APPLE II® BASIC

BY DAVID C. GOODFELLOW

73.87221
G651

APPLE II® BASIC

BY DAVID C. GOODFELLOW

8550079

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

DR87/14

FIRST EDITION

SECOND PRINTING

Library of Congress Cataloging in Publication Data

Goodfellow, David C.
Apple II BASIC.

Includes index.
1. Basic (Computer program language) I. Title.
II. Title: Bootstrap B.A.S.I.C.
QA76.73.B3G65 1983 001.64'24 82-19300
ISBN 0-8306-0113-9
ISBN 0-8306-1513-X (pbk.)

Introduction

I think I must have read most of the many excellent books on BASIC available today. From these books I learned the functions of GOTO, GOSUB, If-Then, Print, Clear, and so forth, but they never did teach me how to use these statements to make the computer do a useful task. I suspect those books assumed I had the imagination to take it from there. I didn't.

The problem with books written by experts for beginners is that by the time the author becomes a true expert, he often has forgotten what it is like to be a beginner—and therefore assumes his reader will understand his directions immediately. Since I have a way to go before I become an expert, I hope I have not made this mistake.

This book is about applications—the how-to of hard-copy formatting, disk operations, accurate number round off, operator-oriented displays, etc. It's intended to be a tool for the newcomer to microcomputing, who knows what the command statements are but doesn't know what to do with them. I call it "bootstrap BASIC" because I hope it will help you pull yourself up by the bootstraps. I hope, even the expert will find something of value here.

I sometimes buy a book with the subconscious idea that just by reading it I will automatically learn whatever information the author has put in it. If only it were true! It takes study. It takes practice. It takes work. I learned more about BASIC while writing this book than I had ever learned previously, because I had to organize my information and test the results. If you *really* want to learn a subject, write a book about it. If you don't have the time, study this one. If you work with it, this book will do its job.

An important part of this book is its library routines. Since every program I write uses some, if not most, of these routines, I keep them in a single package on disk. My first step when coding a program is to transfer this library to memory. This single step can accomplish half the coding for the new program!

The library is used to build a name and address program called Bootstrap BASIC, which lets you store and find (of all things) names and addresses. This program can be expanded or modified to keep recipes, inventories, or whatever you like. It could be expanded further to keep clients and charges in two separate files with one common element (like a project number) for billing.

Although library routines appear throughout the book as examples, the complete library is reproduced as a listing in Appendix A. If you plan to copy the listing, do so from the Appendix, as it has not been subjected to retyping and typesetting, and should be free from error.

Appendix B is a complete listing of the sample program. Appendix C contains listings of programs which convert sequential text files to random access text files, and vice-versa. Appendix D contains listings of all complete programs used as examples throughout the book.

Many areas of the book are built upon information provided earlier. Therefore, for best results read it through and do the exercises in order.

The book is written with the Apple II/Applesoft ROM in mind, but the routines will run in any BASIC with slight modification.*

*Take that statement and add salt to taste. Some programmers will find "slight modification" to mean just that. Others will find it means hours of frustration. The difference is in how far apart the two dialects of BASIC are, and the programmer's knowledge of and competence in the two.

Contents

Chapter 1

Ground Rules

Some things are so fundamental to programming that no one ever talks about them. The result is that every newcomer must learn them by trial and error. Eventually each beginner does learn, but it is so *slow*! Get a head start by taking this chapter to heart. Absorb it. Live with it. Always make it a part of your programming projects. You'll save yourself hours of work, and the project will be much more satisfying to you and to whoever uses your program.

DEVELOP A SUBROUTINE LIBRARY

If you do a lot of programming (and you will), you'll find that many of the subroutines you develop will be used again in other programs. There's no point in reinventing a subroutine, or even rekeyboarding one, for a new program.

A real timesaver is to write a library program—one that does nothing itself, but contains every subroutine you have which could be used in more than one program. A library program is given in Appendix A.

When you first start a programming project, load the library program and build on it for your new program. This practice not only eliminates much of your keyboard typing, but it also standardizes the line numbers used for particular functions.

For example, you'll soon know by habit that a **GOSUB 180** command in any of your programs will turn on the printer. Not only will your programming become easier, but debugging will as well. The similarities among your programs, even when they are designed for different tasks, will allow you to interpret their listings with ease.

When the program is finished, delete any library subroutines that are not needed to speed up operation and reduce the amount of memory required to run the program.

DEFINE THE PROJECT

The more complex a program is, the more difficult it is to write and debug, because as your programming progresses it becomes harder to keep track of what must be, and has been, done. The solution is to define the whole project in terms of a number of simple tasks. This does two things for you: first, it gives you a clear picture of the project; second, and equally important, it gives you a psychological advantage. Although the project seen in total may be intimidating, each miniproject within it will be small enough for you to tackle with confidence.

Use flowcharts. Make one for the overall project to identify major tasks. Study it, and make sure it addresses the whole project. From that flowchart, list all the smaller tasks. Assign each a name, and make a flowchart for each. You'll find a discussion of flowcharts in Chapter 2, and examples throughout the book.

Identify the routines that will be used more than once and turn them into subroutines. If they look as if they could be used in other programs, add them to your library. There's no point in rewriting a job each time it must be done when you can simply tell the computer to do it again! I can't speak for your computer, but my computer loves the drudgery of repeating the same job. No imagination whatsoever.

Put the remaining routines in order of performance. Write and test each before going on to the next. This practice will simplify debugging the program, for when the job is done each element will have already been debugged. The only room left for bugs is in the interaction between the elements.

A few gems of wisdom: At the end of a subroutine, clear all variables unless their values must be used elsewhere. If you don't, the values may pop up elsewhere in the program, discombobulating your homeostasis—a very discomforting situation.

Define each variable before you begin, and keep a list. Try to make the variable name reflect its definition. For example, date could be DT$. You'll recognize it more easily in the program listing. If available memory or speed of operation is not a problem, record these variables and their definitions in remark (REM) statements at the end of the program. Keep them in a master program, but strip them from a working copy for speed of operation.

Assign the lowest line numbers to the subroutines used most often. Since the computer searches for line numbers by beginning at the lowest line number and working up in its search for the selected subroutine, placing these early in the program will add to its speed of operation.

PROGRAM FOR OTHER USERS

My early programs were written for me. What need did I have of well-conceived prompts? I wrote it, didn't I? I don't need to have the computer's input needs spelled out, do I?

You bet I do! Some of my early programs required operator retraining after every coffee break. Now, my programs not only tell me when they're waiting for my input, but they also tell me what kind of input they need—and they squawk if I feed them the wrong kind of data!

The Apple II has three excellent ways of getting your attention: reverse video, flashing video, and the audio beep. I use all three.

Reverse video separates computer prompts from my replies, allowing me to tell which is which on a cluttered screen. I need all the help I can get, and I suspect others do, too.

I use flashing statements for prompting when a stupid response on my part could do bad things—such as sort on the wrong field.

I add a beep if these bad things could be disastrous (such as deleting the wrong record), and five beeps if I ask for information that isn't there (usually because I deleted the wrong record anyway).

Let's face it. The computer's a marvelous device, but it has two attributes that make it dangerous: it's fast and it's dumb. Within limits, it will do whatever you tell it to do, including destroying all your files. Therefore, you *must* include traps in your programming that make the computer say, "Whoa! Are you sure that's what you want to do?" That's why sound effects and reverse and flashing video are such valuable tools.

The HOME (in Applesoft) and CALL —936 commands are terrific for keeping the screen uncluttered, but use them sparingly! Clear the screen only when you have no further use for any of the information on it. Obvious, you say? Not really.

A hard decision to make is when are you through with the information. As long as any information on the screen can be used as reference to your next step, leave it up there. But get rid of it when it's just taking up space.

Provide for editing your input. Program to reject or challenge input outside reasonable parameters. For example, format dates so that the number representing a month cannot exceed 12 without the computer rejecting it; likewise, have the computer reject your input for the day of the month if the number representing it exceeds 31.

Obviously bad data should be rejected by the computer. Questionable input should be challenged but not necessarily rejected. The number representing the year (for example) should be challenged if it is less than 1970 or more than 2000, if only the years within these boundaries are considered reasonable or normal input. All a challenge does is point out that this is an unusual input and asks you to confirm it before continuing.

When a block of data has been input, call it back for verification, with each item on a separate, numbered line so that if it is in error you can identify and correct it by line number. A Y response to the question: IS THIS CORRECT? will record it; an N will allow you to change it by line number.

If you ever write a program for someone else, remember that others may not be so enamored (not yet enlightened) with computers. So give your Apple some manners. Keep it polite, even subservient: no insults.

LEAVE YOUR OPTIONS OPEN

We've all heard of the man who built a boat in his basement and then couldn't get it out, and his brother who painted himself into a corner. They're called psychoceramics (crackpots). In programming, it's so easy to do you may count yourself lucky if it doesn't happen! If it does, don't worry about it. Another name for crackpot is "visionary."

Ways to avoid that problem are to treat each task as a separate routine, calling that routine from the master program, and avoiding consecutive line numbers. (Assign line numbers by tens, so you'll be able to insert statements that you'll inevitably forget in your original coding.)

But, if it happens that you've closed the door on a routine by putting it in the master program, you can always turn it into a sometimes subroutine by installing a programming switch. With the switch off, it's just a humdrum little statement. But when you turn the switch on, it steps into a phonebooth and . . . more on this later.

FORMAT YOUR PROGRAM LISTINGS

The colon has three functions in programming, the most important of which is to allow multiple statements in a single program line. The other two functions are strictly cosmetic but can be important tools.

Have you ever searched a long program listing for the first line of a particular program section? If that line is buried within a mass of single-spaced lines, it's hard to find. However, following a line number with a colon and *nothing else* gives the effect of double spacing, and the line sticks out like a sore thumb.

The best way to use this is to end a program section with one or two line-number/colon combinations and begin the next section with a REM statement, as shown below:

```
10 ::::::::::::::::::::REM ***BOOTSTRAP BASIC***

20 REM NAME/ADDRESS PROGRAM

30 :

40 :

50 REM ***CONFIGURATION***

60 DIM A$(50): DIM B$(50)

70 D$ = CHR$(4)
```

Line 10 shows the other cosmetic use of the colon. Here the colon string indents the statement, allowing such things as for-next loops (or anything else you wish to highlight) to stand out.

Although both uses of the colon eat memory and slow down execution, rarely will either be noticeable, and highlighting program areas this way will make them easier to find in the listing.

WRAPUP

All these things are easily done. You'll find further discussions of them, with examples, in later chapters. Subroutines and programs follow, with explanations. Note that there's probably a better way to do everything shown here, but these work!

181 - 57.

Chapter 2
Flowcharts

Chapter 1 discussed the importance of flowcharts. Boaters and flyers use charts; programmers use flowcharts. We can get along without charts and flowcharts if we can see where we're going or don't care, but if we don't know the way and don't want to get lost, we have to have help.

Programmers have some advantages over flyers and boaters. We can make our own maps without knowing the terrain. If we get lost in spite of them, our lives aren't at stake—only our sanity. (Some would say a programmer's sanity is already open to question.)

A flowchart is simply a map of the route you want your program to take. There's nothing difficult about it; no secret skills are required. In fact, simple as flowcharts are, they make the whole programming job immeasureably easier, and result in a more efficient program.

HOW TO MAKE A FLOWCHART

Hotshot programmers seem to think they need a great many symbols to show what their program should do, and this tends to intimidate the rest of us.

Actually, most programs can be flowcharted with five symbols or less:

1. Ovals show the beginning and end of a program or program element.
2. Rectangles indicate processing operations (turn on a device, add, subtract, multiply, divide, etc.).
3. Parallelograms indicate input and output operations.
4. Diamonds indicate decisions (IF-THEN).
5. Arrows show the direction of program flow.

These symbols are not sacred. Some programmers use different symbols to represent specific actions within these functions, and those symbols are also correct. These symbols are widely accepted. If we stray too far from them our flowcharts may confuse another programmer, and we may someday want him to understand what we've done.

The symbols listed above and shown in Fig. 2-1 are those we'll use in this book. The balance of this chapter shows the various flowcharts drawn for the Bootstrap BASIC program used as an example in this book. Figure 2-1 is a sample flowchart showing the symbols we'll be using and has no other significance.

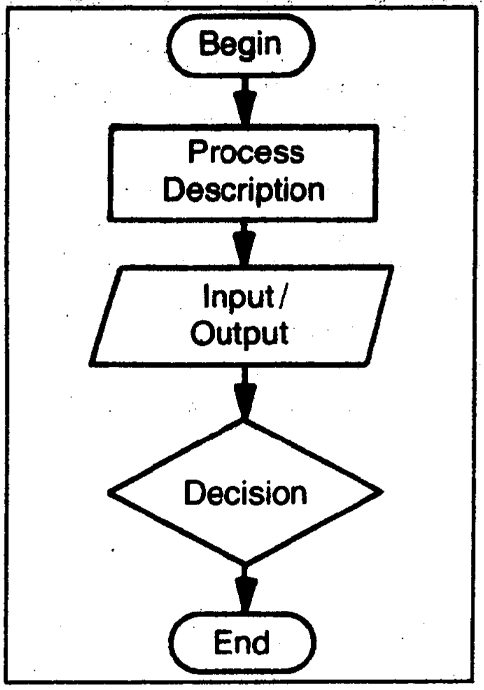

Fig. 2-1. Flowchart symbols.

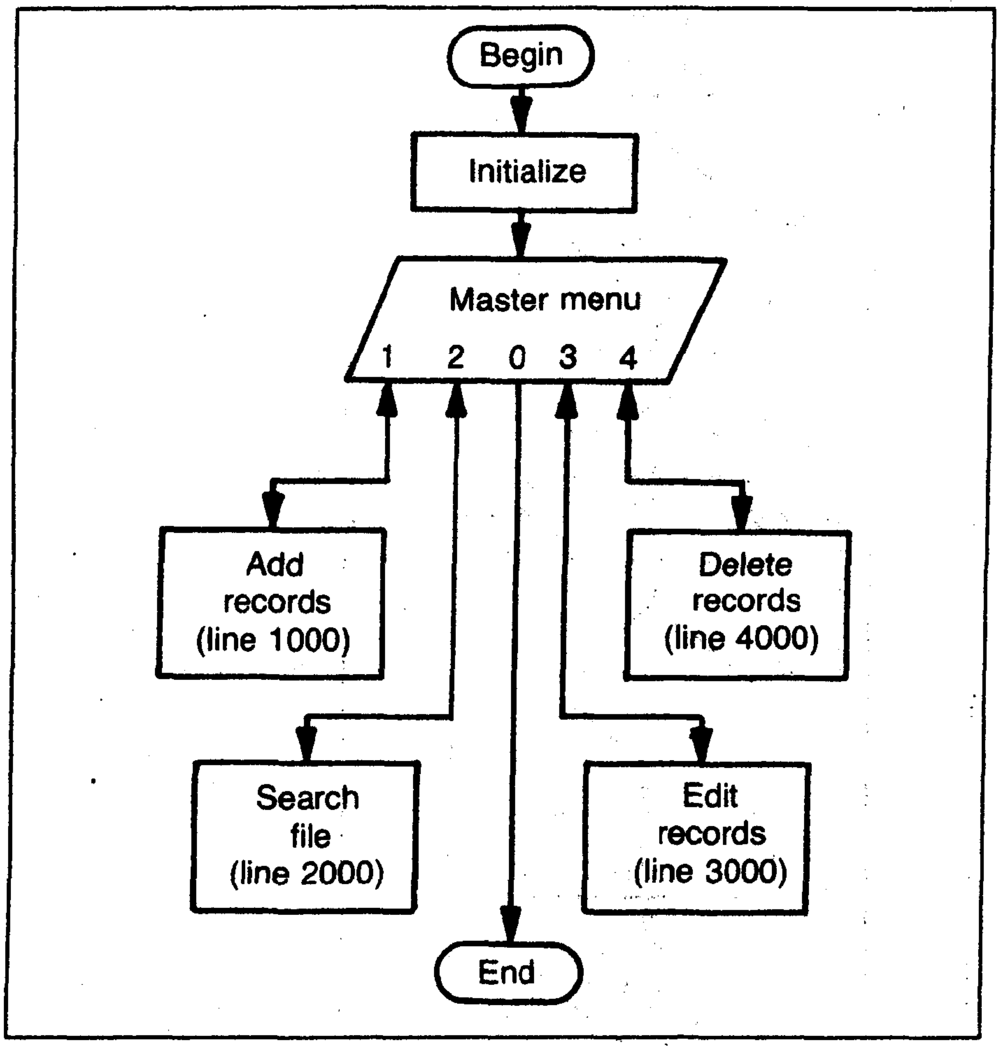

Fig. 2-2. Flowchart, Bootstrap BASIC overall plan.

Figure 2-2 shows the overall plan of the Bootstrap BASIC program in terms of the master menu. Note that the four major program functions exit back to the master menu, and that the fifth function (0) exits the program. This makes the master menu the only legitimate exit from the program, allowing any required housekeeping to be done before you put the disk away.

Figure 2-3 shows the ADD function. The program will ask the operator if he wishes to unlock the file. This is represented by the first diamond. Of course,