220 :
230 rem   initialize
240 open 15, 8, 15 : rem disk channel
250 print "[clr]" : rem clear screen
255 print : print : print tab(7) "Create or edi
    t 'WORKLIST'" : print
260 print : input "File name "; f$
270 :
280 print "'N'ew or 'O'ld file (Press 'N' or 'O
    ')"
290 get  r$ : if r$ <> "n" and r$ <> "o" then 2
    90
300 if r$ = "o" then 350
310 :
320 rem   create f$ by open and close
330 open 1, 8, 8, f$ + ",s,w" : close 1 : gosub
     800
340 :
350 open 1, 8, 8, f$ + ",s,r" : gosub 800
360 open 2, 8, 2, "tempfile,s,w" : gosub 800
370 :
380 print : print "'A'dd or 'D'elete data (Pres
    s 'A' or 'D')"
390 get r$ : if r$ <> "a" and r$ <> "d" then 39
    0
400 if r$ = "a" then 590
410 :
420 rem   delete from file routine
```

Answer Key

Remember to try to debug your own programs before looking at our suggestions for solutions. Check back through the preceding chapters for clues and ideas.

1.
```
100 rem   prob 6-1 solution-copy address file
110 :
120 rem   variables used
130 rem      n$ = name (20)
140 rem      a$ = addrss (20)
150 rem      c$ = city (10)
160 rem      s$ = state (2)
170 rem      z$ = zip (5)
180 rem      d$ = entire dataset (57)
190 rem      f$ = copy file name
200 :
210 rem   seq. files: address, addresscopy (use
     r entered)
220 rem   dataset format: d$ (concatenated data
     set)
230 :
240 rem   initialize
250 print : input "Name for copy file "; f$
260 open 1, 8, 8, "address,s,r"
270 open 2, 8, 2, f$ + ",s,w"
280 :
290 rem   read/copy file
300 input#1, d$
310 let sv = st
320 print#2, d$
330 if sv = 0 then 300
340 :
350 rem   close file
360 close 1 : close 2
370 print : print "Address file copy completed
     with name " f$ "."
380 :
```

2. (a)
```
100 rem   prob 6-2a solution--create maglist# f
     iles
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     m$ = data (magazine)
150 rem     r$ = user response
160 :
170 rem   files used
180 rem     seq: maglist1, maglist2
190 rem     dataset: m$ (mag. titles)
```

```
200 :
210 rem   initialization
220 print : input "File name "; f$
230 open 1, 8, 8, f$ + ",s,w"
240 :
250 rem   data entry
260 print : print "If done, type 'STOP'."
270 print : input "Magazine name "; m$
280 if m$ = "stop" or m$ = "STOP" then 360
290 if m$ = "" then print "Please enter a name.
    " : goto 270
300 rem   other data entry tests here
310 :
320 print#1, m$
330 print "[clr]" : rem clear
340 goto 260
350 :
360 rem    close file
370 close 1
380 print "File '" f$ "' closed."
```

2. (b)

```
100 rem   prob 6-2b solution--read & display ma
    glist# files
110 :
120 rem   variables used
130 rem     f$ = file name
140 rem     m$ = data (magazine)
150 rem     r$ = user response
160 :
170 rem   files used
180 rem     seq: maglist1, maglist2
190 rem     dataset: m$ (mag. titles)
200 :
210 rem   initialization
220 print : input "File name "; f$
230 open 1, 8, 8, f$ + ",s,r"
240 :
250 rem   read and display data
260 input#1, m$
270 print m$
280 if st = 0 then 260
290 :
300 rem    close file
310 close 1
320 print "File '" f$ "' read & closed."
```

2. (c)
```
100 rem   prob 6-2c solution--merge maglist# fi
      les
110 :
120 rem   variables used
130 rem     f1$, f2$, f3$ = file names
140 rem     d1$, d2$ = data from files
150 rem     r$ = user response
160 :
170 rem   files used
180 rem     seq: maglist1, maglist2, magmerge
190 rem     dataset: d$ (mag. titles)
200 :
210 rem   initialization
220 :
230 print : input "Input file #1 "; f1$
240 print : input "Input file #2 "; f2$
250 print : input "Write file name "; f3$
260 :
270 open 1, 8, 8, f1$ + ",s,r"
280 open 2, 8, 2, f2$ + ",s,r"
290 open 3, 8, 3, "@0:" + f3$ + ",s,w"
300 print : print "Working...merge in progress"

310 :
320 rem   read file 1
330 input#1, m$ : let s1 = st
340 if s1 <> 0 then 570
350 :
360 rem   read file 2
370 input#2, d2$ : let s2 = st
380 if s2 <> 0 then 630
390 :
400 rem   merge testing
410 if d1$ < d2$ then 450
420 goto 510
430 :
440 rem   print file 1 to 3
450 print#3, d1$
460 if s1 <> 0 then 570
470 input#1, d1$ : let s1 = st
480 goto 410
490 :
500 rem   print file 2 to 3
510 print#3, d2$
520 if s2 <> 0 then 630
530 input#2, d2$ : let s2 = st
540 goto 410
550 :
560 rem   dump file 2 to 3
570 print#3, d2$
580 if s2 <> 0 then 690
590 input#2, d2$ : let s2 = st
```

```
600 goto 570
610 :
620 rem   dump file 1 to 3
630 print#3, d1$
640 if s1 <> 0 then 690
650 input#1, d1$ : let s1 = st
660 goto 630
670 :
680 rem   close files
690 close 1 : close 2 : close 3
700 print "[clr][down][down][down]Merge complet
     ed."
710 print : print "Would you like to see the me
     rged file ?"
720 print "(Press 'Y' or 'N')"
730 get r$ : if r$ <> "y" and r$ <> "n" then 73
     0
740 if r$ = "n" then end
750 :
760 rem   display merge file contents
770 open 3, 8, 3, f3$ + ",s,r"
780 input#3, d$
790 print d$
800 if st = 0 then 780
810 :
820 rem    close file
830 close 3
840 print : print "File displayed and closed."
```

3.

```
100 rem   prob 6-3 solution
110 rem   create or edit worklist file
120 :
130 rem   variables used
140 rem     f$ = file name
150 rem     r$, c$ = response string
160 rem     m$ = task description
170 :
180 rem   files used
190 rem     seq.: worklist (user entered)
200 rem           tempfile
210 rem     dataset: m$ (one string)
220 :
230 rem   initialize
240 open 15, 8, 15 : rem disk channel
250 print "[clr]" : rem clear screen
255 print tab(7) "[down][down]Create or edit [r
    vs]WORKLIST[off]" : print
260 print : input "File name "; f$
270 :
280 print "[rvs]N[off]ew or [rvs]O[off]ld file
    (Press [rvs]N[off] or [rvs]O[off])"
290 get  r$ : if r$ <> "n" and r$ <> "o" then 2
    90
300 if r$ = "o" then 350
310 :
320 rem   create f$ by open and close
330 open 1, 8, 8, f$ + ",s,w" : close 1 : gosub
     800
340 :
350 open 1, 8, 8, f$ + ",s,r" : gosub 800
360 open 2, 8, 2, "tempfile,s,w" : gosub 800
370 :
380 print : print "[rvs]A[off]dd or [rvs]D[off]
    elete data (Press [rvs]A[off] or [rvs]D[off]
    )"
390 get r$ : if r$ <> "a" and r$ <> "d" then 39
    0
400 if r$ = "a" then 590
410 :
420 rem   delete from file routine
430 print : print "Press [rvs]RETURN[off] to ac
    cept data displayed."
440 print "Press [rvs]D[off] to delete." : prin
    t
450 :
460 input#1, m$ : let s1 = st
470 print : print  m$ :  print
480 print "[rvs]RETURN[off] or [rvs]D[off]elete
    "
490 get r$ : if r$ <> "d" and r$ <> chr$(13) th
    en 490
```

```
500 if r$ = chr$(13) then 540
510 if s1 = 0 then 460
520 goto 700: rem  quit if e-o-f
530 :
540 print#2, m$
550 if s1 = 0 then 460
560 goto 630
570 :
580 rem   add to file routine
590 input#1, m$ : let s1 = st
600 print#2, m$
610 if s1 = 0 then 590
620 :
630 print : input "New task or [rvs]STOP[off] "
     ; m$
640 if m$ = "" then print "[up][up][up]" : goto
      630
650 if m$ = "stop" or m$ =  "STOP" then 700
660 print#2, m$
670 goto 630
680 :
690 rem   close files and rename tempfile
700 close 1 : close 2 : gosub 800
710 print#15, "s0:" + f$ : gosub 800
720 print#15, "r0:" + f$ "=tempfile" : gosub 80
     0
730 print : print "File updated and closed."
740 end
750 :
800 rem disk error channel
810 input#15, x, x$, y, z
820 if x < 20 then return
830 print : print "[down][rvs]Disk error: " x$
840 close 15 : end
```

CHAPTER SEVEN

Relative Data Files

Objectives: When you complete this chapter, you will be able to create, verify, copy, and change relative disk data files. You will be able to properly use the POINTER ("P") command, in addition to those familiar file commands (OPEN, CLOSE, PRINT#, INPUT#) that you used with sequential files.

WHAT IS A RELATIVE FILE?

A random access, or relative, data file is a disk file divided into sections called records. Each record can contain one complete dataset. The typical random access data file format of placing only one entire dataset into each record makes finding and changing data easy. The structure also allows for fast access of data, whether located in the first or last record in the file. These two strengths of random access files are the greatest weakness of sequential data files.

Commodore uses the name *relative* for a file structure where each record can be individually accessed. On most other computers, these files are usually referred to as "random access" files. However, be aware that on Commodore equipment, there is a direct access file capability, using direct commands to the disk controller. Commodore sometimes refers to direct access files as "random access" files, which is why the file type discussed in this chapter has been given the name *relative*. These direct access files require extremely careful and complex programming, since the programmer must take care of all details, including where to write data on the disk.

Relative files use the same BASIC file manipulation statements as sequential files. The only difference in statement formats is the provision for the record number and the length of the record. Relative files on your Commodore computer use what is called a variable length record. This means that the programmer determines how long, in bytes, the records for the file will be. Once established, each record in the file has the same length.

The length of the record is dependent upon the amount of data in the dataset being written to the file. In previous chapters, we discussed the storage requirements of data that are placed in a file. With relative files, it is imperative that you plan your file structure based on storage requirements or you will experience file errors and have data disappear. To review, the storage requirement for string information is one byte per character in the string, plus a carriage return (or other

character used for separation). If you include a twenty-character name in each dataset, then each name will occupy twenty-one bytes of storage. Numeric information is stored as if a STR$ operation had been performed; it takes one byte per digit, one leading byte for the minus sign or space (if the number is positive), one byte for a trailing space, and one byte for the carriage return. The integer '–99' takes 5 bytes of storage, 542.42 requires 8—six for the digits (including the decimal point), two for leading and following spaces, and one for the carriage return.

(a) In relative file application that uses a twenty-character name, a twenty-character address, and a twelve-character phone number string, how large will the record need to be in bytes? _____

(a) Fifty-five bytes.

For each relative file, you will need to compute the record size based on the dataset that is used for that file. It is important that you indicate the record size in the introductory module of your program so that the record size is permanently recorded somewhere. Once a file program is written, there is no instruction that will help you find the record size. You should include the record size in the introductory module of the program, and in any other documentation you prepare. This is as important as documenting the dataset formats; it should not be taken lightly.

The variable size record available on Commodore disk drives means that the use of disk space is very efficient. Some computers use fixed record lengths; if your dataset is short, the extra space is wasted. You will be able to tailor the record size of your relative files to the length of the dataset actually used. If your dataset is thirty-seven characters long, you will be able to set a record length of exactly thirty-seven characters. Commodore's disk operating system is able to write records overlapping disk sector boundaries, so the space taken on the disk will be thirty-seven characters per record.

Relative files require more planning and more carefully designed systems for organizing and using data. Once planned, relative files may require much less programming to accomplish the same activities as sequential files. Relative files are best used when the data in the files will change frequently. This might be the case with a customer charge account file or when you have a large database, such as a credit information file that will be accessed in no particular order (randomly). For large-scale applications, you may find yourself designing systems that use both sequential files and some relative files.

(a) What are two advantages of relative files over sequential files?

_____

_____

_____

_____

(a) Fast access to all datasets (records), regardless of position within the file, and ease of changing data within a particular dataset or record.

OPENING RELATIVE FILES

For relative files, OPEN serves the same purpose of opening the file and assigning the buffer. In addition, the OPEN statement assigns the length of the record in bytes. Here are some examples of the format of relative file OPEN statements:

```
120 OPEN 1, 8, 8, "RELFILE,L," + CHR$(50)

120 OPEN 1, 8, 8, F$ + ",L," + CHR$(50)
```

As with sequential files, the comma must not be preceded or followed by a space. Note inclusion of the commas within quotation marks in the second example. You may not get an error message if you use an incorrect format in the OPEN statement, but you may not open the file the way you intended, either.

Because the length of the record is set within a CHR$, you are limited to the allowed range for CHR$—1 to 254. (The maximum length of a relative file record is 254). If you need a longer data field, you will have to use two records.

The L *must* be included in the OPEN statement the first time a relative file is OPENed (created), or the file will default to a sequential file. The L is not essential once the file has been created, and may be omitted.

OPENing an existing file with L followed by a different record length than that used to create the file will result in an error message. If both a sequential and relative file of the same name exist on the disk, you *must* use the L.

STRUCTURE OF A RELATIVE FILE

The structure of a relative file on the disk is different from that for programs or for sequential files. When you OPEN a relative file, at least two sectors of disk space are allocated. One sector is the relative file sector. The other is a "side sector," which contains pointers to the location of the sectors of the relative file for the disk controller. Each side sector can hold pointers for up to 120 relative file sectors. A file may contain up to six side sectors, allowing a relative file to take up all the sectors on the disk ($6 \times 120 = 720$; there are 664 sectors free on a Commodore 1541, 2031, or 4040 disk).

Relative files require three buffers instead of the two used by sequential files. Because of the side sectors, disk access is very efficient. It takes a maximum of

three disk reads to locate your data. Here's an example. Let's assume you want to read record 200, the file is open, and the record length is 100. One buffer holds a side sector, the other holds a data sector (the third buffer holds data for input or output). The record starts at the 199*100th byte (19900), which is found in sector 78 (19900/254). Sector 78 can be found by looking at the pointers in side sector 0 (78/120). Since each side sector holds pointers to all other side sectors, the disk drive can directly search for and read the needed side sector. It locates the pointers to the track and sector of the data block, and reads the appropriate sector on the disk. Since some of the data may overflow into the next sector of the relative file, a third disk read may be necessary. Contrast this with a sequential file, which would have required seventy-eight reads to get to the seventy-eighth sector.

SIMPLE READ AND WRITE OPERATIONS TO RANDOM ACCESS FILES

Our first relative file application is to create an inventory of repair parts. The dataset includes a six-digit product number entered as a string, a product description of twenty characters, and a numeric quantity that will be no larger than 999, with no fractional amount.

(a) What is the record size needed for this application? _____

(b) Here is the introductory module. Complete the OPEN statement by filling in line 300.

```
100 REM    INVENTORY RELATIVE FILE
110 :
120 REM    VARIABLES USED
130 REM      N$ = PROD NUMBER (6)
140 REM      P$ = PROD DESCR (20)
150 REM      Q  = QUANTITY (<=999)
160 REM      R1 = RECORD COUNT
170 REM      H  = HI BYTE POINTER
180 REM      L  = LO BYTE POINTER
190 REM      R$ = USER RESPONSE
200 :
210 REM    FILES USED
220 REM      RELATIVE FILE: INVEN
230 REM      RECORD SIZE: 32  BYTES
240 REM      DATASET: N$, P$, Q
250 :
260 REM    INITIALIZE
270 LET R1 = 2 : REM RECORD POINTER
280 OPEN 15, 8, 15 : REM DISK CHANNEL
290 PRINT#15, "S0:INVEN" : REM DELETE IF EXISTS

300 _____ : REM OPEN REL
      FILE
310 GOSUB 800
320 :
```

(a) Thirty-two bytes: six + one for the product number, twenty + one for the description, and three + one for the quantity.
(b)
```
300 OPEN 1, 8, 8, "INVEN,L," + CHR$(32)
```

In line 270 in question (b) we initialized the variable R1 to two (2). This variable is used in this program to keep track of the number of records in the file. We have set aside record one for purposes which will be explained soon. Meanwhile, dataset one is stored as record number *two*, dataset two is stored as record *three*, and so forth. It is worth developing a standard way of dealing with this difference between the number of the dataset and its record number in the file, so you don't become confused and skip data.

Here is the data entry module for this application. We have left out the data entry tests so that the structure of the program is more clearly revealed in the program listings. By now, you know how to design good data entry error traps, and your completed programs should include them. You will see how difficult accurate data entry can be if you use the "bare bones" program listed below.

```
330 REM   DATA ENTRY MODULE
340 :
350 PRINT "[CLR][DOWN][DOWN]" : REM CLEAR & CRS
    R DOWN 2
360 PRINT : INPUT "PRODUCT NUMBER (6) "; N$
370 REM   DATA ENTRY TESTS
380 PRINT : INPUT "PROD. DESCR. (20) "; P$
390 REM   DATA ENTRY TESTS
400 PRINT : INPUT "QUANTITY (<=999) "; Q
410 REM   DATA ENTRY TESTS
420 :
```

PRINTING DATA TO A RELATIVE FILE

The file is OPEN and the data have been entered. The next operation is to print the data to the file in the second record. The PRINT# statement for relative files is similar to that for sequential files, with one important twist.

In a sequential file, each PRINT# statement prints to the next space in the file. Since there are no dividers in a sequential file, the data are simply written to the next available space, with the PRINT# statement advancing the file pointer. In a relative file, each PRINT# also advances the record pointer, so that separate PRINT# statements will print data to separate records. Here's an example:

```
250 PRINT#2, N$ : PRINT#2, Q$ : PRINT#2, G$
```

will print N$ into the first record, Q$ into the second, and G$ into the third. Therefore, the first fundamental rule of relative files is:

Print each dataset in one PRINT# statement

In other words, the line above should read

```
250 PRINT#2, N$ ;CHR$(13); Q$ ;CHR$(13); G$
```

Note that the semicolons may be left out when printing only string data, but are essential if *any* of the variables is numeric. To be safe, we recommend that you *always* include the semicolons, and use the format shown above.

LOCATING THE RECORD POINTER

Now that we can print data to the relative file, we have to tell the disk drive where in the file to print it. This requires the POINTER statement (RECORD for BASIC 4.0). The POINTER statement sets the file record pointer to the record where you want to print your data. Like many other BASIC 2.0 disk commands, it is done within the disk command channel:

```
240 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(1)
```

This statement consists of a print to the disk command channel, the "P" for POINTER, the *secondary address* of the file open for PRINTing, the low and high bytes of the record number, and the position of the byte in the record at which the PRINT# statement will start printing.

In this case, we are printing to the file with a secondary address of 2, the first record in the file (lo byte = 1, hi byte = 0), starting at the first byte of the record. Let's look at this in more detail.

This statement assumes that file 15 has been opened to the disk (OPEN 15, 8, 15). Again, using file number 15 for the disk command channel is helpful in keeping numbers straight. The P must be in quotes, and stands for POINTER. The next four numbers must be sent as CHR$; they could be concatenated with plus signs (+), or separated by semicolons (;), but neither is necessary.

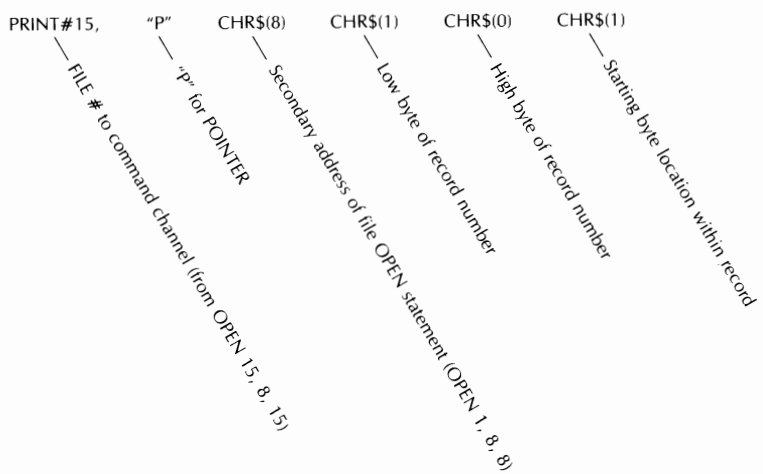


Figure 7–1 The POINTER statement

THE FIRST CHR$( ) IN THE POINTER STATEMENT: THE SECONDARY ADDRESS

The number (2) in the first CHR$( ) is the *secondary address* of the file you are printing to. If the OPEN statement is

```
OPEN 2, 8, 2, "FILE"
```

then the file and secondary address are the same, and again, your life will be simpler. Remember that secondary addresses zero (0) and one (1) are reserved; when you open file 1 you may find it convenient to use secondary address 8.

(a) Fill in the correct number in this incomplete POINTER statement to match the OPEN statement:

```
100 OPEN 1, 8, 5, "FILE,L," + CHR$(61)
 .
 .
 .
200 PRINT#15, "P" CHR$(_) ...
```

(a) `200 PRINT#15, "P" CHR$(5)`
The secondary address of the file must be used.

THE LAST CHR$( ) IN THE POINTER STATEMENT: THE RECORD BYTE POINTER

In general, the value in the last CHR$ statement in a POINTER statement will always be one, since you generally will write an entire record at a time (starting at byte 1). In the next chapter, we will give you a short example of how you would use the byte pointer to read or write starting at a location other than the beginning of a record.

THE MIDDLE CHR$( )s IN THE POINTER STATEMENT: THE RECORD POINTER

Now, how do you figure out this "low/high byte" business? We'll give you a neat formula. But first, a brief explanation. The Commodore (like most other microcomputers) stores each character in one byte, made up of eight bits (0s and 1s). Eight bits can store two to the eighth power, or 256, different values—so one byte can hold a number from 0 to 255 (that's really 256 different values). What happens if the number is greater than 256—which is a pretty short file! We use a second byte, called the high byte, because it tells us how many 256s there are in our number. Here's an abbreviated table (then on to the formula):

Number	Low byte	High byte
01	01	00
10	10	00
197	197	00
255	255	00
256	01	01
511	255	01
512	01	02

Here's the formula:

```
HI = INT(NUMBER/256)
LO = NUMBER - HI * 256
```

(a) Calculate the LO and HI values for the following numbers:

Number	Hi	Lo
2	____,	____
100	____,	____
256	____,	____
500	____,	____
750	____,	____

(a) 2: 0, 2
100: 0, 100
256: 1, 0
500: 1, 244
750: 2, 188

THE POINTER AND RELATIVE FILE PROBLEMS

There are reports of occasional problems with relative files that seem to be related to the POINTER statement. The Commodore 4040 disk drive manual recommends that you repeat the POINTER statement following the PRINT# statement to avoid destroying record data in a specific situation when reading a relative file sequentially:

```
300 PRINT#15, "P" CHR$(8)CHR$(L)CHR$(H)CHR$(1)
310 PRINT#1, A$
320 PRINT#15, "P" CHR$(8)CHR$(L)CHR$(H)CHR$(1)
```

With the 1541 drive, occasional problems with relative files seem to be improved by repeating the POINTER statement twice before PRINTing to the file, then waiting a second before writing the next datum. If you have problems with data in your relative file, you might want to try one or both of these techniques.

(a) Here is the next part of our inventory program. Fill in the blanks at lines 440, 450, and 460.

```
430 REM   PRINT TO FILE
440 _____________________________________ : REM
    SET HI/LO
450 _____________________________________ : REM
    SET POINTER
460 _____________________________________ : REM
    PRINT TO FILE
470 GOSUB 800
480 :
490 PRINT :  PRINT "MORE ENTRIES ('Y' OR 'N') ?
    "
500 GET R$ : IF R$ <> "Y" AND  R$ <> "N" THEN 5
    00
510 IF R$ = "N" THEN 590
520 :
530 REM   INCREASE RECORD COUNT
540 LET R1 = R1 + 1
550 GOTO 350
560 :
```

(b) What is the purpose of line 540?

_____

_____

(a)
```
440 LET H = INT(R1/256) : LET L = R1 - H*256 :
    REM SET HI/LO BYTES
450 PRINT#15,"P" CHR$(8) CHR$(L) CHR$(H) CHR$(1
    )
460 PRINT#1, N$ ;CHR$(13); P$ ;CHR$(13); Q
```

(b) Increments the record number by one so that if another dataset is entered, it will be recorded in the next relative record.

The final program module is the file close routine. The format of the relative CLOSE statement is the same as that used with sequential files.

(a) Complete line 590 to close the file.

```
570 REM   CLOSE FILE
580 :
590 _____________________________________ : REM
    CLOSE
600 PRINT : PRINT "FILE CLOSED."
610 END
620 :
790 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, X, X$, Y, Z : REM READ
820 IF X < 20 OR X = 50 THEN RETURN
830 PRINT : PRINT "DISK ERROR: "X$
840 PRINT : PRINT "'C'ONTINUE OR 'S'TOP ?"
850 GET R$ : IF R$ <> "S" AND R$ <> "C" THEN 85
    0
860 IF R$ = "C" THEN RETURN
870 CLOSE 15 : END
```

(a) `590 CLOSE 1 : GOSUB 800 : CLOSE 15`

After closing the file, it is good practice to check the disk error channel to be sure the file was properly closed; then the disk channel can be closed.

Here is the complete listing of our relative file printing inventory application.

```
100 REM   INVENTORY RANDOM FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (6)
140 REM     P$ = PROD DESCR (20)
150 REM     Q  = QUANTITY (<=999)
160 REM     R1 = RECORD COUNT
170 REM     H  = HI BYTE POINTER
180 REM     L  = LO BYTE POINTER
190 REM     R$ = USER RESPONSE
200 :
210 REM   FILES USED
220 REM     RELATIVE FILE: INVEN
230 REM     RECORD SIZE: 32  BYTES
240 REM     DATASET: N$, P$, Q
250 :
260 REM   INITIALIZE
270 LET R1 = 2 : REM RECORD POINTER
280 OPEN 15, 8, 15 : REM DISK CHANNEL
290 PRINT#15, "S0:INVEN" : REM DELETE IF EXISTS

300 OPEN 1, 8, 8, "INVEN,L,"+CHR$(32)
310 GOSUB 800
320 :
```

```
330 REM   DATA ENTRY MODULE
340 :
350 PRINT "[CLR][DOWN][DOWN]" : REM CLEAR & CRS
    R DOWN 2
360 PRINT : INPUT "PRODUCT NUMBER (6) "; N$
370 REM   DATA ENTRY TESTS
380 PRINT : INPUT "PROD. DESCR. (20) "; P$
390 REM   DATA ENTRY TESTS
400 PRINT : INPUT "QUANTITY (<=999) "; Q
410 REM   DATA ENTRY TESTS
420 :
430 REM   PRINT TO FILE
440 LET H = INT(R1/256) : LET L = R1 - INT(H*25
    6) : REM SET HI/LO BYTES
450 PRINT#15,"P" CHR$(8) CHR$(L) CHR$(H) CHR$(1
    )
460 PRINT#1, N$ ;CHR$(13); P$ ;CHR$(13); Q
470 GOSUB 800
480 :
490 PRINT :  PRINT "MORE ENTRIES ('Y' OR 'N') ?
    "
500 GET R$ : IF R$ <> "Y" AND  R$ <> "N" THEN 5
    00
510 IF R$ = "N" THEN 590
520 :
530 REM   INCREASE RECORD COUNT
540 LET R1 = R1 + 1
550 GOTO 350
560 :
570 REM   CLOSE FILE
580 :
590 CLOSE 1 : GOSUB 800 : CLOSE 15
600 PRINT : PRINT "FILE CLOSED."
610 END
790 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, X, X$, Y, Z : REM READ
820 IF X < 20 OR X = 50 THEN RETURN
830 PRINT : PRINT "[DOWN][RVS]DISK ERROR: "X$
840 CLOSE 15 : END
```

Many uses of relative files require that the BASIC program accessing the file know how many datasets (records) exist in the file. As no system command is available to count or display the number of records in a file, your programs to create and use relative files should provide a counting variable to keep track of the total number of records that are used in the file. This process is often used in programming applications.

For many reasons, it is convenient to use the first record in a file for such "housekeeping" information as the number of records. This means that all your datasets will be offset—stored in a record with a number one greater. There are several ways to deal with this. We find it easiest to start our record count at two, and use the counter to store the number of the record for the last dataset. A file

with the last dataset stored in record number 200 will actually have 199 records, because the first record is used to store this number, not a dataset.

When you want to add data to a relative file, you will follow these steps:

1. OPEN the file.
2. Set the pointer to record number one.
3. Read the record number of the last filled record.
4. Enter the data.
5. Increment the counter.
6. PRINT# data to the file.
7. Ask for more entries:
   a. if yes, go to 4;
   b. if no, continue.
8. Store the current record counter in the first record.
9. Close the file.

When creating a relative file, a counting statement such as LET R1 = R1 + 1 can be used. The placement of the counting statement within a program is crucial for counting accuracy. Only datasets actually entered must be counted, so the counting statement is usually placed after the dataset PRINT statement. In this way, if no more data are forthcoming, the record number will not have already been increased.

Notice where the record counting statement is placed in the previous program. The logic in this case is to increase the record counting variable by one after the user responds "yes" to the question, MORE ENTRIES?

In the example program to create the INVEN file, no provision is made to store the record count for future reference, or for use by BASIC programs that access the file. Our strategy is to store the record count in record one (the first record in the file).

(a) Help us write the POINTER statement to access the first record in a file opened with a secondary address of 2:

```
240 PRINT#15, _____
```

(a)
```
240 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(1)
```

Caution: Don't accidentally type the letter O (oh) for the number zero.

(a) Modify the program that creates the INVEN file so that the total number of records containing data (record count) is placed in R1. This routine should be included in the Close File module.

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

(a)
```
560 :
570 REM   CLOSE FILE
580 PRINT#15,"P" CHR$(8) CHR$(1) CHR$(0) CHR$(1
    )
585 PRINT#1, R1
590 CLOSE 1 : GOSUB 800 : CLOSE 15
600 PRINT : PRINT "FILE CLOSED."
610 END
790 :
```

Enter and RUN the modified program. Create the file INVEN for use in this section, as well as later programs.

READING FROM A RELATIVE FILE

Now let's write a separate program to display the contents of this relative file. Here is the introductory module and initialization module.

```
100 REM   INVEN READ/PRINT (REL)
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (6)
140 REM     P$ = PROD DESCR (20)
150 REM     Q  = QUANTITY (<=999)
160 REM     R1 = RECORD COUNT
170 REM     H  = HI BYTE POINTER
180 REM     L  = LO BYTE POINTER
190 REM     R$ = USER RESPONSE
200 :
210 REM   FILES USED
220 REM     RELATIVE FILE: INVEN
230 REM     RECORD SIZE: 32  BYTES
240 REM     DATASET: N$, P$, Q
250 :
260 REM   INITIALIZE
270 LET R1 = 2
280 OPEN 15, 8, 15 : REM DISK CHANNEL
290 OPEN 2, 8, 2, "INVEN"
300 GOSUB 800
310 :
```

(a) What is the purpose of line 270 above?

_____

_____

(b) What kind of file is opened in line 290?

_____

(a) Assigns the number two (2) to R1 to initialize the record counting variable. R1 is set to 2 because record 1 is used to store R1.

(b) A relative file. If the file does not yet exist, you'll get a "?FILE NOT FOUND" error. A previous program *must* have created the relative file with an OPEN . . ."INVEN,L,"+CHR$(32). Otherwise, a file with no indicators defaults to sequential read.

The INPUT# statement works in exactly the same way as it does for a sequential file; its variable (or variables) read successive data. Like PRINT#, INPUT# must be preceded by a POINTER statement to ensure correct access to a relative file. It is possible to read a relative file as if it were a sequential file by a series of INPUT#s, each reading the next record, since INPUT#, like PRINT#, increments the disk controller's file pointer.

(a) Assuming H is set to the high byte and L to the low byte of a record to be read, and that file number and secondary address of the file are 2, write the appropriate POINTER statement, followed by an INPUT statement to read N$, P$, and Q:

```
380 PRINT#15, _____
400 _____
```

(a)
```
380 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(1)
400 INPUT#2, N$, P$, Q
```

We still need a way to find the end of a file. We used ST for sequential files. With relative files, *there is no neat way to detect the end of the file.* ST is set to 64 at the end of each RECORD. We can use a disk error to signal the end of the file, but only approximately. When the disk drive tries to read past the end of the file, error message 50, "RECORD NOT PRESENT" is generated. We'll check for that to recognize the end of the file and stop our program. However, the end of the file on the disk is always set by entire sectors—256 bytes of storage. If you open a relative file with records of 32 bytes, eight records will be created by the disk drive in allocating the first sector, even if you only write one dataset to the disk. A character 255 (pi) is stored as the first character in each of the unused records. You are likely to see a few when you run the program completed below. We can't simply check for the pi character, because unused records within the file may be followed by records containing data, which wouldn't be read if we stopped at the first pi.

The INVEN READ/PRINT program which we are developing continues with the following routines which read the file and print the data read.

```
320 REM   PRINT HEADING
330 PRINT "[CLR][DOWN][DOWN][DOWN]PROD #" TAB(1
    0) "PROD DESCR." TAB(26) "QUANTITY" : PRINT
340 :
350 REM   FILE READ/PRINT
360 :
370 LET H = INT(R1/256) : LET L = R1 - H
380 : PRINT#15,"P" CHR$(2) CHR$(L) CHR$(H) CHR$
    (1)
390 : GOSUB 800 : IF DX = 50 THEN 460
400 : INPUT#2, N$, P$, Q
410 : PRINT N$ TAB(10) P$ TAB(31) Q
420 : LET R1 = R1 + 1
430 GOTO 370
440 :
450 REM   CLOSE FILES
460 CLOSE 1 : GOSUB 800 : CLOSE 15
470 PRINT : PRINT "FILE READ."
480 END
490 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

The INPUT statement in 400 has the same format as we used with sequential files. Notice the placement of the check for error 50—right after the POINTER statement. If we're past the end of the file, we don't want to read, so the check for error 50 must branch to the close file routine.

The only real differences between this program and a sequential file reader are the use of the POINTER statement, and the different check to stop reading the file.

(a) What is the purpose of line 420 above?

- - - - - - - - - - - - - - - - - - -

(a) It increments the record number variable by one so the next record in the file will be read.

Now let's make use of the record count, instead of depending upon an error condition, to determine the end of the file. You can do this by using a FOR NEXT loop to read only the number of datasets (records) that contain information. Notice how important this makes the accuracy of the count (and the consistency of the way you count records). An extra count may lead to (pi) being read as data when the program tries to read an unwritten record. On the other hand, if the count is short, you will leave out data.

First the record count is accessed and assigned to the variable R1:

```
360  PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(1)
370  INPUT#2, R1
```

Next, the value of R1 is used to tell the FOR NEXT loop how many datasets to read, and the FOR NEXT loop control variable X is used to count off the records.

```
390 FOR  X = 2 TO R1
400 : LET H = INT(X/256) : LET L = X - H*256
410 : PRINT#15,"P" CHR$(2) CHR$(L) CHR$(H) CHR$
      (1)
420 : INPUT#2, N$, P$, Q
430 : PRINT N$ TAB(10) P$ TAB(31) Q
440 NEXT X
```

(a) In which line is the record number to INPUT determined? _____

(b) What is the record number of the first dataset accessed? _____

(c) How many records will have been accessed when the FOR NEXT loop finishes execution? _____

(a) Line 390 (value of FOR NEXT loop control variable, X).

(b) One.

(c) Equal to value of R1–1.

Below is another version of the program. Enter the program (and the first version if you wish) and display the contents of the INVEN file on your screen.

```
100 REM   INVEN READ/PRINT (REL)
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (6)
140 REM     P$ = PROD DESCR (20)
150 REM     Q  = QUANTITY (<=999)
160 REM     R1 = RECORD COUNT
170 REM     H  = HI BYTE POINTER
180 REM     L  = LO BYTE POINTER
190 REM     R$ = USER RESPONSE
200 :
210 REM   FILES USED
220 REM     RELATIVE FILE: INVEN
230 REM     RECORD SIZE: 32  BYTES
240 REM     DATASET: N$, P$, Q
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15 : REM DISK CHANNEL
280 OPEN 2, 8, 2, "INVEN"
```

```
290 GOSUB 800
300 :
310 REM   PRINT HEADING
320 PRINT "[CLR][DOWN][DOWN][DOWN]PROD #" TAB(1
    0) "PROD DESCR." TAB(26) "QUANTITY" : PRINT
330 :
340 REM   FILE READ/PRINT
350 :
360 PRINT#15,"P" CHR$(2) CHR$(1) CHR$(0) CHR$(1
    )
370 INPUT#2, R1
380 :
390 FOR  X = 2 TO R1
400 : LET H = INT(X/256) : LET L = X - H*256
410 : PRINT#15,"P" CHR$(2) CHR$(L) CHR$(H) CHR$
    (1)
420 : INPUT#2, N$, P$, Q
430 : PRINT N$ TAB(10) P$ TAB(31) Q
440 NEXT X
450 :
460 REM   CLOSE FILES
470 CLOSE 1 : GOSUB 800 : CLOSE 15
480 PRINT : PRINT "FILE READ"
490 END
500 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

ADDING DATA TO THE END OF A RELATIVE FILE

In the next application we want a program to add new datasets to an already existing relative file. To make it easy, we will add data to the current end of an existing file, rather than insert new records into the middle of the file.

First, create the relative file to which you will later be asked to add or change data. Name the file PHONE. The program should keep track of the number of records used in the file and place this information in record R1 before closing the file. The dataset has the following items entered as strings:

customer number (five characters)
customer name (twenty-character maximum)
customer phone number (eight characters, e.g., 999–9999)

Here is the introductory module. You complete the program.

(a)

```
100 REM   CREATE 'PHONE' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUSTOMER # (5)
140 REM     C$ = CUSTOMER NAME (20)
150 REM     P$ = PHONE # (XXX-XXXX)
160 REM     R$ = USER RESPONSE
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: PHONE
230 REM     RECORD LENGTH: 36 BYTES
240 REM     DATASET: N$, C$, P$
250 :
260 REM   INITIALIZE
```

(a)

```
100 REM   CREATE 'PHONE' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUSTOMER # (5)
140 REM     C$ = CUSTOMER NAME (20)
150 REM     P$ = PHONE # (XXX-XXXX)
160 REM     R$ = USER RESPONSE
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: PHONE
230 REM     RECORD LENGTH: 36 BYTES
240 REM     DATASET: N$, C$, P$
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15 : REM DISK CHANNEL
280 OPEN 1, 8, 8, "PHONE,L," + CHR$(36) : GOSUB
      800
290 LET R1 = 1
300 :
310 REM   DATA ENTRY MODULE
320 PRINT "[CLR][DOWN][DOWN][DOWN]   PHONE FILE
      PROGRAM"
330 PRINT : PRINT "TYPE 'STOP' FOR CUST. NUMBER
      WHEN DONE."
340 PRINT : INPUT "CUSTOMER # (5 CHR.) "; N$
350 IF N$ = "STOP" THEN 530
360 LET R1 = R1 + 1
370 :
```

```
380 REM   DATA ENTRY TESTS
390 PRINT : INPUT "CUSTOMER NAME (20 CRS.) "; C
    $
400 REM  DATA ENTRY TESTS
410 PRINT : INPUT "PHONE NUMBER "; P$
420 REM  DATA ENTRY TESTS
430 :
440 REM   WRITE TO FILE
450 LET H = INT(R1/256) : LET L = R1 - H*256
460 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
    1)
470 PRINT#1, N$ ;CR$; C$ ;CR$; P$
480 GOSUB 800
490 :
500 GOTO 330
510 :
520 REM   CLOSE FILE
530 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
540 PRINT#1, R1
550 CLOSE 1 : GOSUB 800 : CLOSE 15
560 PRINT : PRINT "FILE CLOSED."
570 END
580 :
590 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "DISK ERROR: " DX$
840 CLOSE 15 : END
```

Next, write a companion program that will display the contents of PHONE, using the FOR NEXT loop technique to cycle through the records in the file.

(a)

```
100 REM   READ 'PHONE' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUSTOMER # (5)
140 REM     C$ = CUSTOMER NAME (20)
150 REM     P$ = PHONE # (XXX-XXXX)
160 REM     R$ = USER RESPONSE
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: PHONE
230 REM     RECORD LENGTH: 36 BYTES
240 REM     DATASET: N$, C$, P$
```

```
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 OPEN 1, 8, 8, "PHONE"
290 GOSUB 800
300 :
310 REM   READ # RECORDS
320 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
330 GOSUB 800 : IF DX = 50 THEN PRINT "FILE IS
    EMPTY." : GOTO 460
340 INPUT#1, R1
350 :
360 REM   READ/DISPLAY ROUTINE
370 :
380 FOR X = 2 TO R1
390 :  LET H = INT(X/256) : LET L = X - H*256
400 :  PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CH
    R$(1)
410 :  INPUT#1, N$, C$, P$
420 :  PRINT N$ TAB(7) C$ TAB(30) P$
430 NEXT X
440 :
450 REM   CLOSE FILE
460 CLOSE 1 : GOSUB 800 : CLOSE 15
470 PRINT : PRINT "FILE CLOSED."
480 END
490 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

Our relative file is a customer list entered by customer number. The dataset includes the customer number, name, and phone number. To add new datasets to the file, we must follow these steps:

1. OPEN the file.
2. Ascertain the number of records in the file containing information.
3. Enter new data.
4. Set the pointer and PRINT data to the file.
5. Increment the record count.
6. Return to step 3, if more data.
7. Set the pointer and PRINT the new record count to the first record.
8. CLOSE the file.

If you will be adding many datasets to a file, there is an advantage to creating a "dummy" dataset at the end of the file; then the entire set of records will be created at once, and the user won't have to keep waiting while the disk drive

allocates the next sector for your datasets. For example, if you planned to put 300 people into your 'PHONE' file, you might want to write dummy data to record 300 when you first create the file.

Here are the introductory module and initialization module. (Nothing really new here!)

```
100 REM   ADD TO 'PHONE' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUSTOMER # (5)
140 REM     C$ = CUSTOMER NAME (20)
150 REM     P$ = PHONE # (XXX-XXXX)
160 REM     R$ = USER RESPONSE
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: PHONE
230 REM     RECORD LENGTH: 36 BYTES
240 REM     DATASET: N$, C$, P$
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 OPEN 1, 8, 8, "PHONE"
290 GOSUB 800
300 :
```

The next program module ascertains the end of file location by reading record 1. Complete lines 320 and 330.

(a)

```
300 :
310 REM   FIND LAST FULL RECORD
320 _____________________________________ : REM S
    ET POINTER
330 _____________________________________ : REM R
    EAD RECORD COUNT
340 PRINT "RECORD COUNT = " R1 - 1 : PRINT
350 :
```

(b) Why did we print R1 – 1 in line 340?

(a)
```
300 :
310 REM   FIND LAST FULL RECORD
320 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
     1)
330 INPUT#1, R1
340 PRINT "RECORD COUNT = " R1 - 1 : PRINT
350 :
```

(b) The actual number of records with phone data is one less than the count, because one record (the first) is used to store the number of records (R1).

Next comes the data entry module and the file PRINT module. Fill in lines 470, 480, and 520 below. You may also wish to construct data entry checks now—we're leaving them up to you.

(a)
```
350 :
360 REM   DATA ENTRY MODULE
370 LET R1 = R1 + 1
380 PRINT : INPUT "CUSTOMER # "; N$
390 REM   DATA ENTRY TESTS
400 PRINT : INPUT "CUST. NAME (20 CHRS) "; C$
410 REM   DATA ENTRY TEST
420 PRINT : INPUT "PHONE # "; P$
430 REM   DATA ENTRY TESTS
440 :
450 REM   PRINT TO FILE
460 LET H = INT(R1/256) : LET L = R1 - H*256
470 ___________________________________
480 ___________________________________
490 :
500 PRINT : PRINT "MORE ENTRIES ('Y' OR 'N') ?"

510 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 51
     0
520 IF R$ = "Y" THEN _________________ : REM FIL
     L IN LINE NUMBER
530 :
```

(b)
```
440 :
450 REM   PRINT TO FILE
460 LET H = INT(R1/256) : LET L = R1 - H*256
470 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
480 PRINT#1, N$ ;CR$; C$ ;CR$; P$
490 :
520 IF R$ = "Y" THEN 370
530 :
```

The final program segment shown below closes the file and posts the record count to record one.

```
530 :
540 REM   CLOSE FILES
550 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
560 PRINT#1, R1
570 CLOSE 1 : GOSUB 800 : CLOSE 15
580 PRINT : PRINT "NEW RECORD COUNT =" R1
590 PRINT : PRINT "FILE CLOSED."
600 END
610 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

Here is the complete listing of the program to add data to an existing relative file.

```
100 REM   ADD TO 'PHONE' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUSTOMER # (5)
140 REM     C$ = CUSTOMER NAME (20)
150 REM     P$ = PHONE # (XXX-XXXX)
160 REM     R$ = USER RESPONSE
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: PHONE
230 REM     RECORD LENGTH: 36 BYTES
240 REM     DATASET: N$, C$, P$
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 OPEN 1, 8, 8, "PHONE"
290 GOSUB 800
300 :
310 REM   FIND LAST FULL RECORD
320 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
330 INPUT#1, R1
340 PRINT "RECORD COUNT = " R1 - 1 : PRINT
350 :
360 REM   DATA ENTRY MODULE
```

```
370 LET R1 = R1 + 1
380 PRINT : INPUT "CUSTOMER # "; N$
390 REM   DATA ENTRY TESTS
400 PRINT : INPUT "CUST. NAME (20 CHRS) "; C$
410 REM   DATA ENTRY TEST
420 PRINT : INPUT "PHONE # "; P$
430 REM   DATA ENTRY TESTS
440 :
450 REM   PRINT TO FILE
460 LET H = INT(R1/256) : LET L = R1 - H*256
470 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
    1)
480 PRINT#1, N$ ;CR$; C$ ;CR$; P$
490 :
500 PRINT : PRINT "MORE ENTRIES ('Y' OR 'N') ?"

510 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 51
    0
520 IF R$ = "Y" THEN 370
530 :
540 REM   CLOSE FILES
550 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
560 PRINT#1, R1
570 CLOSE 1 : GOSUB 800 : CLOSE 15
580 PRINT : PRINT "NEW RECORD COUNT =" R1 - 1
590 PRINT : PRINT "FILE CLOSED."
600 END
610 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$

840 CLOSE 15 : END
```

Enter the program and add data to PHONE. Then use the previously written program that reads and displays PHONE to verify that the additions are now in the file.

RELATIVE FILE UTILITY PROGRAMS

A UNIVERSAL RELATIVE FILE READER

Sooner or later one of your relative file reader programs will quit with the message "FILE DATA ERROR." After typing CLOSE 15 to close all open files, you may wish to look at the data actually stored on the disk. We'll develop a relative file reader using GET that can read your data, no matter how they are stored.

(a) Complete the introductory module by writing lines 120 and 130:

```
100 PRINT "[CLR][DOWN][DOWN][DOWN]RELATIVE FILE
    READER WITH GET"
110 INPUT "[DOWN]FILE NAME "; F$
120 ____________________________________
130 ____________________________________
140 INPUT "[DOWN]RECORD SIZE "; FS
150 INPUT "[DOWN]RECORD NUMBER (0 TO QUIT) "; R
    1
160 IF R1 = 0 THEN 300
170 LET H = INT(R1/256) : LET L = R1 - (H*256)
180 :
```

(b) Now write the CLOSE statement:

```
280 :
290 REM   CLOSE
300 ____________________________________
310 END
320 :
```

(a) and (b)

```
100 PRINT "[CLR][DOWN][DOWN][DOWN]RELATIVE FILE
    READER WITH GET"
110 INPUT "[DOWN]FILE NAME "; F$
120 OPEN 15, 8, 15
130 OPEN 2, 8, 2, F$ :GOSUB800
140 INPUT "[DOWN]RECORD SIZE "; FS
150 INPUT "[DOWN]RECORD NUMBER (0 TO QUIT) "; R
    1
160 IF R1 = 0 THEN 300
170 LET H = INT(R1/256) : LET L = R1 - (H*256)
180 :
190 REM   READ FROM POS 1 TO FS
200 FOR P = 1 TO FS
210 :  PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CH
    R$(P) : GOSUB 800
220 :   IF DX = 50 THEN PRINT "[DOWN]END OF FIL
    E" : GOTO 150
230 :  GET#2, A$ : GOSUB 800
240 :  PRINT A$;
250 :   IF A$ = "" OR A$ = "[PI]" THEN 150
260 NEXT P
270 GOTO 150
280 :
290 REM   CLOSE
300 CLOSE  2 : GOSUB800 : CLOSE 15
310 END
```

```
320 :
790 :
800 REM DISK ERROR
810 INPUT#15, DX, DX$, DY, DZ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR #"DX; DX$
840 CLOSE 15 : END
```

The FOR NEXT loop (lines 200–260) is a bit peculiar. It takes advantage of the last character in the POINTER statement, which lets you specify the byte in the record. The POINTER statement in line 210 thus points in turn to each byte (character) of the record, so the GET statement in line 230 takes each character from the record.

Lines 220 and 250 stop reading at the end of the file. If a pi [CHR$(255)] is encountered, all following characters will be blank—there's no point in waiting to GET dozens of null characters. Unused records have CHR$(255) (pi) as the first character. Line 250 also checks for null characters, since unused bytes of a record are filled by binary 00, which are read as null characters. Of course, if you have gaps within your record, this program will stop reading the record at the gap. To display the entire contents of a record, change line 250 to read

```
250 IF A$ = "[PI]" THEN 150
```

Line 220 checks to see if we are past the end of the file, in which case there is no point in reading further.

A RELATIVE FILE COPY PROGRAM

Having covered the essentials of using relative files, let's write two file utility programs to further your understanding and provide models for similar programs you can write. The first program simply copies the data from one relative file into another relative file, record for record. The data are both alphabetic and numeric. Copy programs are not essential, because the disk COPY command will copy relative files. However, a copy program is easily revised to allow you to edit data, so we will construct a copy program. The advantage of using the disk COPY command is that you don't have to worry about record length or data format—the disk drive takes care of everything for you.

Write a program to create a relative file named MASTER. This file will be used later in this section by a file utility program that makes a copy of a relative file. You can decide what information corresponds to the variables listed in the introductory module given below. Use your imagination!

(a)

```
100 REM   CREATE 'MASTER' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     G$ = (20 CHRS.)
140 REM     S  = (8 DIGITS )
150 REM     Q  = (4 DIGITS )
160 REM     M$ = (30 CHRS.)
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: MASTER
230 REM     RECORD LENGTH: 66 BYTES
240 REM     DATASET: G$, S, Q, M$
250 :
260 REM   INITIALIZE
```

(a)

```
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 OPEN 1, 8, 8, "MASTER,L," + CHR$(66)
290 GOSUB 800
300 LET R1 = 2
310 :
320 REM   DATA ENTRY MODULE
330 PRINT "[CLR][DOWN][DOWN][DOWN]" : REM CLEAR
      & CRSR DOWN 2
340 PRINT : INPUT "STRING #1 (20 CHRS) "; G$
350 REM   DATA ENTRY TESTS
360 PRINT : INPUT "NUMERIC (8 DIGITS) "; S
370 REM   DATA ENTRY TEST
380 PRINT : INPUT "NUMERIC (4 DIGITS) "; Q
390 REM   DATA ENTRY TEST
400 PRINT : INPUT "STRING #2 (30 CHRS) "; M$
410 REM   DATA ENTRY TESTS
420 :
```

```
430 REM   PRINT TO FILE
440 LET H = INT(R1/256) : LET L = R1 - H*256
450 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
    1)
460 PRINT#1, G$ ;CR$; S ;CR$; Q ;CR$; M$
470 :
480 PRINT : PRINT "MORE ENTRIES ('Y' OR 'N') ?"

490 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 49
    0
500 IF R$ = "Y" THEN LET R1 = R1 + 1 : GOTO 330

510 :
520 REM   CLOSE FILES
530 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
540 PRINT#1, R1
550 CLOSE 1 : GOSUB 800 : CLOSE 15
560 PRINT : PRINT "FILE CLOSED."
570 END
580 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

Now write a companion program to read and display the contents of MASTER. Allow the user to enter the file name. Include a "PRESS RETURN TO DISPLAY NEXT DATASET" routine inside the READ/DISPLAY loop.

(a)

```
100 REM   READ  & DISPLAY 'MASTER' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     G$ = (20 CHRS.)
140 REM     S  = (8 DIGITS )
150 REM     Q  = (4 DIGITS )
160 REM     M$ = (30 CHRS.)
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: MASTER
230 REM     RECORD LENGTH: 66 BYTES
240 REM     DATASET: G$, S, Q, M$
250 :
```

(a)

```
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
290 PRINT : INPUT "NAME OF FILE "; F$
300 REM   DATA ENTRY TESTS HERE
310 OPEN 1, 8, 8, F$
320 :
330 REM   DATA ENTRY MODULE
340 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
     1)
350 INPUT#1, R1
360 :
370 FOR X = 2 TO R1
380 :  LET H = INT(X/256) : LET L = X - H*256
390 :  PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CH
    R$(1)
400 :  INPUT#1, G$, S, Q, M$
410 :  GOSUB 800
420 :
430 :  PRINT "[CLR][DOWN][DOWN][DOWN]RECORD" X
    ":"
440 :  PRINT G$ : PRINT S : PRINT Q : PRINT M$
450 :
460 PRINT : PRINT "PRESS [RVS]RETURN[OFF] FOR N
     EXT DATASET."
470 :  GET R$ : IF R$ <> CHR$(13) THEN 470
480 NEXT X
490 :
500 REM   CLOSE FILE
510 CLOSE  1 : GOSUB 800 : CLOSE 15
520 PRINT : PRINT  "FILE DISPLAYED AND CLOSED."

530 END
540 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

Follow these steps to create a relative file-copying program:

1. OPEN the source file.
2. OPEN and clear the copy file.
3. Determine the record count.
4. Position the pointer and read the source file record.

5. Position the pointer and print the copy file.
6. Return to step 4 until end of data.
7. CLOSE files after posting the record count in the copy file.

We will now help you write a program that will make a copy of MASTER. The copy file is named STORE1. Here's the introductory module:

```
100 REM    COPY 'MASTER' REL FILE
110 :
120 REM    VARIABLES USED
130 REM      G$ = (20 CHRS.)
140 REM      S  = (8 DIGITS )
150 REM      Q  = (4 DIGITS )
160 REM      M$ = (30 CHRS.)
170 :        CR$ = CHR$(13)
180 REM      L  = LO BYTE
190 REM      H  = HI BYTE
195 REM      R1 = RECORD COUNT
200 :
210 REM    FILES USED
220 REM      REL FILE: MASTER
230 REM      COPY FILE: STORE1
240 REM      RECORD LENGTH: 66 BYTES
250 REM      DATASET: G$, S, Q, M$
260 :
```

Notice that we have only indicated the type and the length of the variables; what data they represent is not important (and has been left to your imagination).

Complete the following segment to initialize two files by filling in lines 280, 290, 310, and 320.

(a)
```
260 :
270 REM    INITIALIZE
280 ----------------------------------
290 ----------------------------------
300 GOSUB 820
310 ----------------------------------
320 ----------------------------------
330 GOSUB 820
340 :
```

(a)
```
260 :
270 REM    INITIALIZE
280 OPEN 15, 8, 15
290 OPEN 1, 8, 8, "MASTER"
300 GOSUB 820
310 PRINT#15,"S0:STORE1" : GOSUB 820
320 OPEN 2, 8, 2, "STORE1,L," + CHR$(66)
330 GOSUB 820
340 :
```

Line 310 SCRATCHes any existing file named "STORE1" to assure line 320 will create a new relative file.

You cannot use the replacement function with relative files (the OPEN "@0:filename"), as you could with sequential files. Instead you must first SCRATCH the file, then OPEN it with the length (",L,") indicator. Simply OPENing a relative file, with or without the length specifier, will not change any data in the file. SCRATCHing a file simply erases the directory entry, without actually erasing any data on the disk. When you OPEN ",L," a new relative file, as each sector is written to, all unused records are filled with a pi [CHR$(255)] followed by binary 00s, thereby erasing any old data in the file.

You can force OPEN to write over old data by writing a "dummy" record to a record number about as large as the file will actually hold:

```
150 OPEN 2, 8, 2, "DEMO,L" + CHR$(66)
160 PRINT#15, "P" CHR$(2) CHR$(0) CHR$(1)
    CHR$(1)
170 PRINT#2, "*"
```

This forces the computer to write empty records up to the dummy record (in this case, number 256), thereby writing over all existing records. If you use this technique, you must be careful that the file name you are creating doesn't already exist, so that you don't accidentally destroy a file!

The next section reads from the source file and prints to the copy file. Fill in the blanks (lines 360, 370, 400, 410, 420, 450, and 460). Note that you can use the same values of H and L in both pointer statements, since both files are dealing with the same record.

(a)
```
340 :
350 REM   READ SOURCE FILE
360 ____________________ ______________
370 ____________________________________
380 :
390 FOR X = 2 TO R1
400 ____________________________________ : REM S
     ET HI/LO
410 ____________________________________
415 : GOSUB 820 : IF DX = 50 THEN R1 =  X :GOTO
      590 : REM READ PAST END OF FILE
420 ____________________________________
430 :
440 REM   PRINT TO COPY FILE
450 ____________________________________
460 ____________________________________
470 NEXT X
480 :
```

(a)
```
340 :
350 REM   READ SOURCE FILE
360 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
370 INPUT#1, R1
380 :
390 FOR X = 2 TO R1
400 :  LET H = INT(X/256) : LET L = X - H*256
410 :  PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CH
    R$(1)
415 : GOSUB 820 : IF DX = 50 THEN R1 =  X :GOTO
     590 : REM READ PAST END OF FILE
420 :  INPUT#1, G$, S, Q, M$
430 :
440 REM   PRINT TO COPY FILE
450 :  PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CH
    R$(1)
460 :  PRINT#2, G$ ;CR$; S ;CR$; Q ;CR$; M$
470 NEXT X
480 :
```

You probably found completing that program easy. Relative files are easy to manipulate, once you get the hang of it. The real trick is in setting the high and low bytes, and in carefully writing the pointer statements. Correct use of the pointer is the critical element. Relative file programs that don't work often have mistakes in the pointer statement (a CHR$ left out, an accidental extra character, reversed order of the low and high bytes, etc.).

Here is the complete program.

```
100 REM   COPY 'MASTER' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     G$ = (20 CHRS.)
140 REM     S  = (8 DIGITS )
150 REM     Q  = (4 DIGITS )
160 REM     M$ = (30 CHRS.)
170 :       CR$ = CHR$(13)
180 REM     L  = LO BYTE
190 REM     H  = HI BYTE
195 REM     R1 = RECORD COUNT
200 :
210 REM   FILES USED
220 REM     REL FILE: MASTER
230 REM     COPY FILE: STORE1
240 REM     RECORD LENGTH: 66 BYTES
250 REM     DATASET: G$, S, Q, M$
260 :
270 REM   INITIALIZE
280 OPEN 15, 8, 15
```

```
290 OPEN 1, 8, 8, "MASTER"
300 GOSUB 820
310 PRINT#15,"S0:STORE1" : GOSUB 820
320 OPEN 2, 8, 2, "STORE1,L," + CHR$(66)
330 GOSUB 820
340 :
350 REM   READ SOURCE FILE
360 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
     1)
370 INPUT#1, R1
380 :
390 FOR X = 2 TO R1
400 :  LET H = INT(X/256) : LET L = X - H*256
410 :  PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CH
    R$(1)
415 : GOSUB 820 : IF DX = 50 THEN R1 =  X :GOTO
      590 : REM READ PAST END OF FILE
420 :  INPUT#1, G$, S, Q, M$
430 :
440 REM   PRINT TO COPY FILE
450 :  PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CH
    R$(1)
460 :  PRINT#2, G$ ;CR$; S ;CR$; Q ;CR$; M$
470 NEXT X
480 :
490 REM   CLOSE FILES
500 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
     1)
510 PRINT#2, R1
520 CLOSE 1 : CLOSE 2 : GOSUB 820: CLOSE 15
530 PRINT : PRINT "FILE COPY COMPLETE."
540 END
550 :
800 :
810 REM   DISK ERROR ROUTINE
820 INPUT#15, DX, DX$, DY, DZ : REM READ
830 IF DX < 20 OR DX = 50 THEN RETURN
840 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$

850 CLOSE 15 : END
```

(a) Check your understanding of the file-copying program by filling in the corresponding program line number(s) for each step in the outline.

1. OPEN the source file. _____
2. OPEN and clear the copy file. _____
3. Determine the record count. _____
4. Position the pointer and read the source file record. _____
5. Position the pointer and print the copy file. _____

6. Return to step 4 until end of data.
7. CLOSE files after posting the record count in the copy file.

(a) 1. 290
2. 310, 320
3. 360–370
4. 410, 420
5. 450–460
6. 470 (410–460)
7. 500–520

EDITING DATA IN AN EXISTING RELATIVE FILE

So far, you have learned how to add data to a relative file and how to make a copy of a relative file. Next, let's consider a versatile utility program that allows a number of options for changing the data in a relative file. We will be using the INVEN file you created earlier in this chapter. We will use the complete dataset with product code number, product description, quantity available, and record count stored in R1. You want your program to display the datasets in the file, one record at a time, and allow the user the following options:

1. Change all data items.
2. Change the code number only.
3. Change the description only.
4. Change the quantity only.
5. No change to this record.

Follow these steps:

1. OPEN the file.
2. Determine record count.
3. READ a dataset.
4. Display the dataset.
5. Display the "menu" of choices.
6. Request and test choice.
7. Branch to appropriate subroutines according to choice made.
8. Return to step 3 above.
9. CLOSE the file.

There are some cautions in writing data to an existing relative file.

1. Be sure the pointer is set to the correct record, or you will overwrite the wrong data!
2. When using 1540 or 1541 drives, or writing a program for use on any Commodore drive, repeat the POINTER statement.

3. Read and write the entire dataset. Although the byte pointer allows you to position to any character within the record, its use is tricky.
4. Check the length of your new dataset to be sure it doesn't exceed the record length. If it does, the message OVERFLOW IN RECORD will result. Only the data that will fit into the record size will actually be written to the disk; the remainder will be ignored.

Here is the complete program.

```
100 REM    INVEN EDITOR (REL)
110 :
120 REM    VARIABLES USED
130 REM      N$ = PROD NUMBER (6)
140 REM      P$ = PROD DESCR (20)
150 REM      Q  = QUANTITY (<=999)
160 REM      R1 = RECORD COUNT
170 REM      H  = HI BYTE POINTER
180 REM      L  = LO BYTE POINTER
190 REM      R$ = USER RESPONSE
195 :        CR$ = CHR$(13)
200 :
210 REM    FILES USED
220 REM      RELATIVE FILE: INVEN
230 REM      RECORD SIZE: 32 BYTES
240 REM      DATASET: N$, P$, Q
250 :
260 REM    INITIALIZE
270 OPEN 15, 8, 15 : REM DISK CHANNEL
280 OPEN 2, 8, 2, "INVEN"
290 GOSUB 1000
300 :
310 REM    READ ONE  RECORD
320 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
     1)
330 INPUT#2, R1
340 :
350 FOR X = 2 TO R1
360 :
370 LET H = INT(X/256) : LET L = X - H*256
380 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
390 INPUT#2, C$, P$, Q
400 :
410 REM    DISPLAY DATASET & OPTIONS
420 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
430 PRINT "PRODUCT #: " C$
440 PRINT "DESCRIPT.: " P$
450 PRINT "QUANTITY :" Q
460 :
470 PRINT : PRINT "PRESS KEY FOR OPTION TO CHAN
     GE: "
```

```
480 PRINT "[RVS]A[OFF]LL (A)"
490 PRINT "[RVS]N[OFF]UMBER ONLY (N)"
500 PRINT "[RVS]D[OFF]ESCRIPTION ONLY (D)"
510 PRINT "[RVS]Q[OFF]UANTITY ONLY (Q)"
520 PRINT "[RVS]X[OFF]= NO CHANGE (X)"
530 PRINT
540 GET R$ : IF R$ = "" THEN 540
550 IF R$ = "A" THEN GOSUB 680: GOSUB 720: GOSU
     B 760: GOSUB 810: GOTO 620
560 IF R$ = "N" THEN GOSUB 680: GOSUB 810: GOTO
      620
570 IF R$ = "D" THEN GOSUB 720: GOSUB 810: GOTO
      620
580 IF R$ = "Q" THEN GOSUB 760: GOSUB 810: GOTO
      620
590 IF R$ = "X" THEN 620: REM NEXT
600 GOTO 540
610 :
620 NEXT X
630 :
640 GOTO 880: REM CLOSE
650 :
660 REM   DATA ENTRY SUBROUTINES
670 :
680 PRINT : INPUT "NEW PRODUCT CODE "; C$
690 REM   DATA ENTRY TESTS
700 RETURN
710 :
720 PRINT : INPUT "NEW DESCRIPTION "; P$
730 REM   DATA ENTRY TESTS
740 RETURN
750 :
760 PRINT : INPUT  "NEW QUANTITY "; Q
770 REM   DATA ENTRY TESTS
780 RETURN
790 :
800 REM   FILE PRINT SUBROUTINE
810 LET H = INT(X/256) : LET L = X - H*256
820 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
830 PRINT#2, C$ ;CR$; P$ ;CR$; Q
840 RETURN
850 :
860 :
870 REM   CLOSE FILE
880 CLOSE 2 : GOSUB 1000: CLOSE 15
890 END
900 :
910 :
1000 REM   DISK ERROR ROUTINE
1010 INPUT#15, DX, DX$, DY, DZ : REM READ
1020 IF DX < 20 OR DX = 50 THEN RETURN
1030 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$

1040 CLOSE 15 : END
```

(a) Study the program carefully and write the corresponding line numbers for each step in the outline shown below.

1. OPEN the file. _____
2. Determine record count. _____
3. READ a dataset. _____
4. Display the dataset. _____
5. Display the "menu" of choices. _____
6. Request and test choice. _____
7. Branch to appropriate subroutines according to choice made. _____
8. Return to step 3 above. _____
9. CLOSE the file. _____

(a)
1. 280
2. 320–330
3. 380–390
4. 420–450
5. 470–520
6. 540–600
7. 550–590
8. 620
9. 880

Since a relative file allows you to place a dataset at any record you select, the editing program could have another choice which would allow you to specify to which record the dataset should be written. This would allow you to change the order of records within the file. Of course, you should add appropriate checks on the selected record to determine what data are there, and to ask the user if they wish to overwrite them.

CONVERTING SEQUENTIAL FILES TO RELATIVE FILES

Another useful file utility program is one that converts (copies) a sequential file to a relative file. The procedure involves making a copy of the sequential file and placing one dataset from the sequential file into one record in a relative file. If at some point you want to standardize your entire software collection or system into relative file format, a program modeled on the one you are about to write would do the job.

The example is a small business-type application where a sequential file contains data in this format:

customer number = five-character string
customer name = twenty-character string
credit status code = single-digit number, one to five—one-character numeric value

You may recognize this as the format of the customer credit file named CREDIT, a sequential file you created in the Chapter 5 Self-Test. It is the same file you used in Chapter 6 for file-editing application programs. The task is to copy a sequential data file into a relative file, one dataset (as described above) per record. The outline of steps is as follows:

1. OPEN the sequential file.
2. OPEN the relative file.
3. INPUT a dataset from the sequential file.
4. PRINT a dataset to the relative file.
5. Increment the record counter.
6. Check for end of the sequential file (ST).
7. If not end, return to step 3 above.
8. PRINT the record count to the file and CLOSE the files.

Here are the introductory and initializing modules. Read them over carefully.

```
100 REM   COPY SEQ (CREDIT) TO REL (R-CREDIT)
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUST. # (5 CHRS)
140 REM     C$ = CUST. NAME (20 CHRS)
150 REM     R  = CREDIT RATING (1 CHR)
160 REM     R1 = RECORD COUNT
170 :       CR$= CHR$(13)
180 :
190 REM   FILES USED
200 :
210 REM     SEQ FILE: CREDIT
220 REM     REL FILE: R-CREDIT
230 REM     RECORD LENGTH: 29 BYTES
240 REM     DATASET: N$, C$, R
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 PRINT "[CLR][DOWN][DOWN]COPY [RVS]CREDIT[RV
    S] SEQ TO [RVS]R-CREDIT[OFF] REL."
290 PRINT : PRINT "WORKING...
300 LET R1 = 1
310 OPEN 1, 8, 8, "CREDIT,S,R"
320 GOSUB 800
330 OPEN 2, 8, 2, "R-CREDIT,L," + CHR$(29)
340 GOSUB 800
350 :
```

(a) What is the length of the relative file record? _____

_____

(b) Which record should the program PRINT to? _____

_____

(a) Twenty-nine bytes [CHR$(29) in line 330].
(b) Record 2; record 1 is reserved for the record counter.

Here is the rest of the program. Complete lines 380, 390, 440, 450, 460, 520, 530, and 550. Don't forget to store the STatus after the INPUT from the sequential file!

(a)

```
350 :
360 REM   READ SEQ DATA
370 :
380 _____ : REM RE
    AD
390 _____
400 PRINT "[HOME][DOWN][DOWN][DOWN][DOWN][DOWN]
    [DOWN] READING: "N$; C$; R
410 :
420 REM   PRINT TO REL FILE
430 LET R1 = R1 + 1
440 _____ : REM HI
    /LO
450 _____ : REM PO
    INTER
460 _____ : REM WR
    ITE
470 :
480 IF SV = Ø THEN 380
490 :
500 :
510 REM   SAVE RECORD COUNTER & CLOSE FILES
520 _____ : REM PO
    INTER
530 _____ : REM WR
    ITE COUNT
540 :
550 _____ : REM CL
    OSE
560 PRINT : PRINT "FILES CLOSED, COPY COMPLETE.
    "
```

(b) Why did we increment the record counter *before* writing to the relative file?

(a)

```
350 :
360 REM   READ SEQ DATA
370 :
380 INPUT#1, N$, C$, R
390 LET SV = ST
400 PRINT "[HOME][DOWN][DOWN][DOWN][DOWN][DOWN]
    [DOWN] READING: "N$; C$; R
410 :
420 REM   PRINT TO REL FILE
430 LET R1 = R1 + 1
440 LET H = INT(R1/256) : LET L = R1 - H*256
450 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
460 PRINT#2, N$ ;CR$; C$ ;CR$; R
470 :
480 IF SV = 0 THEN 380
490 :
500 :
510 REM   CLOSE FILES
520 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
530 PRINT#2, R1
540 :
550 CLOSE 1 : GOSUB 800 : CLOSE 2 : GOSUB 800 :
     CLOSE 15
560 PRINT : PRINT "FILES CLOSED, COPY COMPLETE.
    "
570 END
```

(b) So the counter was set to the *second* record. After the first dataset, the counter could be incremented anywhere between lines 380 and 480. Again, a consistent approach is strongly recommended!

Here is the complete file conversion program. Look it over and complete the outline that follows with the corresponding line numbers.

```
100 REM   COPY SEQ (CREDIT) TO REL (R-CREDIT)
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUST. # (5 CHRS)
140 REM     C$ = CUST. NAME (20 CHRS)
150 REM     R  = CREDIT RATING (1 CHR)
160 REM     R1 = RECORD COUNT
170 :       CR$= CHR$(13)
180 :
190 REM   FILES USED
200 :
210 REM     SEQ FILE: CREDIT
220 REM     REL FILE: R-CREDIT
230 REM     RECORD LENGTH: 29 BYTES
240 REM     DATASET: N$, C$, R
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 PRINT "[CLR][DOWN][DOWN]COPY [RVS]CREDIT[RV
    S] SEQ TO [RVS]R-CREDIT[OFF] REL."
290 PRINT : PRINT "WORKING...
300 LET R1 = 1
310 OPEN 1, 8, 8, "CREDIT,S,R"
320 GOSUB 800
330 OPEN 2, 8, 2, "R-CREDIT,L," + CHR$(29)
340 GOSUB 800
350 :
360 REM   READ SEQ DATA
370 :
380 INPUT#1, N$, C$, R
390 LET SV = ST
400 PRINT "[HOME][DOWN][DOWN][DOWN][DOWN][DOWN]
    [DOWN] READING: "N$; C$; R
410 :
420 REM   PRINT TO REL FILE
430 LET R1 = R1 + 1
440 LET H = INT(R1/256) : LET L = R1 - H*256
450 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
460 PRINT#2, N$ ;CR$; C$ ;CR$; R
470 :
480 IF SV = 0 THEN 380
490 :
500 :
510 REM   CLOSE FILES
520 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
530 PRINT#2, R1
540 :
550 CLOSE 1 : GOSUB 800 : CLOSE 2 : GOSUB 800 :
     CLOSE 15
560 PRINT : PRINT "FILES CLOSED, COPY COMPLETE.
    "
```

```
570 END
580 :
590 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

(a) 1. OPEN the sequential file. _____

2. OPEN the relative file. _____

3. INPUT a dataset from the sequential file. _____

4. PRINT a dataset to the relative file. _____

5. Increment the record counter. _____

6. Check for end of the sequential file (ST). _____

7. If not end, return to step 3 above. _____

8. PRINT the record count to the file and CLOSE the files. _____

(a) 1. 310
2. 330
3. 380
4. 440–460
5. 430
6. 480 (ST saved in line 390)
7. 480
8. 520–530 and 550

Write a program to display the relative CREDIT file. Below is the introductory module.

(a)
```
100 REM   DISPLAY 'R-CREDIT' REL FILE
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUST. # (5 CHRS)
140 REM     C$ = CUST. NAME (20 CHRS)
150 REM     R  = CREDIT RATING (1 CHR)
160 REM     R1 = RECORD COUNT
170 REM     X  = FOR...NEXT VARIABLE
180 REM     R$ = USER RESPONSE
190 :       CR$= CHR$(13)
200 :
210 REM   FILES USED
220 :
230 REM     SEQ FILE: CREDIT
240 REM     REL FILE: R-CREDIT
250 REM     RECORD LENGTH: 29 BYTES
260 REM     DATASET: N$, C$, R
270 :
```

(a)

```
280 REM   INITIALIZE
290 OPEN 15, 8, 15
300 OPEN 2, 8, 2, "R-CREDIT"
310 GOSUB 800
320 :
330 REM   READ/PRINT REL FILE
340 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
     1)
350 INPUT#2, R1
360 :
370 FOR X = 2 TO R1
380 LET H = INT(X/256) : LET L = X - H*256
390 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
400 INPUT#2, N$, C$, R
410 PRINT : PRINT "RECORD #" X ":"
420 PRINT N$, C$, R
430 PRINT : PRINT "PRESS 'RETURN' TO CONTINUE."

440 GET R$ : IF R$ <> CHR$(13) THEN 440
450 NEXT X
460 :
470 :
480 REM   CLOSE FILES
490 :
500 CLOSE 2 : GOSUB 800 : CLOSE 15
510 PRINT : PRINT "FILES READ AND CLOSED."
520 END
530 :
540 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$

840 CLOSE 15 : END
```

Chapter 7 Self-Test

1. (a) Write a program to create a relative data file that contains the inventory of products carried by an imaginary business. Each relative record contains the following data for one item of inventory in the order shown below. Numbers in parentheses indicate maximum character counts. Name this file BUSINESS INVEN. Create the file with your program.

N$ = product number (4)
P$ = description of inventory item (20)
S$ = supplier (20)
L = reorder point (how low the stock of item can be before reordering) (3)
Y = reorder quantity (4)
Q = quantity available (currently in stock) (4)
C = cost (from supplier) (6)
U = unit selling price (what the item is sold for) (6)

Here is the introductory module and a sample RUN.

```
100 REM   PROB 1A BUS INVENTORY
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (4)
140 REM     P$ = PROD DESCR. (20)
150 REM     S$ = SUPPLIER (20)
160 REM     R  = REORDER PT. (3)
170 REM     Y  = REORDER QUAN (4)
180 REM     Q  = QUAN IN STOCK (4)
190 REM     C  = COST (TO RETAIL) (6)
200 REM     U  = UNIT (RETL) PRICE (6)
210 REM     R$ = USER RESPONSE
220 REM     R1 = RECORD COUNT
230 REM     H  = HI BYTE
240 REM     L  = LOW BYTE
250 :       CR$= CHR$(13)
260 :
270 REM   FILES USED
280 REM     REL FIL: BUSINESS INVEN
290 REM     RECORD LENGTH: 75 BYTES
300 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
310 :
320 REM   INITIALIZE
```

```
RUN
BUSINESS INVENTORY RELATIVE FILE
PRODUCT # (4 DIGITS) ? 1234
PROD. DESCR. (20 CHRS) ? SAMPLE DATA
SUPPLIER (20 CHRS) ? SOULE SOURCE
```

```
REORDER POINT ? 12
REORDER QUANTITY ? 24
QUANTITY IN STOCK ? 36
WHOLESALE COST ? .55
UNIT SELLING PRICE ? 1.95
MORE DATA (PRESS 'Y' OR 'N') ? N

TOTAL DATASETS = 1
FILE CLOSED.

READY.
```

1. (b) Using the program from Problem 1(a), create a relative file named BUSINESS INVEN. Make up your own data for at least five records (inventory items) and enter them into the file. This file will be used in examples and activities in Chapter 8.

   Write a program to display the contents of BUSINESS INVEN, including the record count.

1. (c) Write a program to create a sequential (not relative) file called INDEX that contains the following two items in each dataset:

   1. Account numbers from BUSINESS INVEN file (a four-character string).
   2. The record number (a numeric value) corresponding to the record location of each account number.

   The program should read the first data item from each record in BUSINESS INVEN and write the account number (four-character string) and the record count number for that record into the sequential file called INDEX.

```
100 REM   PROB7-1C  SEQ INDEX FILE FOR
110 REM   'BUSINESS INVEN' (REL) FILE
120 :
130 REM   VARIABLES USED
140 REM     N$ = PROD NUMBER (4)
150 REM     P$ = PROD DESCR. (20)
160 REM     S$ = SUPPLIER (20)
170 REM     R  = REORDER PT. (3)
180 REM     Y  = REORDER QUAN (4)
190 REM     Q  = QUAN IN STOCK (4)
200 REM     C  = COST (TO RETAIL) (6)
210 REM     U  = UNIT (RETL) PRICE (6)
220 REM     R$ = USER RESPONSE
230 REM     R1 = RECORD COUNT
240 REM     H  = HI BYTE
250 REM     L  = LOW BYTE
260 :       CR$= CHR$(13)
270 :
280 REM   FILES USED
290 REM     REL FIL: BUSINESS INVEN
300 REM     RECORD LENGTH: 75 BYTES
310 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
320 REM     SEQ FILE: INDEX
330 :
340 REM   INITIALIZE
```

1. (d) Write a program to read and display the data items in INDEX.

2. Write a program to make a copy of the relative file named R-CREDIT that you transferred from a sequential file in the last example program in Chapter 7. The copy should be another relative file named R-CREDIT COPY.

   Here is the introductory module:

```
100 REM   PROB. 7-2:COPY R-CREDIT (REL)
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUST. # (5 CHRS)
140 REM     C$ = CUST. NAME (20 CHRS)
150 REM     R  = CREDIT RATING (1 CHR)
160 REM     R1 = RECORD COUNT
170 REM     L  = LO BYTE
180 REM     H  = HI BYTE
190 REM     R$ = USER RESPONSE
200 REM     X  = FOR...NEXT VARIABLE
210 :       CR$= CHR$(13)
220 :
230 REM   FILES USED
240 REM     SOURCE REL FILE: R-CREDIT
250 REM     COPY REL FILE: R-CREDIT COPY
260 REM     RECORD LENGTH: 29 BYTES
270 REM     DATASET: N$, C$, R
280 :
290 REM   INITIALIZE
```

3. Write a program to display the contents of the original data file and the copy in the previous problem (2), for verification of the completeness and accuracy of the copy. The program should display the data in record 1 of the original file, and then the data from record 1 in the file copy, then the data from record 2 in the original file, followed by the data from record 2 in the copy, and so on to the end of the files.

```
100 REM   PROB. 3--READ  & DISPLAY 2 REL FILES
110 :
120 REM   VARIABLES USED
130 REM     N$,N1$ = CUST. # (5 CHRS)
140 REM     C$,C1$ = CUST. NAME (20 CHRS)
150 REM     C, C1  = CREDIT RATING (1 CHR)
160 REM     R, R1  = RECORD COUNT
170 REM     L  = LO BYTE
```

```
180 REM      H  = HI BYTE
190 REM      R$ = USER RESPONSE
200 REM      X  = FOR...NEXT VARIABLE
210 :
220 REM   FILES USED
230 REM      REL FILE #1: R-CREDIT
240 REM      REL FILE #2: R-CREDIT COPY
250 REM      RECORD LENGTH: 29 BYTES
260 REM      DATASET: N$, C$, C
270 :
280 REM   INITIALIZE
290 OPEN 15, 8, 15
300 PRINT "[CLR][DOWN][DOWN]READ AND DISPLAY [R
     VS]R-CREDIT[OFF] AND          [RVS]R-CREDIT C
     OPY[OFF]."
310 PRINT : PRINT "WORKING...

RUN

READ AND DISPLAY 'R-CREDIT' AND
'R-CREDIT COPY'.

WORKING...

'R-CREDIT' HAS 5 RECORDS.

'R-CREDIT COPY' HAS 5 RECORDS.

ORIG.: 11111      PAUL HACKER  1
COPY: 11111      PAUL HACKER  1

PRESS 'RETURN' FOR NEXT DATASETS.


ORIG.: 11225      JANET PROGRAM  5
COPY: 11225      JANET PROGRAM  5

PRESS 'RETURN' FOR NEXT DATASETS.


ORIG.: 19467      SUE B. ANTHONY  3
COPY: 19467      SUE B. ANTHONY  3

PRESS 'RETURN' FOR NEXT DATASETS.


ORIG.: 34977      JOHN WILEY   2
COPY: 34977      JOHN WILEY   2

PRESS 'RETURN' FOR NEXT DATASETS.


ORIG.: 94671      KING ARTHUR  4
COPY: 94671      KING ARTHUR  4

PRESS 'RETURN' FOR NEXT DATASETS.

FILES CLOSED, REPORT DONE.
```

Answer Key

1. (a)

```
100 REM   PROB 1A BUS INVENTORY
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (4)
140 REM     P$ = PROD DESCR. (20)
150 REM     S$ = SUPPLIER (20)
160 REM     R  = REORDER PT. (3)
170 REM     Y  = REORDER QUAN (4)
180 REM     Q  = QUAN IN STOCK (4)
190 REM     C  = COST (TO RETAIL) (6)
200 REM     U  = UNIT (RETL) PRICE (6)
210 REM     R$ = USER RESPONSE
220 REM     R1 = RECORD COUNT
230 REM     H  = HI BYTE
240 REM     L  = LOW BYTE
250 :       CR$= CHR$(13)
260 :
270 REM   FILES USED
280 REM     REL FIL: BUSINESS INVEN
290 REM     RECORD LENGTH: 75 BYTES
300 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
310 :
320 REM   INITIALIZE
330 PRINT "[CLR][DOWN][DOWN][DOWN]   BUSINESS I
     NVENTORY RELATIVE FILE"
340 LET R1 = 2
350 OPEN 15, 8, 15
360 OPEN 2, 8, 2, "BUSINESS INVEN,L," + CHR$(75
     )
370 GOSUB 800
380 :
390 REM   DATA ENTRY MODULE
400 REM     DATA ENTRY TESTS OMITTED
410 :
420 PRINT : INPUT "PRODUCT # (4 DIGITS) "; N$
430 REM   (DATA ENTRY TESTS)
440 PRINT : INPUT "PROD. DESCR. (20 CHRS) "; P$

450 REM   (DATA ENTRY TESTS)
460 PRINT : INPUT "SUPPLIER (20 CHRS) "; S$
470 REM   (DATA ENTRY TESTS)
480 PRINT : INPUT "REORDER POINT "; R
490 REM   (DATA ENTRY TESTS)
500 PRINT : INPUT "REORDER QUANTITY "; Y
510 REM   (DATA ENTRY TESTS)
520 PRINT : INPUT "QUANTITY IN STOCK "; Q
530 REM   (DATA ENTRY TESTS)
540 PRINT : INPUT "WHOLESALE COST "; C
550 REM   (DATA ENTRY TESTS)
560 PRINT : INPUT  "UNIT SELLING PRICE "; U
```

```
570 REM    (DATA ENTRY TESTS)
580 :
590 REM    PRINT DATASET TO FILE
600 LET H = INT(R1/256) : LET L = R1 - H*256
610 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
620 PRINT#2, N$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR$;
    Y ;CR$; Q ;CR$; C ;CR$; U
630 :
640 PRINT : PRINT "MORE DATA (PRESS [RVS]Y[OFF]
    OR [RVS]N[OFF]) ?"
650 GET R$: IF R$ <> "Y" AND R$ <> "N" THEN 650

660 IF R$ = "Y" THEN LET R1 = R1 + 1 : PRINT "[
    CLR][DOWN][DOWN]" : GOTO 420
670 :
680 REM    CLOSE FILE
690 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
700 PRINT#2, R1
710 :
720 CLOSE 2 : GOSUB 800 : CLOSE 15
730 PRINT : PRINT "TOTAL DATASETS ="R1 - 1
740 PRINT "FILE CLOSED."
750 END
760 :
770 :
800 REM    DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

1. (b)

```
100 REM    PROB 7-1B:  'BUSINESS INVEN' (REL) RE
    ADER
110 :
120 REM    VARIABLES USED
130 REM      N$ = PROD NUMBER (4)
140 REM      P$ = PROD DESCR. (20)
150 REM      S$ = SUPPLIER (20)
160 REM      R  = REORDER PT. (3)
170 REM      Y  = REORDER QUAN (4)
180 REM      Q  = QUAN IN STOCK (4)
190 REM      C  = COST (TO RETAIL) (6)
200 REM      U  = UNIT (RETL) PRICE (6)
210 REM      R$ = USER RESPONSE
220 REM      R1 = RECORD COUNT
230 REM      H  = HI BYTE
240 REM      L  = LOW BYTE
250 :        CR$= CHR$(13)
260 :
```

```
270 REM   FILES USED
280 REM     REL FIL: BUSINESS INVEN
290 REM     RECORD LENGTH: 75 BYTES
300 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
310 :
320 REM   INITIALIZE
330 PRINT "[CLR][DOWN][DOWN][DOWN]   BUSINESS I
    NVENTORY FILE READER"
340 OPEN 15, 8, 15
350 OPEN 2, 8, 2, "BUSINESS INVEN"
360 GOSUB 800
370 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
380 INPUT#2, R1
390 :
400 REM   READ AND DISPLAY
410 FOR X = 2 TO R1
420 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
    DOWN 2
430 LET H = INT(X/256) : LET L = X - H*256
440 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H)CHR$(1
    )
450 INPUT#2, N$, P$, S$, R, Y, Q, C, U
460 PRINT N$, P$ : PRINT S$
470 PRINT R, Y, Q : PRINT C,  U
480 PRINT : PRINT "PRESS [RVS]RETURN[OFF] FOR N
    EXT DATASET."
490 GET R$ : IF R$ <> CHR$(13) THEN 490
500 NEXT X
510 :
520 :
530 CLOSE 2 : GOSUB 800 : CLOSE 15
540 PRINT : PRINT "TOTAL DATASETS ="R1 - 1
550 PRINT "FILE DISPLAYED AND CLOSED."
560 END
570 :
580 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[DOWN]DISK ERROR: " DX$
840 CLOSE 15 : END
```

1. (c)

```
100 REM   PROB7-1C  SEQ INDEX FILE FOR
110 REM   'BUSINESS INVEN' (REL) FILE
120 :
130 REM   VARIABLES USED
140 REM     N$ = PROD NUMBER (4)
150 REM     P$ = PROD DESCR. (20)
160 REM     S$ = SUPPLIER (20)
170 REM     R  = REORDER PT. (3)
180 REM     Y  = REORDER QUAN (4)
190 REM     Q  = QUAN IN STOCK (4)
200 REM     C  = COST (TO RETAIL) (6)
210 REM     U  = UNIT (RETL) PRICE (6)
220 REM     R$ = USER RESPONSE
230 REM     R1 = RECORD COUNT
240 REM     H  = HI BYTE
250 REM     L  = LOW BYTE
260 :       CR$= CHR$(13)
270 :
280 REM   FILES USED
290 REM     REL FIL: BUSINESS INVEN
300 REM     RECORD LENGTH: 75 BYTES
310 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
320 REM     SEQ FILE: INDEX
330 :
340 REM   INITIALIZE
350 PRINT "[CLR][DOWN][DOWN][DOWN]CREATE INDEX
     FOR 'BUSINESS INVEN' FILE"
360 OPEN 15, 8, 15
370 OPEN 2, 8, 2, "BUSINESS INVEN"
380 GOSUB 800
390 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
     1)
400 INPUT#2, R1
410 :
420 OPEN 5, 8, 5, "INDEX,S,W"
430 GOSUB 800
440 :
450 REM   READ PROD # FROM REL
460 REM   PRINT PROD # AND RECORD # TO
470 REM   'INDEX' SEQ FILE
480 FOR X = 2 TO R1
490 LET H = INT(X/256) : LET L = X - H*256
500 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H)CHR$(1
     )
510 INPUT#2, N$
520 :
530 REM   PRINT TO SEQ FILE
540 PRINT#5, N$ : PRINT#5, X
550 PRINT "[HOME][DOWN][DOWN][DOWN][DOWN][DOWN]
     [DOWN]WRITING:" X; N$
560 NEXT X
570 :
```

```
580 :
590 REM   CLOSE FILES
600 CLOSE 2 : GOSUB 800: CLOSE 5 : GOSUB 800 :
     CLOSE 15
610 PRINT : PRINT "[RVS]INDEX[OFF] FILE CREATED
     ."
620 PRINT "FILES CLOSED."
630 END
640 :
650 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[DOWN][RVS]DISK ERROR: " DX$

840 CLOSE 15 : END
```

1. (d)

```
100 REM   PROB 7-1D: INDEX FILE READER:CREDIT (
     REL)
110 :
120 REM   VARIABLES USED
130 REM     N$ = ACCOUNT NUMBER
140 REM     R1 = RECORD COUNT
150 REM     R$ = USER RESPONSE
160 :
170 REM   FILES USED
180 REM     SEQ FILE: INDEX
190 REM     DATASET: N$, R1
200 :
210 REM   INITIALIZE
220 OPEN  15, 8, 15
230 OPEN 1, 8, 8, "INDEX.S,R"
240 GOSUB 800
250 :
260 REM   READ & DISPLAY
270 PRINT "[CLR][DOWN][DOWN][DOWN]ACCT. #", "RE
     CORD #"
280 INPUT#1, N$, R1
290 LET SV = ST : REM 'GET' SETS ST TO 0
300 PRINT  N$, R1
310 :
320 PRINT : PRINT "PRESS 'RETURN' FOR NEXT DATA
     ."
330 GET R$ : IF R$ <> CHR$(13) THEN 330
340 IF SV =  0 THEN 270
350 :
360 :
370 REM   CLOSE FILE
380 CLOSE 1 : GOSUB 800 : CLOSE 15
390 PRINT : PRINT "FILE  [RVS]INDEX[OFF] READ &
     CLOSED."
```

```
400 END
410 :
420 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

2.

```
100 REM   PROB. 7-2:COPY R-CREDIT (REL)
110 :
120 REM   VARIABLES USED
130 REM     N$ = CUST. # (5 CHRS)
140 REM     C$ = CUST. NAME (20 CHRS)
150 REM     R  = CREDIT RATING (1 CHR)
160 REM     R1 = RECORD COUNT
170 REM     L  = LO BYTE
180 REM     H  = HI BYTE
190 REM     R$ = USER RESPONSE
200 REM     X  = FOR...NEXT VARIABLE
210 :       CR$= CHR$(13)
220 :
230 REM   FILES USED
240 REM     SOURCE REL FILE: R-CREDIT
250 REM     COPY REL FILE: R-CREDIT COPY
260 REM     RECORD LENGTH: 29 BYTES
270 REM     DATASET: N$, C$, R
280 :
290 REM   INITIALIZE
300 OPEN 15, 8, 15
310 PRINT "[CLR][DOWN][DOWN]COPY [RVS]R-CREDIT[
     OFF] TO [RVS]R-CREDIT COPY[OFF]"
320 PRINT : PRINT "WORKING...
330 OPEN 1, 8, 8, "R-CREDIT"
340 GOSUB 800
350 PRINT#15, "S0:R-CREDIT COPY" : GOSUB 800
360 OPEN 2, 8, 2, "R-CREDIT COPY,L," + CHR$(29)

370 GOSUB 800
380 :
390 REM   COPY ROUTINE
400 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
     1)
410 INPUT#1,R1
420 :
430 FOR X = 2 TO R1
440 LET H = INT(X/256) : LET L = X - H*256
450 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
460 INPUT#1, N$, C$, R
470 :
```

```
480 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
490 PRINT#2, N$ ;CR$; C$ ;CR$; R
500 :
510 NEXT X
520 :
530 :
540 REM   CLOSE FILES
550 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(Ø) CHR$(
    1)
560 PRINT#2, R1
570 :
580 CLOSE 1 : GOSUB 8ØØ : CLOSE 2 : GOSUB 8ØØ :
     CLOSE 15
590 PRINT : PRINT "FILES CLOSED, COPY COMPLETE.
    "
6ØØ END
610 :
620 :
8ØØ REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 2Ø OR DX = 5Ø THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

3.

```
1ØØ REM   PROB7-3  READ  & DISPLAY 2 REL FILES
110 :
120 REM   VARIABLES USED
130 REM     N$,N1$ = CUST. # (5 CHRS)
140 REM     C$,C1$ = CUST. NAME (2Ø CHRS)
150 REM     C, C1  = CREDIT RATING (1 CHR)
160 REM     R, R1  = RECORD COUNT
170 REM     L  = LO BYTE
180 REM     H  = HI BYTE
190 REM     R$ = USER RESPONSE
2ØØ REM     X  = FOR...NEXT VARIABLE
210 :
220 REM   FILES USED
230 REM     REL FILE #1: R-CREDIT
240 REM     REL FILE #2: R-CREDIT COPY
250 REM     RECORD LENGTH: 29 BYTES
260 REM     DATASET: N$, C$, C
270 :
280 REM   INITIALIZE
290 OPEN 15, 8, 15
3ØØ PRINT "[CLR][DOWN][DOWN]READ AND DISPLAY [R
    VS]R-CREDIT[OFF] AND          [RVS]R-CREDIT C
    OPY[OFF]."
310 PRINT : PRINT "WORKING...
320 OPEN 1, 8, 8, "R-CREDIT"
330 GOSUB 8ØØ
```

```
340 OPEN 2, 8, 2, "R-CREDIT COPY"
350 GOSUB 800
360 :
370 REM   RECORD COUNTS
380 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1)
390 INPUT#1,R
400 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
410 INPUT#2,R1
420 PRINT : PRINT "[RVS]R-CREDIT[OFF] HAS" R "R
    ECORDS."
430 PRINT : PRINT "[RVS]R-CREDIT COPY[OFF] HAS"
     R1 "RECORDS."
440 :
450 FOR X = 2 TO R
460 LET H = INT(X/256) : LET L = X - H*256
470 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
    1)
480 INPUT#1, N$, C$, C
490 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
500 INPUT#2, N1$, C1$, C1
510 :
520 PRINT :  PRINT "ORIG.: " N$, C$ " " C
530 PRINT "COPY: " N1$, C1$ " " C1
540 PRINT : PRINT "PRESS [RVS]RETURN[OFF] FOR N
    EXT DATASETS."
550 GET R$ : IF R$ <> CHR$(13) THEN 550
560 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN][
    DOWN][DOWN][DOWN][DOWN]" : REM CLR & CRSR DO
    WN 8
570 :
580 NEXT X
590 :
600 :
610 REM   CLOSE FILES
620 :
630 CLOSE 1 : GOSUB 800 : CLOSE 2 : GOSUB 800 :
     CLOSE 15
640 PRINT : PRINT "FILES CLOSED, REPORT DONE."
650 END
660 :
670 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

CHAPTER EIGHT

Relative File Applications

Objectives: In this chapter you will learn expanded techniques for relative data file applications, including how to write or read from a specific byte of a record. You will also learn how to use sequential "index" data files as an index for relative data files.

SEQUENTIAL INDEX FILES FOR RELATIVE FILES

Two file applications are designed to be somewhat typical of the programs you might encounter as you design your own computer software systems and write your own programs. The programs are not really long, as you might expect, but they are only one component of a larger software system composed of many programs.

The first exercise is an inventory control application that uses both a sequential file and a relative file in the same program. The objective is to show how to use a sequential "index" file and how to change data located in a relative file record. The application could as well have been a mailing list, a credit information file, or any sort of master file application. While an index file may be superfluous in our simple example, the technique may be valuable in more complex software systems.

In this case, all the data regarding the inventory of products carried are stored in a relative file named BUSINESS INVEN. Each relative record contains the following data for one item of inventory in the order shown below:

```
N$ = PROD # (4)
P$ = DESCRIPTION (20)
S$ = SUPPLIER (20)
L  = REORDER POINT (3)
Y  = REORDER QUANTITY (4)
Q  = QUANTITY AVAILABLE (4)
C  = COST (6)
U  = UNIT SELLING PRICE (6)
```

If you wanted to change some data from product number 9827, you would have to search through the relative file records one at a time, until you found product number 9827. Alternatively you could add a sequential "index" file that contains the product numbers (in a string variable) followed by the record number where the proper dataset is located in the relative file. To change the cost and selling price data in the random access file, follow these steps:

1. Enter product number.
2. Quickly search the sequential index file for the product number and corresponding record location.
3. Access the correct relative record.
4. Make the changes in the relative file record.

It looks easy, but there are a few "tricks." Here is the first part of the program. Read it through carefully.

```
100 REM   SEQ INDEX FILE TO REL FILE
110 REM   'BUSINESS INVEN'
120 REM   PERMITS USER TO CHANGE COST
130 REM   AND UNIT PRICE FOR EXISTING
140 REM   INVENTORY ITEM
150 :
160 REM   VARIABLES USED
170 REM     R1 = RECORD COUNT
180 REM     N$, N1$, N2$ = PROD # (4)
190 REM     P$ = PROD DESCR (20)
200 REM     S$ = SUPPLIER (20)
210 REM     R  = REORDER POINT (3)
220 REM     Y  = REORDER QUANT (3)
230 REM     Q  = QUANTITY IN STOCK (3)
240 REM     C, C1 = COST (6)
250 REM     U, U1 = UNIT PRICE (6)
260 REM     L  = LO BYTE
270 REM     H  = HI BYTE
280 REM     Z$ = USER RESPONSE
290 :       CR$ = CHR$(13)
300 :
310 REM   FILES USED
320 REM     SEQ FILE: INDEX
330 REM     DATASET: N$, R1
340 REM     REL FILE: BUSINESS INVEN
350 REM     FILE LENGTH: 75 BYTES
360 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
370 :
380 REM   INITIALIZE
390 OPEN 15, 8, 15
400 OPEN 1, 8, 8, "BUSINESS INVEN"
410 GOSUB 1000
420 REM   'INDEX' OPENED AT SEARCH
430 :
```

```
440 REM   DATA ENTRY MODULE
450 PRINT : INPUT "PRODUCT # (4 CHR) "; N2$
460 REM   DATA ENTRY TESTS
470 :
```

This segment provides for entry and testing of the product number. It is time to search the sequential file for the record location for this product number in the relative file. On the chance that the operator made an entry error that escaped the error tests or entered a nonexistent product number, include a provision for reading to the end of the sequential file without finding a matching product number. This error message routine is shown below in lines 570–610. Fill in lines 500–530 and complete line 550 (checking for end of the sequential file).

(a)

```
470 :
480 REM   SEARCH INDEX FILE
490 PRINT : PRINT "SEARCHING FOR RECORD . . ."
500 ____________________________________
510 ____________________________________
520 ____________________________________
530 ____________________________________
540 IF N1$ = N2$ THEN CLOSE 2 : GOTO 650 : REM
     READ REL FILE
550 IF ________________ THEN ________ : REM N
     EXT
560 :
570 REM   END-OF-FILE
580 CLOSE 2
590 PRINT : PRINT "PRODUCT NUMBER [RVS]" N2$ "[
     OFF] NOT IN FILE."
600 PRINT "CHECK NUMBER AND RE-ENTER."
610 GOTO 450
620 :
```

(b) In which variable is the record number of the relative file stored? _____

_____

(c) Under what conditions is the 'INDEX' file closed? _____

_____

_____

(d) Why must it be closed if the product number is not found? _____

_____

_____

_____

(a)
```
470 :
480 REM   SEARCH INDEX FILE
490 PRINT : PRINT "SEARCHING FOR RECORD . . ."
500 OPEN 2, 8, 2, "INDEX,S,R"
510 GOSUB 1000
520 INPUT#2, N1$, R1
530 LET SV = ST
540 IF N1$ = N2$ THEN CLOSE 2 : GOTO 650 : REM
     READ REL FILE
550 IF SV = 0 THEN 520 : REM NEXT
560 :
570 REM   END-OF-FILE
580 CLOSE 2
590 PRINT : PRINT "PRODUCT NUMBER [RVS]" N2$ "[
     OFF] NOT IN FILE."
600 PRINT "CHECK NUMBER AND RE-ENTER."
610 GOTO 450
620 :
```

(b) R1

(c) If the account number entered by the user is found (line 540), or if the end of file is encountered (lines 550–580).

(d) To reset the data counter to the beginning of the sequential file.

Next, the correct dataset is read from the relative file. Fill in lines 650–670.

(a)
```
620 :
630 REM   READ FROM REL FILE
640 :
650 ____________________________________
660 ____________________________________
670 ____________________________________
680 :
```

(a)
```
620 :
630 REM   READ FROM REL FILE
640 :
650 LET H = INT(R1/256) : LET L = R1 - H*256
660 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
670 INPUT#1, N$, P$, S$, R, Y, Q, C1, U1
680 :
```

Complete lines 810 and 820 below.

(a)
```
680 :
690 REM   ENTER DATA CHANGES
700 :
710 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
720 PRINT "PRODUCT: [RVS]" P$
730 PRINT "OLD COST: [RVS]" C1
740 INPUT "NEW COST "; C
750 REM   DATA ENTRY TESTS
760 PRINT "OLD UNIT SELLING PRICE: [RVS]" U1
770 INPUT "NEW UNIT PRICE "; U
780 REM   DATA ENTRY TESTS
790 :
800 REM   REPLACE DATA
810 _____________________________________
820 _____________________________________
830 :
```

(a)
```
680 :
690 REM   ENTER DATA CHANGES
700 :
710 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
720 PRINT "PRODUCT: [RVS]" P$
730 PRINT "OLD COST: [RVS]" C1
740 INPUT "NEW COST "; C
750 REM   DATA ENTRY TESTS
760 PRINT "OLD UNIT SELLING PRICE: [RVS]" U1
770 INPUT "NEW UNIT PRICE "; U
780 REM   DATA ENTRY TESTS
790 :
800 REM   REPLACE DATA
810 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
820 PRINT#1, N$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR$;
      Y ;CR$; Q ;CR$; C ;CR$; U
830 :
```

The remainder of the program looks like this:

```
830 :
840 REM   MORE ?
850 PRINT : PRINT "MORE ENTRIES (PRESS [RVS]Y[O
    FF] OR [RVS]N[OFF]) ?"
860 GET Z$ : IF Z$ <> "Y" AND Z$ <> "N" THEN 86
    0
870 IF Z$ = "Y" THEN 450 : REM ENTER PROD #
880 :
890 REM   CLOSE
900 CLOSE 1 : GOSUB 1000 : CLOSE 15
910 END
920 :
930 :
1000 REM   DISK ERROR ROUTINE
1010 INPUT#15, DX, DX$, DY, DZ : REM READ
1020 IF DX < 20 OR DX = 50 THEN RETURN
1030 PRINT : PRINT "[RVS]DISK ERROR: " DX$
1040 CLOSE 15 : END
```

This completes the first relative file application—one part of an entire product inventory application. Now enter and RUN the program. After that, display the contents of BUSINESS INVEN to verify the changes.

```
100 REM   SEQ INDEX FILE TO REL FILE
110 REM   'BUSINESS INVEN'
120 REM   PERMITS USER TO CHANGE COST
130 REM   AND UNIT PRICE FOR EXISTING
140 REM   INVENTORY ITEM
150 :
160 REM   VARIABLES USED
170 REM     R1 = RECORD COUNT
180 REM     N$, N1$, N2$ = PROD # (4)
190 REM     P$ = PROD DESCR (20)
200 REM     S$ = SUPPLIER (20)
210 REM     R  = REORDER POINT (3)
220 REM     Y  = REORDER QUANT (3)
230 REM     Q  = QUANTITY IN STOCK (3)
240 REM     C, C1 = COST (6)
250 REM     U, U1 = UNIT PRICE (6)
260 REM     L  = LO BYTE
270 REM     H  = HI BYTE
280 REM     Z$ = USER RESPONSE
290 :       CR$ = CHR$(13)
300 :
310 REM   FILES USED
320 REM     SEQ FILE: INDEX
330 REM     DATASET: N$, R1
340 REM     REL FILE: BUSINESS INVEN
350 REM     FILE LENGTH: 75 BYTES
360 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
```

```
370 :
380 REM   INITIALIZE
390 OPEN 15, 8, 15
400 OPEN 1, 8, 8, "BUSINESS INVEN"
410 GOSUB 1000
420 REM   'INDEX' OPENED AT SEARCH
430 :
440 REM   DATA ENTRY MODULE
450 PRINT : INPUT "PRODUCT # (4 CHR) "; N2$
460 REM   DATA ENTRY TESTS
470 :
480 REM   SEARCH INDEX FILE
490 PRINT : PRINT "SEARCHING FOR RECORD . . ."
500 OPEN 2, 8, 2, "INDEX,S,R"
510 GOSUB 1000
520 INPUT#2, N1$, R1
530 LET SV = ST
540 IF N1$ = N2$ THEN CLOSE 2 : GOTO 650 : REM
     READ REL FILE
550 IF SV = 0 THEN 520 : REM NEXT
560 :
570 REM   END-OF-FILE
580 CLOSE 2
590 PRINT : PRINT "PRODUCT NUMBER [RVS]" N2$ "[
     OFF] NOT IN FILE."
600 PRINT "CHECK NUMBER AND RE-ENTER."
610 GOTO 450
620 :
630 REM   READ FROM REL FILE
640 :
650 LET H = INT(R1/256) : LET L = R1 - H*256
660 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
670 INPUT#1, N$, P$, S$, R, Y, Q, C1, U1
680 :
690 REM   ENTER DATA CHANGES
700 :
710 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
720 PRINT "PRODUCT: [RVS]" P$
730 PRINT "OLD COST: [RVS]" C1
740 INPUT "NEW COST "; C
750 REM   DATA ENTRY TESTS
760 PRINT "OLD UNIT SELLING PRICE: [RVS]" U1
770 INPUT "NEW UNIT PRICE "; U
780 REM   DATA ENTRY TESTS
790 :
800 REM   REPLACE DATA
810 PRINT#15, "P" CHR$(8) CHR$(L) CHR$(H) CHR$(
     1)
820 PRINT#1, N$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR$;
      Y ;CR$; Q ;CR$; C ;CR$; U
```

```
830 :
840 REM   MORE ?
850 PRINT : PRINT "MORE ENTRIES (PRESS [RVS]Y[O
     FF] OR [RVS]N[OFF]) ?"
860 GET Z$ : IF Z$ <> "Y" AND Z$ <> "N" THEN 86
     0
870 IF Z$ = "Y" THEN 450 : REM ENTER PROD #
880 :
890 REM   CLOSE
900 CLOSE 1 : GOSUB 1000 : CLOSE 15
910 END
920 :
930 :
1000 REM   DISK ERROR ROUTINE
1010 INPUT#15, DX, DX$, DY, DZ : REM READ
1020 IF DX < 20 OR DX = 50 THEN RETURN
1030 PRINT : PRINT "[RVS]DISK ERROR: " DX$
1040 CLOSE 15 : END
```

(a) What other programs are needed to complete this series of application programs? _____

_____

_____

(a) 1. Add new inventory items, 2. Delete inventory items, 3. Change supplier and/or description, 4. Change reorder point, etc., to name a few.

A UNIVERSAL FILE READER

Now that you've had some practice with more advanced relative files, and are beginning to write programs using both sequential and relative files, you'll find it handy to have a program that will read either kind of file. Here's a simple program to read either a sequential or relative file. It reads a relative file as if it were a sequential file. We'll use the fact that any access to a file moves the file pointer to the next field (record) to read the relative file. The only tricky part is finding the correct way to check for end of file.

(a) Help complete the Introduction and READ part of the program:

```
100 REM UNIV READ
110 INPUT "FILE NAME"; F$
120 INPUT "FILE  TYPE (R/S) "; T$
130 ___________________________________ : REM T
     AKE FIRST CHARACTER OF T$
```

```
140 _____________________________________ : REM O
    PEN DISK CHANNEL
150 _____________________________________ : REM O
    PEN FILE
160 _____________________________________ : REM R
    EAD & PRINT DATA
```

(a)
```
100 REM UNIV READ
110 INPUT "FILE NAME"; F$
120 INPUT "FILE  TYPE (R/S) "; T$
130 LET T$ = LEFT$(T$,1)
140 OPEN 15, 8, 15
150 OPEN 8, 8, 8, F$
160 INPUT#8, A$ : PRINT A$ : LET SV = ST
```

Are you surprised that you can use one OPEN statement to read either a sequential or a relative file? Also notice that we stored the STatus for later use after reading the file in line 160.

To complete the program, we must check the disk channel for any error messages:

```
170 INPUT#15, X
```

Now we check for end of file.

Complete the following statements:

(a) For a sequential file, the end of file is signaled by _____

_____

(b) For a relative file, the end of file is signaled by _____

_____

(a) STatus changes from 0 to 64.
(b) The disk status, read from the disk channel, is 50 (RECORD NOT PRESENT).

Using this information, we can easily complete the checks for end of file. Then the files are closed. Here's the complete program. It may not read all the data in a relative file dataset (for example, if carriage returns are present within a record). For a more complete look at a file, you will need to use a program which uses GET (such as we developed for sequential files in Chapter 6, and for relative files in Chapter 7).

```
100 REM UNIV READ
110 INPUT "FILE NAME"; F$
120 INPUT "FILE  TYPE (R/S) "; T$
130 LET T$ = LEFT$(T$,1)
140 OPEN 15, 8, 15
150 OPEN 8, 8, 8, F$
160 INPUT#8, A$ : PRINT A$ : LET SV = ST
170 INPUT#15, X
180 IF T$ = "S" AND SV = 0 THEN 160: REM SEQ E-
    O-F
190 IF T$ = "R" AND X <> 50 THEN 160: REM  REL
    E-O-F
200 CLOSE 8 : CLOSE 15
210 PRINT "FILE READ"
```

WRITING AND READING WITHIN A RELATIVE FILE RECORD

Commodore's relative file structure allows you to write (or read) data directly from within a record, starting at any location you wish. So far, we've always set the position pointer to location 1, so we've always read or written starting at the first byte of a record. The two programs following are revisions of the program "INVEN" from Chapter 7 illustrating the use of the position pointer to read and write data within a record. If you use this feature, you must remember that when you write, you write to the remainder of the record. Look at the following illustration:

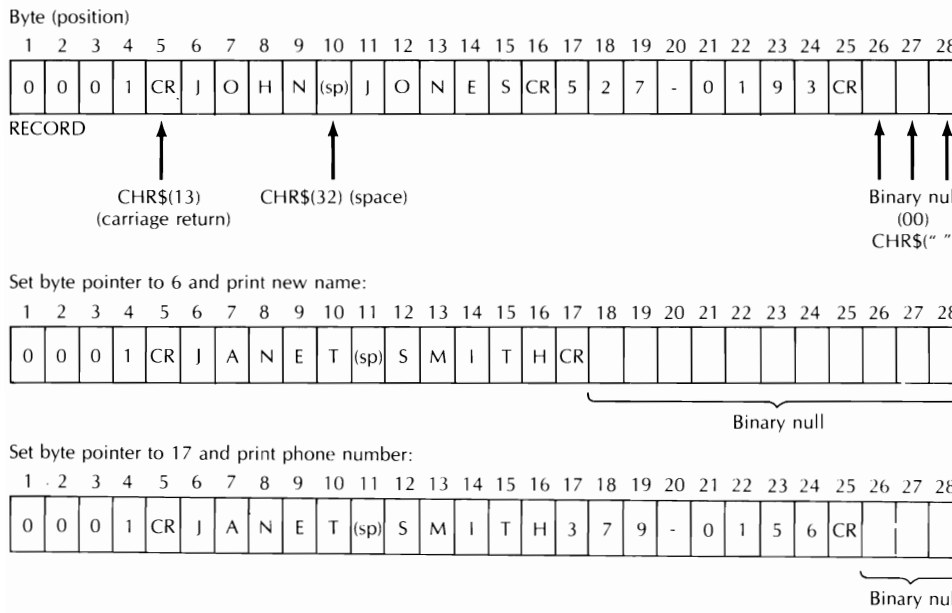


Figure 8–1. When using the byte pointer, as this illustration shows, each item in the dataset must have a fixed size. *Preferably* the dataset is concatenated so it can be read with one INPUT# statement (carriage returns are not needed to separate items within a record).

As you can see, if you want to change one part of the record, you must write to all following parts or lose those data.

This capability to write within a record is most useful if you write your data to the file as a concatenated dataset, without carriage returns. While it allows you to save some space, the added complications and problems of carefully locating the byte pointer to the correct information may not be worth it. Using MID$ to separate data from a concatenated dataset within a program is no problem. Writing and reading concatenated datasets avoids intrarecord zeros and other nasty surprises.

We recommend that you build a relative file record in memory, then write the entire, concatenated dataset to the disk at once. The entire record is then read with one INPUT# statement, and MID$ is used within the program to separate component data.

```
100 REM   INVENTORY RELATIVE FILE
110 REM   EXAMPLE OF USING BYTE
120 REM    POINTER & FIXED FIELDS
130 :
140 REM   VARIABLES USED
150 REM     N$ = PROD NUMBER (6)
160 REM     P$ = PROD DESCR (20)
170 REM     Q  = QUANTITY (<=999)
180 REM     R1 = RECORD COUNT
190 REM     H  = HI BYTE POINTER
200 REM     L  = LO BYTE POINTER
210 REM     P()= BYTE LOC. POINTER
220 REM     R$ = USER RESPONSE
230 :
240 REM   FILES USED
250 REM     RELATIVE FILE: INVEN
260 REM     RECORD SIZE: 30  BYTES
270 REM     DATASET: N$, P$, Q
280 REM     POSITIONS: 1, 7, 27
290 :
300 REM   INITIALIZE
310 LET R1 = 2 : REM RECORD POINTER
320 OPEN 15, 8, 15 : REM DISK CHANNEL
330 OPEN 1, 8, 8, "INVEN1,L," + CHR$(32)
340 GOSUB 800 : IF DX = 63 THEN CLOSE 1 : PRINT
    #15, "S0:INVEN1" : GOTO 330
350 P(1) = 1 : P(2) = 7 :  P(3) = 27 : REM POIN
    TERS
360 :
370 REM   DATA ENTRY MODULE
380 :
390 PRINT "[CLR][DOWN][DOWN]" : REM CLEAR & CRS
    R DOWN 2
```

```
400 PRINT : INPUT "PRODUCT NUMBER (6) "; N$(1)
410 REM   DATA ENTRY TESTS
420 PRINT : INPUT "PROD. DESCR. (20) "; N$(2)
430 REM   DATA ENTRY TESTS
440 PRINT : INPUT "QUANTITY (<=999) "; N$(3)
450 REM   DATA ENTRY TESTS
460 :
470 REM   PRINT TO FILE
480 LET H = INT(R1/256) : LET L = R1 - INT(H*25
    6) : REM SET HI/LO BYTES
490 FOR I = 1 TO 3
500 PRINT#15,"P" CHR$(8) CHR$(L) CHR$(H) CHR$(P
    (I)) : GOSUB 800
510 PRINT#1, N$(I)
520 NEXT I
530 :
540 PRINT :  PRINT "MORE ENTRIES ([RVS]Y[OFF] O
    R [RVS]N[OFF]) ?"
550 GET R$ : IF R$ <> "Y" AND  R$ <> "N" THEN 5
    50
560 IF R$ = "N" THEN 630
570 :
580 REM   INCREASE RECORD COUNT
590 LET R1 = R1 + 1
600 GOTO 390
610 :
620 REM   CLOSE FILE
630 PRINT#15, "P" CHR$(8) CHR$(1) CHR$(0) CHR$(
    1) : PRINT#1, R1
640 CLOSE 1 : GOSUB 800 : CLOSE 15
650 PRINT : PRINT "FILE CLOSED."
660 END
670 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, X, X$, Y, Z : REM READ
820 IF X < 20 OR X = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: "X$
840 CLOSE 15 : END

100 REM   INVEN READ/PRINT (REL)
105 REM   USING LOC POINTER
110 :
120 REM   VARIABLES USED
130 REM     N$ = PROD NUMBER (6)
140 REM     P$ = PROD DESCR (20)
150 REM     Q  = QUANTITY (<=999)
160 REM     R1 = RECORD COUNT
170 REM     H  = HI BYTE POINTER
180 REM     L  = LO BYTE POINTER
190 REM     R$ = USER RESPONSE
195 REM     P()= BYTE LOC POINTER
200 :
210 REM   FILES USED
```

```
220 REM      RELATIVE FILE: INVEN
230 REM      RECORD SIZE: 30  BYTES
240 REM      DATASET: N$, P$, Q
245 REM      POSITIONS: 1, 7, 27
250 :
260 REM    INITIALIZE
270 OPEN 15, 8, 15 : REM DISK CHANNEL
280 OPEN 2, 8, 2, "INVEN1"
290 GOSUB 800
295 :  P(1) = 1 : P(2) = 7 : P(3) = 27 : REM BY
    TE POSITIONS
300 :
310 REM    PRINT HEADING
320 PRINT "[CLR][DOWN][DOWN][DOWN]PROD #" TAB(1
    0) "PROD DESCR." TAB(26) "QUANTITY" : PRINT
330 :
340 REM    FILE READ/PRINT
350 :
360 PRINT#15,"P" CHR$(2) CHR$(1) CHR$(0) CHR$(1
    )
370 INPUT#2, R1
380 :
390 FOR  X = 2 TO R1
400 : LET H = INT(X/256) : LET L = X - H*256
405 :: FOR I = 1 TO 3
410 ::    PRINT#15,"P" CHR$(2) CHR$(L) CHR$(H) C
    HR$(P(I))
420 ::    INPUT#2, N$(I)
425 :: NEXT I
430 : PRINT LEFT$(N$(1),6) TAB(10) LEFT$(N$(2),
    20) TAB(31) N$(3)
431 REM LEFT$ LIMITS STRING IF NO SPACES IN CON
    CATENATED RECORD
440 NEXT X
450 :
460 REM    CLOSE FILES
470 CLOSE 1 : GOSUB 800 : CLOSE 15
480 PRINT : PRINT "FILE READ"
490 END
500 :
800 REM    DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

In both programs, we have used a FOR NEXT loop to write each part of the dataset, using an array which stores the location of the first byte of each datum within the dataset. This information can often be used effectively if it is stored in a DATA statement and READ within the FOR NEXT loop. The program Mailing List Read and Write in the 1541 User's Manual section on Relative Files illustrates this technique. (However, the version in the second edition had a typo which had to be corrected before it would actually run properly.)

PERSONAL MONEY MANAGEMENT APPLICATION

The second example program in this chapter could form part of a large home financial management software package. The example gives some hints for setting up your own home finance programs. The objectives of this application are to show you how to process a "transaction" file and to demonstrate how account numbers can be used to point out the file and record in a relative file.

The first step is to decide exactly what expenditures you want to computerize. Record all income and all expenditures into particular accounts. Include the capability to discern taxable from non-taxable items so these records can be used as data for your income tax returns. To keep things simple, the following chart of accounts has been prepared for this application:

```
1001    TAXABLE SALARIES
1002    TAXABLE INTEREST
1003    TAXABLE DIVIDENDS
1004    TAXABLE OTHER INC.
1005    NON-TAXABLE INCOME
1006    MISC. NON-TAXABLE

2001    GROCERIES
2002    NON FOOD STAPLES
2003    MORTGAGE
2004    GAS/ELECTRICITY
2005    WATER & GARBAGE
2006    TELEPHONE
2007    HOME INSURANCE
2008    PROPERTY TAXES
2009    FURNITURE
2010    AUTO PAYMENTS
2011    GAS AND OIL
2012    AUTO REPAIR
2013    PARKING/TOLLS
2014    AUTO INSURANCE
2015    FATHER'S CLOTHES
2016    MOTHER'S CLOTHES
2017    SON'S CLOTHES
2018    DAUGHTER'S CLOTHES
2019    CLOTHES CLEANING
2020    SPORTS FEES/TICKETS
2021    SPORTS EQUIPMENT
2022    MAGAZINES/BOOKS
2023    MOVIES/PLAYS
2024    ALCOHOL
2025    DINING OUT
2026    VACATION EXPENSES
2027    POSTAGE
```

```
2028    SCHOOL/HOUSE EXP.
3001    LEGAL/ACCTG FEES
3002    LIFE INSURANCE
3003    MEDICAL INSURANCE
3004    DENTAL INSURANCE
3005    MEDICAL EXPENSES
3006    DRUG EXPENSES
3007    EDUC. FEES/TUITION
3008    BOOKS & SUPPLIES
3009    EXCESS SALES TAX
3010    CONTRIBUTIONS
3011    SAVINGS DEPOSITS
3012    INVESTMENTS
```

The account number has important significance. The first digit of the account number is the number of the relative file in which the account details can be found. All relative files are called BUDGET #. The details of the taxable salaries account are found in file BUDGET1 (account number 1001). The details of the telephone account are in file BUDGET2 (account number 2006).

(a) Which file contains the details of the dining out account? _____

- - - - - - - - - - - - - - - - - - -

(a) BUDGET2 (account number 2006).

The last three digits of the account number indicate the record number of the relative file containing the account details. The investment account (3010) will be found in the file BUDGET3, record number 10.

(a) The legal/accounting account details are found in file _____,

record number _____.

- - - - - - - - - - - - - - - - - - -

(a) BUDGET2, record 30.

For convenience, the account number is always entered as a string variable so that you can use the LEFT$ and RIGHT$ functions to separate the file number and record number.

To demonstrate the file number concept, we use three separate files (BUDGET1, BUDGET2, and BUDGET3) for this small list of accounts. Of course, all these accounts could be placed in one file, but that will not be the case when your account list grows. At that point you may want to use this scheme.

The relative files (BUDGET#) contain the details of each account. Each record contains the following information in the order shown.

```
N$ = ACCOUNT # (4)
A$ = ACCOUNT NAME (20)
B$ = BUDGETED AMOUNT (8). ANNUAL BUDGET
E$ = EXPENDED/EARNED AMOUNT (8). YEAR-TO-DATE
```

Write one program that you can use to create three relative file named BUDGET1, BUDGET2, and BUDGET3, using the dataset shown above as the format in each record. Using the chart of accounts we have provided, enter the correct number of datasets (one per record) for each file; i.e., six records in BUDGET1, twenty-eight records in BUDGET2, and twelve records in BUDGET3. Use the value of the rightmost three digits of the account chart number (N$) to determine the record number into which each dataset will be placed. You decide on the value for BUDGETED AMOUNT in each record, and enter zero (0) as the value for EXPENDED/EARNED amount in all records in all files (happy new fiscal year). Also write the companion program to display the contents of the file one dataset at a time.

```
100 REM   CREATE BUDGET# REL FILES
110 :
120 REM   VARIABLES USED
130 REM     N$ = ACCOUNT CHART # (4)
140 REM     A$ = ACCOUNT NAME (20)
150 REM     B$ = BUDGETED AMT (8)
160 REM     E$ = EXPENDED/EARNED AMT (8)
170 REM     R1 = RECORD COUNTER
180 :       CR$ = CHR$(13)
190 REM     L  = LO BYTE
200 REM     H  = HI BYTE
210 :
220 REM    FILES USED
230 REM     REL FILE: BUDGET1, 2, 3
240 REM     DATASET: N$, A$, B, E
250 REM     RECORD LENGTH: 44
260 :
270 REM   INITIALIZE
```

```
100 REM   CREATE BUDGET# REL FILES
110 :
120 REM   VARIABLES USED
130 REM     N$ = ACCOUNT CHART # (4)
140 REM     A$ = ACCOUNT NAME (20)
150 REM     B$ = BUDGETED AMT (8)
160 REM     E$ = EXPENDED/EARNED AMT (8)
170 REM     R1 = RECORD COUNTER
180 :       CR$ = CHR$(13)
190 REM     L  = LO BYTE
200 REM     H  = HI BYTE
210 :
220 REM    FILES USED
230 REM     REL FILE: BUDGET1, 2, 3
```

```
240 REM      DATASET: N$, A$, B, E
250 REM      RECORD LENGTH: 44
260 :
270 REM    INITIALIZE
280 LET R1 = 2
290 OPEN 15, 8, 15, "I0"
300 PRINT "[CLR][DOWN][DOWN][DOWN] CREATE BUDGE
     T FILES"
310 PRINT : INPUT "WHICH BUDGET FILE (1, 2, OR
     3) "; F$
320 REM    DATA ENTRY TESTS
330 LET F$ = "BUDGET" + F$
340 OPEN 2, 8, 2, F$ + ",L," + CHR$(44)
350 GOSUB 800
360 :
370 REM    ENTER DATA
380 PRINT : INPUT "ACCOUNT NUMBER (4) "; N$
390 REM    DATA ENTRY TESTS
400 PRINT : INPUT "ACCOUNT NAME (20) "; A$
410 REM    DATA ENTRY TESTS
420 PRINT : INPUT "BUDGETED AMT. (8) "; B$
430 REM    DATA ENTRY TESTS
440 PRINT : INPUT "EXP/EARN AMT. (8) "; E$
450 REM    DATA ENTRY TESTS
460 :
470 REM    PRINT TO FILE
480 LET H = INT(R1/256) : LET L = R1 - H*256
490 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
500 GOSUB 800
510 PRINT#2, N$ ;CR$; A$ ;CR$; B$ ;CR$; E$
520 PRINT : PRINT "MORE DATA (PRESS 'Y' OR 'N')
      ?"
530 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 53
     0
540 IF R$ = "N" THEN 590: REM CLOSE
550 LET R1 = R1 + 1
560 GOTO 380 : REM NEXT DATASET
570 :
580 REM    CLOSE FILE
590 CLOSE 2 : GOSUB 800: CLOSE 15
600 PRINT : PRINT "FILE READ AND CLOSED."
610 END
620 :
630 :
800 REM    DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

```
100 REM   READ BUDGET# REL FILES
110 :
120 REM   VARIABLES USED
130 REM     N$ = ACCOUNT CHART # (4)
140 REM     A$ = ACCOUNT NAME (20)
150 REM     B$ = BUDGETED AMT (8)
160 REM     E$ = EXPENDED/EARNED AMT (8)
170 REM     R1 = RECORD COUNTER
180 :       CR$ = CHR$(13)
190 REM     L  = LO BYTE
200 REM     H  = HI BYTE
210 :
220 REM    FILES USED
230 REM     REL FILE: BUDGET1, 2, 3
240 REM     DATASET: N$, A$, B, E
250 REM     RECORD LENGTH: 44
260 :
270 REM   INITIALIZE
280 LET R1 = 2
290 OPEN 15, 8, 15, "I0"
300 PRINT "[CLR][DOWN][DOWN][DOWN] READ BUDGET
    FILES"
310 PRINT : INPUT "WHICH BUDGET FILE (1, 2, OR
    3) "; F$
320 REM   DATA ENTRY TESTS
330 LET F$ = "BUDGET" + F$
340 OPEN 2, 8, 2, F$
350 GOSUB 800
360 :
370 REM   READ FILE
380 LET H = INT(R1/256) : LET L = R1 - H*256
390 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
400 GOSUB 800 : IF DX = 50 THEN 500
410 INPUT#2, N$, A$, B$, E$
415 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN]" : REM
     CLR & CRSR DOWN 4
420 PRINT "RECORD #" R1 - 1 ":"
430 PRINT N$, A$ : PRINT B$, E$
440 PRINT : PRINT "PRESS [RVS]RETURN[OFF] TO CO
    NTINUE."
450 GET R$ : IF R$ <> CR$ THEN 450
460 LET R1 = R1 + 1
470 GOTO 380
480 :
490 REM   CLOSE FILE
500 CLOSE 2 : GOSUB 800 : CLOSE 15
510 PRINT : PRINT "FILE READ AND CLOSED."
520 END
530 :
540 :
```

```
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

(a) In line 420, the record number displayed was R1 – 1. Why? _____

_____

_____

(a) The actual number of records with data is one less than the count, because one record (the first) is used to store the number of records (R1).

You have now created the budget files for the personal money management system of programs. A second set of files is needed to store data on all money transactions. Each month a new sequential transaction file is created containing the information found in your checking account check register. For the month of January, the file is called MONTH1. March is MONTH3, etc. You may keep "old" files on your disk for other analyses you may want to do. Each month you will create a transaction file, then process or "post" it to the BUDGET # file. Each sequential transaction file entry includes the following information in the order shown:

```
C = CHECK #/DEPOSIT SLIP #
Y$ = DATE (6)
W$ = PARTY TO WHOM CHECK IS DRAWN/SOURCE OF FUNDS (20)
A$ = ACCOUNT # (4)
D = DOLLAR AMOUNT
```

Notice that the format is set up to be used with deposits and payments and that the transaction file includes more information than you will actually be using. This file, however, can be used for other things as well, so all this information is included.

(a) Using the dataset information above as a guide, write a program that allows you to create the sequential monthly transaction file. Use your checkbook register or your imagination for the monthly checks and deposits to enter in the file. Then write the companion program to display MONTH#, using the "PRESS RETURN TO CONTINUE" technique.

```
100 REM   CREATE CHECKBOOK SEQ FILE
110 REM    FOR EACH MONTH OF YEAR
120 :
130 REM   VARIABLES USED
140 REM     C  = CHECK/DEPOSIT # (3)
150 REM     Y$ = DATE (XX-XX-XX)(8)
160 REM     W$ = CHECK PARTY OR SOURCE OF FUNDS
     (20)
170 REM     N$ = ACCOUNT # (4)
180 REM     D  = DOLLAR AMT (8)
190 REM     M  = USER-ENTERED MONTH#
200 REM     F$ = FILE NAME (USER ENTERED)
210 REM     R$ = USER RESPONSE
220 :
230 REM   FILES
240 REM     SEQ: MONTH#
250 REM     DATASET: C, Y$, W$, N$, D
260 :
270 REM   INITIALIZE
280 OPEN 15, 8, 15
290 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
```

```
100 REM   CREATE CHECKBOOK SEQ FILE
110 REM    FOR EACH MONTH OF YEAR
120 :
130 REM   VARIABLES USED
140 REM     C  = CHECK/DEPOSIT # (3)
150 REM     Y$ = DATE (XX-XX-XX)(8)
160 REM     W$ = CHECK PARTY OR SOURCE OF FUNDS
    (20)
170 REM     N$ = ACCOUNT # (4)
180 REM     D  = DOLLAR AMT (8)
190 REM     M  = USER-ENTERED MONTH#
200 REM     F$ = FILE NAME (USER ENTERED)
210 REM     R$ = USER RESPONSE
220 :
230 REM   FILES
240 REM     SEQ: MONTH#
250 REM     DATASET: C, Y$, W$, N$, D
260 :
270 REM   INITIALIZE
280 OPEN 15, 8, 15
290 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
    DOWN 2
300 PRINT : INPUT "MONTH NUMBER "; M
310 REM   DATA ENTRY CHECKS
320 LET R$ = STR$(M)
330 LET R$ = RIGHT$(R$,LEN(R$)-1) : REM STRIP P
    RECEEDING SPACE
340 OPEN 1, 8, 8, "MONTH" + R$ + ",S,W"
350 GOSUB 800
360 :
370 REM   DATA ENTRY
```

```
380 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]"
    : REM CLR & CRSR DOWN 5
390 PRINT : INPUT "CHECK/DEPOSIT # "; C
400 REM   DATA ENTRY TESTS
410 PRINT : INPUT "DATE (XX-XX-XX) "; Y$
420 REM   DATA ENTRY TESTS
430 PRINT : INPUT "PARTY/SOURCE (20) "; W$
440 REM   DATA ENTRY TESTS
450 PRINT : INPUT "ACCT. # "; N$
460 REM   DATA ENTRY TESTS
470 PRINT : INPUT "DOLLAR AMT. "; D
480 REM   DATA ENTRY TESTS
490 :
500 REM   PRINT TO FILE
510 PRINT#1, C : PRINT#1, Y$ : PRINT#1, W$ : PR
    INT#1, N$ : PRINT#1, D
520 :
530 REM   MORE
540 PRINT : PRINT "MORE DATA (PRESS [RVS]Y[OFF]
    OR [RVS]N[OFF]) ?"
550 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 55
    0
560 IF R$ = "Y" THEN 380 : REM NEXT
570 :
580 REM   CLOSE FILES
590 CLOSE 1 : GOSUB 800 : CLOSE 15
600 END
610 :
620 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END

100 REM   READ CHECKBOOK SEQ FILE
110 REM    FOR ANY MONTH OF YEAR
120 :
130 REM   VARIABLES USED
140 REM     C  = CHECK/DEPOSIT # (3)
150 REM     Y$ = DATE (XX-XX-XX)(8)
160 REM     W$ = CHECK PARTY OR SOURCE OF FUNDS
    (20)
170 REM     N$ = ACCOUNT # (4)
180 REM     D  = DOLLAR AMT (8)
190 REM     M  = USER-ENTERED MONTH#
200 REM     F$ = FILE NAME (USER ENTERED)
210 REM     R$ = USER RESPONSE
220 :
230 REM   FILES
240 REM     SEQ: MONTH#
250 REM     DATASET: C, Y$, W$, N$, D
```

```
260 :
270 REM   INITIALIZE
280 OPEN 15, 8, 15
290 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
    DOWN 2
300 PRINT "READ MONTHLY CHECK RECORD"
310 PRINT : INPUT "MONTH NUMBER "; M
320 REM   DATA ENTRY CHECKS
330 LET R$ = STR$(M)
340 LET R$ = RIGHT$(R$,LEN(R$)-1) : REM STRIP P
    RECEEDING SPACE
350 OPEN 1, 8, 8, "MONTH" + R$ + ",S,R"
360 GOSUB 800
370 :
380 REM   READ FILE
390 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]"
      : REM CLR & CRSR DOWN 5
400 INPUT#1, C, Y$, W$, N$, D
410 LET SV = ST
420 PRINT : PRINT C, Y$, W$
430 PRINT N$, D
440 :
450 PRINT : PRINT "PRESS [RVS]RETURN[OFF] TO CO
    NTINUE"
460 GET R$ : IF R$ <> CHR$(13) THEN 460
470 IF SV = 0 THEN 390
480 :
490 REM   CLOSE FILES
500 CLOSE 1 : GOSUB 800 : CLOSE 15
510 PRINT : PRINT "FILE READ AND CLOSED."
520 END
530 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

Note the sleight-of-hand in lines 330 and 340 to convert the numeric value to a string. It is easier to simply use a string variable in the INPUT statement. The RIGHT$ function is necessary, because STR$ converts a number to a string consisting of either a space (for a positive number) or minus sign, then the digits of the number. For positive numbers, there will be a leading space. If we had simply used the STR$ function, the result in F$ would be "MONTH 3." The program would never find the file, since the name used was MONTH3. If you forget about the extra space, you can get into trouble.

Let's review the application. Each year, create relative files (BUDGET#) that contain the beginning status of all your personal accounts. This status includes a yearly budget estimate. Each month create a sequential file (MONTH#) using the information found in your checkbook register. After the MONTH# file is completed, process or post it to the BUDGET# files. Periodically, you can print a status report of the BUDGET# files.

The task is to write the program that processes the monthly transaction file. Here is the introductory module with the file initialization module:

```
100 REM   PERSONAL MONEY MANAGEMENT
110 REM    SEQ/REL FILE APPLICATION
120 :
130 REM   VARIABLES USED
140 REM     N$, N1$ = ACCOUNT # (4)
150 REM     A$ = ACCOUNT NAME (20)
160 REM     Y$ = DATE (8)
170 REM     W$ = PARTY/SOURCE (20)
180 REM     M  = USER ENTERED MONTH
190 REM     N  = BUDGET FILE# (FROM N$)
200 REM     C  = CHECK/DEPOSIT # (3)
210 REM     D  = DOLLAR AMT (6)
220 REM     B$ = BUDGETED AMT (8)
230 REM     E$ = EARNED/EXPENDED TO DATE (8)
240 REM     F$ = SEQ FILE NAME
250 REM     F1$ = REL FILE NAME
260 REM     R1 = RECORD # (FROM N$)
270 REM     Z$ = USER RESPONSE
280 REM     L  = LO BYTE
290 REM     H  = HI BYTE
300 :       CR$ = CHR$(13)
310 :
320 REM   FILES USED
330 REM     SEQ: MONTH# (TRANSACTION FILE; # IS
      USER SELECTED)
340 REM     DATASET: C, Y$, W$, A$, D
350 REM     REL: BUDGET# (# FROM N$)
360 REM     DATASET:  N$, A$, B$, E$
370 REM     RECORD LENGTH: 44
380 :
390 REM   INITIALIZATION
400 OPEN 15, 8, 15
410 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
     DOWN 2
420 PRINT "PROCESS SEQ. TRANSACTION FILE"
430 PRINT : INPUT "WHICH MONTH "; M
440 REM   DATA ENTRY TESTS
450 LET Z$ = STR$(M) : LET Z$ = RIGHT$(Z$,LEN(Z
     $)-1) : REM REMOVE SPACE
460 LET F$ = "MONTH" + Z$
470 PRINT : PRINT "WORKING..."
480 :
```

(a) In lines 430 through 460, if the user enters 3 for M, what will the file name (F$) be in line 500? _____

(a) MONTH3

Line 540 stores the STatus value to test for the end of the transaction file (in line 820). When all datasets in that file have been read, the program terminates. Line 520 reads an entire dataset from the transaction file. Then the file number is extracted (in line 570) from the account number, to be used in line 580 to open the file to the correct BUDGET# file. Complete line 570, extracting the file number from the account number (it's the first digit of N$), and concatenating it with the word "BUDGET".

(a) 570 LET F1$ = _____

(a) 570 LET F1$ = "BUDGET" + LEFT$(N$,1)

```
480 :
490 REM   READ SEQ FILE
500 OPEN 1, 8, 8, F$ + ",S,R"
510 GOSUB 1000
520 INPUT#1, C, Y$, W$, N$, D
530 PRINT N$, W$ ,D
540 LET SV = ST
550 :
560 REM   FIND FILE#/OPEN REL FIL
570 LET F1$ = "BUDGET" + LEFT$(N$,1)
580 OPEN 2, 8, 2, F1$
590 :
```

The next operation extracts the record number from the account number (the last three digits of N$).

Complete line 610:

(a) 610 LET R1 = _____

(a) 610 LET R1 = VAL(RIGHT$(N$,3))

(This time, we need it as a value, since we need to use it to calculate the hi and lo bytes in the POINTER statement. Warning: Don't forget the double closing parentheses!)

The remaining module accesses the proper relative file and record, updates the amount expended/earned, and prints the new value back to the file. Complete this module (lines 640–660, 690, 710, 750 and 760, 790, and line 820).

(a)
```
620 :
630 REM   READ REL FILE RECORD
640 _____________________________________
650 _____________________________________
660 _____________________________________
670 :
680 REM   MAKE CHANGES TO DATA
690 _____________________________________
700 LET E = E + D
710 _____________________________________
720 PRINT "UPDATED BALANCE: " E
730 :
740 REM   UPDATE BUDGET# FILE
750 _____________________________________
760 _____________________________________
770 :
780 REM   CLOSE BUDGET FILE
790 _____________________________________
800 :
810 REM   RETURN FOR NEXT TRANSACTION
820 IF _________________ THEN ________ : REM READ
      NEXT SEQ DATASET
830 :
840 REM   CLOSE SEQ FILE
850 CLOSE 1 : GOSUB 1000: CLOSE 15
860 PRINT : PRINT "TRANSACTIONS POSTED AND FILE
     S CLOSED."
870 END
880 :
```

(a)
```
620 :
630 REM   READ REL FILE RECORD
640 LET H = INT(R1/256) : LET L = R1 - H*256
650 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
660 INPUT#2, N1$, A$, B$, E$
670 :
680 REM   MAKE CHANGES TO DATA
690 LET E = VAL(E$)
700 LET E = E + D
710 LET E$ = STR$(E) : LET E$ = RIGHT$(E$,LEN(E
     $) - 1)
720 PRINT "UPDATED BALANCE: " E
730 :
740 REM   UPDATE BUDGET# FILE
750 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
760 PRINT#2, N1$ ;CR$; A$ ;CR$; B$ ;CR$; E$
770 :
780 REM   CLOSE BUDGET FILE
```

```
790 CLOSE 2 : GOSUB 1000
800 :
810 REM   RETURN FOR NEXT TRANSACTION
820 IF SV = 0 THEN 520 : REM READ NEXT SEQ DATA
     SET
830 :
840 REM   CLOSE SEQ FILE
850 CLOSE 1 : GOSUB 1000: CLOSE 15
860 PRINT : PRINT "TRANSACTIONS POSTED AND FILE
     S CLOSED."
870 END
880 :
```

This completes the program. It will continue reading checking transactions and processing them until the end of the transaction file is reached, at which point files are closed and the program ends. This program keeps your disk drive working, but does nothing on your screen or printer.

Enter and RUN the program, then read and display the BUDGET# files to see the posted and updated accounts.

```
100 REM   PERSONAL MONEY MANAGEMENT
110 REM    SEQ/REL FILE APPLICATION
120 :
130 REM   VARIABLES USED
140 REM     N$, N1$ = ACCOUNT # (4)
150 REM     A$ = ACCOUNT NAME (20)
160 REM     Y$ = DATE (8)
170 REM     W$ = PARTY/SOURCE (20)
180 REM     M  = USER ENTERED MONTH
190 REM     N  = BUDGET FILE# (FROM N$)
200 REM     C  = CHECK/DEPOSIT # (3)
210 REM     D  = DOLLAR AMT (6)
220 REM     B$ = BUDGETED AMT (8)
230 REM     E$ = EARNED/EXPENDED TO DATE (8)
240 REM     F$ = SEQ FILE NAME
250 REM     F1$ = REL FILE NAME
260 REM     R1 = RECORD # (FROM N$)
270 REM     Z$ = USER RESPONSE
280 REM     L  = LO BYTE
290 REM     H  = HI BYTE
300 :       CR$ = CHR$(13)
310 :
320 REM   FILES USED
330 REM     SEQ: MONTH# (TRANSACTION FILE; # IS
      USER SELECTED)
340 REM     DATASET: C, Y$, W$, A$, D
350 REM     REL: BUDGET# (# FROM N$)
360 REM     DATASET:  N$, A$, B$, E$
370 REM     RECORD LENGTH: 44
380 :
390 REM   INITIALIZATION
```

```
400 OPEN 15, 8, 15
410 PRINT "[CLR][DOWN][DOWN]" : REM CLR & CRSR
    DOWN 2
420 PRINT "PROCESS SEQ. TRANSACTION FILE"
430 PRINT : INPUT "WHICH MONTH "; M
440 REM   DATA ENTRY TESTS
450 LET Z$ = STR$(M) : LET Z$ = RIGHT$(Z$,LEN(Z
    $)-1) : REM REMOVE SPACE
460 LET F$ = "MONTH" + Z$
470 PRINT : PRINT "WORKING..."
480 :
490 REM   READ SEQ FILE
500 OPEN 1, 8, 8, F$ + ",S,R"
510 GOSUB 1000
520 INPUT#1, C, Y$, W$, N$, D
530 PRINT N$, W$ ,D
540 LET SV = ST
550 :
560 REM   FIND FILE#/OPEN REL FIL
570 LET F1$ = "BUDGET" + LEFT$(N$,1)
580 OPEN 2, 8, 2, F1$
590 :
600 REM   CONVERT RECORD #
610 LET R1 = VAL(RIGHT$(N$,3))
620 :
630 REM   READ REL FILE RECORD
640 LET H = INT(R1/256) : LET L = R1 - H*256
650 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
660 INPUT#2, N1$, A$, B$, E$
670 :
680 REM   MAKE CHANGES TO DATA
690 LET E = VAL(E$)
700 LET E = E + D
710 LET E$ = STR$(E) : LET E$ = RIGHT$(E$,LEN(E
    $) - 1)
720 PRINT "UPDATED BALANCE: " E
730 :
740 REM   UPDATE BUDGET# FILE
750 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
    1)
760 PRINT#2, N1$ ;CR$; A$ ;CR$; B$ ;CR$; E$
770 :
780 REM   CLOSE BUDGET FILE
790 CLOSE 2 : GOSUB 1000
800 :
810 REM   RETURN FOR NEXT TRANSACTION
820 IF SV = 0 THEN 520 : REM READ NEXT SEQ DATA
    SET
830 :
840 REM   CLOSE SEQ FILE
850 CLOSE 1 : GOSUB 1000 : CLOSE 15
```

```
860 PRINT : PRINT "TRANSACTIONS POSTED AND FILE
    S CLOSED."
870 END
880 :
890 :
1000 REM   DISK ERROR ROUTINE
1010 INPUT#15, DX, DX$, DY, DZ : REM READ
1020 IF DX < 20 OR DX = 50 THEN RETURN
1030 PRINT : PRINT "[RVS]DISK ERROR: " DX$
1040 CLOSE 15 : END
```

(a) Only one small component of this application has been completed. List the other programs you would need to make a complete personal finance management system.

(a) Programs:
1. Edit MONTH# file for entry errors
2. Print BUDGET# file accounts
3. "Exception report" showing over budget accounts or projected over budget accounts

We have found relative files much easier to use than sequential files. But let's not forget that sequential files have their place in computing. With the knowledge gained from this book, you should now be able to read the reference manual for your computer with new understanding. You should also be able to write your own data file programs and read programs written by others.

Chapter 8 Self-Test

1. (a) The first application in this chapter was an inventory control system. Before you continue you may want to review the system description so you are familiar with the contents of each file and how they interact.

To this system is added a third file; a sequential transaction file in which is placed the data regarding each transaction that affects the inventory. Two types of transactions will affect inventory:

Type 1—units are added to inventory.
Type 2—units are taken from inventory.

Data are recorded in the sequential transaction file in this format:

```
T = TRANSACTION TYPE (1 OR 2)
Y$ = DATE (6)
N$ = INVOICE # (5)
P2$ = PROD # (4)
Q1 = QUANTITY ADDED OR DEDUCTED
```

Write a program to create the sequential data file named TRANSACT with this dataset. Make certain the product numbers entered in the file correspond to product numbers that exist in BUSINESS INVEN (you may want to run the program that displays BUSINESS INVEN and jot them down). Include both types of transaction types in your file data. The entry of only one dataset is shown in the sample run.

```
RUN

TRANSACTION CODES:
'1' FOR UNITS ADDED TO INVENTORY
'2' FOR UNITS TAKEN FROM INVENTORY
TRANSACTION TYPE ? 1
TRANSACTION DATE ? 01-01-84
INVOICE/RECEIPT # ? 3019
PROD. # (4 CHRS.) ? 0001
QUANTITY ? 3

MORE TRANSACTIONS ('Y' OR 'N') ? N

READY.
```

Complete the rest of the program following the introductory module.

```
100 REM   PROB. 8-1  CREATE 'TRANSACT' SEQ FILE

110 REM    INVENTORY CHANGES FOR
120 REM    'BUSINESS INVEN' REL FILE
130 :
140 REM   VARIABLES USED
150 REM     T = TRANSACTION TYPE (1,2)
160 REM     Y$ = DATE (XX-XX-XX)
170 REM     I$ = INVENTORY/RECEIPT  #
180 REM     N$ = PROD. # (4)
190 REM     Q1 = QUAN. ADDED OR SUBTRACTED FROM
      INVENTORY (3)
200 REM     R$ = USER RESPONSE
210 :
220 REM   FILES USED
230 REM     SEQ FILE: TRANSACT
240 REM     DATASET: T, Y$, I$, N$, Q1
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
280 OPEN 1, 8, 8, "TRANSACT,S,W"
290 GOSUB 800 : IF DX <> 63 THEN 310
300 CLOSE 1 : PRINT#15, "S0:TRANSACT" : GOSUB 8
     00 : GOTO 280
310 :
320 REM   DATA ENTRY
```

1. (b) Write a companion program to display the results of the program you wrote for Problem 1(a) (read and display TRANSACT sequential file). Use the same introductory module you used for the program which created the file.

   Here's a sample run:

```
RUN

READ 'TRANSACT' FILE

DISPLAY ON 'S'CREEN OR 'P'RINTER ? S

TRANS. TYPE: 1      DATE: 01-01-84
INV/REC #: 3019     PROD. #: 0001
QUANTITY:  3

PRESS 'RETURN' TO CONTINUE.
FILE READ AND DISPLAYED.
READY.
```

2. Write a program to process the data in TRANSACT to BUSINESS INVEN (update the 'quantity in stock' information). For each dataset input from TRANSACT, use the sequential file INDEX to locate the record in BUSINESS INVEN to be updated or posted with the data from TRANSACT.

```
100 REM    PROB.8-2  POST 'TRANSACT' SEQ FILE
110 REM     INVENTORY CHANGES TO
120 REM     'BUSINESS INVEN' REL FILE
130 :
140 REM    VARIABLES USED
150 REM      T = TRANSACTION TYPE (1,2)
160 REM      Y$ = DATE (XX-XX-XX)
170 REM      I$ = INVENTORY/RECEIPT  #
180 REM      N$,N1$,N2$= PROD. # (4)
190 REM      Q1 = QUAN. ADDED OR SUBTRACTED FROM
      STOCK (3)
200 REM      R$ = USER RESPONSE
210 REM      P$ = PROD. DESCR. (20)
220 REM      S$ = SUPPLIER (20)
230 REM      R  = REORDER PT. (3)
240 REM      Y  = REORDER QTY. (3)
250 REM      Q  = QUAN. IN STOCK (3)
260 REM      C  = COST (6)
270 REM      R1 = RECORD NUMBER
280 REM      U  = UNIT PRICE (6)
290 REM      H  = HI BYTE
300 REM      L  = LO BYTE
310 :        CR$= CHR$(13)
```

```
320 :
330 REM   FILES USED
340 REM     SEQ FILE: TRANSACT
350 REM     DATASET: T, Y$, I$, N$, Q1
360 REM     SEQ FILE: INDEX
370 REM     DATASET: N$, R1
380 REM     REL FILE: BUSINESS INVEN
390 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
400 REM     RECORD LENGTH: 75 BYTES
410 :
420 REM   INITIALIZE
430 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
     POST [RVS]TRANSACT[OFF] TO [RVS]BUSINESS INV
     EN[OFF]."
440 PRINT : PRINT "WORKING..."
450 OPEN 15, 8, 15
460 OPEN 1, 8, 8, "TRANSACT,S,R"
470 GOSUB 1000
```

3. Write a program that, after all the transactions have been processed, will search the entire BUSINESS INVEN file and print a report of products that have fallen below the reorder point and need reordering. Allow the user to select a report printed to a printer or displayed on the screen. (*Note:* This could be a routine added to the end of the program written in Problem 3.)

```
RUN

'BUSINESS INVEN' REORDER REPORT.

REPORT TO 'S'CREEN OR 'P'RINTER ? [S]

PROD. #: 0003

WILEY FILES CPT 7&8

SUPPLIER : FINKEL PLASTICS
REORDER PT. : 5
REORDER QUAN.: 25
QUAN. IN STOCK: 3
COST: $ 25.37
UNIT PRICE: $ 73.99
REORDER COST: $634.25

PRESS 'RETURN' TO CONTINUE.

REORDER REPORT COMPLETED; FILES CLOSED.
READY.
```

Here's the introductory module.

```
100 REM   PROB.8-3  REORDER REPORT FROM
110 REM    'BUSINESS INVEN' REL FILE
120 :
130 REM   VARIABLES USED
140 REM     N$ = PROD. # (4)
150 REM     R$ = USER RESPONSE
160 REM     P$ = PROD. DESCR. (20)
170 REM     S$ = SUPPLIER (20)
180 REM     R  = REORDER PT. (3)
190 REM     Y  = REORDER QTY. (3)
200 REM     Q  = QUAN. IN STOCK (3)
210 REM     C  = COST (6)
220 REM     R1 = RECORD NUMBER
230 REM     U  = UNIT PRICE (6)
240 REM     H  = HI BYTE
250 REM     L  = LO BYTE
260 REM     X  = FOR...NEXT VARIABLE
270 :
280 REM   FILES USED
290 REM     REL FILE: BUSINESS INVEN
300 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
310 REM     RECORD LENGTH: 75 BYTES
320 :
330 REM   INITIALIZE
```

Answer Key

1. (a)
```
100 REM   PROB. 8-1  CREATE 'TRANSACT' SEQ FILE

110 REM    INVENTORY CHANGES FOR
120 REM    'BUSINESS INVEN' REL FILE
130 :
140 REM   VARIABLES USED
150 REM     T = TRANSACTION TYPE (1,2)
160 REM     Y$ = DATE (XX-XX-XX)
170 REM     I$ = INVENTORY/RECEIPT  #
180 REM     N$ = PROD. # (4)
190 REM     Q1 = QUAN. ADDED OR SUBTRACTED FROM
      INVENTORY (3)
200 REM     R$ = USER RESPONSE
210 :
220 REM   FILES USED
230 REM     SEQ FILE: TRANSACT
240 REM     DATASET: T, Y$, I$, N$, Q1
250 :
260 REM   INITIALIZE
270 OPEN 15, 8, 15
```

```
280 OPEN 1, 8, 8, "TRANSACT,S,W"
290 GOSUB 800 : IF DX <> 63 THEN 310
300 CLOSE 1 : PRINT#15, "S0:TRANSACT" : GOSUB 8
    00 : GOTO 280
310 :
320 REM   DATA ENTRY
330 PRINT "[CLR][DOWN][DOWN][DOWN]TRANSACTION C
    ODES:"
340 PRINT "'1' FOR UNITS ADDED TO INVENTORY"
350 PRINT "'2' FOR UNITS TAKEN FROM INVENTORY"
360 PRINT : INPUT "TRANSACTION TYPE "; T
370 IF T < 1 OR T > 2 THEN PRINT "[RVS] 1 OR 2
    ONLY![UP][UP][UP]" : GOTO 360
380 PRINT : INPUT "TRANSACTION DATE "; Y$
390 REM   DATA ENTRY TESTS
400 PRINT : INPUT  "INVOICE/RECEIPT # "; I$
410 REM   DATA ENTRY TESTS
420 PRINT : INPUT "PROD. # (4 CHRS.) "; N$
430 REM   DATA ENTRY TESTS
440 PRINT : INPUT "QUANTITY "; Q1
450 REM   DATA ENTRY TESTS
460 :
470 REM   PRINT TO FILE
480 PRINT#1, T : PRINT#1, Y$ : PRINT#1, I$ : PR
    INT#1, N$ : PRINT#1, Q1
490 :
500 REM   MORE
510 PRINT : PRINT "MORE TRANSACTIONS ('Y' OR 'N
    ') ?"
520 GET R$ : IF R$ <> "Y" AND R$ <> "N" THEN 52
    0
530 IF R$ = "Y" THEN 330
540 :
550 REM   CLOSE FILES
560 CLOSE 1 : GOSUB 800 : CLOSE 15
570 END
580 :
590 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

1. (b)
```
100 REM   PROB.8-1B READ 'TRANSACT' SEQ FILE
110 REM    INVENTORY CHANGES FOR
120 REM    'BUSINESS INVEN' REL FILE
130 :
140 REM   VARIABLES USED
150 REM     T = TRANSACTION TYPE (1,2)
160 REM     Y$ = DATE (XX-XX-XX)
170 REM     I$ = INVENTORY/RECEIPT #
180 REM     N$ = PROD. # (4)
190 REM     Q1 = QUAN. ADDED OR SUBTRACTED FROM
      INVENTORY (3)
200 REM     R$ = USER RESPONSE
210 REM     DN = DEVICE NUMBER
220 :
230 REM   FILES USED
240 REM     SEQ FILE: TRANSACT
250 REM     DATASET: T, Y$, I$, N$, Q1
260 :
270 REM   INITIALIZE
280 OPEN 15, 8, 15
290 OPEN 1, 8, 8, "TRANSACT,S,R"
300 GOSUB 800
310 :
320 REM   READ &  DISPLAY
330 PRINT "[CLR][DOWN][DOWN][DOWN]   READ 'TRAN
     SACT' FILE"
340 PRINT : PRINT "DISPLAY ON  [RVS]S[OFF]CREEN
      OR [RVS]P[OFF]RINTER ?"
350 GET R$ : IF R$ <> "S" AND R$ <> "P" THEN 35
     0
360 LET DN = 3 : IF R$ = "P" THEN DN = 4
370 OPEN 4, DN
380 INPUT#1, T, Y$, I$, N$, Q1
390 LET SV = ST
400 PRINT#4, "" : PRINT#4, "TRANS. TYPE: " T "
      DATE: " Y$
410 PRINT#4, "INV/REC #: " I$ "   ACCOUNT #: "
     N$
420 PRINT#4, "QUANTITY: " Q1
430 IF DN = 4 THEN 470
440 PRINT : PRINT "PRESS 'RETURN' TO CONTINUE."

450 GET R$ : IF R$ <> CHR$(13) THEN 450
460 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]"
      : REM CLR & CRSR DOWN 5
470 IF SV = 0 THEN 380
480 :
490 REM   CLOSE FILES
500 PRINT#4 : CLOSE 4 : CLOSE 1 : GOSUB 800 : C
     LOSE 15
510 PRINT "FILE READ AND DISPLAYED."
520 END
```

```
530 :
540 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT : PRINT "[RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

2.

```
100 REM   PROB.8-2  POST 'TRANSACT' SEQ FILE
110 REM    INVENTORY CHANGES TO
120 REM    'BUSINESS INVEN' REL FILE
130 :
140 REM   VARIABLES USED
150 REM     T = TRANSACTION TYPE (1,2)
160 REM     Y$ = DATE (XX-XX-XX)
170 REM     I$ = INVENTORY/RECEIPT  #
180 REM     N$,N1$,N2$= PROD. # (4)
190 REM     Q1 = QUAN. ADDED OR SUBTRACTED FROM
      STOCK (3)
200 REM     R$ = USER RESPONSE
210 REM     P$ = PROD. DESCR. (20)
220 REM     S$ = SUPPLIER (20)
230 REM     R  = REORDER PT. (3)
240 REM     Y  = REORDER QTY. (3)
250 REM     Q  = QUAN. IN STOCK (3)
260 REM     C  = COST (6)
270 REM     R1 = RECORD NUMBER
280 REM     U  = UNIT PRICE (6)
290 REM     H  = HI BYTE
300 REM     L  = LO BYTE
310 :       CR$= CHR$(13)
320 :
330 REM   FILES USED
340 REM     SEQ FILE: TRANSACT
350 REM     DATASET: T, Y$, I$, N$, Q1
360 REM     SEQ FILE: INDEX
370 REM     DATASET: N$, R1
380 REM     REL FILE: BUSINESS INVEN
390 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
400 REM     RECORD LENGTH: 75 BYTES
410 :
420 REM   INITIALIZE
430 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
     POST [RVS]TRANSACT[OFF] TO [RVS]BUSINESS INV
     EN[OFF]."
440 PRINT : PRINT "WORKING..."
450 OPEN 15, 8, 15
460 OPEN 1, 8, 8, "TRANSACT,S,R"
470 GOSUB 1000
480 OPEN 2, 8, 2, "BUSINESS INVEN"
490 GOSUB 1000
```

```
500 :
510 REM   READ 'TRANSACT' & FIND RECORD # FROM
     'INDEX'
520 INPUT#1, T, Y$, I$, N1$, Q1
530 PRINT "UPDATING PROD. " N1$
540 LET SV = ST
550 OPEN 3, 8, 3, "INDEX,S,R" : GOSUB 1000
560 INPUT#3, N$, R1
570 LET S2 = ST
580 IF N$ = N1$ THEN CLOSE 3 : GOTO 650
590 IF S2 = 0 THEN 560
600 :
610 REM   RECORD NOT FOUND
620 PRINT : PRINT "[RVS] PROD. # NOT FOUND IN '
     INDEX.'" : CLOSE 3 : GOTO 920
630 :
640 REM   FIND & CHANGE Q IN REL FILE
650 LET H = INT(R1/256) : LET L = R1 - H*256
660 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
670 GOSUB 1000 : IF DX = 50 THEN 910: REM NO RE
     CORD #
680 INPUT#2, N2$, P$, S$, R, Y, Q, C, U
690 IF T = 1 THEN LET Q = Q + Q1 : GOTO 730
700 IF T = 2 THEN LET Q = Q - Q1 : GOTO 730
710 :
720 REM   PRINT UPDATE TO REL FILE
730 PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR$(
     1)
740 GOSUB 1000
750 PRINT#2, N2$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR$
     ; Y ;CR$; Q ;CR$; C ;CR$; U
760 IF SV = 0 THEN 520 : REM NEXT TRANSACT
770 :
780 REM   CLOSE FILES
790 PRINT : PRINT "TRANSACTIONS POSTED, FILES C
     LOSED."
800 CLOSE 1 : GOSUB 1000 : CLOSE 2 : GOSUB 1000
      : CLOSE 15
810 END
820 :
830 :
900 REM   ERROR IN 'BUSINESS INVEN'
910 PRINT : PRINT "[RVS] RECORD NUMBER NOT FOUN
     D IN 'BUSINESS INVEN'."
920 PRINT "TRANSACTION DATED " Y$
930 PRINT "INVOICE/RECEIPT # " I$
940 PRINT "FOR PRODUCT # " N1$
950 PRINT "NOT PROCESSED ! "
960 GOTO 760 : REM NEXT TRANSACT
970 :
```

```
980 :
1000 REM   DISK ERROR ROUTINE
1010 INPUT#15, DX, DX$, DY, DZ : REM READ
1020 IF DX < 20 OR DX = 50 THEN RETURN
1030 PRINT : PRINT "[RVS]DISK ERROR: " DX$
1040 CLOSE 15 : END
```

3.

```
100 REM   PROB.8-3  REORDER REPORT FROM
110 REM    'BUSINESS INVEN' REL FILE
120 :
130 REM   VARIABLES USED
140 REM     N$ = PROD. # (4)
150 REM     R$ = USER RESPONSE
160 REM     P$ = PROD. DESCR. (20)
170 REM     S$ = SUPPLIER (20)
180 REM     R  = REORDER PT. (3)
190 REM     Y  = REORDER QTY. (3)
200 REM     Q  = QUAN. IN STOCK (3)
210 REM     C  = COST (6)
220 REM     R1 = RECORD NUMBER
230 REM     U  = UNIT PRICE (6)
240 REM     H  = HI BYTE
250 REM     L  = LO BYTE
260 REM     X  = FOR...NEXT VARIABLE
270 :
280 REM   FILES USED
290 REM     REL FILE: BUSINESS INVEN
300 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
310 REM     RECORD LENGTH: 75 BYTES
320 :
330 REM   INITIALIZE
340 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
     [RVS]BUSINESS INVEN[OFF] REORDER REPORT."
350 PRINT : PRINT "REPORT TO [RVS]S[OFF]CREEN O
     R [RVS]P[OFF]RINTER ?"
360 GET R$ : IF R$ <> "S" AND R$ <> "P" THEN 36
     0
370 LET DN = 3 : IF R$ = "P" THEN LET DN = 4
380 OPEN 4, DN
390 PRINT : PRINT "WORKING..."
400 OPEN 15, 8, 15
410 OPEN 2, 8, 2, "BUSINESS INVEN"
420 GOSUB 900
430 :
440 REM   READ DATASET, CHECK
450 REM   INVENTORY AGAINST REORDER
460 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
     1)
470 INPUT#2, R1
480 :
490 FOR X = 2 TO R1
```

```
500 : LET H = INT(X/256) : LET L = X - H*256
510 : PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR
     $(1)
520 : INPUT#2, N$, P$, S$, R, Y, Q, C, U
530 : IF Q < R THEN GOSUB 630 : REM REPORT
540 NEXT X
550 :
560 REM   CLOSE FILE
570 PRINT#4 : CLOSE 4
580 CLOSE 2 : GOSUB 900 : CLOSE 15
590 PRINT : PRINT "REORDER REPORT COMPLETED; FI
     LES CLOSED."
600 END
610 :
620 :
630 REM   REPORT SUBROUTINE
640 REM   DN SET FOR PRINTER OR SCREEN
650 PRINT#4, "" : PRINT#4, "PROD. #: " N$
660 PRINT#4, "SUPPLIER: " S$
670 PRINT#4, "REORDER PT.: " R
680 PRINT#4, "REORDER QUAN.: " Y
690 PRINT#4, "QUAN. IN STOCK: " Q
700 PRINT#4, "COST: $" C
710 PRINT#4, "UNIT PRICE: $" U
720 PRINT#4, "REORDER COST: $" C*R
730 IF DN = 4 THEN RETURN
740 PRINT : PRINT "PRESS [RVS]RETURN[OFF] TO CO
     NTINUE."
750 GET R$ : IF R$ <> CHR$(13) THEN 750
760 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]"
      : REM CLR & CRSR DOWN 5
770 RETURN
780 :
790 :
900 REM   DISK ERROR ROUTINE
910 INPUT#15, DX, DX$, DY, DZ : REM READ
920 IF DX < 20 OR DX = 50 THEN RETURN
930 PRINT : PRINT "[RVS]DISK ERROR: " DX$
940 CLOSE 15 : END
```

Final Self-Test

1. Write a program to create a relative file of names, addresses, and phone numbers. We call this file BLACKBUK. The data are entered and assigned to string variables as indicated in the introductory module shown below, and stored as strings in the file. The sample RUN shows the entries for the first record. You complete the program in the space below the introductory module.

```
RUN

Create BLACKBUK rel file.

Last name ? Brown
First name ? Jerald
Address ? 13140 Bambi Lane
City ? Sebastopol
State ? Cal
Only 2 letters.
State ? CA
Zip ? 75472
Area Code ? 800
Enter phone as 999-9999.
Phone ? 555-1234
Only first 30 characters of notes will be filed.
Notes ? Independent Filmmaker

More data (Press 'Y' or 'N') ? [N]

File closed.
ready.
```

```
100 rem  prob 1 final self test
110 rem   personal phone directory
120 :
130 rem   variables used
140 rem     l$ = last name (15)
150 rem     f$ = first name (15)
160 rem     a$ = address (25)
170 rem     c$ = city (20)
180 rem     s$ = state (2)
190 rem     z$ = zip code (5)
200 rem     pc$ = area code (3)
210 rem     p$ = phone (8)
220 rem     n$ = notes (30)
230 rem     r1 = record number
240 rem     h  = hi byte
250 rem     l  = lo byte
260 :       cr$ = chr$(13)
270 :
280 rem   files used
290 rem     rel: blackbuk
300 rem     record length: 132
310 :
320 rem   initialize
330 print "[clr][down][down][down]  Create  'BL
     ACKBUK' rel file."
340 open 15, 8, 15
350 open 2, 8, 2, "blackbuk,l," + chr$(132)
360 gosub 1000 : if dx = 63 then close 2 : prin
     t#15, "s0:blackbuk" : goto 350
370 :
400 rem   data entry
```

2. Write a program to display all the records in BLACKBUK, with the data items displayed as shown in the sample RUN.

```
run

Read BLACKBUK rel file.

Jerald  Brown
13140 Bambi Lane
Sebastopol   CA 95472
(800) 555-1234
Independent Filmmaker

Press 'RETURN' for next dataset.

File read and closed.
ready.
```

```
100 rem  prob 2 final self test
110 rem   display personal phone directory
120 :
130 rem   variables used
140 rem     l$ = last name (15)
150 rem     f$ = first name (15)
160 rem     a$ = address (25)
170 rem     c$ = city (20)
180 rem     s$ = state (2)
190 rem     z$ = zip code (5)
200 rem     pc$ = area code (3)
210 rem     p$ = phone (8)
220 rem     n$ = notes (30)
230 rem     r1 = record number
240 rem     h  = hi byte
250 rem     l  = lo byte
260 :       cr$ = chr$(13)
270 :
280 rem   files used
290 rem     rel: blackbuk
300 rem     record length: 132
310 :
320 rem   initialize
330 print "[clr][down][down][down]  Read 'BLACK
    BUK' rel file."
340 open 15, 8, 15
350 open 2, 8, 2, "blackbuk"
360 gosub 800
370 :
400 rem   read & display file
```

3. Write a program that will select and display all names, addresses, and numbers in a user-selected area code from BLACKBUK. The program you wrote for Final Self-Test Problem 2 is easily modified for this purpose.

```
run

Read BLACKBUK rel file for area code.

Area code ? (800)
Exactly 3 digits--no parentheses.
Area code ? 800
Jerald  Brown
13140 Bambi Lane
Sebastopol   CA 95472
(800) 555-1234
Independent Filmmaker

Press 'RETURN' for next dataset.

File read and closed.
ready.
```

4. Write a program to change each dataset in BUSINESS INVEN by increasing the unit sales price of each item by 10 percent. The program should display the product number, the old price, and the new price.

```
RUN

PROD. #  OLD PRICE  NEW PRICE
0001     $14.23     $15.65
0002     $15.12     $16.63
0003     $3.75      $4.12
0004     $23.47     $25.82
0005     $3.38      $3.72

PRICE UPDATE COMPLETED; FILE CLOSED.

READY.
```

```
100 REM   PROB. 4 FINAL SELF-TEST
110 REM    'BUSINESS INVEN' REL FILE
120 REM    CHANGE PRODUCT PRICE
130 :
140 REM   VARIABLES USED
150 REM     N$ = PROD. # (4)
160 REM     R$ = USER RESPONSE
170 REM     P$ = PROD. DESCR. (20)
180 REM     S$ = SUPPLIER (20)
190 REM     R  = REORDER PT. (3)
200 REM     Y  = REORDER QTY. (3)
210 REM     Q  = QUAN. IN STOCK (3)
220 REM     C  = COST (6)
230 REM     U,U1 = UNIT PRICE (6)
240 REM     R1 = RECORD NUMBER
250 REM     H  = HI BYTE
260 REM     L  = LO BYTE
270 REM     X  = FOR...NEXT VARIABLE
280 :       CR$= CHR$(13)
290 :
300 REM   FILES USED
310 REM     REL FILE: BUSINESS INVEN
320 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
330 REM     RECORD LENGTH: 75 BYTES
340 :
350 REM   INITIALIZE
360 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
     [RVS]BUSINESS INVEN[OFF] PRICE UPDATE."
370 PRINT : PRINT "WORKING..."
```

Answer Key

1.

```
100 rem  prob 1 final self test
110 rem   personal phone directory
120 :
130 rem   variables used
140 rem     l$ = last name (15)
150 rem     f$ = first name (15)
160 rem     a$ = address (25)
170 rem     c$ = city (20)
180 rem     s$ = state (2)
190 rem     z$ = zip code (5)
200 rem     pc$ = area code (3)
210 rem     p$ = phone (8)
220 rem     n$ = notes (30)
230 rem     r1 = record number
240 rem     h  = hi byte
250 rem     l  = lo byte
260 :       cr$ = chr$(13)
270 :
280 rem   files used
290 rem     rel: blackbuk
300 rem     record length: 132
310 :
320 rem   initialize
330 print "[clr][down][down][down]  Create  'BL
     ACKBUK' rel file."
340 open 15, 8, 15
350 open 2, 8, 2, "blackbuk,l," + chr$(132)
360 gosub 1000 : if dx = 63 then close 2 : prin
     t#15, "s0:blackbuk" : goto 350
370 :
400 rem   data entry
```

```
410 let r1 = 2
415 print "[clr][down][down]"
420 print : input "Last name[shsp][shsp][shsp][
    shsp][left][left][left]"; l$
430 if l$ = "[shsp]" or len(l$) = 0 then print
    "[up][up][up]" : goto 420
440 if len(l$) > 15 then print "Only 15 letters
    ." : goto 420
450 :
460 print : input "First name[shsp][shsp][shsp]
    [shsp][left][left][left]"; f$
470 if f$ = "[shsp]" or len(f$) = 0 then print
    "[up][up][up]" : goto 460
480 if len(f$) > 15 then print "Only 15 letters
    ." : goto 460
490 :
500 print : input "Address[shsp][shsp][shsp][sh
    sp][left][left][left]"; a$
510 if a$ = "[shsp]" or len(a$) = 0 then print
    "[up][up][up]" : goto 500
520 if len(a$) > 25 then print "Only 25 letters
    ." : goto 500
530 :
540 print : input "City[shsp][shsp][shsp][shsp]
    [left][left][left]"; c$
550 if len(c$) > 20 then print "Only 20 letters
    ." : goto 540
560 if c$ = "[shsp]" or len(c$) = 0 then print
    "[up][up][up]" : goto 540
570 :
580 print : input "State[shsp][shsp][shsp][shsp
    ][left][left][left]"; s$
590 if s$ = "[shsp]" or len(s$) = 0 then print
    "[up][up][up]" : goto 580
600 if len(s$) > 2 then print "Only 2 letters."
     : goto 580
610 :
620 print : input "Zip[shsp][shsp][shsp][shsp][
    left][left][left]"; z$
630 if z$ = "[shsp]" or len(z$) = 0 then print
    "[up][up][up]" : goto 620
640 if len(z$) <> 5 then print "Exactly 5 digit
    s." : goto 620
650 :
660 print : input "Area Code[shsp][shsp][shsp][
    shsp][left][left][left]"; pc$
670 if pc$ = "[shsp]" or len(pc$) = 0 then prin
    t "[up][up][up]" : goto 660
680 if len(pc$) <> 3 then print "Exactly 3 digi
    ts." : goto 660
690 :
700 print : print "Enter phone as 999-9999."
```

```
710 print : input "Phone[shsp][shsp][shsp][shsp
    ][left][left][left]"; p$
720 if p$ = "[shsp]" or len(p$) = 0 then print
    "[up][up][up]" : goto 710
730 if len(p$) <> 8 then print "Exactly 8 digit
    s as XXX-XXXX." : goto 710
740 :
750 print : print "Only first 30 characters of
    notes will  be filed."
760 print : input "Notes[shsp][shsp][shsp][shsp
    ][left][left][left]"; n$
770 if n$ = "[shsp]" or len(n$) = 0 then print
    "[up][up][up]" : goto 760
780 let n$ = left$(n$,30)
790 :
800 rem   print to file
810 let h = int(r1/256) : let l = r1 - h*256
820 gosub 1000
830 print#15, "p" chr$(2) chr$(l) chr$(h) chr$(
    1)
840 print#2,f$;cr$;l$;cr$;a$;cr$;c$;cr$;s$;cr$;
    z$;cr$;pc$;cr$;p$;cr$;n$
850 :
860 rem   more ?
870 print : print "More data (Press 'Y' or 'N')
    ?"
880 get r$ : if r$ <> "y" and r$ <> "n" then 88
    0
890 if r$ = "y" then let r1 = r1 + 1 : goto 415

900 :
910 rem   close
920 print#15, "p" chr$(2) chr$(1) chr$(0) chr$(
    1)
930 print#2, r1
940 close 2 : gosub 1000 : close 15
950 print : print "File closed."
960 end
970 :
1000 rem   disk error routine
1010 input#15, dx, dx$, dy, dz : rem read
1020 if dx < 20 or dx = 50 then return
1030 print : print "[rvs]Disk error: " dx$
1040 close 15 : end
```

Carefully look at what we have done in lines 350–360. Line 350 OPENs the relative file with the L (length) specifier, which will create a new relative file if none exists. Line 360 checks the disk error subroutine. If error 63, "FILE EXISTS", is indicated, then the file is SCRATCHed (erased from the disk), and the file is re-created by line 350. Why all this? Because any other way of writing over old file data may leave the old data there to be read.

2.

```
100 rem  prob 2 final self test
110 rem   display personal phone directory
120 :
130 rem   variables used
140 rem     l$ = last name (15)
150 rem     f$ = first name (15)
160 rem     a$ = address (25)
170 rem     c$ = city (20)
180 rem     s$ = state (2)
190 rem     z$ = zip code (5)
200 rem     pc$ = area code (3)
210 rem     p$ = phone (8)
220 rem     n$ = notes (30)
230 rem     r1 = record number
240 rem     h  = hi byte
250 rem     l  = lo byte
260 :       cr$ = chr$(13)
270 :
280 rem   files used
290 rem     rel: blackbuk
300 rem     record length: 132
310 :
320 rem   initialize
330 print "[clr][down][down][down]  Read 'BLACK
     BUK' rel file."
340 open 15, 8, 15
350 open 2, 8, 2, "blackbuk"
360 gosub 800
370 :
400 rem   read & display file
410 print#15, "p" chr$(2) chr$(1) chr$(0) chr$(
     1)
420 input#2, r1
430 :
440 for x = 2 to r1
450 let h = int(x/256) : let l = x - h*256
460 print#15, "p" chr$(2) chr$(l) chr$(h) chr$(
     1)
470 input#2, f$, l$, a$, c$, s$, z$, pc$, p$, n
     $
480 :
490 rem   display
500 print "[clr][down][down][down]" f$ "  " l$
510 print a$
520 print c$ tab(22) s$ tab(25) z$
530 print "(" pc$ ")" tab(6) p$
540 print n$
550 print : print "Press [rvs]RETURN[off] for n
     ext dataset."
560 get r$ : if r$ <> chr$(13) then 560
```

```
570 next x
580 :
590 :
600 rem   close files
610 close 2 : gosub 800 : close 15
620 print : print "File read and closed."
630 end
640 :
800 rem   disk error routine
810 input#15, dx, dx$, dy, dz : rem read
820 if dx < 20 or dx = 50 then return
830 print "[down][rvs]Disk error: " dx$
840 close 15 : end
```

3.

```
100 rem  prob 3 final self test
110 rem   display by area code
120 :
130 rem   variables used
140 rem     l$ = last name (15)
150 rem     f$ = first name (15)
160 rem     a$ = address (25)
170 rem     c$ = city (20)
180 rem     s$ = state (2)
190 rem     z$ = zip code (5)
200 rem     pc$ = area code (3)
210 rem     p$ = phone (8)
220 rem     n$ = notes (30)
230 rem     r1 = record number
240 rem     h  = hi byte
250 rem     l  = lo byte
260 :       cr$ = chr$(13)
270 :
280 rem   files used
290 rem     rel: blackbuk
300 rem     record length: 132
310 :
320 rem   initialize
330 print "[clr][down][down][down]Read [rvs]BLA
     CKBUK[off] rel file for area code."
340 open 15, 8, 15
350 open 2, 8, 2, "blackbuk"
360 gosub 800
370 :
380 rem   select area code
390 print : input "Area code[shsp][shsp][shsp][
     shsp][left][left][left]"; r$
400 if r$ = "[shsp]" or len (r$) = 0 then print
      "[up][up][up]" : goto 390
410 if len(r$) <> 3 then print "Exactly 3 digit
     s--no parentheses." : goto 390
420 :
430 rem   read
```

```
440 print#15, "p" chr$(2) chr$(1) chr$(0) chr$(
    1)
450 input#2, r1
460 :
470 for x = 2 to r1
480 let h = int(x/256) : let l = x - h*256
490 print#15, "p" chr$(2) chr$(l) chr$(h) chr$(
    1)
500 gosub 800 : if dx = 50 then 670: rem close
510 input#2, f$, l$, a$, c$, s$, z$, pc$, p$, n
    $
520 :
530 rem   test
540 if pc$ <> r$ then 630: rem next
550 rem   display
560 print "[clr][down][down][down]" f$ "  " l$
570 print a$
580 print c$ tab(22) s$ tab(25) z$
590 print "(" pc$ ")" tab(6) p$
600 print n$
610 print : print "Press [rvs]RETURN[off] for n
    ext dataset."
620 get z$ : if z$ <> chr$(13) then 620
630 next x
640 :
650 :
660 rem   close files
670 close 2 : gosub 800 : close 15
680 print : print "File read and closed."
690 end
700 :
800 rem   disk error routine
810 input#15, dx, dx$, dy, dz : rem read
820 if dx < 20 or dx = 50 then return
830 print "[down][rvs]Disk error: " dx$
840 close 15 : end
```

4.

```
100 REM   PROB. 4 FINAL SELF-TEST
110 REM    'BUSINESS INVEN' REL FILE
120 REM    CHANGE PRODUCT PRICE
130 :
140 REM   VARIABLES USED
150 REM     N$ = PROD. # (4)
160 REM     R$ = USER RESPONSE
170 REM     P$ = PROD. DESCR. (20)
180 REM     S$ = SUPPLIER (20)
190 REM     R  = REORDER PT. (3)
200 REM     Y  = REORDER QTY. (3)
210 REM     Q  = QUAN. IN STOCK (3)
220 REM     C  = COST (6)
230 REM     U,U1 = UNIT PRICE (6)
240 REM     R1 = RECORD NUMBER
```

```
250 REM      H  = HI BYTE
260 REM      L  = LO BYTE
270 REM      X  = FOR...NEXT VARIABLE
280 :        CR$= CHR$(13)
290 :
300 REM   FILES USED
310 REM      REL FILE: BUSINESS INVEN
320 REM      DATASET: N$,P$,S$,R,Y,Q,C,U
330 REM      RECORD LENGTH: 75 BYTES
340 :
350 REM   INITIALIZE
360 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
    [RVS]BUSINESS INVEN[OFF] PRICE UPDATE."
370 PRINT : PRINT "WORKING..."
380 OPEN 15, 8, 15
390 OPEN 2, 8, 2, "BUSINESS INVEN"
400 GOSUB 800
410 :
420 REM   READ DATASET, INCREASE
430 REM   PRICE
440 PRINT "[CLR][DOWN][DOWN][DOWN]PROD. #", "OL
    D PRICE", "NEW PRICE" : PRINT
450 PRINT#15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(
    1)
460 INPUT#2, R1
470 :
480 FOR X = 2 TO R1
490 : LET H = INT(X/256) : LET L = X - H*256
500 : PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR
    $(1)
510 : INPUT#2, N$, P$, S$, R, Y, Q, C, U
520 : U1 = 1.1 * U
530 : U1 = INT(.5 + U1 * 100)/100 : REM ROUND D
    ECIMAL
540 : PRINT N$, U, U1
550 : PRINT#15, "P" CHR$(2) CHR$(L) CHR$(H) CHR
    $(1)
560 : PRINT#2, N$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR
    $; Y ;CR$; Q ;CR$; C ;CR$; U1
570 NEXT X
580 REM   CLOSE
590 CLOSE 2 : GOSUB 800 : CLOSE 15
600 PRINT : PRINT "PRICE UPDATE COMPLETED; FILE
     CLOSED."
610 END
620 :
630 :
800 REM   DISK ERROR ROUTINE
810 INPUT#15, DX, DX$, DY, DZ : REM READ
820 IF DX < 20 OR DX = 50 THEN RETURN
830 PRINT "[DOWN][RVS]DISK ERROR: " DX$
840 CLOSE 15 : END
```

APPENDIX ONE

Commodore ASCII Codes

Chart of Commodore 64 and PET ASCII characters for upper/lowercase modes.
* indicates C-64 only.

CHR$( )	Upper/graphics	Lower/upper
0		
1		
2		
3	(STOP)	
4		
5	WHITE *	
6		
7		
8	(disables SHIFT-C<) *	
9	(enables SHIFT-C<) *	
10	LINEFEED	
11		
12		
13	RETURN	
14	(lowercase) *	
15		
16	(TAB or ;)	
17	CRSR DOWN	
18	RVS ON	
19	HOME	
20	DEL	
21		
22		
23		
24		
25		
26		
27		
28	RED *	
29	CRSR RIGHT	
30	GRN *	
31	BLUE *	
32	SPACE	

CHR$()	Upper/graphics	Lower/upper
33	!	!
34	"	"
35	#	#
36	$	$
37	%	%
38	&	&
39	'	'
40	(	(
41	)	)
42	*	*
43	+	+
44	,	,
45	-	-
46	.	.
47	/	/
48	0	0
49	1	1
50	2	2
51	3	3
52	4	4
53	5	5
54	6	6
55	7	7
56	8	8
57	9	9
58	:	:
59	;	;
60	<	<
61	=	=
62	>	>
63	?	?
64	@	@
65	A	a
66	B	b
67	C	c
68	D	d
69	E	e
70	F	f
71	G	g
72	H	h
73	I	i
74	J	j
75	K	k
76	L	l
77	M	m
78	N	n
79	O	o
80	P	p
81	Q	q
82	R	r

CHR$()	Upper/graphics	Lower/upper
83	S	s
84	T	t
85	U	u
86	V	v
87	W	w
88	X	x
89	Y	y
90	Z	z
91	[	[
92	£	£
93	]	]
94	↑	↑
95	←	←
96	─	─
97	♠	A
98	│	B
99	─	C
100	─	D
101	─	E
102	─	F
103	│	G
104	│	H
105	╮	I
106	╮	J
107	╮	K
108	└	L
109	╲	M
110	╱	N
111	┌	O
112	┐	P
113	●	Q
114	─	R
115	♥	S
116	│	T
117	╭	U
118	╳	V
119	○	W
120	♣	X
121	│	Y
122	♦	Z
123	+	+
124	▒	▒
125	│	│
126	π	▚
127	◥	▒
128		
129	ORANGE *	
130		
131	(RUN)	
132		

CHR$()	Upper/graphics	Lower/upper
133	F1 key *	
134	F3 key *	
135	F5 key *	
136	F7 key *	
137	F2 key *	
138	F4 key *	
139	F6 key *	
140	F8 key *	
141	(shifted RETURN)	
142	(UPPERCASE) *	
143		
144	BLACK *	
145	CRSR UP (shifts of 17–20)	
146	RVS OFF	
147	CLEAR SCREEN	
148	INSERT	
149	BROWN *	
150	LIGHT RED *	
151	DARK GRAY *	
152	MEDIUM GRAY *	
153	LIGHT GREEN *	
154	LIGHT BLUE *	
155	LIGHT GRAY *	
156	PURPLE *	
157	CRSR LEFT	
158	YELLOW *	
159	CYAN *	
160	SHIFTed SPACE	
161		
162		
163		
164		
165		
166		
167		
168		
169		
170		
171		
172		
173		
174		
175		
176		
177		
178		
179		
180		
181		
182		
183		
184		

CHR$()	Upper/graphics	Lower/upper
185		
186		
187		
188		
189		
190		
191		
192		
193		A
194		B
195		C
196		D
197		E
198		F
199		G
200		H
201		I
202		J
203		K
204		L
205		M
206		N
207		O
208		P
209		Q
210		R
211		S
212		T
213		U
214		V
215		W
216		X
217		Y
218		Z
219		+
220		
221		
222		
223		
224		
225		
226		
227		
228		
229		
230		
231		
232		
233		
234		

CHR$()	Upper/graphics	Lower/upper
235	├	├
236	▚	▚
237	└	└
238	┐	┐
239	▬	▬
240	┌	┌
241	┴	┴
242	┬	┬
243	┤	┤
244	│	│
245	▌	▌
246	▐	▐
247	▁	▁
248	▔	▔
249	▂	▂
250	┘	✓
251	▖	▖
252	▝	▝
253	┘	┘
254	▘	▘
255	π	▚

(*Note:* the PET codes for 97–127 repeat 33–63.)

The program that printed these characters:

```
100 REM  PRINT COMMODORE CHARACTERS
110 REM  ON COMMODORE PRINTER
120 OPEN 4,4,0 : REM UPPER/GRAPHICS
130 OPEN 5,4,7 : REM LOWER/UPPER
140 FOR I = 33 TO 127
150 PRINT#4, I "       " CHR$(I) "        ";
160 PRINT#5, CHR$(I)
170 NEXT I
180 :  REM  GRAPHICS SET
190 FOR I = 160 TO 255
200 PRINT#4, I "       " CHR$(I) "        ";
210 PRINT#5, CHR$(I)
220 NEXT I
230 END
```

APPENDIX TWO

Basic Keywords and Abbreviations

This glossary lists BASIC keywords, their abbreviated form, a sample use, and a brief description of their meaning.

KEYWORD	ABBREVIATION	EXAMPLE	DESCRIPTION
AND	A shift-N		logical operator
ASC	A shift-S	ASC(X$)	gives the PETASCII values of the first character in X$
ASC		ASC("Q")	returns the PETASCII values of the character in quotes
CHR$	C shift-H	CHR$(X)	gives the PETASCII character for X (a number from 0 to 255)
CLOSE	CL shift-O	CLOSE 2	closes a file and flushes the buffer contents
CLR		CLR	erases all variables: used when restarting a program (i.e., CLR : GOTO 100)
CMD	C shift-M	CMD 4	assigns output to the device number specified in the OPEN statement (usually used for listing to the printer)
DATA	D shift-A		stores constants for a READ statement
DIM	D shift-I	DIM M(25)	assigns memory space for arrays of more than ten elements
END	E shift-N		stops program execution; usually used before subroutines to avoid accidental execution
FOR	F shift-O	FOR I = 1 TO 100	part of repeat command to execute lines several times; NEXT closes the loop
GET	G shift-E	GET A$ / GET#6, P$	reads a single character at a time; can be used with file number to read single characters from a file

KEYWORD	ABBREVIATION	EXAMPLE	DESCRIPTION
GOSUB	GO shift-S	GOSUB 250	sends execution to subroutine until RETURN is encountered
GOTO	G shift-O	GOTO 550	transfers execution to line number listed
IF		IF X = 5 THEN 300	used with THEN to test a statement and control program branching
INPUT		INPUT A$ / INPUT "NAME"; A$	gets a line of input from keyboard (doesn't accept , ; :) (accepts quotes ["] only as *first* character)
INPUT#	I shift-N	INPUT#1, A$	gets a line of input from device specified by OPEN
INT		INT (X/3)	truncates number to integer (whole number) part
LEFT$	LE shift-F	LEFT$(X$,3)	takes the number of characters specified, starting at the left
LEN		LEN (Z$)	returns the number of characters in a string (its LENgth)
LET	L shift-E	LET X = 3 / x = 3	assigns a value to a variable (optional)
LIST	L shift-I		lists a program, line ranges may be specified (LIST 300–350)
LOAD	L shift-O	LOAD "TRIAL",8	loads a program from the device specified (tape is default)
MID$	M shift-I	MID$(A$,5,3)	MID$(A$,N,X) selects X characters of the string A$ starting at the Nth character
		MID$(A$,6)	MID$(A$,N) selects the remainder of the string A$ starting at the Nth character
NEW			erases a program from the computer's memory
NEXT	N shift-E	NEXT J	ends a FOR NEXT loop
NOT	N shift-O		logical operator
OPEN	O shift-P	OPEN 4,4 OPEN 4,4,7 OPEN 1,8,8, "SEQ FILE,S,W" OPEN 2,8,2 "REL FILE,L,"+ CHR$(32)	assigns a file to a device number; opens buffers in computer (and disk drive)
OR			logical operator
PEEK	P shift-E	PEEK (50003)	returns the value of the memory location specified
POKE	P shift-O	POKE 59468,12	puts the value given (12) into the memory location (59468)
PRINT	?		prints on the screen
PRINT#	P shift-R	PRINT# 4, X$	prints a string or variable value to the file (device) specified

KEYWORD	ABBREVIATION	EXAMPLE	DESCRIPTION
READ	R shift-E	READ A$, B$	used to read constants in a DATA statement
REM			allows non-executed programmer remarks
RESTORE	RE shift-S		resets READ pointer to first DATA statement item
RETURN	RE shift-T		ends a subroutine; returns to statement immediately following the calling GOSUB
RIGHT$	R shift-I	RIGHT$(X$,3)	RIGHT$(A$,X) returns rightmost X characters of A$
RUN	R shift-U		starts execution of a program; starting line may be specified (RUN 525)
SAVE	S shift-A	SAVE "SAMPLE",8 SAVE "TAPE"	saves a program to the device specified (tape is default)
SPC(	S shift-P	SPC(10)	spaces over by the number specified; used to format printer output as TAB doesn't work properly on printer
STATUS	ST		holds the status of the last input/output operation
STEP	ST shift-E	FOR I = 1 TO X STEP 3	sets the step in a FOR NEXT loop
STOP	S shift-T		halts execution, program resumes by typing CONTinue; usually used only for debugging
STR$	ST shift-R	STR$(N)	changes a number into a string
TAB(	T shift-A	TAB(12)	tabs to Nth position on the screen; usually doesn't work on printers [see SPC(]
THEN	T shift-H	IF X < 6 THEN ? "TOO SMALL"	part of IF . . . THEN statement
TIME	TI		number of "jiffies" (one-sixtieth second) since computer was turned on
TIME$	TI$		string indicating hh/mm/ss since computer was turned on
VAL	V shift-A	VAL(X$)	returns the numeric value of X$; if X$ is nonnumeric, returns zero
VERIFY	V shift-E	VERIFY "TEST",8 VERIFY "TAPE" VERIFY	compares program in memory with program of given name on device specified; tape is default

APPENDIX THREE
Error Messages

Glossary of computer and disk/cassette error messages, possible cause, and action to clear. This is not an exhaustive list of Commodore error messages—see your computer, tape recorder, and disk drive manuals. This is a list of those errors where corrective action is unclear or not explained in the manual, and of those errors relating directly to data file programming covered in this book.

ERROR	PROBABLE CAUSE/ACTION TO TAKE
BAD DATA	Reading a string with a numeric variable > Check for mistake in INPUT (after the variable, did you leave out $ or space before $?), or mistake in PRINT#—do your INPUT and PRINT statements match for number of variables and variable types?
DEVICE NOT PRESENT	Device specified in an OPEN statement is not connected or not turned on > Check for correct device number in OPEN statement; check for secure connections; check power on. May cause crash; RUN/STOP-RESTORE and PRINT ST (?ST); if –128, this is problem.
FILE DATA	INPUT# reads a string with a numeric variable > (see BAD DATA)
FILE NOT FOUND	File specified was not found. On tape, an END OF TAPE (from OPEN1,1,2) > Misspelled name; wrong tape or disk. On tape, if more than one file is saved with END OF TAPE, you have to manually fast-forward past the first before running a program to read the next. Either use END OF FILE (OPEN 1,1,1), or save just one file on each tape. *Note:* SCRATCH doesn't report FILE NOT FOUND, but gives zero as number of files scratched: 00, FILES SCRATCHED, 00, 00 (the next to last 00 indicates the number of files scratched).

ERROR	PROBABLE CAUSE/ACTION TO TAKE
FILE NOT OPEN	File specified is either not OPENed, or has been closed > Check for match of file numbers or missing OPEN statement.
FILE OPEN	An OPEN statement has referenced a file number already OPENed > *This error closes files in the computer.* When using disk files, to properly close files on your disk, immediately open, read, and close the disk error channel Use > if you have loaded the Wedge. For the PET, type: OPEN 15,8,15:CLOSE15 For the C-64, type and RUN this line: 9999 OPEN 15,8,15 : INPUT#15,X : CLOSE 15 RUN 9999 In a program, check your file numbers. Most commonly happens in direct mode (such as when printing listings), where it is a harmless annoyance.
ILLEGAL QUANTITY	The argument of a function is too large or too small > Usually a typing fault; most parameters are 0 to 255. LEFT$, RIGHT$, and MID$ require 1 to 255. If using an evaluated expression, check the results of your calculation [for example, LEN (A$) – 6 resulting in 0, in MID$(A$,LEN(A$)–6)].
LOAD	Program isn't loading properly from tape or disk > For tape, check the STATUS by typing ?ST; if less than 32, the program may still run. Clean and demagnetize the recorder's tape heads; if problem persists, have your tape recorder's head realigned. Sometimes, removing and reinserting the tape will help. For disk, try reinserting disk, reading the Directory, or INITIALIZing the disk.
NOT INPUT FILE	Attempt was made to read (INPUT# or GET#) from write file > You need to close a write file before you can read. Or did you type INPUT# when you meant PRINT#? Also commonly a case of switched file numbers.
NOT OUTPUT FILE	Attempt was made to write (PRINT#) data to read file > Close read file before opening write file; see 'NOT INPUT FILE'.
OUT OF MEMORY	Out of memory; too many nested FOR NEXT loops; too many nested GOSUBs > Type ?FRE(O). If number is small (under 100), you're out of memory. Large DIMensioned arrays eat up space fast; try dimensioning as small as possible. If number is negative (C-64), or is large (PET or C–64), then you have too many nested FOR NEXTs or GOSUBs. Common fault is to have a GOSUB that contains another GOSUB without ever making it to the RETURN. Improperly closing a FOR NEXT loop (FOR X = 1 TO 6 . . . NEXT Y), or calling

ERROR	PROBABLE CAUSE/ACTION TO TAKE
	GOSUBs without RETURNs from a FOR NEXT loop will give this error. Infrequent, but annoying and very hard to find.
REDIM'D ARRAY	A DIM statement includes a variable which was already dimensioned > Most commonly encountered when a GOTO returns to the beginning of the program. Either precede the GOTO with CLR to reset all variables, move the DIM statement to an earlier line, or change the GOTO to a line number past the DIM statement.
REDO FROM START	Computer wants numerals and you've typed a string > As programmer, you need to write your programs so the user never sees this message; they will try to start from the beginning of the program. . . .
RETURN WITHOUT GOSUB	A RETURN statement was encountered without a previous GOSUB > You forgot the END statement after the last line of your program, before your subroutines; the program then ran the first subroutine until it came to RETURN. Or, you used GOTO to go to a subroutine, then used RETURN instead of another GOTO to return (learn to use GOSUB properly!).
STRING TOO LONG	A string is more than 255 characters long > Concatenating a string that is more than 255 characters from two legal-length strings; reading incorrectly written data. Check your PRINT# statements to be sure you either have each variable in a separate PRINT# statement, or have carriage returns separating each variable. A missing carriage return can concatenate data in a file. Forgetting the semicolon between variables, missing quotes around commas between variables. (You always use separate PRINT# statements, don't you?)
VERIFY	Program on tape/disk doesn't match program in memory > Did you type the correct name? For tape, did you rewind the tape? Did you remember to press *both* the play and record keys to SAVE your program? For disk: You may have had a disk error when you tried to SAVE the program, so that it wasn't saved. Typically, you tried to SAVE a program using a name existing on that disk, so the program in memory was not saved. Check your spelling, give it a new name, or use 'save-with-replace' (SAVE "@0:NEWPROGR",8).

DISK ERRORS

ERROR	PROBABLE CAUSE/ACTION TO TAKE
20, READ ERROR 21, READ ERROR	Drive can't read disk > Disk is not NEWed (formatted), disk is not present, disk door is not closed, disk is not seated properly (open door, remove, and reinsert disk), or catastrophic failure of disk. If you get these errors frequently on commercial disks, your drive may need to have its timing and alignment checked.
23, READ ERROR 24, READ ERROR 27, READ ERROR	Data read, but is in error > Disk not seated properly, problems with disk; data is on the disk, and can be read, but is garbled. This is why you make backups!
25, WRITE ERROR	Data written to disk don't match data in buffer > Commodore does "Write with verify:" after data is written to the disk, it is read back to check against the original (just like VERIFY). While it slows down the SAVE, it helps protect your data. Try VALIDATing (COLLECTing) the disk, or use another disk. In a 4040 or 8050, try the other drive. Most common using DUPLICATE (BACKUP) or COPY; reNEW the copy disk and try again. If it occurs during DUPLICATE (BACKUP) or VALIDATE (COLLECT), remove disk, turn drive off, then back on, reinsert disk. If BACKUP doesn't work, try NEWing disk, then using COPY (C1=0).
26, WRITE PROTECT ON	Disk has tab over write-enable notch > Be sure you really want to SAVE to that disk, then remove the tab.
29, DISK ID MISMATCH	Disk has not been initialized > Whenever you change disks, you should initialize the drive (this causes it to reread the map of where data is stored on the disk, so it writes to the correct places). You should also try to use a unique ID for each disk when you NEW (HEADER) it. To initialize, type OPEN1, 8, 15, "I0" : CLOSE 1 Whenever your program requires changing disks, include the initialization command in your program (assuming file 15 is open as the disk error channel) by PRINT#15, "I0" For dual drive systems, you initialize drive 1 with "I1". The 8050 and 8250 automatically initialize a disk when it is inserted. 1541 drives often report this when trying to read disks created on a 4040.
30–39, SYNTAX ERRORS	See your disk manual, where they are clearly explained. Usually either you misspelled something, or you didn't put it in the correct format.

RELATIVE FILE ERRORS

ERROR	PROBABLE CAUSE/ACTION TO TAKE
50, RECORD NOT PRESENT	Relative files: read past last record using INPUT# or GET# > May be ignored if you are expanding the file, otherwise, reposition the pointer before trying to INPUT# or GET#. One reason for creating a dummy last record when creating a relative file is to avoid this error.
51, OVERFLOW IN RECORD	Data written exceeds space allocated > Include error checks in your program to detect records which are too long and either shorten them, or have the user re-enter them.
52, FILE TOO LARGE	Not enough room on disk > You need either less records, or less space in each record. Since relative files allocate space on the disk when they are set up, if you follow the suggestion of writing a "dummy" last record, you can catch this error at the beginning of your program, and give the user a message to decrease either the number of records, or the amount of space allocated to each record, or to use another disk. If you encounter this error with an almost full file, you are likely to terminate your program and lose data.

DISK FILE MESSAGES

ERROR	PROBABLE CAUSE/ACTION TO TAKE
60, WRITE FILE OPEN	You are trying to OPEN a file for reading before CLOSing it. > Within a program, you may have forgotten the CLOSE, closed the wrong file, or had a bug in your program which kept it from reaching the CLOSE statement. Alternatively, you may have removed the disk (or had a problem) with an open file. Try opening, reading, and closing the disk error channel (type < when using the Wedge). For PETs, type OPEN 15, 8, 15 : CLOSE 15 For 64s, type and run this line: 9999 OPEN 15, 8, 15 : INPUT#15, X :CLOSE 15 RUN 9999 Then check the Directory. Unclosed files have an "*" before their type: *PRG, or *SEQ. If closing disk files didn't fix it, you *must immediately VALIDATE* your disk. *Never* SCRATCH an unCLOSEd file; you will corrupt your disk and leave it open for all sorts of peculiar problems. To VALIDATE, type OPEN 15, 8, 15, "V0" : CLOSE 15

ERROR	PROBABLE CAUSE/ACTION TO TAKE
61, FILE NOT OPEN	A file number is used that has not been OPENed > Commonly, you simply typed a wrong number in a PRINT# or INPUT# statement. Also caused by closing the disk error channel, then trying to access a file. Remember, closing the disk error channel closes all open files IN THE DISK DRIVE (even though they may still be open in the computer). Check that you close the disk error channel *after* you close all open files.
62, FILE NOT FOUND	A file or program with that name is not on the disk > Usually, a typing problem. Check that you have the correct disk. Check that you are typing the correct name. LOAD "SAMPLE" will give FILE NOT FOUND if the name is SAMPLE1 or SAMPLE.PRG. Spaces are a frequent cause: "SAMPLE 1" instead of "SAMPLE1".
63, FILE EXISTS	That name is already on the disk > Use another name, or use SAVE with replace (if you want to). Usually caused by leaving out the @ sign when replacing: SAVE "@0:SAMPLE",8 is the correct format.
64, FILE TYPE MISMATCH	The disk operation requested doesn't match the file type > You are trying to LOAD a SEQ or REL file (you can only LOAD "PRG" files), or you are OPENing a file with the name of a PROGRAM on the disk. For example, calling your mailing label program, "MAIL" and using "MAIL" for the name of your SEQ file will generate this error!
70, NO CHANNEL	No channel (buffer) is available > The disk drive has ten available buffers; a sequential file takes two and a relative file takes three. CLOSE any files not in use.
72, DISK FULL	No disk space left, or directory is full > SAVE your program on another disk. Check the directory: on 4040s, the file may be left unclosed and you must immediately VALIDATE the disk. 1541s close files on this error (but check the directory anyway before you SCRATCH). If you are using a SAVE with replace, SCRATCH the old file first, then SAVE directly; SAVE with REPLACE essentially saves the program to a new space on the disk then changes the directory, so it takes twice the actual length of the program; SCRATCH and SAVE frees that space.
73, DOS MISMATCH	Disk recording format doesn't match > Some commercial disks are specially formatted so you can't copy them, and give this error when you try to copy them, or save to them. Save to your own NEWed disk.
74, DRIVE NOT READY	> Is your drive on? (Take out any disks before you turn it on) Is there a disk in the drive? 1541s can't handle numbers as first characters of file names—SAVE "3RD COPY" will give this error as it tries to save to drive 3. Include drive number: SAVE "0:3RD COPY".

APPENDIX FOUR

Differences between the PET and the C-64

This appendix covers program and memory location differences between the version of 2.0 BASIC used in the Commodore 64, and PETs with 2.0 BASIC (***COMMODORE BASIC***, mostly designated "2001"), and with 4.0 BASIC (###COMMODORE BASIC 4.0###, designated 4016, 4032, 8032).

RESERVED VARIABLES

BASIC 4.0 uses several new commands, and also has two reserved variables. Variable names cannot start with the first two letters of these commands (CO, for example). Programs written to run in both BASIC 2.0 and BASIC 4.0 must *not* use the following words, or their first two letters: DS, DS$, CONCAT, DOPEN, DCLOSE, RECORD, HEADER, COLLECT, BACKUP, COPY, APPEND, DSAVE, CATALOG, RENAME, SCRATCH, and DIRECTORY. Improper use will result in "?SYNTAX ERROR," and possibly in destruction of program or file data on disks.

INPUT

On the Commodore 64, pressing RETURN with no input simply enters an empty string, or keeps a previous value of the variable, as INPUT works on most other computers. (This is a source of many hard-to-find bugs!) On the PET, pressing RETURN before any other character stops the program. While it can be restarted with CONT, this is not helpful to users. There are three ways to avoid this problem. One is to use a GET routine for all INPUT, although that has its own problem: the RUN/STOP key can abort the program. The other two involve modifying the INPUT statement.

One solution is to open a file to the keyboard (device zero). Since it won't accept an empty (null) entry, the problem is solved. A typical use would be:

```
100 OPEN 3, 0, 1 : REM OPEN KEYBOARD FILE
110 :
200 PRINT "YOUR NAME "; : INPUT#3, N$ : PRINT
210 IF LEN(N$) = 0 THEN PRINT "[UP][UP]" : GOTO 200
220 :
```

Note that the INPUT# statement cannot have a prompt; the prompt message must be provided by a separate PRINT statement. By following the PRINT statement with a semicolon, the '?' from the INPUT will follow the prompt, just as in a normal INPUT statement.

The other solution is to put some character at the location of the cursor, so an accidental RETURN will enter that character. While it's not as foolproof, it's simple:

```
210 INPUT "YOUR NAME[SHSP][SHSP][SHSP][SHSP][SHSP][LEFT][
    LEFT][LEFT]"; N$
220 IF N$ = "[SHSP]" THEN PRINT "[UP][UP]" : GOTO 210
230 :
```

Following "NAME" are four SHIFT-SPACES, followed by three CRSR-LEFTs (which also are SHIFTed). This places a shifted space under the cursor. A RETURN enters a shifted space, which is checked for in line 220. The print statement has two CRSR-UPs (also shifted), so the line is simply reprinted in the same location on the screen. If your programs are likely to be used on both PETs (or CBMs) and Commodore 64s, you need to consider one of these solutions to the problem!

There is an interesting quirk in the INPUT routine in the PET, VIC 20, and Commodore 64, which causes a problem on the latter. If a prompt in an INPUT statement is thirty-eight or more characters long, the '?' will appear on the next line. On the Commodore 64, pressing RETURN will cause the prompt to be entered as part of the input, a very nasty situation. Beware! If your prompt plus the expected input exceeds thirty-eight characters, consider putting the prompt in a PRINT statement, and the INPUT on a line by itself.

PEEKS AND POKES

Although the PET, CBM, and Commodore 64 can interchange programs, their memory locations are not similar. If you write programs using special features of the 64 (such as function keys), they can't be used on the PET. The problem you are most likely to run into is the use of upper/lowercase, involving POKEs to the computer's memory.

For the PET and CBM, POKE 59468,14 sets the machine to lower/uppercase mode, and there is no way to reset it from the keyboard without typing another POKE. On the Commodore 64, even if you set the upper/lowercase with a POKE, it can be changed with the SHIFT-COMMODORE keys. While there is a way to disable the effect of the SHIFT-COMMODORE keys, it may cause problems on the CBM computers, where re-enabling it activates the screen TAB function.

There is a way to set the appropriate mode for both machines, and avoid the problems. Here it is:

```
101 IF PEEK(50003) <> 0 THEN POKE 59468,14 : REM PET/CBM
102 IF PEEK(50003) = 0 THEN PRINT CHR$(14) CHR$(8) : REM
    C-64
```

CHR$(14) sets lowercase mode and CHR$(8) disables the SHIFT-COMMODORE keys, so the user cannot accidentally change the mode. Location 59468 in PET/CBM computers selects uppercase/graphics (12) or lowercase (14) modes.

Location 50003 contains the following values:

1.0	BASIC PET	0
2.0	BASIC PET/CBM	1
4.0	BASIC PET/CBM	160
2.0	BASIC C-64	0

The value is dependent only upon the version of BASIC and not the version of the machine. Since there are few 1.0 BASIC (small keyboard and built-in tape recorder) models still in use, you probably don't need to take them into account. They cannot access disk drives, so disk programs may safely ignore them.

At the end of your program, it is courteous to reset the machine with the following:

```
9998 IF PEEK(50003) <> 0 THEN POKE 59468,12 : REM PET/CBM
9999 IF PEEK(50003) = 0  THEN PRINT CHR$(142) CHR$(9) : RE
     M C-64
```

The CHR$(9) reenables the SHIFT-COMMODORE key, and CHR$(142) sets uppercase mode, as does the POKE.

GARBAGE COLLECTION

As strings are concatenated, used, and discarded, they fill up memory. When they fill up all available memory, "garbage collection" gets rid of unused and discarded strings to reclaim space. BASIC 4.0 is very efficient in handling garbage collection. However, in BASIC 2.0 (older PETs, C-64s), garbage collection can take from 20 seconds to 20 minutes, or longer. Programs written for the Commodore 64 should make minimal use of concatenation and other string operations to minimize garbage collection. For example, this statement

```
100 IF LEN(A$) < 15 THEN A$ = A$ + " " : GOTO 100
```

should be replaced by

```
100 A$ = LEFT$(A$, A$ + "              ", 15)
```

This substitutes two string actions for up to fourteen, takes about thirty bytes instead of about seventy, and leaves fewer pointers to be checked.

If you are running a program and your computer suddenly seems to "go to sleep"—it may be garbage collection. It might also be a program error; tracing your program will tell you which. If the hang-up always happens in the same line, it's your program.

BASIC 4.0 DISK COMMANDS

If you are using a PET or CBM with 4.0 BASIC, you may want to use the convenient and powerful 4.0 BASIC disk commands (see Appendix 5). However, if you use the disk commands used in this book, your program will run on any PET, CBM, or Commodore 64.

BASIC 1.0—SMALL KEYBOARD PETS

BASIC 2.0—MODEL 2001 PETS

Model 2001 PETs which display "### Commodore Basic ###" and small keyboard PETs ("*** Commodore BASIC ***") must end *all* PRINT# statements with

```
;CHR$(13);
```

BASIC 1.0 PETs can't use disk drives, but must make the same change when using cassette files: each PRINT# statement must end with

```
;CHR$(13);
```

BASIC 1.0 and 2.0 PRINT# statements sent a carriage return followed by a linefeed [CHR$(10)]. This extra linefeed shows up as part of your data, causing all sorts of problems. The CHR$(13) sends the necessary carriage return, and the semicolon supresses the 'end of PRINT,' so the linefeed is not sent. Commodore 64 BASIC and BASIC 4.0 send only a carriage return without linefeed.

Need we remind you to put each variable in its own PRINT# statement?

BASIC 4.0: `100 PRINT#2, A$ : PRINT#2, B$`

BASIC 1.0/2.0: `100 PRINT#2, A$ ;CHR$(13); : PRINT#2, B$ ;CHR$(13);`

Of course, you could define a variable (CR$, for example) as CHR$(13):

```
110 CR$ = CHR$(13)

250 PRINT#2, A$ ;CR$; : PRINT #2, B$ ;CR$;
```

APPENDIX FIVE

Basic 4.0 Disk and File Commands

If you are using a computer with BASIC 4.0, the disk commands are built in, and both simpler and more powerful than those available in BASIC 2.0. The tradeoff is that programs written using these commands *cannot* be used on BASIC 2.0 computers—older PETs, and the C-64. This appendix is written as a translation guide to give you the BASIC 4.0 format for the equivalent BASIC 2.0 commands as used in this book.

When using variables within BASIC 4.0 commands, they must be enclosed in parentheses: DCLOSE# (FN); SCRATCH (F$); RENAME (F$), D0 TO (F2$).

With a dual drive system, in BASIC 4.0, the second drive is specified by adding ,D1 to the command:

```
DSAVE "SAMPLE" ,D1
```

In BASIC 2.0, the drive number must be specified; in BASIC 4.0, drive 0 does not have to be specified; it is the default value.

Pattern matching can be used with some disk commands. A '?' allows any character, and '*' matches anything. Pattern matching can be used with LOAD, OPEN for read (after the file exists), and SCRATCH. Be careful when using pattern matching with SCRATCH, as it is easy to erase the wrong files. Pattern matching is most useful with LOAD. See your disk manual for more information and for examples.

The BASIC 2.0 commands are printed at the left margin; the BASIC 4.0 commands are indented on the next line, along with their abbreviation.

DISK ERROR CHANNEL

```
INPUT#15, X, X$ ,Y ,Z
PRINT X$
      ? DS$
```

Reads and displays the disk error channel. *Note that* DS *and* DS$ *are reserved words in* BASIC 4.0; programs being written to run on both versions must *not* use DS or DS$ as variables. INPUT DS$ will give a ?SYNTAX ERROR in BASIC 4.0.

PROGRAM COMMANDS

LOAD "name", 8

DLOAD "name" D shift-L

for drive 1 in dual drives, DLOAD "name", D1 The default is drive 0.

SAVE "name", 8

DSAVE "name" D shift-S

for dual drives, drive 1 must be given: DSAVE "name", D1. For replacement, DSAVE "@name"

VERIFY "name", 8

VERIFY "name",8 V shift-E

VERIFY checks the version stored on the disk against that in memory. Besides "?DS$", this is another way to check for a correct SAVE. VERIFY is often used to check if any changes have been made to a program since the last SAVE (a VERIFY error indicates changes have been made—the version in memory is no longer identical to that on the disk). As shown, will check both drives on a dual drive; to specify the drive, use VERIFY "0:name",8 (for drive 0).

FILE COMMANDS

APPEND

OPEN 5, 8, 5, "file,A"

APPEND #5, "file" A shift-P

OPEN 1, 8, 2, "file,S,R"

DOPEN #1, "file" D shift-O

Defaults are SEQUENTIAL, READ, drive 0. To WRITE, DOPEN #1, "file" ,W. Note that the ,W are *outside* the quotes!

OPEN 2, 8, 2, "file,L," + CHR$(64)

DOPEN #1, "file", L64 D shift-O

To open a relative file for writing, the record length must be given (here it is 64). When reading, OPEN 2, 8, 2, "file" is replaced by DOPEN #2, "file" : DOS knows it is a relative file.

CLOSE 1

DCLOSE #1 DC shift-L

Must be used to close files opened with DOPEN. DCLOSE without a file

number will close *all* open files; it is equivalent to a CLOSE 15 following an OPEN 15, 8, 15 (the disk error/command channel).

PRINT#15, "P" CHR$(2)CHR$(3)CHR$(6)

RECORD#2, 56, 20 R shift-E

RECORD#2, 56

RECORD uses the *file number* (as opposed to the *channel* or *secondary address* number used by "P"); then the record number (56); the byte number (20) is optional. RECORD positions the file pointer for PRINT#, INPUT# or GET#.

DISK COMMANDS

The major change is that while all BASIC 2.0 commands have to be in a PRINT# statement OPENed to the disk command/error channel, BASIC 4.0 commands stand alone. Also note the reversal of the order in COPY, DUPLICATE, and RENAME.

FORMATTING A NEW DISK

PRINT#15, "NO:NAME,ID"

HEADER "name", Iid, D0 H shift-E

The id designation must be preceded by an I; it is important that each disk have a unique ID so the disk controller recognizes when you have changed disks and reloads the BAM (specifying what space is available on the disk). Using F as the first character of the ID will result in SYNTAX ERROR as BASIC reads it as the keyword "IF" (see third example). The drive must be explicitly given as D0 or D1. Examples:

HEADER "NEW DISK", IND, D0

HEADER "EXAMPLE", IE9, D1

HEADER "FILES #1", I1F, D1

LOADING A DIRECTORY

LOAD "$",8 or LOAD "$0",8

DIRECTORY DI shift-R

CATALOG C shift-A

Either DIRECTORY or CATALOG can be used to read the disk directory; the directory is *not* loaded into memory, and so can be displayed without destroying the program currently in memory. On a dual drive, both drives' catalogs will be displayed; specify the drive by DIRECTORY, D1 or CATALOG D0 (note CATALOG does NOT require a comma, but DIRECTORY does).

DUPLICATE

PRINT#15, "D1=0"

BACKUP D0 to D1 BA shift-C

For use on dual drive systems only. Note reversal of drive numbers from BASIC 2.0 to BASIC 4.0!

COPYING FILES
PRINT#15, "C0:new=0:old"

COPY "old",D0 TO "new",D0 C shift-O
Copies file onto same disk with another name; note reversal of drive numbers.

PRINT#15, "C0:new=1:old"

COPY "old",D1 TO "new",D0
Copies file onto another disk with whatever name you choose (can be the same)

PRINT#15, "C0 = 1"

COPY D1 TO D0
Copies entire disk; dual drives only. Note reversal of drive numbers. Destination disk must be HEADERed first. Useful for coping contents of two partially full disks onto one disk, if all file names are different. Also useful when DUPLICATE or BACKUP give errors; COPY is less demanding and will often work.

CONCATENATING FILES
PRINT#15, "C0:new = C0:old1, 0:old2, 0:old3, 0:old4"

CONCAT "old",D0 TO "new",D0 CON shift-C
In BASIC 2.0, COPY can concatenate up to four files; in BASIC 4.0, CONCAT concatenates only two; "new" *must* be an existing file, or you will get an error message. You can concatenate files from one disk to another: CONCAT "old", D1 TO "new", D0.

RENAME
PRINT#15, "R0:new = old"

RENAME "old" ,D0 TO "new" RE shift-N
The drive designation, if used, must either preceed or follow "old": RENAME D0, "old" TO "new" or RENAME "old",D0 TO "new"

SCRATCH
PRINT#15, "S0:filename"

SCRATCH "filename" SC shift-R
For drive 1, SCRATCH "file",D1

VALIDATE
PRINT#15, "V0

COLLECT D0 CO shift-L
COLLECT reads through the chain of sectors for each entry in the directory and then rewrites the BAM (the disk's map of used and unused spaces). If COLLECT quits with an error message, the disk will be left unchanged. (But be sure to INITIALIZE the drive before any further action, to reload the disk's BAM!) COLLECT is the only safe way to remove *ed files (unclosed files) from a disk. Files marked with * in the directory should never be SCRATCHed, as that may corrupt the disk and cause later problems. COLLECT is generally used to erase *ed files or to re-allocate space on the disk. COLLECT should *never* be used on disks

which contain true "random" (i.e. "direct access") files; it will unallocate data blocks, since direct access files don't indicate used blocks in the directory. Don't confuse with "VERIFY" which only checks the accuracy of a SAVE.

The following two programs illustrate the use of BASIC 4.0 commands; one is from the Sequential Utilities chapter (6), the other from the Relative Utilities chapter (8). Compare them with the original versions.

```
100 rem   prob 6-3 solution, basic 4.0
110 rem   create or edit worklist file
120 :
130 rem   variables used
140 rem     f$ = file name
150 rem     r$, c$ = response string
160 rem     m$ = task description
170 :
180 rem   files used
190 rem     seq.: worklist (user entered)
200 rem           tempfile
210 rem     dataset: m$ (one string)
220 :
230 rem   initialize
250 print "[clr]" : rem clear screen
255 print : print : print tab(7) "Create or edi
    t [rvs]WORKLIST[off]" : print
260 print : input "File name "; f$
270 :
280 print "[rvs]N[off]ew or [rvs]O[off]ld file
    (Press [rvs]N[off] or [rvs]O[off])"
290 get  r$ : if r$ <> "n" and r$ <> "o" then 2
    90
300 if r$ = "o" then 350
310 :
320 rem   create f$ by open and close
330 dopen #1, (f$) ,w : dclose #1 : gosub 800
340 :
350 dopen #1, (f$) : gosub 800
360 dopen #2, "tempfile" ,w : gosub 800
370 :
380 print : print "[rvs]A[off]dd or [rvs]D[off]
    elete data (Press [rvs]A[off] or [rvs]D[off]
    )"
390 get r$ : if r$ <> "a" and r$ <> "d" then 39
    0
400 if r$ = "a" then 590
410 :
420 rem   delete from file routine
430 print : print "Press [rvs]RETURN[off] to ac
    cept data displayed."
440 print "Press [rvs]D[off] to delete." : prin
    t
```

```
450 :
460 input#1, m$ : let s1 = st
470 print : print  m$ :  print
480 print "[rvs]RETURN[off] or [rvs]D[off]elete
    "
490 get r$ : if r$ <> "d" and r$ <> chr$(13) th
    en 490
500 if r$ = chr$(13) then 540
510 if s1 = 0 then 460
520 goto 700 : rem  quit if e-o-f
530 :
540 print#2, m$
550 if s1 = 0 then 460
560 goto 630
570 :
580 rem   add to file routine
590 input#1, m$ : let s1 = st
600 print#2, m$
610 if s1 = 0 then 590
620 :
630 print : input "New task or [rvs]STOP[off] "
    ; m$
640 if m$ = "" then print "[up][up][up]" : goto
    630
650 if m$ = "stop" or m$ =  "STOP" then 700
660 print#2, m$
670 goto 630
680 :
690 rem   close files and rename tempfile
700 dclose : gosub 800
710 scratch (f$) : gosub 800
720 rename "tempfile" to (f$) : gosub 800
730 print : print "File updated and closed."
740 end
750 :
800 rem disk error channel
810 input#15, x, x$, y, z
820 if ds < 20 then return
830 print : print "[down][rvs]Disk error: " ds$

840 end
```

```
100 REM   POST 'TRANSACT' SEQ FILE
110 REM    INVENTORY CHANGES TO
120 REM    'BUSINESS INVEN' REL FILE
125 REM     >>> BASIC 4.0 VERSION <<<
130 :
140 REM   VARIABLES USED
150 REM     T = TRANSACTION TYPE (1,2)
160 REM     Y$ = DATE (XX-XX-XX)
170 REM     I$ = INVENTORY/RECEIPT  #
180 REM     N$,N1$,N2$= PROD. # (4)
190 REM     Q1 = QUAN. ADDED OR SUBTRACTED FROM
      STOCK (3)
200 REM     R$ = USER RESPONSE
210 REM     P$ = PROD. DESCR. (20)
220 REM     S$ = SUPPLIER (20)
230 REM     R  = REORDER PT. (3)
240 REM     Y  = REORDER QTY. (3)
250 REM     Q  = QUAN. IN STOCK (3)
260 REM     C  = COST (6)
270 REM     R1 = RECORD NUMBER
280 REM     U  = UNIT PRICE (6)
290 REM     H  = HI BYTE
300 REM     L  = LO BYTE
310 :       CR$= CHR$(13)
320 :
330 REM   FILES USED
340 REM     SEQ FILE: TRANSACT
350 REM     DATASET: T, Y$, I$, N$, Q1
360 REM     SEQ FILE: INDEX
370 REM     DATASET: N$, R1
380 REM     REL FILE: BUSINESS INVEN
390 REM     DATASET: N$,P$,S$,R,Y,Q,C,U
400 REM     RECORD LENGTH: 75 BYTES
410 :
420 REM   INITIALIZE
430 PRINT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN]
     POST [RVS]TRANSACT[OFF] TO [RVS]BUSINESS INV
     EN[OFF]."
440 PRINT : PRINT "WORKING..."
460 DOPEN #1, "TRANSACT"
461 REM  'L' IS REQUIRED FOR RELATIVE WRITE, BU
     T NOT FOR READ
470 GOSUB 1000
480 DOPEN #2, "BUSINESS INVEN"
490 GOSUB 1000
500 :
510 REM   READ 'TRANSACT' & FIND RECORD # FROM
     'INDEX'
520 INPUT#1, T, Y$, I$, N1$, Q1
530 PRINT "UPDATING PROD. " N1$
540 LET SV = ST
550 DOPEN #3, "INDEX" : GOSUB 1000
```

```
560 INPUT#3, N$, R1
570 LET S2 = ST
580 IF N$ = N1$ THEN DCLOSE #3 : GOTO 650
590 IF S2 = 0 THEN 560
600 :
610 REM   RECORD NOT FOUND
620 PRINT : PRINT "[RVS] PROD. # NOT FOUND IN '
     INDEX.'" : DCLOSE #3 : GOTO 920
630 :
640 REM   FIND & CHANGE Q IN REL FILE
660 RECORD #1, (R1)
670 GOSUB 1000 : IF DS = 50 THEN 910: REM NO RE
     CORD #
680 INPUT#2, N2$, P$, S$, R, Y, Q, C, U
690 IF T = 1 THEN LET Q = Q + Q1 : GOTO 730
700 IF T = 2 THEN LET Q = Q - Q1 : GOTO 730
710 :
720 REM   PRINT UPDATE TO REL FILE
730 RECORD #2, (R1)
740 GOSUB 1000
750 PRINT#2, N2$ ;CR$; P$ ;CR$; S$ ;CR$; R ;CR$
     ; Y ;CR$; Q ;CR$; C ;CR$; U
760 IF SV = 0 THEN 520 : REM NEXT TRANSACT
770 :
780 REM   CLOSE FILES
790 PRINT : PRINT "TRANSACTIONS POSTED, FILES C
     LOSED."
800 DCLOSE : GOSUB 1000
810 END
820 :
830 :
900 REM   ERROR IN 'BUSINESS INVEN'
910 PRINT : PRINT "[RVS] RECORD NUMBER NOT FOUN
     D IN 'BUSINESS INVEN'."
920 PRINT "TRANSACTION DATED " Y$
930 PRINT "INVOICE/RECEIPT # " I$
940 PRINT "FOR PRODUCT # " N1$
950 PRINT "[RVS]NOT PROCESSED ! "
960 GOTO 760 : REM NEXT TRANSACT
970 :
980 :
1000 REM   DISK ERROR ROUTINE
1020 IF DS < 20 OR DS = 50 THEN RETURN
1030 PRINT : PRINT "[RVS]DISK ERROR: " DS$
1040 DCLOSE : END
```

Index to Programs

Program titles are shown in CAPITALS. The page number is preceded by the chapter number, F for Final Self-Test, or A for Appendix (e.g., 4–150 is on page 150 in Chapter 4).

The description for the program creating the file includes a brief indication of the contents of each variable.

Check for numeric **3–112**
Use ASCII to check for numeric data in entry string

Check for numeric **3–112**
Use STR$ to check for numeric data in entry string

CHECKBOOK create **8–424**
Creates CHECKBOOK (seq) monthly transaction files
Dataset: C(check#), Y$(date), W$(source), N$(acct#), D (amt)

CHECKBOOK display **8–425**
Displays data in CHECKBOOK (seq) file
Dataset: C, Y$, W$, N$, D

Compare files **7–401**
Read and display datasets from RCREDIT & RCREDIT COPY (rel)
Dataset: N$, C$, R

CREDIT (1) edit **6–278**
Change CREDIT (seq) data
Dataset: C$(cust#), N$(name), R(rating)

CREDIT (1) edit **6–283**
Change CREDIT (seq) data, uses 'Press RETURN for next dataset'
Dataset: C$, N$, R

CREDIT (2) edit **6–284**
Delete or change CREDIT (seq) data
Dataset: C$, N$, R

CREDIT (3) edit **6–292**
Insert, delete, or change CREDIT (seq) data
Dataset: C$, N$, R

CREDIT create **5–256**
Create CREDIT (seq) file of customer credit ratings
Dataset: C$(cust#), N$(name), R(rating)

CREDIT display **5–258**
Display CREDIT (seq) data
Dataset: C$, N$, R

CREDIT tape edit **4–170**
Read, edit, and rewrite CREDIT (tape) data
Dataset: C$(X,1)(cust#), C$(X,2)(name), C$(X,3)(rating)

CREDIT to RCREDIT **7–380**
Copy CREDIT (seq) to RCREDIT (rel)
Dataset: N$(cust #), C$(name), R(rating)

GET seq reader2 **5–220**
Prints ASCII value of characters in seq file

GET tape reader **4–139**
Concatenates characters and prints as string (S$)

GET tape reader **4–140**
Prints ASCII value of each character in file

GROCERY create **5–255**
Create GROCERY (seq) file of item and quantity to buy
Dataset: D$(item), Q(quan)

GROCERY display **5–256**
Display GROCERY (seq) data
Dataset: D$, Q

GROCERY tape add **4–162**
Adds to tape file using arrays
Dataset: D$(descr), N(#)

INDEX create **7–398**
Create INDEX (seq) index to BUSINESS INVEN (rel) to find data quickly
Dataset: N$(prod. #), R1(relative file record)

INDEX display **7–399**
Display INDEX (seq) datasets
Dataset: N$, R1

INPUT seq reader **5–219**
Uses INPUT to read any seq file
Dataset: A$

INPUT tape reader **4–139**
Reads tape file with INPUT
Dataset: A$

INVEN byte create **8–413**
Create INVEN (rel) inventory file using byte pointer
Dataset: N$(acct#), P$(prod. descr), Q(quantity)

INVEN byte display **8–414**
Display INVEN (rel) file using byte pointer
Dataset: N$, P$, Q

INVEN create **7–345**
Create relative file of inventory information
Dataset: N$(prod#), P$(descr), Q(quan)

Month search **2–52**
Find and return month number given 3 letter abbreviation (FEB returns '2')

ORDER create **4–185**
Creates ORDER (tape) of product ordering information
Dataset: D$(descr), Q(quan)

ORDER display **4–186**
Displays ORDER (tape) data
Dataset: D$(descr), Q(quan)

Personal money management **8–430**
Updates BUDGET# (rel) acc't files from MONTH# (seq) monthly transaction files
Dataset: C, Y$, W$, N$, D; N$, A$, B, E

PHONE add **7–360**
Add data to PHONE (rel) file
Dataset: N$, C$, P$

PHONE create **7–354**
Creates file of customer phone numbers (rel)
Dataset: N$(cust #), C$(name), P$(phone)

PHONE display **7–356**
Displays PHONE (rel) data
Dataset: N$, C$, P$

Product Code **3–113**
Enter & edit product code information
Dataset: M$(concatenated)

PROP INVENTORY create **5–210**
Creates file to store inventory of personal property and its value
Dataset: D$(descr), N(#), V($)

PROP INVENTORY display **5–216**
Displays file
Dataset: N$, Q, D

PROPERTY tape create **4–130**
Creates file of inventory data as PROPERTY (tape)
Dataset: D$(descr), N(#), V($)

PROPERTY tape display **4–136**
Displays PROPERTY (tape) data
Dataset: D$, N$, V

TRANSAC# Create **5–258**
Create TRANSAC1, TRANSAC2 . . . (seq) retail sales transaction files
Dataset: D1$ (concatenated dataset)

TRANSAC# display **5–260**
Display any TRANSAC# (seq) file
Dataset: D1$

TRANSAC# merge **6–305**
Merges data from TRANSAC1 & TRANSAC2 to TRANSMERGE (seq)
Dataset: D1$, D2$

TRANSACT create **8–440**
Creates TRANSACT (seq) file of inventory transactions
Dataset: T(type), Y$(date), I$(inven#), N$(prod#), Q1 (chg in quan)

TRANSACT display **8–442**
Displays TRANSACT datasets
Dataset: T(type), Y$(date), I$(inven#), N$(prod#), Q1 (chg in quan)

TRANSACT posted to BUSINESS INVEN **8–443**
Posts inventory changes from TRANSACT (seq) to BUSINESS INVEN (rel)

TRANSACT posted to BUSINESS INVEN (BASIC 4.0) **A–484**
Posts inventory changes from TRANSACT (seq) to BUSINESS INVEN (rel)

Universal file reader **8–412**
Program reads any sequential or relative file using INPUT
Dataset: A$

WORKLIST (BASIC 4.0) **A–482**
Create and edit list of tasks
Dataset: M$

WORKLIST create/edit **6–334**
Create, add to, or delete reminders of work from seq file
Dataset: M$(task)

Subject Index

NOW AVAILABLE

All the powerful programs listed in this book will make your COMMODORE 64™ more effective than ever. The programs and subroutines to set up, maintain and modify data files can go to work for *you* today!.

Save time and don't risk introducing keyboarding errors into your programs.

The COMMODORE 64™ DATA FILE PROGRAM DISK is available at your favorite book or computer store. Or use the handy order card below.

COMMODORE 64 DATA FILE PROGRAM DISK

Yes I want to manage my data files better. Please send me_____copies of the COMMODORE 64 DATA FILE PROGRAM DISK at $24.95 for each disk.

1-80752-4 $24.95

__Payment enclosed (including state sales tax). __Bill me.

__Wiley pays shipping and handling charges. __Bill my company.

____Charge to my credit card: __Visa __Master Card

Card number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Expiration Date Signature

Name Title

Company

Address

City State Zip Code

1-80752-4 263 Signature (order invalid unless signed)

We normally ship within ten days. If payment accompanies your order and shipment cannot be made within 90 days, payment will be refunded.

PUT YOUR COMMODORE 64 TO WORK TODAY!

Buy the 5¼" disk at your favorite computer store, or order from Wiley:

In the United States:	John Wiley & Sons 1 Wiley Drive Somerset, NJ 08873
In the United Kingdom and Europe:	John Wiley & Sons, Ltd. Baffins Lane, Chichester Sussex PO 19 1UD UNITED KINGDOM
In Canada:	John Wiley & Sons Canada, Ltd. 22 Worcester Road Rexdale, Ontario M9W 1L1 CANADA
In Australia:	Jacaranda Wiley, Ltd. GPO Box 859 Brisbane, Queensland AUSTRALIA

Fisher—COMMODORE 64 DATA FILE PROGRAM DISK 1-80752-4

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 2277 NEW YORK, NY

Postage will be paid by addressee:

John Wiley & Sons, Inc.
1 Wiley Drive
Somerset, N.J. 08873

Attn: Commodore 64 Data File Program Disk

Computers $14.95

COMMODORE 64™ DATA FILE PROGRAMMING

GLENN FISHER
LEROY FINKEL
JERALD R. BROWN

This easy-to-follow, self-instructional guide shows you how to program and maintain data files for your Commodore 64, Commodore Pet, or CBM, and use them to keep track of billings, inventories, and expenses ... catalog material and mailing lists ... manipulate numerical and statistical information ... and much more.

You'll learn the principles of file organization, then go on to more advanced programming techniques. Assisted by dozens of sample programs and practical advice, you'll find out how to write your own data file programs, use your tape recorder or disk drive as an alternative way to store information, modify programs you've already purchased, even adapt programs using data files found in magazines and other sources.

Commodore 64 Data File Programming's unique self-teaching format includes self-tests, objectives and exercises that help you learn at your own pace to get the absolute maximum benefit from your Commodore computer. It's the same proven approach used in the authors' earlier data file programming guides for the Apple®, TRS-80®, and IBM® PC, which *Microcomputing* called "excellent, well worth study" and *Interface Age* "highly recommended."

Glenn Fisher has been a computer programmer and owner of a software company. He has taught adults to program and to use computers for the past five years. He is currently Computer Specialist at a county office of education in California.

LeRoy Finkel and **Jerald R. Brown** have been teaching BASIC to novice computer users for over ten years. They are founders of the People's Computing Company and co-authors of twelve other Wiley Self-Teaching Guides. LeRoy Finkel is also Instructional Computing Coordinator with the San Mateo County Office of Education.

Wiley Self-Teaching Guides have taught more than two million people to use, program, and enjoy microcomputers. Look for them all at your favorite bookshop or computer store.

Commodore 64™ is a registered trademark of Commodore Electronics,Ltd.

ISBN 0 471-80734-6

WILEY PRESS
a division of JOHN WILEY & SONS, Inc.
605 Third Avenue, New York, N.Y. 10158
New York • Chichester • Brisbane • Toronto • Singapore