ALLEN B. TUCKER, JR.

BASIC/ apple IIe

A PROGRAMMING GUIDE

BASIC/Apple IIe

A Programming Guide

Allen B. Tucker, Jr.
Associate Professor of Computer Science
Georgetown University

Holt, Rinehart and Winston
New York Chicago San Francisco Philadelphia
Montrial Toronto London Sydney Tokyo
Mexico City Rio de Janeiro Madrid

This book is a tutorial guide to BASIC for the Apple *IIe* and not a formal specification of the software as delivered to the buyer now or in the future software revisions. Apple Computer, Inc. makes no warranties with respect to this book or to its accuracy in describing any version of the Apple BASIC software product.

The photographs on the cover and page 2 are courtesy of Apple Computer, Inc.

Address correspondence to:
383 Madison Avenue, New York, NY 10017

Library of Congress Cataloging in Publication Data

Tucker, Allen B.
BASIC/Apple *IIe*.

(Apple programming series)
Includes index.
1. Apple II (Computer)—Programming. 2. Basic (Computer program language) I. Title. II. Series.
QA76.8.A662'83 1983 001.64'2 83–4359

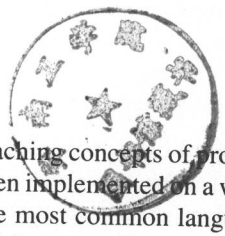
ISBN 0-03-063747-3

Printed in the United States of America
Published simultaneously in Canada

3 4 5 6 039 9 8 7 6 5 4 3 2 1

CBS COLLEGE PUBLISHING
Holt, Rinehart and Winston
The Dryden Press
Saunders College Publishing

Preface



BASIC has, for many years, been a primary tool for teaching concepts of programming. It was designed specifically for this purpose and has been implemented on a wide variety of mini- and microcomputers. Although BASIC is the most common language in this area of computing, it has many dialects. That is, the BASIC language which is available for the Apple *IIe* computer has some important features that distinguish it from, say, the BASIC language which is available for the PDP-11 computer.

The purpose of this book is to provide a unified introduction to programming and BASIC, from the viewpoint of the Apple *IIe* BASIC system, also known as "Applesoft" and running under DOS version 3.3. This particular system has become very popular in recent years with the Apple II, and now the same system can be used on the Apple *IIe*. We hope that this book will reach the wide variety of students and others who use the Apple *IIe* in their studies or professions. This book encourages self-paced learning and frequent reinforcement of concepts by hands-on programming lab exercises.

These lab exercises are grouped into three different subject areas. which we call "business," "math/science," and "general." They are organized so that the reader may consolidate his or her understanding of any particular programming concept by choosing a lab exercise in a subject area appropriate to his or her own interests. For instance, LAB05 tests the reader's mastery of elementary loops. The reader may choose among LAB05B (the "business" problem), LAB05M (the "math/science" problem), or LAB05G (the "general" problem) to demonstrate that mastery. LAB05B asks for a

program that prints an interest payment table, LAB05M asks for a program that prints a list of factorials, and LAB05G asks for a program that averages *m* numbers. All three require an elementary loop.

This book also reflects our firm belief that a language is best taught by first introducing a subset that will allow the reader to solve several elementary problems and master basic programming concepts. We have defined such a subset for BASIC and dubbed it SSB, for "Six Statement BASIC." The six statements of SSB are fully introduced and illustrated in Chapter 2 and then are summarized in Appendix G for easy reference.

We also believe that any complete introduction to programming should provide a selection of programming problems and topics that truly reflects the variety of ways in which computers are being used today. To satisfy this requirement, we have collected about 60 different programming lab problems (20 in each of the aforementioned subject areas). Of these, LAB01 through LAB06 test essential programming skills and the features of the SSB subset language. LAB07 through LAB20 are individually associated with the following topics:

Arrays
Subprograms
Formatted input/output and graphics
Cross-tabulation and statistics
Simulation of board games
File processing
Natural language text processing

Thus, the lab problems provide a most varied and representative selection of computer applications for the uninitiated reader. This broad selection also serves to demonstrate the versatility of Apple *IIe* BASIC as a programming language.

The individual chapters should be covered in order. There is sufficient material here for a one-semester introductory course in programming. LAB01 through LAB06 should be done while covering Chapter 2, while the rest are keyed to Chapters 4 through 9 in the following manner:

Chapter	*LAB*
4. Simple Arrays	07–09
5. Subroutines and Functions	10–12
6. Input/Output Options and Graphics	13, 14
7. Multidimensional Arrays	15, 16
8. File Handling	17, 18
9. Character Strings and Their Uses	19, 20

There are also exercises, for drill and practice with different elements of BASIC syntax and program tracing, at the end of each chapter. Answers to many of these

exercises are found in Appendix F. No answers are provided for the LAB problems themselves, although helpful hints are given with some.

Most texts which introduce BASIC tend to avoid any hardware-specific details, leaving the readers to learn for themselves the extensions and idiosyncrasies of a particular BASIC system. We take a different point of view on this matter. To properly introduce program development, which is the main subject of Chapters 1 and 3, we cannot avoid system-dependent features, such as "how to change a line of program text." Thus, we have integrated all aspects of creating and modifying a BASIC program *on the Apple IIe computer* into our discussions of program development. Similarly, we have included topics like "how to implement random access files" in the discussion of file processing in Chapter 8 so that the reader will appreciate this very important programming technique and see how it is done in Apple *IIe* BASIC. In this, our reliance on peculiarities of the Apple *IIe* is unavoidable. Thus, unlike most texts, this one is strongly committed to the Apple *IIe* version of BASIC. Readers who might use this text with other computers should thus beware of the differences.

We assume without further mention that the reader knows how to mount the proper diskettes (usually the DOS 3.3 System Master in drive 1) and turn the Apple *IIe* power on and off. These operations, as well as the various utility programs that appear on the System Master are easily understood and will not be belabored herein.

A few words are in order about the Apple *IIe* itself. It is, generally, an extension and refinement of the very popular Apple II computer. In terms of BASIC programming, the Apple *IIe* is "upward compatable," that is, Apple II BASIC programs generally run on the Apple *IIe* without modification. The main difference that the experienced Apple II programmer must remember is that the <CAPS LOCK> key on the Apple *IIe* is down at all times. The Apple *IIe* has a full upper and lowercase keyboard, but BASIC does not accept lowercase letters in its programs.

There are many other differences between the Apple *IIe* and the Apple II. For instance, the optional 80-column screen (which is available from Apple) combined with the full keyboard, makes the Apple *IIe* extremely attractive for handling one's word processing needs. A full description of the differences between the Apple *IIe* and the Apple II is given in Appendix J.

Finally, I am grateful to those who have supported this book's development. To Apple Computer, Inc., for the opportunity to work with an Apple *IIe* prototype model. To Brete Harrison and Holt, Rinehart and Winston, for giving me the opportunity to do this project. To Donna Tomlinson, for typing and proofreading the manuscript with dedication, speed, and precision under unusually demanding production schedules. To the reviewers, for helping to improve earlier versions of this manuscript. To my students at Georgetown who continually revitalize the teaching and learning methodology that motivated the book in the first place. To my family—Maida, Jenny, and Brian—for their continuing love and durability through these many textbook projects.

Allen B. Tucker, Jr.

Contents

CHAPTER 1

The Computing Process

Computers have a "static" aspect and a "dynamic" aspect. The static aspect consists of the components, whereas the dynamic aspect represents the actual movement and manipulation of the data that occurs when the components are activated. Both the static and the dynamic aspects of computers must be clearly understood in order to master the art of programming itself.

1.1 Apple *IIe* Computer Organization

We first describe the static aspect of computers, which is known as *computer organization*. Five general components comprise the organization of a computer, as shown in Figure 1.1. In the center is the computer's *central processing unit* (CPU). Leading into the CPU from the left is the *input,* and leading out to the right is the *output*. The CPU itself has three parts, the *memory,* the *control,* and the *arithmetic/logic* circuitry.

The arrows that connect these components denote paths through which information can flow. That is, information can flow from the input to the memory, from the memory to the output, and in either direction between memory, control, and arithmetic/logic.

Figure 1.2 shows a picture of an Apple *IIe* computer, with its five basic components identified. Here, the reader can get an idea of what these components actually look like.

The *memory* of a computer holds two kinds of information, the *program* and some *data*. The *program* itself consists of a series of instructions that tells exactly what steps to

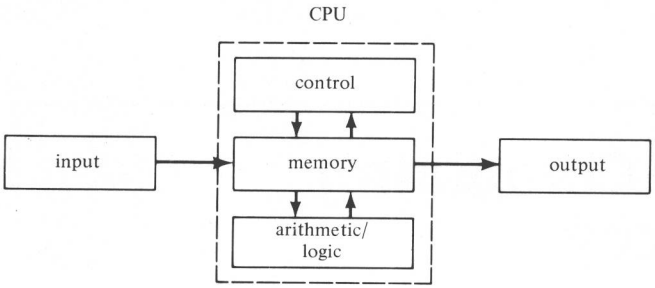


Figure 1.1
The components of a computer.

perform. These instructions are actually carried out, or "executed," by the *control unit*. Some of the instructions tell the control to transfer information from the input to the memory; this is known as a *read* operation. Others tell the control to transfer information from the memory to the output; this is known as a *write* operation. Still others tell the control to perform an arithmetic operation (e.g., addition) or a comparison of two data values (e.g., to see which is greater). These kinds of operations are actually carried out by the *arithmetic/logic* part of the CPU.

Now, the actual *input* and *output* information can be represented in any of several different "computer-readable" media, including punched cards, an interactive terminal,

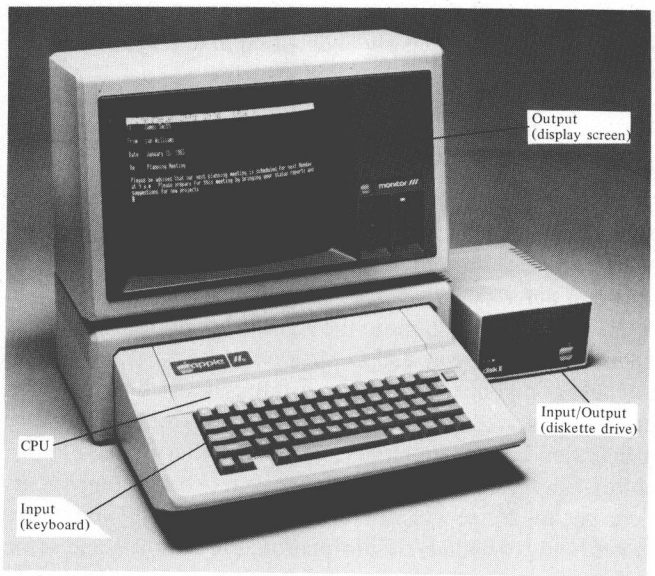


Figure 1.2
Components of an Apple *IIe* computer.

video display, printed paper, magnetic tape, and magnetic disk. Shown with the Apple *IIe* in Figure 1.2 are two magnetic disk units, an interactive terminal, video display, and a line printer. Each magnetic disk unit holds one cartridge, called a *diskette,* which may contain programs and data (see Figure 1.3). Diskettes may be interchangeably mounted on the disk drive, but at any one time only one diskette may be present.

1.2 Programs

The *program* is a sequence of instructions which, when executed by the control unit, defines exactly what should be done with the input data in order to produce a particular output. That is, the program specifies the "dynamic" aspect of the computer. Without the program, the components described in Section 1.1 would be just a passive collection of hardware.

Functionally, the program is like a recipe; followed precisely, the recipe will yield the desired result. Yet, a program must be precisely and, sometimes, excruciatingly specified in order to fully define the task to be performed. Also, like a recipe, the program must be written in a very exact syntactic form in order to be understood and properly executed. This form is known as the *programming language.*

There are many different programming languages in use today, such as ALGOL, COBOL, FORTRAN, PL/I, APL, BASIC, LISP, and Pascal. Each has it special strengths in one of the wide variety of application areas where programmers are working. In this book, we teach the programming language BASIC because it is widely known, permits good programming style, is easily taught and learned, and can be effectively used in the various programming situations that occur in mathematics, science, business, the humanities, government, and personal computing.

Because BASIC has many different kinds of statements (see Appendix B), we shall first teach an elementary part of it, so that the reader may master simple programming

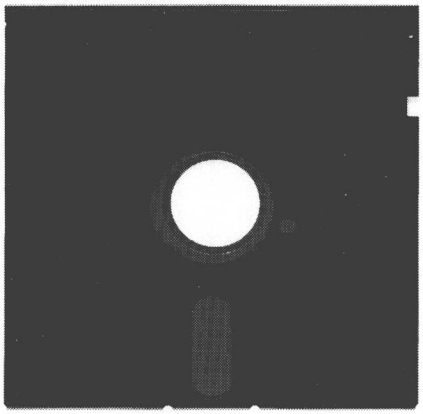


Figure 1.3
A diskette storage cartridge.

techniques before proceeding to advanced material. We have dubbed that elementary part as "Six Statement BASIC," or SSB for short, which is the subject of Chapter 2. Additional BASIC features will be described, illustrated, and exercised in later chapters.

1.3 Program Development: An Overview

Although we teach the programming language BASIC, we have a far more important purpose in this book, that is, to introduce and teach the elements of *program development*. It is one thing to follow a recipe successfully and end up with an edible cake, but it is quite another to design and correctly describe the recipe in the first place.

More precisely, *program development* has as its purpose to design and demonstrate the correct functioning of a (BASIC) program that carries out a prescribed task. Examples of typical "prescribed tasks" are the following:

1. Add two numbers and display the resulting sum, given the original two numbers.
2. Compute the average of all three tests taken by each student in a class of 25, given the original 75 individual test scores (three per student).
3. Translate a text from Spanish into English, given the original Spanish text.

As the reader can see, these examples range in difficulty from trivial to complex. This is the domain of program development. In this book, most of the program development tasks are like that of Example 2, not trivial but achievable in a reasonable amount of time.

We prescribe these tasks as so-called labs, numbered LAB00 through LAB20. LAB00 will be presented, programmed, and discussed in its entirety in this chapter; in fact, LAB00 *is* Example 1 given above. The labs are organized into three general subject-area groups: business, math/science, and general. Readers are encouraged to select labs that correspond with their subject-area interests. Labs are coordinated so that, for instance, LAB13B (that is, a business task) and LAB13M (that is, a math/science task) exercise the *same* BASIC features and program development techniques. (The suffix B, M, or G affixed to the LAB number identifies its subject area as business, math/science, or general, respectively.)

Returning to the question of *program development,* this process can be subdivided into the following sequence of distinct steps:

1. Problem specification
2. Algorithm design
3. Program coding
4. Program preparation
5. Program execution
6. Program diagnosis and error correction

The following paragraphs describe each of these steps, using Example 1 for illustration.

Problem Specification

A clear and concise statement of the programming problem to be solved is, of course, a prerequisite to the development of the program itself. Recall the problem statement of Example 1:

1. Add two numbers together and display the resulting sum, given the original two numbers.

There is a kind of innate tedium in any such problem statement, which is due to the requirements for precision and completeness. The statement must always be reflective of the general capabilities and limitations of computers and programming. Moreover, the problem statement must be totally clear and fully comprehensible to the person who will write the program.

In this book, the programming problem statements are already developed, in the form of LAB00 through LAB20. Our purpose here is to teach programming and problem-solving skills, rather than to teach the development of problem statements themselves. The area of computing in which problem statements are developed is known as *systems analysis and design*.

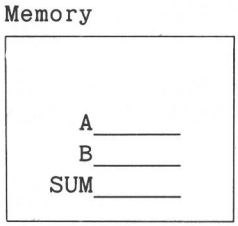
One element of a good problem statement is that it not only portrays the programming task to be performed (e.g., "add two numbers" in Example 1) but also identifies the input data (e.g., "given the original two numbers") and the desired output (e.g., "display the resulting sum").

Algorithm Design

Here, the programmer translates the problem statement into a precise description of how the computer program will solve the problem. In general, algorithm* design begins with a sketch, in English, of the sequence of steps that the computer should follow to solve the problem. At this point, the programmer identifies all memory locations, known as *variables,* that are necessary for the program to perform properly. A memory location can be visualized as a place within the computer's memory which can hold a single data value, such as a number or an alphabetic character. A variable can be visualized as a memory location which is associated (by the program) with a unique name, such as A or SUM. For instance, an algorithm design for Example 1 can be given as follows:

1. Identify A and B as the variables which will contain the two numbers to be added, and SUM as the variable which will contain their sum, as shown in the following picture of memory:

*The term *algorithm* means "a precise description of a computing task which will terminate in a finite number of steps." That description can be done in any suitable language, such as English or any programming language (e.g., Pascal, FORTRAN, COBOL, PL/I, or BASIC). When done in a programming language, the algorithm is known as a *program*.



2. The sequence of steps required to solve this problem are:
   a. Transfer the two numbers from the input to variables A and B, respectively.
   b. Add the values of A and B, leaving the result in SUM.
   c. Transfer the value of SUM to the output.

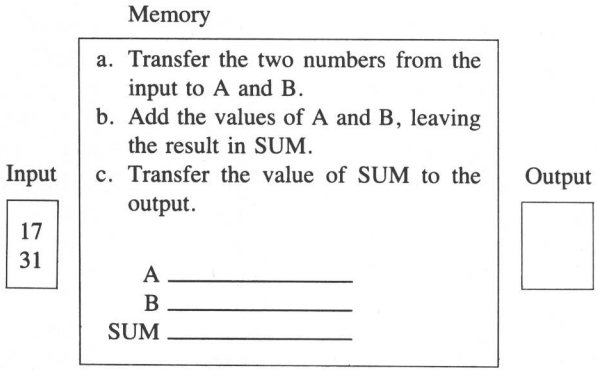
An algorithm design always presumes certain overall operational or mechanical characteristics of computer programs:

The input values must be brought into specific memory locations, or variables, before any arithmetic or other operations can be performed with them.

The computer's control unit carries out the individual steps of a program in the order in which they are written.

The result(s) of a program must be transferred from memory to the output in order for it to be displayed. The memory is an electronic medium, hidden from view. The input and output (e.g., a terminal screen or a printer) provide visible representations for data values and results.

To illustrate, the following diagram shows a complete configuration of input data, program, variables, and output immediately *before* steps a, b, and c of the algorithm are carried out by the computer's control unit:



Here, the algorithm is given 17 and 31 as two sample input values. In general, an algorithm is designed to handle *any* input that is suitable to the problem statement (e.g., any pair of numbers, in the case of Example 1).

After the control has carried out all three steps specified by the algorithm, the resulting configuration will be as shown below: