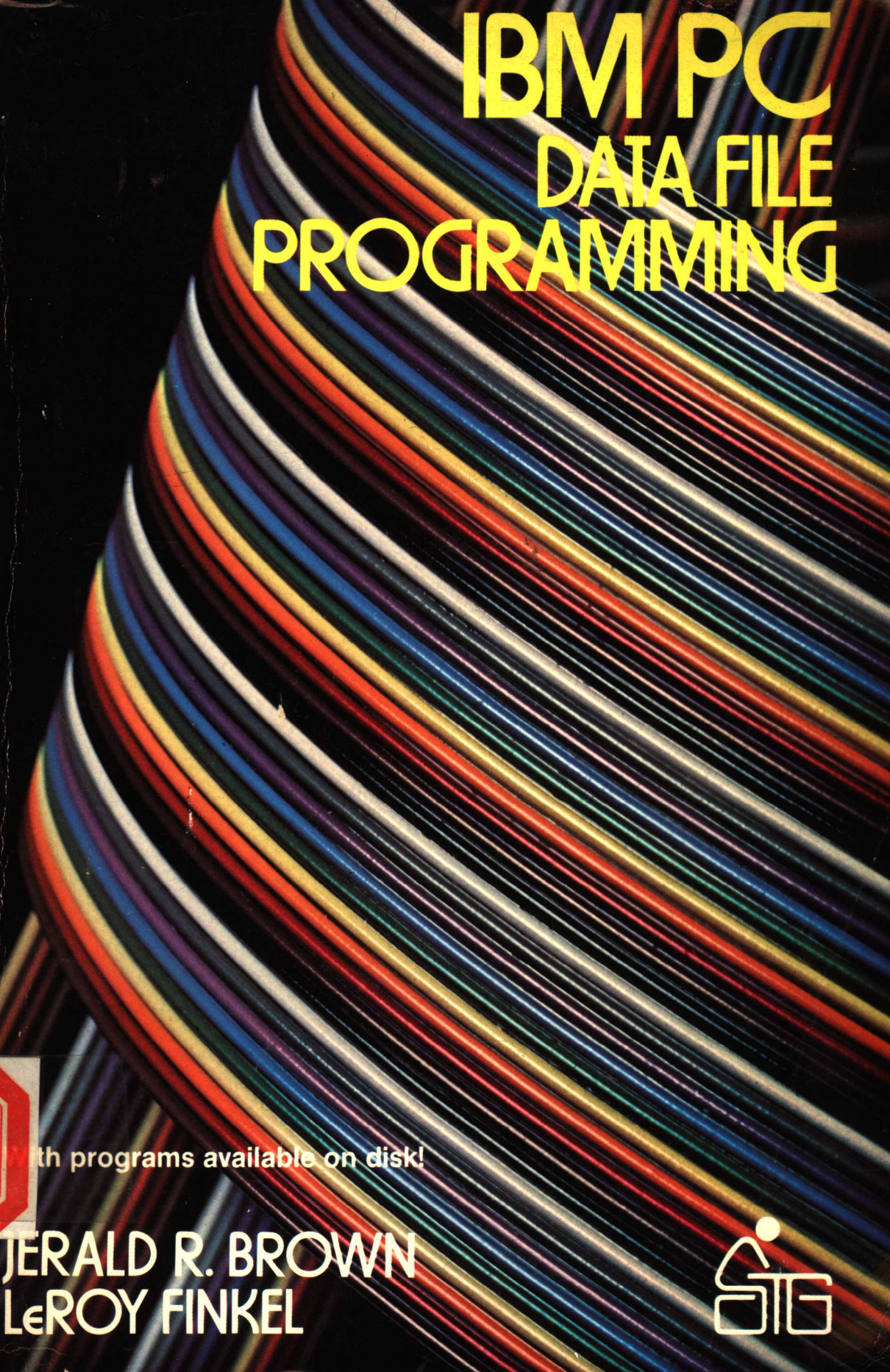
IBM PC DATA FILE PROGRAMMING



th programs available on disk!

JERALD R. BROWN
LeROY FINKEL

STG

IBM PC:
DATA FILE PROGRAMMING

JERALD R. BROWN
Educational Consultant

LEROY FINKEL
San Carlos High School

Wiley IBM PC Series: Series Editor, Laurence Press, Ph.D.

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

Publisher: Judy V. Wilson
Editor: Dianne Littwin
Managing Editor: Maria Colligan
Composition and Make-Up: Cobb/Dunlop, Inc.

Library of Congress Cataloging in Publication Data

Brown, Jerald, 1940-
IBM PC: data file programming.

(Wiley IBM PC series)
Includes index.
1. IBM Personal Computer—Programming. 2. File organization (Computer science) I. Finkel, LeRoy. II. Title. III. Title: I.B.M. P.C. IV. Series.
QA76.8.I2594B76 1983 001.64'2 82-24849
ISBN 0-471-89717-5

Printed in the United States of America

83 84 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 IBM Personal Computer (IBM PC) with one or two disk drives. The IBM PC Disk Operating System (DOS), Version 1.10, includes three versions of BASIC; cassette, disk, and advanced. The book is also compatible with IBM's new DOS Version 2.0. We used the disk version to develop this book. Cassette BASIC users can use only the files described in Chapters 4 and 5. Advanced BASIC users will find that all our programs will also work using that version of BASIC. Another popular version of BASIC called Microsoft BASIC-80 is also available for those using the CPM Operating System on the IBM PC. The programs and procedures described in this book will also work for those of you using that 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, including the manuals for *BASIC*, *DOS*, and the *Guide to Operations*.

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.

PUT YOUR IBM PC 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

Brown—IBM PC DATA FILES PROGRAM DISK 1-88906-7

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 2277, NEW YORK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

JOHN WILEY & SONS, Inc.
1 Wiley Drive
Somerset, N.J. 08873

Attn: IBM PC Data Files Program Disk

NOW AVAILABLE

All the powerful programs listed in this book will make your IBM PC 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 IBM PC DATA FILES PROGRAM DISK is available at your favorite computer store. Or use the handy order card below.

THE IBM PC DATA FILES PROGRAM DISK

Yes, I want to manage my data files better. Please send me____copies of the IBM PC DATA FILES PROGRAM DISK at $19.95 each.

1-88906-7 $19.95

___ 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 _____

1-88906-7 263 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.

Contents

How to Use This Book v

Preface vii

Chapter 1 **Writing BASIC Programs for Clarity, Readability, and Logic** 1

Introduction The BASIC Language The BASIC Language You Should Use: Conservative Programming Writing Readable Programs The Top-to-Bottom Organization REMARK Statements GOTO Statements A Format for the Introductory Module The Modules that Follow the Introduction Subroutines Just for Looks: Spacing Other Techniques to Enhance Looks and Readability Undoing It All to Save Space and Speed Up Run Time Chapter 1 Self-Test

Chapter 2 **An Important Review of BASIC Statements** 19

Introduction Variable Names String Variables READ-DATA Assignment Statements Understanding INPUT, an Important Assignment Statement The LINE INPUT Statement Concatenation The IF . . . THEN Statements IF . . . THEN String Comparisons and the ASCII Code The LEN Function Substring Functions: Versatile Tools to Manipulate String Data String Searches with INSTR Mult-Branching with ON . . . GOTO FOR NEXT Statements Multiple Statement Lines Get to Know Your IBM PC Chapter 2 Self-Test

Chapter 3 **Building Data Entry and Error Checking Routines** 61

Introduction Data Field Length Checking Data Entries for Acceptable Length "Padding" Entries with Spaces to Correct Field Lengths Stripping the Padding Spaces from Substrings in Fields Checking Entries for No Response Replacement of Data Items in a String with Defined Data Fields The VAL Function in Data Entry Checks Using STR$ to Convert Values to Strings Checking for Illegal Characters A Discussion of Data Entry and Checking Procedures Chapter 3 Self-Test

Chapter 4 **Creating and Reading Back Sequential Data Files** **106**

Introduction Data Storage on Disks Sequential Versus Random Access Data Files Initializing Sequential Data Files: Opening the File The Buffer Problem: Closing the File Placing Data into a Sequential Data File Using WRITE # or PRINT # Writing a File Reading Data from a File Permanently Removing Files from Disks Multiple File Operations in One Program Displaying One Dataset at a Time from a File Chapter 4 Self-Test

Chapter 5 **Sequential Data File Utility Programs** **168**

Making a Data File Copy Adding Data to the End of a Sequential File Changing Data in a File Editing, Deleting, and Inserting Sequential File Data Merging the Contents of Two Sequential Files Problems with Sequential Data Files A Letter-Writing Program Chapter 5 Self-Test

Chapter 6 **Random Access Data Files** **236**

Introduction What Is a Random Access File? Initializing Random Access Files Buffer Fields Simple READ and WRITE Operations to Random Access Files with String Data Adding Data to Random Access Files Using Random Access Files with Numeric Data Counting Custom-Length Records in a File Random Access File Copy Utility Programs Editing Random Access File Records Converting Sequential Files to Random Access Files A Universal Random Access and Sequential File Display Program Chapter 6 Self-Test

Chapter 7 **Random Access File Applications** **297**

Inventory Control Application Personal Money Management Application Chapter 7 Self-Test

Final Self-Test **341**

Appendix A **Basic Reference Guide for Keywords Used in This Book** **353**

Appendix B **ASCII Chart** **355**

Index to Example Programs **357**

Alphabetical Index to Data File Names **362**

Subject Index **364**

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 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 IBM Personal Computer (IBM PC) with one or two disk drives. The IBM PC Disk Operating System (DOS), Version 1.10, includes three versions of BASIC: cassette, disk, and advanced. The disk version was used to develop this book. Cassette BASIC users will need the instruction provided in Chapters 4 and 5. Advanced BASIC users will find that all our programs will also work using that version of BASIC. The programs and activities also work for IBM PC DOS version 2.0. Another popular version of BASIC, called Microsoft BASIC-80, is also available for those using the CP/M Operating System on the IBM PC. The programs and procedures described in this book will also work for those of you using that 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 IBM PC, but not new to BASIC, then it is especially important that you review the following manuals from the IBM Personal Computer Hardware and Software Reference Libraries:

Guide to Operations
DOS (Version 1.10 used for this book)
BASIC

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 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 an IBM PC using disk BASIC. 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. We did not attempt to show off all the bells and whistles found in IBM PC BASIC, but rather, to present easy-to-understand programs that will run on or be easily adapted 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 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 do not work the same 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 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 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 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 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 using an apostrophe (') as a substitute for the word REMARK, or by merely typing a line number followed by a colon (:). A line number followed by REM (e.g., 150 REM) can also be used. These nearly blank lines will help to set off and highlight program modules or routines.

```
100 REMARK       DATA ENTRY MODULE
110 REMARK **** READ DATA FROM DATA STATEMENTS 9000-9090
120 '
130 :
140 REM
.
.
190 '
200 REMARK  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. Use an apostrophe or colon consistently. Some writers use the **** shown in line 110 to set off REMARK statements containing comments from other REMARK statements; others use spaces four to six places 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 LINE INPUT "NAME? ";N$
```

Good

```
150 GOTO 300
.
.
.
.
.
299 REM   DATA ENTRY
300 LINE 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 REMARK    PAYROLL SUBSYSTEM
110 REMARK    COPYRIGHT CONSUMER PROGRAMMING CORP 1979
120 REMARK
130 REMARK    HP 2000 BASIC
140 REMARK    MODIFIED FOR IBM PC BY J. BROWN
150 REMARK    USING DISK BASIC VERSION 1.10
160 REMARK
```

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

A handy little trick used by some programmers is to place a SAVE command in a REMark statement near the beginning of every program, for example:

```
2 'SAVE "PAYSUB"
```

This allows the programmer simply to edit this line to delete the line number, the space after the line number, and the apostrophe (the shorthand for REM), and then to press the End and enter keys to execute the SAVE command.

```
EDIT 2
SAVE "PAYSUB"
Ok
```

←Press Del(ete) key 3 times so that line 2 looks like this, then press the End and enter keys, and the program is SAVEd.

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 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     IT1 = FEDL. INCOME TAX
260 REM     IT2 = 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)
320 REM     PN$ = EMPLOYEE NUMBER (5)
330 REM
340 REM   CONSTANTS
350 LET FR = .0613: REM              SOC. SEC. RATE
360 LET SDR = .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 and 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. 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