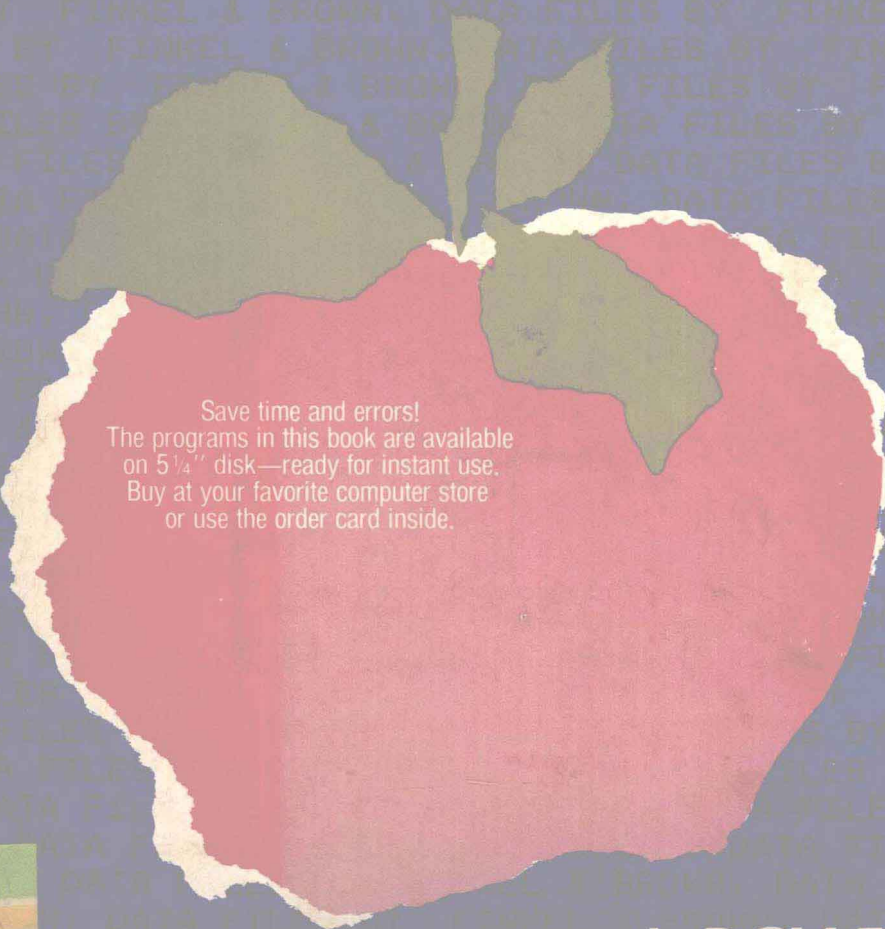


APPLE BASIC: DATA FILE PROGRAMMING

A SELF-TEACHING GUIDE



Save time and errors!
The programs in this book are available
on 5¼" disk—ready for instant use.
Buy at your favorite computer store
or use the order card inside.

LeROY FINKEL
JERALD R. BROWN

APPLETM BASIC: DATA FILE PROGRAMMING

LEROY FINKEL

San Carlos High School

and

JERALD R. BROWN

Educational Consultant



John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy V. Wilson
Editor: Dianne Littwin
Composition and Make-up: Trotta Composition

Copyright © 1982, by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data

Finkel, LeRoy.

Apple BASIC, data file programming.

(Wiley self-teaching guides)

Includes index.

1. Basic (Computer-program language) 2. Apple computer—Programming. I. Brown, Jerald, 1940-
II. Title. III. Series: Self-teaching guide.

QA76.73.B3F52 001.64'24 81-13100

ISBN 0-471-09157-X

Printed in the United States of America

82 83 10 9 8 7 6 5 4 3 2

APPLETM BASIC: DATA FILE PROGRAMMING

More than a million people have learned to program, use, and enjoy microcomputers with Wiley paperback guides. Look for them all at your favorite bookshop or computer store:

BASIC, 2nd ed., Albrecht, Finkel, & Brown
BASIC for Home Computers, Albrecht, Finkel, & Brown
TRS-80 BASIC, Albrecht, Inman, & Zamora
More TRS-80 BASIC, Inman, Zamora, & Albrecht
ATARI BASIC, Albrecht, Finkel, & Brown
Data File Programming in BASIC, Finkel & Brown
Data File Programming for the Apple Computer, Finkel & Brown
ATARI Sound & Graphics, Moore, Lower, & Albrecht
Using CP/M, Fernandez & Ashley
Introduction to 8080/8085 Assembly Language Programming, Fernandez & Ashley
8080/Z80 Assembly Language, Miller
Personal Computing, McGlynn
Why Do You Need a Personal Computer? Leventhal & Stafford
Problem-Solving on the TRS-80 Pocket Computer, Inman & Conlan
Using Programmable Calculators for Business, Hohenstein
How to Buy the Right Small Business Computer System, Smolin
The TRS-80 Means Business, Lewis
ANS COBOL, 2nd ed., Ashley
Structured COBOL, Ashley
FORTRAN IV, 2nd ed., Friedmann, Greenberg, & Hoffberg
Job Control Language, Ashley & Fernandez
Background Math for a Computer World, 2nd ed., Ashley
Flowcharting, Stern
Introduction to Data Processing, 2nd ed., Harris

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 APPLESOFT* 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 APPLE II computer (with at least one disk drive). The questions are carefully designed to call your attention to important points in the examples and explanations and to help apply what is being explained or demonstrated.

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-test can be used as a review of the material covered in the chapter. You can test yourself immediately after reading the chapter. Or you can read a chapter, take a break, and save the self-test as a review before you begin the next chapter. At the end of the book is a final self-test to assess your overall understanding of data file programming.

This book is designed to be used with an APPLE 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 teach yourself to use data files in BASIC.

*APPLE and APPLESOFT are registered trademarks of Apple Computer, Inc.

Preface

This text will teach you to program data files in APPLESOFT 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 APPLESOFT BASIC. Data files 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 reference manual for your computer system.

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.

*For programming data files in TRS-80 BASIC, MICROSOFT BASIC-80, and Northstar BASIC, read our other book, *Data File Programming in BASIC*, Finkel, LeRoy and Brown, Jerald R., John Wiley & Sons, Inc., Self-Teaching Guide, N.Y., 1981.

NOW AVAILABLE! *

All the powerful programs listed in this book will make your APPLE II™ 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. Buy the 5¼" diskette from 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

Finkel-APPLE™ BASIC DATA FILE PROGRAM DISK 0-471-86836-1

*Available April, 1982

Contents

Chapter 1	Writing BASIC Programs for Clarity, Readability, and Logic	1
Chapter 2	An Important Review of BASIC Statements	15
Chapter 3	Building Data Entry and Error Checking Routines	49
Chapter 4	Creating and Reading Back Sequential Data Files	79
Chapter 5	Sequential Data File Utility Programs	134
Chapter 6	Random Access Data Files	198
Chapter 7	Random Access File Applications	252
	Final Self-Test	281
Appendix A	ASCII Chart Code	294
Appendix B	List of Programs	296
Index		302

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 seven rules to write programs that save memory space.

INTRODUCTION

This text will teach you to use data files in APPLESOFT 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 is not a book for the absolute novice in BASIC, but is 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 APPLE II computer and the BASIC languages found on it.

Data files in other versions of BASIC and for other computers will be similar, but not identical, to those in this book. (If you are using a computer other than the APPLE II, you may want to read *Data Files Programming in BASIC*, available at your local computer store or bookstore.) You will find this text most useful when used in conjunction with the APPLE II reference manuals and the Disk Operating System (DOS) Manual: It is not a substitute for your careful reading of the APPLE II DOS Manual, though the workings of sequential and random access files are explained here in far more depth and with more examples.

Since it is assumed 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 with its own special characteristics and quirks.

In the microcomputer field, the most widely used versions of BASIC were developed by the Microsoft Company and are generally referred to as MICROSOFT BASICs. These BASICs are available on a variety of microcomputers but, unfortunately, the language is implemented differently on each computer system. The APPLE version of MICROSOFT BASIC is called APPLESOFT.

The programs and runs shown in this text were actually performed on an APPLE II and an APPLE II PLUS computer using Disk Operating System (DOS) 3.3. (They will work in DOS 3.2, as well.) We wrote all of our programs using APPLESOFT BASIC. To use the programs in INTEGER BASIC, you will have to make the usual APPLESOFT to INTEGER modifications described in your reference manual. The file commands described in this text may be used in APPLESOFT or INTEGER BASIC. For INTEGER BASIC you may have to modify the file input and output statements, as described in your DOS Manual.

Where possible, we use BASIC language features that are common to all versions of BASIC, regardless of manufacturer. We do not attempt to show off all of the bells and whistles found in APPLESOFT BASIC, but rather to present easy-to-understand programs that will be readily adaptable to a variety of 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 isolate and locate.* (Yes, you will still make errors. We all do!) This means that you should NOT use all the fanciest features available in APPLESOFT 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 the fancy features, many of which relate to printing or graphic output and do not work the same on other computers. Some are special functions that simply do not exist on other computers. Leave them out of your programs unless you feel you must include them. *The more conservative your programming techniques, the less chance 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 the one you will use one year from now; you may replace or 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 “glitch.”
- 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 the eye. 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 present 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 one to module two 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 using REMARK statements or nearly blank program lines. In general, programs designed for readability make liberal use of REMARK statements, but don't be overzealous. A nearly blank program line can be created by typing a line number followed by a colon (150:). A line number followed by REM (150 REM) can also be used.

```
100 REM DATA ENTRY MODULE
110 REM **** READ DATA FROM DATA STATEMENTS 9000-9090
120
130 REM
200 REM COMPUTATION MODULE
210 REM ARK
```

(Note: Your Apple computer will split the word REMARK into two words, as shown in line 210. Because this looks awkward, we encourage use of the word REM in place of the complete word.)

Begin each program module, block, or subroutine with an explanatory REM statement (line 100 and 110) and end it with a nearly blank line (line 120) or blank REM statement (line 130) indicating the end of the section.

Consistency in your use of REMs enhances readability. Use either REM or the nearly blank line with a colon, but be consistent. Some writers use the asterisks (****) shown in line 110 to set off REM statements containing actual remarks from blank REM statements; others use spaces four to six places after the REM before they add a comment (line 200). Both formats effectively separate REM statements from BASIC code.

You can place remarks on the same line as BASIC code using multiple statement lines, but be sure your REM 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
```

Using REMs to explain what the program is doing is desirable, but don't overuse it. (LET C = A + B does not require a REM or explanation!) REM should add information, not merely state an obvious step.

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 all 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 REM. If you or the next user of your program run short of memory space you will delete extra REM statements. This, in turn, requires you to change all of your GOTO line numbers, so plan ahead first. Some BASICs do not even allow a program to branch to a statement starting with REM.

	Bad		Good
150	GOTO 300	150	GOTO 300
300	REM DATA ENTRY	299	REM DATA ENTRY
310	INPUT "ENTER NAME:";N\$	300	INPUT "ENTER 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 REM 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's or programmer's name and the date. For the benefit of someone else who may like to use your program, include a REM 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. 9/82
120 REM
130 REM HP 2000 BASIC
140 REM MODIFIED FOR APPLESOFT BASIC BY J. BROWN
150 REM ON APPLE II, 48K
```

Follow these remarks with a brief explanation of what the program does, contained either in REM statements or in 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
```

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.

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 REMs.

```

220 REM  VARIABLES USED
230 REM  G=GROSS PAY
240 REM  N=NET PAY
250 REM  T1=FEDERAL INCOME TAX
260 REM  T2=STATE INCOME TAX
270 REM  F=SOC. SEC. TAX
280 REM  D=DISABILITY (SDI) TAX
290 REM  X,Y,Z=FOR-NEXT LOOP CONTROL VARIABLE
300 REM  H(X)=HOURS ARRAY
310 REM  N$=EMPLOYEE NAME (20 CHAR)
320 REM  PN$=EMPLOYEE NO. (5 CHAR)
330 REM
340 REM  CONSTANTS
350 LET FR = .0613: REM      SOC. SEC. RATE
360 LET DR = .01: REM      SDI RATE
370 REM
380 REM  FILES USED
390 REM  ITM=FEDL. TAX MASTER FILE
400 REM  STM=STATE TAX MASTER FILE
410 REM

```

(Notice the method used to indicate string length in lines 310 and 320.)

(Notice the use of on-line remarks in lines 350 and 360.)

The final part of the introductory module is the initialization section. In this section, dimension the size of all single and double arrays and all string arrays, even though DIMENSION is not required by your computer. This is valuable information for a reader. Any variables that need to be initialized to zero should 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.

```

410 REM  INITIALIZE
420 :
430 : DIM H(7),R(10,13),N$(30)
440 :
450 REM

```

THE MODULES THAT FOLLOW THE INTRODUCTION

The remainder of your program consists of major function modules and subroutines (and DATA statements, when they are used). Remember to separate each module from others by a blank line REM statement and a remark identifying the module. These modules can be further divided into user-defined program blocks, each separated by a blank line REM 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 some 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 "ENTER 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 the END statement) using REM 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 NUMBER ORDER.
9410 REM 10 ANSWERS, MULT. CHOICE 1-5
9420 :
9430 DATA 4,5,1,3,2,1,1,4,4,5
9440 :
9450 REM RESPONDENTS ANSWERS TO QUIZ
9460 REM DATA STATEMENT FORMAT:
9470 REM RESP. ID # FOLLOWED BY 10 RESPONSES TO QUIZ QUESTIONS
9480 :
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
9560 :
```

You can think of DATA statements as comprising a separate program module. The “inbetween” program modules might do computations, data handling, file reading and writing, and report writing. 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.

SUBROUTINES

Program control flows smoothly from one module to the next. A well-designed module has *one* entry point at its beginning and *one* exit point at its end. The exception to this is a mid-module exit to a subroutine.

```
290 :
300 REM COMPUTATION MODULE
310 :
320 LET T = (V * X) / Q
330 LET T9 = T9 + T
340 GOSUB 800
350 :
360 REM REPORT PRINTING MODULE
370 :
```