IBM PCjr

Introduction, BASIC Programming and Applications

Larry Joel Goldstein

Brady

$1495
40515

IBM PCjr®: Introduction, BASIC Programming and Applications

Larry Joel Goldstein

published by
Robert J. Brady Co. • Bowie, Maryland
A Prentice-Hall Publishing and Communications Company

IBM PCjr: Introduction, BASIC Programming and Applications

Copyright © 1984 by Robert J. Brady Co.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Robert J. Brady Co., Bowie, Maryland 20715.

Library of Congress Cataloging in Publication Data
Goldstein, Larry Joel.
IBM PCjr.

Includes index.
1. IBM PCjr (Computer) 2. IBM PCjr (Computer)—Programming. 3 Basic (Computer program language) I. Title. II. Title: I.B.M. PCjr. III. Title: IBM PC jr.
QA76.8.I2593G653 1984 001.64 84-3081
ISBN 0-89303-539-4

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

84 85 86 87 88 89 90 91 92 93 94 10 9 8 7 6 5 4 3 2 1

Publishing Director: David Culverwell
Acquisitions Editor: Leslie Ehrin
Production Editor/Text Designer: Karen A. Zack
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa
Cover Photo: George Dodson

Copy Editor: Elyse Finger
Typesetter: Alexander Typesetting, Inc., Indianapolis, IN
Typefaces: Souvenir (text), OCR-B (programs)
Printer: Fairfield Graphics, Fairfield, PA

Contents

PART I—
GETTING STARTED WITH YOUR PCjr

1. A First Look at Computers **1**

1.1 Introduction 1
1.2 What is a Computer? 3
1.3 A Grand Tour of Your PCjr 5
1.4 The Keyboard 7
1.5 System Unit 8
1.6 Cassette and Diskette Storage 15
1.7 Displays 16
1.8 Printers 17
1.9 Communications 18
1.10 The Serial Port 19

2. Using Your PCjr For the First Time **21**

2.1 Starting Your PCjr (Without DOS) 21
2.2 Diskettes and Diskette Drives 22
2.3 Starting Your PCjr (With DOS) 24
2.4 More About Diskette Drives 28
2.5 Backing Up Your DOS Diskette 28
2.6 The Keyboard 31

3. An Introduction to DOS **37**

3.1 Files and File Names 37
3.2 File Specifications 39
3.3 Executing Commands and Programs 42
3.4 The COPY Command 44
3.5 COPYing and FORMATing Diskettes 46
3.6 Other DOS Internal Commands 50
3.7 Other DOS External Commands 52
3.8 Creating Your Own DOS Commands-Batch Files 53

PART II—
AN INTRODUCTION TO PCjr BASIC

4. Getting Started in BASIC **59**

4.1 Beginning BASIC 59
4.2 BASIC Statements in Immediate Mode 59

4.3 BASIC Constants and Arithmetic 62
4.4 Running BASIC Programs 70
4.5 Writing BASIC Programs 71
4.6 Giving Names to Numbers and Words 77
4.7 Some BASIC Commands 86
4.8 Some Programming Tips 94
4.9 Using the BASIC Editor 95

5. Controlling the Flow of Your Program 99

5.1 Doing Repetitive Operations 99
5.2 Letting Your Computer Make Decisions 110
5.3 Structuring Solutions to Problems 125
5.4 Subroutines 127

6. Working With Data 135

6.1 Working With Tabular Data 135
6.2 Inputting Data 143
6.3 Formatting Your Output 149
6.4 Gambling With Your Computer 158

7. Easing Programming Frustrations 167

7.1 Flow Charting 167
7.2 Errors and Debugging 170
7.3 Some Common Error Messages 174
7.4 Further Debugging Hints 175

8. Your Computer as a File Cabinet 179

8.1 What Are Files? 179
8.2 Sequential Files 180
8.3 More About Sequential Files 188
8.4 Random Access Files 191
8.5 An Application of Random Access Files 199
8.6 Sorting Techniques 204
8.7 BASIC File Commands 209

9. String Manipulation 215

9.1 ASCII Character Codes 215
9.2 Operations on Strings 219

9.3 Control Characters 225

10. An Introduction to Computer Graphics 229

10.1 Line Graphics 229
10.2 Colors and Graphics Modes 237
10.3 Lines, Rectangles, and Circles 243
10.4 Computer Art 254
10.5 Drawing Bar Charts 257
10.6 Drawing Pie Charts 261
10.7 Painting Regions of the Screen 264
10.8 The Graphics Macro Language* 267
10.9 Saving and Recalling Graphics Images 274
10.10 VIEW and WINDOW 278
10.11 Sound and Music on the PCjr 282

11. Word Processing 289

11.1 What is Word Processing? 289
11.2 Using Your Computer as a Word Processor 289
11.3 A Do-It-Yourself Word Processor 291

12. Some Additional Programming Tools 295

12.1 The INKEY$ Function 295
12.2 The Function Keys and Event Trapping 296
12.3 Error Trapping 301
12.4 Chaining Programs 303

13. Computer Games 307

13.1 Telling Time With Your Computer 307
13.2 Blind Target Shoot 312
13.3 Shooting Gallery 316
13.4 TIC TAC TOE 319

14. Different Kinds of Numbers in PC BASIC 325

14.1 Single- and Double-Precision Numbers 325
14.2 Variable Types 330
14.3 Mathematical Functions in BASIC 333
14.4 Defining Your Own Functions 338

*Registered trademark of Microsoft Corporation.

15. Computer Generated Simulations **341**

15.1 Simulation 341
15.2 Simulation of Traffic Through a Computer Store 343

Answers to Selected Exercises 349

Index 377

Documentation for Optional Program Diskette 385

Preface

IBM's PCjr is a computer designed to bring the computer revolution to the home. Able to run many of the same programs as its big brothers, the PC and PC/XT, the PCjr will be used for many purposes, including computer literacy, word processing, accounting, graphics, education, and entertainment. This book is an introduction to the PCjr for persons interested in any of these applications.

I begin the book with a detailed description of the PCjr to show what the various components and connectors are, and how they may be used. Next, we give a fairly detailed description of the disk operating system DOS 2.1. It is necessary to learn this material in order to use many of the disk-based programs that you purchase. Beginning in Chapter 4, we give a fairly comprehensive course on programming in BASIC on the PCjr.

The emphasis in this book is on applications. We have included dozens of programs for you to try. In fact, all the major programs have been collected and are available on the optional diskette. Using this diskette will save you the trouble of keying in the programs.

I have structured this book as a tutorial. The style is relaxed and conversational. My goal has been to construct an introductory course on the PC which would be similar to a course I would present to a small group of students. Accordingly, I have included TEST YOUR UNDERSTANDING questions, which are designed to immediately test your comprehension of the concepts presented. The answers to these questions are located at the end of the section. Most sections end with a detailed exercise set. The answers to many of the exercise questions are located at the back of the book.

My sincere thanks go to all of my readers who have taken the time to communicate with me and to share their ideas, their enthusiasm and their frustration. Many of their ideas and suggestions have found their way into the book.

I owe a debt of gratitude to Martin Goldstein for researching much of the data which ultimately found its way into the book.

I would also like to thank Parker Foley, Richard Freeland, and Beth Merrill of Computerland of Rockville, Maryland, and Bob Zimmerman and Rush Simonson of Computerland of West Palm Beach, Florida, for supplying much useful information about the PCjr and for graciously allowing me to use their facilities to test the programs.

I wish to express my deepest appreciation to all my friends at the Robert J. Brady Co. for their efforts which go well beyond the call. Our special thanks to Karen Zack and Paula Huber for their guidance of the production process, to Jessie Katz for assistance in arranging for reviewers and for handling the flow of correspondence with our readers, to Don Sellers for his imaginative cover design, to Bernard Vervin for his excellent renditions of my sloppy drawings, to George Dodson for his beautiful photography, and to Joan Caldwell, John Allison and Sue Drosdzal for their sales and promotion efforts. Last but not least, I would like to thank my

close friends and associates, David Culverwell, Publishing Director and Harry Gaines, former president, for their support and enthusiastic encouragement.

Dr. Larry Joel Goldstein
Silver Spring, Maryland
March 4, 1984

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Trademarks of material in this text

PCjr, IBM PC, and IBM PC/XT are registered trademarks of International Business Machines Corporation.

Graphics Marco Language and Music Macro Language are registered trademarks of Microsoft Corporation.

Note to Authors

Do you have a manuscript or a software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Co. produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Robert J. Brady Co., Bowie, Maryland 20715.

Dedication

To Sandy
who fills my life with happiness

1

GETTING STARTED WITH YOUR PCjr

1

A FIRST LOOK AT COMPUTERS

1.1 Introduction

The computer age is barely 30 years old, but it has already had a profound effect on all our lives. Indeed, computers are now prevalent in the office, the factory, and even the supermarket. In the last three or four years, the computer has even become commonplace in the home, as people have purchased millions of computer games and millions of personal computers. Computers are so common today that it is hard to imagine even a single day in which a computer will not somehow affect us.

In spite of the explosion of computer use in our society, most people know very little about computers. They view the computer as an "electronic brain" and do not know how a computer works, how it may be used, and how greatly it may simplify various everyday tasks. This does not reflect a lack of interest. Most people realize that computers are here to stay and are interested in finding out how to use them. If you are so inclined, then this book is for you!

This book is an introduction to personal computing for the novice. You may be a student, teacher, homemaker, business person, or other curious individual. I assume that you have had little or no previous exposure to computers and want to learn the fundamentals. I will guide you as you turn on your IBM PCjr for the first time. (There's really nothing to it!) From there, I will lead you through the fundamentals of the disk operating system (DOS—for those of you who have a PCjr equipped with a diskette drive). Then I'll teach you to talk with your computer in the BASIC language. Throughout, I'll provide exercises for you to test your understanding of the material. Together we will explore the many ways *you can use your computer*. The exercises suggest programs you can write. Many of the exercises are designed to give you insight into how computers are used in business, home and industry. For good measure we'll even build a few computer games!

What is personal computing?

In the early days of computing (the 1940s and 1950s), the typical computer was a huge mass of electronic parts that occupied several rooms. In those days, it often was necessary to reinforce the floor of a computer room

and to install special air conditioning so the computer could function properly. Moreover, an early computer was likely to cost several million dollars. Over the years, the cost of computers has decreased dramatically and, thanks to microminiaturization, their size has shrunk even faster.

In the late 1970s, the first "personal" computers were put on the market. These computers were reasonably inexpensive and were designed to allow the average person to learn about the computer and to use it to solve everyday problems. These personal computers proved to be incredibly popular and have stirred the imaginations of people in all walks of life. It is no exaggeration to say that a computer revolution is now underway, as millions of people are learning to fit computers into their everyday lives.

The personal computer is not a toy. It is a genuine computer with most of the features of its big brothers, the so-called "mainframe" computers, which still cost several million dollars. A personal computer can be equipped with enough capacity to handle the accounting and inventory control tasks of most small businesses. It also can perform computations for engineers and scientists, and it can even be used to keep track of home finances and personal clerical chores. It would be quite impossible to give a complete list of the possible applications of personal computers. However, the following list can suggest the range of possibilities:

For the business person—
- Accounting
- Recordkeeping
- Clerical chores
- Inventory
- Cash management
- Payroll
- Graph and chart preparation
- Word processing
- Data analysis
- Networking

For the home—
- Recordkeeping
- Budget management
- Investment analysis
- Correspondence
- Energy conservation
- Home security
- On-line information retrieval
- Tax return preparation

For the student—
- Computer literacy
- Preparation of term papers
- Analysis of experiments
- Preparation of graphs and charts
- Project schedules
- Storage and organization of notes

For the professional—
- Billing
- Data analysis
- Report generation
- Correspondence
- Stock market data access
- Scientific/engineering calculations

For recreation—
- Computer games
- Computer graphics
- Computer art

As you can see, the list is quite extensive. If your interests aren't listed, don't worry! There's plenty of room for those of you who are just plain curious about computers and wish to learn about them as a hobby.

The IBM PCjr*

This book will introduce you to personal computing on IBM's home computer, the IBM PCjr.* This machine is an incredibly sophisticated device that incorporates many of the features of its big brothers, the IBM PC and PC/XT, which currently are being used by millions of people in homes and offices throughout the world. Before we begin to discuss these particular features of the IBM PCjr, let's discuss the features common to all computers.

1.2 What is a Computer?

At the heart of every computer is a **central processing unit** (or **CPU**) that performs the commands you specify. This unit carries out arithmetic, makes

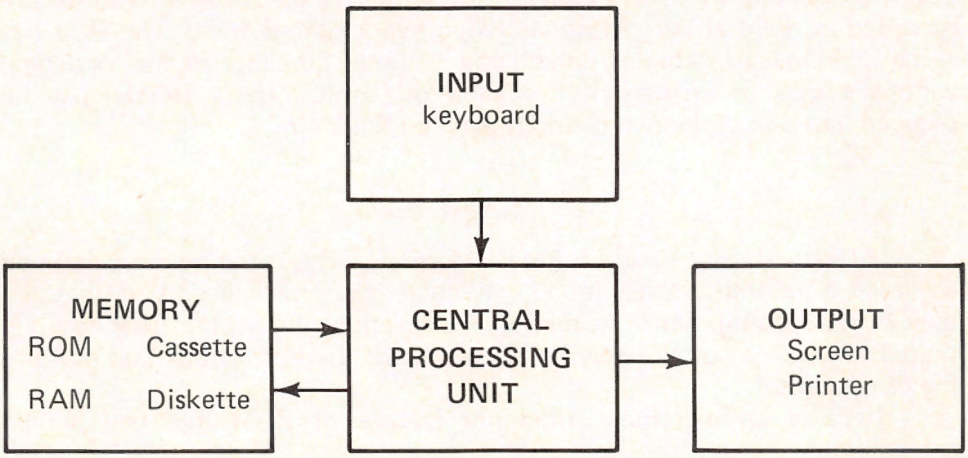


Figure 1-1. The main components of a computer.

* PCjr (pronounced "PC junior") is a registered trademark of International Business Machines Corporation.

logical decisions, and so forth. In essence, the CPU is the "brain" of the computer. The **memory** of a computer allows it to "remember" numbers, words, and paragraphs, as well as the list of commands you wish the computer to perform. An **input unit** allows you to send information to the computer; an **output unit** allows the computer to send information to you. The relationship of these four basic components of a computer is shown in Figure 1-1.

In an IBM PCjr, the CPU is contained in a tiny electronic chip, called an **8088 microprocessor**. As a computer novice, you don't need to know about the electronics of the CPU. For now, view the CPU as a "magic device" somewhere inside the case of your computer and don't give it another thought!

The main input device of the PCjr is the computer keyboard. We will discuss the special features of the keyboard in Section 2.5. For now, think of the keyboard as a typewriter. By typing symbols on the keyboard, you are inputting them to the computer.

The PCjr has a number of output devices. The most basic is the "TV screen" (sometimes called the **video monitor** or **video display**). You may also use a printer to provide output on paper. In computer jargon, printed output is called **hard copy**.

There are four types of memory in a PCjr: **ROM**, **RAM**, **cassette**, and **diskette**. Each of these types of memory has its advantages and disadvantages. Microcomputers attempt to make memory as versatile as possible by using several kinds of memory, thereby allowing them to take advantage of the good features of each.

ROM

ROM stands for "Read Only Memory." The computer can read ROM but cannot write anything in it. ROM is reserved for certain very important programs necessary to the operation of the machine. These programs are recorded in ROM at the factory and you cannot change them. The PCjr has some ROM in its system unit. In addition, you may purchase ROMs containing various games or an enhanced version of BASIC. These ROMs may be plugged into one of the two cartridge slots on the PCjr.

RAM

RAM stands for "Random Access Memory." This is the memory that you can read from and write to. If you type characters on the keyboard, they are then stored in RAM. Similarly, results of calculations are kept in RAM awaiting output to you. As you will see, RAM even holds the instructions that perform the calculations!

There is an extremely important feature of RAM that you should remember:

If the computer is turned off, then RAM is erased.

Therefore, RAM may not be used to store data in permanent form. Nevertheless, it is used as the computer's main working storage because of its great speed. (It takes about a millionth of a second to store or retrieve a piece of data from RAM.)

The size of RAM is measured in **bytes**. Essentially, a byte is a single character (such as "A" or "!"). You often will hear statements like "The PCjr comes with 64K of RAM." The abbreviation "K" stands for the number 1,024. And 64K stands for 64 times 1,024 or 65,536 bytes.

Cassette Recorder

To make permanent copies of programs and data, you can use either a cassette recorder or a diskette drive.

The cassette recorder is just a tape recorder that allows recording of information in a form the computer can understand. The recording tape is the same type you use for musical recordings.

Diskette Drives

A diskette drive (Figure 1-2) records information on flexible diskettes that resemble phonograph records. The diskettes are often called "floppy disks," and they can store several hundred thousand characters each! (A double-spaced typed page contains about 3,000 characters.) (See Figure 1-3.) A diskette drive can provide access to information in much less time, on the average, than a cassette recorder. On the other hand, diskette drives are more costly.

1.3 A Grand Tour of the PCjr

Now that we know the elements of computer systems in general, let's take our first look at IBM's PCjr.

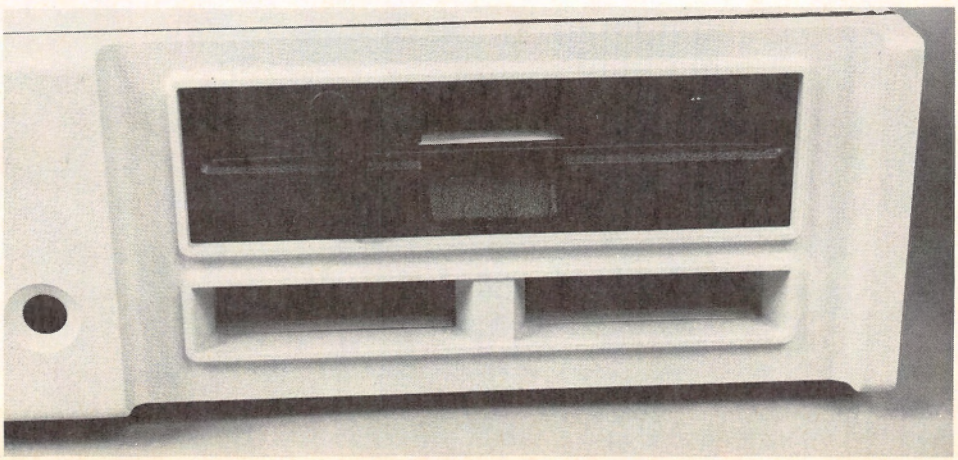


Figure 1-2. A diskette drive.

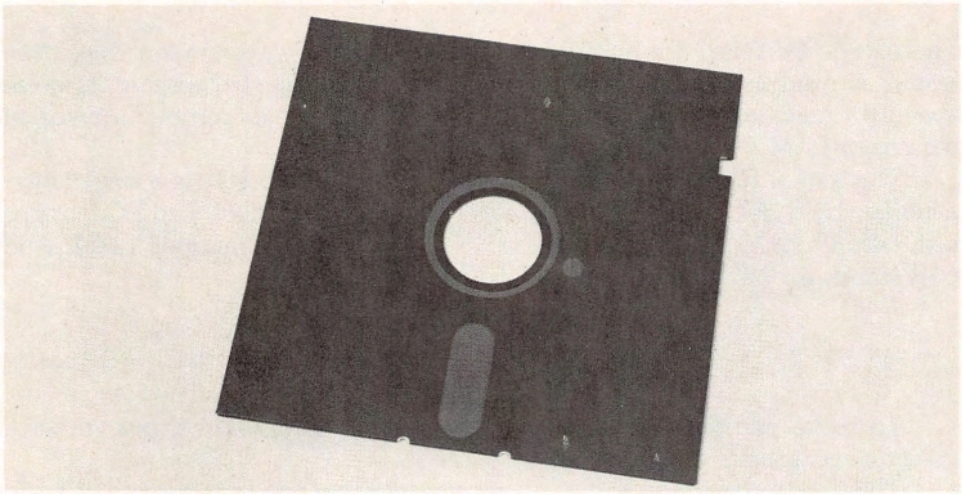


Figure 1-3. A floppy diskette.

The system comes in two versions: basic and enhanced. Let's describe the basic version, shown in Figure 1-4.

Note that there are three component parts to the basic system. The **keyboard** is the unit that looks like a typewriter. The **system unit** is the large box, which contains the 8088 microprocessor and most of the electronics of the

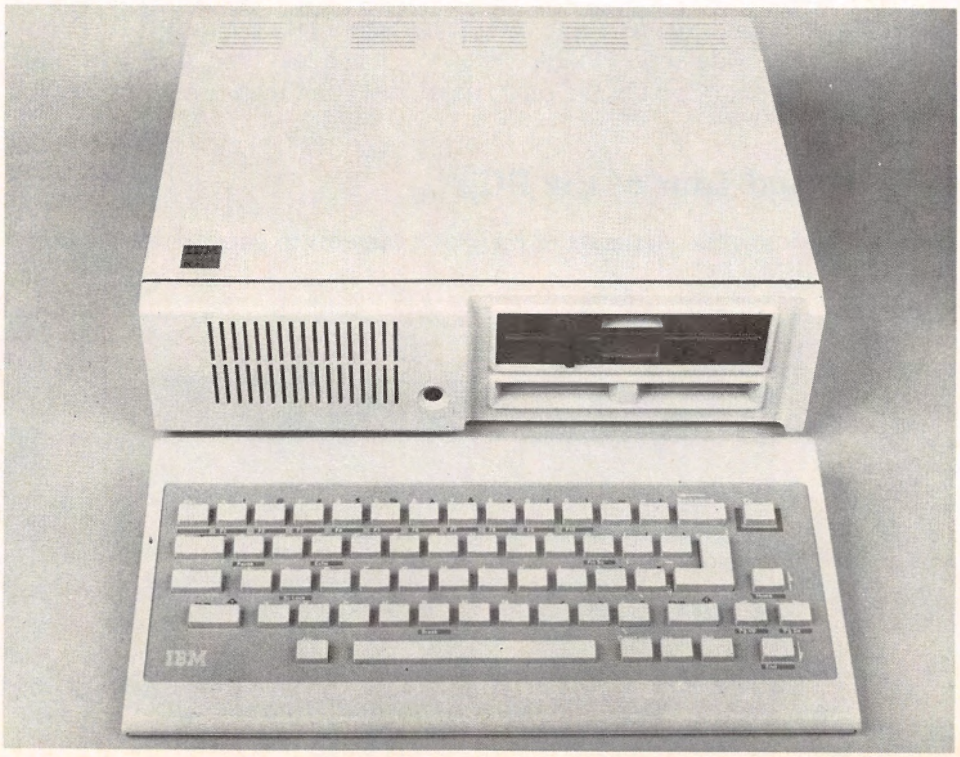


Figure 1-4. The PCjr basic system.

PCjr. Inside the system unit is 64K of RAM. The system unit has two cartridge slots into which you can insert program ROMs.

The other box is the transformer, which converts household electric current to a level the PCjr can use.

The enhanced system is shown in Figure 1-5. Note that this system has a diskette drive in the system unit. (The diskette drive is the long, rectangular door at the bottom of the unit.) In addition, the enhanced system has 128K of RAM which, among other things, allows you to display an 80-character wide line.

I will describe the various PCjr components in greater detail in the next few sections.

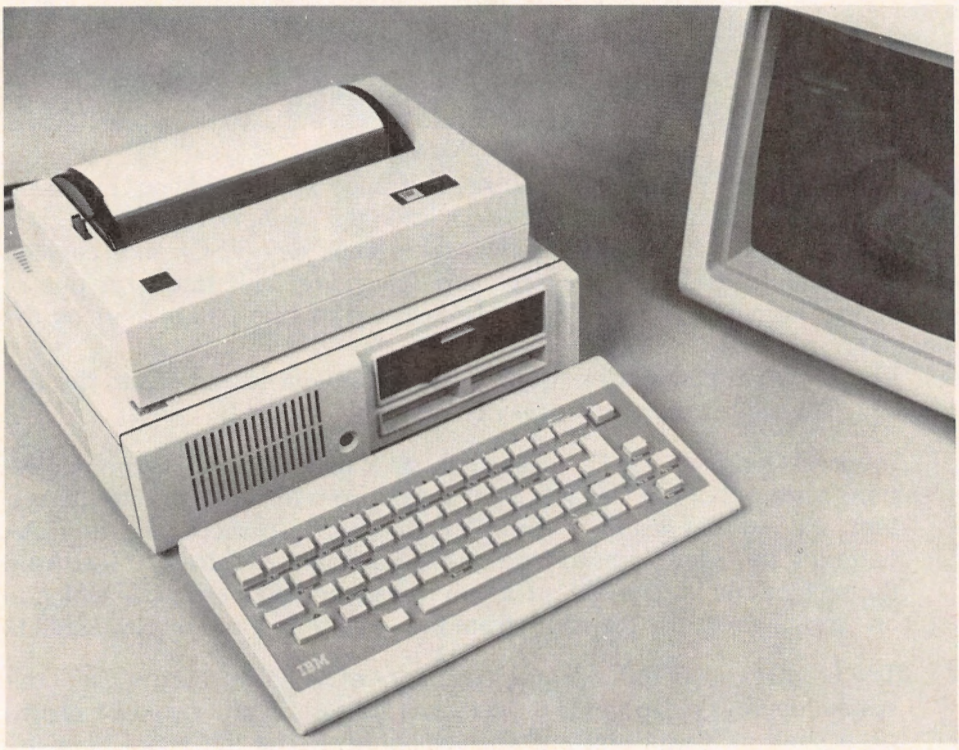


Figure 1-5. An enhanced PCjr system.

1.4 The Keyboard

The keyboard is the device you use to communicate directly with the computer. As you type, your keystrokes are recorded in RAM, awaiting action by the computer. What you type is called **input**.

The PCjr keyboard is very similar to that of a typewriter. However, a number of special keys that are not found on a typewriter keyboard allow it to perform many more functions. The keyboard of the PCjr is shown in Figure 1-6.

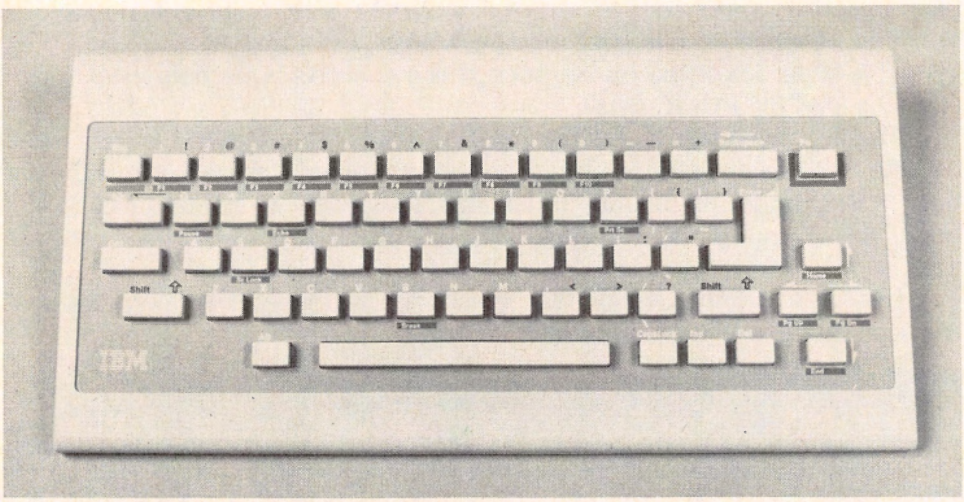


Figure 1-6. The PCjr keyboard.

At first, the PCjr keyboard may seem strange and forbidding. However, you can easily learn about its various keys using the *Keyboard Adventure* tutorial, which is stored in ROM. By hitting the escape key (ESC) once cassette BASIC is started, you will start *Keyboard Adventure*. This tutorial is one of several that comes with the PCjr and is designed to help you in learning the keyboard.

The keyboard contains electronics to transfer your keystrokes to the system unit. Transferring keystrokes may be done in either of two ways:

Infrared Optical Link—The keyboard has a built-in infrared communications system that can transmit information from the keyboard to the system unit without using a cord. This system is powered by four AA batteries and allows you to type on the keyboard as far as 20 feet from the system unit. It is possible for you to use the keyboard while sitting in an easy chair across the room from the system unit and display screen.

The cordless keyboard also gives you a lot of flexibility in arranging your system in crowded rooms. For instance, the system unit could be placed on a shelf in a wall unit and the keyboard across the room near the television.

The infrared optical link has a restriction. It cannot "turn corners." There must be an unobstructed path from the system unit to the keyboard.

Cable Connection—You may connect the keyboard to the system unit via an optional cable. This is advisable in classroom situations or may be necessary where there are several PCjrs in one room.

1.5 System Unit

Of all the components of the PCjr system, the system unit is the most complex. Within its small cabinet is all the ingenious electronic circuitry that makes the

computer work. In this section, you'll get familiar with the system unit, first from the outside, and then on the inside.

Let's begin by looking at the front of the system unit, shown in Figure 1-7.

At the bottom right are two slots into which you may insert program cartridges. Initially, IBM is offering assorted program cartridges containing games. In addition, you may purchase Cartridge BASIC, an enhanced version of the Cassette BASIC permanently stored in the PCjr's memory.

Actually, the cartridges are Read Only Memories (ROMs). By plugging in a cartridge, you are extending the PCjr's memory by 64K. However, you can't write to cartridge memory. You can only read from it.

The system unit may contain a diskette drive. (We'll discuss how to use a diskette drive later on.) If the diskette drive is installed, you will find it at the top right of the system unit. The enhanced model of the PCjr comes with a diskette drive as standard equipment. However, even if you buy the basic model, it is very simple to add a diskette drive later.

If you look at the sides and rear of the system unit, you will notice a number of plugs ("connectors" in computerese). (See Figures 1-8 and 1-9.) These allow you to connect a number of different devices to the system unit. The connectors are coded by letters. For instance, the connector into which you can plug a display is labeled "D." The various connectors on the PCjr include:

Color Display (D) or Television (T)—You must connect the display to the system unit with a cable. The type of cable and the connector into which it is plugged depend on the display you choose. (See the next section.)

Power (P)—All electric power for the system unit passes through the transformer. The transformer is connected to the system unit with a cable.

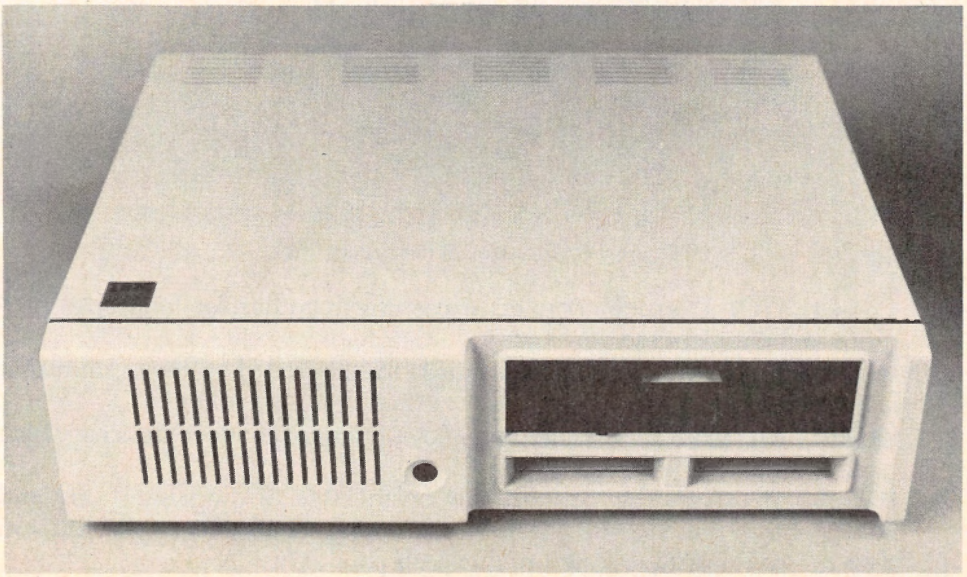


Figure 1-7. The front of the system unit.

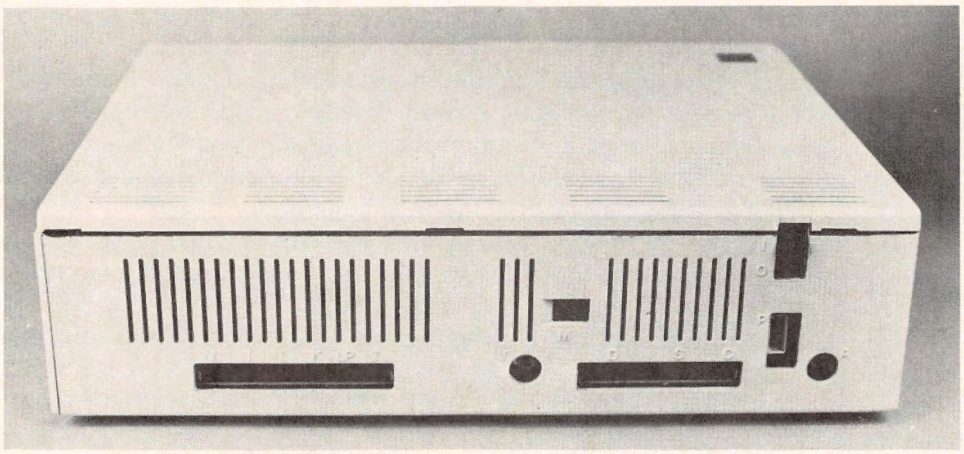


Figure 1-8. The rear of the system unit.

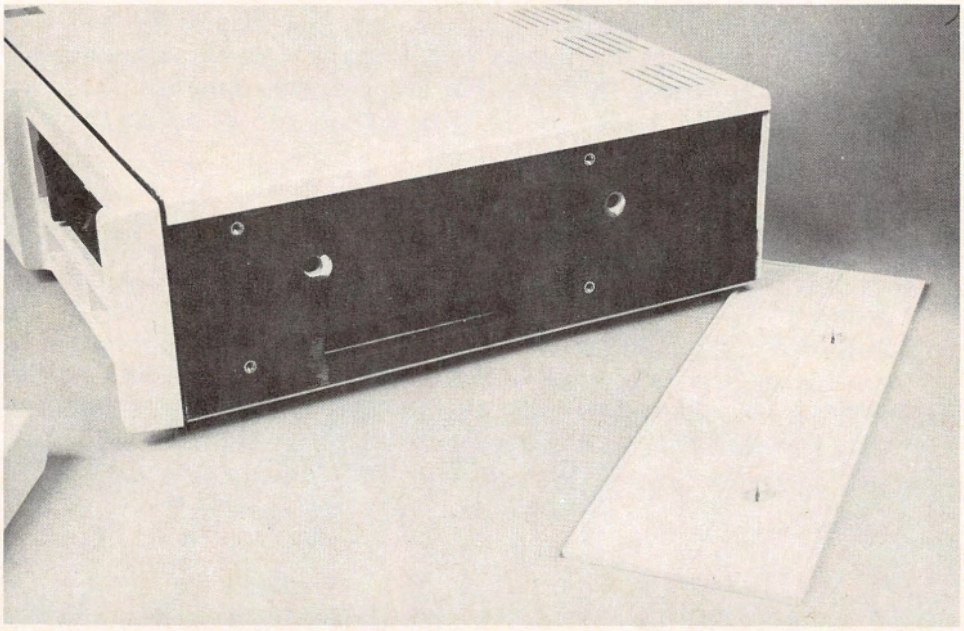


Figure 1-9. The sides of the system unit.

Cassette (C)—You may connect a cassette recorder for long term program and data storage.

Keyboard Cord (K)—You may attach the keyboard to the system unit with an optional cord.

Serial Port (S)—This connector allows you to connect devices which have a serial interface.

Parallel Printer—This connector is on the side of the system unit and allows you to connect the parallel printer adapter. This is required if you wish to use the IBM graphics printer with your PCjr.

Printer (P)—Here is where you connect the IBM compact printer.

Joystick (J)—You may equip your computer system with an optional joystick. (See Figure 1-10.) The joysticks you use with a personal computer are similar to those used with the home video games that have become so popular. The system unit already contains the circuitry necessary to use the joystick, which you plug into the rear of the unit.

Light Pen (LP)—A light pen is a device that lets you input information just by pointing to the screen. (See Figure 1-11.) For example, the computer might display a list of actions. You could use a light pen to point to

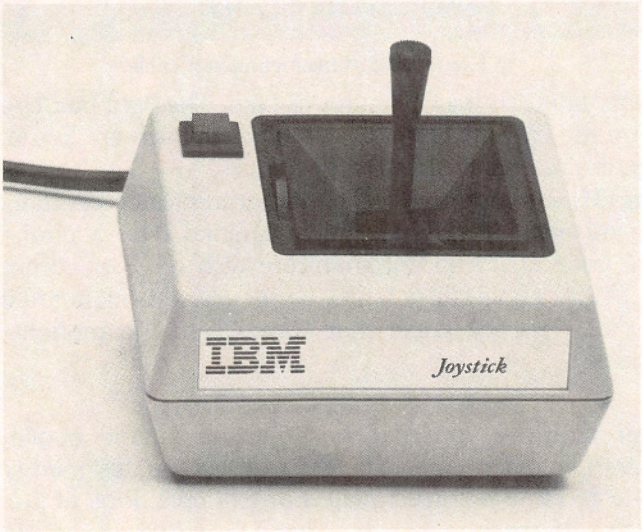


Figure 1-10. A joystick.

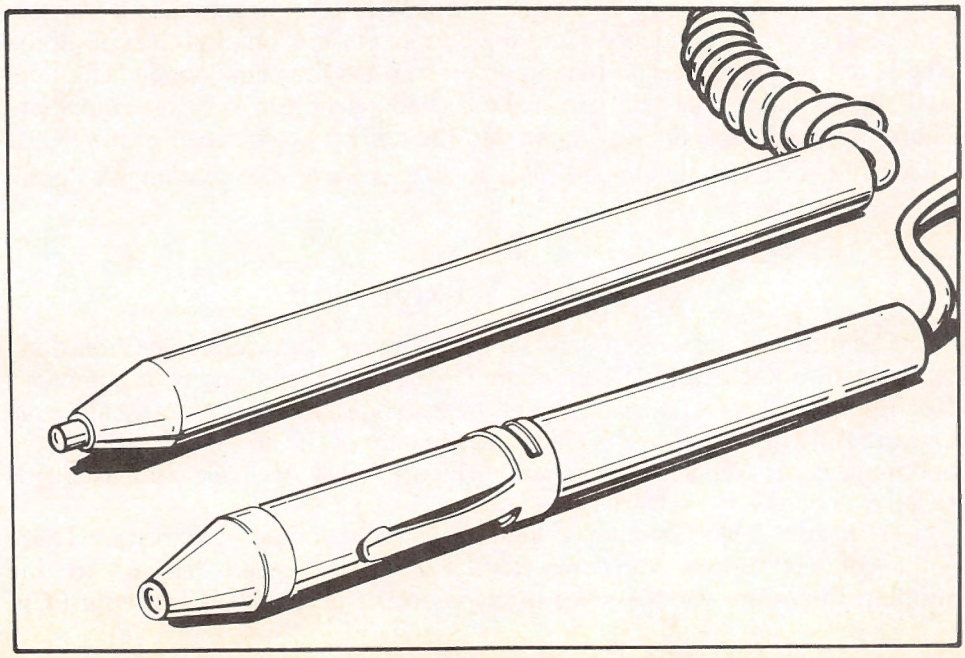


Figure 1-11. Light pens.

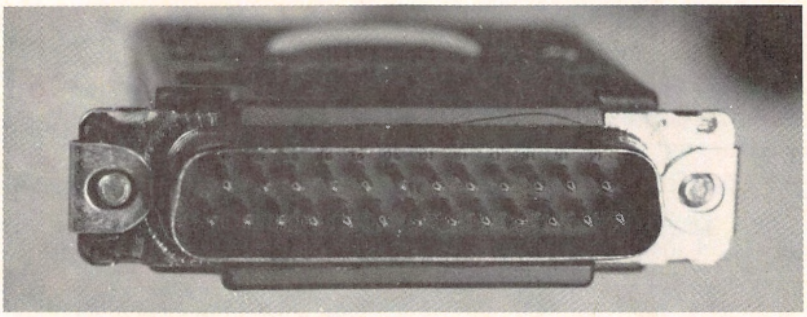


Figure 1-12. A microcomputer cable.

the action you wish the computer to take. The light pen circuitry is able to tell where the tip of the light pen touches the screen. You may plug a light pen into the rear of the system unit.

Modem (M)—A modem is a device that translates internal computer data into electric signals that can be transmitted over a phone line. Using a modem, your PCjr can communicate with another computer or a data service. (More about these later.) If you add a modem to your PCjr, you'll need to connect the cable that comes with the modem to a modular phone jack.

More About Cables—You may be confused by the above discussion of cabling requirements. Don't be. The PCjr cables are designed so that they can only be connected to the correct plug. For example, look at the plug in Figure 1-12. It contains 25 pins.

Each pin carries an electric signal with a particular meaning, so it's important for each pin to be plugged into the correct place. Note that the end of the plug is not rectangular. The mating connector has the same shape. The unusual shapes mean that you can make the connection in only one direction. Therefore, each pin will be plugged into the correct place. Each of the PCjr's cables has a differently shaped plug so that you can be guaranteed "goof-proof" connections.

Inside the System Unit

The inside of the system unit is a marvel of modern electronics. You don't really need to know anything about the circuitry to enjoy using your computer. Nevertheless, I'll spend a few moments describing what's inside the system unit.

Circuit Boards—Most of the electronic circuitry of the computer is organized on circuit cards like the one shown in Figure 1-13. A circuit card usually is called a "board."

A circuit board contains paths which conduct electricity. These paths connect various semiconductor devices (also called "chips") to one another. There are various types of chips on the circuit boards of your PCjr,

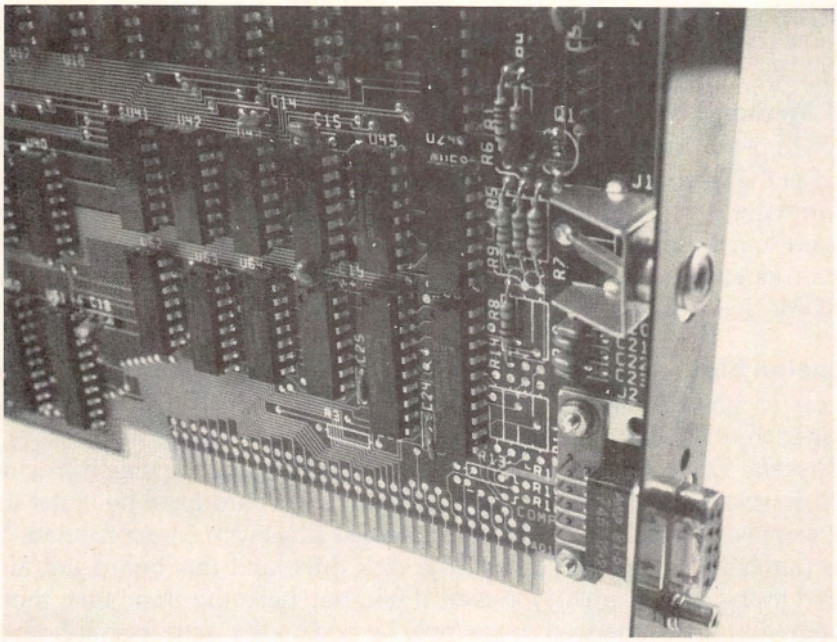


Figure 1-13. A circuit board.

but the most important is the 8088 microprocessor, which is the "brain" of the computer. The 8088 is approximately an inch long, but contains tens of thousands of transistors that control the inner workings of the computer.

RAM Memory—On the circuit boards of the system unit are memory chips, both ROM and RAM. To get an idea of the size of such a chip, look at Figure 1-14, which shows a RAM chip beside a paper clip.

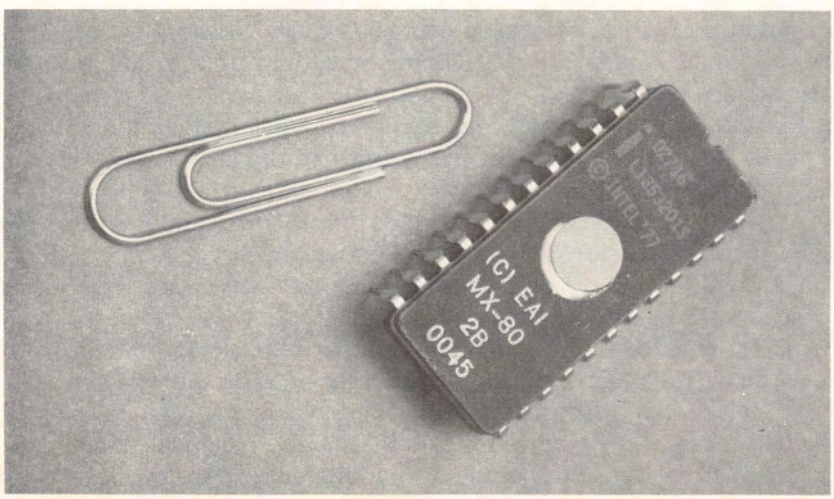


Figure 1-14. A RAM chip.

RAM memory is used for many purposes. When you type in a BASIC program, it is stored in RAM. Sections of RAM are devoted to communications with the printer, screen, and keyboard.

ROM Memory—Some of the memory chips inside the computer are Read Only Memories (ROMs). These chips contain information that has been prerecorded at the factory and that you are unable to change by normal operation of the computer. Within the ROMs are various diagnostic routines, a tutorial to help you learn the keyboard, and the computer language—Cassette BASIC. The cartridges that you plug into the slots on the front of the system unit are just ROMs that extend the ROM memory otherwise available within the system unit.

Expansion Slots—Inside the system unit on the main circuitry board are three slots into which you may insert additional circuit boards. The slots are designed so that each will take only one type of board. Here are the possibilities:

Diskette Drive Adapter Board—This board contains the electronics necessary to control a disk drive. To use a disk drive, this card must be in the appropriate expansion slot and connected to the disk drive with an appropriate cable. If you purchase an enhanced unit, the disk drive and this board are already installed in the system unit. However, if you purchase the basic unit, then you can upgrade to an enhanced unit simply by adding the extra circuit board and disk drive.

Internal Modem—This circuit board allows you to connect your computer to a telephone jack to establish computer-to-computer communications over standard telephone lines.

64K Memory Expansion and Display Adapter Board—This board increases total RAM to 128K and allows you enhanced display capabilities, including:

* 80-character line width.
* Use of high-resolution graphics.

(For a further discussion of display capabilities, see Section 3.3.) This expansion board comes standard in the enhanced system unit. It may be added to the basic system unit.

Other Possibilities—The IBM PC has five expansion slots; the IBM PC/XT has eight. By comparison, the PCjr is short on expansion slots. Before you commit yourself to filling all your slots, you should consult your dealer on the latest offerings from other companies. If the past is any guide, a large number of multipurpose expansion boards that incorporate three, four or more functions on a single board will be available. By using such boards, you can make most efficient use of your precious expansion slots.

In addition to the three predefined expansion slots (modem, memory, diskette), the PCjr has an expansion connector on the right side of the system unit. As of this writing, IBM has not indicated how many options may be connected via this connector; but it appears that all the electrical signals are present for adding an expansion module with additional expansion slots into which you could connect additional diskette drives, a hard disk, additional memory, and so forth.

1.6 Cassette and Diskette Storage

The PCjr can use either cassettes or diskettes for long term storage. You will require one or the other of these devices with your PCjr system if you are to be spared the chore of repeatedly typing the same programs and data each time you wish to use them. (Recall that RAM is erased when the computer is turned off.)

Cassette Recorder

You may connect a standard cassette recorder, (see Figure 1-15) using an optional cable, to one of the connectors at the rear of the system unit. Cassette BASIC, the version of the BASIC language that is in ROM and automatically comes with every PCjr, contains instructions to save and recall programs from the cassette. In addition, Cassette BASIC allows you to read and write data files (like address lists, form letters, financial data) to and from the cassette.

As your storage medium, you may use the same cassettes you now use to tape music or conversations. Just stay away from the bargain basement cassettes. A small flaw in a tape may show up as static in a song. However, it could spell disaster for a stored computer program or a crucial data file. If you stick to good quality tapes, however, you should have no problems.

Note that your PCjr comes with all the electronics necessary to read and write cassettes. The only additional components you must add are the cassette recorder itself and the optional cable purchased from IBM (and, of course, a supply of cassettes). You may use any cassette recorder you have around the house. The only requirement is that the recorder have a connector into which you can plug the cable from PCjr. And almost all cassette recorders have such a connector.

Diskette Drive

You may use a diskette drive with the PCjr. If you have purchased a basic unit, then you may add the diskette drive later. However, if you purchased the enhanced unit, then a diskette drive is standard.

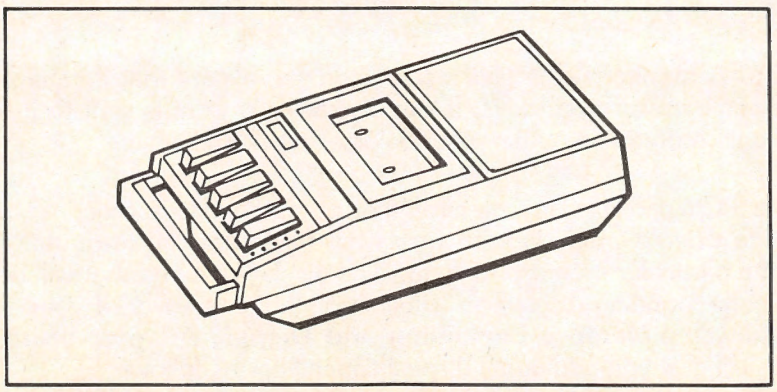


Figure 1-15. A standard cassette recorder.

The visible portion of the diskette drive is the door at the top of the front of the system unit. This door is for inserting diskettes (more about that later). The electronic circuitry necessary to operate the diskette drive is contained on the **diskette drive adapter** that occupies one of the expansion slots inside the system unit.

The diskette drive for the PCjr is a "half-height" drive and can write to either single-sided or double-sided diskettes. It uses "double-density" diskettes and can write 180K bytes on each side of the diskette. We'll discuss the detailed operation of the diskette drives in a later chapter.

1.7 Displays

The display of your PCjr system is an extremely important component. As you type on the keyboard, the keystrokes are displayed on the screen. (In computer jargon, the keystrokes are **echoed** to the screen.) In addition, the display is an output device to which programs send data.

You have three choices for the display of your system:

A television set
A "video composite" monitor
A "direct drive" or "RGB" monitor, such as the IBM color display

I'll describe the features of each of these displays.

Television Set—You can use a television set (black and white or color) as a display. To connect it to the PCjr, you will need the optional PCjr Connector for TV. You probably have used a similar connector if you ever have connected a video game system to your television. The connector attaches to the video input of your television (usually with two screws on the rear of the set). It has two positions: TV and Computer. On the PCjr, the cable is plugged into connector T on the rear of the system unit.

A television set allows you to display 25 lines of 40 characters each. In graphics mode, a television screen is divided into as many as 200 lines of 320 dots each. In the case of a color set, you may display color text and graphics. Moreover, the PCjr can produce sound using the speaker system in the television set.

Monitor—A monitor resembles a television set, except that it cannot receive regular commercial broadcasts. It can display only what is sent to it from the system unit. Monitors may be either color or black and white.

The PCjr can use any monitor that accepts what is called a "standard composite video signal." Don't worry what this means. If in doubt, ask your dealer if a particular monitor will work. The display on a monitor will be clearer than a corresponding display on a television set. Moreover, if your system is equipped with a **Memory Expansion and Display Adapter** board, then a monitor will give you additional display capabilities as follows:

Text lines may be either 40 or 80 characters wide.

You may use the **high-resolution graphics mode**, in which the screen is divided into 200 lines of 640 dots each. However, you should note that in high-resolution graphics mode you may use only four colors simultaneously, as opposed to 16 in low- and medium-resolution modes.

A monitor will require a special cable to plug it into the connector on the display adapter. (Note that the adapter cable for a television set will not do the job.) This cable is plugged into connector D on the rear of the system unit.

IBM Color Display or other RGB Monitor—This display is a color monitor that produces particularly striking colors. You may use this display on any of IBM's personal computers (PCjr, PC, or PC/XT). To use it on the PCjr, you will need a special cable to connect it to the system unit (plugged into connector D).

1.8 Printers

There are several printers available for use with the PCjr. They span quite a wide range of printer technology as well as a wide price range. They are discussed in order of increasing price.

IBM Compact Printer

This is an inexpensive printer that takes up only a small amount of space. It forms characters from dots, with each character printed in a 5-dot x 8-dot rectangle.

This printer can print 50 characters per second and is capable of printing a total of 191 different characters (the usual letters, numbers, and punctuation marks, as well as a set of graphics characters).

The IBM Compact Printer uses a special thermal paper onto which the printer "burns" the letters. You may purchase suitable paper in individual sheets, rolls, or fanfold from your PCjr dealer.

You may print an 8-inch line on standard 8½-inch wide paper. You can print standard width characters (10 per inch), double width characters (5 per inch), compressed characters (17.5 per inch), and compressed, double width characters (8.75 per inch).

The IBM Compact Printer has a graphics mode in which you may control the individual dots in each character rectangle. In this mode, you can print lines that are 560 dots wide and 8 dots high. This corresponds to a resolution of 70 dots per inch horizontally and 48 dots per inch vertically.

The IBM Compact Printer plugs directly into the system unit with a cable that comes with the printer.

IBM 80 CPS Graphics Printer

This is the next printer in order of sophistication and price. It still prints characters as rectangles of dots. However, each character is formed in a 9 x 12 rectangle. The advantages of this printer are:

Its speed is 80 characters per second.
It delivers better print quality than the IBM Compact Printer.
It can use ordinary as opposed to thermal paper.
This printer has several graphics modes, allowing much higher resolution than is possible on the Compact Printer.

The main disadvantages are:

It takes up more space.
It costs more.
It is considerably noisier.

The IBM 80 CPS Graphics Printer requires a **parallel interface**. To supply this interface in the PCjr, you must attach the Parallel Printer Attachment to the right side of the system unit. You then plug your printer into the connector on the attachment.

Other Printers

A host of non-IBM printers are compatible with the PCjr. Almost any printer with a parallel interface will do. The EPSON FX-80 belongs to the same family as the IBM Graphics Printer but has greater capabilities and twice the print speed. Another possibility is the IBM Color Printer, which allows you to print in four colors. There are many other printers. See your computer dealer for details.

1.9 Communications

Very significant among the features of the PCjr is its communications capability. You may equip your PCjr system with a device called a **modem** that will enable you to transfer computer data over ordinary phone lines. You can use this capability in many interesting applications, which I'll explore in the next chapter.

Internal versus External Modems

With the PCjr, you may use either an internal modem, which fits into one of the expansion slots, or an external modem that connects to the serial port (RS232-C interface) at the rear of the system unit.

The PCjr Internal Modem—The PCjr has an expansion slot designed to take the **PCjr Internal Modem** board. This board contains the complete electronics for converting the electric signals that represent data within the computer into corresponding electric signals that can travel along ordinary phone lines and then be reconverted into computer data at the receiving end of the line. (Of course, the conversion at the receiving end must be done by another modem.)

The PCjr Internal Modem has a single cable that extends from the system unit. This cable must be plugged into a modular phone jack. Most phone installations done in the past 10 years use modular jacks. If you can unplug your phone from the outlet in the wall, then the socket remaining after you unplug the phone is a modular jack. To connect the modem, just insert the plug at the end of the cable into the modular jack. You will hear a slight "click" as the plug seats itself into the jack.

PCjr BASIC has the instructions for operating the modem. These instructions allow you to dial a number or repeatedly dial a number until a connection is established. (This is useful for calling a line that is often busy.)

The speed of data transmission is measured in **baud**. Roughly speaking, 10 baud corresponds to one character per second. For example, a transmission speed of 110 baud corresponds to about 11 characters per second. The PCjr Internal Modem gives you a choice of two transmission speeds, 110 and 300 baud.

Using an External Modem—You also may connect an external modem to your PCjr. To do this, you will need a modem cable to run from the serial port at the rear of the system unit to the modem. There are two types of external modems. The older (and less expensive) type is the **acoustic modem**. This type of modem uses the acoustics in the phone receiver as part of the transmission process. To use an acoustic modem, you must rest the phone receiver in a special cradle on the modem. The computer data is then converted into audible tones that are then converted, by the receiver, into electric signals that travel along the phone lines. An acoustic modem can transmit at speeds up to 300 baud. An acoustic modem is a possibility if you don't have modular phone jacks.

The more modern type of modem is a **direct-connect** modem, which connects directly to a modular phone jack, just like the PCjr Internal Modem. A direct-connect modem can transmit at speeds up to 1200 baud. If you will be transmitting many lengthy documents, you will find that 300 baud is agonizingly slow. In this case, you should consider the added speed of a direct-connect modem. (At 300 baud, it takes about 10 seconds to transmit a single 300-word page. Imagine transmitting a 300-page report!)

Setting Communications Parameters

In order for two computers to communicate, their transmissions must have the same characteristics. In particular, they must be set for the same baud rate. BASIC has the software for adjusting the baud rate, as well as a number of other communications parameters. In addition, each of the communicating computers must be running a communications program to synchronize the transmission of data.

1.10 The Serial Port

There are many devices you can connect to your PCjr, and other companies are sure to increase the number of such devices in the months ahead. Most

external devices are connected to the PCjr via the serial port, whose connector can be found on the rear of the system unit.

A serial port (also called an RS232-C interface) uses a 25-pin connector, where the electric signal carried by each pin of the connector is defined by an international standard.

Using the serial port you may, for example, connect some letter quality printers and plotters to your PCjr. Moreover, you may use the serial port to directly wire two PCjrs together, so that they may exchange information.

In spite of the international standard of the RS232-C interface, there are many variations requiring special cables. In using the serial port, you may require a special cable for each device to be connected.

BASIC contains instructions for communicating data to the serial port.

Introducing —

Programs to Accompany IBM PCjr: An Introduction to DOS, BASIC Programming Applications

Larry Joel Goldstein

You can get started right away, because all the program keyboarding has already been done for you!

This companion software diskette offers you *faster* and *easier* access to PCjr programming and applications. Just look at what you'll get:

- All the programs (64 in all) listed in the book. Including ready-to-run games . . . word processing . . . personal financial management, and much more.
- Hours and hours of your valuable time saved. With the aid of this handy diskette, you'll be programming in minutes!
- Frustrating, mind-numbing keyboarding errors eliminated with a single step. Leaves your mind clear, and your fingers nimble to create and enjoy.

Here's How To Order Your Copy of *Programs to Accompany IBM PCjr: An Introduction to DOS, BASIC Programming Applications*

Enclose a check or money order for $24.95 plus local sales tax. Slip in this handy order envelope — and mail! No postage needed. Or charge it to your VISA or MasterCard. Just complete the information below.

ORDER TODAY!

☐ **YES!** I want to make bringing up "jr" fast and easy. Please rush me **Programs to Accompany IBM PCjr: An Introduction to DOS, BASIC Programming Applications/** D5378-8. I have enclosed payment of $24.95 plus local sales tax.

Name _____

Address _____

City _____ State _____ Zip _____

Charge my Credit Card Instead

☐ VISA ☐ MasterCard

_____
Account Number

_____
Expiration Date

_____
Signature as it appears on Card

Brady

Robert J. Brady Co. • Bowie, MD 20715
A Prentice-Hall Publishing and Communications Company
Dept. Y B1085-SI(5)

Now you can make bringing up "jr" even easier!

See over for details . . .

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1976 BOWIE, MD

POSTAGE WILL BE PAID BY ADDRESSEE

Robert J. Brady Co.
A Prentice-Hall Company
Bowie, Maryland 20715

2

USING YOUR PCjr FOR THE FIRST TIME

In this chapter, I will guide you through your first session with the PCjr. If you don't have a disk drive, read only Section 2.1, which teaches you how to turn on the machine, followed by Section 2.6, which teaches you the keyboard fundamentals. You can skip Chapter 3 entirely and proceed directly to the introduction to BASIC, beginning in Chapter 4.

If your PCjr is equipped with a diskette drive, you should begin reading with Section 2.2.

At various points in the discussion, I include **TEST YOUR UNDERSTANDING** questions on your understanding of the material just covered. The answers to these questions are at the end of the section. In addition, there are exercises with answers at the back of the book.

2.1 Starting Your PCjr (Without DOS)

The following instructions tell you how to start you PCjr without using the Disk Operating System (DOS). You should follow these instructions either if your

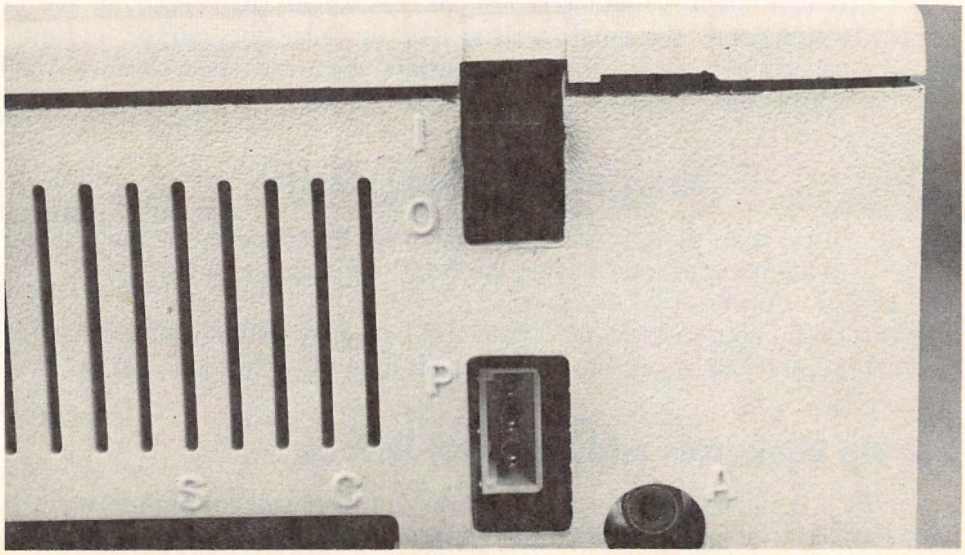


Figure 2-1.

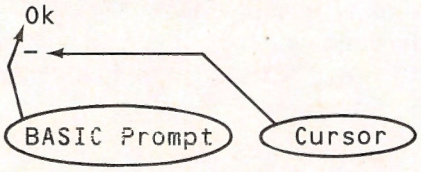
computer doesn't have a diskette drive or if you wish to operate the computer without using the diskette drive.

There are two versions of BASIC: Cassette BASIC, which is stored in ROM and is always available, and Cartridge BASIC, which is available in a program cartridge. The starting procedures are slightly different depending on the version of BASIC you wish to use.

Starting Cassette BASIC

1. Turn on the monitor.
2. On the system unit you will find the computer On-Off switch. (See Figure 2-1.) Flick it to the On position.
3. For about 30 to 45 seconds, the computer does diagnostic testing to determine if all its components are in good working order. When the tests are complete, the computer should respond with a beep and a display similar to:

```
The IBM Personal Computer Basic
Version C1.00 Copyright IBM Corp. 1981
xxxxx Bytes Free
Ok
_
```




```
The C in the Version number indicates Cassette Basic.
```

4. Note the letters **Ok** in the last line of the display. These letters are the **BASIC prompt** and indicate that cassette BASIC is ready to accept instructions. The small blinking dash is called the **cursor** and indicates the place on the screen where the next typed character will appear.

Starting Cartridge BASIC

1. Insert the program cartridge containing Cartridge BASIC into either of the two cartridge slots in the front of the system unit.

2-5. Perform steps 1-4 above.

The version number in the final display will be slightly different from the one given above and will be preceded by a J indicating PCjr Cartridge BASIC.

2.2 On Diskettes and Diskette Drives

If your PCjr is equipped with a diskette drive, it is a critical part of the system. It will allow you to store and retrieve both programs and data. Before we proceed any further, let's get acquainted with these remarkable devices.

The Anatomy of a Diskette

To store information, the diskette drives use 5-1/4-inch floppy diskettes. Diskettes come in single-sided and double-sided versions. The single-sided diskettes may be written on only one side, a double-sided diskette on both sides. The diskette drive of your PCjr uses double-sided diskettes that can accommodate roughly 360,000 characters (about 100 double-spaced typed pages).

Figure 2-2 shows the essential parts of a diskette. The diskette itself is a magnetically coated circular piece of mylar plastic that rotates freely within a stiff jacket. The jacket is designed to protect the diskette. The interior of the jacket contains a lubricant that helps the diskette rotate easily. The diskette is sealed inside. You should *never* attempt to open the protective jacket. The labels on the jacket identify the contents of the diskette.

The diskette drive reads and writes on the diskette through the **read-write** window. NEVER, under any circumstances, touch the surface of the diskette. Diskettes are very fragile. A small piece of dust or even oil from a fingerprint could damage the diskette and render parts of the information on it useless.

The **write protect notch** allows you to prevent changes to information on the diskette. When this notch is covered with one of the write protect labels provided with the diskettes, the computer may read the diskette, but it will not write or change any information on the diskette. To write on a diskette, the write protect notch must be uncovered.

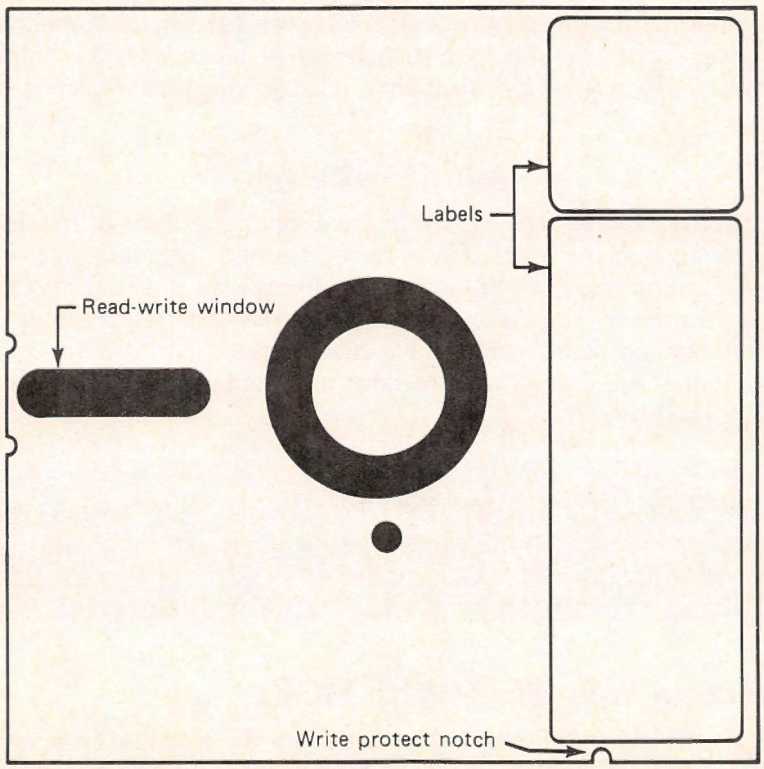


Figure 2-2. A diskette.

You should have a few blank diskettes on hand. Take a moment to inspect one of them and locate the parts of the diskette described above. (The labeling on your diskettes may differ from that shown in Figure 2-2.)

Cautions in Handling Diskettes

Diskettes are extremely sensitive. Here are some tips in using them.

1. Always keep a diskette in its paper envelope when it is not in use.
2. Store diskettes in a vertical position just like you would a phonograph record.
3. Never touch the surface of a diskette or try to wipe the surface of a diskette with a rag, handkerchief, or other piece of cloth.
4. Keep diskettes away from extreme heat, such as radiators, direct sun, or other sources of heat.
5. Never bend a diskette.
6. When writing on a diskette label, use only a felt tipped pen. Never use any sort of instrument with a sharp point.
7. Keep diskettes away from magnetic fields, such as those generated by electric motors, radios, TVs, tape recorders, telephones, and other electric devices. A strong magnetic field may erase data on a diskette.
8. Never remove a diskette while the drive is running. (You can tell if a drive is running by the sound of the motor and the "in use" light on the front of the drive.) Doing so may cause permanent damage to the diskette.

The above list of precautions may seem overwhelming to someone starting out. However, once you set up a suitable set of procedures for handling and storing diskettes, you'll find that they are a reliable, long-lasting storage medium.

Using Diskettes

To insert a diskette into a diskette drive, open the door of the drive. Turn the diskette so that the label side is facing up and the read-write window is closest to the computer. Gently push the diskette all the way into the drive. Close the drive door. The diskette now can be read by the computer.

To remove a diskette from a drive, first be sure that the light to the left of the drive door is off. Open the drive door and gently pull the diskette forward and out of the drive.

TEST YOUR UNDERSTANDING 1

Take a blank diskette and practice inserting it in the diskette drive on the left. Remove the diskette from the drive.

2.3 Starting Your PCjr (With DOS)

To control the flow of information to and from the diskette drive, we need a program called an **operating system**. Such a program acts as a manager for all the activities that go on in the computer. More specifically, it coordinates the

flow of information among the keyboard, video display, RAM, ROM, diskette drive, and any other peripheral devices you may add to your computer system.

The official operating system of the IBM family of personal computers is called **IBM DOS** (IBM Disk Operating System, pronounced IBM dóss), also called **MS-DOS*** , **PC-DOS**, or just **DOS** for short. Actually, DOS has undergone several revisions since its conception. The latest version is DOS 2.10, and this version runs on all three IBM personal computers: the PCjr, the PC, and the PC/XT.

When you purchased your system, you also should have bought a copy of the **Disk Operating System** manual. Just inside the rear cover of the DOS manual is a plastic jacket that contains two diskettes, one labeled **DOS** and the other **DOS Supplementary Programs**. The first diskette is your master copy of the programs necessary to operate your diskette drives. This diskette is extremely important. So important, in fact, that it does not have a write protect notch. This means that you can never write on this diskette. (No chance for accidentally altering its programs!) I refer to this diskette as the **master DOS diskette**.

To use DOS, it's first necessary to read DOS into the computer. Ordinarily, this would be done with a *copy* of the master DOS diskette, rather than with the original itself. However, on your first pass, you don't yet have any extra copies of the DOS diskette, so we must use the master. Here's the procedure to follow.

Starting Your Computer

1. Insert the DOS diskette into the diskette drive. The label side should be up. Push the diskette all the way to the rear of the drive. Close the drive door.

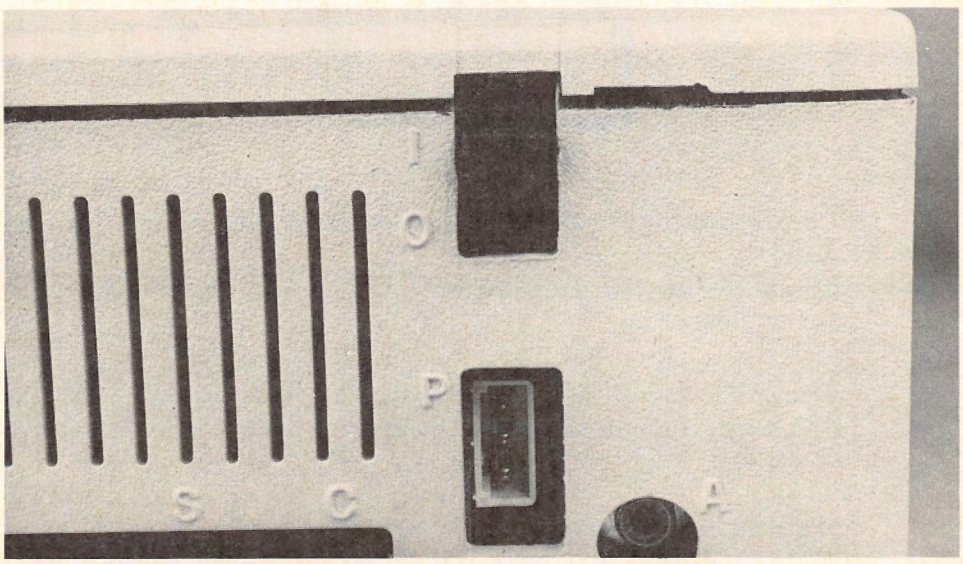


Figure 2-3.

*MS stands for Microsoft, the corporation that designed the operating system.

2. Insert the BASIC cartridge into either cartridge slot (at the bottom of the front of the system unit). Be sure to push the cartridge all the way into the slot.
3. Turn on your monitor.
4. Turn on the printer (if one is connected).
5. On the system unit (the box in which the diskette drive sits) you will find the computer On-Off switch. (See Figure 2-3.) Flick it to the up position. For 30 to 45 seconds, the computer does diagnostic testing to determine if all its components are in good working order. When the tests are complete, the computer should respond with a beep and the display:

```
Current date is Tue 1-01-1980
Enter new date:
```

6. Type in today's date (in the format 4-22-99 for April 22, 1999). Press the **ENTER** key, which is the large key with the symbol:

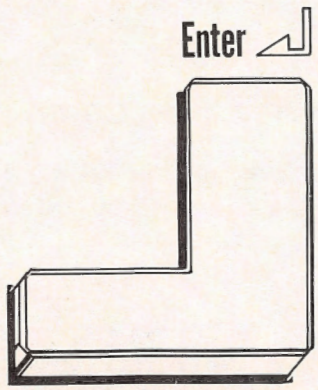


Fig. 2-4a

The computer will respond with a display similar to:

```
Current time is 0:00:00.00
Enter new time
```

Type in the correct time in the format 14:03:00 for 2:03 PM. (The PCjr uses a 24-hour clock.) The computer will respond with a display similar to:

```
The IBM Personal Computer DOS
Version 2.10 Copyright IBM Corp. 1981,1982,1983
A>_
```

DOS prompt

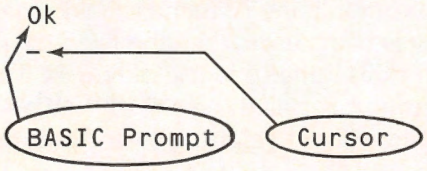
The symbol A> is called the **DOS prompt** and it tells you that DOS is loaded and ready to accept commands.

7. In this chapter, we'll learn to use many of the DOS commands. At this point, let's give the command to read the BASIC programming language from the DOS diskette into RAM. Type:

```
basic
```

Press the **ENTER** key. The computer will respond with a display similar to:

```
The IBM Personal Computer Basic
Version J1.00
Copyright IBM Corp. 1981,1982,1983
xxxxx Bytes Free
Ok
_
```



Note the letters **Ok** in the last line of the display. These letters are the **BASIC prompt** and indicate that the computer language (called BASIC) is ready to accept instructions. The small blinking line is called the **cursor** and indicates the place on the screen where the next typed character will appear.

8. To return from BASIC to DOS, type:

```
system
```

and press ENTER. The computer will display the DOS prompt A> .

TEST YOUR UNDERSTANDING 2* (answer on page 28)

a. Turn on your computer and load DOS.
b. Load BASIC.
c. Return to DOS.

Turning Off the Computer

1. Turn off the system unit.
2. Turn off the monitor.
3. Remove any diskettes from the drives.

*Answers to TEST YOUR UNDERSTANDING questions (where appropriate) are given at the end of the current section.

ANSWERS TO TEST YOUR UNDERSTANDING

2. a. Follow the instructions 1-8 for Starting Your Computer.
   b. Follow instruction 7.
   c. Follow instruction 8.

2.4 More About Diskette Drives

The PCjr can be equipped with only a single diskette drive. However, DOS is capable of recognizing up to four drives. The diskette drives are given the names A:, B:, C:, and D: (note the colons). Drive names are used to refer to a drive within a command.

The single diskette drive on the PCjr actually can be referred to by drive name A: or B:. At any given moment, only one of these names is in effect. You can tell which by looking at the DOS prompt. If the A: name is being used, then the prompt will be A>. If the B: name is being used, then the DOS prompt will be B>. Below you'll see the reason for this rather odd use of names. The drive whose name is shown in the DOS prompt is called the **current drive** or the **default drive**. If you give a command without referring to a drive, then the current drive is assumed.

To change the current drive:

1. Obtain the DOS prompt A> or B>.
2. Type the name of the new current drive (remember the colon). Press ENTER.

TEST YOUR UNDERSTANDING 1 (Answers on page 28)

a. Turn on your computer and change the current drive from A: to B: .
b. Change the current drive back to A: .

ANSWERS TO TEST YOUR UNDERSTANDING

1. a. After turning on the computer and obtaining the DOS prompt, type B: and press ENTER.
   b. Type A: and press ENTER.

2.5 Backing Up Your DOS Diskette

Good programming practice dictates that you keep duplicate copies of all your diskettes. In computer language, a copy is called a **backup**. Making backups

reduces the chance that you'll lose your programs and data due to accidents (power blackout, coffee spilled on a diskette, and so forth). It is an especially good idea to make a copy of the master DOS diskette the first time you use it. Later, you should use only the copy. The original DOS diskette should be stored in a safe place so that yet another copy can be made if the first copy is damaged. Here is the procedure for making a backup copy of a diskette.

Copying One Diskette Onto Another

You must copy the contents of the master DOS diskette onto a blank diskette.

1. Obtain the DOS prompt A>. Insert the DOS diskette into drive A:. Type:

   ```
   DISKCOPY
   ```

   Then press the **ENTER** key. The computer will respond with the display:

   ```
   Insert source diskette in drive A:
   Strike any key when ready
   ```

2. The source diskette, namely the DOS diskette, is already in drive A:, so strike any key. The computer will respond with the prompt:

   ```
   Copying 9 sectors per track, 1 sides(s)
   ```

   DOS will copy a portion of DOS from the diskette into RAM. When it has copied as much as it can, it will display:

   ```
   Insert target diskette in drive A
   Strike any key when ready
   ```

   Remove the DOS diskette and insert a blank diskette. Next, press any key. The computer will now copy the data in RAM onto the diskette. If there is more data to be copied from the DOS diskette, you will be directed to reinsert the DOS diskette. Then steps 1 and 2 will be repeated a number of times. After all the data has been copied, the computer will display the prompt:

   ```
   Copy complete
   Copy another? (Y/N)
   ```

   Answer N to indicate that we do not wish to copy another diskette. The computer will now display the DOS prompt:

   ```
   A>
   ```

3. Your blank diskette is now an exact copy of the original. At this point, you may give another DOS command or request BASIC.

TEST YOUR UNDERSTANDING 1 (Answer on page 31)
Make a copy of the DOS system diskette supplied with your diskette operating system.

From now on you should use only the **copy** of the master DOS diskette, and **not** the original. Put the original in a safe place, so that it can be used to make yet another copy if the current copy is damaged.

A Word to the Wise

The backup procedure just described may be used to copy the contents of a diskette onto another.* Because diskettes are fragile, I strongly urge you to maintain duplicate copies of all your diskettes. A good procedure is to update your copies at the end of each session with the computer. This may seem like a big bother, but it will prevent untold grief if, by some mishap, a diskette with critical programs or data is erased or damaged.

TEST YOUR UNDERSTANDING 2 (Answer on page 31)
Use your copy of the master DOS diskette to make another copy. (A copy of a copy is just as good as the original!)

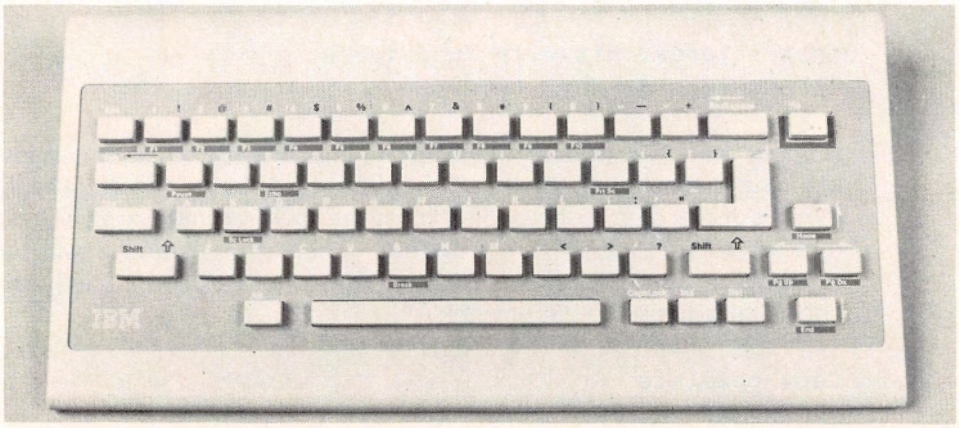


Figure 2-4. The PCjr keyboard.

*Note, however, that some diskettes containing programs are copy-protected by their manufacturers. Such diskettes cannot be copied using the DISKCOPY command.

ANSWERS TO TEST YOUR UNDERSTANDING

1: Follow instructions 1-3 on page 29 .
2: Follow instructions 1-3 on page 29 , but start with the copy DOS diskette in drive A:.

2.6 The Keyboard

Examine the PCjr keyboard. (See Figure 2-4.) This keyboard looks complex, but I'll cover the functions of the various keys slowly to give you time to feel comfortable with the design.

The central section is very much like a typewriter keyboard. There are a few symbols that are not present on a typewriter, such as:

< > ^ ~ [ ] \ { }

Also, you should note the following important differences from a typewriter keyboard:

1. There are separate keys for 1 (one) and l (el). (Many typewriters let the lowercase l serve as a one.)
2. The number Ø (zero) is included with the other numbers. It has a slash through it. This is to distinguish it from the letter O.

Here are the functions of the other keys in the central portion of the keyboard:

Space bar. Generates a blank space just like the space bar on a typewriter.

Shift key. Shifts keys to their uppercase versions. For keys with two symbols, the rightmost symbol takes effect. The uppercase versions are in effect only as long as the shift key is held down. Releasing the shift key causes keys to assume their lowercase meanings. Note that there are two shift keys, one on each side of the keyboard.

Caps Lock key. In most computer work it is convenient to type using only capital letters. For one thing, capitals are larger and easier to read on the screen. You may turn off the lowercase letters by pressing the Caps Lock key. In this mode, the letter keys are typed as capitals. *Note, however, that the nonletter keys (such as 1 and !) still have two meanings. To type the upper symbol, you must still use the shift key.* To exit from the all-capitals mode, once again press the Caps Lock key. With the Caps Lock key engaged, if you press the shift key and a letter key, a lowercase letter will be displayed.

Backspace key. Moves the cursor back one space. Erases any letter it backs over.

ENTER key. Similar to a carriage return key on a typewriter. Used to end a line and to place the cursor at the beginning of the next line. A line may be corrected with backspaces until ENTER is pressed.

Tab key. Works like the tab key on a typewriter. Moves the cursor to the next tab stop.

Control key. Used in combination with other keys. For example, the key combination Ctrl-A means to simultaneously press Ctrl and A. Such combinations are used to generate control codes for the screen and printer.

Escape key. Used to indicate that certain sequences of letters are to be interpreted as control codes.

Function key. Used in combination with other keys to generate various function sequences.

Turn on your PCjr and obtain the DOS prompt A>. (If you are not using DOS, obtain the BASIC prompt Ok.) Strike a few keys to get the feel of the keyboard. Note that as you type, the corresponding characters will appear on the screen. Note, also, how the cursor travels along the typing line. It always sits where the next typed character will appear.

As you type, you should notice the similarities between the PCjr keyboard and that of a typewriter. However, you also should note the differences. At the end of a typewriter line you return the carriage manually or, on an electric typewriter, with a carriage return key. Of course, your screen has no carriage to return. However, you still must tell the computer that you are ready to move on to the next line. This is accomplished by hitting the **ENTER** key. If you press the **ENTER** key, the cursor will then return to the next line and position itself at the extreme left side of the screen. The **ENTER** key also has another function. It signals the computer to accept the line just typed. Until you hit the **ENTER** key, you may add to the line, change portions, or even erase it. (You'll learn to do these editing procedures shortly.)

Keep typing until you are at the bottom of the screen. If you hit **ENTER**, the entire contents of the screen will move up by one line and the line at the top of the screen will disappear. This movement of lines on and off the screen is called **scrolling**

As you may have noticed, the computer will respond to some of your typed lines with error messages. Don't worry about these now. The computer has been taught to respond only to certain typed commands. If it encounters a command that it doesn't recognize, it will tell you so with an error message. It is extremely important for you to realize that these errors will in no way harm the computer. In fact, there is little you can do to hurt your computer (except by means of physical abuse, of course). Don't be intimidated by the occasional slaps on the wrist handed out by your computer. Whatever happens, don't let these "slaps" stop you from experimenting. The worst that can happen is that you might have to turn your computer off and start all over!

System Reset

You may restart the computer from the keyboard by pressing the **Ctrl**, **Alt**, and **Del** keys simultaneously. This key sequence will return the computer to the state it was in just after being turned on. Both RAM and the screen will be erased.

Printing the Screen

The PCjr provides several features that allow you to print what appears on the screen. First, make sure your printer is turned on. Obtain the DOS prompt A> and press the key combination Fn followed by Echo (which shares the E key). We will write this key combination as Fn-Echo. All subsequent text that appears on your screen also will be printed. This provides you with a written record of a session at the computer. To turn off the printing, press Fn-Echo again.

You may obtain a printed copy of the current screen by pressing the key combination Fn-PrtSc. (PrtSc shares the P key.)

TEST YOUR UNDERSTANDING 1 (Answers on page 35)
Print the current contents of the screen.

Keyboard Usage in BASIC

Many of the keys have special meanings while BASIC is running. To illustrate this keyboard usage, load BASIC by obtaining the DOS prompt and typing:

```
basic
```

followed by ENTER. When you obtain the BASIC prompt, begin typing. Notice that if you neglect to end a line, it spills over onto the next. However, after several lines (255 characters) BASIC automatically terminates the line, just as if you had pressed ENTER.

Scrolling and corrections using the backspace key work pretty much the same in BASIC as they do in DOS. Fill your screen with eight or ten lines of text. To erase the screen, use this key combination:

```
Ctrl-Fn-Home
```

In using the Ctrl key in connection with any other key, press the other key while holding down Ctrl. In this case, push Fn while holding down Ctrl. Release these keys and then press Home. In response to the key combination Ctrl-Fn-Home, all characters on the screen will be erased, and only the cursor will remain. The cursor is positioned in the upper left corner of the screen, its so-called "home" position.

Editing Keys

Cursor Motion Keys. These four arrow keys are used to move the cursor in the indicated directions. Note that these keys move the cursor only in BASIC. Don't confuse the up arrow with the shift key.

Insert (INS) Key. When this key is pressed, you may insert text at the current cursor position. As text is inserted, existing text is moved to the right to accommodate the new letters. The effect of the Ins key is cancelled either by pressing Ins again, by pressing Del, by pressing ENTER, or by using the cursor motion keys.

Delete Key. When this key is pressed, one letter is deleted at the cursor position.
Note that the cursor motion keys (arrow keys) have the following alternate designations:

PgUp on the left arrow
PgDn on the right arrow
End on the down arrow
Home on the up arrow

These alternate designations refer to ways in which these keys are used in combination with the Fn key.

TEST YOUR UNDERSTANDING 2 (Answers on page 35)

a. Type your name on the screen.
b. Erase the screen.
c. Repeat a. using all capital letters. (Don't worry about the computer's response to your typing!)

Line Width The basic PCjr unit displays lines having up to 40 characters. The enhanced unit allows line widths of either 40 or 80 characters. To switch from one line width to the other, use the WIDTH command. To switch to 40 characters per line, type:

```
WIDTH 40
```

followed by ENTER. To return to 80 characters per line, type:

```
WIDTH 80
```

followed by ENTER. In the rest of this text, we will assume that the lines are 80 characters wide. If you use a 40 character line width, your displays may look somewhat different than those indicated.
Function Key Display Note that the last line of the screen is filled with data that does not change as you type. This data displays the assignment of certain user-programmable key combinations, called F1-F10. You may generate user function F1 with key combination Fn-1. User function F2 may be generated with the key combination Fn-2, and so forth. The current assignments of these keys are displayed on the last line of the screen. Figure 2-5 shows the initial function key assignment. Note, however, that if your screen width is currently 40, then only the definitions of keys F1-F5 are displayed.
You may turn off the function key display by typing:

```
KEY OFF
```

followed by **ENTER**. If you wish to turn on the display, type:

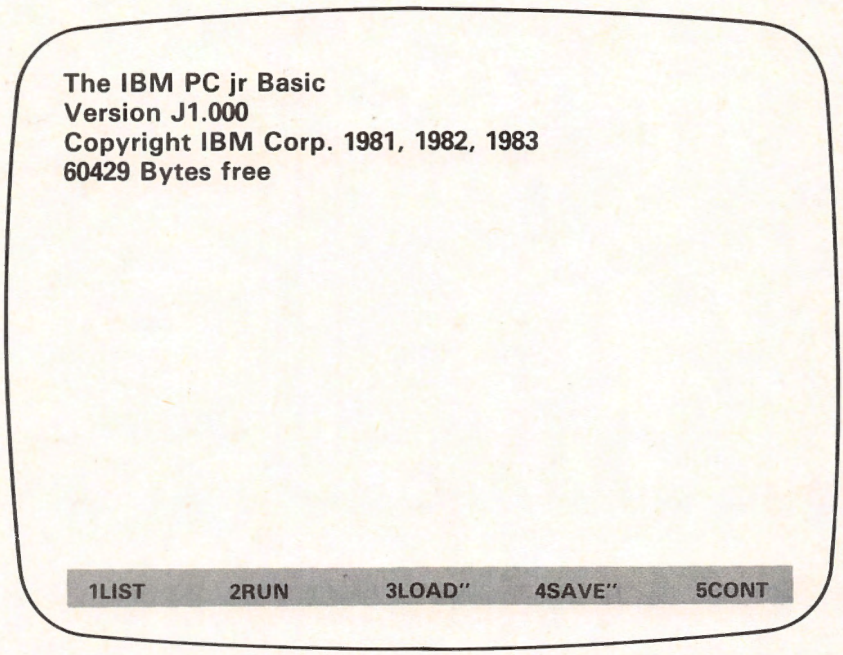


Figure 2-5. Function key display.

```
KEY ON
```

By keeping the display line off, you make the last screen line available for program use.

Exercises

Type the following expressions on the screen. After each numbered exercise, clear the screen. (These exercises contain some typical expressions you will be typing when you use BASIC. They are provided as practice in typing and manipulating the keyboard.)

1. `10 Print "Hello."`
3. `10 PRINT 3+7`
5. `20 .05*68`
7. `10 X=5: PRINT X`
9. `10 LET X=10`
   `20 LET Y=50.35`
2. `10 ARITH1=1.5378`
4. `20 LET A = 3-5`
6. `10 IF 38 > -5 THEN 50`
8. `20 IF X>0 THEN 50`
10. `200 Y = X*2 - 5`
    `300 PRINT Y,"Y"`

ANSWERS TO TEST YOUR UNDERSTANDING

1. Press Fn-PrtSc simultaneously.
2. a. Type your name, ending the line with ENTER.
   b. Hit Ctrl, Fn, and Home simultaneously.
   c. Hit Caps Lock. Now repeat part a.

3

AN INTRODUCTION TO DOS

DOS is involved in every aspect of PCjr diskette use. It is no exaggeration to say that every time you use your diskettes, you are using DOS. In this chapter, we will learn to use some of the most essential DOS commands.

3.1 Files and File Names

The contents of a diskette are broken into units called **files**. For the present, you may think of a file as a collection of characters. For example, the characters found in Section 3.1 of this book, when stored on diskette, might comprise one file. (In fact, as this book was being written, the sections were stored on diskette in exactly that way.)

A diskette can contain as many as 110 files. However, diskette files may be classified into two broad categories—programs and data files.

Programs A program is a sequence of computer instructions. Throughout this book, we will be discussing programs of one sort or another—programs to compute loan interest, to play TIC TAC TOE, and to print form letters, to mention but a few.

Data files A data file contains data, such as payroll information, personnel data, recipes, train and airline schedules, appointment calendars, and so forth. Programs often make use of data files. This is done by including instructions within the program for reading (or writing) on particular data files. In this way you may, for example, look up appointments and let the computer make decisions based on data in the file.

File Names

Each file is identified by a **file name**. Here are some examples of valid file names:

BASIC.COM
FORMAT.COM
PAYROLL
GAME.001

A file name consists of two parts—the main file name (BASIC, FORMAT, PAYROLL, GAME) and an optional extension (COM, COM, no extension,

001). The main file name may contain as many as eight characters, the extension as many as three. The two parts of the file name are separated by a period.

The following characters are allowed in a file name:
The letters A-Z;
The digits 0-9;
Any of these characters:

! @ # $ % & ( ) - _ { } ' `

Note that a file name CANNOT include any of the following characters:

¦ \ < > , / ? " ~ : + = * ^

The only period allowed in a file name is the one that separates the two parts of the file name. Moreover, a file name cannot have any spaces.

A file name may be spelled with either upper- or lowercase letters. However, DOS will convert the file name into uppercase. So, for example, the file names:

```
JOHN      John
```

refer to the same file.

TEST YOUR UNDERSTANDING 1 (answers on page 39)
What is wrong with the following file names?
(a) ALICE 01 (b) #2324/1 (c) alphabetical

A particular diskette can have only one file with a particular name. However, there is nothing wrong with using the same file name to refer to different files on **different** diskettes.

When you name a file, choose a name that somehow suggests the contents. For example, if you generate a monthly payroll file, you can name the various monthly files:

```
PAYROLL.JAN, PAYROLL.FEB, PAYROLL.MAR,...
```

and so forth.

The Directory

Each diskette has a directory that lists the name of each file on the diskette as well as some descriptive information about the file. Let's analyze one directory entry, the one for the file named DISKCOPY.COM. Actually this file is a program, and one that we've already used; namely, the program for copying the contents of one diskette onto another. We used this program to back up the master DOS diskette. The directory entry for DISKCOPY.COM reads:

```
DISKCOPY COM 2444 3-08-83 12:00p
```

The first two parts of the entry, namely DISKCOPY and COM, give the file name and the extension. The next part of the entry gives the size of the file in bytes. DISKCOPY.COM is 2444 bytes long. The final two parts of the directory entry give the date and time the file was last changed. In this example, DISKCOPY.COM was last altered on 3-08-83 at 12:00 pm.

DOS maintains the directory automatically. Each time a file is added or changed, DOS makes the appropriate changes in the directory, so that it always accurately reflects the contents of the diskette.

You may examine the directory of a diskette using the DOS command DIR. For example, to examine the directory of the disk currently in the diskette drive, you would type:

```
DIR
```

and press ENTER. The directory of drive A: then will be displayed on the screen.

TEST YOUR UNDERSTANDING 2 (answer on page 39)
Display the directory of your DOS diskette.

Exercises (answers on page 349)

Which of the following file names are valid? If invalid, tell why.

1. `SALLY.001`
2. `EXAMPLE.TXT`
3. `E>`
4. `S:001`
5. `#$%&{ }`
6. `A.B.C`
7. `ACCOUNT.0123`
8. `DEMONSTRATION.823`

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. Illegal space
   b. Illegal character (/)
   c. Too many characters

2: Place the DOS diskette in the current drive, type:
   `DIR`
   and press ENTER.

3.2 File Specifications

As mentioned in the preceding section, two different files may have the same name as long as they are on separate diskettes. But what happens when you

put one of the diskettes in drive A: and one in drive B:? Clearly, there is potential for dangerous confusion here. In order to specify a file without confusion, you must give, in addition to the file name, the location of the file.

PC Device Names

DOS specifies the various components of the PC using the following abbreviations:

A:, B:, C:, D: disk drives
CON: the console (monitor and keyboard)
SCRN: the screen
LPT1: printer #1
LPT2: printer #2
COM1: communications adapter #1
COM2: internal modem

Note that DOS 2.1 allows for four diskette drives even though the PCjr has only one. This is because the same version of DOS 2.1 as you use on the PCjr is also used on the PC and the PC/XT. On the PCjr, your single drive is the one named A:. However, to clarify certain commands that ordinarily would involve two drives, your single drive will be considered as drive B:. For example, if you wish to copy a file from one diskette to another, you can specify a copy operation from drive A: to drive B:. While DOS is reading the file from your source diskette, the drive is called A:. When DOS must write the copy of the file, the diskette drive will become drive B:. You will be directed to insert the target diskette into drive B:. After the copy operation is complete, the identity of the drive once again becomes A:.

Note also that PCjr systems will have only one printer attached. However, DOS allows for expansion.

File Specifications

The combination of a device name and a file name is called a **file specification**. Here are some examples:

A:ACCOUNTING
COM1:XYZ.01
LPT1:PRINT.2

Each file has an associated file specification that tells the location of the file and the file name. The file specification contains enough information to find the file without any problems.

In order for a file specification to be valid, both the device name and the file name must be valid.

Wild Card Characters

The characters * and ? in a file name have a special meaning for DOS.

The character * may be used as either the main file name or the extension. The portion of the file name replaced by * may be anything at all. For example, consider this file name:

```
*.COM
```

It stands for any file with the extension COM. Similarly, the file name:

```
WS.*
```

stands for any file that has as its main name WS. Finally, the file name:

```
*.*
```

stands for any file.

Such "ambiguous" file names can shorten various DOS commands. For example, we may copy a file from one diskette to another using the COPY command. (See Section 4.) To copy all files from the diskette in drive A: to the diskette in drive B:, we give the command:

```
COPY A:*.* B:
```

Similarly, to copy all files on the diskette in drive A: having an extension COM onto the diskette in drive B:, we use the command:

```
COPY A:*.COM B:
```

The character ? in a file name allows a single character to be ambiguous. For example, consider this file name:

```
EXAMPLE.00?
```

The third letter in the extension may be anything. Similarly, consider this file name:

```
NIG??.000
```

The main file name has five letters, begins with NIG, and the last two letters of the main file name may be anything.

File names using * and ? may be used in file specifications. For example, A:B*.TXT refers to any file on A: whose name begins with B and has the extension TXT.

Exercises (answers on page 349)

Write file specifications for the following files:

1. ALICE.3 on COM1:
2. MESSAGE on LPT1:

3. Any file on the diskette in drive A:.
4. Any file on the current drive.
5. A file on drive A: in which the main file name has only one letter.
6. A file on drive A: in which the main file name is RALPH and the extension may be anything.

3.3 Executing Commands and Programs

DOS has many commands that perform "housekeeping" functions for the PC. For example, we have already met the command DISKCOPY that lets you copy the contents of one diskette onto another, and the DIR command, which displays a diskette directory. In the rest of this chapter, we'll discuss the various DOS commands and how to use them. However, let's first make some general comments about DOS commands.

To execute a DOS command, the DOS prompt must be displayed. Then do the following:

1. Type the DOS command.
2. Press ENTER.

DOS will execute the command. When execution is complete, DOS will redisplay the DOS prompt.

The above procedure already was explained in our discussions of the DOS commands DIR and DISKCOPY.

Before you press ENTER, you may use the BACKSPACE key to correct mistakes. If you make an error in a command, DOS is quite tolerant. For example, give the command XXXXXX. (There is no such command.) DOS will respond with:

```
Bad Command or File Name
A>
```

You may now give another command.

Until you press the ENTER key, you may cancel the command on the current line by pressing the ESC key. This puts a \ on the current line (indicating that it is cancelled), moves the cursor to the next line and allows you to retype the command.

Here's another type of error that can occur. Remove the DOS diskette and type the command DISKCOPY A: B:. DOS will attempt to read the DISKCOPY program from the nonexistent diskette. After a few seconds, DOS will respond with the prompt:

```
Error reading drive A:
Repeat(R), Ignore(I), or Abort(A)
```

Type R to repeat the command (presumably after you have replaced the diskette), I to ignore the error (in this case, you will generate the same error message), or A (in which case the command will be canceled and the DOS prompt redisplayed). Note that in typing R, I, or A, you don't need to press ENTER.

Internal versus External DOS Commands

We already have noted that the command DISKCOPY is contained in the file DISKCOPY.COM on the DOS diskette. However, if you inspect the directory of the DOS diskette, you won't find a file named anything like DIR.COM. Where does DOS obtain the program corresponding to the command DIR ? The answer is in the way DOS works.

When you start your computer system, you read part of DOS into RAM. This portion of DOS stays in RAM throughout your session with the computer. The most important DOS commands are contained in this portion of DOS so that they can be available without getting them from the diskette. Such commands are called **internal commands**, and DIR is an example. You may remove the DOS diskette from the current drive and still have the internal DOS commands available.

It would be nice to have all DOS commands in RAM all the time. For one thing, they would execute more quickly. However, this gain must be balanced against the permanent decrease in the amount of RAM. Any decrease in RAM would lower the allowable size of application programs. As a compromise, the least frequently used DOS commands are stored on diskette. These commands are called **external commands**. When you request an external command, the corresponding diskette file is read into memory and executed. However, upon completion, the memory is made available for the next program or command.

TEST YOUR UNDERSTANDING 1 (answer on page 43)
Remove the DOS diskette from drive A: . Give the command:

```
DISKCOPY A: B:
```

What happens? Why?

Running Programs Under DOS

We have described the procedure for executing DOS commands. However, the same procedure may be used for programs. Many programs you buy will be stored in files with the extensions COM or EXE. To run such programs, just type the file name without the extension and press ENTER. For example, the BASIC language is one of the programs on the DOS diskette, and its file name is BASICA.COM. To run BASIC, type BASICA and press ENTER.

ANSWER TO TEST YOUR UNDERSTANDING

1: DOS reports an error reading drive A:, since it cannot find the file DISKCOPY.COM to read.

3.4 The COPY Command

You may move a file from one place to another within the computer using the COPY command. As you might guess, this command is very important. Here is how to use it.

1. Obtain the DOS prompt A>. (If you are in BASIC, type SYSTEM and press ENTER to obtain this prompt.)
2. To copy file specification <filespec1> to file specification <filespec2>, type:

```
COPY <filespec1> <filespec2>
```

   and press ENTER. (Note that there is a space between the two file specifications.)

For example, to copy A:BASICA.COM (the copy of the BASIC language BASICA.COM found on the diskette in drive A:) to drive B:, just type:

```
COPY A:BASICA.COM B:BASICA.COM
```

and press ENTER. The computer will make a copy of BASICA.COM on the diskette in drive B:.

Actually, if you want to leave the file name the same in the copy, you may include only the device name in the second file specification. For example, the above copying operation also could be accomplished by typing:

```
COPY A:BASICA.COM B:
```

followed by ENTER.

Creating a Diskette File

We can use the COPY command to copy from the keyboard (device name CON:) directly to a diskette file. Here's how. Sit down at your computer and obtain the DOS prompt A>. Type:

```
COPY CON: A:TEST
```

and press ENTER.

We have just told DOS that we want to copy a file from the console (keyboard) to drive A: and give the resulting file the name TEST. Now type:

```
This is a test.
We are creating the file TEST on drive A:.
```

End each line with ENTER. Note that the above lines are displayed on the screen. Moreover, DOS temporarily stores input lines in RAM. To indicate that we are done inputting data, press function key F6, followed by ENTER. DOS will copy the input lines from RAM to a diskette file, as you requested. You will

see the drive light on drive A: go on. This means that the writing operation is in progress. The computer will respond with the message:

```
1 File(s) copied
```

You have just created the file TEST. If you are not convinced, list the directory of drive A: by typing:

```
DIR A:
```

followed by ENTER. Among the data appearing on the screen will be a line in this form:

```
TEST        62       2/25/83     11:15a
```

This **directory entry** tells you that the name of the file is TEST, that it contains 62 bytes (62 characters, counting spaces, ENTERs, and so forth). The file was created on 2/25/83 at 11:15 am. (The computer will compute the date and time from the data you specified when you turned the computer on.)

If you still are not convinced that you have created a file, let's copy the file back to the screen. Type:

```
COPY TEST CON:
```

and press ENTER. We have just requested that DOS copy TEST from the current drive (A:) to the console. (To the computer's way of thinking, the console consists of both the keyboard and the screen.) Note that the contents of the file will be displayed on the screen.

Finally, let's copy the file TEST to the printer with the command:

```
COPY TEST LPT1:
```

followed by ENTER. (Before giving the command, check that the printer is on.) The printer will print the contents of the file.

TEST YOUR UNDERSTANDING 1 (answer on page 46)
Create a file TEST2 on drive B: containing the following data:
This line is part of TEST2 on drive B:

TEST YOUR UNDERSTANDING 2 (answer on page 46)
Redisplay TEST2 on the screen.

Using Wild Card Characters with COPY

The wild card characters ? and * are very useful in describing COPY operations. Recall that the character * replaces any sequence of characters within a

main file name or an extension. For example, the file name *.001 refers to all files with an extension of 001. Some examples of file names that qualify are:

```
JANE.001        HOWARD.001        MONEY.001        A.001
```

A command of the form:

```
COPY A:*.001 B:
```

will copy all files on A: with extension 001 onto B:.

To copy all the files on A: to B:, you may use this command:

```
COPY A:*.* B:
```

Recall that the wild card character ? stands for a single character. Thus, for example, the file name ??ME.001 can stand for the file names FAME.001 and NAME.001, as well as LAME.001.

TEST YOUR UNDERSTANDING 3 (answer on page 46)
Write a command that copies all files on B: with an extension BAS to A:

Exercises (answers on page 349)

Write DOS commands to:

1. Print B:TEST.
2. Copy all files with the extension COM from A: to B: .
3. Display A:TEST.
4. Copy A:TEST to B: with the new name TEST3.
5. Copy from A: to B: all files whose file names begin with D and where the main file name has eight characters.

ANSWERS TO TEST YOUR UNDERSTANDING

1: Obtain the DOS prompt A> . Type the line followed by ENTER. Press F6 followed by ENTER.
2: Obtain the DOS prompt. Type COPY B:TEST2 CON:
3: COPY B:*.BAS A:

3.5 COPYing and FORMATting Diskettes

In Chapter 2 we made several copies of the DOS master diskette. However, this diskette, important as it is, is not the only diskette we will need. Indeed, the DOS diskette has very little unused space. We need a diskette with plenty of room to write our own programs and data files. In this section, we'll learn to prepare such diskettes.

Formatting a Diskette

When DOS writes on a diskette, it does so in an orderly fashion. Data is written in circular rings called **tracks**. (See Figure 3-1.) Each track is divided into a number of sectors (eight or nine depending on the option you choose when creating the tracks). (See Figure 3-2.)

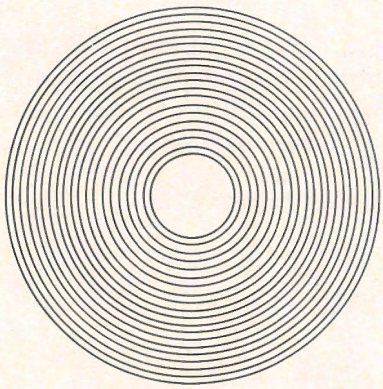


Figure 3-1. The tracks of a diskette.

In order for DOS to write on a diskette, the track and sector boundaries must be written on the diskette. The IBM PC uses **soft-sectored diskettes**, which means that the tracks and sector boundaries are not prerecorded at the factory. Rather, it is *your* job to prepare a diskette for use by first writing these boundaries on it. This task is called **formatting** and is carried out by the DOS command FORMAT.

The FORMAT command is an external command. To use it, you must start with the DOS supplementary diskette in the current drive. Type:

```
FORMAT <drive>
```

Here <drive> is the name of the drive that will contain the diskette to be formatted. For example, to format a diskette in drive A:, you would type:

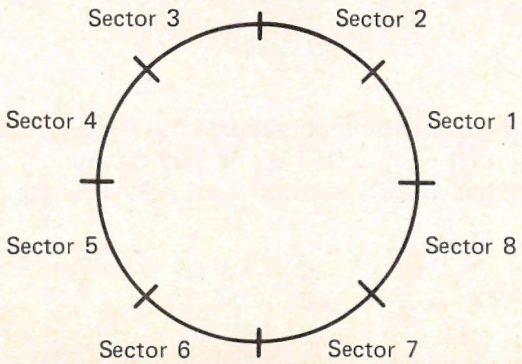


Figure 3-2. The sectors of a diskette track.

```
FORMAT A:
```

DOS will respond with the prompt:

```
Insert new diskette for drive A:
and strike any key when ready
```

Place a blank diskette into drive A:. (If the DOS diskette is in drive A:, don't worry you may remove it since the FORMAT program is already in RAM at this point.) Press any key. The computer will proceed to format the diskette. Eventually, the display will look something like this:

```
Formatting...Format Complete

362496 bytes on diskette
362496 bytes available


Format another (Y/N)?
```

At this point, you may answer the question with Y(=YES) to format another diskette or N(=NO), in which case DOS will terminate the FORMAT operation, redisplay the DOS prompt, and await your next command.

The above procedure was designed so that you can format diskettes one after another. It's a good idea to format an entire box of diskettes when you first buy it. By doing this, you know that all the blank diskettes you have lying around are ready for use.

The numbers displayed in your final FORMAT prompt may be different from those above. The FORMAT command automatically will format your diskette as a double-sided diskette. If, however, you are formatting a diskette on a double-sided drive but wish to then use it on a single-sided drive (say on a PC), you must instruct DOS to format the diskette as single-sided. This is done using the /1 option. For example, to format the diskette in drive A: as a single-sided diskette, use the command:

```
FORMAT A: /1
```

If you are formatting a diskette that might be used on an IBM PC using an early version of DOS (either DOS 1.0, 1.05 or 1.1) then you'll need to format the diskette so that it has only 8 sectors per track. This can be done with the command:

```
FORMAT A: /8
```

You may use the /1 and /8 options together to produce a diskette that is single-sided, with 8 sectors per track. In any case, don't be concerned about

these alternate diskette formats. They will run on your PCjr without any problems. The only effect will be on the number of bytes of storage available on the formatted diskette.

NOTE: You may reformat a diskette that already has been formatted. This erases all data on the diskette. (This is a sure way of destroying sensitive information you don't want lying around.)

TEST YOUR UNDERSTANDING 1
Format a blank diskette.

The number of bytes available usually will be the same as the number of bytes on the diskette. Occasionally, a diskette will contain microscopic flaws that prevent DOS from formatting some sectors. DOS hides these sectors in an invisible file called BADTRACK. You never need to worry about these sectors being used in one of your files and ruining your data. However, if any sectors are placed in BADTRACK, the number of bytes available on the diskette is reduced.

The diskettes produced by the above procedure are totally blank. In particular, they do not contain the DOS files necessary to start the computer. You may include the DOS internal commands on the formatted diskette by using this command:

```
FORMAT A: /S
```

A diskette produced by this command may be used to start the computer. The DOS internal commands occupy a rather small portion of the diskette. Therefore, most of the diskette is available for your use. When you format a diskette with the /S option, the final display looks something like this:

```
 Formatting...Format Complete

362496 bytes on diskette
 39936 bytes used by system
322560 bytes available


Format another (Y/N)?
```

TEST YOUR UNDERSTANDING 2 (answer on page 50)
Format a blank diskette with the /S option.

a. Use this diskette to restart the computer.
b. Display the directory of A: . Can you explain what you see?

In Chapter 2, we learned to copy a diskette using the DISKCOPY command. We used this command to make a copy of the master DOS diskette. However, we made no mention of FORMATting in that discussion. The reason is that the DISKCOPY command automatically formats the diskette onto which it is copying (the **target diskette**). This formatting is performed only if necessary.

ANSWERS TO TEST YOUR UNDERSTANDING

2: a. Place the formatted diskette in drive A: and press CTRL-ALT-DEL simultaneously.

b. The only file in the directory is COMMAND.COM, which occupies 17664 bytes (in DOS 2.10). The system occupies 39936 bytes. The remaining bytes are contained in the main DOS files, called IBMBIOS.COM and IBMDOS.COM, which are invisible as far as the directory is concerned.

3.6 Other DOS Internal Commands

This section gives a brief survey of the most commonly used DOS internal commands. Remember that these commands may be used whenever the DOS prompt is displayed. They do not require any information from the DOS diskette.

ERASE allows you to erase a file. For example, to erase the file EXAMPLE.TXT on the diskette in drive A:, you could use the command:

```
ERASE A:EXAMPLE.TXT
```

If the drive designation is omitted, then the current drive is assumed. For example:

```
ERASE EXAMPLE.TXT
```

will erase EXAMPLE.TXT on the diskette in the current drive. The erase command may be used with the wild card characters * and ?. For example, to erase all files on the diskette in drive A:, use the command:

```
ERASE A:*.*
```

TEST YOUR UNDERSTANDING 1 (answer on page 52)

Write a command that erases all files on the current drive with a five-character main name and an extension of BAT.

RENAME allows you to rename a file. For example, to rename A:OLDFILE with the name NEWFILE, you could use the command:

```
RENAME A:OLDFILE NEWFILE
```

Note that the current file name comes first and then the new file name. If you do not give a drive designation with the current file name, then the current drive is assumed.

TEST YOUR UNDERSTANDING 2 (answer on page 52)
Write a command which renames A:TEST.COM to A:T.COM .

DATE allows you to set the date. For example, to set the date to 4-12-83, you could use the command:

```
DATE 4-12-83
```

TEST YOUR UNDERSTANDING 3 (answer on page 52)
Write a command which sets the date to Dec. 12, 1984.

TIME allows you to set the time. For example, to set the time to 1:04:00 pm, you could use the command:

```
TIME 13:04:00
```

TEST YOUR UNDERSTANDING 4 (answer on page 52)
Write a command to set the time to 12:00:00 am.

TYPE allows you to display the contents of a file. For example, to display the contents of the file A:TEST1, you could use the command:

```
TYPE A:TEST1
```

If you try to display a program, it usually will look like a bunch of gibberish. Program files are designed for the convenience of the computer, not for humans. However, a text file will be displayed in readable form.

To obtain a written copy of a file, first press Ctrl-PrtSc. Then give the TYPE command. The file will be displayed on the screen and also printed on your printer.

COMP allows you to compare two files to determine whether they are identical. For example, suppose that we wish to compare FILE1 on the diskette in drive A: with FILE2 on the diskette in drive B:. Give the command:

```
COMP A:FILE1 B:FILE2
```

This command can be used to check on the results of a COPY operation to determine whether the copy is identical to the original. If the files to be compared are on different diskettes, then DOS will prompt you to remove the diskette with the first file and insert the diskette containing the second.

ANSWERS TO TEST YOUR UNDERSTANDING

1: `ERASE ?????.BAT`
2: `RENAME A:TEST.COM A:T.COM`
3: `DATE 12-12-84`
4: `TIME 00:00:00`

3.7 Other DOS External Commands

In this section, we summarize some of the most commonly used DOS external commands. Note that, in order to use any of these commands, DOS must obtain the appropriate program from the DOS diskette.

DISKCOMP allows you to compare the contents of two diskettes, byte by byte. For example, to compare the diskettes in drives A: and B:, you could use the command:

```
DISKCOMP A: B:
```

You also use this command for diskette comparison, even though you don't have a drive B:. DOS will prompt you to swap the diskettes in your single drive so that a comparison can be made.

CHKDSK allows you to check on the number of bytes remaining on a diskette. It also performs a check to determine if any inconsistencies exist in the way the files are stored. To perform a CHKDSK operation on the diskette in drive A:, you could use the command:

```
CHKDSK A:
```

The result of this command is a display of the form:

```
362496 bytes total disk space
 22272 bytes in 2 hidden files
 45455 in 4 user files
294769 bytes available on disk
131072 bytes total memory
118321 bytes free
```

As usual, your numbers may vary, depending on your system, how your diskette has been formatted, and so forth.

You should execute a CHKDSK every so often for each of your diskettes, to assure the integrity of your files and to determine the space remaining on the diskette.

TEST YOUR UNDERSTANDING 1 (answer on page 53)
Suppose that the DOS diskette is in drive B: and that drive A: is the current drive. Write a command for performing CHKDSK on the diskette in drive A: .

ANSWER TO TEST YOUR UNDERSTANDING
1: B:CHKDSK A:

3.8 Creating Your Own DOS Commands—Batch Files

In the preceding sections, we learned about the most useful DOS commands. Most often, you will execute DOS commands by typing them directly from the keyboard, as described earlier in the chapter. However, in many applications, it is necessary to execute the same sequence of DOS commands repeatedly. For example, consider the following situation.

Suppose that you have a diskette containing four files, named ACCOUNTS.MAY, PROFIT.MAY, PAYABLE.MAY, and SALES.MAY. Your business is computerized and every one of your 10 managers has an IBM PC. Rather than distribute paper copies of the contents of the files, you wish to send each manager a copy of the files on diskette.

A simple solution would be to use DISKCOPY to make 10 copies of the diskette containing the files. Suppose, however, that your diskette also contains some sensitive information that you do not wish to circulate. In this case, you may prepare the duplicate diskettes by copying the files one at a time. This may be done using the COPY command. Here are the DOS commands required to prepare one duplicate diskette, starting from an unformatted diskette:

```
FORMAT B: /S
COPY A:ACCOUNTS.MAY B:
COPY A:PROFIT.MAY B:
COPY A:PAYABLE.MAY B:
COPY A:SALES.MAY B:
```

Assume that your files are contained on the same diskette as FORMAT.COM and that this diskette is in drive A:. The duplicate diskette is in drive B:.

It is possible to prepare the 10 duplicates by typing these commands in manually. But what a chore! And it is easy to make a mistake in typing, especially as the afternoon draws to a close. Fortunately, there is a much better way to proceed: Use a batch file.

A **batch file** is a diskette file consisting of a list of DOS commands. A batch file must have a file name with the extension BAT. In our case, let's name the batch file C.BAT and store it on the diskette in drive A:. To create the batch file, use the COPY command. Type:

```
COPY CON: A:C.BAT
```

and press ENTER. Now type in the DOS commands exactly as they appear in the above list. At the end of each line, press ENTER. After the last line, press function key F6 and then ENTER. DOS will respond with the message:

```
1 file(s) copied
```

The file C.BAT is now on the diskette in drive A:.

To execute the list of DOS commands, type the letter C and press ENTER. (It is just as if we created a new DOS command with the name C.) DOS then will search the current diskette (A:), find the batch file, and execute the various commands, in the order specified.

TEST YOUR UNDERSTANDING 1 (answer on page 56)
Modify the above list of DOS commands so that they include a check that the copies of the files are identical to the originals.

Now our copying job is cut down to size:

1. Insert a blank diskette into drive B:.
2. Type C and press ENTER.
3. Wait for the commands to be executed.
4. Repeat operations 1-3 until all 10 copies are made.

The AUTOEXEC.BAT File

The AUTOEXEC.BAT file is a batch file that is executed automatically whenever DOS is started. If a diskette contains a file with the name AUTOEXEC.BAT, then it is executed on DOS startup without any operator action. For example, suppose that you want your PCjr to start BASIC automatically whenever DOS is started. Just create a diskette file called AUTOEXEC.BAT containing the command:

```
BASIC
```

Note that you may have only one AUTOEXEC.BAT file on a given diskette. On the other hand, you may have many batch files.

TEST YOUR UNDERSTANDING 2 (answer on page 56)
Modify your DOS diskette so that BASIC is started whenever you start DOS.

The AUTOEXEC.BAT file may be used for some clever purposes. For example, let's return to our example of a company with 10 managers. Suppose that you wish to include a covering memo that reads:

```
TO:MANAGERS
HERE ARE THE STATEMENTS FOR MAY.
WE'LL MEET TO DISCUSS THEM ON 6/4
AT 5:30 pm.
          JR
```

Here is how the message can be displayed automatically:

1. Create a file on your diskette that contains the text of the message. Call the file MSSG.
2. Create a file AUTOEXEC.BAT that contains the DOS command:

```
TYPE MSSG
```

3. Modify the batch file C.BAT so that it copies MSSG and AUTOEXEC.BAT onto each of the 10 copies.

Each manager will start his or her PCjr using a duplicate diskette. The AUTOEXEC.BAT file will cause the file MSSG to be displayed on the screen.

Parameters

Let's stick with our fictitious company. Suppose that the 10 diskettes are to be prepared and sent every month. The file names are always the same, but the month abbreviations, as given in the file name extensions, vary. You could prepare a new batch file C.BAT every month. However, there is a better way. Designate the month abbreviation by the symbol %1. (% is an abbreviation for **parameter** and 1 is the number of the parameter.) The commands of the batch file then are written:

```
FORMAT B:/S
COPY A:ACCOUNTS.%1 B:
COPY A:PROFIT.%1 B:
COPY A:PAYABLE.%1 B:
COPY A:SALES.%1 B:
```

For the month of MAY, we would give the batch command:

```
C MAY
```

For the month of JUNE, give the batch command:

```
C JUN
```

And so forth.

You may use up to nine parameters %1, %2, ..., %9. You specify the values of these parameters when you give the batch command, with consecutive parameter values separated by spaces. For example, if a batch file D used the two parameters %1 and %2, then to execute the batch file with %1 = JAN and %2 = FEB, we would use the command:

```
D JAN FEB
```

ANSWERS TO TEST YOUR UNDERSTANDING

1: Add the DOS commands:

```
COMP A:ACCOUNTS.MAY B:
COMP A:PROFIT.MAY B:
COMP A:PAYABLE.MAY B:
COMP A:SALES.MAY B:
```

2: Add the file AUTOEXEC.BAT, which consists of the single DOS command:

```
BASIC
```

II

AN INTRODUCTION TO PCjr BASIC

4

GETTING STARTED IN BASIC

4.1 Beginning BASIC

In Chapter 2, we learned to manipulate the keyboard and display screen of the IBM PCjr. Now we'll learn how to communicate instructions to the computer.

Just as humans use languages to communicate with one another, computers use languages to communicate with other electronic devices (such as printers), human operators, and even other computers. There are hundreds of computer languages in use today. And your IBM PCjr is capable of 'speaking" quite a few of them. Among these languages, BASIC is both versatile and very easy to learn. It was developed especially for computer novices by John Kemeny and Thomas Kurtz at Dartmouth College. In the next few chapters, I will concentrate on teaching the fundamentals of BASIC. In the process, you will learn a great deal about the way in which a computer may be used to solve problems.

The PCjr actually comes with two different versions of the BASIC language. The least powerful version of BASIC is called **Cassette BASIC**. This is the BASIC version that is supplied with all PCjrs and it is stored in ROM. You may purchase the more powerful language called **Cartridge BASIC**. This version includes all the commands of Cassette BASIC plus additional commands that allow you to make use of your diskette drives and advanced graphics capabilities.

4.2 BASIC Statements in Immediate Mode

Assume that you have loaded BASIC and have obtained the BASIC prompt. Now give the computer some instructions in BASIC. Type:

```
PRINT 3+2
```

and press ENTER. The computer will immediately fire back the answer:

```
5
Ok
```

The Ok prompt indicates that BASIC is awaiting another instruction. Type:

```
CLS
```

and press ENTER. The screen will be erased and the cursor positioned in the upper left corner (the so-called **Home** position).

Now try some other BASIC instructions. Assuming that you have a color monitor, type:

```
SCREEN 1,0
```

and press ENTER. This instruction tells BASIC to enter medium- resolution graphics mode (SCREEN 1). The "0" portion of the command enables color. Next, set the background and text colors with the statement:

```
COLOR 1,2 <ENTER>
```

(From now on, I will write <ENTER> to mean "and press the ENTER key".) Notice that the screen turns blue and the Ok prompt is displayed in yellow.

TEST YOUR UNDERSTANDING 1 Try out the statement:

```
COLOR 2,4 <ENTER>
```

What does it do?

TEST YOUR UNDERSTANDING 2 Try out the statements:

```
COLOR n <ENTER>
```

where n=0,1,2,3,4,... How many different background colors are possible?

BASIC is equipped with an incredible array of statements that perform a variety of tasks. As just a hint of things to come, try out a few graphics and music statements.

Type the statement:

```
PLAY "CDEF" <ENTER>
```

The computer plays four notes. These notes are C, D, E, and F. PCjr BASIC has a Music Macro Language* that enables you to play quite complicated musical compositions in up to three-part harmony. To get a taste of the possibilities, try this command:

*Registered trademark of Microsoft Corporation.

```
SOUND ON <ENTER>
PLAY "C","E","G" <ENTER>
```

If your PCjr is connected to a display with an extended speaker (such as your home television set), you will hear a C major chord. (The chord consisting of the notes C, E, and G.)

Next, clear the screen and type the statement:

```
CIRCLE (100,100),75
```

BASIC will draw a circle as in Figure 4-1.

Actually, BASIC has a large repertoire of graphics statements that you will learn in due course.

The exercises below will give you an opportunity to explore a few more of BASIC's instructions.

Exercises (Answers on page 349)

Determine the effect of the following BASIC instructions:

1. `LOCATE 3,4`
2. `LOCATE 12,8`
3. `LOCATE x,y`
4. `PRINT 3*6`
5. `PRINT 2*9`
6. `PRINT X*Y`
7. `PRINT 5 MOD 3`

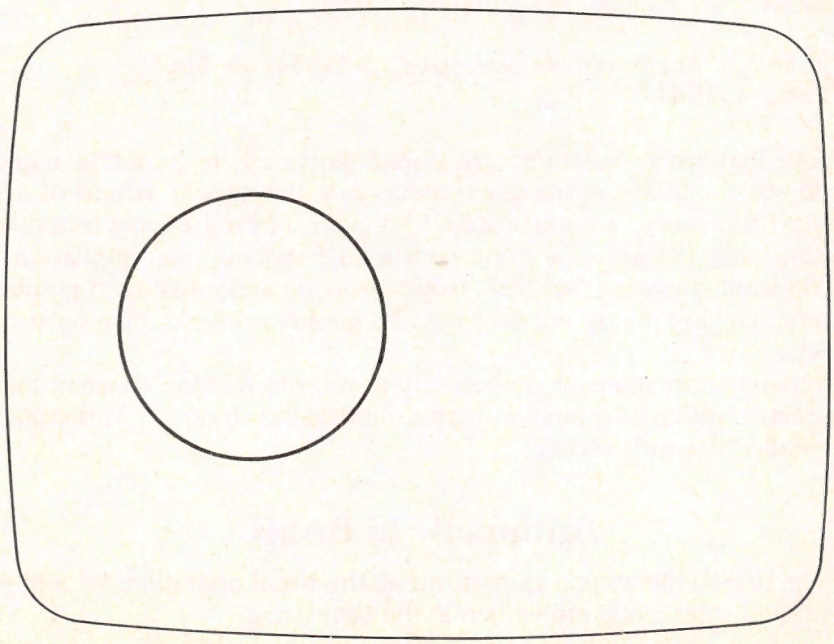


Figure 4-1. A circle centered at (100,100) with radius 75.

```
8.  PRINT 6 MOD 2
9.  PRINT 7 MOD 5
10. PRINT X MOD Y
```

4.3 BASIC Constants and Arithmetic

In learning to use a language, you first must learn the alphabet of the language. Next, you must learn the vocabulary of the language. Finally, you must study the way in which words are put together into sentences. In learning the BASIC language, I will follow the progression just described. In Chapter 2, you learned about the characters of the IBM PCjr keyboard. These characters are the alphabet of BASIC. Next, you'll learn some vocabulary words. The simplest "words" are the so-called constants.

BASIC Constants

BASIC allows you to manipulate numbers and text. The rules for manipulating numeric data differ from those for handling text, however. In BASIC, we distinguish between these two types of data as follows: a **numeric constant** is a number, and a **string constant** is a sequence of keyboard characters that may include letters, numbers, or any other keyboard symbols. The following are examples of numeric constants:

```
5, -2, 3.145, 23456, 456.7834, 27134000000000
```

The following are examples of string constants:

```
"John", "Accounts Receivable", "$234.45 Due",
"Dec. 4,1981"
```

Note that string constants are always enclosed in quotation marks. In order to avoid vagueness, quotation marks may not appear as part of a string constant. (In practice, an apostrophe (') should be used as a substitute for a quotation mark (") within a string constant.) Numbers may appear within a string constant, such as "$45.30". However, you cannot use such numbers in arithmetic. Only numbers not enclosed by quotation marks may be used for arithmetic.

In many applications, it is necessary to refer to a string constant that has no characters within its quotation marks, namely the string "". This string constant is called the **null string**.

Arithmetic in BASIC

PCjr BASIC allows you to perform all the usual operations of arithmetic. Addition and subtraction are written in the usual way:

```
5 + 4, 9 - 8.
```

Multiplication, however, is typed using the symbol * , which shares the '8" key. As an example, the product of 5 and 3 is typed:

```
5*3 .
```

Division is typed using /. For example, 8.2 divided by 15 is typed:

```
8.2/15
```

All elementary arithmetic operations (addition, subtraction, multiplication, division) are carried out to seven decimal places. So, for example, the result of the statement:

```
PRINT 8.2/15
```

is the display:

```
 .5466666
Ok
```

Example 1. Write a BASIC program to calculate the sum of 54.75, 78.83, and 548.

Solution. The sum is indicated by typing:

```
54.75 + 78.83 + 548
```

The BASIC instruction for printing data on the screen is **PRINT**, so write the program as follows:

```
10 PRINT 54.75 + 78.83 + 548
20 END
```

The Order of Operations and Parentheses

BASIC carries out arithmetic operations in a special order. It scans an expression and carries out all multiplication and division, proceeding in left-to-right order. Then it returns to the left side of the expression and performs addition and subtraction in the same order.

For example, consider the expression:

```
2*3 + 4*5 + 3*3
```

BASIC first scans the expression from left to right and performs all multiplications and divisions in the order in which they are encountered. It simplifies the expression to:

```
6 + 20 + 9
```

BASIC then begins over at the left and performs all addition and subtraction operations in the order encountered. This gives the result:

```
35
```

The order of operations is extremely important. Let's try another example:

```
1 - 3/2*5
```

BASIC first performs the division 3/2. This simplifies the expression to:

```
1 - 1.5*5
```

Next, it performs the multiplication 1.5*5 to obtain:

```
1 - 7.5
```

Finally, it starts from the left again and performs addition and subtraction, to obtain:

```
-6.5
```

Knowing the order of operations helps you to correctly translate familiar arithmetic procedures into computer language. For example, consider the following fraction:

```
5 + 3/2
-------
 5*8
```

According to the rules of arithmetic, you simplify this fraction by first simplifying the numerator and denominator to obtain:

```
6.5
---
40
```

Note that you must perform the operations specified in the numerator and denominator **before** performing the division indicated in the fraction. You may indicate this in BASIC (as in algebra) by using parentheses:

```
(5+3/2)/(5*8)
```

BASIC simplifies an expression by first removing the parentheses. For example, in the above expression, the parentheses (5+3/2) and (5*8) are evaluated first, to give:

```
6.5/40
```

BASIC then performs the division.

TEST YOUR UNDERSTANDING 1

Evaluate the expression: `3*5 - 4*3/2 + 4 - 8/2`

In evaluating parentheses, BASIC uses the same rules stated above: First perform all multiplications and divisions in left-to-right order. Then perform all additions and subtractions in left-to-right order.

What about parentheses within parentheses? Well, you have enough knowledge to figure out what BASIC does. Work out the example:

```
(1+3*(4+5))*(1+4)
```

BASIC looks at the expression and decides it must first evaluate the left-most parenthesis (1+3*(4+5)). When it attempts to evaluate it, however, it encounters a parenthesis within, namely (4+5), which must be evaluated first. So the first simplification is:

```
(1+3*9)(1+4)
```

Now BASIC begins all over. It evaluates the left-most parenthesis to get:

```
28*(1+4)
```

Next, it evaluates the right parenthesis to get:

```
28*5
```

Finally, it performs the multiplication to obtain the answer:

```
140
```

Example 2. What numeric values will BASIC calculate from these expressions?

```
a. (5 + 7)/2          b. 5 + 7/2

c. 5 + 7*3/2          d. (5 + 7*3)/2
```

Solution. a. The computer first applies its rules for the order of calculation to determine the value in the parentheses, namely 12. It then divides 12 by 2 to obtain 6.

b. The computer scans the expression from left to right performing all multiplication and division in the order encountered. First it divides 7 by 2 to obtain

3.5. It then rescans the line and performs all additions and subtractions in order. This gives us:

```
5 + 3.5 = 8.5
```

c. The computer first performs all multiplication and division in order:

```
5 + 10.5
```

Now it performs addition to obtain 15.5.

d. The computer calculates the value of all parentheses first. In this case, it computes 5 + 7*3 = 26. (Note that it does the multiplication first!) Next it rescans the line which now looks like this:

```
26/2
```

It performs the division to obtain 13.

TEST YOUR UNDERSTANDING 2 (answer on page 69)
Calculate `5+3/2+2` and `(5+3)/(2+2)`.

Example 3. Write a BASIC instruction to calculate the quantity

$$\frac{22\text{x}18 + 34\text{x}11 - 12.5\text{x}8.}{27.8 + 42.1}$$

Solution. Here is the instruction:

```
PRINT (22*18 + 34*11 - 12.5*8)/(27.8+42.1)
```

The parentheses in line 10 tell BASIC to calculate the values of the numerator and denominator before doing the division implied by the fraction. First calculate (22*18 + 34*11 - 12.5*8) and (27.8 + 42.1) before performing the division.

TEST YOUR UNDERSTANDING 3 (answer on page 69)
Write BASIC programs to calculate:

a. ((4x3 + 5x8 + 7x9)/(7x9 + 4x3 + 8x7)) x 48.7
b. 27.8 % of (112 + 38 + 42)
c. The average of the numbers 88, 78, 84, 49, 63

Scientific Notation

For certain applications, you may wish to specify your numeric constants in **exponential format** (also called **scientific notation**). This will be espe-

cially helpful in the case of very large and very small numbers. Consider the number 15,300,000,000. It is very inconvenient to type all the zeros, and it can be written as 1.53E10. The 1.53 indicates the first three digits of the number. E10 means that you move the decimal point in the 1.53 to the right 10 places. Similarly, the number -237,000 may be written in the exponential format as −2.37E5. Exponential format also may be used for very small numbers. For example, the number 0.00000000054 may be written in exponential format as 5.4E-10. The -10 indicates that the decimal point in 5.4 is to be moved 10 places to the *left*.

TEST YOUR UNDERSTANDING 4 (answer on page 69)

a. Write these numbers in exponential format: .00048 and −1374.5

b. Write these numbers in decimal format: −9.7E3, 9.7E−3 and −9.7E−3

BASIC can display at most seven significant digits of a number. If you ask it to display a number with more than seven significant digits, BASIC will automatically shift to scientific notation. Thus, for example, the statement:

```
X=123456789:PRINT X
```

will produce the display:

```
1.234568E+09
```

Note that the initial seven digits are obtained by rounding the given 10 digits.

Exponentiation

Suppose that A is a number and N is a positive whole number (this means that N is one of the numbers 1,2,3,4,...). Then A **raised to the Nth power** is the product of A times itself N times. This quantity usually is denoted A^N, and the process of calculating it is called **exponentiation**. For example,

$2^3 = 2*2*2 = 8$, $5^7 = 5*5*5*5*5*5*5 = 78125$.

It is possible to calculate A^N by repeated multiplication. However, if N is large, this can be tiresome to type. BASIC provides a shortcut for typing this function. Exponentiation is denoted by the symbol ^, which is produced by hitting the key with the upward-pointing arrow (this symbol shares the "6" key at the top of the keyboard). For example, 2^3 is denoted 2^3. The operation of exponentiation is done before multiplication and division. This is illustrated in the following example.

Example 5. Determine the value that BASIC assigns to this expression:

```
20*3 - 5*2^3
```

Solution. The exponentiation is performed first to yield:

```
20*3 - 5*8 = 60 - 40
           = 20
```

TEST YOUR UNDERSTANDING 5 (answer on page 69)
Evaluate the following, first manually, and then using an IBM PCjr program.

a. $2^4 \times 3^3$
b. $2^2 \times 3^3 - 12^2/3^2 \times 2$

Integer Division

Recall the days when you first learned division. Your first problems involved dividing one whole number by another. You were taught to express the answer as a quotient and a remainder. For example, the result of dividing 14 by 5 is the quotient 2 and the remainder 4. This type of division may be performed in BASIC using the operations \ and mod. For example:

```
14\5 = 2
```

and

```
14 mod 5 = 4
```

That is, 14\5 equals the (whole number) quotient of 14 divided by 5; 14 mod 5 equals the remainder. The symbol \ is called a **backslash** and should not be confused with the ordinary slash /.

Here is a table showing the order in which \ and mod are performed in relationship to the other operations. The operations that are higher in the list are performed first.

```
^
*, /
\
mod
+, -
```

Consider this expression:

```
5*3\2*2 mod 2
```

The multiplications are performed first, to obtain:

```
15\4 mod 2
```

Next, the \ is performed, to obtain:

```
3 mod 2
```

Finally, this last expression is simplified to obtain:

```
1
```

Exercises (answer on page 350)

Write BASIC programs to calculate the following quantities:

1. 57 + 23 + 48
2. 57.83 x (48.27 − 12.54)
3. 127.86/38
4. $365/.005 + 1.02^5$

Convert the following numbers to exponential format:

5. 23,000,000
6. 175.25
7. −200,000,000
8. .00014
9. −.000000000275
10. 53,420,000,000,000,000

Convert the following numbers in exponential format to standard format:

11. 1.59E5
12. −20.3456E6
13. −7.456E-12
14. 2.39456E-18

Calculate the following quantities:

15. 18\6
16. 17 mod 3
17. 25 mod 2*3
18. (17\4 mod 3)^2
19. (17\4) mod 3^2

ANSWERS TO TEST YOUR UNDERSTANDING

1: 9

2: 8.5 and 2

3: a. PRINT ((4*3 + 5*8 + 7*9)/(7*9 + 4*3 + 8*7))*48.7
   b. PRINT .278*(112+38+42)
   c. PRINT (88+78+84+49+63)/5

4: a. 4.8E−4, −1.3745E3
   b. −9700, .0097, −.0097

5: a. 432
   b. 76

4.4 Running BASIC Programs

Sections 2 and 3 gave examples of several BASIC statements. You told BASIC to execute a statement by typing it and then pressing the ENTER key. This method of executing statements is called the **immediate mode** and is used for executing a single instruction at a time.

In order to make BASIC do anything really complex, it's necessary to string together many instructions (sometimes as many as several thousand). A sequence of instructions is called a **program**. You will learn to write programs that do arithmetic, draw charts, and even play TIC TAC TOE. Before that, however, let's look at one that IBM has prepared especially to demonstrate the power of its computer. (Apologies to readers who don't have a diskette drive. The following discussion is going to exclude you.)

Start the computer, and obtain the BASIC prompt **Ok**, as described in Chapter 2. IBM has included many interesting programs on the PCjr Sampler diskette as well as on the DOS Supplementary diskette.

One of my favorites is on the DOS Supplementary diskette, so remove the DOS diskette and insert the DOS Supplementary diskette into the drive. To obtain a list of the programs on the diskette, type:

```
FILES <ENTER>
```

The names of the programs on the diskette will be displayed on the screen.

One of the most impressive programs is MUSIC. (Note that it is listed under the name MUSIC.BAS. The extension BAS indicates that the program is written in BASIC.) To load the program MUSIC from the diskette into RAM, type:

```
LOAD "MUSIC" <ENTER>
```

(Note the quotation marks.) The diskette drive light will go on, you will hear the drive at work, and the program MUSIC will be loaded into RAM. The drive light then will go out and the drive will stop.

Now let's make the computer perform the instructions in the program. (In computer jargon, we **run the program**.) Type:

```
RUN
```

and press ENTER. The computer draws a piano keyboard on the screen and displays the names of some songs. To play a song, press the key indicated. Why not spend a few minutes enjoying the computer-generated music. Note also how the computer 'animates'' the keyboard by displaying a moving note, which indicates the key being played.

Sooner or later, you will want to interrupt a computer program while it is running. This is done by **simultaneously** pressing the Fn and Break keys. It's a two-handed operation and with good reason. The keys are arranged so that you won't interrupt programs accidentally. To illustrate how you may interrupt a program, run MUSIC and play a song. In the middle of the song, simultane-

ously hit Fn and Break. The program will stop. The screen will display a message like this:

```
Break in line xxxx
Ok
_
```

The line xxxx gives the place in the program at which you stopped the computer. (You'll learn about line numbers in the next section.) The BASIC prompt Ok indicates that BASIC is awaiting another command. Interrupting a program does not erase it from RAM. To run the program again, just type RUN and press ENTER.

Well, enough music for now! Let's end the program. According to the instructions on the screen, you may "EXIT" the program by pressing Esc, a key located on the upper left side of the keyboard. Press this key. Note that the BASIC prompt Ok is displayed, indicating that BASIC is awaiting a command. You probably are curious to see the set of instructions for MUSIC. Nothing could be easier. Type:

```
LIST
```

and press ENTER. You will see the instructions of the program displayed on the screen. Of course, they are going by too quickly to read them. Later, you'll learn how to stop the display where you want or to obtain a written copy on the printer.

TEST YOUR UNDERSTANDING 1

Pick out a program on the PCjr Sampler diskette, load it into memory, and run it.

ANSWER TO TEST YOUR UNDERSTANDING

1. Start from the BASIC prompt. Type LOAD <program name> and press ENTER. Here <program name> is the name of the program you wish to run. Omit the extension BAS. Now type RUN and press ENTER.

4.5 Writing BASIC Programs

You may be intimidated by the number of instructions in the program MUSIC. Don't be. In no time at all, you will be writing programs just as complicated. Take one step at a time and first learn to write some simple BASIC programs.

Assume that you have followed the startup instructions of the Chapter 1 and the computer shows that it is ready to accept further instructions by displaying the BASIC prompt:

```
Ok
```

From this point on, a typical session with your computer might go like this:

1. Type in a program.
2. Locate and correct any errors in the program.
3. Run the program.
4. Obtain the output requested by the program.
5. Either: run the program again, or repeat steps 1-4 for a new program, or end the programming session (turn off the computer and go have lunch).

To fully understand what is involved in these five steps, consider a particular example, namely, a program to add 5 and 7. First, you would type the following instructions:

```
10 PRINT 5 + 7
20 END
```

This sequence of two instructions constitutes a program to calculate 5 + 7.

As you type the program, the computer records your instructions, **but does not carry them out**. (The line numbers 10 and 20 tell BASIC that the instructions are not to be carried out immediately.) As you are typing a program, the computer provides you with an opportunity to change, delete, and correct instruction lines. (More on how to do this later.) Once you are content with your program, tell the computer to run it (that is, to execute the instructions) by typing the command*:

```
RUN
```

The computer will run the program and display the desired answer:

```
12
Ok
```

If you wish the computer to run the program a second time, type **RUN** again.

Running a program does not erase it from RAM. Therefore, if you wish to add instructions to the program or change the program, you may continue typing just as if the **RUN** command had not intervened. For example, if you wish to include in your program the problem of calculating 5 − 7, type the additional line:

```
15 PRINT 5 - 7
```

To see the program currently in memory, type **LIST** (no line number) and then hit the **ENTER** key. The program consists of the following three lines, now displayed on the screen:

---

*Don't forget to follow the command with **ENTER**. The computer will not recognize a line unless it has been sent to it by hitting the **ENTER** key.

```
10 PRINT 5 + 7
15 PRINT 5 - 7
20 END
```

Note how the computer puts line 15 in proper sequence. If you type **RUN** again, the computer will display the two answers:

```
12
-2
```

Note that line numbers need not be consecutive. For example, it is perfectly acceptable to have a program with line numbers 10, 23, 47, 55, and 100. Also note that it is not necessary to type instructions in numeric order. You could type line 20 and then go back and type line 10. The computer will sort out the lines and rearrange them according to increasing number. This feature is especially helpful in case you accidentally omit a line while typing your program.

Here is another important fact about line numbering. If you type two lines with the same line number, the computer erases the first version and remembers the second version. This feature is very useful for correcting errors: If a line has an error, just retype it and press ENTER.

Let's go on to another program by typing the command:

```
NEW
```

This erases the previous program from RAM and prepares the computer to accept a new program. You should always remember the following important fact:

RAM can contain only one program at a time.

TEST YOUR UNDERSTANDING 1 (answers on page 77)

a. Write and type in a BASIC program to calculate 12.1 + 98 + 5.32.
b. Run the program of a.
c. Erase the program of a from RAM.
d. Write a program to calculate 48.75 − 1.674.
e. Type in and run the program of d.

Immediate Mode and Execute Mode

BASIC on the IBM PCjr operates in two distinct modes. In **immediate mode** (also called **command mode**), the computer accepts typed program lines and commands (like **RUN** and **NEW**) used to manipulate programs. The computer identifies a program line by its line number. Program lines are not immediately executed. Rather, they are stored in RAM until you tell the com-

puter what to do with them. On the other hand, commands are executed as soon as they are given.

While BASIC is running a program, it is in **execute mode**.

When you turn the computer on it is automatically in immediate mode, indicated by the presence of the **Ok** prompt on the screen. The **RUN** command puts the computer into execute mode. After the computer finishes running a program, it redisplays the **Ok** prompt indicating that it is back in immediate mode.

Uppercase versus Lowercase and Extra Spaces

The computer is a stern taskmaster! It has a very limited vocabulary (BASIC), and this vocabulary must be used according to very specific rules concerning the order of words, punctuation, and so forth. However, BASIC allows for some freedom of expression. For example, BASIC commands may be typed in capitals, lowercase, or a mixture of the two. Also, any extra spaces are ignored. Thus, BASIC will interpret all of the following instructions as the same:

```
10 PRINT A
10 print a
10 Print A
10 print        A
10     print A
```

Note, however, that BASIC expects spaces in certain places. For example, there must be a space separating PRINT and A in the above command. Otherwise, BASIC will read the command as PRINTA, which is not in its vocabulary!

A Word of Warning

Many people think of a computer as an "electronic brain" that somehow has the power of human thought. This is very far from the truth. The electronics of the computer and the rules of the BASIC language allow it to recognize a very limited vocabulary, and to take various actions based on the data that is given to it. It is very important to realize that the computer does not have "common sense." The computer will attempt to interpret whatever data you input. If what you input is a recognizable command, the computer will perform it. It does not matter that the command makes no sense in a particular context. The computer has no way to make such judgments. It can only do what you instruct it to do. Because of the computer's inflexibility in interpreting commands, you must tell the computer **exactly** what you want it to do. Don't worry about confusing the computer. If you communicate a command in an incorrect form, you won't damage the machine in any way! However, in order to make the machine do our bidding, it is necessary to learn to speak its language precisely.

Printing Words

So far, you have used the **PRINT** statement only to display the answers to numeric problems. However, this instruction is very versatile. It also allows you to display string constants. For example, consider this instruction:

```
10 PRINT "Patient History"
```

During program execution, this statement will create the following display:

```
Patient History
```

In order to display several string constants on the same line, separate them by commas in a single **PRINT** statement. For example, consider the instruction:

```
10 PRINT "AGE", "SEX", "ADDRESS"
```

It will cause three words to be printed as follows:

```
AGE       SEX       ADDRESS
```

Both numeric constants and string constants may be included in a single **PRINT** statement. For example:

```
100 PRINT "AGE", 65.43, 65000
```

Here is how the computer determines the spacing on a line as follows. Each line is divided into **print zones**. In 80-character width, the first five print zones each have 14 spaces and the sixth 10 spaces. In 40-character width in cassette BASIC, there are three print zones, each with 14 characters and the third with 12 characters. In 40-character width in Cartridge BASIC, there are two print zones, each with 14 characters. By placing a comma in a PRINT statement, you are telling the computer to start the next string of text at the beginning of the next print zone. Thus, for example, the four words above begin in columns 1,15,29,43 respectively, assuming an 80-character width. (See Figure 4-2.)

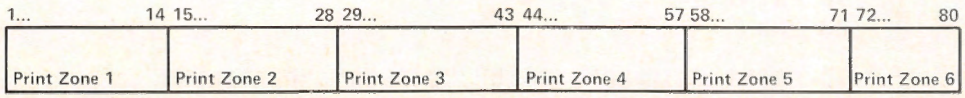


Figure 4-2. Print zones in 80-column mode.

TEST YOUR UNDERSTANDING 2 (answer on page 77)
Write a program to print the following display.

```
                NAME
LAST            FIRST           GRADE
SMITH           JOHN            87
```

TEST YOUR UNDERSTANDING 3 (answer on page 77)
Write a computer program which creates the following display.

```
                        BUDGET-APRIL
FOOD                    387.50
CAR                     475.00
GAS                     123.71
UTILITIES               146.00
ENTERTAINMENT           100.00
                        ------
TOTAL                (Calculate total)
```

Exercises (answers on page 350)

1. Make a table of the first, second, third, and fourth powers of the numbers 2, 3, 4, 5, and 6. Put all first powers in a column, all second powers in another column, and so forth.
2. Mrs. Anita Smith went to her doctor with a broken leg. Her bill consists of $45 for removal of the cast, $35 for therapy, and $5 for drugs. Her major medical policy will pay 80 percent directly to the doctor. Use the computer to prepare an invoice for Mrs. Smith.
3. A school board election is held to elect a representative for a district consisting of Wards 1, 2, 3, and 4. There are three candidates: Mr. Thacker, Ms. Hoving, and Mrs. Weatherby. The tallies by candidate and ward are as follows:

	Ward 1	Ward 2	Ward 3	Ward 4
Thacker	698	732	129	487
Hoving	148	928	246	201
Weatherby	379	1087	148	641

Write a BASIC computer program to calculate the total number of votes achieved by each candidate, as well as the total number of votes cast.
Describe the output from each of these programs:

4.
```
10 PRINT 8*2 - 3*(2^4 - 10)
20 END
```

5.
```
10 PRINT "SILVER","GOLD","COPPER","PLATINUM"
20 PRINT 327,448,1052,2
30 END
```

6.
```
10 PRINT , "GROCERIES","MEATS","DRUGS"
20 PRINT "MON", "1,245","2,348","2,531"
30 PRINT "TUE", " 248","3,459","2,148"
40 END
```

ANSWERS TO TEST YOUR UNDERSTANDING

1: a.
```
10 PRINT 12.1 + 98 + 5.32
20 END
```
b. Type RUN and press ENTER.
c. Type NEW and press ENTER.
d.
```
10 PRINT 48.75 - 1.674
20 END
```
e. Type in the program. Type RUN and press ENTER.

2:
```
10 PRINT ,"NAME"
20 PRINT
30 PRINT "LAST","FIRST","MIDDLE","GRADE"
40 PRINT
50 PRINT "SMITH","JOHN","DAVID",87
60 END
```

3:
```
10 PRINT ," BUDGET-APRIL"
20 PRINT "FOOD",,387.50
30 PRINT "CAR",, 475.00
40 PRINT "GAS",, 123.71
50 PRINT "UTILITIES",, 146.00
60 PRINT "ENTERTAINMENT",, 100.00
70 PRINT ,, "_____"
80 PRINT "TOTAL",
   387.50+475.00+123.71+146.00+100.00
90 END
```

4.6 Giving Names to Numbers and Words

In the examples and exercises of the preceding section, you probably noticed that you were wasting considerable time retyping certain numbers over and over. Not only does this retyping waste time, it also is a likely source of errors. Fortunately, such retyping is unnecessary if you use variables.

A **variable** is a collection of characters used to represent a number. A variable name must begin with a letter and can contain as many as 40 characters. Therefore, you may use variables named PAYROLL, TAX, REFUND, and BALANCE. Note, however, that not every sequence of characters is a legal variable name. You must avoid any sequences of characters that are reserved by BASIC for special meanings. Examples of such words are:

```
IF, ON, OR, TO, THEN, GOTO
```

Once you become familiar with BASIC, it will be second nature to avoid using these and other reserved words as variable names.

A variable name *cannot* begin with a number. For example, 1A is *not* a legal variable name. If you attempt to use a variable name that begins with a number, BASIC will provide an error message.

At any given moment, a variable has a particular value. For example, the variable A might have the value 5 while B might have the value -2.137845. One method for changing the value of a variable is through use of the **LET** statement. The statement

```
10 LET A = 7
```

sets the value of A equal to seven. Any previous value of A is erased.

Once the value of a variable has been set, the variable may be used throughout the program. The computer will insert the appropriate value wherever the variable occurs. For instance, if A has the value 7, then the expression:

```
A + 5
```

is evaluated as 7 + 5 or 12. The expression:

```
3*A - 10
```

is evaluated as 3*7 - 10 = 21 - 10 = 11. The expression 2*A^2 is:

```
2*7^2 = 2*49 = 98.
```

TEST YOUR UNDERSTANDING 1 (answer on page 86)
Suppose that A has the value 4 and B has the value 3. What is the value of the expression A^2/2*B^2 ?

Note the following important fact:

> If you do not specify a value for a variable, BASIC will assign it the value 0.

Here are three useful shortcuts.

Three Shortcuts

1. The word LET is optional. For example, the statement:

```
10 LET A=5
```

may be abbreviated as:

```
10 A=5
```

2. Several statements may be included on one line. To do so, just separate the various statements by colons. In particular, a single line may be used to assign values to several variables. For instance, the instruction:

```
100 LET C = 18: LET D = 23: LET E = 2.718
```

assigns C the value 18, D the value 23, and E the value 2.718. Using shortcut 1, you may write this instruction in the simpler form:

```
100 C=18:D=23:E=2.718
```

3. You may use statements that extend beyond a single line. This is especially useful when assigning values to many variables as in shortcut 2 above. When you reach the end of the physical line (40 or 80 characters wide) just keep on typing. Hit ENTER when you are finished with the material to be included with the current line number. An extended line may contain as many as 255 characters. When an extended line reaches 255 characters, BASIC will automatically terminate it just as if you had pressed ENTER.

Variables in PRINT Statements

Variables also may be used in **PRINT** statements. For example, the statement:

```
10 PRINT A
```

will cause the computer to print the current value of A (in the first print zone , of course!). The statement:

```
20 PRINT A,B,C
```

will result in printing the current values of A, B, and C in print zones 1, 2 and 3, respectively.

TEST YOUR UNDERSTANDING 2 (answer on page 86)
Suppose that A has the value 5. What will be the result of the instruction:

```
10 PRINT A,A^2,2*A^2
```

Example 1. Consider the three numbers 5.71, 3.23, and 4.05. Calculate their sum, their product, and the sum of their squares (i.e., the sum of their second powers; such a sum is often used in statistics.).
Solution. Introduce the variables A, B, and C and set them equal, respectively, to the three numbers. Then compute the desired quantities:

```
10 LET A = 5.71: B = 3.23: C = 4.05
20 PRINT "THE SUM IS", A+B+C
30 PRINT "THE PRODUCT IS", A*B*C
40 PRINT "THE SUM OF SQUARES IS", A^2+B^2+C^2
50 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 86)
Consider the numbers 101,102,103,104,105,and 106. Write a program which calculates the product of the first two, the first three, the first four, the first five, and then all six numbers.

The following mental imagery is often helpful in understanding how BASIC handles variables. When BASIC first encounters a variable, say A, it sets up a box (actually a memory location) that it labels "A". (See Figure 4-3.) It stores the current value of A in this box. When you request a change in the value of A, the computer throws out the current contents of the box and inserts the new value.

Note that the value of a variable need not remain the same throughout a program. At any point in the program, you may change the value of a variable (with a **LET** statement, for example). If a program is called on to evaluate an expression involving a variable, it always will use the **current** value of the variable, ignoring any previous values the variable may have had at earlier points in the program.

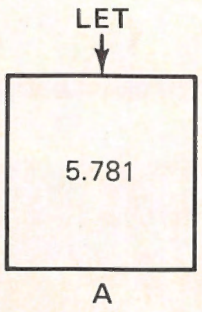


Figure 4-3. The variable A.

TEST YOUR UNDERSTANDING 4 (answer on page 86)
Suppose that a loan for $5,000 has an interest rate of 1.5 percent on the unpaid balance at the end of each month. Write a program to calculate the interest at the end of the first month. Suppose that at the end of the first month, you make a payment of $150 (after the interest is added). Design your program to calculate the balance after the payment. (Begin by letting B = the loan balance, I = the interest, and P = the payment. After the payment, the new balance is B+I−P.)

Example 2. What will be the output of the following computer program:

```
10 LET A = 10: B = 20
20 LET A = 5
30 PRINT A + B + C, A*B*C
40 END
```

Solution. Note that no value for C is specified, so C = 0. Also note that the value of A initially is set to 10. However, in line 20, this value is changed to 5. So in line 30, A, B, and C have the respective values 5, 20, and 0. Therefore, the output will be:

```
25              0
```

To the computer, the statement:

```
LET A =
```

means that the current value of A is to be **replaced** with whatever appears to the right of the equal sign. Therefore, if you write:

```
LET A = A + 1
```

you are asking the computer to replace the current value of A with A + 1. So if the current value of A is 4, the value of A after performing the instruction is 4 + 1, or 5.

TEST YOUR UNDERSTANDING 5 (answer on page 86)
What is the output of the following program?

```
10 LET A = 5.3
20 LET A = A+1
30 LET A = 2*A
40 LET A = A+B
50 PRINT A
60 END
```

The variables you have been using are called **single-precision numeric variables** and are capable of holding up to seven significant digits of information. (Later on, I'll talk about double-precision numeric variables, which can hold more than seven significant digits.) If you set a single-precision variable equal to a number with more than seven significant digits, BASIC automatically will round the number to seven significant digits. Moreover, if displaying the value of a number will require more than seven digits due to zeros before or after the decimal place, BASIC automatically will shift to scientific notation. Thus, for example, the statement:

```
X=123456789: PRINT X
```

will produce the display:

```
1.234568E+09
```

Note that the initial seven digits are obtained by rounding the given 10 digits. Scientific notation is used because the rounded number, namely, 123456800, requires more than seven digits to display.

String Variables

So far, all of the variables discussed have represented numeric values. However, BASIC also allows variables to assume string constants (sequences of characters) as values. The variables for doing this are called *string variables* and are denoted by a variable name followed by a dollar sign ($). Thus, **A$**, **B1$**, and **ZZ$** are all valid names of string variables. To assign a value to a string variable, use the **LET** statement with the desired value inserted in quotation marks after the equal sign. To set A$ equal to the string "Balance Sheet", use the statement:

```
LET A$ = "Balance Sheet"
```

You may print the value of a string variable just as you print the value of a numeric variable. For example, if A$ has the value just assigned, the statement:

```
PRINT A$
```

will result in the following screen output:

```
Balance Sheet
```

Example 3. What will be the output of the following program:

```
10 LET A$ = "RECEIPTS":B$ = "EXPENSES"
20 LET A = 20373.10: B = 17584.31
30 PRINT A$,B$
40 PRINT A,B
50 END
```

Solution. Line 30 prints the values of the two string variables A$ and B$, namely, "RECEIPTS" and "EXPENSES", at the beginning of two print zones. Line 40 displays the values of A and B. Here is the output of the program:

```
RECEIPTS                    EXPENSES
 20373.10                    17584.31
```

Note that we have used the variables A and A$ (as well as B and B$) in the same program. The variables A and A$ are considered *different* by the computer. One further comment about spacing: Note that the numbers do not exactly align with the headings, but are offset by one space. This is because BASIC allows room for a sign (+ or -) in front of a number. In the case of positive numbers, the sign is left out, but the space remains.

The SWAP Statement

Suppose that your program involves the two variables A and B and that you wish to reassign the values of these variables so that A assumes the value of B, and B the value of A. This may be accomplished using the BASIC statement:

```
10 SWAP A, B
```

For example, if A currently has the value 1.8 and B the value 7.5, then after the above statement is executed, A will have the value 7.5 and B the value 1.8.

Note that SWAP also may be used to exchange the values of two string variables, as in the statement:

```
20 SWAP A$, B$
```

However, you may never SWAP values between a string variable and a numeric variable. BASIC will report an error if you try this.

TEST YOUR UNDERSTANDING 6 (answer on page 86)
Write a BASIC program to exchange the values of the variables A and B without using the SWAP statement. (It's tricky. That's why BASIC includes the SWAP statement.)

Remarks in Programs

It is very convenient to explain programs using remarks. For one thing, remarks make programs easier to be read by a human being. Remarks also assist in finding errors and making modifications in a program. To insert a remark in a program, you may use the **REM** statement. For example, consider the line:

```
520 REM X DENOTES THE STAR SHIP POSITION
```

Since the line starts with **REM**, it will be ignored during program execution. As a substitute for REM, you use an apostrophe, as in the following example:

```
1040 ' Y IS THE LASER FORCE
```

To insert a remark on the same line as a program statement, use a colon followed by an apostrophe (or **REM**), as in this example:

```
10 LET A = PI*R^2 : ' A IS THE AREA,R IS THE RADIUS
```

Note, however, that everything after an apostrophe is ignored. Therefore, you cannot put an instruction after a remark. In the line

```
20 LET B=A^2: 'B is the area: C=B+8
```

the instruction C = B + 8 will be ignored.

The importance of remarks cannot be overemphasized. In writing BASIC programs, it is all too easy to write programs that no one (you included) can decipher. You should aim at writing programs that can be read like text. And the most significant step in this direction is to include many remarks in your programs. In what follows, we will be generous in our use of remarks, not only to make the programs easier to read, but also to set an example of good programming style.

TEST YOUR UNDERSTANDING 7 (answer on page 86)
What is the result of the following program line?

```
10 LET A=7:B$="COST":C$="TOTAL":PRINT C$,B$,"=",A
```

Using a Printer

In writing programs and analyzing their output, it is often easier to rely on written output rather than output on the screen. In computer terminology, written output is called **hard copy** and may be provided by a wide variety of printers, ranging from a dot-matrix printer costing only a few hundred dollars to a

daisy wheel printer costing several thousand dollars. As you begin to make serious use of your computer, you will find it difficult to do without hard copy. Indeed, writing programs is much easier if you can consult a hard copy listing of your program at various stages of program development. (One reason is that in printed output you are not confined to looking at your program in 25-line "snapshots.") Also, you will want to use the printer to produce output of programs, ranging from tables of numeric data to address lists and text files.

You may produce hard copy on your printer by using the BASIC statement **LPRINT**. For example, the statement:

```
10 LPRINT A,A$
```

will print the current values of A and A$ on the printer, in print zones one and two. (As is the case with the screen, BASIC divides the printer line into print zones that are 14 columns wide.) Moreover, the statement:

```
20 LPRINT "Customer","Credit Limit","Most Recent Pchs"
```

will result in printing three headings in the first three print zones, namely:

```
Customer          Credit Limit      Most Recent Pchs
```

Printing on the printer proceeds very much like printing on the screen. It is important to realize, however, that in order to print on both the screen *and* the printer, it is necessary to use *both* statements **PRINT** and **LPRINT**. For example, to print the values of A and A$ on both the screen and the printer, we must give two instructions, as follows:

```
10 PRINT A,A$
20 LPRINT A,A$
```

Exercises (answers on page 351)

In Exercises 1-6, determine the output of the given program.

1.
```
10 LET A = 5:B = 5
20 PRINT A + B
30 END
```

2.
```
10 LET AA = 5
20 PRINT AA*B
30 END
```

3.
```
10 LET A1 = 5
20 PRINT A1^2 +5*A1
30 END
```

4.
```
10 LET A = 2: B = 7: C = 9
20 PRINT A+B, A-C, A*C
30 END
```

5.
```
10 LET A$ = "JOHN JONES"
20 LET B$ = "AGE": C = 38
30 PRINT A$, B$, C
40 END
```

6.
```
10 LET X = 11: Y = 19
20 PRINT 2*X
30 PRINT 3*Y
40 END
```

What is wrong with the following BASIC statements?

7. `10 LET A = "YOUTH"`
8. `10 LET AA = -12`
9. `10 LET A$ = 57`
10. `LET ZZ$ = Address`
11. `250 LET AAA = -9`
12. `10000 LET 1A = -2.34567`

13. Consider the numbers 2.3758, 4.58321, and 58.11. Write a program that computes their sum, product, and the sum of their squares.
14. A company has three divisions: Office Supplies, Computers, and Newsletters. The revenues of these three divisions for the preceding quarter were, respectively, $346,712, $459,321, and $376,872. The expenses for the quarter were $176,894, $584,837, and $402,195, respectively. Write a program that displays this data on the screen, with appropriate explanatory headings. Your program also should compute and display the net profit (loss) from each division and the net profit (loss) for the company as a whole.

ANSWERS TO TEST YOUR UNDERSTANDING

1: 72
2: It prints the display:

```
5            25           50
```

3:
```
10 LET A=101:B=102:C=103:D=104:E=105:F=106
20 PRINT A*B
30 PRINT A*B*C
40 PRINT A*B*C*D
50 PRINT A*B*C*D*E
60 PRINT A*B*C*D*E*F
70 END
```

4:
```
10 LET B = 5000: I = .015: P = 150.00
20 IN = I*B
30 PRINT "INTEREST EQUALS", IN
40 B = B+IN
50 PRINT " BALANCE WITH INTEREST EQUALS", B
60 B = B - P
70 PRINT "BALANCE AFTER PAYMENT EQUALS", B
80 END
```

5. 12.6
6.
```
10 TEMPORARY=A
20 A=B
30 B=TEMPORARY
```

7. It creates the display:

```
TOTAL       COST       =       7
```

4.7 Some BASIC Commands

Thus far, most of our attention has been focused on learning statements to insert **inside** programs. Now let's learn a few of the commands available for **manipulating** programs and the computer. The **NEW** command, previously discussed, is in this category. Remember the following facts about BASIC commands:

BASIC Commands

1. Commands are typed *without* using a line number.
2. You must press the **ENTER** key after typing a command.
3. A command may be given whenever the computer is in the command mode. (Recall that whenever the computer enters the command mode, it displays the **Ok** message. The computer remains in the command mode until a **RUN** command is given.)
4. The computer executes commands as soon as they are received.

Listing a Program

To obtain a list of all program lines of the current program in RAM, you may type the command:

```
LIST<ENTER>
```

For example, suppose that RAM contains the following program:

```
10 PRINT 5+7, 5-7
20 PRINT 5*7,5/7
30 END
```

(This program may or may not be currently displayed on the screen.) If you type **LIST**, then the above three instruction lines will be displayed, followed by the **Ok** message.

In developing a program, you often will find that it is necessary to add program lines to sections of the program already written. This will require you to input lines in nonconsecutive order. Also, it may be necessary to correct lines already input. In either event, the screen often will not indicate the current version of the program. Typing **LIST** every so often will assist in keeping track of what has been changed. **LIST**ing is particularly helpful in checking a program or determining why a program won't run.

Note that you may display up to 25 lines of text on the screen at one time. This means you can display only 23 program statements at one time. (The "Ok" prompt takes one line, as does the cursor.) Therefore, it is often necessary to list only selected lines, rather than the entire program. To **LIST** only those statements with line numbers from 1 to 25, use the command:

```
LIST 1-25 <ENTER>
```

In a similar fashion, list any collection of consecutive program lines.

There are several other variations of the **LIST** command. To list the program lines from the beginning of the program to line 75, use the command:

```
LIST -75 <ENTER>
```

Similarly, to list the program lines from 100 to the end of the program, use the command:

```
LIST 100- <ENTER>
```

To list line 100 , use the command:

```
LIST 100 <ENTER>
```

TEST YOUR UNDERSTANDING 1 (answers on page 94)
Write a command to:

a. List line 200
b. List lines 300 to 330
c. List lines 300 to the end

Test these commands with a program.

Helpful Shortcut

If you press function key F1 and then ENTER, the PCjr will display a listing of the current program.

Printed Listings

You will find that it is difficult to write a long program relying only on screen listings. For more complex programs, a printed listing is essential. You may generate such a listing using your printer. To list the program currently in RAM, type:

```
LLIST
```

and press ENTER. All the variations of the LIST command apply also to the LLIST command. For example, you may list only those lines with line numbers in a certain range, lines from the beginning of the program to a given line number, and so forth.

Deleting Program Lines

When typing a program or revising an existing program, it is often necessary to delete lines that are already part of the program. One simple way is to type the line number followed by **ENTER**. For example:

```
275
```

(followed by hitting the **ENTER** key) will delete line 275. The **DELETE** command also may be used for the same purpose. For example, you may delete line 275 using the command:

```
DELETE 275 <ENTER>
```

The **DELETE** command has a number of variations which make it quite flexible. For example, to delete lines 200 to 500 inclusive, use the command:

```
DELETE 200-500 <ENTER>
```

To delete all lines from the beginning of the program to 350, inclusive, use the command:

```
DELETE -350 <ENTER>
```

Note, however, that in Cassette BASIC, the **DELETE** command always must include a last line number to be deleted. This is to prevent unfortunate mishaps by which you mistakenly erase most of a program. If you wish to delete all lines from 100 to the end of the program, you must specify a deletion from 100 to the last line number. If you don't remember the last line number, **LIST** the program first, determine the final line number, and then carry out the appropriate **DELETE**.

Helpful Shortcut (Cassette BASIC Only)

If your program is long, you may want to avoid listing it in order to determine the number of the last line. Here is how to delete to the end of the program without listing it. The largest possible line number is 65535. Therefore, type:

```
65535 END
```

and give the command

```
DELETE 100-65535
```

TEST YOUR UNDERSTANDING 2 (answers on page 94)
What is wrong with the following commands?

a. DELETE 450-
b. LIST 450-
c. DELETE 300-200

Saving a Program

Diskette Once you have typed a program into RAM, you may save a copy on cassette or diskette. At any future time, you may read the saved copy back into RAM. At that point, you may reexecute the program, modify it, or add to it. For the sake of concreteness, suppose that the following program is in RAM:

```
10 PRINT 5+7
20 END
```

Program Names In order to save a program, you must first assign the program a name. A program name is a string of letters or numbers and may contain as many as eight characters. In addition, you may include an extension consisting of a period followed by three characters. If you specify more than eight characters in a program name, characters nine, 10, and 11 are assumed to be an extension. Here are some valid program names for programs saved on diskette:

```
ACCOUNTING1 , GAMES.JOE , STORY.003
```

The first program name is equivalent to:

```
ACCOUNTI.NG1
```

If you do not specify an extension in a program name, then BASIC will automatically add the extension .BAS.

Saving Programs Suppose that you choose the name RETAIN for your program. You may save this program on the diskette in either disk drive. To save RETAIN on drive B:, for example, you would use the command:

```
SAVE "B:RETAIN"
```

When the computer finishes writing a copy of the program onto the designated diskette, it will display the **Ok** prompt. Saving a program does not alter the copy of the program in RAM.

Cassette The name of a program on cassette is limited to eight characters with no extension. To save the program 'RETAIN' on cassette, use the command:

```
SAVE "CAS1:RETAIN"
```

If your system does not have a diskette drive, you should omit the portion CAS1:.

Helpful Shortcut

To save a program press function key F4. BASIC will display:

```
SAVE "
```

You then may fill in the program name and press ENTER.

Recalling a Program

To read a program from diskette into RAM, use the **LOAD** command. For example, to read RETAIN from the diskette in drive B, use the command:

```
LOAD "B:RETAIN"
```

To recall the program "RETAIN" from cassette, you must position the tape at the beginning (or at least at a position so that the cassette recorder may reach the program only by going in the forward direction) and use the command:

```
LOAD "CAS1:RETAIN"
```

If your system does not have a diskette drive, you should omit the portion CAS1:.

You should try the above sequence of commands using the given program. After saving the program, erase the program from RAM (by typing **NEW**). Then load the program. Just to check that the program has indeed been retrieved, you should **LIST** it.

Helpful Shortcut

To load a program press function key F3. BASIC will display:

```
LOAD "
```

You then may fill in the program name and press ENTER.

Erasing a Program From Diskette

You may erase a program from diskette using the KILL command. To use this command, you must recognize that if you specify no extension in your program name when you saved it, then BASIC automatically added the

extension .BAS. For example, the program RETAIN is actually stored under the name RETAIN.BAS. To erase this program, you may use the command:

```
KILL "B:RETAIN.BAS"
```

The only way to erase a program from cassette is to record over it.

Manipulating Line Numbers

BASIC provides several commands that can ease your burden in dealing with line numbers.

The AUTO command may be used to automatically generate line numbers. To use this feature, type:

```
AUTO
```

and press ENTER. BASIC will generate line numbers 10, 20, 30, 40,... . A line number will be displayed and the cursor moved to the second space after the line number. In response, type the corresponding program line. As usual, end the line by pressing ENTER. The computer then will automatically display the next line number.

To disable the AUTO feature, simultaneously press the Fn and Break keys. The BASIC prompt then will be displayed.

You may have noticed that we always use line numbers that are multiples of 10. There is a good reason for this seeming waste of line numbers. It is often necessary to add instructions between program lines. Our numbering scheme leaves rooms for up to nine such additions. (In between lines 40 and 50, for instance, you could add instruction lines 41, 42, ..., 49.)

There are several useful variations of the AUTO command. You may start the automatic line number generation from any point. For example, to generate the line numbers:

```
55, 65, 75, 85, ... ,
```

use the command:

```
AUTO 55
```

You also may adjust the spacing between line numbers. For example, to generate the sequence of line numbers

```
38, 43, 48, 53, 58, ...,
```

which begins with 38 and has a spacing sequence of 5, just use the command:

```
AUTO 38,5
```

BASIC also provides for automatic renumbering of lines. This is helpful, for example, when it is necessary to MERGE two programs whose line numbers overlap. The command

```
RENUM
```

causes BASIC to renumber all line numbers. The renumbered program will start with line 10 and use a spacing of 10. As with AUTO, the RENUM command has several useful variations. To renumber a program so that the line numbers begin with 1000, use the command:

```
RENUM 1000
```

Renumbering may be restricted to a portion of the current program. To renumber lines 200 onward with the new line numbers beginning with 1000, use the command:

```
RENUM 1000,200
```

All lines with numbers below 200 are not renumbered. You may even vary the spacing of the renumbered lines. To renumber lines 200 onward with the new line numbers beginning with 1000 and having a spacing sequence of 100, use the command:

```
RENUM 1000,200,100
```

To summarize, the general form of the RENUM command is:

```
RENUM <new line> <,old line> <,increment>
```

Exercises (answers on page 351)
Exercises 1-7 refer to the following program:

```
10 LET A = 19.1: B = 17.5
20 PRINT A+B,A*B
30 END
```

1. Type the above program into RAM and RUN it. Use the AUTO feature to generate the line numbers.
2. Erase the screen without erasing RAM. LIST the program.
3. Save the program and erase RAM.
4. Recall the program and LIST it. RUN the program again.
5. Add the following line to the program:

```
25 PRINT A^2 + B^2
```

(Do not retype the entire program!) LIST and RUN the new program.
6. Save the new program without destroying the old one.

7. Recall the new program. Delete line 20 and RUN the resulting program.
8. Renumber the lines so that the line numbers are 100, 200, 300.
9. Renumber the lines so that the line numbers are 10, 2000, 2005.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. LIST 200
   b. LIST 300-330
   c. LIST 300-

2: a. OK in Cartridge BASIC. In Cassette BASIC, the line number of the last line to be deleted must be specified. It should read:

```
DELETE -450
```

   b. Nothing wrong.
   c. The lower line number must come first. The command should read:

```
DELETE 200-300
```

4.8 Some Programming Tips

Writing programs in BASIC is not difficult. However, it does require a certain amount of care and meticulous attention to detail. Each person must develop an individual programming style. Here are a few tips that may help you over some of the rough spots of writing those first few programs.

Programming Tips

1. Carefully think your program through. Break up the computation into steps. Describe each step in clear English. (If you can't tell yourself what you want the computer to do, it is unlikely that you can tell the computer.)

2. Write a set of instructions corresponding to each step. Check your instructions carefully, with an eagle eye for misspellings, missing parentheses, and other errors.

3. Pepper your work with remarks. Next week (or next month), you may wish to modify your program. It's embarrassing not to be able to figure out how your own works!

4. Type your program so that you can read it like a story. (More on how to do this in the next chapter.)

5. Work through your program by hand, pretending that you are the computer. Don't rush. Go through your program one step at a time and check that it does what you want it to do.

6. Have you given all variables the values you want? Remember, if you do not specify the value of a variable, BASIC will automatically assign it the value zero. This may not be the value you intend!

In the upcoming chapters, I will not only teach you how to program in BASIC. I will also encourage you to develop good programming habits and a useful programming style. In the process, we will add to the above list of programming tips.

4.9 Using the BASIC Editor

Suppose that you discover a program line with an error in it. How can you correct it? Up to now, the only way was to retype the line. There is a much better way. The PCjr has a powerful **full-screen editor**. This editor allows you to add, delete, or change text in existing program lines. This section is designed to teach you to use the editor.

The editing process (the process of changing or correcting characters already typed) consists of three steps:

1. Indicate the location of the change.
2. Input the change.
3. Send the change to the computer by using the **ENTER** key.

These steps make use of a number of special editing keys. Most of these keys are found on the right side of the keyboard.

The best way to understand the editing process is to work through several examples. If at all possible, follow these examples by typing them out on your keyboard. Suppose that you have typed the following program lines:

```
10 PRIMT X,Y,Z
20 IF A = 5 THN 50 ELSE 30
_
```

The third line indicates the cursor position. There are two spelling errors: PRIMT and THN. (If the computer had any common sense, it would have known what you meant.) In addition, suppose that you wish to change X, Y, and Z in the first line to read: A, X, Y, Z. Finally, suppose you wish to delete the ELSE 30 on the second line. Let's use the editing process to correct them. The first step is to position the cursor at the first character to be corrected. To do this, use the various keys on the numeric keypad which move the cursor like this:

↑ - Cursor up one line
↓ - Cursor down one line
← - Cursor left one character
→ - Cursor right one character

(There are other cursor motion keys, but let's study only these for now.) To correct the PRIMT error, we must first position the cursor at the M. To do this, we first hit the ↑ key twice. This moves the cursor up two lines. The display now looks like this:

```
10 PRIMT X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

Next we hit the → key six times to move the cursor to the right six spaces. (Note that the space between 0 and P counts.) The display now looks like this:

```
10 PRIMT X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

We have now accomplished step 1: The cursor is at the character to be corrected. Now we execute step 2: We type in the change. In this case, we type N. Note that the N replaces the M. Here is the display:

```
10 PRINT X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

The first error has now been corrected. Note, however, that the correction has not yet been sent to the computer via the **ENTER** key. We could do so at this point, but it wouldn't make much sense since there is another error to correct on the same line. Let's tend to that error now. To do so, we must insert the characters A and , before the X. Move the cursor two spaces to the right. Here is the display:

```
10 PRINT X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

To insert text at the cursor position, we hit the **Ins** key and type the material to be inserted: A, . The **Ins** key puts the computer in **insert mode**. In this mode, typed text is inserted at the current position and all other text moves to the right. Here is the current display:

```
10 PRINT A,X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

Since we have finished the insertion, we cancel the insert mode. This may be done in several ways. One method is to hit the **Ins** key again. This would allow us to continue to make further corrections on the same line. Another method (in this case the preferred one) is to hit the **ENTER** key. This cancels the insert mode and sends the corrected line to the computer. Note that the cursor may be in any position on the line when the **ENTER** command is given. Here is the display after **ENTER**.

```
10 PRINT A,X,Y,Z
20 IF A = 5 THN 50 ELSE 30
```

Note that the cursor is now at the first character of line 20. We correct the misspelling of THEN by moving the cursor to the N (14 spaces to the right), typing **Ins** followed by E, followed by **Ins**. Here is how the display looks now.

```
10 PRINT A,X,Y,Z
20 IF A = 5 THEN 50 ELSE 30
```

The final correction is to delete ELSE 30. This is done using the Del key. First we position the cursor on the E in ELSE. Then we hit the Del key seven times. Each repetition of the Del key deletes the character at the current cursor position and moves the remaining text to the left. For example, after hitting Del the first time, the display looks like this:

```
10 PRINT A,X,Y,Z
20 IF A = 5 THEN 50 LSE 30
```

After seven repetitions of the Del key, the display looks like this.

```
10 PRINT A,X,Y,Z
20 IF A = 5 THEN 50 _
```

The corrections are now complete. We send the line to the computer via the **ENTER** key.

The above example illustrates various editing features of the IBM PCjr. We may use the editing keys in the same way, to alter any line on the screen. If you wish to alter a program line which is not currently on the screen, you may display the desired line using the **LIST** command. Editing would then take place as shown.

There are a number of other keys which make editing faster. For example, to speed up cursor movement, we have the following keys:

Fn-Home This key moves the cursor to the upper left corner of the screen (the so-called "home" position).

Ctrl-Fn-Home This key combination clears the screen and brings the cursor to the home position.

Fn-End This key combination moves the cursor to the end of the current line.

Ctrl-Fn-End This key combination erases from the current cursor position to the end of the line.

Ctrl-PgDn This key combination moves the cursor to the space to the right of the beginning of the next word. (Think of a word as any sequence of characters not containing spaces. This is not exactly correct, but is close enough for practical purposes.)

Ctrl-PgUp This key combination moves the cursor to the space to the left of the beginning of the next word.

End This key moves the cursor to the end of the current line.

In addition to the editing keys described above, the following two key combinations are useful.

Ctrl-Fn-End erases input from the current cursor position to the end of the line.

Fn-Break cancels all editing changes in the current line.

IMPORTANT NOTE: Editing changes occur only in the copy of the program in RAM. In order for changes to be reflected in copies of the program on cassette or diskette, it is necessary to save the edited copy of the program. The moral:

AFTER MAKING CORRECTIONS, SAVE YOUR PROGRAM!

Exercises

What keystrokes accomplish the following editing functions?

1. Move the cursor four spaces to the right.
2. Delete the fourth letter to the right of the cursor.
3. Insert the characters 538 at the current cursor position.
4. Delete the portion of the line to the right of the cursor position.
5. Move the cursor up eight spaces.
6. Move the cursor to the right three spaces.
7. List the current version of the line.
8. Change 0 to 1 at the current cursor position.
9. Delete the letter "a" eight spaces to the left of the current cursor position.
10. Cancel all changes in the current line.

Use the line editor to make the indicated changes in the following program line. The exercises are to be done in order.

```
300 FOR M = 11 TO 99, SETP .5 : X = M^2 - 5
```

11. Delete the ,
12. Correct the misspelling of the word STEP.
13. Change M^2 - 5 to M^3 - 2 .
14. Change .5 to -1.5 .
15. Add the following characters to the end of the line. : Y = M + 1 .

5

CONTROLLING THE FLOW OF YOUR PROGRAM

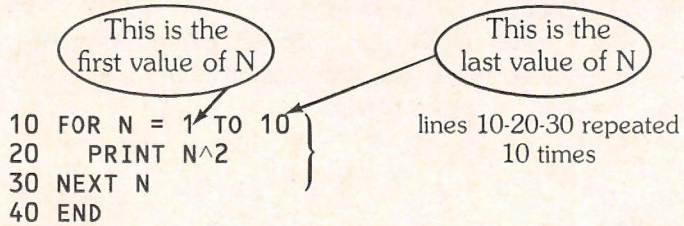
In this chapter we will continue our introduction to diskette BASIC on the IBM PCjr. Our discussion will center on the instructions for controlling the flow of statement execution.

5.1 Doing Repetitive Operations

Suppose that we wish to solve 50 similar multiplication problems. It is certainly possible to type in the 50 problems one at a time and let the computer solve them. However, this is a very clumsy way to proceed. Suppose that instead of 50 problems there were 500, or even 5000. Typing the problems one at a time would not be practical. If, however, we can describe to the computer the entire class of problems we want solved, then we can instruct the computer to solve them using only a few BASIC statements. Let us consider a concrete problem. Suppose that we wish to calculate the quantities

$1^2, 2^2, 3^2, \ldots, 10^2$

That is, we wish to calculate a table of squares of integers from 1 to 10. This calculation can be described to the computer as calculating N^2, where the variable N is allowed to assume, one at a time, each of the values 1,2,3,...,10. Here is a sequence of BASIC statements which accomplishes the calculations:



The sequence of statements 10,20,30 is called a **loop**. When the computer encounters the **FOR** statement, it sets N equal to 1 and continues executing the statements. Statement 20 calls for printing N^2. Since N is equal to 1, we have N^2 = 1^2 = 1. So the computer will print a 1. Next comes statement 30, which calls for the next N. This instructs the computer to return to the **FOR** statement in 10, increase N to 2, and to repeat instructions 20 and 30. This time, N^2 = 2^2 = 4. Line 20 then prints a 4. Line 30 says to go back to line

10 and increase N to 3 and so forth. Lines 10, 20, and 30 are repeated 10 times! After the computer executes lines 10,20,and 30 with N = 10, it will leave the loop and execute line 40.

Type in the above program and give the **RUN** command. The output will look like this:

```
1
4
9
16
25
36
49
64
81
100
Ok
```

The variable N is called the **loop variable**. It may be used inside the loop just like you would any other variable. For example, it may be used in algebraic calculations and **PRINT** statements.

TEST YOUR UNDERSTANDING 1 (answer on page 110)

a. Devise a loop allowing N to assume the values 3 to 77.
b. Write a program which calculates N^2 for N = 3 to 77.

Making Loops More Readable Note that we have indented the textual portion of line 20. This allows us to clearly see the beginning and end of the loop. It is good programming practice to always indent loops in this way since it increases program readability. The TAB key (the key with the two symbols ←| and →|) may be used to indent. BASIC sets up tab stops every five spaces. These are just like the tab stops on a typewriter. Whenever you press the TAB key, the cursor moves over to the next tab stop.

Let's modify the above program to include on each line of output not only N^2, but also the value of N. To make the table easier to read, let's also add two column headings. The new program reads:

```
10 PRINT "N","N^2"
20 FOR N=1 TO 10
30   PRINT N,N^2
40 NEXT N
50 END
```

The output now looks like this:

```
N             N^2
1             1
2             4
```

```
3             9
4             16
5             25
6             36
7             49
8             64
9             81
10            100
Ok
```

TEST YOUR UNDERSTANDING 2 (answer on page 110)
What would happen if we change the number of line 10 to 25?

Let us now illustrate some of the many uses loops have by means of some examples.

Example 1. Write a BASIC program to calculate $1+2+3+...+100$.
Solution. Let us use a variable S (for sum) to contain the sum. Let us start S at 0 and use a loop to successively add to S the numbers 1,2,3,...,100. Here is the program.

```
10 LET S = 0
20 FOR N = 1 TO  100 }     These instructions
30   LET S = S + N   }     repeated 100 times
40 NEXT N            }
50 PRINT S
60 END
```

When we enter the loop the first time, S = 0 and N = 1. Line 30 then replaces S by S + N, or 0 + 1. Line 40 sends us back to line 20, where the value of N is now set equal to 2. In line 30, S (which is now 0 + 1) is replaced by S + N, or 0 + 1 + 2. Line 40 now sends us back to line 20, where N is now set equal to 3. Line 30 then sets S equal to 0 + 1 + 2 + 3. Finally, on the 100th time through the loop, S is replaced by 0 + 1 + 2 + ... + 100, the desired sum. If we run the program, we derive the output

```
5050
Ok
```

TEST YOUR UNDERSTANDING 3 (answer on page 110)
Write a BASIC program to calculate $101+102+...+110$.

TEST YOUR UNDERSTANDING 4 (answer on page 110)
Write a BASIC program to calculate and display the numbers 2,2^2,2^3,...,2^20.

Example 2. Write a program to calculate the sum:

$1 \times 2 + 2 \times 3 + 3 \times 4 + \ldots + 49 \times 50$

Solution. We let the sum be contained in the variable S, as in the preceding example. The quantities to be added are just the numbers N*(N+1) for N=1,2,3,...,49. Here is our program:

```
10 LET S = 0
20 FOR N = 1 TO 49
30   LET S = S + N*(N+1)
40 NEXT N
50 PRINT S
60 END
```

Some Cautions

Here are two of the errors you are most likely to make in dealing with loops:

1. Every FOR statement must have a corresponding NEXT. Otherwise, BASIC will halt your program and display the error message:

```
FOR without NEXT in line xxxxx
```

2. Be sure that the loop variable is not already used with some other meaning. For example, suppose that the loop variable N is used before the loop begins. Then the loop will destroy the old value of N and there is no way to get it back after the loop is completed.

Nested Loops

In many applications, it is necessary to execute a loop within a loop. For example, suppose that we wish to compute the following series of numbers:

```
1^2, 2^2, 3^2, ..., 10^2,
101^2, 102^2, 103^2, ..., 110^2,
...
...
2001^2, 2002^2, 2003^2, ..., 2010^2
```

There are 21 groups of 10 numbers each. Each line may be computed using a loop. For example, the first line may be computed using:

```
100 FOR I=1 TO 10
110   PRINT I^2
120 NEXT I
```

The second line may be computed using:

```
100 FOR I=1 TO 10
110  PRINT (100+I)^2
120 NEXT I
```

And the last line may be computed using:

```
100 FOR I=1 TO 10
110  PRINT (2000+I)^2
120 NEXT I
```

We could compute the desired numbers by repeating essentially the same instructions 21 times. However, it is much easier to do the repetition using a loop. The numbers to be added to I range from 0 (which is 0*100) for the first line, to 100 (which is 1*100) for the second line, to 2000 (which is 20*100) for the last line. This suggests that we represent these numbers as J*100, where J is a loop variable which runs from 0 to 20. We may then compute our desired table of numbers using this program:

```
10 FOR J=0 TO 20
100   FOR I=1 TO 10
110      PRINT (100*J+I)^2
120   NEXT I
200 NEXT J
```

The instructions which are indented one level are repeated 21 times, corresponding to the values J=0 through J=20. On the first repetition (J=0), lines 100-120 print the numbers in the first line; on the second repetition (J=1), lines 100-120 print the numbers in the second line, and so forth. Note how the indentations help to read the program. This is an example of good programming style.

If a loop is contained within a loop, then we say that the loops are **nested**. BASIC allows you to have nesting in as many layers as you wish (a loop within a loop within a loop, and so forth.)

TEST YOUR UNDERSTANDING 5 (answer on page 110)
Write a BASIC program to print the following table of numbers.

1	11	21	31
2	12	22	32
.			
.			
.			
9	19	29	39

Warning: Nested loops may not "overlap." That is, the following sequence is not allowed:

```
10 FOR J=1 TO 100
20   FOR K=1 TO 50
.
.
.
80 NEXT J
90   NEXT K
```

Rather, the NEXT K statement must precede the NEXT J, so that the K loop is "completely inside" the J-loop.

Applications of Loops

Example 3. You borrow $7000 to buy a car. You finance the balance for 36 months at an interest rate of one percent per month. Your monthly payments are $232.50. Write a program which computes the amount of interest each month, the amount of the loan which is repaid, and the balance owed.

Solution. Let B denote the balance owed. Initially we have B equal to $7000. At the end of each month let us compute the interest (I) owed for that month, namely .01*B. For example, at the end of the first month, the interest owed is .01*7000.00 = $70.00. Let P = 232.50 to denote the monthly payment, and let R denote the amount repaid out of the current payment. Then R = P - I. For example, at the end of the first month, the amount of the loan repaid is 232.50 -70.00 = 162.50. The balance owed may then be calculated as B - R. At the end of the first month, the balance owed is 7000.00 - 162.50 = 6837.50. Here is a program which performs these calculations:

```
10 PRINT "MONTH","INTEREST","BALANCE"
20 LET B = 7000       :'B=initial balance
25 LET P = 232.50     :'P=monthly payment
30 FOR M = 1 TO 36    :'M is month number
40   LET I = .01*B    :'Calculate interest for month
50   LET R = P - I    :'Calculate repayment
60   LET B = B - R    :'Calculate new balance
70   PRINT M,B        :'Print out data for month
80 NEXT M
90 END
```

You should attempt to run this program. Notice that it runs, but it is pretty useless because the screen will not contain all of the output. Most of the output goes flying by before you can read it. One method for remedying this situation is to press **Fn** and **Pause** simultaneously as the output scrolls by on the screen. This will pause execution of the program and freeze the contents of the screen. To resume execution and unfreeze the screen, press any key. The output will

begin to scroll again. To use this technique requires some manual dexterity. Moreover, it is not possible to guarantee where the scrolling will stop.

TEST YOUR UNDERSTANDING 6

RUN the program of Example 3 and practice freezing the output on the screen. It may take several runs before you are comfortable with the procedure.

Let us now describe another method of adapting the output to our screen size by printing only 12 months of data at one time. This amount of data will fit since the screen contains 24 lines. We will use a second loop to keep track of 12-month periods. The variable for the new loop will be Y (for "years"), and Y will go from 0 to 2. The month variable will be M as before, but now M will go only from 1 to 12. The month number will now be 12*Y + M (12 times the number of years plus the number of months). Here is the revised program.

```
10 LET B=7000
20 LET P =232.50
30 FOR Y = 0 TO 2    :'Y=year number
40   PRINT "MONTH","INTEREST","PAYMENT","BALANCE"
50   FOR M = 1 TO 12 :'Run through the months of year Y
60        LET I = .01*B:'Calculate interest for month
70        LET R = P - I:'Calculate repayment for month
80        LET B = B - R:'Calculate balance for month
90        PRINT 12*Y+M,B:'Print data for month
100  NEXT M
110 STOP          :' Halts execution
120 CLS           :' Clears Screen
130 NEXT Y        :' Goes to next 12 months
140 END
```

This program uses several new statements. In line 110, we use the **STOP** statement. This causes the computer to stop execution of the program. The computer remembers where it stops, however, and all values of the variables are preserved. The **STOP** statement also leaves unchanged the contents of the screen. You can take as long as you wish to examine the data on the screen. When you are ready for the program to continue, type **CONT** and press ENTER. The computer will resume where it left off. The first instruction it encounters is in line 120. **CLS** clears the screen. So, after being told to continue, the computer clears the screen and goes on to the next value of Y—the next 12 months of data. Here is a copy of the output. The underlined statements are those you type.

```
Ok

RUN

MONTH           BALANCE
 1              6837.5
 2              6673.375
 3              6507.609
 4              6340.185
 5              6171.087
 6              6000.298
 7              5827.8
 8              5653.578
 9              5477.614
 10             5299.89
 11             5120.389
 12             4939.093

Break in 100
Ok
CONT

MONTH           BALANCE
 13             4755.984
 14             4571.044
 15             4384.255
 16             4195.597
 17             4005.053
 18             3812.603
 19             3618.229
 20             3421.912
 21             3223.631
 22             3023.367
 23             2821.101
 24             2616.812

Break in 100
Ok
CONT

MONTH           BALANCE
 25             2410.48
 26             2202.085
 27             1991.606
 28             1779.022
 29             1564.312
 30             1347.455
 31             1128.43
 32             907.2138
```

```
 33            683.7859
 34            458.1238
 35            230.205
 36            7.034302E-03

Ok
```

Note that the data in the output is carried out to seven figures, even though the problem deals with dollars and cents. We will look at the problem of rounding numbers later. Also note the balance listed at the end of month 36. It is in scientific notation. The -03 indicates that the decimal point is to be moved three places to the left. The number listed is .007034302 or about .70 cents (less than one cent)! The computer shifted to scientific notation since the usual notation (.007034302) requires more than seven digits. The computer made the choice of which form of the number to display.

Using Loops to Create Delays

By using a loop we can create a delay inside the computer. Consider the following sequence of instructions:

```
10 FOR N = 1 TO 3000
20 NEXT N
```

This loop doesn't do anything! However, the computer repeats instructions 10 and 20 three thousand times! This may seem like a lot of work. But not for a computer. To obtain a feel for the speed at which the computer works, you should time this sequence of instructions. Such a loop may be used as a delay. For example, when you wish to keep some data on the screen without stopping the program, just build in a delay. Here is a program that prints two screens of text. A delay is imposed to give you time to read the first screen.

```
10 PRINT "THIS IS A GRAPHICS PROGRAM
20 PRINT "TO DISPLAY SALES"
30 PRINT "FOR THE YEAR TO DATE"
40 FOR N = 1 TO 5000   } Delay Loop
50 NEXT N:
60 CLS
70 PRINT "YOU MUST SUPPLY"
80 PRINT "THE FOLLOWING PARAMETERS:"
90 PRINT "PRODUCT, TERRITORY, SALESPERSON"
100 END
```

Example 4. Use a loop to produce a blinking display for a security system. Suppose that your security system is tied in with your computer and the system detects that an intruder is in your warehouse. Let us print out the message

```
INTRUDER-ZONE 2
```

For attention, let us blink this message on and off by alternately printing the message and clearing the screen.

Solution: Here is our program.

```
10 FOR N = 1 TO 2000
20   PRINT "INTRUDER-ZONE 2"
30   FOR K = 1 TO 50
40   NEXT K
50   CLS
60 NEXT N
70 END
```

The loop in lines 30-40 is a delay loop to keep the message on the screen for a moment. Line 50 turns the message off, but the **PRINT** statement in line 20 turns it back on. The message will blink 2000 times.

TEST YOUR UNDERSTANDING 7 (answer on page 110)
Write a program that blinks your name on the screen 500 times, leaving your name on the screen for a loop of length 50 each time.

More About Loops

In all of our loop examples, the loop variable increased by one with each repetition of the loop. However, it is possible to have the loop variable change by any amount. For example, the instructions

```
10 FOR N = 1 TO 5000 STEP 2
.
.
.
1000 NEXT N
```

define a loop in which N jumps by 2 for each repetition, so N will assume the values:

```
1, 3, 5, 7, 9 ,..., 4999
```

Similarly, use of STEP .5 in the above loop will cause N to advance by .5 and assume the values:

```
1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, ... , 5000
```

It is even possible to have a negative step. In this case, the loop variable will run backwards. For example, the instructions

```
10 FOR N = 100 TO 1 STEP -1
.
.
.
100 NEXT N
```

will "count down" from N = 100 to N = 1 one unit at a time. We will give some applications of such instructions in the Exercises.

TEST YOUR UNDERSTANDING 8 (answer on page 110)
Write instructions to allow N to assume the following sequences of values:
a. 95, 96.7, 98.4 ,..., 112
b. 200, 199.5, 199 ,..., 100

Exercises (answers on page 351)

Write BASIC programs to compute the following quantities.

1. $1^2+2^2+3^2+...+25^2$
2. $(1/2)^0 + (1/2)^1 + (1/2)^2 + ... + (1/2)^{10}$
3. $1^3+2^3+3^3+...+10^3$
4. $1 + (1/2) + (1/3) + ... + (1/100)$
5. Write a program to compute N^2, N^3, and N^4 for N = 1,...,12. The format of your output should be as follows:
   N N^2 N^3 N^4
   1
   2
   3
   .
   .
   .
   12
6. Suppose that you have a car loan whose current balance is $4,000.00. The monthly payment is $125.33 and the interest is one percent per month on the unpaid balance. Make a table of the interest payments and balances for the next 12 months.
7. Suppose you deposit $1,000 on January 1 of each year into a savings account paying 10 percent interest. Suppose that the interest is computed on January 1 of each year, based on the balance for the preceding year. Calculate the balances in the account for each of the next 15 years.
8. A stock market analyst predicts that Tyro Computers, Inc. will achieve a 20 percent growth in sales in each of the next three years, but profits will grow at a 30 percent annual rate. Last year's sales were $35 million and last year's profits were $5.54 million. Project the sales and profits for the next three years, based on the analyst's prediction.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a.
```
10 FOR N=3 TO 77
 .
 .
100 NEXT N
```
b.
```
10 FOR N=3 TO 77
20     PRINT N^2
30 NEXT N
40 END
```

2: The heading
```
N                N^2
```
would be printed before each entry of the table.

3:
```
10 S=0
20 FOR N=101 TO 110
30   S=S+N
40 NEXT N
50 PRINT S
60 END
```

4:
```
10 FOR N=1 TO 20
20 PRINT 2^N
30 NEXT N
40 END
```

5:
```
10 FOR J=1 TO 9
20     FOR I=0 TO 3
30         PRINT 10*I+J
40     NEXT I
50 NEXT J
```

7:
```
10 FOR N=1 TO 500
20     PRINT "<YOUR NAME>"
30     FOR K=1 TO 50
40     NEXT K
50     CLS
60 NEXT N
70 END
```

8: a.
```
10 FOR N=95 TO 112 STEP 1.7
```
b.
```
20 FOR N=200 TO 100 STEP -.5
```

5.2 Letting Your Computer Make Decisions

One of the principal features which makes computers useful as problem-solving tools is their ability to make decisions. BASIC contains instructions which allow you to ask a question. The computer will determine the answer and will take an action which depends on the answer. Here are some examples of questions the computer can answer:

```
IS A GREATER THAN ZERO?

IS A^2 AT LEAST 200?
```

```
DOES THE STRING NAME$ BEGIN WITH A "Z" ?

IS AT LEAST ONE OF THE VARIABLES A, B OR C NEGATIVE?
```

Here are two BASIC statements which allow you to ask such questions: The IF...THEN statement and the IF...THEN...ELSE statement. The first of these statements has the form:

```
IF <question> THEN <statement or line number>
```

Here is how this statement works:

1. The "question" part of an IF...THEN statement allows you to ask questions like those above.
2. If the answer to the question is YES, the program executes the portion of the statement following THEN.
   a. If a statement follows THEN, this statement is executed.
   b. If a line number follows THEN, the program continues execution with this line number.
3. If the answer to the question is NO, the program continues with the next statement.

For example, consider this instruction:

```
500 IF N = 0 THEN PRINT "CALCULATION DONE"
```

The question portion of this instruction is N = 0; the portion following THEN is the statement: PRINT "CALCULATION DONE". When the computer encounters this statement, it first determines if N is equal to zero. If so, it prints "CALCULATION DONE" and proceeds with the next instruction after line 500. However, if N is not zero, the program immediately goes to the next instruction line after 500. (It ignores the statement after THEN.)

Here is another example:

```
600 IF A^2 < 1 THEN 300
```

When the program reaches this instruction, it will examine the value of A^2. If A^2 is less than 1, the program will go to line 300. Otherwise, the program will go on to the next instruction.

The IF...THEN...ELSE statement is similar to an IF...THEN statement, but it offers added flexibility in case the answer to the question is NO. The form of the IF...THEN...ELSE statement is:

```
IF <question> THEN <statement or line number>
ELSE <statement or line number>
```

This statement works as follows: The computer asks the given question. If the answer is YES, the program executes the THEN portion; if the answer is NO, the program executes the ELSE portion.

Here is an example:

```
500 IF N = 0 THEN PRINT "CALCULATION DONE" ELSE 250
```

The computer first determines if N equals 0. If so, it prints CALCULATION DONE. If N is not equal to 0, the program continues execution at line 250.

Another possibility is for both **THEN** and **ELSE** to be followed by instructions, as in this example:

```
600 IF A + B >= 100 THEN PRINT A + B ELSE PRINT A
```

In executing this instruction, the computer will determine whether A+B is greater than or equal to 100. If so, it will print the value of A+B; if not, it will print the value of A. In both cases, execution continues with the next instruction after line 600.

After **IF**, you may insert any expression which the computer may test for truth or falsity. Here are some examples:

```
N = 0

N > 5 (N is greater than 5)

N < 12.9 (N is less than 12.9)

N >= 0 (N is greater than or equal to 0)

N <= -1 (N is less than or equal to -1)

N >< 0 (N is not equal to 0)

A + B <> C (A + B is not equal to C)

A^2 + B^2 <= C^2 (A² + B² is less than or equal to C²)
```

You may even combine statements using the words AND and OR, as in the following examples:

```
N = 0 OR A > B (Either N = 0 or A > B or both)

N > M AND I = 0 (Both N > M and I = 0)
```

For clarity, it's advisable to put the individual statements within parentheses. For example, the last two statements would be clearer if written in the form

```
(N=0) OR (A>B)

(N>M) AND (I=0)
```

TEST YOUR UNDERSTANDING 1 (answers on page 124)
Write instructions which do the following:

a. If A is less than B, then print the value of A plus B; if not then go to the end.
b. If $A^2 + D$ is at least 5000 then go to line 300; if not go to line 500.
c. If N is larger than the sum of I and K, set N equal to the sum of I and K; otherwise, let N equal K.

Important Note that if the condition of an IF...THEN statement is false, then the program goes to the next **line number**. If there are other statements on the same line as the IF...THEN, they will be executed only if the condition is true. Consider, for example, the following statements:

```
200 IF X>0 THEN X=X+1: GOTO 300
210 X=0
```

If X is greater than 0, then X is replaced by X + 1 and the program goes to the next statement, namely GOTO 300. On the other hand, if X is not greater than 0, then the program skips the statement GOTO 300 and proceeds to line 210.

The **IF ... THEN** and **IF ... THEN ... ELSE** statements may be used to interrupt the normal sequence of executing program lines, based on the truth or falsity of some condition. In many applications, however, we will want to perform instructions out of the normal sequence, independent of any conditions. For such applications, we may use the **GOTO** instruction. (This is not a typographical error! There is no space between GO and TO.) This instruction has the form

```
GOTO < line number >
```

For example, the instruction

```
1000 GOTO 300
```

will send the computer back to line 300 for its next instruction.

The next few examples illustrate some of the uses of the **IF ... THEN**, **IF ... THEN ... ELSE**, and **GOTO** statements.

Example 1. A lumber supply house has a policy that a credit invoice may not exceed $1,000, including a 10 percent processing fee and 5 percent sales tax. A customer orders 150 2x4 studs at $1.99 each, 30 sheets of plywood at $14.00 each, 300 pounds of nails at $1.14 per pound, two double hung insulated windows at $187.95 each. Write a program which prepares an invoice and decides whether the order is over the credit limit.

Solution. Let's use the variables A1, A2, A3, and A4 to denote, respectively, the numbers of studs, sheets of plywood, pounds of nails, and windows. Let's

use the variables B1, B2, B3, and B4 to denote the unit costs of these four items. The cost of the order is then computed as:

```
A1*B1+A2*B2+A3*B3+A4*B4.
```

We add 10 percent of this amount to cover processing and form the sum to obtain the total order. Next, we compute 5 percent of the last amount as tax and add it to the total to obtain the total amount due. Finally, we determine if the total amount due is more than $1,000. If it is, we print out the message: ORDER EXCEEDS $1,000. CREDIT SALE NOT PERMITTED. Here is our program.

```
10 LET A1=150:A2=30:A3=300:A4=2:          'Assign quantities
20 LET B1=1.99:B2=14:B3=1.14:B4=187.95: 'Assign prices
30 LET T= A1*B1+A2*B2+A3*B3+A4*B4:       'T=total price
40 PRINT "TOTAL ORDER",T
50 LET P = .1*T:                          'P=processing fee
60 PRINT "PROCESSING FEE";P
70 LET TX = .05*(P+T):                    'TX = tax
80 PRINT "SALES TAX",TX
90 DU = T + P + TX:                       'DU=Amount due
100 PRINT "AMOUNT DUE", DU
110 IF DU > 1000 THEN 200 ELSE 300:       'Order > $1000 ?
200 PRINT "ORDER EXCEEDS $1,000"
210 PRINT "CREDIT SALE NOT PERMITTED"
220 GOTO 400:                             'End program
300 PRINT "CREDIT SALE OK"
400 END
```

Note the decision in line 110: If the amount due exceeds $1,000 then the computer goes to line 200 where it prints out a message denying credit. In line 220, the computer is sent to line 400 which is the END of the program. On the other hand, if the amount due is less than $1,000, the computer is sent to line 300, where credit is approved.

TEST YOUR UNDERSTANDING 2 (answer on page 124)
Suppose that a credit card charges 1.5 percent per month on any unpaid balance up to $500 and 1 percent per month on any excess over $500.

a. Write a program which computes the service charge and the new balance.
b. Test your program on the unpaid balances of $1300 and $275.

TEST YOUR UNDERSTANDING 3 (answer on page 124)
Consider the following sequence of instructions.

```
100 IF A>=5 THEN 200
110 IF A>=4 THEN 300
120 IF A>=3 THEN 400
130 IF A>=2 THEN 500
```

Suppose that the current value of A is 3. List the sequence of line numbers which will be executed.

Example 2. At $20 per square yard, a family can afford up to 500 square feet of carpet for their dining room. They wish to install the carpet in a circular shape. It has been decided that the radius of the carpet is to be a whole number of feet. What is the radius of the largest carpet they can afford? (The area of a circle of radius "R" is PI times R^2, where PI equals approximately 3.14159.)

Solution. Let us compute the area of the circle of radius 1, 2, 3, 4 ,... and determine which of the areas are less than 500.

```
10 PI = 3.14159
20 R = 1 :                    'R=radius
30 A = PI*R^2 :               'A=area
40 'Is A>=500 ? If so, END. Otherwise, PRINT R .
50 IF A >= 500 THEN 100 ELSE PRINT R
60 LET R = R + 1 :            'Go to next radius
70 GOTO 30:                   'Repeat
100 END
```

Note that line 50 contains an **IF ... THEN** statement. If A, as computed in line 30, is 500 or more, then the computer goes to line 100, **END**. If A is less than 500, the computer proceeds to the next line, namely 50. It then prints out the current radius, increases the radius by 1, and goes back to line 30 to repeat the entire procedure. Note that lines 30-40-50-60-70 are repeated until the area becomes at least 500. In effect, this sequence of five instructions forms a loop. However, we did not use a **FOR ... NEXT** instruction because we did not know in advance how many times we wanted to execute the loop. We let the computer decide the stopping point using the **IF ... THEN** instruction.

In Section 1 of this chapter, we discussed the notion of a loop. In this section, we have discussed decision-making. The WHILE...WEND pair of statements combines the two procedures. This statement pair has the form:

```
WHILE <expression>
.
.
.
WEND
```

The statements in between WHILE and WEND are repeated so long as <expression> is true. Note, however, that the statements between WHILE

and WEND may never be executed. If <expression> is initially false, the program skips to the next statement after WEND. The WHILE...WEND pair is useful in executing loops for which you cannot specify in advance the number of repetitions.

Example 2′. Rewrite the program of Example 2 using the WHILE...WEND pair of statements.
Solution. Here is the program adaptation.

```
10 PI = 3.14159
20 R = 1 :                  ' R=radius
30 WHILE A < 500
40       A = PI*R^2 :       ' A=area
50       PRINT R
60       R = R + 1 :        ' Go to next radius
70 WEND:                    ' Repeat
100 END
```

Example 3. A school board race involves two candidates. The returns from the four wards of the town are as follows:

	Ward 1	Ward 2	Ward 3	Ward 4
Mr. Thompson	487	229	1540	1211
Ms. Wilson	1870	438	110	597

Calculate the total number of votes achieved by each candidate, the percentage achieved by each candidate, and decide who won the election.
Solution. Let A1, A2, A3, and A4 be the totals for Mr. Thompson in the four wards; let B1-B4 be the corresponding numbers for Ms. Wilson. Let TA and TB denote the total votes, respectively, for Mr. Thompson and Ms. Wilson. Here is our program:

```
10 A1 = 487: A2 = 229: A3 = 1540: A4 = 1211
20 B1 = 1870: B2 = 438: B3 = 110: B4 = 597
30 TA = A1+A2+A3+A4 :          'Total for Thompson
40 TB = B1+B2+B3+B4 :          'Total for Wilson
50 T = TA + TB :               'Total Votes Cast
60 PA = 100*TA/T :             'Percentage for Thompson
70 ' TA/T is the ratio of votes for Thompson.
80 ' Multiply by 100 to convert to a percentage.
90 PB = 100*TB/T :             'Percentage for Wilson
100 A$ = "THOMPSON"
110 B$ = "WILSON"
120 ' Lines 130-150 print the percentages of the
    candidates
130 PRINT "CANDIDATE","VOTES","PERCENTAGE"
140 PRINT A$,TA,PA
150 PRINT B$,TB,PB
160 ' Lines 170-400 decide the winner.
170 IF TA > TB THEN 300:       'Thompson wins
180 IF TA < TB THEN 400:       'Wilson wins
```

```
190 PRINT A$, "AND", B$, "ARE TIED!":   'Otherwise a tie
200 GOTO 1000:                  'End
300 PRINT A$, "WINS"
310 GOTO 1000                   'End
400 PRINT B$, "WINS"
1000 END
```

Note the logic used for deciding who won. In line 170 we compare the votes TA and TB. If TA is the larger, then A (Thompson) is the winner. We then go to 300, print the result, and END. On the other hand, if TA > TB is *false*, then either B wins, or the two are tied. According to the program, if TA > TB is false, we go to line 180, where we determine if TA < TB. If this is true, then B is the winner, we go to 400, print the result, and END. On the other hand, if TA < TB is false, then the only possibility left is that TA = TB. According to the program, if TA = TB, we go to 200, where we print the proper result, and then END.

Infinite Loops and Fn-Break

As we have seen above, it is very convenient to be able to execute a loop without knowing in advance how many times the loop will be executed. However, with this convenience comes a danger. It is perfectly possible to create a loop which will be repeated an infinite number of times! For example, consider the following program:

```
10 LET N = 1
20 PRINT N
30 LET N = N+1
40 GOTO 20
50 END
```

The variable N starts off at 1. We print it and then increase N by 1 (to 2), print it, increase N by 1 (to 3), print it, and so forth. This program will go on forever! Such programs should clearly be avoided. However, even experienced programmers occasionally create infinite loops. When this happens, there is no need to panic. There is a way of stopping the computer. Just press the **Fn** and **Break** keys simultaneously. (In the following we will refer to this key combination as **Fn-Break**. This key sequence will interrupt the program currently in progress and return the computer to the command mode. The computer is then ready to accept a command from the keyboard. Note, however, that any program in RAM is undisturbed.

TEST YOUR UNDERSTANDING 4

Type the above program, RUN it, and stop it using the Fn-Break key combination. After stopping it, RUN the program again.

The INPUT Statement

It is very convenient to have the computer request information from you while the program is actually running. This can be accomplished via the **INPUT** statement. To see how, consider the statement:

```
570 INPUT A
```

When the computer encounters this statement in the course of executing the program, it displays a ? and waits for you to respond by typing the desired value of A (and then hitting the **ENTER** key). The computer then sets A equal to the numeric value you specified and continues running the program.

You may use an **INPUT** statement to specify the values of several different variables at one time. These variables may be numeric or string variables. For example, suppose that the computer encounters the statement:

```
50 INPUT A,B,C$
```

It will display:

```
?
```

You then type in the desired values for A, B, and C$, in the same order as in the program, and separate them by commas. For example, suppose that you type

```
10.5, 11.42, BEARINGS
```

followed by an **ENTER**. The computer then will set

```
A = 10.5, B = 11.42, C$ = "BEARINGS"
```

If you respond to the above question mark by typing only a single number, 10.5, for example, the computer will respond with:

```
? Redo from start
?
```

to indicate that you should repeat the input from the beginning. If you attempt to specify a string constant where you should have a numeric constant, the computer will respond with the message

```
? Redo from start
?
```

and will wait for you to repeat the **INPUT** operation.

It is helpful to include a prompting message which describes the input the computer is expecting. To do so, just put the message in quotation marks after the word **INPUT** and place a semicolon after the message (before the list of variables to be input). For example, consider the statement:

```
175 INPUT "ENTER COMPANY, AMOUNT"; A$, B
```

When the computer encounters this program line, the dialog will be as follows:

```
ENTER COMPANY, AMOUNT? AJAX OFFICE SUPPLIES, 2579.48
```

The underlined portion indicates your response to the prompt. The computer will now assign these values:

```
A$ = "AJAX OFFICE SUPPLIES", B = 2579.48
```

TEST YOUR UNDERSTANDING 5 (answer on page 124)
Write a program which allows you to set variables A and B to any desired values via an **INPUT** statement. Use the program to set A equal to 12 and B equal to 17.

The next two examples illustrate the use of the **INPUT** statement and provide further practice in using the **IF ... THEN** statement.

Example 4. You are a teacher compiling semester grades. Suppose there are four grades for each student and that each grade is on the traditional 0 to 100 scale. Write a program which accepts the grades as input, computes the semester average, and assigns grades according to the following scale:

90-100 A
80-89.9 B
70-79.9 C
60-69.9 D
< 60 F

Solution. We will use an **INPUT** statement to enter the grades into the computer. Our program will allow you to compute the grades of the students, one after the other, via a loop. You may terminate the loop by entering a negative grade. Here is our program.

```
10 PRINT "ENTER STUDENT'S 4 GRADES."
20 PRINT "SEPARATE GRADES BY COMMAS."
30 PRINT "FOLLOW LAST GRADE WITH ENTER."
40 PRINT "TO END PROGRAM, INCLUDE NEGATIVE GRADE."
50 INPUT A1,A2,A3,A4
60 IF A1 < 0 THEN 400
70 IF A2 < 0 THEN 400
80 IF A3 < 0 THEN 400
90 IF A4 < 0 THEN 400
100 LET A = (A1+A2+A3+A4)/4
110 PRINT "SEMESTER AVERAGE", A
120 IF A >= 90 THEN PRINT "SEMESTER GRADE = A" :GOTO 10
130 IF A >= 80 THEN PRINT "SEMESTER GRADE = B" :GOTO 10
```

```
140 IF A >= 70 THEN PRINT "SEMESTER GRADE = C" : GOTO 10
150 IF A >= 60 THEN PRINT "SEMESTER GRADE = D" : GOTO 10
160 PRINT "SEMESTER GRADE = F" :GOTO 10
400 END
```

Note the logic for printing out the semester grades. First compute the semester average A. In line 120 we ask if A is greater than or equal to 90. If so, we assign the grade A and go on to the next line, 130, which sends us to line 10 to obtain the next grade . In case A is less than 90, line 120 sends us to line 140. In line 140, we ask if A is greater than or equal to 80. If so, then we assign the grade B. (The point is that the only way we can get to line 140 is for A to be less than 90. So if A is greater than or equal to 80, we know that A lies in the B range.) If not, we go to line 160, and so forth. This logic may seem a trifle confusing at first, but after repeated use, it will seem quite natural.

Example 5. Write a program to maintain your checkbook. The program should allow you to record an initial balance, enter deposits, and enter checks. It also should warn you of overdrafts.

Solution. Let the variable B always contain the current balance in the checkbook. The program will ask for the type of transaction you wish to record. A "D" will mean that you wish to record a deposit; a "C" will mean that you wish to record a check; a "Q" will mean that you are done entering transactions and wish to terminate the program. After entering each transaction, the computer will figure your new balance, report it to you, check for an overdraft, and report any overdraft to you. In case of an overdraft, the program will allow you to cancel the preceding check!

```
10 INPUT "WHAT IS YOUR STARTING BALANCE"; B
20 INPUT "WHAT TRANSACTION TYPE (D,C,or Q)"; A$
30 IF A$ = "Q" THEN 1000:      'End
40 IF A$ = "C" THEN 200
100 'Process Deposit
110 INPUT "DEPOSIT AMOUNT"; D
120 LET B = B + D :          ' Add desposit to balance
130 PRINT "YOUR NEW BALANCE IS", B
140 GOTO 20
200 'Process check
210 INPUT "CHECK AMOUNT"; C
220 LET B = B - C :         'Deduct check amount
230 IF B < 0 THEN 300 :     'Test for overdraft
240 PRINT "YOUR NEW BALANCE IS", B
250 GOTO 20
300 'Process overdraft
310 PRINT "LAST CHECK CAUSES OVERDRAFT"
320 INPUT "DO YOU WISH TO CANCEL CHECK(Y/N)"; E$
330 IF E$ = "Y" THEN 400
340 PRINT "YOUR NEW BALANCE IS", B
```

```
350 GOTO 20
400 'Cancel check
410 LET B = B + C:          'Cancel last check
420 GOTO 20
1000 END
```

You should scan this program carefully to make sure you understand how each of the **INPUT** and **IF ... THEN** statements are used. In addition, you should use this program to obtain a feel for the dialog between you and your computer when **INPUT** statements are used.

Note how the above program is divided into sections. For visual purposes, each section begins with a line number which is a multiple of 100. Moreover, each section begins with a comment which identifies the function of the section. In order to write a complex program, you should break the program into manageable sections. Don't get caught in a maze of complexity. Work out one section at a time and carefully comment on each section. Then put the various sections together into one program.

Example 6. Write a BASIC program which tests mastery in addition of two-digit numbers. Let the user suggest the problems, and let the program keep score of the number correct out of ten.

Solution. Request that the program user suggest pairs of numbers via an **INPUT** statement. The sum also will be requested via an **INPUT** statement. An **IF ... THEN** statement will be used to judge the correctness. The variable R will keep track of the number correct. We will use a loop to repeat the process ten times.

```
10 FOR N = 1 TO 10 :        'Loop to give 10 problems
20    INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30    INPUT "WHAT IS THEIR SUM"; C
40    IF A + B = C THEN 200
100   'Respond to incorrect answer
110   PRINT "SORRY. THE CORRECT ANSWER IS",A+B
120   GO TO 300 :           'Go to the next problem
200   'Respond to correct answer
210   PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
220   LET R = R+1 :         'Increase score by 1
300 NEXT N
400 'Print score for 10 problems
410 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
510 PRINT "TO TRY AGAIN, TYPE RUN"
600 END
```

More About INPUTting Data

The INPUT statement, as we have seen, may be used to input one or more constants (string or numeric) to a running program. However, the INPUT statement has a serious defect. To explain this defect, consider the following statement:

```
10 INPUT A$,B$
```

Suppose that you wish to set A$ equal to the string:

```
"Washington,George"
```

and B$ to the string:

```
"Jefferson,Thomas"
```

Suppose that you respond to the INPUT prompt by typing:

```
Washington,George, Jefferson,Thomas
```

BASIC will report an error:

```
? Redo from start
```

Here is the reason. INPUT looks for commas to separate the data items. The first comma occurs between "Washington" and "George". So INPUT assigns A$ the string "Washington" and B$ the string "George". But this gives excess data. So BASIC declares an error. There's a simple way around this. Whenever you wish to INPUT data containing a comma, surround the appropriate strings with quotation marks. In our example, the response

```
"Washington,George","Jefferson,Thomas"
```

will assign A$ and B$ as we wished.

It is something of a bother to surround strings with quotation marks, so BASIC provides another statement which is not sensitive to commas, namely LINE INPUT. The LINE INPUT statement may be used to assign only one variable at a time. It reads the input until it encounters ENTER. So, for example, suppose that we use the statement:

```
30 LINE INPUT A$
```

The computer waits for a response. Suppose that we respond with the string

```
Washington,George
```

and press ENTER. LINE INPUT then will assign A$ the string constant "Washington,George". LINE INPUT may be used only to input data to a string variable.

You may use a prompt with LINE INPUT exactly as you do with INPUT. For example, the statement

```
40 LINE INPUT "Type NAME?";A$
```

will result in the prompt:

```
Type NAME?
```

to which you would respond. Note that LINE INPUT does not automatically display a ? like the INPUT statement. In the above example, the ? came from the prompt.

There is a third statement which you may use to input data from the keyboard, namely INPUT$. This statement allows you to specify an input of only a specified length. For example, consider the statement:

```
10 A$=INPUT$(5)
```

It will cause the program to wait for five characters from the keyboard and will assign them to A$. For example, if you type GEORGE, then A$ will be assigned the string constant "GEORGE". INPUT$ is a more specialized statement than either INPUT or LINE INPUT because of the following facts:

1. INPUT$ does not automatically display the input characters on the screen. If you want them displayed, it is your responsibility to display them.
2. INPUT$ accepts all keyboard characters, including **Backspace** and ENTER. In particular, it does not allow you to correct your input.

If you are a beginning programmer, it's probably wisest to stick to INPUT and LINE INPUT, but we mention INPUT$ mainly for completeness.

Exercises (answers on page 353)

1. Write a program to calculate all perfect squares which are less than 45,000. (Perfect squares are the numbers 1, 4, 9, 16, 25, 36, 49,... .)
2. Write a program to determine all of the circles of integer radius and area less than or equal to 5,000 square feet. (The area of a circle of radius R is PI*R^2, where PI = 3.14159, approximately.)
3. Write a program to determine the sizes of all those boxes which are perfect cubes, have integer dimensions, and have volumes of less than 175,000 cubic feet. (That is, find all integers X for which X^3 is less than 175,000.)
4. Modify the arithmetic testing program of Example 6 so that the operation tested is for multiplication instead of addition.
5. Modify the arithmetic testing program of Example 6 so that it allows you to choose, at the beginning of each group of ten problems, from among these operations: addition, subtraction, or multiplication.
6. Write a program which accepts three numbers via an INPUT statement and determines the largest of the three.
7. Write a program which accepts three numbers via an INPUT statement and determines the smallest of the three.
8. Write a program which accepts a set of numbers via INPUT statements and determines the largest among them.

9. Write a program which accepts a set of numbers via INPUT statements and determines the smallest among them.
10. The following data were collected by a sociologist. Six cities experienced the following numbers of burglaries in 1980 and 1981:

City	Burglaries 1980	Burglaries 1981
A	5,782	6,548
B	4,811	6,129
C	3,865	4,270
D	7,950	8,137
E	4,781	4,248
F	6,598	7,048

For each city, calculate the increase (decrease) in the number of burglaries. Determine which had an increase of more than 500 burglaries.

11. Write a program which does the arithmetic of a cash register. That is, let the program accept purchases via INPUT statements, then total the purchases, figure out the sales tax (assume 5 percent), and compute the total purchase. Let the program ask for the amount of payment given, and then let it compute the change due.
12. Write a program which analyzes cash flow. Let the program ask for cash on hand as well as accounts expected to be received in the next month. Let the program also compute the total anticipated cash for the month. Let the program ask for the bills due in the next month, and let it compute the total accounts payable during the month. By comparing the amounts to be received and to be paid out, let the program compute the net cash flow for the month and report either a surplus or a deficit.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a.
```
IF A<B THEN PRINT A+B ELSE END
```
b.
```
IF A^2+D>=5000 THEN 300 ELSE 500
```
c.
```
IF N>I+K THEN N=I+K ELSE N=K
```

2:
```
10 B = <put unpaid balance here>
20 IF B<=500 THEN IN= .015*B: GOTO 300
100 LET C=B-500
110 IN=.015*500 + .01*C
300 PRINT "INTEREST EQUALS";IN
310 PRINT "NEW BALANCE EQUALS";B+IN
320 END
```

3:
```
100-110-120-400
```

5:
```
10 INPUT "THE VALUES OF A AND B ARE";A,B
20 END
```

5.3 Structuring Solutions to Problems

You may have noticed our programs getting longer. There is no way around this. In order to use the computer to solve real-life problems, programs often must be quite long and must use the full range of capabilities of the computer. This poses a number of problems:

1. Long programs are difficult to plan.
2. Long programs are difficult to write and correct.
3. Long programs are hard to read.

All three problems will confront you in programming your computer. Let's discuss some ways to deal with them.

As an example of program planning, let's take the last program of the preceding section. Recall that this is the program which tests addition. Suppose that you are given the job of building such a program. How should you proceed? Your first inclination might be to start writing BASIC statements. At all costs, resist the temptation! Your first job is to plan the program.

The first step in program planning is to decide on the input and output. What data does the user give and what responses does the computer give? Make a list:

User input: Answers to questions
Computer output: Questions to answer
Responses to answers
a. Response to correct answer
b. Response to incorrect answer
c. Report of score
Question: Another set of problems?

The next step is to organize these inputs and outputs into a sequence of steps which follow one another in logical order. Don't worry about computer instructions at this point. Rather, describe reasonably general steps which, in the end, may actually correspond to several computer instructions. Here is how our addition program might be described.

1. Computer requests question
2. User responds
3. Computer requests answer.
4. User enters answer
5. Computer analyzes answers and responds
   a. Reports whether answer is correct
   b. Keeps score
6. Steps 1-4 are repeated 10 times
7. Computer reports score
8. Computer queries user whether to begin again.

The third step of program planning is to sketch out the structure of the program. We see from step 6 that we will need a loop to keep track of the problems. Moreover, we know that steps 1-4 are one line computer commands. Let's lump them together into one section of the program. On the

other hand, handling correct answers is different from handling wrong answers. Let's have a separate section of the program for each of these tasks. Moreover, let's have a separate section of the program for steps 7 and 8. You should write all this down (on paper) as follows:

```
10 FOR N=1 TO 10

(lines 20-80 are reserved for steps 1 to 3.)

100   'Respond to incorrect answer
200   'Respond to correct answer
300 NEXT N
400 'Print score for 10 problems
500 'Run again?
600 END
```

As the fourth step, you should begin to fill in the various steps in the above outline. Here is where you may begin writing BASIC instructions, defining variables, and so forth. Each of the steps corresponds to only a few program statements. So the program becomes easy to write and our final product is something like the following program.

```
10 FOR N = 1 TO 10 :       'Loop to give 10 problems
20    INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30    INPUT "WHAT IS THEIR SUM"; C
40    IF A+B=C THEN 200
100   'Respond to incorrect answer
110   PRINT "SORRY. THE CORRECT ANSWER IS",A+B
120   GO TO 300 : 'Go to the next problem
200   'Respond to correct answer
210   PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
220   LET R=R+1 : 'Increase score by 1
300 NEXT N:             'Go to next problem
400 'Print score for 10 problems
410 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
500 'Run again?
510 PRINT "TO TRY AGAIN, TYPE RUN"
600 END
```

It is possible that some of the steps correspond to complex sequences of operations. If so, break such steps into smaller steps, just like we have done for the entire program. Eventually, you should reduce your program to an organized sequence of steps, each of which corresponds to no more than about a dozen statements. (The actual number may be more or less, corresponding to your comfort level. But don't allow the number to be too large. This is the way errors creep into your program!)

In organizing a program, you cannot plan the various steps in total isolation from one another. Here are some pitfalls to be aware of:

1. If a variable is to be used in two steps, then it must be given by the same name in each.
2. If a step assumes that the value of a variable has been assigned in a previous step, be sure that this is done.
3. Don't mistakenly use the same variable to mean two different things. This is an easy error to make. After several hours at the keyboard, you may forget that you already used a variable name to mean something else. No harm is done if the two variables are used in two isolated sections of the program. However, you may set the variable with one meaning in mind, only to have the program then use it with the other meaning. This can make your results incorrect.
4. Be sure to assign each variable its proper starting value. (This is called **variable initialization**.) Remember that if you do not assign a value to a variable, then BASIC will assign it the value 0. It is good programming practice to even assign these zero values explicitly.

The procedure for program planning described above automatically incorporates your documentation into your program. This makes it easier to read your program to correct mistakes or to alter it at a later date.

The discussion of this section just scratches the surface of the subject of program planning. Hopefully, it will ease the burden of writing and understanding BASIC programs and will lead you to develop your own approach to program planning and organization. We'll have more to say about the subject in Chapter 7.

5.4 Subroutines

In writing programs it is often necessary to use the same sequence of instructions more than once. It may not be convenient (or even feasible) to retype the set of instructions each time it is needed. Fortunately, BASIC offers a convenient alternative: the subroutine.

A **subroutine** is a program which is incorporated within another, larger program. The subroutine may be used any number of times by the larger program. Often, the lines corresponding to a subroutine are isolated toward the end of the larger program. This arrangement is illustrated in Figure 5-1. The arrow to the subroutine indicates the point in the larger program at which the subroutine is used. The arrow pointing away from the subroutine indicates that, after completion of the subroutine, execution of the main program resumes at the point at which it was interrupted.

Subroutines are handled with the pair of instructions **GOSUB** and **RETURN**. The statement

```
100 GOSUB 1000
```

sends the computer to the subroutine which begins at line 1000. The computer starts at line 1000 and carries out statements in order. When a **RETURN** statement in the subroutine is reached, the computer goes back to the main program, starting at the first line after 100.

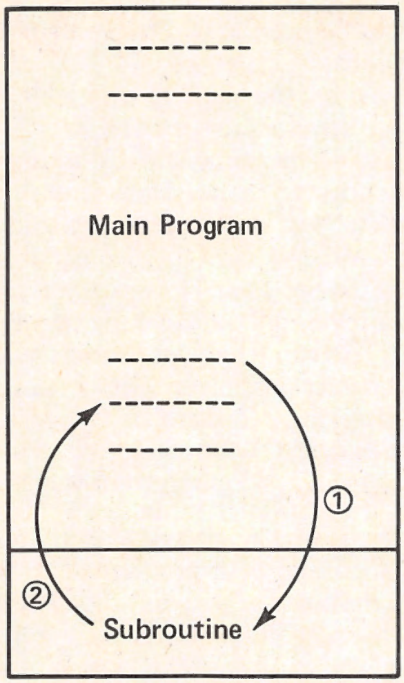


Figure 5-1. A subroutine.

Subroutines may serve as user-defined commands, as Example 1 illustrates.

TEST YOUR UNDERSTANDING 1 (answer on page 133)
Consider the following program.

```
10 GOSUB 40
20 PRINT "LINE 20"
30 END
40 PRINT "LINE 40"
50 RETURN
```

List the line numbers in order of execution.

Example 1. Design a BASIC subroutine which erases a specified line of the screen and positions the cursor at the left end of the erased line.

Solution. The task described is required by many programs. It is a prelude to writing on the line. In fact, it may be required many times within the same program. It would be wasteful to write separate instruction lines for each repetition. So let's write a general subroutine which can be called whenever required. Suppose that line L is to be erased. This may be accomplished in the following steps:

1. Position the cursor at the left-most position on the line.
2. Write 80 spaces (40 if you are in WIDTH 40).
3. Reposition the cursor at the left-most position on the line.

To position the cursor at row r and column c, we use the statement:

```
LOCATE r,c
```

We could use a loop to generate the spaces. However, there is an easier way. The string SPACE$(n) is a string of n spaces. By printing this string, we may "blank out" n spaces beginning at the current cursor position. Here is our subroutine:

```
5000 'Blank out line L
5010 LOCATE L,1:   'Position cursor at left of line L
5020 PRINT SPACE$(80)
5030 LOCATE L,1
5040 RETURN
```

Whenever we wish to use this subroutine, we first set the value of L to the line number to be erased. Next, we execute GOSUB 5000. Note that the value of L must be set before the GOSUB 5000 instruction is issued.

TEST YOUR UNDERSTANDING 2 (answer on page 133)
Write a subroutine which erases the first M columns of line L and positions the cursor in the upper left corner of the screen.

Example 2. Write a program which turns the computer into an electronic cash register. The program should accept as entries both taxable and non-taxable amounts. It should keep track of the totals of each. On command, it should display the totals, compute the tax, and compute the grand total owed.
Solution. A listing of the program is included below. This program illustrates some of the tricks involved in planning "user-friendly" programs. We allow the user to choose from four requests displayed on the screen. Such a display is called a **menu**. Here are the four possible requests.

1. **New Customer.** Zero all totals, clear the screen and display identifying headings as in Figure 5-2.
2. Enter item. Accept the amount of an item. The program asks whether the item is taxable. The amount is displayed under the appropriate heading and is added to the appropriate total (taxable or non-taxable).
3. Compute totals. Compute tax and display totals.
4. Exit. End program.

The instructions for displaying the menu are in lines 1000-1080. Note that the subroutine to blank out a particular line begins in line 6000. In many parts of the program, we will want to write on a line. We will use the subroutine beginning in line 6000 to erase that particular line.

In line 1100, the user is asked to make a choice of activity from the menu by typing one of the numbers 1-4. Based on the user response, the program goes to the subroutine at 2000, 3000, or 4000, or, in the case of choice 4, goes to line 5000 (no subroutine—more about that below). After the subroutine is

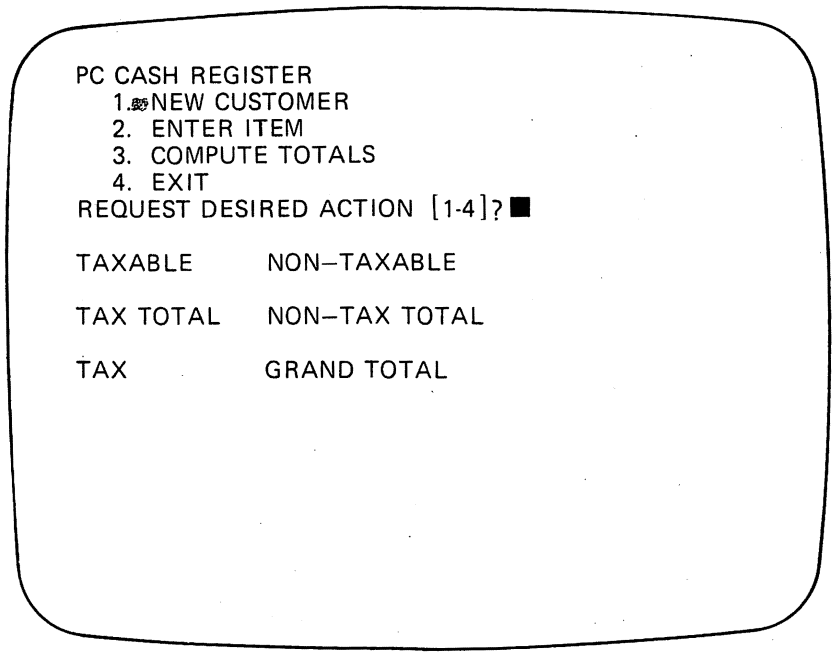


Figure 5-2. Screen Layout for the PCjr Cash Register.

executed, the program returns to the line after the one calling the subroutine and will make its way to line 1150. This line sends the program back to 1090, which erases line 7 and requests another activity. You may use the program all day. You may end the program by choosing option 4 on the menu.

Option 4 on the menu causes the program to go to line 5000, where the screen is cleared and the program is terminated. We did not use a subroutine to get to line 5000 since we did not expect to return. In this particular case, a GOSUB 5000 could have been used with no harm. Since the program ends, the computer will not look for a place to return. However, it is good programming practice to use a subroutine only in instances in which the program is guaranteed to reach a RETURN instruction.

```
1000 'Display Menu
1010   CLS
1020   LOCATE 1,1: 'Home cursor
1030   PRINT "PCjr CASH REGISTER"
1040   PRINT "      1. NEW CUSTOMER"
1050   PRINT "      2. ENTER ITEM"
1060   PRINT "      3. COMPUTE TOTALS"
1070   PRINT "      4. EXIT"
1080   PRINT
1090   L=7:GOSUB 6000:   'Blank out entry line
1100   INPUT "REQUEST DESIRED ACTION (1-4)";REPLY$
1110   IF REPLY$="1" THEN GOSUB 2000
1120   IF REPLY$="2" THEN GOSUB 3000
1130   IF REPLY$="3" THEN GOSUB 4000
1140   IF REPLY$="4" THEN 5000
1150   GOTO 1090
```

```
2000 'New customer subroutine
2010   'Reset totals
2020   TAXTOTAL=0:NONTAXTOTAL=0:GRANDTOTAL=0
2030   'Blank out lines 8-24 of screen
2040   FOR L=8 TO 20
2050           LOCATE L,1
2060           PRINT SPACE$(40)
2070   NEXT L
2080   'Print titles
2090   LOCATE 9,1
2100   PRINT "TAXABLE","NON-TAXABLE"
2110   LOCATE 12,1
2120   PRINT "TAX TOTAL","NON-TAX TOTAL"
2130   LOCATE 15,1
2140   PRINT "TAX", "GRAND TOTAL"
2150 RETURN
3000 'Enter item subroutine
3010   L=7:GOSUB 6000:       'Clear entry line
3020   INPUT "AMOUNT (NO DOLLAR SIGN)"; AMOUNT
3030   L=7:GOSUB 6000
3040   INPUT "TAXABLE=1,NON-TAXABLE=0";STATUS
3050   L=10:GOSUB 6000
3060   IF STATUS=1 THEN PRINT AMOUNT,""
3070   IF STATUS=0 THEN PRINT "",AMOUNT
3080   IF STATUS=1 THEN TAXABLE=TAXABLE+AMOUNT
3090   IF STATUS=0 THEN NONTAXABLE=NONTAXABLE+AMOUNT
3100 RETURN
4000 'Compute totals subroutine
4010   L=10:GOSUB 6000:       'Clear entry line
4020   L=13:GOSUB 6000:       'Clear first total line
4030   PRINT TAXABLE,NONTAXABLE
4040   TAX=.05*TAXABLE
4050   GRANDTOTAL=TAXABLE+TAX+NONTAXABLE
4060   L=16:GOSUB 6000
4070   PRINT TAX,GRANDTOTAL
4080 RETURN
5000 'Exit subroutine
5010 CLS
5020 END
6000 'Clear entry line L
6010   LOCATE L,1
6020   PRINT SPACE$(40): 'Clear Entry Line
6030   LOCATE L,1
6040 RETURN
```

TEST YOUR UNDERSTANDING 3 (answer on page 133)
Enhance the program of Example 2 so that as a part of computing the totals, it asks you the amount presented ($10 bill, $20 bill, and so forth) and computes the change.

Nested Subroutines

In Example 2, we used a number of subroutines which were contained within subroutines. For example, lines 4000-4080 are a subroutine. However, on lines 4020 and 4030, we called the subroutine at 6000. Such subroutines are said to be **nested**. BASIC is able to handle such nesting. You may use nesting to any level. (A subroutine within a subroutine within a subroutine, and so forth.) However, you should be aware that a RETURN instruction always refers to the **innermost** subroutine. To put it another way, a RETURN always refers to the subroutine which was called most recently.

Caution: It is possible to accidentally create an infinite nesting of subroutines by repeatedly issuing GOSUB instructions, as in this program:

```
10 GOTO 20
20 GOSUB 10
```

The computer eventually will run out of memory to keep track of this nesting and an error will result.

The ON ... GOSUB Instruction

In Example 2, we organized the program around four main subroutines, corresponding to the four possible choices on the MENU. It took several instructions to properly channel the program to the proper subroutine. BASIC provides a convenient shortcut for use in such situations: the ON...GOSUB instruction. The form of this instruction is:

```
ON <expression> GOSUB <line1>,<line2>,...
```

When BASIC encounters this instruction, it evaluates <expression>, which should yield an integer value; if the resulting value is 1, the program executes a GOSUB to <line1>; if the value is 2, the program executes a GOSUB to <line2>, and so forth. If the value is zero or more than the number of line numbers provided, the instruction will be ignored. (If <expression> yields a negative value or an integer value larger than 255, an Illegal Function Call error results.)

For example, lines 1110-1130 of Example 2 may be replaced by the single line:

```
1110 ON VAL(REPLY$) GOSUB 2000,3000,4000
```

Here, the expression VAL(REPLY$) converts the string REPLY$ into its numeric equivalent ("1" converts to 1, "2" to 2, and so forth). If this value is 1, the program executes a GOSUB 2000, if the value is 2, a GOSUB 3000, and if the value is 3, a GOSUB 4000.

Exercises (answers on page 355)

1. Write a subroutine which prints 10 asterisks (*) beginning at the left-most column of row L.
2. Write a subroutine which prints M asterisks beginning at the left-most column of row L.
3. Write a subroutine which prints M asterisks beginning at column K of row L.
4. Write a program which uses the subroutine of Exercise 3 to print rows of asterisks corresponding to K=5, L=3, M=30; K=4, L=5, M=35; K=8, L=7, and M=12.
5. Consider this instruction:

```
10 ON J-2 GOSUB 100,200,300,400 500
```

   What will be its effect if:
   (a) J=4 (b) J=7 (c) J=2 (d)J=10 (e) J=0 ?
6. Consider the program

```
10 Y=5
20 J=3
30 S=Y-J
40 ON S GOSUB 100,200,300,400
50 CLS
60 END
100 RETURN
200 RETURN
300 RETURN
400 RETURN
```

   What are the two lines executed immediately after line 40?

ANSWERS TO TEST YOUR UNDERSTANDING

1: 10-40-50-20-30

2:
```
5000 'Blank out 1st M columns of line L
5010 LOCATE L,1:     'Position cursor at left of line L
5020 PRINT SPACE$(M)
5030 LOCATE 1,1
5040 RETURN
```

3: Add the following program lines

```
4071 LOCATE 20,1
4072 INPUT "AMOUNT PAID";PAID
4073 PRINT "PAID","CHANGE"
4074 PRINT PAID,PAID-GRANDTOTAL
```

6

WORKING WITH DATA

6.1 Working With Tabular Data—Arrays

In Chapter 4 we introduced the notion of a variable and used variable names like:

AA, B1, CZ, W0

Unfortunately, the supply of variables available to us is not sufficient for many programs. Indeed, as we shall see in this chapter, there are relatively innocent programs which require hundreds or even thousands of variables. To meet the needs of such programs, BASIC allows for the use of so-called **subscripted variables**. Such variables are used constantly by mathematicians and are identifed by numbered subscripts attached to a letter. For instance, here is a list of 1000 variables as they might appear in a mathematical work:

$A_1, A_2, A_3, \ldots, A_{1000}.$

The numbers used to distinguish the variables are called **subscripts**. Likewise, the BASIC language allows definition of variables to be distinguished by subscripts. However, since the computer has difficulty placing the numbers in the traditional position, they are placed in parentheses on the same line as the letter. For example, the above list of 1000 different variables would be written in BASIC as

```
A(1), A(2), A(3) ,..., A(1000).
```

Please note that the variable A(1) is not the same as the variable A1. You may use both of them in the same program and BASIC will interpret them as being different.

A subscripted variable is really a group of variables with a common letter identification distinguished by different integer "subscripts." For instance, the above group of variables would constitute the subscripted variable A( ). It is often useful to view a subscripted variable as a table or array. For example, the

subscripted variable A( ) considered above can be viewed as providing the following table of information:

A(1)
A(2)
A(3)
.
.
.
A(1000)

As shown here, the subscripted variable defines a table consisting of 1000 rows. Suppose that J is an integer between 1 and 1000. Then row number J contains a single entry; namely, the value of the variable A(J). The first row contains the value of A(1), the second the value of A(2), and so forth. Since a subscripted variable can be thought of as a table (or array), subscripted variables often are called **arrays**.

The array shown above is a table consisting of 1000 rows and a single column. PCjr BASIC allows you to consider more general arrays. For example, consider the following financial table which records the daily income for three days from each of a chain of four computer stores:

	Store #1	Store #2	Store #3	Store #4
Day 1	1258.38	2437.46	4831.90	987.12
Day 2	1107.83	2045.68	3671.86	1129.47
Day 3	1298.00	2136.88	4016.73	1206.34

This table has three rows and four columns. Its entries may be stored in the computer as a set of 12 variables:

A(1,1) A(1,2) A(1,3) A(1,4)
A(2,1) A(2,2) A(2,3) A(2,4)
A(3,1) A(3,2) A(3,3) A(3,4)

This array of variables is very similar to a subscripted variable, except that there are now two subscripts. The first subscript indicates the row number and the second subscript indicates the column number. For example, the variable A(3,2) is in the third row, second column. A collection of variables such as that given above is called a **two-dimensional array** or a **doubly-subscripted variable**. Each setting of the variables in such an array defines a tabular array. For example, if we assign the values:

A(1,1) = 1258.38, A(1,2) = 2437.46,
A(1,3) = 4831.90, and so forth,

then we will have the table of earnings from the computer store chain.

So far, we have only considered numeric arrays—arrays whose variables can assume only numerical values. However, it is possible to have arrays with

variables that assume string values. (Recall that a string is a sequence of characters: letter, numeral, punctuation mark, or other printable keyboard symbol.) For example, here is an array which can contain string data:

A$(1)
A$(2)
A$(3)
A$(4)

Here the dollar signs indicate that each of the variables of the array is a string variable. If we assign the values

A$(1) = "SLOW", A$(2) = "FAST", A$(3) = "FAST", A$(4) = "STOP"

then the array is this table of words:

SLOW
FAST
FAST
STOP

Similarly, the employee record table

Social Security Number	Age	Sex	Marital Status
178654775	38	M	S
345861023	29	F	M
789257958	34	F	D
375486595	42	M	M
457696064	21	F	S

may be stored in an array of the form B$(I,J), where I assumes any one of the values 1, 2, 3, 4, 5 (I is the row), and J assumes any one of the values 1, 2, 3, 4 (J = the column). For example, B$(1,1) has the value "178654775", B$(1,2) has the value "38", B$(1,3) has the value "M", and so forth.

Note that you may not mix numerical and string entries in the same array. In the example above, it is necessary to store the numbers as strings since the array is defined as a string array. If we wished to store the above numerical data as numerical constants, we could have used four different arrays—two numerical arrays for the first two columns, and two string arrays for the last two columns.

The PCjr even allows you to have arrays with three, four, or even more subscripts. For example, consider the computer store chain array introduced above. Suppose that we had one such array for each of ten consecutive three-day periods. This collection of data could be stored in a three-dimensional array of the form C(I,J,K), where I and J represent the row and column, just as before, and K represents the particular three-day period. (K could assume the values 1, 2, 3 ,..., 10.)

An array may involve up to 255 dimensions. The subscripts corresponding to each dimension may assume values from 0 to 32767. For all practical applications, any size array is permissible.

You must inform the computer of the sizes of the arrays you plan to use in a program. This allows the computer to allocate memory space to house all the values. To specify the size of an array, use a **DIM** (dimension) statement. For example, to define the size of the subscripted variable A(J), J = 1,...,1000, we insert the statement

```
10 DIM A(1000)
```

in the program. This statement informs the computer to expect variables A(0), A(1), ..., A(1000) in the program and that it should set aside memory space for 1001 variables. Note that, in the absence of further instructions from you, BASIC begins all subscripts at 0. If you wish to use A(0), fine. If not, ignore it.

You need not use all the variables defined by a **DIM** statement. For example, in the case of the **DIM** statement above, you might actually use only the variables A(1), ..., A(900). Don't worry about it! Just make sure that you have defined enough variables. Otherwise you could be in trouble. For example, in the case of the subscripted variable above, your program might make use of the variable A(1001). This will create an error condition. Suppose that this variable is used first in line 570. When you attempt to run the program, the computer will report:

```
Subscript out of range in 570
```

Moreover, execution of the program will be halted. To fix the error, merely redo the **DIM** statement to accommodate the undefined subscript.

To define the size of a two-dimensional array, use a **DIM** statement of the form:

```
10 DIM A(5,4)
```

This statement defines an array A(I,J), where I can assume the values 0, 1, 2, 3, 4, 5, and J can assume the values 0, 1, 2, 3, 4. Arrays with three or more subscripts are defined similarly.

TEST YOUR UNDERSTANDING 1 (answer on page 142)

Here is an array.

12 645.80
148 489.75
589 12.89
487 14.50

a. Define an appropriate subscripted variable to store this data.
b. Define an appropriate **DIM** statement.

It is possible to dimension several arrays with one **DIM** statement. For example, the dimension statement

```
10 DIM A(1000), B$(5), A(5,4)
```

defines the array A(0), ..., A(1000), the string array B$(0), ..., B$(5) and the two-dimensional array A(I,J), I=0, ..., 5; J=0, ..., 4.

We now know how to set aside memory space for the variables of an array. We must next take up the problem of assigning values to these variables. We could use individual **LET** statements, but with 1000 variables in an array, this could lead to an unmanageable number of statements. There are more convenient methods which make use of loops. The next two examples illustrate two of these methods.

Example 1. Define an array A(J), J=1, 2, ..., 1000 and assign the following values to the variables of the array:

A(1)=2, A(2)=4, A(3)=6, A(4)=8, ...

Solution. We wish to assign each variable a value equal to twice its subscript. That is, we wish to assign A(J) the value 2*J. To do this we use a loop:

```
10 DIM A(1000)
20 FOR J = 1 TO 1000
30 A(J) = 2*J
40 NEXT J
50 END
```

Note that the program ignores the variable A(0). Like any variable which has not been assigned a value, it has the value zero.

TEST YOUR UNDERSTANDING 2 (answer on page 142)
Write a program which assigns the variables A(0), ..., A(30) the values A(0)=0, A(1)=1, A(2)=4, A(3)=9, ...

When the computer is first turned on or reset, all variables (including those in arrays) are cleared. All numeric variables are set equal to 0, and all string variables are set equal to the null string (the string with no characters in it). If you wish to return all variables to this state during the execution of a program, use the **CLEAR** command. For example, when the computer encounters the command

```
570 CLEAR
```

it will reset all the variables. The **CLEAR** command can be convenient if, for example, you wish to use the same array to store two different sets of information at two different stages of the program. After the first use of the array you could then prepare for the second use by executing a **CLEAR**.

Example 2. Define an array corresponding to the following table of examination grades. Input the values given, print the table on the screen, and calculate the average grade.

Sally Smith
Examination Grades

95
78
85
87
80
70

Solution. Our program will print the headings of the table and then READ the table entries into a numerical array A(J), J=1,2,...,6. We dimension the array as A(6).

```
10  DIM A(6)
20  FOR J=1 TO 6
30    READ A(J)
40  NEXT J
50  CLS
60  PRINT "Sally Smith"
70  PRINT "Examination Grades"
80  FOR J=1 TO 6
90    PRINT A(J)
100  NEXT J
110 END
200 DATA 95, 78, 85, 87, 80, 70
```

TEST YOUR UNDERSTANDING 3 (answer on page 142)
Suppose that your program uses a 9x2 array A$(I,J), a 9x1 array B$(I,J), and a 9x5 array C(I,J). Write an appropriate DIM statement(s).

If you plan to dimension an array, you should always insert the **DIM** statement before the variable first appears in your program. Otherwise, the first time BASIC comes across the array, it will assume that the subscripts go from 0 to 10. If it subsequently comes across a **DIM** statement, it will think you are changing the size of the array in the midst of the program, something which is not allowed. If you try to change the size of an array in the middle of a program, you will get this error message:

```
Duplicate Definition
```

In our discussion above, we have been very casual about ignoring unused subscripts, such as A(0). In some programs, there may be so many large arrays

that memory space becomes precious. Sometimes, considerable memory space may be conserved by carefully planning which subscripts will be used and defining only those variables. You may eliminate unused 0 subscripts using the **OPTION BASE** statement. For example, the statement

```
10 OPTION BASE 1
```

begins all arrays with subscript 1. This statement must be used in a program prior to the dimensioning of any arrays.

Deleting Arrays

It is very simple to create an array which occupies a huge amount of memory space. For example, consider this seemingly harmless statement:

```
10 DIM A(10,10,10,10)
```

It defines an array with 10,000 entries. BASIC requires four bytes for each entry, so the array takes up 40,000 bytes of RAM! For this reason, you must do some planning so that your arrays do not overflow available memory. One technique for this involves deleting arrays from memory in order to make room for other arrays. You may do this using the ERASE statement. For example, to delete the above array, we could use the statement

```
20 ERASE A
```

Once you execute ERASE, all the values of array A are lost, and the DIM statement dimensioning A is cancelled. In particular, you may redimension an array after an ERASE statement.

The **ERASE** statement may be used to delete several arrays at once, as in this statement:

```
30 ERASE B,C,D
```

Exercises (answers on page 356)

For each of the following tables, define an appropriate array and determine the appropriate **DIM** statement.

1. 5
   2
   1.7
   4.9
   11
2. 1.1 2.0 3.5
   1.7 2.4 6.2
3. JOHN
   MARY
   SIDNEY
4. 1 2 3

5.
RENT	575.00
UTILITIES	249.78
CLOTHES	174.98
CAR	348.70

6. Display the following array on the screen:

   Receipts:

	Store #1	Store #2	Store #3
1/1-1/10	57,385.48	89,485.45	38,456.90
1/11-1/20	39,485.98	76,485.49	40,387.86
1/21-1/31	45,467.21	71,494.25	37,983.38

   (This exercise is easiest on an 80-character wide screen. If you are using a 40-character wide screen, it will be necessary to use PRINT TAB statements (see Section 6.3) to position the data into the proper screen positions.)
7. Write a program that displays the array of Exercise 6 along with totals of the receipts from each store.
8. Expand the program in Exercise 7 so that it calculates and displays the totals of ten day periods. (Your screen will not be wide enough to display the ten day totals in a fifth column, so display them in a separate array.)
9. Devise a program which keeps track of the inventory of an appliance store chain. Store the current inventory in an array of the form

	Store #1	Store #2	Store #3	Store #4
Refrig.				
Stove				
Air Cond.				
Vacuum				
Disposal				

Your program should: 1) input the inventory corresponding to the beginning of the day, 2) continually ask for the next transaction—the store number and the number of appliances of each item sold, and 3) in response to each transaction, update the inventory array.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. `A(I,J), I=1,2,3,4; J=1,2`
   b. `DIM A(4,2)`

2:
```
10 DIM A(30)
20 FOR J=0 TO 30
30 A(J)=J^2
40 NEXT J
50 END
```

3: `DIM A$(9,2),B$(9,1),C(9,5)`

6.2 Inputting Data

In the preceding section, we introduced arrays and discussed several methods for assigning values to the variables of an array. The most flexible method was via the **INPUT** statement. However, this can be a tedious method for large arrays. Fortunately, BASIC provides us with an alternate method for inputting data.

A given program may need many different numbers and strings. You may store the data needed in one or more **DATA** statements. A typical data statement has the form

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
```

Note that this data statement consists of four data items, three numeric and one string. The data items are separated by commas. You may include as many data items in a single **DATA** statement as the line allows. Moreover, you may include any number of **DATA** statements in a program and they may be placed anywhere in the program, although a common placement is at the end of the program (just before the **END** statement). Note that we enclosed the string constant "WINGSPAN" in quotation marks. Actually this is not necessary. A string constant in a **DATA** statement does not need quotes, as long as it does not contain a comma, a colon, or start with a blank.

The **DATA** statements may be used to assign values to variables and, in particular, to variables in arrays. Here's how to do this. In conjunction with the **DATA** statements, you use one or more **READ** statements. For example, suppose that the above **DATA** statement appeared in a program. Further, suppose that you wish to assign these values:

```
A = 3.457, B = 2.588, C = 11234, Z$ = "WINGSPAN"
```

This can be accomplished using the **READ** statement:

```
100 READ A,B,C,Z$
```

Here is how the **READ** statement works. On encountering a **READ** statement, the computer will look for a **DATA** statement. It will then assign values to the variables in the **READ** statement by taking the values, in order, from the **DATA** statement. If there is insufficient data in the first **DATA** statement, the computer will continue to assign values using the data in the next **DATA** statement. If necessary, the computer will proceed to the third **DATA** statement, and so forth.

TEST YOUR UNDERSTANDING 1 (answer on page 149)
Assign the following values:
A(1)=5.1, A(2)=4.7, A(3)=5.8, A(4)=3.2, A(5)=7.9, A(6)=6.9.

The computer maintains an internal pointer which points to the next **DATA** item to be used. If the computer encounters a second **READ** statement, it will start reading where it left off. For example, suppose that instead of the above **READ** statement, we use the two read statements:

```
100 READ A,B
200 READ C,Z$
```

Upon encountering the first statement, the computer will look for the location of the pointer. Initially, it will point to the first item in the first **DATA** statement. The computer will assign the values A=3.457 and B=2.588. Moreover, the position of the pointer will be advanced to the third item in the **DATA** statement. Upon encountering the next **READ** statement, the computer will assign values beginning with the one designated by the pointer, namely C=11234 and Z$="WINGSPAN".

TEST YOUR UNDERSTANDING 2 (answer on page 149)
What values are assigned to A and B$ by the following program?

```
10 DATA 10,30,"ENGINE","TACH"
20 READ A,B
30 READ C$,B$
40 END
```

The following example illustrates the use of **DATA** statements in assigning values to an array.

Example 1. Suppose that the monthly electricity costs of a certain family are as follows:

Jan.	$89.74	Feb.	$95.84	March	$79.42
Apr.	78.93	May	72.11	June	115.94
July	158.92	Aug.	164.38	Sep.	105.98
Oct.	90.44	Nov.	89.15	Dec.	93.97

Write a program calculating the average monthly cost of electricity.
Solution. Let us unceremoniously dump all of the numbers shown above into **DATA** statements at the end of the program. Arbitrarily, let's start the **DATA** statements at line 1000, with **END** at 2000. This allows us plenty of room. To calculate the average, we must add up the numbers and divide by 12. To do this, let us first create an array A(J), J=1, 2, ..., 12 and set A(J) equal to the cost of electricity in the Jth month. We do this via a loop and the **READ** statement. Note that we have used a numerical array A(J), since we wish to perform arithmetic on the entries of the array. (Add them up. Divide by 12.) Since we wish to perform arithmetic, we must remove the dollar signs and commas from

the numbers before we put them into the arrays. The arithmetic portion of the program consists of using a loop to add all the A(J)'s and then dividing by 12. Finally, we PRINT the answer. Here is the program.

```
10 DIM A(12)
20 FOR J=1 TO 12
30   READ A(J)
40 NEXT J
50 C=0
60 FOR J=1 TO 12
70   C=C+A(J):        'C ACCUMULATES THE SUM OF THE A(J)
80 NEXT J
90 C=C/12 :           'DIVIDE SUM BY 12
100 PRINT "THE AVERAGE MONTHLY COST OF ELECTRICITY IS",C
1000 DATA 89.74, 95.84, 79.42, 78.93, 72.11, 115.94
1010 DATA 158.92, 164.38, 105.98, 90.44, 89.15, 93.97
2000 END
```

The following program could be helpful in preparing the payroll of a small business.

Example 2. A small business has five employees. Here are their names and hourly wages.

Name	Hourly Wage
Joe Polanski	7.75
Susan Greer	8.50
Allan Cole	8.50
Betsy Palm	6.00
Herman Axler	6.00

Write a program which accepts as input hours worked for the current week, and calculates the current gross pay and the amount of Social Security tax to be withheld from their pay. (Assume that the Social Security tax amounts to 6.7 percent of gross pay.)

Solution. Let us keep the hourly wage rates and names in two arrays, called A(J) and B$(J), respectively, where J = 1, 2, 3, 4, and 5. Note that we can't use a single two-dimensional array for this data since the names are string data, and the hourly wage rates are numerical. (Recall that BASIC does not let us mix the two kinds of data in an array.) The first part of the program will be to assign the values to the variables in the two arrays. Next, the program will, one by one, print out the names of the employees and ask for the number of hours worked during the current week. This data will be stored in the array C(J), J = 1, 2, 3, 4, and 5. The program then will compute the gross wages as A(J)*C(J) (that is, <wage rate> times <number of hours worked>). This piece of data will be stored in the array D(J), J=1, 2, 3, 4, and 5. Next, the program will compute the amount of Social Security tax to be withheld as .0670*D(J). This piece of data will be stored in the array E(J), J=1, 2, 3, 4, 5. Finally, all the computed data will be printed on the screen. Here is the program:

```
10 DIM A(5),B$(5),C(5),D(5),E(5)
20 FOR J=1 TO 5
30 READ B$(J),A(J)
40 NEXT J
50 FOR J=1 TO 5
60   PRINT "TYPE CURRENT HOURS OF", B$(J)
70   INPUT C(J)
80   D(J)=A(J)*C(J)
90   E(J)=.0670*D(J)
100 NEXT J
110 PRINT "EMPLOYEE","GROSS WAGES","SOC.SEC.TAX"
120 FOR J=1 TO 5
130 PRINT B$(J),D(J),E(J)
140 NEXT J
200 DATA JOE POLANSKI, 7.75, SUSAN GREER, 8.50
210 DATA ALLAN COLE, 8.50, BETSY PALM, 6.00
220 DATA HERMAN AXLER, 6.00
1000 END
```

In certain applications, you may wish to read the same **DATA** statements more than once. To do this you must reset the pointer using the **RESTORE** statement. For example, consider the following program.

```
10 DATA 2.3, 5.7, 4.5, 7.3
20 READ A,B
30 RESTORE
40 READ C,D
50 END
```

Line 20 sets A equal to 2.3 and B equal to 5.7. The **RESTORE** statement of line 30 moves the pointer back to the first item of data, 2.3. The **READ** statement of line 40 then sets C equal to 2.3 and D equal to 5.7. Note that without the **RESTORE** in line 30, the **READ** statement in line 40 would set C equal to 4.5 and D equal to 7.3.

There are two common errors in using **READ** and **DATA** statements. First, you might try to **READ** more data than is present in the **DATA** statements. For example, consider the following program.

```
10 DATA 1,2,3,4
20 FOR J=1 TO 5
30   READ A(J)
40 NEXT J
50 END
```

This program attempts to read five pieces of data, but the **DATA** statement only has four. In this case, you will receive an error message:

```
Out of data in 30
```

A second common error is attempting to assign a string value to a numeric variable or vice versa. Such an attempt will lead to a **Type mismatch** error.

Exercises (answers on page 358)

Each of the following programs assigns values to the variables of an array. Determine which values are assigned.

1.
```
10 DIM A(10)
20 FOR J=1 TO 10
30    READ A(J)
40 NEXT J
50 DATA 2,4,6,8,10,12,14,16,18,20
100 END
```

2.
```
10 DIM A(3),B(3)
20 FOR J=0 TO 3
30    READ A(J), B(J)
40 NEXT J
50 DATA 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9
60 END
```

3.
```
10 DIM A(3),B$(3)
20 FOR J=0 TO 3
30    READ A(J)
40 NEXT J
50 FOR J=0 TO 3
60    READ B$(J)
70 NEXT J
80 DATA 1,2,3,4,A,B,C,D
90 END
```

4.
```
10 DIM A(3), B(3)
20 READ A(0),B(0)
30 READ A(1),B(1)
40 RESTORE
50 READ A(2),B(2)
60 READ A(3),B(3)
70 DATA 1,2,3,4,5,6,7,8
80 END
```

5.
```
10 DIM A(3,4)
20 FOR I=1 TO 3
30     FOR J=1 TO 4
40         READ A(I,J)
50     NEXT J
60 NEXT I
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

6.
```
10 DIM A(3,4)
20 FOR J=1 TO 4
30     FOR I=1 TO 3
40          READ A(I,J)
```

```
50      NEXT I
60 NEXT J
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

Each of the following programs contains an error. Find it.

7.
```
10 DIM A(5)
20 FOR J=1 TO 5
30     READ A(J)
40 NEXT J
50 DATA 1,2,3,4
60 END
```

8.
```
10 DIM A(5)
20 FOR J=1 TO 5
30     READ A(J)
40 NEXT J
50 DATA 1,A,2,B
60 END
```

9. Here is a table of Federal Income Tax Withholding of weekly wages for an individual claiming one exemption. Assume that each of the employees, in the business discussed in the text, claims a single exemption. Modify the program of Example 2 so that it correctly computes Federal Withholding and the net amount of wages. (That is, the total after Federal Withholding and Social Security are deducted.)

Wages at Least	But Less Than	Tax Withheld
200	210	29.10
210	220	31.20
220	230	33.80
230	240	36.40
240	250	39.00
250	260	41.60
260	270	44.20
270	280	46.80
280	290	49.40
290	300	52.10
300	310	55.10
310	320	58.10
320	330	61.10
330	340	64.10
340	350	67.10

10. Here is a set of 24 hourly temperature reports as compiled by the National Weather Service. Write a program to compute the average temperature for the last 24 hours. Let your program respond to a query concerning the temperature at a particular hour. (For example, what was the temperature at 2:00 pm?)

	AM	PM
12:00	10	38
1:00	10	39

2:00	9	40
3:00	9	40
4:00	8	42
5:00	11	38
6:00	15	33
7:00	18	27
8:00	20	22
9:00	25	18
10:00	31	15
11:00	35	12

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
10 DATA 5.1,4.7,5.8,3.2,7.9,6.9
20 FOR J=1 TO 6
30    READ A(J)
40 NEXT J
50 END
```

2:
```
A = 10, B$ = "TACH"
```

6.3 Formatting Your Output

In this section, we will discuss the various ways in which you can format output on the screen and on the printer. PCjr BASIC is quite flexible in the form in which you can cast output. You have control over the size of the letters on the screen, placement of output on the line, degree of accuracy to which calculations are displayed, and so forth. Let us begin by reviewing what we have already learned about printing.

Semicolons in PRINT Statements

The screen width may be set for 40- or 80-character lines, using the WIDTH statement. This gives 40 or 80 print positions in each line. These are divided into print zones of 14 characters each.* To start printing at the beginning of the next print zone, insert a comma between the items to be printed.

In many applications, it is necessary to print more columns than there are print zones. Or, output may look better if the columns are less than a full print zone wide. To avoid any space between consecutive print items, separate them in the PRINT statement by a semicolon. Consider the following instruction:

```
10 PRINT "PERSO";"NAL COMPUTER"
```

It will result in the output:

* For an 80-column width, the last print zone has only 10 characters. For a 40-column width, the last print zone has only 12 characters.

```
PERSONAL COMPUTER
```

The semicolon suppresses any space between the display of PERSO and NAL COMPUTER.

In displaying numbers, remember that all positive numbers begin with a blank space, which is in place of the understood plus (+) sign. Negative numbers, however, have a displayed minus (-) sign and do not begin with a blank space. For example, the statement:

```
20 PRINT "THE VALUE OF A IS";2.35
```

will result in the display

```
THE VALUE OF A IS 2.35
```

The space between the S and the 2 comes from the blank which is considered part of the number 2.35. On the other hand, the statement

```
30 PRINT "THE VALUE OF A IS";-2.35
```

will result in the display:

```
THE VALUE OF A IS-2.35
```

To obtain a space between the S and the -, we must include a space in the string constant.

```
30 PRINT "THE VALUE OF A IS ";-2.35
```

TEST YOUR UNDERSTANDING 1 (answer on page 158)
Write a program which allows you to input two numbers. The program then should display them as an addition problem in the form 5 + 7 = 12.

At the completion of a PRINT statement, BASIC will automatically supply an ENTER so that the cursor moves to the beginning of the next line. You may suppress this ENTER by ending the PRINT statement with a semicolon. For example, the statements

```
40 PRINT "THE VALUE+ OF A IS";
50 PRINT 2.35
```

will result in the display:

```
THE VALUE OF A IS 2.35
```

TEST YOUR UNDERSTANDING 2 (answer on page 158)
Describe the output from the following program:

```
10 A=5:B=3:C=8
20 PRINT "THE VALUE OF A IS",A
30 PRINT "THE VALUE OF B";
40 PRINT "IS";B
50 PRINT "THE VALUE OF C IS";-C
```

Our discussion above was oriented to the display of data on the screen. However, you also may use semicolons in LPRINT statements to control spacing of output on the printer.

Horizontal Tabbing

You may begin a print item in any print position. To do this, use the **TAB** command. The print positions are numbered from 1 to 255, going from left to right. (Note that a line may be up to 255 characters long. On the screen, an oversized line will wrap around to the next line. However, the line will print correctly on a printer having a wide enough print line.) The statement TAB(7) means to move to column 7. **TAB** always is used in conjunction with a **PRINT** statement. For example, the print statement

```
50 PRINT TAB(7) A
```

will print the value of the variable A, beginning in print position 7. It is possible to use more than one **TAB** per **PRINT** statement. For example, the statement

```
100 PRINT TAB(5) A; TAB(15) B
```

will print the value of A beginning in print position 5, and the value of B beginning in print position 15. Note the semicolon between the two **TAB** instructions.

TEST YOUR UNDERSTANDING 3 (answer on page 158)
Write an instruction printing the value of A in column 25 and the value of B seven columns further to the right.

In some applications, you may wish to add a certain number of spaces between output items (as opposed to TABbing where the next item appears in a specified column). This may be accomplished using the SPC (=space) function, which works very much like TAB. For example, to print the values of A and B with 5 blank spaces between them, we may use the statement:

```
110 PRINT A; SPC(5) B
```

Example 1. Write a program to print the following table of numbers.

12.5 14.8
11.8 1.8
4.53 .357

Solution. The columns begin, respectively, in columns 1 and 7. So we read the numbers into an array A(I,J) and print the elements of the array using PRINT TAB statements.

```
10 DIM A(3,2)
20 FOR I=1 TO 3
30   FOR J=1 TO 2
40       READ A(I,J)
50   NEXT J
60 NEXT I
70 FOR I=1 TO 3
80   PRINT A(I,1); TAB(7) A(I,2)
90 NEXT I
100 END
200 DATA 12.5,14.8,11.8,1.8,4.53,.357
```

Formatting Numbers

PCjr BASIC has rather extensive provisions for formatting numerical output. Here are some of the things you may specify with regard to printing a number:

Number of digits of accuracy
Alignment of columns (ones column, tens column, hundreds column, and so forth)
Display and positioning of the initial dollar sign
Display of commas in large numbers (as in 1,000,000)
Display and positioning of + and - signs.

All of these formatting options may be requested with the **PRINT USING** statement. Roughly speaking, you tell the computer what you wish your number to look like by specifying a "prototype." For example, suppose you wish to print the value of the variable A with four digits to the left of the decimal point and two digits to the right. This could be done via the instruction:

```
10 PRINT USING "####.##"; A
```

Here, each # stands for a digit and the period stands for the decimal point. If, for example, A was equal to 5432.381, this instruction would round the value of A to the specified two decimal places and would print the value of A as:

5432.38

On the other hand, if the value of A was 932.547, then the computer would print the value as:

```
932.55
```

In this case, the value is printed with a leading blank space, since the format specified four digits to the left of the decimal point. This sort of printing is especially useful in aligning columns of figures like this:

```
  367.1
 1567.2
29573.3
    2.4
```

The above list of numbers could be printed using the following program:

```
10 DATA 367.1, 1567.2, 29573.3, 2.4
20 FOR J=1 TO 4
30   READ A(J)
40   PRINT USING "#####.#";A(J)
50 NEXT J
60 END
```

TEST YOUR UNDERSTANDING 4 (answer on page 158)
Write an instruction which prints the number 456.75387 rounded to two decimal places.

You may use a single **PRINT USING** statement to print several numbers on the same line. For example, the statement:

```
10 PRINT USING "##.##"; A,B,C
```

will print the values of A, B, and C on the same line, all in the format ##.##. Only one space will be allowed between each of the numbers. Additional spaces may be added by using extra #'s. If you wish to print numbers on one line in two different formats, then you must use two different **PRINT USING** statements, with the first ending in a semicolon (;) to indicate a continuation on the same line.

If you try to display a number larger than the prototype, the number will be displayed preceded by a percent (%) symbol. For example, consider the statement:

```
10 PRINT USING "###"; A
```

If the value of A is 5000, then the display will look like:

```
%5000
```

TEST YOUR UNDERSTANDING 5 (answer on page 158)
Write a program to calculate and display the numbers 2J, J = 1, 2, 3, ..., 15. The columns of the numbers should be properly aligned on the right.

You may have the computer insert a dollar sign on a displayed number. The following two statements illustrate the procedure:

```
10 PRINT USING "$####.##"; A
20 PRINT USING "$####.##";A
```

Suppose that the value of A is 34.78. The results of lines 10 and 20 then will be displayed:

```
$  34.78
  $34.78
```

Note the difference between the displays produced by lines 10 and 20. The single $ produces a dollar sign in the fifth position to the left of the decimal point. This is just to the left of the four digits specified in the prototype ####.##. However, the $ in line 20 indicates a "floating dollar sign." The dollar sign is printed in the first position to the left of the number without leaving any space.

Example 1. Here is a list of checks written by a family during the month of March.

$15.32, $387.00, $57.98, $3.47, $15.88

Print the list of checks on the screen with the columns properly aligned and the total displayed below the list of check amounts, in the form of an addition problem.

Solution. We first read the check amounts into an array A(J), J = 1, 2, 3, 4, 5. While we read the amounts, we accumulate the total in the variable B. We use a second loop to print the display in the desired format.

```
10 DATA   15.32, 387.00, 57.98, 3.47, 15.88
20 FOR J=1 TO 5
30   READ A(J)
40   B=B+A(J)
50   PRINT USING "$###.##"; A(J)
60 NEXT J
70 PRINT "______"; B
80 PRINT USING "$###.##"; B
90 END
```

Here is what the output will look like:

```
$ 15.32
$387.00
$ 57.98
$  3.47
$ 15.88
$479.65
```

Note that line 70 is used to print the line under the column of figures.

The **PRINT USING** statement has several other variations. To print commas in large numbers, insert a comma anywhere to the left of the decimal point. For example, consider the statement

```
10 PRINT USING ###,###; A
```

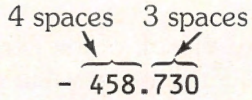
If the value of A is 123456, it will be displayed as:

```
123,456
```

The **PRINT USING** statement also may be used to position plus and minus signs in connection with displayed numbers. A plus sign at the beginning or the end of a prototype will cause the appropriate sign to be printed in the position indicated. For example, consider the statement:

```
10 PRINT USING "+####.###"; A
```

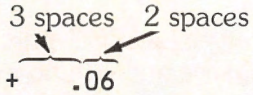
Suppose that the value of A is -458.73. It will be displayed as:



Similarly, consider the statement:

```
10 PRINT USING "+###.##"; A
```

Suppose that A has the value .05873. Then A will be displayed as:



Important Note: In the above discussion, we have only mentioned output on the screen. However, all of the features mentioned may be used on a printer via the **LPRINT USING** instruction. Note, however, that the wider line of the printer allows you to display more data than the screen. In particular, there are more 14-character print fields (just how many depends on which printer you own), and you may **TAB** to a higher numbered column than on the screen.

Recall that BASIC uses two different representations for numbers—the usual decimal representation and scientific (or exponential) notation. You may use to format numbers into scientific notation. For example, to display a number in scientific notation with two digits to the left of the decimal point and two to the right, you would use the format string

"##.##^^^^"

In this format, the number 100 would be displayed as

```
10.00E+01 .
```

Other Variants of PRINT USING

There are several more things you can do with the **PRINT USING** statement. They are especially useful to accountants and others concerned with preparing financial documents.

If you precede the prototype with **, this will cause all unused digit positions in a number to be filled with asterisks. For example, consider the statement

```
10 PRINT USING "**#####.#";A
```

If A has the value 34.86, the value will be displayed as

```
****34.9
```

Note that four asterisks are displayed since six digits to the left of the decimal point are specified in the prototype. The asterisks count, but the value of A uses only two. The remaining four are filled with asterisks.

You may combine the action of ** and $. You should experiment with this combination. It is especially useful for printing dollar amounts of the form:

```
$*******387.98
```

Such a format is especially useful in printing amounts on checks to prevent modification.

By using a minus sign immediately after a prototype, you will print the appropriate number with a trailing minus sign if it is negative and with no sign if it is positive. For example, the statement:

```
10 PRINT USING "####.##-"; A
```

with A equal to -57.88 will result in the display:

```
57.88-
```

On the other hand, if A is equal to 57.88, the display will be:

```
57.88
```

This format for numbers is often used in preparing accounting reports.

Exercises (answer on page 359)

Write programs which generate the following displays. The lines of dots are not meant to be displayed. They are furnished for you to judge spacing.

```
1. THE VALUE OF X IS 5.378   2. THE VALUE OF X IS     5.378
   ..............................................................
3. DATE    QTY    @    COST    DISCOUNT    NET COST
   ..............................................
```

```
4.     6.753                          5.  $  12.82
      15.111                              $ 117.58
     111.850                              $   5.87
       6.702                              $    .99
     -------                              $    .99
                Calculate                 -------
                Sum                                Calculate
                                                   Sum
```

```
6.                                                 Date 3/18/81
     Pay to the Order of      Wildcatters, Inc.
     The sum of                          *********$89,385.00
```

```
7.     5,787                          8.  $385.41
         387                              - $17.85
     127,486                              -------
      38,531                                       Calculate
     -------                                       Difference
                Calculate
                Sum
```

9. Write a program which prints a number rounded to the nearest integer. For example, if the input is 11.7, the output is 12. If the input is 158.2, the output is 158. Your program should accept the number to be rounded via an INPUT statement.
10. Write a program which allows your computer to function as a cash register. Let the program accept purchase amounts with INPUT statements. Let the user tell the program when the list of INPUTs is complete. The program should then print out the purchase amounts, with dollar signs and columns aligned, compute the total purchase, add 5 percent sales tax, compute the total amount due, ask for the amount paid, and compute the change due.
11. Prepare the display of Exercise 4 using 40 characters per line.

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
10 INPUT A,B
20 PRINT A;" +";B;" =";A+B
30 END
```
2:
```
THE VALUE OF A IS    5
THE VALUE OF B IS 3
THE VALUE OF C IS-8
```
3:
```
10 PRINT TAB(25) A;TAB(32) B
```
4:
```
10 PRINT USING "###.##"; 456.7587
```
5:
```
10 FOR J=1 TO 15
20 PRINT USING "#####"; 2^J
30 NEXT J
40 END
```

6.4 Gambling With Your Computer

One of the most interesting features of your computer is its ability to generate events whose outcomes are "random." For example, you may instruct the computer to "throw a pair of dice" and produce a random pair of integers between 1 and 6. You may instruct the computer to "pick a card at random from a deck of 52 cards." You also may program the computer to choose a two-digit number "at random," and so forth. The source of all such random choices is the BASIC function **RND**. To explain how this function works, let us consider the following program:

```
10 FOR X=1 TO 500
20   PRINT RND
30 NEXT X
40 END
```

This program consists of a loop which prints 500 numbers, each called **RND**. Each of these numbers lies between 0.000000 (inclusive) and 1.000000 (exclusive). Each time the program refers to **RND** (as in line 20 here), the computer makes a "random" choice from among the numbers in the indicated range. This is the number that is printed.

To obtain a better idea of what we are talking about, you should generate some random numbers using a program like the one above. Unless you have a printer, 500 numbers will be too many for you to look at in one viewing. You should print four random numbers on one line (one per print zone) and limit yourself to 25 displayed lines at one time. Here is a partial printout of such a program.

```
.245121       .305003       .311866       .515163
.984546       .901159       .727313       6.83401E-03
.896609       .660212       .554489       .818675
.583931       .448163       .86774        .0331043
.137119       .226544       .215274       .876763
```

What makes these numbers "random" is that the procedure the computer uses to select them is "unbiased," with all numbers having an equal likelihood of selection. Moreover, if you generate a large collection of random numbers, then numbers between 0 and .1 will comprise approximately 10 percent of those chosen, those between .5 and 1.0 will comprise 50 percent of those chosen, and so forth. In some sense, the random number generator provides a uniform sample of the numbers between 0 and 1.

TEST YOUR UNDERSTANDING 1 (answer on page 166)
Assume that RND is used to generate 1000 numbers. Approximately how many of these numbers would you expect to lie between .6 and .9?

The random number generator is controlled by a so-called "seed" number, which controls the sequence of numbers generated. Once a particular seed number has been chosen, the sequence of random numbers is fixed. This would make computer games of chance rather uninteresting, since they always would generate the same sequence of play. This may be prevented by changing the seed number using the **RANDOMIZE** command. A command of the form:

```
10 RANDOMIZE
```

will cause the computer to print out the display:

```
Random Number Seed (-32768 to 32767)?
```

You then respond with a number in the indicated interval. Suppose, for example, you choose 129. The computer then will reseed the random number generator with the seed 129 and will generate the sequence of random numbers corresponding to this seed. Another method of choosing a seed number is with a command of this form:

```
20 RANDOMIZE 129
```

This command sets the seed number to 129 without asking you. In Chapter 13, we will show you how to use the computer's internal clock to provide a seed number. This is a method of generating a seed over which no one has any control.

The function **RND** generates random numbers lying between 0 and 1. However, in many applications, we will require randomly chosen **integers** lying in a certain range. For example, suppose that we wish to generate random integers chosen from among 1, 2, 3, 4, 5, 6. Let us multiply **RND** by 6, to obtain 6*RND. This is a random number between 0.00000 and 5.99999. Next, let us add 1 to this number. Then 6*RND+1 is a random number between 1.00000 and 6.99999. To obtain integers from among 1, 2, 3, 4, 5, 6, we must "chop off" the decimal portion of the number 6*RND+1. To do this, we use the **INT** function. If X is any number, then INT(X) is the largest integer less than or equal to X. For example:

```
INT(5.23)=5, INT(7.99)=7, INT(100.001)=100.
```

Be careful in using INT with negative X. The definition we gave is correct, but unless you think things through, it is easy to make an error. For example:

```
INT(-7.4)=-8
```

since the largest integer less than or equal to -7.4 is equal to -8. (Draw -7.4 and -8 on a number line to see the point!) Let us get back to our random numbers. To chop off the decimal portion of 6*RND+1, we compute INT(6*RND+1). This last expression is a random number from among 1, 2, 3, 4, 5, 6. Similarly, the expression:

```
INT(100*RND+1)
```

may be used to generate random numbers from among the integers 1, 2, 3,..., 100.

TEST YOUR UNDERSTANDING 2 (answer on page 166)
Generate random integers from 0 to 1. (This is the computer analog of flipping a coin: 0=heads, 1=tails.) Run this program to generate 50 coin tosses. How many heads and how many tails occur?

Example 1. Write a program which turns the computer into a pair of dice. Your program should report the number rolled on each as well as the total.
Solution. We will hold the value of die #1 in the variable X and the value of die #2 in variable Y. The program will compute values for X and Y, print out the values and the total X+Y.

```
5   RANDOMIZE
10  CLS
20  X=INT(6*RND + 1)
30  Y=INT(6*RND + 1)
40  PRINT "LADIES AND GENTLEMEN, BETS PLEASE!"
50  INPUT "ARE ALL BETS DOWN(Y/N)"; A$
60  IF A$ = "Y" THEN 100 ELSE 40
100 PRINT "THE ROLL IS",X,Y
110 PRINT "THE WINNING TOTAL IS " ; X+Y
120 INPUT "PLAY AGAIN(Y/N)"; B$
130 IF B$="Y" THEN 10
200 PRINT "THE CASINO IS CLOSING. SORRY!"
210 END
```

Note the use of computer-generated conversation on the screen. Note also, how the program uses lines 120-130 to allow the player to control how many times the game will be played. Finally, note the use of the command **RAN-**

DOMIZE in line 5. This will generate a question to allow you to choose a seed number.

TEST YOUR UNDERSTANDING 3 (answer on page 166)
Write a program which flips a "biased coin." Let it report "heads" one-third of the time and tails two-thirds of the time.

You may enhance the realism of a gambling program by letting the computer keep track of bets as in the following example.

Example 2. Write a program which turns the computer into a roulette wheel. Let the computer keep track of bets and winnings for up to five players. For simplicity, assume that the only bets are on single numbers. (In the next example, we will let you remove this restriction!)

Solution. A roulette wheel has 38 positions: 1-36, 0, and 00. In our program, we will represent these as the numbers 1-38, with 37 corresponding to 0 and 38 corresponding to 00. A spin of the wheel will consist of choosing a random integer between 1 and 38. The program will start by asking the number of players. For a typical spin of the wheel, the program will ask for bets by each player. A bet will consist of a number (1-38) and an amount bet. The wheel will then spin. The program will determine the winners and losers. A payoff for a win is 32 times the amount bet. Each player has an account, stored in an array A(J), J = 1, 2, 3, 4, 5. At the end of each spin, the accounts are adjusted and displayed. Just as in Example 1, the program asks if another play is desired. Here is the program.

```
5    RANDOMIZE
10   INPUT "NUMBER OF PLAYERS";N
20   DIM A(5),B(5),C(5):        'At Most 5 Players
30 FOR J=1 TO N :            'Initial Purchase of Chips
40   PRINT "PLAYER "; J
50   INPUT "HOW MANY CHIPS"; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN! PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N :            'Place Bets
120   PRINT "PLAYER "; J
130   INPUT "NUMBER, AMOUNT"; B(J),C(J):'INPUT BET
140 NEXT J
200 X=INT(38*RND + 1): 'Spin the wheel
220 PRINT "THE WINNER IS NUMBER"; X
300 'Compute winnings and losses
310 FOR J=1 TO N
320  IF X=B(J) THEN 400
330   A(J)=A(J)-C(J):          'Player J loses
340   PRINT "PLAYER ";J;"LOSES"
350   GO TO 420
400   A(J)=A(J)+32*C(J):       'Player J wins
410   PRINT "PLAYER ";J;"WINS "; 32*C(J); "DOLLARS"
```

```
420 NEXT J
430 PRINT "PLAYER BANKROLLS": 'Display game status
440 PRINT
450 PRINT "PLAYER", "CHIPS"
460 FOR J=1 TO N
470 PRINT J,A(J)
480 NEXT J
500 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
510 CLS
520 IF R$ = "Y" THEN 100:      'Repeat game
530 PRINT "THE CASINO IS CLOSED. SORRY!"
600 END
```

You should try a few spins of the wheel. The program is fun as well as instructive. Note that the program allows you to bet more chips than you have. We will leave it to the exercises to add in a test that there are enough chips to cover the bet. You could also build lines of credit into the game! In the next example, we will illustrate how the roulette program may be extended to incorporate the bets EVEN and ODD.

Before we proceed to the next example, however, let's discuss one further defect of the program in Example 2. Note that line 5 contains a RANDOMIZE statement. The program will then ask for a random number seed. The person who selects the random number seed has control over the random number sequence and hence over the game. This is most unsatisfactory. However, there is a simple way around this difficulty.

DOS on the PCjr has an internal clock which is set each time you sign on the computer. (The clock is not available if you are not using DOS.) This clock keeps track of time in hours, minutes and seconds. The value of the clock is accessed via the function TIME$. We will discuss use of the clock in detail in Chapter 13. For the moment, however, let's borrow a fact from that discussion. The current reading of the seconds portion of the clock is equal to:

```
VAL(RIGHT$(TIME$,2))
```

Let's use this number as our random number seed. (It is unlikely that anyone can control the precise second at which the game begins.)

Example 3. Modify the roulette program of Example 2 so that it allows bets on EVEN and ODD. A one-dollar bet on either of these pays one dollar in winnings.
Solution. Our program will now allow three different bets: on a number and on EVEN or ODD. Let us design subroutines, corresponding to each of these bets, which determine whether player J wins or loses. For each subroutine, let X be the number (1-38) which results from spinning the wheel. In the preceding program, a bet by player J was described by two numbers: B(J) equals the number bet and C(J) equals the amount bet. Now let us add a third number to describe a bet. Let D(J) equal 1 if J bets on a number, 2 if J bets on EVEN, and 3 if J bets on odd. In case D(J) is 2 or 3, we will again let C(J) equal the amount bet, but B(J) will be ignored. The subroutine for determining the winners of

bets on numbers can be obtained by making small modifications to the corresponding portion of our previous program, as follows:

```
1000 'Bet=NUMBER
1010     IF B(J)=X THEN 1050
1020     PRINT "PLAYER ";J; " LOSES"
1030     A(J)=A(J)-C(J)
1040     GOTO 1070
1050     PRINT "PLAYER ";J; " WINS"; 32*C(J); "DOLLARS"
1060     A(J)=A(J) + 32*C(J)
1070 RETURN
```

Here is the subroutine corresponding to the bet EVEN.

```
2000 'Bet=EVEN
2010     K=0
2020     IF X=2*K THEN 2070 ELSE 2030
2030     K=K+1: IF K>=20 THEN 2040 ELSE 2020
2040     PRINT "PLAYER ";J;" LOSES"
2050     A(J)=A(J)-C(J)
2060     GOTO 2090
2070     PRINT "PLAYER " ;J;" WINS ";C(J);" DOLLARS"
2080     A(J)=A(J)+C(J)
2090 RETURN
```

Finally, here is the subroutine corresponding to the bet ODD.

```
3000 'Bet=ODD
3010      K=0
3020      IF X= 2*K+1 THEN 3070
3030      K=K+1:IF K>=19 THEN 3040 ELSE 3020
3040      PRINT "PLAYER ";J;" LOSES"
3050      A(J)=A(J)-C(J)
3060      GOTO 3090
3070      PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3080      A(J)=A(J)+C(J)
3090 RETURN
```

Now we are ready to assemble the subroutines together with the main portion of the program, which is almost the same as before. The only essential alteration is that we must now determine, for each player, which bet was placed.

```
10 CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 INPUT "NUMBER OF PLAYERS";N
40 DIM A(5),B(5),C(5)
50 FOR J=1 TO N
60   PRINT "PLAYER ";J
70   INPUT "HOW MANY CHIPS";A(J)
```

```
80 NEXT J
90 PRINT "LADIES AND GENTLEMEN! PLACE YOUR BETS PLEASE!"
100 FOR J=1 TO N:          'Place bets
110   PRINT "PLAYER" ;J
120   PRINT "BET TYPE:1=NUMBER BET, 2=EVEN, 3=ODD"
130   INPUT "BET TYPE (1,2, OR 3)";D(J)
140   IF D(J)=1 THEN 170
150   INPUT "AMOUNT";C(J)
160   GOTO 180
170   INPUT "NUMBER, AMOUNT BET";B(J),C(J)
180 NEXT J
190 X=INT(38*RND+1):    'Spin Wheel
200 CLS
210 PRINT "THE WINNER IS NUMBER";X
220 FOR J=1 TO N:          'Determine winnings and losses
230   ON D(J) GOSUB 1000,2000,3000
240 NEXT J
250 PRINT "PLAYER BANKROLLS"
260 PRINT "PLAYER", "CHIPS"
270 FOR J=1 TO N
280   PRINT J,A(J)
290 NEXT J
300 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
310 CLS
320 IF R$="Y" OR R$="y" THEN 90
330 PRINT "THE CASINO IS CLOSED. SORRY!"
340 END
1000 'Bet=NUMBER
1010      IF B(J)=X THEN 1050 ELSE 1020
1020      PRINT "PLAYER ";J; " LOSES"
1030      A(J)=A(J)-C(J)
1040      GOTO 1070
1050      PRINT "PLAYER ";J; " WINS"; 32*C(J); "DOLLARS"
1060      A(J)=A(J)+32*C(J)
1070 RETURN
2000 'Bet=EVEN
2010      K=0
2020      IF X=2*K THEN 2070 ELSE 2030
2030      K=K+1: IF K>=20 THEN 2040 ELSE 2020
2040      PRINT "PLAYER ";J;" LOSES"
2050      A(J)=A(J)-C(J)
2060      GOTO 2090
2070      PRINT "PLAYER " ;J;" WINS ";C(J);" DOLLARS"
2080      A(J)=A(J)+C(J)
2090 RETURN
3000 'Bet=ODD
3010      K=0
3020      IF X= 2*K+1 THEN 3070 ELSE 3030
3030      K=K+1:IF K>=19 THEN 3040 ELSE 3020
3040      PRINT "PLAYER ";J;" LOSES"
3050      A(J)=A(J)-C(J)
3060      GOTO 3090
```

```
3070       PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3080       A(J)=A(J)+C(J)
3090 RETURN
4000 END
```

Note how the subroutines help to organize our programming. Each subroutine is easy to write and each is a small task and you will have less to think about than when considering the entire program. It is advisable to break a long program into a number of subroutines. Not only is it easier to write in terms of subroutines, but it is much easier to check the program and to locate errors since subroutines may be individually tested.

You may treat the output of the random number generator as you would any other number. In particular, you may perform arithmetic operations on the random numbers generated. For example, 5*RND multiplies the output of the random number generator by 5, and RND+2 adds 2 to the output of the random number generator. Such arithmetic operations are useful in producing random numbers from intervals other than 0 to 1. For example, to generate random numbers between 2 and 3, we may use RND+2.

Example 4. Write a program which generates 10 random numbers lying in the interval from 5 to 8.

Solution. Let us build up the desired function in two steps. We start from the function RND, which generates numbers from 0 to 1. First, we adjust for the length of the desired interval. From 5 to 8 is 3 units, so we multiply RND by 3. The function 3*RND generates numbers from 0 to 3. Now we adjust for the starting point of the desired interval, namely 5. By adding 5 to 3*RND, we obtain numbers lying between 0+5 and 3+5, that is, between 5 and 8. Thus, 3*RND+5 generates random numbers between 5 and 8. Here is the program required.

```
10 FOR J=1 TO 10
20   PRINT 3*RND+5
30 NEXT J
40 END
```

Example 5. Write a function to generate random integers from among 5, 6, 7, 8, ..., 12.

Solution. There are 8 consecutive integers possible. Let us start with the function 8*RND, which generates random numbers between 0 and 8. Since we wish our random number to begin with 5, let us add 5 to get 8*RND+5. This produces random numbers between 5.00000 and 12.9999. We now use the INT function to chop off the decimal part. This yields the desired function:

```
INT(8*RND+5)
```

Exercises (answers on page 360)

Write BASIC functions which generate random numbers of the following sorts.

1. Numbers from 0 to 100.

2. Numbers from 100 to 101.
3. Integers from 1 to 50.
4. Integers from 4 to 80.
5. Even integers from 2 to 50.
6. Numbers from 50 to 100.
7. Integers divisible by 3 from 3 to 27.
8. Integers from among 4, 7, 10, 13, 16, 19, and 22.
9. Modify the dice program so that it keeps track of payoffs and bankrolls, much like the roulette program in Example 2. Here are the payoffs on a bet of one dollar for the various bets:

outcome	payoff
2	35
3	17
4	11
5	8
6	6.20
7	5
8	6.20
9	8
10	11
11	17
12	35

10. Modify the roulette program of Example 2 to check that a player has enough chips to cover the bet.
11. Modify the roulette program of Example 2 to allow for a $100 line of credit for each player.
12. Construct a program which tests one-digit arithmetic facts with the problems randomly chosen by the computer.
13. Make up a list of ten names. Write a program which will pick four of the names at random. (This is a way of impartially assigning a nasty task!)

ANSWERS TO TEST YOUR UNDERSTANDING

1: 30 percent

2:
```
10 FOR J=1 TO 50
20   PRINT INT(2*RND+1)
30 NEXT J
40 END
```

3:
```
10 LET X=INT(3*RND + 1)
20 IF X=1 THEN PRINT "HEADS" ELSE PRINT "TAILS"
30 END
```

7

EASING PROGRAMMING FRUSTRATIONS

As you have probably discovered by now, programming can be a tricky and frustrating business. You must first figure out the instructions to give the computer. Next, you must type the instructions into RAM. Finally, you must run the program. Usually after the first run, you must figure out why the program won't work. This process can be tedious and frustrating, especially in dealing with long or complex programs. We should emphasize that programming frustrations often result from the limitations and inflexibility of the computer to understand exactly what you are saying. In talking with another person, you usually sift out irrelevant information, correct minor errors, and still maintain the flow of communication. With a computer, however, you must clear up all of the imprecisions before the conversation can even begin.

Fortunately, your computer has many features designed to ease the programming burdens and help you track down errors and correct them. We will describe these features in this chapter. We will also present some more tips which should help you develop programs quicker and with fewer errors.

7.1 Flowcharting

In the last three chapters, our programs were fairly simple. By the end of Chapter 6, however, we saw them becoming more involved. And there are many programs which are even much more lengthy and complex. You might be wondering how it is possible to plan and execute such programs. The key idea is to reduce large programs to a sequence of smaller programs which can be written and tested separately.

The old saying, "A picture is worth a thousand words," is true for computer programming. In designing a program, especially a long one, it is helpful to draw a picture depicting the instructions of the program and their interrelationships. Such a picture is called a **flowchart**.

A flowchart is a series of boxes connected by arrows. Within each box is a series of one of more computer instructions. The arrows indicate the logical flow of the instructions. For example, the flowchart in Figure 7-1 shows a program for calculating the sum $1+2+3+\ldots+100$.

The arrows indicate the sequence of operations. Note the notation "J = 1,2,...,100" between the second and third boxes. This notation indicates a

loop on the variable J. This means that the operation in box 3 is to be repeated 100 times—for J = 1, 2, ..., 100. Note how easy it is to proceed from the above flowchart to the corresponding BASIC program:

```
10 S=0                          (box 2)
20 FOR J=1 TO 100
30   S=S+J                      (box 3)
40 NEXT J
50 PRINT S                      (box 4)
60 END                          (box 5)
```

There are many flowcharting rules. Different shapes of boxes represent certain programming operations. We will adopt a very simple rule—that all boxes are rectangular, except for decision boxes. Decision boxes are diamond-shaped. The flowchart in Figure 7-2 shows a program which decides whether a credit limit has been exceeded.

Note that the diamond-shaped block contains the decision "Is D > Limit L?". Corresponding to the two possible answers to the question, there are two

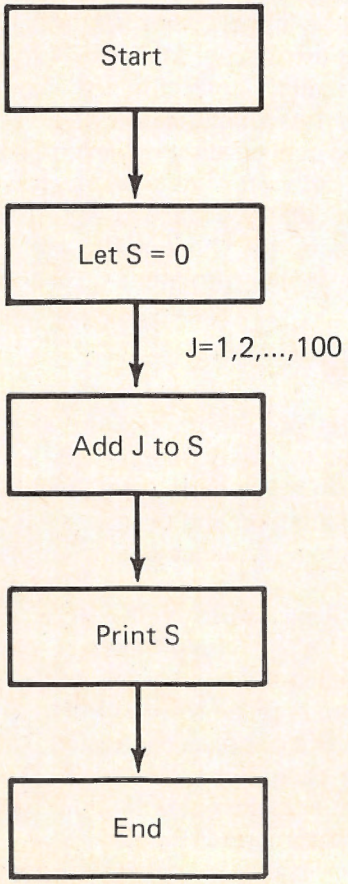
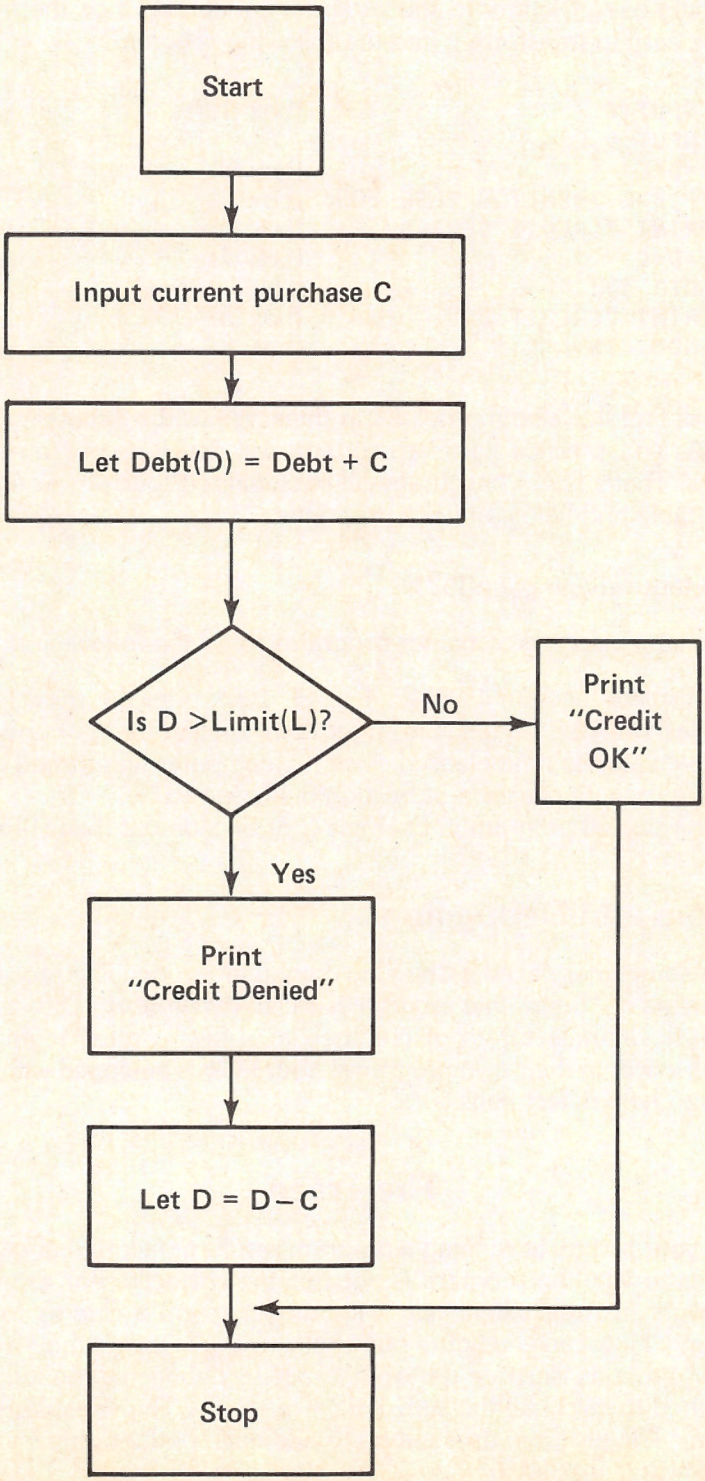


Figure 7-1.

Figure 7-2.


Start
Input current purchase C
Let Debt(D) = Debt + C
Is D >Limit(L)?
No
Print "Credit OK"
Yes
Print "Credit Denied"
Let D = D−C
Stop

arrows leading from the decision box. Note also how we used the various boxes to help assign letters to the program variables. Once the flowchart is written, it is easy to transform it into the following program:

```
10  INPUT C                          (box 1,2)
20  INPUT D,L
30  D=D+C                            (box 3)
40  IF D>L THEN 100 ELSE 200         (box 4)
100 PRINT "CREDIT DENIED"            (box 6)
110 D=D-C                            (box 7)
120 GOTO 300                         ("No" arrow)
200 PRINT "CREDIT OK"                (box 5)
300 END                              (box 8)
```

You will find flowcharting helpful in thinking out the necessary steps of a program. As you practice flowcharting, you will develop your own style and conventions. That's fine. I encourage all personalized touches, as long as they are comfortable and help you write programs.

Exercises (answers on page 362)

Draw flowcharts planning computer programs to do the following.

1. Calculate the sum $1^2 + 2^2 + \ldots + 100^2$, print the result, and determine whether the result is larger than, smaller than, or equal to 487^3.
2. Calculate the time elapsed since the computer was turned on.
3. The roulette program of Section 6.4 (page 161).
4. The payroll program in Example 2 of Section 6.2 (page 146).

7.2 Errors and Debugging

An error is sometimes called a "bug" in computer jargon. The process of finding these errors or "bugs" in a program is called **debugging**. This can often be a ticklish task. Manufacturers of commercial software must regularly repair bugs they discover in their own programs! Your PCjr is equipped with a number of features to help detect bugs.

The Trace

Often your first try at running a program results in failure, but gives you no indication as to why the program is not running correctly. For example, your program might just run indefinitely, without giving you a clue as to what it is actually doing. How can you figure out what's wrong? One method is to use the **trace** feature. Let us illustrate use of the trace by debugging the following program designed to calculate the sum $1+2+\ldots+100$. The variable S is to contain the sum. The program uses a loop to add each of the numbers 1, 2, 3, ..., 100 to S, which is initially 0.

```
10  S=0
20  J=0
30  S=S + J
40  IF J=100 THEN 100 ELSE 200
100 J=J+1
110 GOTO 20
200 PRINT S
300 END
```

This program has two errors in it. (Can you spot them right off?) All you know initially is that the program is not functioning normally. The program runs, but prints out the answer 0, which is nonsense. How can we locate the errors? Let's turn on the trace function by typing **TRON** (TRace ON) and pressing ENTER. The computer will respond by typing **Ok**. Now type **RUN**. The computer will run our program and print out the line numbers of all executed instructions. Here is what our display looks like:

```
TRON
Ok

RUN
[10] [20] [30] [40] [200] 0
[300]
```

The numbers in brackets indicate the line numbers executed. That is, the computer executes, in order, lines 10, 20, 30, 40, 200, and 300. The zero not in brackets is the program output resulting from the execution of line 200. The list of line numbers is not what we were expecting. Our program was designed (or so we thought) to execute line 100 after line 40. No looping is taking place. How did we get to line 200 after line 40? This suggests that we examine line 40: Lo and behold! There is an error. The line numbers 100 and 200 appearing in line 40 have been interchanged (an easy enough mistake to make). Let's correct this error by retyping the line.

```
40 IF J=100 THEN 200 ELSE 100
```

In triumph, we run our program again. Here is the output:

```
[10] [20] [30] [40] [100] [110] [20] [30]

[40] [100] [110] [20] [30] [40] [100] [110]

[20] [30] [40] [100]

Break in 110
```

Actually, the above output goes whizzing by us as the computer races madly on executing the instructions. After about 30 seconds, we sense that something is indeed wrong since it is unlikely that our program could take this long. We stop

execution by means of the Fn-Break key combination. The last line indicates that we interrupted the computer while it was executing line 110. Actually, your screen will be filled with output resembling the above. You will notice that the computer is in a loop. Each time it reaches line 110, the loop goes back to line 20. Why doesn't the loop ever end? In order for the loop to terminate, J must equal 100. Well, can J ever equal 100? Of course not! Every time the computer executes line 20, the value of J is reset to 0. Thus, J is never equal to 100 and line 40 always sends us back to line 20. We clearly don't want to reset J to 0 all the time. After increasing J by 1 (line 100), we wish to add the new J to S. We want to go to 30, not 20. We correct line 110 to read:

```
110 GOTO 30
```

We run our program again. There will be a rush of line numbers on the screen followed by the output 5050, which appears to be correct. Our program is now running properly. We turn off the trace by typing **TROFF** (TRace OFF) and pressing ENTER. Finally, we run our program once more for good measure. The above sequence of operations is summarized in the following display:

```
[40] [200] 5050
[300]
Ok

TROFF
Ok

RUN
5050
Ok
```

In our example above, we displayed all the line numbers executed. For a long program, this may lead to a huge list of line numbers. You may be selective by using TRON and TROFF within your program. Just use them with line numbers, just like any other BASIC instruction. When BASIC encounters a TRON, it begins to display the line numbers executed. When BASIC encounters a TROFF, it stops displaying line numbers. To debug a program, you may temporarily add TRON and TROFF instructions at selected places. As you locate the bugs, remove the corresponding trace instructions.

Error Messages

In the example above, the program actually ran. A more likely occurrence is that there is a program line (or lines) which the computer is unable to understand due to an error or some other sort of problem. In this case, program execution ends too soon. The computer often can help in this instance since it is designed to recognize many of the most common errors. The computer will print an error message indicating the error type and the line number in which it

occurred. The line with the error is automatically displayed, ready for editing. Suppose that the error reads:

```
Syntax Error in 530
530 Y=(X+2(X^2-2)
```

We note that there is an open parenthesis "(" without a corresponding close parenthesis ")". This is enough to trigger an error. We modify line 530 to read

```
530 Y=X+2(X^2-2)
```

We **RUN** the program again and find that there is still a syntax error in line 530! This is the frustrating part since not all errors are easy to spot. However, if you look closely at the expression on the right, you will note that we have omitted the * to indicate the product of 2 and (X^2-2). This is a common mistake, especially for those familiar with the use of algebra. (In algebra, the product is usually indicated without any operation sign.) We correct line 530 again. (You may either retype the line or use the line editor.)

```
530 Y=X+2*(X^2-2)
```

Now there is no longer a syntax error in line 530!

The next section contains a list of the most common error messages. There are a number of errors not included in our list, especially those associated with disk operations. For a complete list of error messages, the reader is referred to the **IBM PCjr BASIC Reference Manual**.

Exercises (answers on page 364)

1. Use the error messages to debug the following program to calculate $(1^2+2^2+\ldots+50^2)(1^3+2^3+\ldots+20^3)$.

```
10 S="0"
20 FOR J=1 TO 50
30   S=S + J(2
40 NEXT K
50 T=0
60 FOR J=1 TO 20
70    T=T +J^3
80 NXT T
90 NEXT T
100 A=ST
110 PRINT THE ANSWER IS, A
120 END
```

2. Use the trace function to debug the following program to determine the smallest integer N for which N^2 is larger than 175263.

```
10 N=0
20 IF N^2 < 175263 THEN 100
30 PRINT "THE FIRST N EQUALS"
100 N=N+1
110 GOTO 10
200 END
```

7.3 Some Common Error Messages

Syntax Error. There is an unclear instruction (misspelled?), mismatched parentheses, incorrect punctuation, illegal character, or illegal variable name in the program.

Undefined line number. The program uses a line number which does not correspond to an instruction. This can easily arise if you delete lines which are mentioned elsewhere. It also can occur when testing a portion of a program which refers to a line not yet written.

Overflow. A number too large for the computer.

Division by zero. Attempting to divide by zero. This may be a hard error to spot. The computer will round to zero any number smaller than the minimum allowed. Use of such a number in subsequent calculations could result in division by zero.

Illegal function call. (For the mathematically-minded.) Attempting to evaluate a function outside of its mathematically defined range. For example, the square root function is defined only for non-negative numbers, the logarithm function only for positive numbers, and the arctangent only for numbers between -1 and 1. Any attempt to evaluate a function at a value outside these respective ranges will result in an illegal function call error.

Missing Operand. Attempting to execute an instruction missing required data.

Subscript Out of Range. Attempting to use an array with one or more subscripts outside the range allowed by the appropriate DIM statement.

String Too Long. Attempting to specify a string containing more than 255 characters.

Out of Memory. Your program will not fit into the computer's memory. This could result from large arrays or too many program steps or a combination of the two.

String Formula Too Complex. Due to the internal processing of your formula, your string formula resulted in a string expression that was too long or com-

plex. This error can be corrected by breaking the string expression into a series of simpler expressions.

Type Mismatch. Attempting to assign a string constant as the value of a numeric variable, or a numeric constant value to a string variable.

Duplicate Definition. Attempting to DIMension an array which has already been dimensioned. Note that once you refer to an array within a program, even if you don't specify the dimensions, the computer will regard it as being dimensioned at 10.

NEXT without FOR. A NEXT statement which does not correspond to a FOR statement.

RETURN without GOSUB. A RETURN statement is encountered while not performing a subroutine.

Out of Data. Attempting to read data which isn't there. This can occur in reading data from DATA statements, cassettes, or diskettes.

Can't Continue. Attempting to give a CONT command after the program has ENDed, or after a line has been modified.

Each error has a corresponding **error number** by means of which you can refer to the error within a program. A complete list of errors and their error numbers is given at the end of the book. Moreover, we will discuss errors further in Chapter 11, where we will learn how to react to errors without ending the program.

7.4 Further Debugging Hints

Debugging is something between a black art and a science. Tracking down program bugs can be a very tricky business and to be good at it, you must be a good detective. In the preceding section, we listed some of the clues which BASIC automatically supplies, namely the error messages. Sometimes, however, these clues are not enough to diagnose a bug. (For example, your program may run without errors. It may just not do what it is supposed to. In this case, no error messages will be triggered.) In such circumstances you must be prepared to supply your own clues. Here are some techniques.

INSERT EXTRA PRINT STATEMENTS

You may temporarily insert extra PRINT statements into your program to print out the values of key variables at various points in the program. This technique allows you to keep track of a variable as your program is executed.

INSERT STOP COMMANDS

It is perfectly possible that your program planning may contain a logical flaw. In this case, it is perfectly possible to write a program which runs without error messages, but which does not perform as you expect it to. You may temporarily insert a STOP command to force a halt after a specified portion of the program.

This debugging technique may be used in several ways.

1. When the program encounters a STOP instruction, it halts execution and prints out the line number at which the program was stopped. If the program does stop, you will know that the instructions just before the STOP were executed. On the other hand, suppose that the program continues on its merry way. This tells you that the program is avoiding the instructions immediately preceding the STOP. If you determine the reason for this behavior, then you likely will correct a bug.

2. When the program is halted, the values of the variables are preserved. You may examine them to determine the behavior of your program. (See below for more information.)

3. You may insert several STOP instructions. After each halt, you may note the behavior of the program (line number, values of key variables, and so forth). You may continue execution by typing CONT and pressing ENTER. Note that if you change a program line during a halt, then you may not continue execution, but must restart the program by typing RUN and pressing ENTER.

EXAMINE VARIABLES IN THE IMMEDIATE MODE

When BASIC stops executing your program, the current values of the program variables are not destroyed. Rather, they are still in memory and may be examined as an indication of program behavior. This is true even if the program is halted by means of a STOP instruction or by hitting Ctrl-Break.

Suppose that a program is halted and that the BASIC prompt Ok is displayed. To determine the current values of the program variables INVOICE and FILENAME$, type

```
PRINT INVOICE, FILENAME$
```

and press ENTER. Note that there is no line number. This instruction is in immediate mode. BASIC will display the current values of the two variables, just as if the PRINT statement were contained in a program:

```
145.83          ACCTPAY.MAR
```

Warning: As soon as you make any alteration in your program (correct a line, add a line), BASIC will reset all the variables. The numeric variables will be reset to zero and the string variables will be set to null. Therefore, if you wish to have an accurate reading of the variable values as they emerge from your program, be sure to request them before making any program changes.

EXECUTE ONLY A PORTION OF YOUR PROGRAM

Sometimes it helps to run only a portion of your program. You may start execution at any line using a variation of the RUN command. For example, to begin execution at line 500, type:

```
RUN 500
```

and press ENTER. Note, however, that the RUN command causes all variables to be reset. If some earlier portion of your program sets some variables, then starting the program in the middle may not give an accurate picture of program operation. To get around this problem, you may set variables in immediate mode and start the program using the GOSUB instruction. For example, suppose that the earlier portion of your program set INVOICE equal to 145.83 and FILENAME$ equal to ACCTPAY.MAR. To accurately run a portion of the program depending on these variable values, you would first type:

```
INVOICE=145.83:FILENAME$="ACCTPAY.MAR"
```

and press ENTER. (These instructions could be entered on separate lines, each followed by ENTER.) To start the program at line 500, you then would type:

```
GOTO 500
```

and press ENTER. Note that it is not sufficient to use the command:

```
RUN 500
```

The RUN command automatically resets the variables.

8

YOUR COMPUTER AS A FILE CABINET

In this chapter we will discuss techniques for using your computer to store and retrieve information.

8.1 What Are Files?

A **file** is a collection of information stored on a mass storage device (diskette, cassette, or hard disk). There are two common types of files: program files and data files.

Program Files When a program is stored on diskette, it is stored as a *program file*. You already have created some program files by saving BASIC programs on diskette. In addition to the programs you create, your DOS diskette contains program files that are necessary to run your computer, such as DOS and the BASIC language.

Data Files Computer programs used in business and industry usually refer to files of information that are kept in mass storage. For example, a personnel department would keep a file of data on each employee: name, age, address, social security number, date employed, position, salary, and so forth. A warehouse would maintain an inventory for each product with the following information: product name, supplier, current inventory, units sold in the last reporting period, date of the last shipment, size of the last shipment, and units sold in the last 12 months. These files are called *data files*.

In this chapter, we will discuss the procedures for handling files in general and data files in particular.

Consider the following example. Suppose that a teacher stores grades in a data file. For each student in the class, there are four exam grades. A typical entry in the data file would contain the following data items:

student name, exam grade #1, exam grade #2,
exam grade #3, exam grade #4

In a data file, the data items are organized in sequence. So the beginning of the above data file might look like this:

"John Smith", 98, 87, 93, 76, "Mary Young",
99, 78, 87, 91, "Sally Ronson", 48, 63, 72,
80, ...

The data file consists of a sequence of string constants (the names) and numeric constants (the grades), with the various data items arranged in a particular pattern (name followed by four grades). This particular arrangement is designed so the file may be read and understood. For instance, if we read the data items above, we know in advance that the data items are in groups of five with the first one a name and the next four the corresponding grades.

In this chapter, we will learn to create data files containing information such as the data in the above example. As we shall see, data may be stored in either of two types of data files—sequential and random access. For each type of file, we will learn to perform the following operations:

1. Create a data file.
2. Write data items to a file.
3. Read data items from a file.
4. Alter data items in a file.
5. Search a file for particular data items.

8.2 Sequential Files

A **sequential file** is a data file in which the data items are accessed in order. That is, the data items are written in consecutive order into the file. The data items are read in the order in which they were written. You may add data items only to the end of a sequential file. If you wish to add a data item somewhere in the middle of the file, it is necessary to rewrite the entire file. Similarly, if you wish to read a data item at the end of a sequential file, it is necessary to read all the data items in order and to ignore those that you don't want.

OPENing and CLOSEing Sequential Files

Before you perform any operations on a sequential file, you must first open the file. You should think of the file as being contained in a file cabinet drawer (the diskette). In order to read the file, you must first open the file drawer. This is accomplished using the BASIC instruction **OPEN**. When **OPEN**ing a file, you must specify the file and indicate whether you will be reading from the file or writing into the file. For example, to **OPEN** the file B:PAYROLL for input (for reading the file), we use a statement of the form:

```
10 OPEN "B:PAYROLL" FOR INPUT AS #1
```

The #1 is a reference number we assign to the file when opening it. As long as the file remains open, you refer to it by its reference number rather than the more cumbersome file specification B:PAYROLL. The reference number is quite arbitrary. You may assign any positive integer you wish. Just make sure that you don't assign two files that are to be open simultaneously to the same reference number. (If you try this, BASIC will give you an error message.)

Here is an instruction for opening the file "GAMES" on cassette for input:

```
10 OPEN "CAS1:GAMES" FOR INPUT AS #1
```

Here is an alternate form of the instruction for opening a file for input:

```
10 OPEN "I",#1,"B:PAYROLL"
```

Here the letter "I" stands for "Input."

To **OPEN** the file B:GRADES.AUG for output (that is, to write in the file), we use an instruction of the form:

```
20 OPEN "B:GRADES.AUG" FOR OUTPUT AS #2
```

Here is an alternate way to write the same instruction:

```
20 OPEN "O",#2,"B:GRADES.AUG"
```

The letter "O" stands for "Output."

BASIC initially allows you to work with three open diskette files at a time. Only one cassette file may be open at a time. This number may be increased by giving the appropriate command when you start BASIC. For example, to allow use of as many as 5 files at once, start BASIC with the command:

```
BASIC /F:5
```

The "switch" /F:5 is what tells BASIC to set aside memory for simultaneous manipulation of up to 5 files.

In maintaining any filing system, it is necessary to be neat and organized. The same is true of computer files. A sequential file may be opened for input or for output, but not both simultaneously. As long as the file remains open, it will accept instructions (input or output) of the same sort designated when it was opened. To change operations, it is necessary to first close the file. For example, to close the file B:PAYROLL in line 10 above, we use the instruction

```
40 CLOSE #1
```

After giving this instruction, we may reopen the file for output using an instruction similar to that given in line 20 above. It is possible to close several files at a time. For example, the statement:

```
50 CLOSE #5,#6
```

closes the files with reference numbers 5 and 6. We may close all currently open files with the instruction:

```
50 CLOSE
```

In an OPEN or CLOSE statement, the # is optional. Thus, it is perfectly acceptable to use:

```
50 OPEN 1,2

50 CLOSE 5,6
```

Good programming practice dictates that all files be closed after use. In any case, the BASIC commands NEW, RUN, and SYSTEM automatically close any files that might have been left open by a preceding program.

WRITEing Data Items Into a Sequential File

Suppose that we wish to create a sequential file called INVOICE.001, which contains the following data items:

```
DJ SALES 50357 4 $358.79 4/5/81
```

That is, we would like to write into the file the string constant "DJ SALES" followed by the two numeric constants 50357 and 4, followed by the two string constants "$358.79" and "4/5/81". Here is a program that does exactly that:

```
100 OPEN "B:INVOICE.001" FOR OUTPUT AS #1
110 WRITE#1, "DJ SALES", 50357,4,"$358.79", "4/5/81"
120 CLOSE #1
```

The #1 portion of line 110 refers to the identification number given to the file in the OPEN instruction in line 100, namely 1. In a WRITE# statement, a comma must follow the file number.

Note that the WRITE instruction works very much like a PRINT statement, except that the data items are "printed" in the file instead of on the screen.

While a file is open, you may execute any number of WRITE instructions to insert data. Moreover, you may WRITE data items that are values of variables, as in the statement:

```
200 WRITE #1, A, A$
```

This instruction will write current values of A and A$ into the file.

Example 1. Write a program to create a file whose data items are the numbers 1, 1^2, 2, 2^2, 3, 3^2, ..., 100, 100^2.

Solution. Let's call the file "SQUARES" and store it on the diskette in drive A:.

```
10 OPEN "A:SQUARES" FOR OUTPUT AS #1
20 FOR J=1 TO 100
30   WRITE#1, J,J^2
40 NEXT J
50 CLOSE #1
60 END
```

Example 2. Create a data file consisting of names, addresses, and telephone numbers from your personal telephone directory. Assume that you will type the addresses into the computer and will tell the computer when the last address has been typed.

Solution. We use **INPUT** statements to enter the various data. Let NME$ denote the name of the current person, ADDRESS$ the street address, CITY$ the city, STATE$ the state, ZIPCODE$ the zip code, and TELEPHONE$ the telephone number. For each entry, there is an **INPUT** statement corresponding to each of these variables. The program then writes the data to the diskette. Here is the program:

```
5  OPEN "TELEPHON" FOR OUTPUT AS #1
10 INPUT "NAME"; NME$
20 INPUT "STREET ADDRESS"; ADDRESS$
30 INPUT "CITY"; CITY$
40 INPUT "STATE"; STATE$
50 INPUT "ZIP CODE"; ZIPCODE$
60 INPUT "TELEPHONE"; TELEPHONE$
70 WRITE#1, NME$, ADDRESS$, CITY$, STATE$,
           ZIPCODE$,TELEPHONE$
80 INPUT "ANOTHER ENTRY (Y/N)"; G$
90 IF G$="Y" THEN 10
100 CLOSE #1
110 END
```

There are several noteworthy points about the above program. Note the unusual spelling of NAME (NME). We are forced into this queer spelling since NAME is a BASIC reserved word. You should use the above program to set up a computerized telephone directory of your own. It is very instructive. Moreover, when coupled with the search program given below, it will allow you to look up addresses and phone numbers using your computer.

TEST YOUR UNDERSTANDING 1

Use the above program to enter the following addresses into the file:

John Jones
1 South Main St. Apt. 308
Phila. Pa. 19107
527-1211

Mary Bell
2510 9th St.
Phila. Pa. 19138
937-4896

Reading Data Items

To read items from a data file, it is first necessary to open the file for INPUT (that is, for INPUT from the diskette.) Consider the telephone file in Example 2. We may open it for input, via the instruction

```
300 OPEN "TELEPHON" FOR INPUT AS #2
```

Once the file is open, it may read via the instruction

```
400 INPUT #2, NME$,ADDRESS$,CITY$,STATE$,
              ZIPCODE$,TELEPHONE$
```

This instruction will read six data items from the file (corresponding to one telephone-address entry), assign NME$ the value of the first data item, ADDRESS$ the second, and so forth.

In order to read a file, it is necessary to know the precise format of the data in the file. For example, the form of the above **INPUT** statement was dictated by the fact that each telephone-address entry was entered into the file as six consecutive string constants. The file INPUT statement works like any other INPUT statement: Faced with a list of variables separated by commas, it assigns values to the indicated variables in the order in which the data items are presented. However, if you attempt to assign a string constant to a numeric variable or vice versa, BASIC will report an error.

As long as a file is open for INPUT, you may continue to INPUT from it, using as many INPUT statements as you like. These may, in turn, be intermingled with statements that have nothing to do with the file you are reading. Each INPUT statement begins reading the file where the preceding INPUT statement left off.

Here's how to determine if you have read all data items in a file. BASIC maintains the functions EOF(1), EOF(2),..., one for each open file. These functions may be used like logical variables. That is, they assume the possible values TRUE or FALSE. You may test for the end of the file using an IF...THEN statement. For example, consider the statement:

```
100 IF EOF(1) THEN 2000 ELSE 10
```

This statement will cause BASIC to determine if you are currently at the end of file #1. If so the program will go to line 2000. Otherwise, the program will go to line 10. Note that you are not at the end of the file until **after** you read the last data item.

If you attempt to read past the end of a file, BASIC will report an **Input Past End** error. Therefore, before reading a file it is a good idea to determine whether you are currently at the end of the file.

Example 3. A data file, called NUMBERS, consists of numerical entries. Write a program to determine the number of entries in the file.

Solution. Let us keep a count of the current number we are reading in the variable COUNT. Our procedure will be to read a number, increase the count, then test for the end of the file.

```
10 COUNT=0
20 OPEN "NUMBERS" FOR INPUT AS #1
30 IF EOF(1) THEN 100
40 INPUT #1,A
50 COUNT=COUNT+1
60 GOTO 30
100 PRINT "THE NUMBER OF NUMBERS IN THE FILE IS",COUNT
110 CLOSE
120 END
```

Example 4. Write a program that searches for a particular entry of the telephone directory file created in Example 2.
Solution. We will **INPUT** the name corresponding to the desired entry. The program then will read the file entries until a match of names occurs. Here is the program:

```
5  OPEN "TELEPHON" FOR INPUT AS #1
10 INPUT "NAME TO SEARCH FOR"; Z$
20 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,ZIPCODE$,
   TELEPHONE$
30 IF NME$ = Z$ THEN 100
40 IF EOF(1)  THEN 200
50 GOTO 20
100 CLS
110 PRINT NME$
120 PRINT ADDRESS$
130 PRINT CITY$,STATE$,ZIPCODE$
140 PRINT TELEPHONE$
150 GOTO 1000
200 CLS
210 PRINT "THE NAME IS NOT ON FILE"
1000 CLOSE 1
1010 END
```

TEST YOUR UNDERSTANDING 2
Use the above program to locate Mary Bell's number in the telephone file created in TEST YOUR UNDERSTANDING 1.

Example 5. (Mailing List Application) Suppose that you have created your computerized telephone directory, using the program in Example 2. Assume that the completed file is called TELEPHON and is on the diskette in drive A:. Write a program that reads the file and prints out the names and addresses onto mailing labels.

Solution. Let's assume that your mailing labels are of the "peel-off" variety, which can be printed continuously on your printer. Further, let's assume that the labels are six printer lines high, so that each label has room for five lines of print with one line space between labels. (These are actual dimensions of labels you can buy.) We will print the name on line 1, the address on line 2, the city, state, and zip codes all on line 3, with the city and state separated by a comma.

```
10 OPEN "TELEPHON" FOR INPUT AS #1
20 IF EOF(1) THEN 1000
30 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,
             ZIPCODE$,TELEPHONE$
40 LPRINT NME$
50 LPRINT ADDRESS$
60 LPRINT CITY$;
70 LPRINT ",";              :'PRINT COMMA
80 LPRINT TAB(10) STATE$;
90 LPRINT TAB(20) ZIPCODE$
100 LPRINT:LPRINT:LPRINT    :'NEXT LABEL
110 GOTO 20
1000 CLOSE 1
1010 END
```

Adding to a Data File

Here is an important fact about writing data files: Writing a file destroys any previous contents of the file. (In contrast, you may read a file any number of times without destroying its contents.) Consider the file "TELEPHON" created in Example 2 above. Suppose we write a program that opens the file for output and writes what we suppose are additional entries in our telephone directory. After this write operation, the file "TELEPHON" will contain *only* the added entries. All of the original entries will have been lost! How, then, may we add items to a file that already exists? Easy. PCjr BASIC has a special instruction to do this. Rather than **OPEN** the file for **OUTPUT**, we **OPEN** the file for **APPEND**, using the instruction:

```
500 OPEN "TELEPHON" FOR APPEND AS #1
```

The computer will locate the current end of the file. Any additional entries to the file will be written beginning at that point. However, the previous entries in the file will be unchanged.

Example 6. Write a program that adds entries to the file TELEPHON. The additions should be typed via INPUT statements. The program may assume that the file is on the diskette in drive A:.

Solution. To add items to the file, we first **OPEN** the file for **APPEND**. We then ask for the new entry via an **INPUT** statement and write the new entry into TELEPHON. Here is the program:

```
10 OPEN "TELEPHON" FOR APPEND AS #1
210 PRINT "TYPE ENTRY:NAME,STREET ADDRESS,CITY, STATE,"
220 PRINT "ZIP CODE, TELEPHONE NO."
230 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,
              ZIPCODE$,TELEPHONE$
240 WRITE#1, NME$,ADDRESS$,CITY$,STATE$,
              ZIPCODE$,TELEPHONE$
250 INPUT "ANOTHER ENTRY  (Y/N)": Z$
260 IF Z$ <> "Y" THEN 500
300 CLS
310 GOTO 210
500 CLOSE 1
510 END
```

TEST YOUR UNDERSTANDING 3

Use the above program to add your name, address, and telephone number to the telephone directory created in TEST YOUR UNDERSTANDING 1.

Exercises (answers on page 364)

1. Write a program creating a diskette data file containing the numbers 5.7, −11.4, 123, 485, and 49.
2. Write a program that reads the data file created in Exercise 1 and displays the data items on the screen.
3. Write a program that adds to the data file of Exercise 1 the data items 5, 78, 4.79, and −1.27.
4. Write a program that reads the expanded file of Exercise 3 and displays all the data items on the screen.
5. Write a program that records the contents of checkbook stubs in a data file. The data items of the file should be as follows:
   check #, date, payee, amount, explanation

Use this program to create a data file corresponding to your previous month's checks.

6. Write a program that reads the data file of Exercise 5 and totals the amounts of all the checks listed in the file.
7. Write a program that keeps track of inventory in a retail store. The inventory should be described by a data file whose entries contain the following information:
   item, current price, units in stock

The program should allow for three different operations: Display the data file entry corresponding to a given item, record receipt of a shipment of a given item, and record the sale of a certain number of units of a given item.

8. Write a program that creates a recipe file to contain your favorite recipes.
9. (For Teachers) Write a program that maintains a student file containing your class roll, attendance, and grades.

10. Write a program maintaining a file of your credit card numbers and the party to notify in case of loss or theft.

8.3 More About Sequential Files

When you WRITE a data item to a sequential file, BASIC automatically includes certain "punctuation" that allows the data to be read:

1. Strings are surrounded by quotation marks.
2. Data items are separated by commas.
3. The last data item in the WRITE# statement is followed by <ENTER>. Here <ENTER> means the ENTER key. To the computer, <ENTER> is a character, just like "A" or ";". It tells the computer to end the current line and move the cursor to the start of the following line. Actually, <ENTER> generates **two** characters: One is a **carriage return** that returns the cursor to the left side of the line. The other is a **line feed** that advances the cursor to the next line. In what follows we will continue to use <ENTER> to stand for the combination of these two characters.
4. Positive numbers are inserted in the file without a leading blank.

For example, suppose that A$ = "JOHN", B$ = "SMITH", C = 1234, and D = −14. Consider the following WRITE# statement:

```
10 WRITE#1, A$,B$,C,D
```

Here is how this statement would WRITE the data into file #1:

```
"JOHN","SMITH",1234,-14<ENTER>
```

When the above data is read by an INPUT# statement, the quotation marks, commas, and ENTER enable BASIC to separate the various data items from one another. For this reason, the punctuation marks are called **delimiters**. In using the WRITE# statement, you need not worry about delimiters. However, in other sequential file statements, you are not so lucky.

Consider, for instance, the PRINT# statement. This statement may be used to PRINT data to a file exactly as if the data were being printed on the screen. All of the usual features of PRINT, such as TAB, SPC, and semicolons, are active. However, the PRINT# statement does not include any delimiters. Consider the above variables A$, B$, C, and D. The statement:

```
20 PRINT#1, A$;B$;C;D
```

will write the following image to file #1:

```
JOHNSMITH 1234-14<ENTER>
```

Note that:

1. The space before the positive number 1234 is included in the file,

2. There are no separations between the data items.
3. There are no quotation marks around the strings.

In order to correctly read the individual data items, you must supply delimiters in your PRINT# statement. Here's how. First, put commas as strings in PRINT#:

```
20 PRINT#1, A$;",";B$;",";C;",";D
```

Here's how the image in the file will now look:

```
JOHN,SMITH, 1234,-14<ENTER>
```

The individual data items now may be read.

This is not quite the end of the story, however! Notice that the strings still do not have quotation marks around them. In this example, no harm will be done. To understand why, let's discuss the operation of the INPUT# statement.

INPUT recognizes the following list of delimiters: commas, ENTER, and form feed (we'll learn about this character later). When faced with a stream of data in a file, here is what INPUT# does:

1. INPUT# scans the characters and peels off characters until it finds a delimiter. This indicates the end of the current data item. (The delimiter is not included as part of the data item.)
2. If a numeric data item has been requested, INPUT# checks that the data item is a number (no illegal characters such as A, $, or ;). If illegal characters are detected, a **Type Mismatch** error occurs.
3. If a string data item has been requested, INPUT# checks to see whether the data item is surrounded by quotation marks. If so, it removes them.

Understanding the above sequence can prevent embarrassing errors. One such error can occur if you wish to include a comma within a data item. For example, suppose that A$="SMITH,JOHN", B$="CARPENTER". The PRINT# statement:

```
30 PRINT#1, A$;",";B$
```

will write the following image to the file:

```
SMITH,JOHN,CARPENTER<ENTER>
```

A subsequent INPUT# statement:

```
40 INPUT#1, A$,B$
```

will result in A$="SMITH" and B$="JOHN". To get around this problem, you must explicitly include quotation marks around strings that include a comma. A string that consists of a quotation mark is just CHR$(34). (34 is the

ASCII code for a quotation mark.) So to include the quotation marks around the string A$="SMITH,JOHN", you may use the statement:

```
50 PRINT#1, CHR$(34);A$;CHR$(34);",";B$
```

The file image will now be:

```
"SMITH,JOHN",CARPENTER<ENTER>
```

Quotation marks must enclose strings containing commas, semicolons, beginning or ending blanks, or ENTERs.

As you can see, the PRINT# statement is much less convenient than WRITE#. In most cases, it is much simpler to use WRITE#. However, PRINT# has its advantages. With a PRINT#, you may include the USING option to format your data. For example, to write the value of the variable A to the file in the format ##.#, we could use the statement:

```
60 PRINT#1, USING "##.#";A
```

The INPUT# statement reads a single data item at a time. However, in some applications you may wish to read an entire line from a file. That is, you wish to read data until you encounter an ENTER. This may be done with the LINE INPUT# statement. For example, suppose that the following data is contained in file #1:

```
SMITH,JOHN,CARPENTER<ENTER>
```

The statement:

```
70 LINE INPUT #1, A$
```

will set A$="SMITH,JOHN,CARPENTER". Note the following curious twist, however. If you saved your string data with quotation marks around it, those quotation marks would be included as part of the string read by LINE INPUT#. If you plan to read data lines via a LINE INPUT# statement, it is usually wise to save the data using PRINT# so that no extraneous quotation marks are generated.

File Buffers

You may have noticed that the drive light does not always turn on when you are writing to a file. For example, try this experiment: OPEN a data file and write a single numerical data item to the file, but don't CLOSE the file. The disk drive does nothing. However, if you run this program a second time, the drive light will go on. This may seem strange. However, it has to do with the way BASIC writes (and reads) diskette files.

Diskette drives are very slow when compared with the speed at which BASIC executes non-diskette operations. In order to speed up diskette opera-

tions, BASIC writes to the diskette using **file buffers**. A file buffer (or "buffer" for short) is an area of RAM where BASIC temporarily stores data to be written to a file. There is one buffer corresponding to each open file. BASIC reserves the space for a buffer as part of the OPEN operation. When you use any file writing operation, BASIC writes the corresponding information into the file's buffer. When the buffer is full, BASIC writes the data to the file.

The CLOSE operation forces all buffers (full or not) to be written to their corresponding files. When you don't close a file (as in our above experiment), the buffer may be sitting with some data that has not yet been written to diskette. In this case, a RUN or END command also will cause the buffers to be written to diskette. Also, as soon as you modify the program in RAM, the buffers will be written to diskette. In our experiment, it was the RUN statement that caused the drive lights to go on, to write the final results of the previous run.

Exercises (answers on page 366)

Suppose that A$ = "MY", B$ = "DOG", C$ = "SAM", D = 1234. What is the format of the data written to file #1 by the following statements?

1. `WRITE#1, A$,B$,C$,D`
2. `PRINT#1, A$,B$,C$,D`
3. `PRINT#1, A$;",";B$;",";C$;",";D`
4. `PRINT#1, CHR$(34);A$;" ";B$;", ";CHR$(34);",";C$;", ";D`

Consider the file as written by Exercise 1. What will be displayed by the following statements?

5. `INPUT#1, E$:PRINT E$`
6. `LINE INPUT #1, E$:PRINT E$`

Consider the files as written by Exercises 2–4. What will be displayed by the following statements?

7. `INPUT#1, E$:PRINT E$`
8. Consider the file as written by Exercise 4. Write a program to display:

```
MY DOG, SAM
1234
```

8.4 Random Access Files

The files considered so far in this chapter are all examples of **sequential files**. That is, the files are all written sequentially, from beginning to end. These files are very easy to create, but are cumbersome in many applications, since they must be read sequentially. In order to read a piece of data from the end of the file, it is necessary to read all data items from the beginning of the file. **Random access files** do not suffer from this difficulty. Using a random access file, it is possible to access the precise piece of data you want. Of course, there is a price to be paid for this convenience. (No free lunches!) You must work a little harder to learn how to use random access files.

A random access file is divided into segments of fixed length called **records**. (See Figure 8-1.) The length of a record is measured in terms of

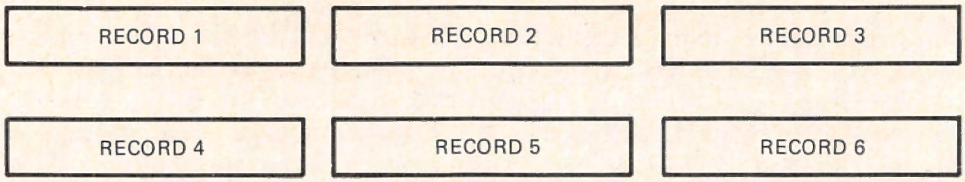


Figure 8-1. A Random Access File.

bytes. For a string constant, each character, including spaces and punctuation marks, counts as a single byte.
For example, the record consisting of the string

```
ACCOUNTING-5
```

is of length 12.

To store a data item in a random access file, all data must be converted into string form. This applies to numeric constants and values of numeric variables. (See below for the special instructions for performing this conversion.) A number (more precisely, a single-precision number) is converted into a string of length 4, no matter how many digits this number has. A record may contain the four data items: ACCOUNTING, 5000, .235, and 7886. These pieces of data are stored in order, with no separations between them. The length of this particular record is 22 bytes (10 for ACCOUNTING and four each for the numerical data items). (See Figure 8-2.)

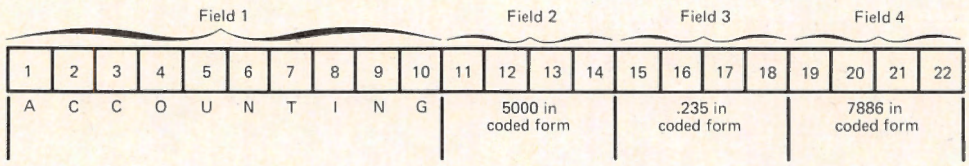


Figure 8-2. A Typical Record.

To write data to a random access file, it is necessary to first open it. To open a file named "DEPTS" as a random access file with a record length of 22, we would use the instruction:

```
10 OPEN "DEPTS" AS #1 LEN=22
```

Next, we must describe the structure of the records of the file. For example, suppose that each record of file #1 is to start with a 10-character string followed by three numbers (converted to string form). Further, suppose that the string represents a department name, the first number the current department income, the second number the department's efficiency rating, and the third number the current department's overhead. We indicate this situation with the instruction:

```
20 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4
   AS EFFICIENCY$, 4 AS OVERHEAD$
```

This instruction identifies the file with the number used when the file was opened. Each section of the record is called a **field**. Each field is identified by a string variable and the number of bytes reserved for that variable.

To write a record to a random access file, it is first necessary to assemble the data corresponding to the various fields. This is done using the LSET and RSET instructions. For example, to set the DEPT$ field to the string "ACCOUNTING", we use the instruction:

```
30 LSET DEPT$="ACCOUNTING"
```

To set the DEPT$ field to the value of the string variable N$, we use:

```
40 LSET DEPT$=N$
```

If N$ contains fewer than 10 characters, the rightmost portion of the field is filled with blanks. This is called **left justification**. If N$ contains more than 10 characters, the field is filled with the leftmost 10 characters.

The instruction RSET works exactly the same as LSET, except that the unused spaces appear on the left side of the field. (The strings are **right justified**.)

To convert numbers to strings for inclusion in random access files, we use the MKS$ (or MKI$ or MKD$) function. For example, to include .753 in the EFFICIENCY$ field, we first replace it by the string MKS$(.753). To include the value of the variable INC in the INCOME$ field, we first replace it by MKS$(INC). After the conversion, we use the LSET (or RSET) commands to insert the string in the field. In the case of the two examples cited, the sequence is carried out by the respective instructions:

```
50 LSET EFFICIENCY$=MKS$(.753)
60 LSET INCOME$=MKS$(INC)
```

Once the fields of a particular record have been set (using LSET or RSET), you may write the record to the file using the PUT instruction. Records are numbered within the file, starting from one. The significant feature of a random access file is that you may record or retrieve information from any particular record. For example, to write the current data into record 38 of file #1, we use the instruction:

```
80 PUT #1, 38
```

TEST YOUR UNDERSTANDING 1 (answer on page 198)
Write a program that creates a file containing the following records:

ACCOUNTING	5000	.235	7886
ENGINEERING	3500	.872	2200
MAINTENANCE	4338	.381	5130
ADVERTISING	10832	.95	12500

To read a random access file, you must first open it using an instruction of the form:

```
90 OPEN "DEPTS" AS #1 LEN=23
```

Note: This is the same as the instruction for opening a random access file for writing. Random access files differ from sequential files in this respect. By opening a random access file you prepare it for both reading and writing. Before closing the file, you may read some records and write others.

The next step in reading a random access file is to define the record structure using a **FIELD** statement, such as

```
100 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4 AS
    EFFICIENCY$, 4 AS OVERHEAD$
```

This is the same instruction we used for writing to the file. Until the **FIELD** instruction is overridden by another, it applies to all reading and writing for file #1.

To perform the actual reading operation, we use the GET statement. For example, to read record 4 of the file, we use the statement

```
110 GET #1, 4
```

The variables DEPT$, INCOME$, EFFICIENCY$, and OVERHEAD$ are now set equal to the appropriate values specified in record 4 of file #1. We could, for example, print the value of DEPT$ using the statement

```
120 PRINT DEPT$
```

If we wish to use the value of EFFICIENCY$ (in a numerical calculation or in a PRINT statement, for instance), it is necessary first to convert it back into numerical form. This is accomplished using the CVS function. The statement:

```
130 PRINT CVS(EFFICIENCY$)
```

will print out the current value of EFFICIENCY$; the statement

```
140 LET N=100*CVS(EFFICIENCY$)
```

sets the value of N equal to 100 times the numerical value of EFFICIENCY$.

It is important to note that field variables such as DEPT$ and EFFICIENCY$ contain the values assigned in the most recent GET statement. In order to manipulate data from more than one GET statement, it is essential to assign the values from one GET statement to some other variables before issuing the next GET statement.

TEST YOUR UNDERSTANDING 2 (answer on page 198)
Consider the random access file of TEST YOUR UNDERSTANDING 1. Write a program to read record 3 of that file and print the corresponding four pieces of data on the screen.

Random access files use no delimiters to separate data items within the file. Rather, the data items are sandwiched together, using the number of characters specified for each field. In order to peel those data items back apart, you must divide the file into records of the correct length and each record into fields of the proper numbers of bytes.

In our discussion above, we have used the instructions MKS$ and CVS to convert numerical data to string format and back to numerical format. These functions apply to single-precision numbers. In addition to single-precision numbers, there are double-precision numbers (up to 17 digits) and integers (whole numbers between -32768 and +32767). To convert a double-precision number to a string, we use the function MKD$; to convert back to numerical form, CVD. To convert an integer to a string, use the function MKI$; to convert back to numerical form, use CVI.

In either numerical form or string form, an integer is represented by 2 bytes, a single-precision number by 4 bytes and a double-precision number by 8 bytes. In particular, this means that MKI$ produces a 2-byte string, MKS$ a 4-byte string, and MKD$ an 8-byte string.

BASIC provides several functions that help you keep track of random access files. The LOF (=Length of File) function gives the actual number of bytes in the file. For example, suppose that file #2 contains 140 bytes. Then:

```
LOF(#2) is equal to 140
```

The LOF function may be used to determine the number of records currently in the file, according to the formula

```
<number of records> = LOF(<file number>)/<record length>
```

Note that random access files cannot have any "holes." That is, if you write record 150, BASIC sets aside space for records 1 through 149, even if you write nothing in these records.

The LOC (Location) function gives the number of the last record read or written to the file. For example, if the last record written or read to file #1 was record 58, then LOC(#1) is equal to 58.

Here is an example that illustrates most of the procedures for using random access files.

Example 1. Write a program to create an address/telephone directory using a random access file. The program should allow for updating the directory and for directory search corresponding to a given name.

Solution. The program first opens the random access file TELEPHON, used to store the various directory entries. Note that the record length is set equal to 128. This allows us to use LOF to calculate the number of records in the file using either BASIC 1.1 or 2.00. The program then displays a menu allowing you to choose from among the various options: Add an entry to the directory, Search the directory, Exit from the program. After an option is completed, the program redisplays the menu to allow you to make another choice. The code corresponding to the three options begins at program lines 1000, 2000, and 3000, respectively. Here is the program:

```
1000 'Telephone File
1010 'Open File For Random Access
1020     OPEN "TELEPHON" AS #1 LEN=128
1030     FIELD#1, 20 AS NME$, 20 AS ADDRESS$, 20 AS
         CITY$, 20 AS STATE$, 5 AS ZIPCODE$, 20 AS
         TELEPHONE$, 23 AS BLANK$
1040     LSET BLANK$=""
1050 'Option Menu
1060     PRINT "OPTIONS"
1070     PRINT "1. MAKE ENTRY IN DIRECTORY"
1080     PRINT "2. SEARCH DIRECTORY"
1090     PRINT "3. EXIT PROGRAM"
1100     INPUT "CHOOSE OPTION (1/2/3)";OPT
1110     ON OPT GOSUB 2000,3010,4010
1120     GOTO 1060
2000 'Add to file
2010     CLS
2020     INPUT "NAME";N$
2030     LSET NME$=N$
2040     INPUT "ADDRESS";N$
2050     LSET ADDRESS$=N$
2060     INPUT "CITY";N$
2070     LSET CITY$=N$
2080     INPUT "STATE";N$
2090     LSET STATE$=N$
2100     INPUT "ZIPCODE";N$
2110     LSET ZIPCODE$=N$
2120     INPUT "TELEPHONE NUMBER";N$
2130     LSET TELEPHONE$ = N$
2140     PUT #1
3000 RETURN
3010 'Search for a name
3020     NREC=LOF(1)/128
3030     INPUT "NAME TO SEARCH FOR";N$
3040     R=1
3050     GET #1, R
3060     GOSUB 5000: IF M$=N$ THEN 3100
3070     R=R+1
3080     IF R>NREC THEN PRINT "NAME IS NOT ON FILE":
         GOTO 4000
3090     GOTO 3050
```

```
3100      PRINT NME$
3110      PRINT ADDRESS$
3120      PRINT CITY$
3130      PRINT STATE$
3140      PRINT ZIPCODE$
3150      PRINT TELEPHONE$
4000 RETURN
4010 'Exit from program
4020      CLOSE
4030 END
5000 'Strip trailing blanks
5010      M$=NME$
5020      IF RIGHT$(M$,1) <> CHR$(32) THEN 5050
5030      M$ = LEFT$(M$,LEN(M$)-1)
5040      GOTO 5020
5050 RETURN
```

Note that line 3020 computes the number of records using the LOF function. In searching the file for a given name N$, the records are read one by one and the first field is compared with N$. Note, however, that the first field is always 20 characters long. If the corresponding name has less than 20 characters, the field contains one or more trailing blanks. In comparing the first field with N$, it is necessary first to remove these blanks. This is done in the subroutine beginning in line 5000.

Setting the Random File Buffer Size

When BASIC is started, it sets aside a portion of RAM to aid in reading and writing random access files. This piece of RAM is called a **random file buffer**. Its size places a limit on the record size of your random access files. You may specify any record size you wish (see below). However, if you don't specify the size of the random access buffer, BASIC will allow only 128 bytes.

Suppose, for example, that you wish to use random access files with record lengths as large as 200 bytes. You may arrange for this by starting BASIC as follows:

1. Obtain the DOS prompt A> .
2. Type

```
BASIC /S:200
```

and press ENTER.

BASIC then will display the Ok prompt and you may program as usual.

If you attempt to use a FIELD statement requiring more bytes than are contained in the random access buffer, a **Field Overflow** error will result.

Exercises (answers on page 366)

1. Write a program that writes the file TELEPHON of Section 8.2 (page 186) as a random access file. Leave 20 characters for the NAME

entry, 25 for the address, 10 for the city, 2 for the state, 5 for the ZIP code, and 10 for the telephone number.

2. Here is a record from a personnel file. For ease in reading this record, we have replaced all blanks with @.

```
JONES@@@@@@@JOHN@@@@@@JFILECLERK4@@@04/15/82HOURLY$5.85
```

Write a field statement that will correctly separate the fields of the record.

3. Suppose that a file named "SALES" consists of 20 records, each containing four numbers. Write a program that reads the file and prints the numbers in four columns on the screen.
4. Write a program that allows you to specify a name in the file TELEPHON. The program locates the file entry and prints out an address label corresponding to the name.

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
10 OPEN "DEPTS" AS #1, LEN=23
20 FIELD #1, 11 AS DEPT$, 4 AS INCOME$, 4 AS
   EFFICIENCY$, 4 AS OVERHEAD$
30 FOR J=1 TO 4
40   READ A$,B,C,D
50   LSET DEPT$=A$
60   LSET INCOME$=MKS$(B)
70   LSET EFFICIENCY$=MKS$(C)
80   LSET OVERHEAD$=MKS$(D)
90   PUT #1,J
100 NEXT J
110 DATA "ACCOUNTING",5000,.235,7886
120 DATA "ENGINEERING",3500,.872,2200
130 DATA "MAINTENANCE",4338,.381,5130
140 DATA "ADVERTISING",10832,.95,12500
150 CLOSE #1
160 END
```

2:
```
10 OPEN "DEPTS" AS #1, LEN=23
20 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4 AS
   EFFICIENCY$,4 AS OVERHEAD$
30 GET #1, 3
40 PRINT "DEPARTMENT","INCOME","EFFICIENCY",
   "OVERHEAD"
50 PRINT DEPT$,CVS(INCOME$),CVS(EFFICIENCY$),
   CVS(OVERHEAD$)
   60 CLOSE #1
   70 END
```

8.5 An Application of Random Access Files

In this section, we work out a detailed example illustrating the application of random access files. We will design and build a "list manager" program, which allows you to manipulate a list. A program of this sort is sometimes called a **database management program**.

Our program will manipulate quite general lists. A typical list is structured into a series of entries, with each entry divided into a series of data items. We have allowed each entry of our list to contain as many as five string items and five numerical items. The string items are listed first. A typical list entry has the form:

```
ITEM #1 (STRING)
ITEM #2 (STRING)
ITEM #3 (STRING)
ITEM #4 (STRING)
ITEM #5 (STRING)
ITEM #6 (NUMBER)
ITEM #7 (NUMBER)
ITEM #8 (NUMBER)
ITEM #9 (NUMBER)
ITEM #10 (NUMBER)
```

It is not necessary to use all 10 items. The entries of a particular list might consist of three strings followed by two numbers, for example. The program asks for the structure of the list entries (number of strings and number of numbers). The program then assumes that all entries of the list contain the specified numbers of data items of each type.

The list manager allows you to perform the following activities:

1. Give a name to a list and create a corresponding random access file to contain the list.

2. Give titles to the various items ("NAME", "ADDRESS", "SALARY", "TELEPHONE #"). An item title may be up to 12 characters long.

3. Enter list items. The program displays the various item names and allows you to type in the various items for the list entry. You may repeat the entry operation as many times as you wish, thereby compiling lists of any length.

4. Change list entries. You may change a list entry by re-entering its data items.

5. Display list entries. You may display a single list entry or an entire set of consecutive list entries.

6. Search the list. You may search the list for entries in which a particular item (say ZIPCODE) has a particular value (say 20001). The program will inform you of a match and give the entry number. It then will ask you if you wish to see the corresponding list entry. If so, it will display the entry for you. After you are done examining the entry, you hit ENTER, and the program will continue to search for further matches.

The following program is highly structured (major tasks correspond to subroutines) and the listing is reasonably self-explanatory. However, you should note the following:

1. The titles of the list are stored in record 1.

2. The actual list entries are stored beginning in record 2. The entry number (list entry 5) is always one less than the corresponding record (record 6).

3. There are two menus. The main menu allows you to choose among the following activities:

Specify Titles
Specify List Entry
Search and Display
Exit

The second menu is displayed if you choose the Search and Display option on the main menu. The various options in this second menu are:

Display Single List Entry
Display Consecutive List Entries
Search

4. Entry items are numbered from 1 to 10, with the strings 1 to 5 and the numbers 6 to 10. This numbering holds even if some items are not used. That is, the first numerical item is always 6.

```
10 'List Manager
20 'Main Program
30 GOSUB 4200:            'Obtain file name and open file
40 GOSUB 1010:             'Display Main Menu
50 ON REPLY GOSUB 2000,3000,4100,4140
60 GOTO 40
1000 'Display main menu
1010      CLS
1020      PRINT "THE LIST MANAGER"
1030      PRINT:PRINT
1040      PRINT "PROGRAM ACTIVITIES"
1050      PRINT
1060      PRINT "1. ASSIGN DATA ITEM TITLES"
1070      PRINT "2. SPECIFY LIST ENTRY"
1080      PRINT "3. SEARCH AND DISPLAY LIST"
1090      PRINT "4. EXIT FROM LIST MANAGER"
1100      PRINT
1110      INPUT "DESIRED ACTIVITY(1-4)";REPLY
1120 RETURN
2000 'Assign Data Item Titles
2010    CLS
2020    IF LOF(1)=1 THEN 2040:       'New File ?
2030    GOSUB 4400:                  'Get old titles
2040    FOR ITEMNUMBER=1 TO 10
2050            PRINT "DATA ITEM #";ITEMNUMBER;TAB(20)
                "CURRENT DEF'N ";
2060            PRINT A$(ITEMNUMBER)
```

```
2070            INPUT "NEW DEF'N: ";TITLE$(ITEMNUMBER)
2080            LSET A$(ITEMNUMBER)=TITLE$(ITEMNUMBER)
2090   NEXT ITEMNUMBER
2100   PUT #1,1:                          'Record new titles
2110 RETURN
3000 'Specify list entry
3010   CLS
3020   INPUT "LIST ENTRY NUMBER (0=NEW ENTRY)";
       ENTRYNUMBER
3030   IF ENTRYNUMBER=0 THEN ENTRYNUMBER=LOC(1)+1 ELSE
       ENTRYNUMBER=ENTRYNUMBER+1
3040   GOSUB 4400:                            'Obtain titles
3050   PRINT "LIST ENTRY #";ENTRYNUMBER-1;TAB(20)
       "SPECIFY ENTRY ITEMS"
3060   FOR ITEMNUMBER=1 TO STRINGFIELDS
3070           PRINT "Data Item Title: ";TITLE$
               (ITEMNUMBER)
3080           INPUT "ENTRY (STRING)";ENTRY$
3090           LSET A$(ITEMNUMBER)=ENTRY$
3100   NEXT ITEMNUMBER
3110   FOR ITEMNUMBER=6 TO 5+NUMERICFIELDS
3120           IF TITLE$(ITEMNUMBER)="" THEN 3160
3130           PRINT "Data Item Title: ";TITLE$
               (ITEMNUMBER)
3140           INPUT "ENTRY (NUMBER)";ENTRY
3150           LSET A$(ITEMNUMBER)=MKS$(ENTRY)
3160   NEXT ITEMNUMBER
3170   PUT #1,ENTRYNUMBER
3180 RETURN
4000 'Various Subroutines
4100 'Search and Display List
4110     GOSUB 4700:           'Search and Display Menu
4120     ON REPLY GOSUB 4800,4900,5100
4130 RETURN
4140 'Exit
4150  CLS
4160  CLOSE #1
4170  END
4180 RETURN
4200 'Obtain file name and open file
4210    CLS
4220    CLOSE
4230    PRINT "THE LIST MANAGER"
4240    INPUT "NAME OF FILE";FILENAME$
4250    INPUT "NUMBER OF STRING FIELDS (1-5)";
        STRINGFIELDS
4260    INPUT "NUMBER OF NUMERIC FIELDS (1-5)";
        NUMERICFIELDS


4270    OPEN FILENAME$ AS #1
4280    FIELD #1, 12 AS A$(1), 12 AS A$(2), 12 AS A$(3),
```

```
        12 AS A$(4),12 AS A$(5),12 AS A$(6), 12 AS
        A$(7), 12 AS A$(8), 12 AS A$(9), 12 AS A$(10)


4290    GOSUB 4400:         'Read old titles
4300  RETURN
4400 'Read old titles
4410    GET #1,1
4420    FOR J=1 TO 10
4430          TITLE$(J)=A$(J)
4440    NEXT J
4450  RETURN
4500 'Display entry
4510    CLS
4520    PRINT:                    'Advance to 2nd line
4530    GOSUB 4400:               'Read titles
4540    IF DISPLAYENTRY > LOF(1)/128 THEN 4680:
        'Non-existant record
4550    GET #1, DISPLAYENTRY
4560    FOR ITEMNUMBER=1 TO STRINGFIELDS
4570            ENTRY$(ITEMNUMBER)=A$(ITEMNUMBER)
4580            PRINT TITLE$(ITEMNUMBER);
                TAB(21) ENTRY$(ITEMNUMBER)
4590    NEXT ITEMNUMBER
4600    FOR ITEMNUMBER=6 TO 5+NUMERICFIELDS
4610            IF A$(ITEMNUMBER)="" THEN 4620 ELSE 4640
4620            PRINT TITLE$(ITEMNUMBER)
4630            GOTO 4660
4640            ENTRY(ITEMNUMBER)=CVS(A$(ITEMNUMBER))
4650            PRINT TITLE$(ITEMNUMBER);
                TAB(21) ENTRY(ITEMNUMBER)
4660    NEXT ITEMNUMBER
4670    LOCATE 1,1
4680  RETURN
4700 'Display and Search Menu
4710    CLS
4720    PRINT "DISPLAY AND SEARCH MENU"
4730    PRINT : PRINT
4740    PRINT "1. DISPLAY ENTRY WITH GIVEN NUMBER"
4750    PRINT "2. DISPLAY CONSECUTIVE ENTRIES"
4760    PRINT "3. SEARCH"
4770    PRINT
4780    INPUT "ACTIVITY(1-3)";REPLY
4790  RETURN
4800 'Display entry with given number
4810    CLS
4820    PRINT:                    'Advance to 2nd line
4830    INPUT "Number of entry to display";
        DISPLAYENTRY
4840    DISPLAYENTRY=DISPLAYENTRY+1
4850    IF DISPLAYENTRY > LOF(1)/128 THEN 4890
4860    GOSUB 4500
```

```
4870    INPUT "TO CONTINUE, HIT ENTER KEY";REPLY$
4880    IF REPLY$="" THEN 4790 ELSE 4670
4890  RETURN
4900 'Display consecutive entries
4910    CLS
4920    PRINT:                    'Advance to 2nd line
4930    INPUT "NUMBER OF FIRST ENTRY TO DISPLAY";
        DISPLAYENTRY
4940    DISPLAYENTRY=DISPLAYENTRY+1
4950    IF DISPLAYENTRY > LOF(1)/128 THEN 5020
4960    GOSUB 4500
4970    LOCATE 1,1
4980    INPUT "DISPLAY NEXT ENTRY=0,RETURN TO
        MAIN MENU=1";REPLY
4990    IF REPLY=1 THEN 5020
5000    DISPLAYENTRY=DISPLAYENTRY+1
5010    GOTO 4950
5020  RETURN
5100 'Search
5110    CLS
5120    INPUT "ITEM NUMBER TO SCAN";ITM
5130    PRINT "LOOK FOR ITEM NUMBER";ITM;" EQUAL TO";
5140    IF ITM<6 THEN INPUT MATCHSTRING$
5150    IF ITM>5 THEN INPUT MATCHNUMBER
5160    L=LEN(MATCHSTRING$):
        'L=length of the match string
5170    MATCHSTRING$=MATCHSTRING$+SPACE$(12-L):
        'Add blanks
5180    LNGTH=LOF(1)/128
5190    FOR J=2 TO LNGTH
5200            GET #1, J
5210            IF ITM > 5 THEN A=CVS(A$(ITM))
                ELSE A$=A$(ITM)
5220            IF ITM < 6 AND A$=MATCHSTRING$
                THEN GOSUB 5300
5230            IF ITM > 5 AND A=MATCHNUMBER
                THEN GOSUB 5300
5240    NEXT J
5250  RETURN
5300 'Response to a match
5310    CLS
5320    LOCATE 1,1
5330    PRINT "MATCH IN ENTRY";J-1
5340    INPUT "Do You Wish to Display
        Entry(1=Yes,0=No)";REPLY
5350    IF REPLY=1 THEN 5360 ELSE 5400
5360    DISPLAYENTRY=J
5370    GOSUB 4500
5380    INPUT "TO CONTINUE, HIT ENTER KEY";REPLY$
5390    IF REPLY$="" THEN 5400 ELSE 5380
5400 RETURN
```

Note that the fields of the file records are all 12 characters wide. This is to accommodate the titles in record 1. Since we do not specify a record length, BASIC assumes that the records are 128 characters long. We are using 120 (12 characters per field x 10 fields) of these characters. If you wish, you may redesign this program to accommodate more data items and longer string items and titles. However, if you use more than 128 characters per record, it will be necessary to initialize BASIC to allow for a sufficiently large random access file buffer.

Exercises

1. Type in the list manager program.
2. Use the list manager program to create a Christmas card list.
3. Practice using the search feature to locate particular data items.

8.6 Sorting Techniques

In the preceding sections, we have discussed the mechanisms to create, read, and write data files. In this section, we discuss the organization of data within such files.

If a data file is to be of much use, we must be able to easily access its data. At first this might seem like a simple requirement. After all, we can always search through a data file, examining records until we find the one we want. Unfortunately, this is just not always possible. Until now, we have been working with rather short data files. However, many applications require dealing with data files containing thousands or even tens of thousands of records. When a data file is large, even the great speed of the computer is insufficient to guarantee a speedy search. Indeed, if we are required to search through an entire file for a piece of data, we might be required to wait for hours! For this reason (as well as others), we usually organize the contents of a file in some way, so that access to its data is improved. Here are some examples of common file organizations:

1. A file of data on customers may be arranged in alphabetical order, according to the customer name.
2. A mailing list may be arranged according to ZIPCODE.
3. An inventory list might be arranged according to part number.
4. A credit card company most likely arranges its customer account files according to their credit card number.

In each example, the records in the data file are arranged in a certain order, based on the value of a particular field in the record (name field, ZIPCODE field, part number field, card number field). In maintaining such files, it is essential to be able to arrange the records in the desired order. The process of arranging a set of data items is called **sorting**. Actually, sorting is an extremely important topic to computer programmers and has been the subject of many research papers and books. In this section, we will give an introduction to sorting by describing one of the more elementary sorting techniques—the **bubble sort**.

Let's begin by stating our problem in simple terms. Let's suppose that we wish to arrange the records of a file according to a particular field, say field 1.

PROBLEM: Arrange the records so that the values in field 1 are in ascending order.

For the sake of our initial discussion, let's suppose that the field values are numbers. (Later, we will deal with fields containing strings.)

Let's set up arrays A() and B() as follows: Read the various values of field 1 into the array A().

A(1) = the value of field 1 for record 1,
A(2) = the value of field 1 for record 2,
A(3) = the value of field 1 for record 3,

and so forth. We wish to rearrange the records according to certain rules. Because the actual records may be quite long, we will deal only with the contents of field 1. In order to keep track of the record to which a particular field value belongs, we will use the array B(). That is,

B(1) = the record number for the field value A(1),
B(2) = the record number for the field value A(2),
B(3) = the record number for the field value A(3),

and so forth. Assume that we initially read the values into array A() according to increasing record number. Then we initially have

B(1)=1, B(2)= 2, B(3)=3, ...

The Bubble Sort Procedure

The **bubble sort** procedure allows you to arrange a set of numbers in increasing order. It involves repeatedly executing a simple reordering process that involves reordering consecutive items. Each repetition of the process is called a **pass.** Let's illustrate the procedure to arrange the following list of numbers in increasing order:

90, 38, 15, 48 , 80, 1

Pass 1. Start from the right end of the list. Compare the adjacent numbers. If they are out of order, switch them. Otherwise leave them alone. Continue this procedure with each pair of adjacent numbers, proceeding from right to left. Here are the results:

90, 38, 15, 48, 1, 80	(1 < 80 so the pair 80,1 is reversed)
90, 38, 15, 1, 48, 80	(1 < 48 so the pair 48,1 is reversed)
90, 38, 1, 15, 48, 80	(1 < 15 so the pair 15,1 is reversed)
90, 1, 38, 15, 48, 80	(1 < 38 so the pair 38,1 is reversed)
1, 90, 38, 15, 48, 80	(1 < 90 so the pair 90,1 is reversed)

This is the end of Step 1. Note that the number 1 has assumed its correct place in the list.

Pass 2. Apply the procedure of Step 1 to the rightmost five numbers of the current list.

1, 90, 38, 15, 48, 80 (48 < 80 so no exchange)
1, 90, 38, 15, 48, 80
1, 90, 15, 38, 48, 80
1, 15, 90, 38, 48, 80

Note that the number 15 has now been moved to its proper position on the list.

Pass 3. Apply the procedure of Step 1 to the rightmost four numbers of the current list.

1, 15, 90, 38, 48, 80
1, 15, 90, 38, 48, 80
1, 15, 38, 90, 48, 80

Pass 4. Apply the procedure of Step 1 to the rightmost three numbers of the current list.

1, 15, 38, 90, 48, 80
1, 15, 38, 48, 90, 80

Pass 5. Apply the procedure of Step 1 to the rightmost two numbers of the current list.

1, 15, 38, 48, 80, 90

The list is now in order.

Note the following characteristic of the bubble sort procedure. At each step, the smallest remaining number is moved to its proper position in the list. Suppose that we view the original list as written vertically:

90
38
15
48
1
80

Then at each step, the least number in the remaining list moves to its proper level in the list. Think of each number as a bubble under water, whose buoyancy is determined by the value of the number. Then at each step, a bubble moves up as far as it can toward the surface. This is the reason for the name **bubble sort**.

We have carried out the manipulations in the above example in excruciating detail to aid us in writing a correct program to implement the bubble sort procedure. Let's suppose that the items to be ordered are stored in the array A() of size N. Here is a program that carries out the bubble sort procedure.

```
200 'Bubble Sort Subroutine
210 FOR I=2 TO N
220     FOR J=N TO I STEP -1
230             IF A(J-1) > A(J) THEN SWAP A(J-1), A(J)
240     NEXT J
250 NEXT I
260 RETURN
```

Note that we have written this program as a subroutine to be included in a larger program. Note that the DIM statement for the array A() as well as the number N of numbers to be sorted must be set in the larger program. You may test this program with the sequence of numbers 100, 99, 98, 97, 96, ..., 1 by inserting the lines of code:

```
10 DIM A(100)
20 N=100
30 FOR J=1 TO N
40   A(J)=101-J
50 NEXT J
60 GOSUB 200
70 FOR J=1 TO 100
80   PRINT J, A(J)
90 NEXT J
100 END
```

We may use this routine to infer some interesting characteristics of sort routines. Here is a set of run times for various values of N, using the sequence N, N-1, N-2,... , 1. (This is the worst case since interchanges are required at each step.)

Value of N	Run Time for Bubble Sort
N = 100	67 seconds
N = 50	17 seconds
N = 20	4 seconds
N = 10	1 second

First note that, with only 100 items to be sorted, the run time is already substantial. Second, note the way that the run time increases as the number of items increases. It appears that if the number of items is doubled then the run time increases by a factor of four. Similarly, multiplying the number of items by three increases the run time by nine. Generally, in this worst-case scenario, multiplying the number of items by k multiplies the run time by k^2. On average, the run times are not this bad. However, we have chosen a particularly bad case to illustrate the manner in which sorting times quickly become unmanageable.

One of the principal problems with our bubble sort procedure is the fact that it is written in interpretive BASIC. In order to do any serious sorting, it is necessary to compile the program into machine language. This may be done using the BASIC compiler.

Let's return to our original problem, namely that of sorting the records of a file. Let's use the bubble sort procedure to sort the array A(). However, at each interchange, we will interchange the corresponding elements of the array B(). At the end of the subroutine, the array A() will be in ascending order and B(J) will equal the number of the record from which A(J) was taken. Here is the program:

```
200 'Bubble Sort Subroutine for File Records
210 FOR I=2 TO N
220     FOR J=N TO I STEP -1
230             IF A(J-1) > A(J) THEN SWAP A(J-1),
                A(J):SWAP B(J-1),B(J)
240     NEXT J
250 NEXT I
260 RETURN
```

The array B() may be stored in a file and used to read out the records of the file according to the increasing order of the particular field.

The bubble sort procedure performs particularly poorly for data that is almost in order and is sorted into the correct order by one of the early passes. The procedure as stated above has no way of knowing that that data is already in order and that no further sorting is necessary. Let's now improve the bubble sort algorithm by building a test into each pass that will determine whether any further sorting is necessary.

Our test is based on the value of a variable SORTFLAG. Initially, we set SORTFLAG equal to 0. During each pass, we set SORTFLAG equal to 1 when an interchange takes place. At the end of the pass, we examine the value of SORTFLAG. If SORTFLAG is 0, then no interchange took place and the algorithm is terminated. Otherwise, SORTFLAG is set equal to 0, and the algorithm goes on to the next pass. Here is the code for the modified bubble sort routine.

```
200 'Modified Bubble Sort Subroutine
210 SORTFLAG=0
220 FOR I=2 TO N
230     FOR J=N TO I STEP -1
240             IF A(J-1) > A(J) THEN SWAP A(J-1), A(J):
                SORTFLAG=1
250     NEXT J
260     IF SORTFLAG=0 THEN I=N ELSE SORTFLAG=0
270 NEXT I
280 RETURN
```

Note the logic in line 260. If SORTFLAG is equal to 0, then the loop variable I is set equal to N. In this case, the NEXT I in line 270 causes the I loop to terminate. Otherwise, SORTFLAG is set equal to 0 and the next value of I is considered.

TEST YOUR UNDERSTANDING 1

Compare the times required by both the original and modified bubble sort routines in sorting the following list of numbers into ascending order:

1, 2, 3, 4, 5, ..., 95, 100, 99, 98, 97, 96

In this section, we have only scratched the surface of the subject of sorting. For an extensive treatment, see **Algorithms + Data Structures = Programs** by Niklaus Wirth, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1976.

Exercises

1. Write a program to alphabetize the elements of the array A$() using the bubble sort procedure.
2. Test the program of Exercise 1 using the array A$(1)="Z", A$(2)="Y", ..., A$(26)="A".
3. Write a program that creates an array of N random numbers, where N is given in an INPUT statement. The program should arrange the array in increasing order and should use the clock to time the operation. Make a table of sort times for various values of N. (Be sure to use RANDOMIZE to create non-repetitive arrays.)
4. Write a program that determines the smallest element in field 1 in the various records of the file "TEST". (Assume that the file is 25600 bytes long, the record length is 128 bytes, and field 1 is 4 bytes.

8.7 BASIC File Commands

Cartridge BASIC has a number of useful commands which you may use to perform various manipulations on files.

The Directory

You may request, from within BASIC, a directory of the files on a given diskette. This may be done using the FILES command. For example, to list all the files on the current diskette, type

```
FILES
```

and press ENTER. This command is similar to the DOS command DIR, except that this command is used within BASIC.

The FILES command is very versatile. You use it to provide a listing of all files matching a given filespec. For example, to list all files having an extension .BAS on drive B:, use the command:

```
FILES "B:*.BAS"
```

Similarly, to list all files on drive B:, use the command:

```
FILES "B:*.*"
```

Note that this last command may be abbreviated to:

```
FILES "B:"
```

TEST YOUR UNDERSTANDING 1 (answer on page 213)
Write a command that lists all files on the current drive with an extension that begins with the letter B.

Erasing Files

You may erase files using the command KILL. The format of this command is:

```
KILL <file specification>
```

For example, to erase the file EXAMPLE.TXT, use the command:

```
KILL "EXAMPLE.TXT"
```

To erase all files on the diskette in drive B:, use the command:

```
KILL "*.*"
```

This last form of the KILL command is very dangerous. You might be erasing some files you don't really want to erase. Use this form of the KILL command with some care.

Note that the KILL command may be used to erase program files as well as data files.

In order to KILL a file, you must include any file name extension in the file name. Be careful here. If the file is a BASIC program and if you saved it without specifying an extension, BASIC automatically added the extension BAS. In order to specify the file name for the KILL command, you must include the extension BAS.

TEST YOUR UNDERSTANDING 2 (answer on page 213)
Write BASIC commands to erase the following files:

a. The BASIC program named "COLORS"
b. The BASIC program "INVOICE.001"

Renaming a File

You may rename a file by using the **NAME** command. To change the name of ROULETTE to GAME, we use the command:

```
NAME "ROULETTE" AS "GAME"
```

Note that the old name always comes first, followed by the new name. An error will occur if either ROULETTE doesn't exist or if there is already a file on the diskette with the name GAME.

To rename a program file, you must include any file name extension in the old file name. Be careful here. If the old file is a BASIC program, and if you saved it without specifying an extension, BASIC automatically added the extension BAS. To specify the file name for the NAME command, you must include the extension BAS.

Saving Programs

As we learned in Chapter 2, you may save the current program on diskette, using the SAVE command. Let's take this opportunity to point out a few additional features of this command. BASIC allows a program to be saved in any of three alternate formats—compressed format, ASCII format, and protected format.

Compressed Format This is the format we have used to save programs up till now. In this format, the various words of BASIC (LET, PRINT, IF, THEN, etc.) are reduced to a numerical shorthand, which allows the program to be stored in reduced space. The compressed format is also called **tokenized**.

ASCII Format In ASCII format, the program is stored letter for letter as you typed it. This requires more diskette space. However, it allows the program to be MERGEd and CHAIN MERGEd. (See below.) Also, a program file must be saved in ASCII format if it is to be used as a source code file for the BASIC compiler.

To save a program in ASCII format, use the command

```
SAVE <filespec>, A
```

For example, to save the program TAXES on the diskette in drive B: in ASCII format, we could use the command

```
SAVE "B:TAXES",A
```

Protected Format Once a program has been saved in protected format, it may not be listed. This provides a mild degree of protection against snoopers. To save TAXES on drive B: in binary format, we could use the command

```
SAVE "B:TAXES",P
```

BASIC provides no way to translate a program from binary back into a listable format, so use this format with some care.

Merging Programs

BASIC has the ability to merge the program currently in RAM with any other program on a diskette. This is especially useful in inserting standard sub-

routines into a program and is accomplished using the **MERGE** command. For example, to merge the current program with the program PAYROLL we use the command:

```
MERGE "PAYROLL"
```

Suppose the program currently in RAM contained lines 10, 20, 30, and 100, and PAYROLL contained lines 40, 50, 60, 70, 80, 90, and 100. The merged program would contain the lines 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Line 100 would be taken from PAYROLL. (The lines of PAYROLL would replace those of the current program in case of duplicate line numbers.) To use the merge feature, the program from diskette must have been **SAVE**d in ASCII format. In the case of the above example, the command that **SAVE**d PAYROLL must have been of the form:

```
SAVE "PAYROLL", A
```

In case PAYROLL was not **SAVE**d using such a command, it is first necessary to **LOAD** "PAYROLL" and resave it using the above command. (Watch out! If you type in a program, say OX as an example, to merge with PAYROLL, remember to save it before giving the **MERGE** command. If you don't, you will lose OX.)

TEST YOUR UNDERSTANDING 3 (answer on page 213)

a. Save the following program in ASCII format under the name GHOST.

```
10 PRINT 5+7
100 END
```

b. Type in the program

```
30 PRINT 7+9
40 PRINT 7-9
```

c. **MERGE** the programs of a. and b.

Exercises (answers on page 367)

1. a. Write a program that computes $1^2+2^2+\ldots+50^2$.
   b. **SAVE** the program under the name SQUARES. Use the **SAVE, A** command.
2. a. Write a program that computes $1^3+2^3+\ldots+30^3$. Write this in such a way that the line numbers do not overlap with those of the program in 1a.
   b. **MERGE** the program of 1a with the program of 2a.
   c. **LIST** the **MERGE**d program.

   d. **RUN** the **MERGE**d program.
   e. **SAVE** the **MERGE**d program under the name COMBINED.
3. Recover the program of 2a without retyping it.
4. Erase the program SQUARES of 1a.

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
FILES "*.B*"
```
2: a.
```
KILL "COLORS.BAS"
```
   b.
```
KILL "INVOICE.001"
```
3: a. Type in the program, then give the command: SAVE "GHOST",A
   b. Type NEW followed by the given program.
   c. Type MERGE "GHOST".

9

STRING MANIPULATION

In this chapter we discuss some of the fine points about strings. You may view this discussion as a prelude to our discussion of word processing in Chapter 11.

9.1 ASCII Character Codes

Each keyboard character is assigned a number between 1 and 255. The code number assigned is called the **ASCII code** of the character. For example, the letter "A" has ASCII code 65, while the letter "a" has ASCII code 97. Also included in this correspondence are the punctuation marks and other keyboard characters. As examples, 40 is the ASCII code of the open parenthesis "(" and 62 is the ASCII code of the "greater than" symbol >.

Even the keys corresponding to non-printable characters have ASCII codes. For example, the space bar has ASCII code 32, and the backspace key, ASCII code 8. The printable characters have ASCII codes between 32 and 127. Table 9-1 lists all these characters and their corresponding ASCII codes.

ASCII Code	Character	ASCII Code	Character
32	blank		
33	!	51	3
34	"	52	4
35	#	53	5
36	$	54	6
37	%	55	7
38	&	56	8
39	'	57	9
40	(	58	:
41	)	59	;
42	*	60	<
43	+	61	=
44	,	62	>
45	–	63	?
46	.	64	@
47	/	65	A
48	0	66	B
49	1	67	C
50	2	68	D

69	E	99	c
70	F	100	d
71	G	101	e
72	H	102	f
73	I	103	g
74	J	104	h
75	K	105	i
76	L	106	j
77	M	107	k
78	N	108	l
79	O	109	m
80	P	110	n
81	Q	111	o
82	R	112	p
83	S	113	q
84	T	114	r
85	U	115	s
86	V	116	t
87	W	117	u
88	X	118	v
89	Y	119	w
90	Z	120	x
91	[	121	y
92	\	122	z
93	]	123	{
94	^	124	¦
95	_	125	}
96	`	126	~
97	a	127	⌂
98	b		

Table 9-1. ASCII character codes for printable characters.

We will discuss the ASCII control codes 0-31 in Section 9.3. For now, however, let's call attention to just two:

ASCII Code	Name	Action
10	Line Feed	Moves cursor down one line
13	Carriage Return	Moves cursor to the leftmost position on the current line

Pushing the ENTER key generates both a carriage return and a line feed. That is, ENTER generates the two ASCII codes 13 and 10.

The computer uses ASCII codes to refer to letters and control operations. Any file, whether it is a program or data, may be reduced to a sequence of ASCII codes. Consider the following address:

John Jones
2 S. Broadway

As a sequence of ASCII codes, it would be stored as

74,111,104,110,32,74,111,110,101,115,13,10
50,32,83,46,32,66,114,111,,97,100,119,97,121,13,10

Note that the spaces are included (number 32) as are the carriage returns and line feeds (created by pressing ENTER) at the end of each line (numbers 13 and 10).

ASCII codes allow us to represent any text generated by the keyboard as a sequence of numbers. This includes all formatting instructions like spaces, carriage returns, upper- and lowercase letters, and so forth. Moreover, once a piece of text has been reduced to a sequence of ASCII codes, it also may be faithfully reproduced on the screen or on a printer.

TEST YOUR UNDERSTANDING 1 (answer on page 219)
Write a sequence of ASCII codes which will reproduce this ad:

FOR SALE: Beagle puppies. Pedigreed.
8 weeks. $125.

You may refer to characters by their ASCII codes by using the function CHR$. For example, CHR$(74) is the character corresponding to ASCII code 74 (uppercase J); CHR$(32) is the character corresponding to ASCII code 32 (space). The **PRINT** and **LPRINT** instructions may be used in connection with CHR$. For example, the instruction

```
10 PRINT CHR$(74)
```

will display an uppercase J in the first position of the first print field.

TEST YOUR UNDERSTANDING 2 (answer on page 219)
Write a program which will print the ad of TEST YOUR UNDERSTANDING 1 from its ASCII codes.

To obtain the ASCII code of a character, use the instruction **ASC**. For example, the instruction

```
20 PRINT ASC("B")
```

will print the ASCII code of the character "B", namely 66. In place of "B", you may use any string. The computer will return the ASCII code of the first character of the string. For example, the instruction

```
30 PRINT ASC(A$)
```

will print the ASCII code of the first character of the string A$.

TEST YOUR UNDERSTANDING 3 (answer on page 219)
Determine the ASCII codes of the characters $, g, X, and + without looking at the chart.

ASCII codes have many uses in writing even the most simple programs. For example, suppose that you wish to print out a quotation mark on the screen. To do so, you must create a string which consists of a quotation mark. The usual way to define a string is to enclose it in quotation marks. However, if you attempt to do that in this case, you arrive at: " " " Unfortunately, here is how BASIC looks at that string: The first quotation mark tells BASIC that a string is about to begin. The second quotation mark tells BASIC that the string just ended. The third quotation mark is ignored! So, for example, the command:

```
PRINT """
```

will print nothing on the screen!

The ASCII codes provide a way out of this dilemma. The ASCII code of " is 34, and CHR$(34) is a string consisting of a single quotation mark. So we may print " on the screen with the statement:

```
PRINT CHR$(34)
```

In a similar fashion, ASCII codes may be used to include carriage returns and line feeds within a string. Note that you cannot type a string which includes carriage returns or line feeds from the keyboard. Hitting ENTER is a signal for BASIC to accept the line just typed. However, it will not include the carriage return and line feed as part of the string. This must be done using ASCII codes. (More about how this is done in Section 2.)

Exercises (answers on page 367)

1. Determine the ASCII codes of the following characters without looking at the table: A, a, B, b, C, c, D, d .
2. Generalize from Exercise 1 to state a relationship between the ASCII code of a capital letter and the ASCII code of the corresponding lowercase letter (that is, between A and a).
3. Display the sequence of ASCII codes corresponding to the following sentence.

He said to me, "The PCjr is an excellent computer".

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
70,79,82,32,83,65,76,69,58,32,66,101,97,103,108,
101,32,112,117,112,112,105,101,115,46,32,80,101,
100,105,103,114,101,101,100,46,13,10,56,32,119,
101,101,107,115,46,32,36,49,50,53,46,13,10
```

2:
```
10 DATA 70,79,.........(insert data from 1)
11 DATA .........
12 DATA .........
20 FOR J=1 TO 44
30   READ A
40   PRINT CHR$(A);
50 NEXT J
60 END
```

3:
```
10 DATA $,g,X,+
20 FOR J=1 TO 4
30    READ A$
40    B=ASC(A$)
50    PRINT A$, B
60 NEXT J
70 END
```

9.2 Operations on Strings

In earlier chapters, our strings contained only printable characters. Let us now extend that definition to allow characters corresponding to any ASCII code. So, for example, a string may now include line feeds, carriage returns, and any of the other control characters we soon will define. The control characters in a string are treated just like any of the other characters.

BASIC lets you perform a number of different operations on strings. The most fundamental operation is **string addition** (or, in computer jargon, **string concatenation**). Suppose that A$ and B$ are strings, with A$ = "word" and B$ = "processor". Then the sum of A$ and B$, denoted A$ + B$, is the string obtained by adjoining A$ and B$, namely:

```
"wordprocessor"
```

Note that no space is left between the two strings. To include a space, suppose that C$ = " ". C$ is the string which consists of a single space. Then A$ + C$ + B$ is the string

```
"word processor"
```

TEST YOUR UNDERSTANDING 1 (answer on page 225)
If A$="4" and B$="7", what is A$+B$?

TEST YOUR UNDERSTANDING 2 (answer on page 225)
Set A$ equal to the string:
He said, "No".<carriage return><line feed>

You may compute the length of a string by using the **LEN** function. For example,

```
LEN("BOUGHT")
```

is equal to six, since the string "BOUGHT" has six letters. Similarly, if A$ is equal to the string

```
"Family Income"
```

then **LEN(A$)** is equal to 13. (The space between the words counts!) Note that carriage returns, line feeds, and other control characters count in the length.

Here is an application of the **LEN** instruction.

Example 1. Write a program which inputs the string A$ and then centers it on a line of the display. (Assume an 80-character line.)
Solution. A line is 80 characters long, with the spaces numbered from 1 to 80. The string A$ takes up LEN(A$) of these spaces, so there are 80-LEN(A$) spaces to be distributed on either side of A$. The line should begin with half of the 80-LEN(A$) spaces, or with (80-LEN(A$))/2 spaces. So we should tab to column (80-LEN(A$))/2+1. Here is our program.

```
10 INPUT A$
20 CLS
30 PRINT TAB((80-LEN(A$))/2+1) A$
40 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 225)
Use the program of Example 1 to center the string "THE IBM PCjr".

It is possible to dissect strings using the three instructions **LEFT$**, **RIGHT$**, and **MID$**. These instructions allow you to construct a string consisting of a specified number of characters taken from the left, right, or middle of a designated string. Consider the instruction

```
10 A$=LEFT$("LOVE",2)
```

The string A$ consists of the two leftmost characters of the string "LOVE". That is, A$ = "LO". Similarly, the instructions

```
20 B$="tennis"
30 C$=RIGHT$(B$,3)
```

set C$ equal to the string consisting of the three rightmost letters of the string B$, namely C$ = "nis". Similarly, if A$ = "Republican", then the instruction

```
40 D$=MID$(A$,5,3)
```

sets D$ equal to the string which consists of the three characters starting with the fifth character of A$, which is D$ = "bli".

TEST YOUR UNDERSTANDING 4 (answer on page 225)
Determine the string constant:

```
RIGHT$(LEFT$("computer",4),3)
```

In manipulating strings, it is important to recognize the difference between numerical data and string data. The number 14 is denoted by 14 ; the string consisting of the two characters 14 is denoted "14". The first is a numerical constant and the second a string constant. We can perform arithmetic using the numerical constants. However, we cannot perform any of the character manipulation supplied by the instructions **RIGHT$**, **MID$** and **LEFT$**. Such manipulation may only be performed on strings. How may we perform character manipulation on numerical constants? BASIC provides a simple method. We first convert the numerical constants to string constants by using **STR$**. For example, the number 14 may be converted into the string " 14" using the instruction:

```
10 A$=STR$(14)
```

As a result of this instruction, A$ has the value " 14". Note the blank in front of the 14. This occurs because BASIC automatically leaves a space for the sign of a number. If the number is positive, then the sign prints out as a space. If the number is negative, then the sign prints out as a minus (-). As another example, suppose that the variable B has the value 1.457. **STR$(B)** is then equal to the string " 1.457".

To convert strings consisting of numbers into numerical constants, use **VAL**. Consider this instruction:

```
20 B=VAL("3.78")
```

This instruction sets B equal to 3.78. You may even use **VAL** for strings consisting of a number followed by other characters. **VAL** will pick off the initial

number portion and throw away the part of the string which begins with the first non-numerical character. For example, VAL("12.5 inches") is equal to 12.5.

TEST YOUR UNDERSTANDING 5 (answer on page 225)
Suppose that A$ equals "5 percent" and B$ equals "758.45 dollars". Write a program which starts from A$ and B$ and computes five percent of $758.45.

The INSTR Statement

In some applications, it is necessary to search a string for a particular pattern. Here are some examples of such searches:

Find the location of the first "A" in the string A$.
Find the location of the first period in the string B$.
Find the location of the first "1" in A$ occurring after the eighth character.
Does the sequence of characters "ABS" occur anywhere in the string A$?

All such searches are greatly simplified using the INSTR (=INSTRing) function. This function may be used in either of two formats. The simplest is:

```
10 P=INSTR(A$,B$)
```

In response to this statement, P is set equal to the location of the first occurrence of B$ in A$. For example, suppose that:

```
A$="This is a test of the INSTR statement."
B$="te"
```

In this case, the first occurrence of B$ in A$ is at the beginning of the word "test". The location of the initial t is the eleventh character. So INSTR(A$,B$) has the value 11.

If B$ does not occur in A$, then INSTR has the value zero. Therefore, to determine whether the string "ABS" occurs in A$, we could use the program:

```
10 P=INSTR(A$,"ABS")
20 IF P=0 THEN PRINT "ABS DOES NOT OCCUR"
30 IF P>0 THEN PRINT "ABS OCCURS"
```

The second format of the INSTR statement allows you to begin the search for B$ beginning with a designated location m. In this format the statement has the form

```
P=INSTR(m,A$,B$)
```

For example, if we wish the search for B$ to begin with the eighth character of A$, we could use the instruction

```
P=INSTR(8,A$,B$)
```

Order Relations Among Strings

We arrange single characters in order of their respective ASCII codes. We say that a character A$ is **less than** the character B$ provided that A$ comes before B$ in the ASCII table. If A$ is less than B$, we write:

```
A$ <B$
```

For example, the following are valid inequalities among characters:

"A" < "B" ("A" has ASCII code 65,
"B" has ASCII code 66)
"a" < "b" ("a" has ASCII code 97,
"b" has ASCII code 98)

Note that arranging alphabetic characters in ascending order amounts to arranging them in alphabetic order. However, the following additional comparisons are valid and are not usually considered in alphabetic arrangements:

"A" < "a"
"0" < "a" ("0" has ASCII code 48)
"*" > "#" ("*" has ASCII code 42,
"#" has ASCII code 35)
" " < "0" (" " has ASCII code 32)

Strings having more than a single letter are compared as follows: First compare first letters. If they are the same, compare second letters. If the first two letters are the same, compare third letters. And so forth. For example, consider the two strings "Smith" and "SMITH". Their first letters are the same, so we compare their second letters "m" and "M", respectively. According to their ASCII codes "M" comes before "m", so:

```
"SMITH" < "Smith"
```

If the compared strings consist of only uppercase or only lowercase letters, then this comparison procedure will arrange the strings in the usual alphabetic order. However, the procedure may be used to compare any strings. For example:

```
"**#" < "**0"
```

Here is a bit of useful notation for strings: The notation A$>=B$ means that either A$>B$ or A$=B$. Simply, this means that A$ either succeeds B$ in alphabetical order, or A$ and B$ are the same. The notation A$<=B$ has a similar meaning.

Using the above string order relation, we may design a modified bubble sort procedure for sorting a string array A$() into increasing order. Here is the subroutine:

```
300 'Modified Bubble Sort Subroutine for Strings
310 SORTFLAG=0
320 FOR I=2 TO N
330     FOR J=N TO I STEP -1
340             IF A$(J-1) > A$(J) THEN SWAP A$(J-1), A$(J):
                SORTFLAG=1
350     NEXT J
360     IF SORTFLAG=0 THEN I=N ELSE SORTFLAG=0
370 NEXT I
380 RETURN
```

When this routine is used to sort an array consisting only of uppercase or only of lowercase letters, it will sort the array into alphabetical order. Here is an example of this procedure.

Example 2. Write a program which alphabetizes the following list of words: egg, celery, ball, bag, glove, coat, pants, suit, clover, weed, grass, cow, and chicken.

Solution. We set up a string array A$(J) which contains these 13 words and apply the bubble sort subroutine.

```
100 DIM A$(13)
110 DATA egg,celery,ball,bag,glove,coat
120 DATA pants, suit, clover, weed, grass
130 DATA cow, chicken
140 'Set up array A$
150      FOR J=1 TO 13
160              READ A$(J)
170      NEXT J
180 'Sort array A$()
190 GOSUB 300
200 'Print Sorted Array
210 FOR J=1 TO 13
220  PRINT A$(J)
230 NEXT J
240 END
300 'Modified Bubble Sort Subroutine for Strings
310 SORTFLAG=0
320 FOR I=2 TO N
330     FOR J=N TO I STEP -1
340             IF A$(J-1) > A$(J) THEN SWAP A$(J-1), A$(J):
                SORTFLAG=1
350     NEXT J
360     IF SORTFLAG=0 THEN I=N ELSE SORTFLAG=0
370 NEXT I
380 RETURN
```

This program can be modified to make a program alphabetizing any collection of strings. We will leave the details to the exercises.

Exercises (answers on page 368)

1. Use the program of Example 2 to alphabetize the following sequence of words: justify, center, proof, character, capitalize, search, replace, indent, store, and password.
2. Write a program which rewrites the addition problem 15+48+97 = 160 in the form

```
 15
 48
 97
160
```

3. Write a program which inputs the string constants "$6718.49" and "$4801.96" and calculates the sum of the given dollar amounts.

ANSWERS TO TEST YOUR UNDERSTANDING

1: `"47"`

2:
```
A$="He said, "+CHR$(34)+"No"+CHR$(34)+"."+
CHR$(13)+CHR$(10)
```

3: Type RUN and press ENTER. When prompted, type in the given string.

4: `"omp"`

5:
```
10 A$="5 percent":B$="758.45 dollars"
20 A=VAL(A$):B=VAL(B$)
30 PRINT A$,"OF",B$,"IS"
40 PRINT A*B*.01
50 END
```

9.3 Control Characters

Table 9-2 contains a list of the control characters corresponding to ASCII codes 0-31. Some comments on the functions of the various codes are in order.

Code 000 (null) is exactly what its name suggests. It is a character which does nothing. It often is used in communications, where a message will be started with a string of nulls.

Codes 001-006 are graphics characters. You use them for games.

Code 7 (beep) beeps the speaker of the computer.

Code 8 (backspace) backspaces the cursor one space.

Code 9 (tab) moves the cursor to the next tab stop. BASIC automatically places tab stops every five characters across the line.

ASCII value	Character	Control character
000	(null)	NUL
001	☺	SOH
002	☻	STX
003	♥	ETX
004	♦	EOT
005	♣	ENQ
006	♠	ACK
007	(beep)	BEL
008	◘	BS
009	(tab)	HT
010	(line feed)	LF
011	(home)	VT
012	(form feed)	FF
013	(carriage return)	CR
014	♫	SO
015	☼	SI
016	►	DLE
017	◄	DC1
018	↕	DC2
019	!!	DC3
020	¶	DC4
021	§	NAK
022	▬	SYN
023	↨	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESC
028	(cursor right)	FS
029	(cursor left)	GS
030	(cursor up)	RS
031	(cursor down)	US

*Coutesy of International Business Machines Corporation.

Table 9-2. ASCII codes for control characters.

Code 10 (line feed) advances the cursor one line down.

Code 11 (home) positions the cursor at the upper left corner of the screen.

Code 12 (form feed) advances the paper on the printer to the top of the next page.

Code 13 (carriage return) returns the cursor to the leftmost position on the current line.

Codes 14-27 are further graphics characters for use in displays.

Code 28 (cursor right) moves the cursor to the right one space.

Code 29 (cursor left) moves the cursor to the left one space.

Code 30 (cursor up) moves the cursor up one space.

Code 31 (cursor down) moves the cursor down one space.

To use the ASCII control codes, you PRINT them as if they were printable characters. For example, to move the cursor one space up, use the statement

```
PRINT CHR$(30);
```

Note that the statement ended with a semicolon (;). This is to prevent BASIC from issuing a carriage return and line feed following the PRINT statement. Otherwise, they would ruin the cursor positioning accomplished by control character 30.

More on the Cursor

The ASCII codes controlling cursor motion allow you to position the cursor relative to its current position. You may move the cursor to a specific position on the screen using the LOCATE statement. The format of this statement is:

```
LOCATE row,column
```

For example, to position the cursor in column 5 of row 20, use the statement:

```
LOCATE 20,5
```

We can determine the column in which the cursor is currently located by using the BASIC function **POS(0)**. For example, if the cursor currently is located in column 37, then **POS(0)** is equal to 37. The variable **CSRLIN** always equals the number of the line in which the cursor currently is located.

For example, if the cursor currently is located in line 5, then **CSRLIN** is equal to 5. You may use **POS(0)** and **CSRLIN** exactly as you would any other variables in BASIC.

TEST YOUR UNDERSTANDING 1 (answer on page 228)
Write a program to move the cursor two spaces to the right and two spaces down.

Exercises (answers on page 368)

1. Print the string "HELLO" and then move the cursor to the H.
2. Write a program which asks the user for some input. In response to an S, it moves the cursor to the left; in response to a D, it moves the cursor to the right; in response to an E, it moves the cursor up; in response to an X, it moves the cursor down. In response to any other input, the program should do nothing.
3. Modify the program of Exercise 2 so that it accepts a series of cursor moves of the type "SSDDXEEEESSSDDD".
4. Practice moving the cursor to various positions on the screen.
5. Write a program moving the cursor to the bottom of the column in which it currently resides.
6. Write a program moving the cursor to the left of the screen in the row it is now on.

ANSWER TO TEST YOUR UNDERSTANDING

```
1:  10 PRINT CHR$(28);CHR$(28);CHR$(31);CHR$(31);
    20 END
```

10

INTRODUCTION TO GRAPHICS AND SOUND ON THE IBM PCjr

The PCjr is capable of quite sophisticated graphics and sound. In this chapter, we introduce you to these capabilities.

10.1 Graphics in Text Mode

When you first start BASIC, the screen is in **text mode**, in which the video display can display only characters from the PCjr character set. (More about that below.) In text mode, the display contains 25 rows of either 40 or 80 characters each. You may change from 80-character to 40-character width using the WIDTH statement. The various character positions divide the screen into small rectangles. Figure 10-1 shows the subdivision of the screen corresponding to an 80-character line width.

The rectangles into which we have divided the screen are arranged in rows and columns. The rows are numbered from 1 to 25, with row 1 at the top of the screen and row 25 at the bottom. The columns are numbered from 1 to 80, with column 1 at the extreme left and column 80 at the extreme right. Each rectangle on the screen is identified by a pair of numbers, indicating the row and column. For example, the rectangle in the 12th row and 16th column is shown in Figure 10-2.

We may print characters on the screen using the PRINT and PRINT USING instructions. For graphics purposes, it is important to be able to precisely position characters on the screen. This may be done using the **LOCATE** instruction. Remember that printing always occurs at the current cursor location. To locate the cursor at row x and column y, we use the instruction

```
100 LOCATE x,y
```

Example 1. Write a set of BASIC instructions to print the words "IBM PCjr Computer" beginning at row 20, column 10.

Figure 10-1. Screen layout for text mode (80-character width).

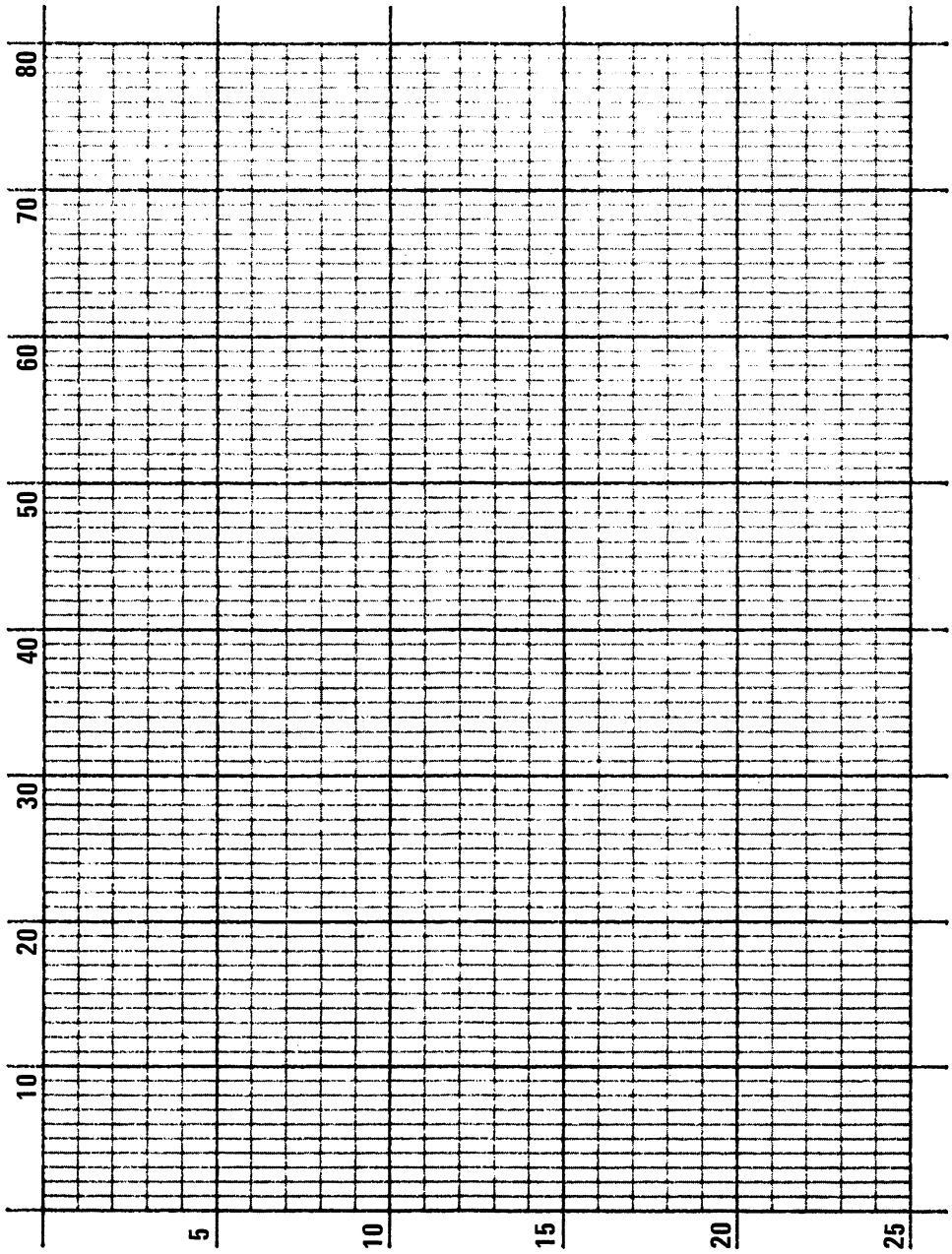
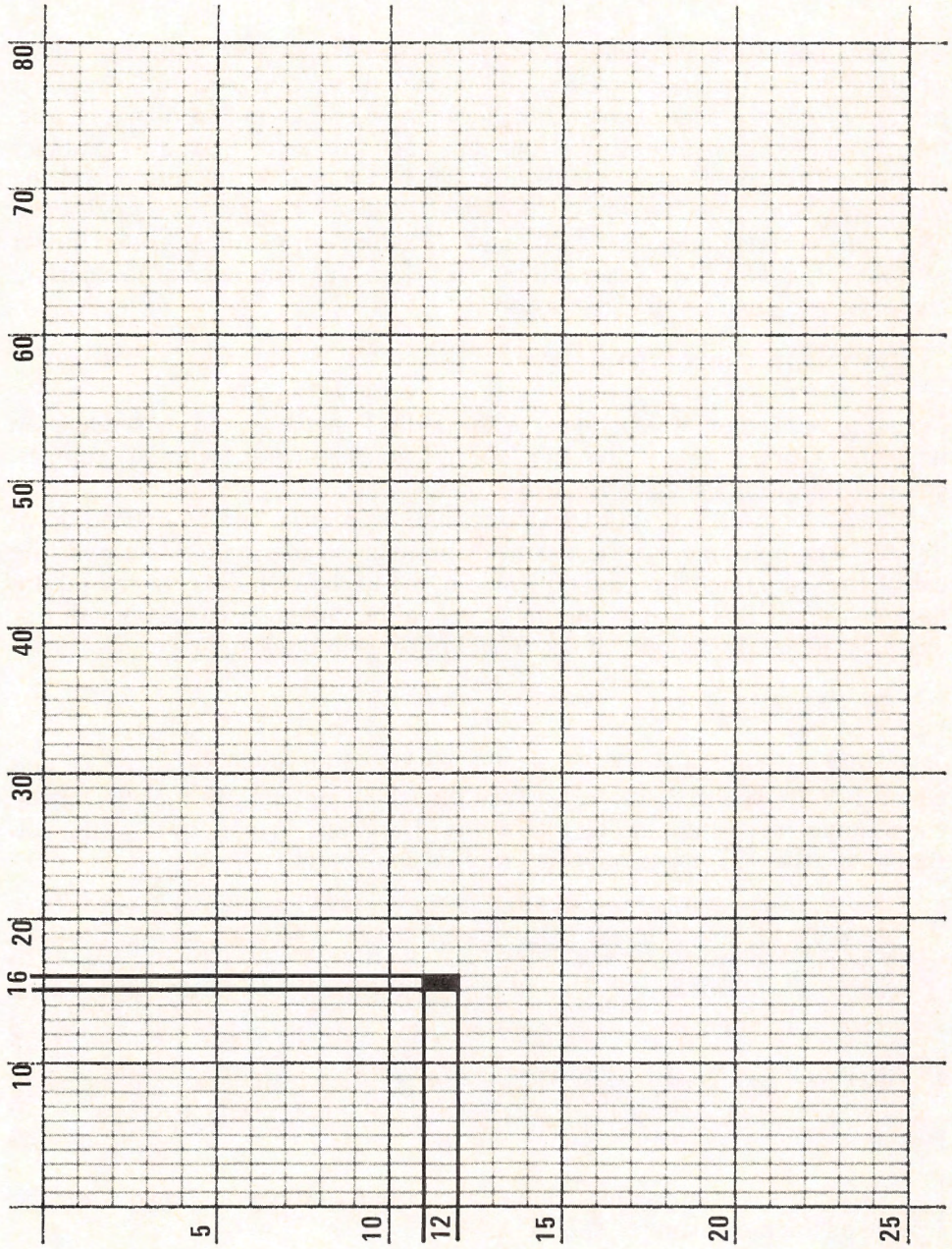


Figure 10-2.



Solution.

```
10 LOCATE 20,10
20 PRINT "IBM PCjr Computer"
```

Until now, we have printed only characters such as those found on a typewriter keyboard (letters, numbers, and punctuation marks). Actually, the PCjr has a very extensive set of characters, including a collection of graphics characters, as shown in Figure 10-3. Note that each character (including graphics characters) is identified by an ASCII code. In Chapter 9, we introduced the characters corresponding to ASCII codes 0-127. In Figure 10-3, we list the characters corresponding to ASCII codes 128-255. For example, the character with ASCII code 179 is a vertical line. To place this character at the current cursor position, we use the instruction

```
30 PRINT CHR$(179);
```

Note the semicolon which prevents the PRINT statement from sending an unwanted carriage return and line feed. (In most printing involving graphics, you will want to use the semicolon.)

You may insert a graphics character into a program line by holding down the ALT key and entering the character number on the numeric keypad (the calculator-like numbers on the right side of the keyboard). This has the advantage that in a PRINT statement, you can see the character to be printed. For example, the above statement line would appear on the screen as

```
30 PRINT │;
```

where the symbol │ is entered from the keyboard by holding down ALT and typing 179. In what follows, we will use the CHR$ notation to make clear the code numbers of the various characters. However, in you own work, you should use the ALT key to indicate graphics characters.

TEST YOUR UNDERSTANDING 1 (answer on page 237)
Write a set of instructions to print graphics character 179 in row 18, column 22.

TEST YOUR UNDERSTANDING 2 (answer on page 237)
Write a program to display all 128 graphics characters on the screen.

We may use the graphics characters to build up various images on the screen, as the next example shows.

Example 2. Write a program that draws a horizontal line across row 10 of the screen. (Assume that you have a 40-column screen.)
Solution. Just in case the screen contains some unrelated characters, begin by clearing the screen using the **CLS** instruction. Then print character 196 (a horizontal line) across row 10 of the screen. Here is the program:

Figure 10-3. PCjr graphics and special characters.

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
128	Ç	166	ª	204	╠	242	≥
129	ü	167	º	205	═	243	≤
130	é	168	¿	206	╬	244	⌠
131	â	169	⌐	207	╧	245	⌡
132	ä	170	¬	208	╨	246	÷
133	à	171	½	209	╤	247	≈
134	å	172	¼	210	╥	248	°
135	ç	173	¡	211	╙	249	∙
136	ê	174	«	212	╘	250	·
137	ë	175	»	213	╒	251	√
138	è	176	▒	214	╓	252	n
139	ï	177	▒	215	╫	253	²
140	î	178	▓	216	╪	254	■
141	ì	179	│	217	┘	255	(blank 'FF')
142	Ä	180	┤	218	┌		
143	Å	181	╡	219	▉		
144	É	182	╢	220	▄		
145	æ	183	╖	221	▌		
146	Æ	184	╕	222	▐		
147	ô	185	╣	223	▀		
148	ö	186	║	224	α		
149	ò	187	╗	225	β		
150	û	188	╝	226	Γ		
151	ù	189	╜	227	π		
152	ÿ	190	╛	228	Σ		
153	Ö	191	┐	229	σ		
154	Ü	192	└	230	μ		
155	¢	193	┴	231	τ		
156	£	194	┬	232	Φ		
157	¥	195	├	233	θ		
158	Pt	196	─	234	Ω		
159	ƒ	197	┼	235	δ		
160	á	198	╞	236	∞		
161	í	199	╟	237	Ø		
162	ó	200	╚	238	∈		
163	ú	201	╔	239	∩		
164	ñ	202	╩	240	≡		
165	Ñ	203	╦	241	±		

```
10 CLS
20 LOCATE 10,1
30 FOR J=1 TO 40
40   PRINT CHR$(196);
50 NEXT J
60 END
```

Note that the semicolon in the **PRINT** statement causes the characters to be printed in consecutive positions. Lines 30-60 may be abbreviated using the STRING$ function. The function value STRING$(40,196) equals a string consisting of 40 copies of character 196. So lines 30-60 could be written more simply:

```
30 PRINT STRING$(40,196);
```

Example 3. Write a program that draws a vertical line in column 25 from row 5 to row 15. The program should blink the line 50 times.

Solution. The blinking effect may be achieved by repeatedly clearing the screen. Here is the program:

```
10 CLS
20 FOR K=1 TO 50:'K CONTROLS BLINKING
30   FOR J=5 TO 15
40        LOCATE J,25
50        PRINT CHR$(179);
60   NEXT J
70   CLS
80 NEXT K
```

TEST YOUR UNDERSTANDING 3 (answer on page 237)
Write a program to draw a vertical line from row 2 to row 20 in column 8.

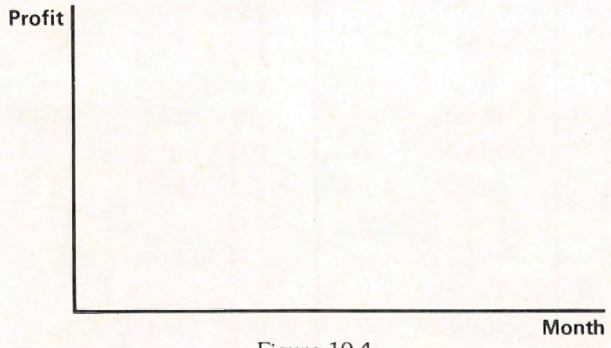


Figure 10-4.

Example 4. Draw a pair of x- and y-axes as shown in Figure 10-4. Label the vertical axis with the word "Profit" and the horizontal axis with the word "Month." (Assume that you have a 40-column screen.)

Solution. The program must draw two lines and print two words. The only real problem is to determine the positioning. The word "Profit" has six letters.

Let's start the vertical line in the position corresponding to the seventh character column. We'll run the vertical line from the top of the screen (row 1) to within two character rows from the bottom. (On the next-to-last row, we will place the word "month." We won't print in the last row, as this will cause some of the material printed above to scroll off the screen!) The layout of the screen is shown in Figure 10-5. Here is our program to generate the display.

```
10  WIDTH 40:CLS
20  LOCATE 1,1
25  PRINT "Profit"
30  LOCATE 23,35
35  PRINT "Month"
40  FOR J=1 TO 23
50   LOCATE J,7: PRINT CHR$(179);
60  NEXT J
65  LOCATE 22,7: PRINT CHR$(192);
70  FOR J=8 TO 40
80   LOCATE 22,J: PRINT CHR$(196);
90  NEXT J
100 GOTO 100
```

Note the infinite loop in line 100. This loop will keep the display on the screen indefinitely while the computer spins its wheels. To stop the program, press **Fn-Break**. To see the reason for the infinite loop, try running the program after deleting line 100. Note how the **Ok** interferes with the graphics. The infinite loop prevents the BASIC prompt from appearing on the screen.

Exercises (answers on page 369)

Draw the following straight lines. (Assume an 80-column screen width.)

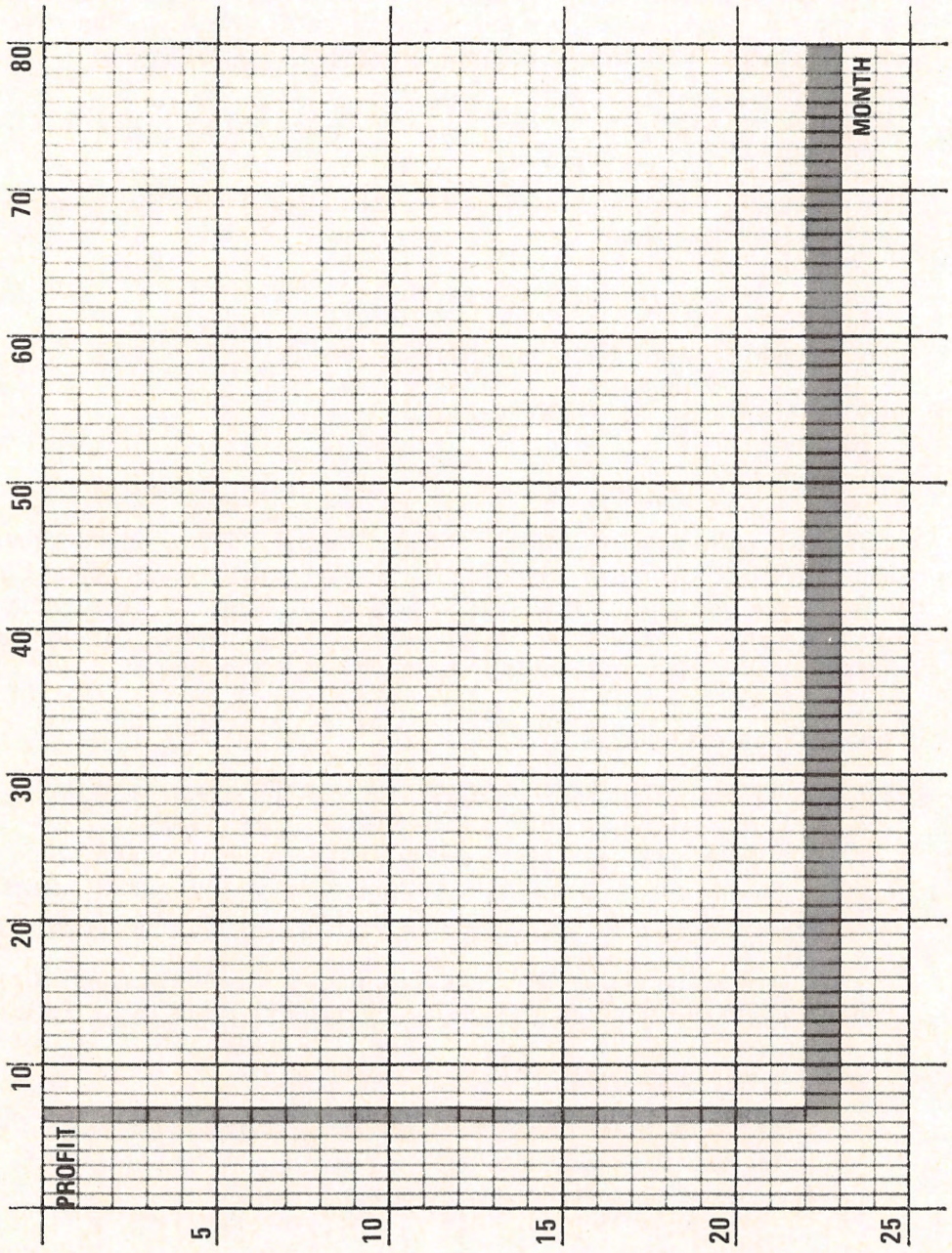
1. A horizontal line completely across the screen in row 18.
2. A vertical line completely up and down the screen in column 17.
3. A pair of straight lines that divide the screen into four equal squares.
4. Horizontal and vertical lines that convert the screen into a tic-tac-toe board.
5. A vertical line of double thickness from rows 1 to 24 in column 30.
6. A diagonal line going through the character positions (1,1), (2,2), ... (24,24).
7. A horizontal line with "tick marks" as follows:

(Hint: Look for a graphics character that will form the tick marks.)

8. A vertical line with tick marks as follows:

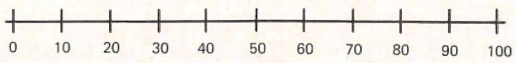
9. Display your name in a box formed with asterisks:

Figure 10-5. Display layout for chart of Figure 10-4.

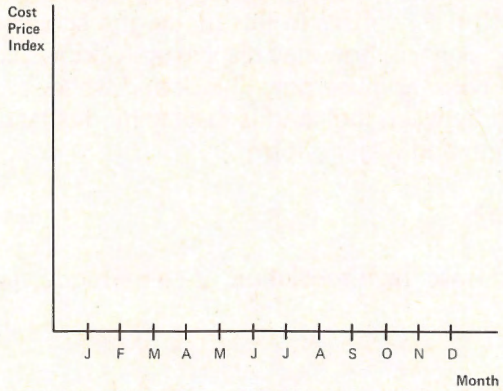


```
*****************
*   Your Name   *
*****************
```

10. Display a number axis as follows:



11. Write a program to display a graphics character that you specify in an **INPUT** statement.
12. Create a display of the following form:



ANSWERS TO TEST YOUR UNDERSTANDING

```
1: 10 LOCATE 18,22
   20 PRINT CHR$(179)
2: 10 FOR J=128 TO 255
   20   PRINT CHR$(J); " "; :' ONE SPACE BETWEEN CHARS
   30 NEXT J
   40 END
3: 10 CLS
   20 FOR J=2 TO 20
   30   LOCATE J,8: PRINT CHR$(179);
   40 NEXT J
   50 END
```

10.2 Colors and Graphics Modes

There are 7 screen modes on the PCjr, as listed below:

Screen Mode	Description
0	text
1	medium-resolution, 4 colors
2	high-resolution, 2 colors
3	low-resolution, 16 colors

4	medium-resolution, 4 colors
5	medium-resolution, 16 colors
6	high-resolution, 4 colors

So far, we have discussed only text mode. For the rest of this chapter, let's focus on the six graphics modes.

Low-Resolution Graphics In this mode, the screen is divided into 200 rows of 160 rectangles each. You may display up to 16 colors simultaneously.

Medium-Resolution Graphics In these modes, the screen is divided into 200 rows of 320 rectangles each. You may display up to 16 or 4 colors, depending on the mode.

High-Resolution Graphics In this mode, the screen is divided into 200 rows of 640 rectangles each. You may display up to four colors simultaneously, or you may disable color and use only "black and white."

You may select between the various display modes by using the **SCREEN** command. This command has the form:

```
SCREEN <mode>
```

For example, to choose high-resolution, 2-color mode, you would give the command

```
SCREEN 2
```

You may use the SCREEN command to switch from one display mode to another, either within a program or by using a keyboard command. Note, however, that the SCREEN command automatically clears the screen. When BASIC is started, the display is automatically in text mode. (SCREEN 0)

Graphics modes 5 and 6 are available only in Cartridge BASIC and only in a PCjr with 128k of RAM. Graphics modes 5 and 6 require you to reserve graphics memory using the CLEAR statement:

```
CLEAR ,,,32768
```

This statement should precede the SCREEN statement. If you attempt to enter graphics mode 5 or 6 without reserving graphics memory, BASIC will report an Illegal Function Call.

Pixels

Each of the small screen rectangles (more properly, dots) is called a **pixel** (="picture element"). You may color each pixel on an individual basis.

Graphics Coordinates Each pixel is specified by a pair of coordinates (x,y), where x is the column number and y is the row number. Note the following important facts:

1. Rows and columns are numbered beginning with 0 (not 1, as in text mode). In low-resolution graphics mode, the rows are numbered from 0 to 199

and the columns are numbered from 0 to 159. In the medium-resolution graphics mode, the rows are numbered from 0 to 199 and the columns from 0 to 319. In high-resolution graphics mode, the rows are numbered from 0 to 199 and the columns from 0 to 639.

2. Coordinates in graphics mode are specified with the column (x-coordinate) first. This is the opposite of the coordinates in text mode. (For example, the LOCATE statement requires the row first.)

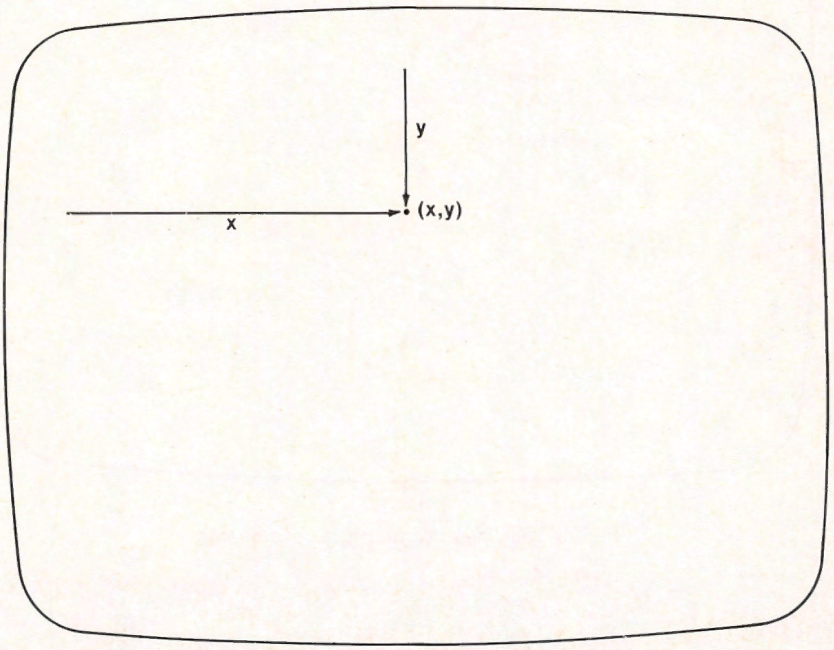


Figure 10-6. Coordinates in graphics mode.

Relative Coordinates in Graphics Mode In graphics mode, the cursor is not visible. Instead, the computer keeps track of the **last point referenced**. This is the point whose coordinates were most recently used in a graphics statement. You may specify the position of new points by giving coordinates relative to the last point referenced. Such coordinates are called **relative coordinates**. Relative coordinates always are preceded by the word STEP. For example, suppose that the last point referenced is (100,75). Then here is a point specified by relative coordinates

```
STEP (20,30)
```

This is the point that is 20 units to the right and 30 units up from the last referenced point. This is the point with coordinates (120,105). Similarly, consider the point specified by the relative coordinates:

```
STEP (-10,-40)
```

This is the point that is 10 units to the left and 40 units down from the point (120,105); that is, the point (90,35).

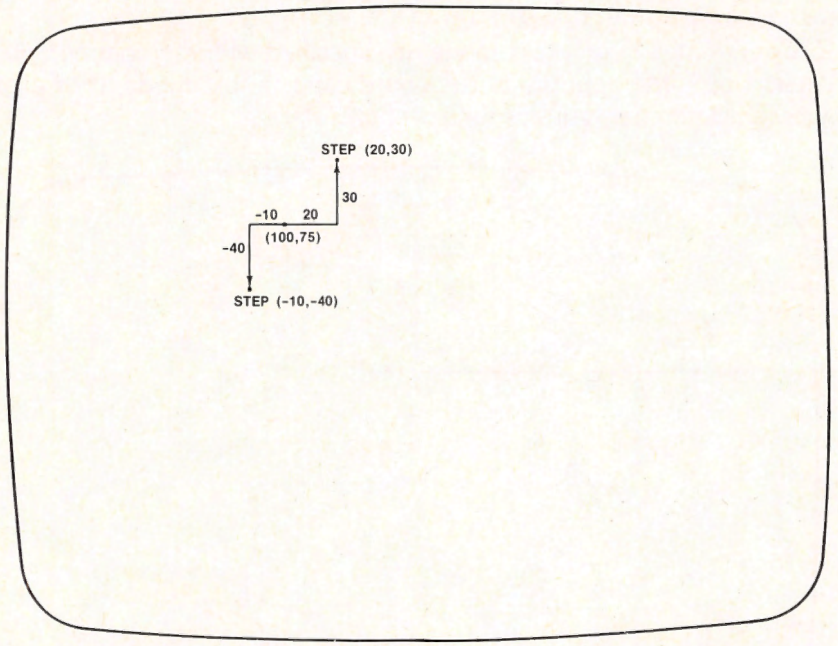


Figure 10-7. Relative graphics coordinates.

TEST YOUR UNDERSTANDING 1 (answer on page 243)
Suppose that the last referenced point is (50,80). Determine the coordinates of the following points:

a. STEP (50,50)
b. STEP (-20,10)
c. STEP (10,-40)
d. STEP (-20,-50)

Colors

To use color in your display, you must first enable color with the SCREEN statement:

```
SCREEN 1,0 (means low-resolution graphics, color ON)
```

You may disable color with the statement:

```
SCREEN 1,1 (means low-resolution graphics, color OFF)
```

Once color has been enabled, you may choose both background and foreground colors. A pixel is considered part of the **background** (at a particular

moment) unless its color has been explicitly set by a graphics statement. When you execute CLS, all pixels are set equal to the background color. A non-background pixel is said to belong to the **foreground**.

Here are the possible screen colors, numbered 0-15.

0	black	8	gray
1	blue	9	light blue
2	green	10	light green
3	cyan	11	light cyan
4	red	12	light red
5	magenta	13	light magenta
6	brown	14	yellow
7	white	15	high intensity white

Attributes and Palettes

An **attribute** is a number used to describe a color on the screen. For example, you might draw a picture and specify that a certain part has attribute 1, a second section has attribute 2, and a third section has attribute 3. You then may specify that attribute 1 corresponds to color 3, attribute 2 to color 4, and attribute 3 to color 13. If you don't like the coloring of the picture, you may reassign the colors corresponding to the attributes.

The number of different possible screen attributes depends on the screen mode. For example, in screen mode 6, there are four colors, so there are four attributes, numbered 0 through 3. Similarly, in screen mode 5, there are 16 colors, so there are 16 attributes, numbered 0 through 15.

At any given moment, the assignment of colors to attributes is governed by a table, called a **palette**, which is maintained by BASIC. You may change the entries in the palette using the PALETTE statement. For example, to set attribute 4 equal to color 2, use the statement

```
PALETTE 4,2
```

If you don't specify a palette, then BASIC will assume that each attribute is assigned the color with the corresponding number.

BASIC provides a more convenient method for changing many entries of a palette in one step, using the PALETTE USING statement. To use this statement, you first create an array containing 16 entries, say A(0),...,A(15). Set A(0) equal to the color assigned to attribute 0, A(1) equal to the color assigned to attribute 1, and so forth. If you wish to leave the color assigned to an attribute unchanged, set the corresponding array entry equal to -1. After you set the array entries equal to the desired colors, use the statement

```
PALETTE USING A(0)
```

The 0 in the statement tells BASIC the entry at which to start reading the array. BASIC will read as many attributes as required by the current screen mode starting with the specified entry in the array.

Choosing Colors

Background and foreground colors are set using the **COLOR** statement:

```
100 COLOR 12,0
```

This statement sets the background color as black (color 0) and the attribute of the foreground color as 0. These choices remain in effect until they are changed with another **COLOR** statement. Note that the background is specified as an actual color and not an attribute. On the other hand, the foreground is specified in terms of an attribute.

Also note that the above description of the color statement works in all screen modes except screen 1 (medium resolution, 4-color mode). To maintain compatability with earlier versions of BASIC, IBM allows the COLOR statement in this case to work as in BASIC 2.00. However, since medium-resolution 4-color mode is available as mode 4, we can safely ignore screen mode 1.

TEST YOUR UNDERSTANDING 2 (answer on page 243)
Write BASIC statements that select the medium-resolution graphics 16-color mode, set the background color to high intensity white, and the foreground to attribute 1.

Illuminating Pixels The **PSET** statement is used to illuminate a pixel. For example, the statement:

```
200 PSET (100,150),1
```

will illuminate the pixel at (100,150) in attribute 1 of the currently chosen palette. Similarly, to turn off this pixel, use the statement:

```
300 PRESET (100,150)
```

Actually, this last instruction turns on pixel (100,150) in the background color. This is equivalent to turning it off. In using the **PSET** and **PRESET** statements, you may specify the pixel in **relative form**. For example, the statement

```
400 PSET STEP (100,-150), 2
```

will turn on the pixel that is 100 blocks to the right and 150 blocks up from the current cursor position, using attribute 2.

Exercises (answers on page 371)

Write BASIC instructions that:

1. Select the background color magenta and the foreground color as attribute 5.

2. Select the background color light red and the foreground color as attribute 7.
3. Turn on pixel (200,80) with attribute 1 of the current palette.
4. Turn on pixel (100,100) in red with background color cyan.
5. Set the pixel that is 200 blocks to the left and 100 blocks above the last referenced point. Use attribute 3.
6. Turn on the pixel that is 100 units to the right of the last referenced point.
7. Set the current palette so that the attributes are exactly reversed. That is, assign attribute 0 to color 15, attribute 1 to color 14, and so forth.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. `(100,130)` b. `(30,90)` c. `(60,40)`
d. `(30,30)`

2:
```
10 CLEAR ,,,32768:SCREEN 5:COLOR 1,15
```

10.3 Lines, Rectangles, and Circles

Straight Lines

You may use the **PSET** and **PRESET** statements to design color graphics displays. However, BASIC has a rich repertoire of instructions that greatly sim-

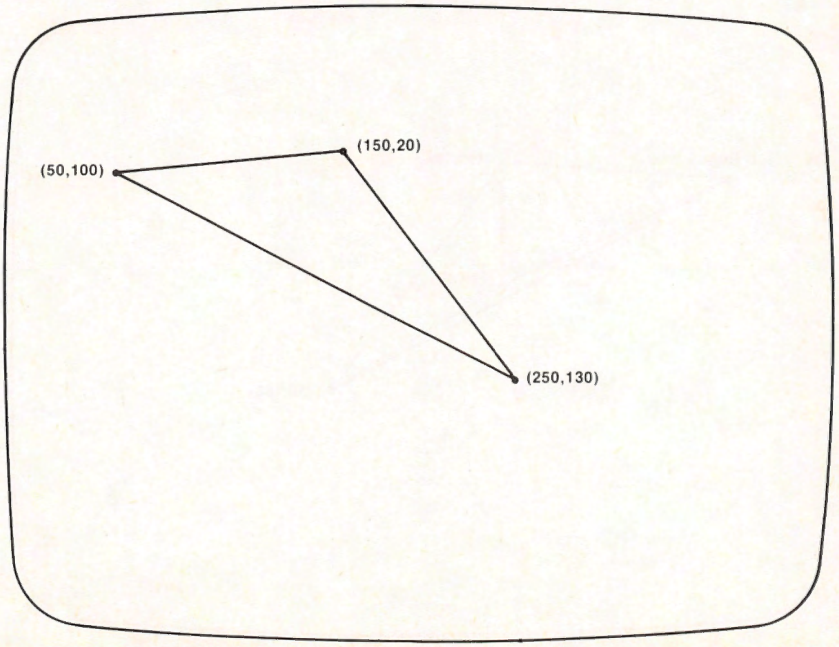


Figure 10-8. A triangle.

plify the task. For the task of drawing straight lines, you may use the **LINE** statement. For example, to draw a line connecting the pixels (20,50) and (80,199), we use the statement:

```
10 LINE (20,50)-(80,199)
```

Example 1. Draw a triangle in medium-resolution mode with corners at the three points (150,20), (50,100), and (250,130). (See Figure 10-8.)
Solution. We must draw three lines: From (150,20) to (50,100); from (50,100) to (250,130); and from (250,130) to (150,20). Here is the program:

```
10 SCREEN 2
20 LINE (150,20)-(50,100)
30 LINE (50,100)-(250,130)
40 LINE (250,130)-(150,20)
50 END
```

To draw a line from the last referenced point to (100,90), use the statement:

```
20 LINE -(100,90)
```

To draw a line from the last referenced point to the point 80 units to the right and 100 units above, use the statement

```
30 LINE -STEP(80,-100)
```

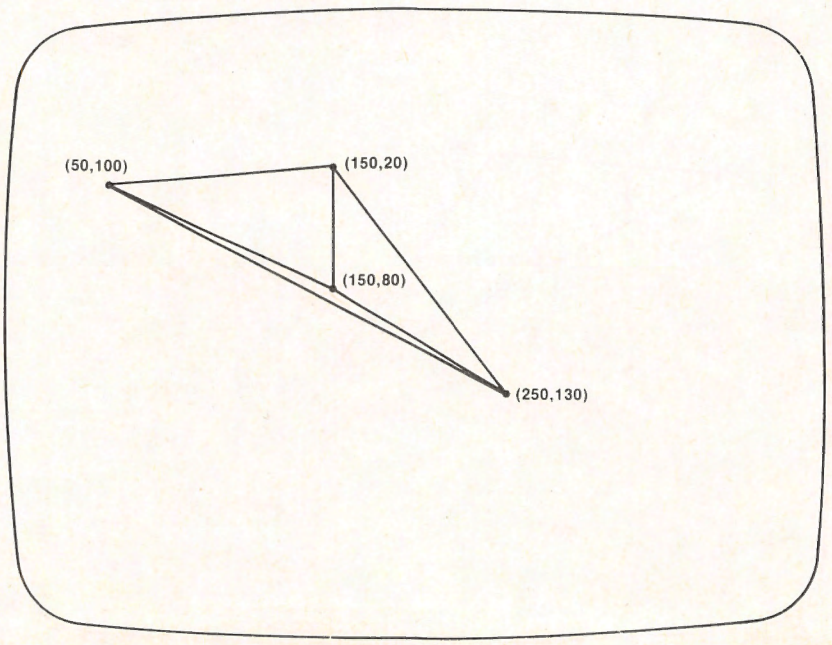


Figure 10-9. More triangles.

Example 2. Let's reconsider the triangle of Example 1. The point (150,80) is inside the triangle. Draw lines connecting this point to each of the corners of the triangle. (See Figure 10-9.)

Solution. The point (150,80) needs to go with three line statements. So we use the shorthand form to draw lines from this point to the three corners of the triangle. To make (150,80) the last referenced point, we first PSET it.

```
10 SCREEN 2
20 LINE (150,20)-(50,100)
30 LINE -(250,130)
40 LINE -(150,20)
50 PSET (150,80) LINE -(150,20)
60 PSET (150,80): LINE -(50,100)
70 PSET (150,80): LINE -(250,130)
80 END
```

You also may specify the color of a line. For example, if you wish to draw the line in statement 10 in attribute 1 of the current palette, use the statement:

```
40 LINE (20,50)-(80,199),1
```

This line is drawn in Figure 10-10.

Note that there are lines the computer cannot draw perfectly. Lines on a diagonal are displayed as a series of visible "steps." This is as close as the computer can get to a straight line within the limited resolution provided by the

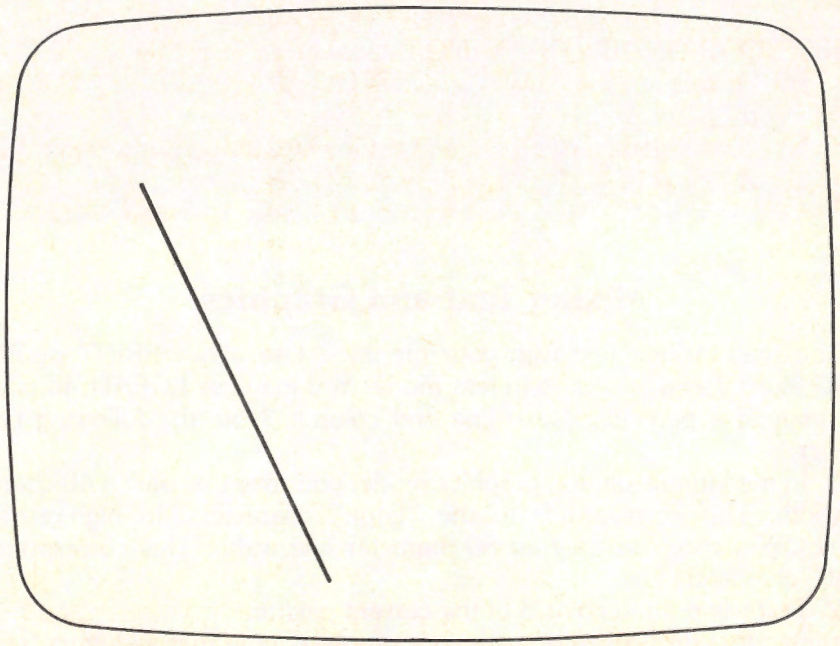


Figure 10-10.

graphics modes. The higher the resolution (that is, the more pixels on the screen), the better your straight lines will look.

TEST YOUR UNDERSTANDING 1 (answer on page 254)

a. Draw a line connecting (0,100) to (50,75) in attribute 2.
b. Draw the triangle with vertices (0,0), (50,50), and (100,30).

Rectangles

The **LINE** statement has several very sophisticated variations. To draw a rectangle you need to specify a pair of opposite corners in a **LINE** statement and add the code B (for BOX) at the end of the statement. For example, to draw a rectangle, two of whose corners are at (50,100) and (90,175), use the statement:

```
50 LINE (50,100)-(90,175),1,B
```

This statement will draw the desired rectangle with the sides in color 1 of the current palette (see Figure 10-11(a)). The inside of the rectangle will be in the background color. You may paint the inside of the rectangle in the same color as the sides by changing the B to BF (B = Box, BF = Box Filled). (See Figure 10.11(b).) These instructions greatly simplify drawing complex line displays.

TEST YOUR UNDERSTANDING 2 (answer on page 254)

a. Draw a rectangle with corners at (10,10), (10,100), (50,100), and (50,10).
b. Draw the rectangle of a. and color it and its interior with attribute 2.

Mixing Text and Graphics

You may include text with your graphics. Use either PRINT or PRINT USING exactly as if you were in text mode. You may use LOCATE to position the cursor at a particular (text) line and column. Note the following points, however:

1. In medium-resolution graphics mode, you may use only a 40-character line width. This corresponds to the "large" characters. In high-resolution graphics, you may use only an 80-character line width. This corresponds to "small" characters.

2. Text will print in color 3 of the current palette.

In planning text displays to go with your graphics, note that all letters (regardless of line width) are 8 pixels wide and 8 pixels high. Thus, for example, the

Figure 10-11(a). The B Option.

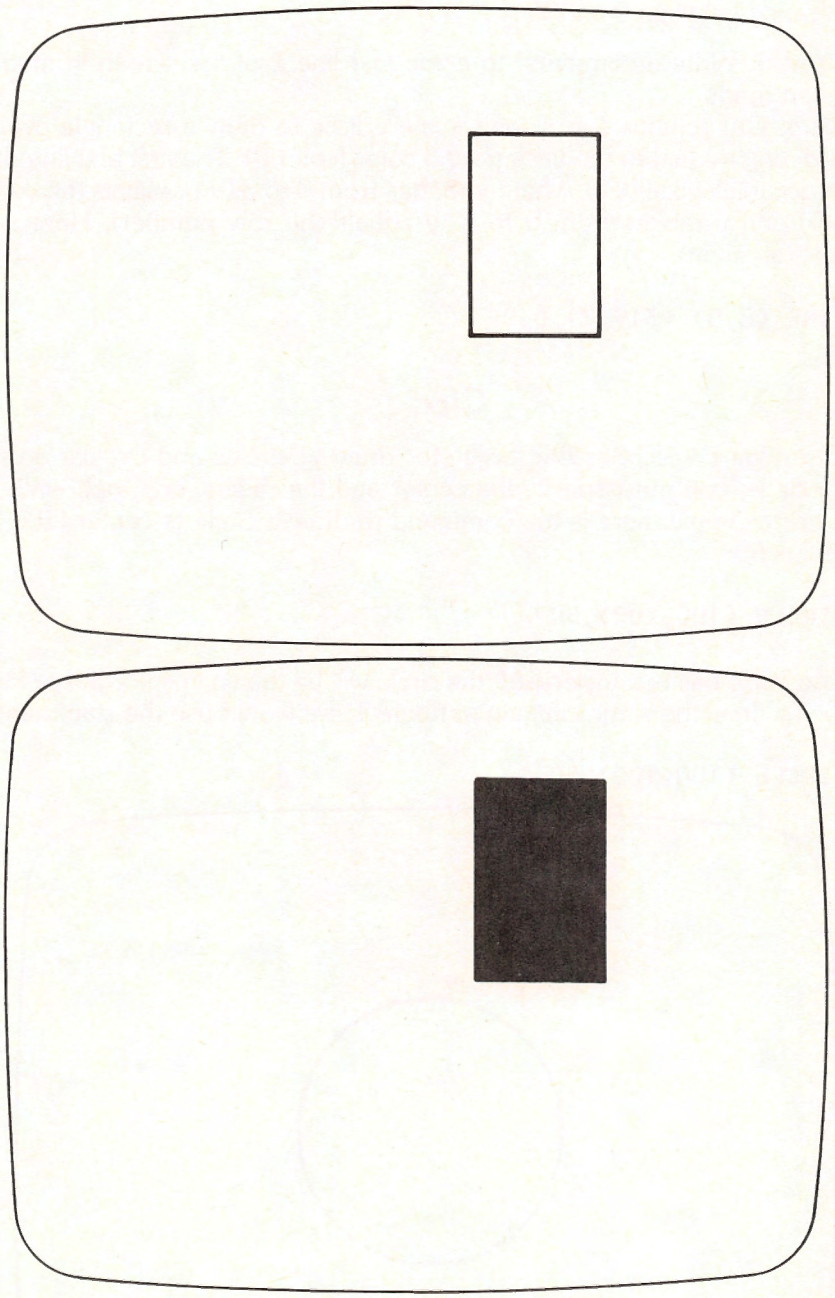


Figure 10-11(b). The BF Option.

character at the top left corner of the screen occupies pixels (x,y), where x and y both range between 0 and 7.

Example 3. Write a command to erase text line 1 of the screen in medium-resolution mode.

Solution. Our scheme for erasing a line will be to draw a rectangle over the line, with color equal to the background color (color 0). The first text line of the screen occupies pixel (x,y), where x ranges from 0 to 319 (x equals the column number) and y ranges from 0 to 7 (y equals the row number). Here is the desired statement:

```
LINE (0,0)-(319,7),0,BF
```

Circles

Cartridge BASIC has the facility for drawing circles and circular arcs. To draw a circle, you must specify the center and the radius, and, optionally, the color. For example, here is the command to draw a circle at center (100,100) and radius 50:

```
CIRCLE (100,100),50
```

Since no color has been specified, the circle will be drawn in color 3 (see Figure 10-12). To draw the same circle in attribute 1, we would use the statement:

```
CIRCLE (100,100),50,1
```

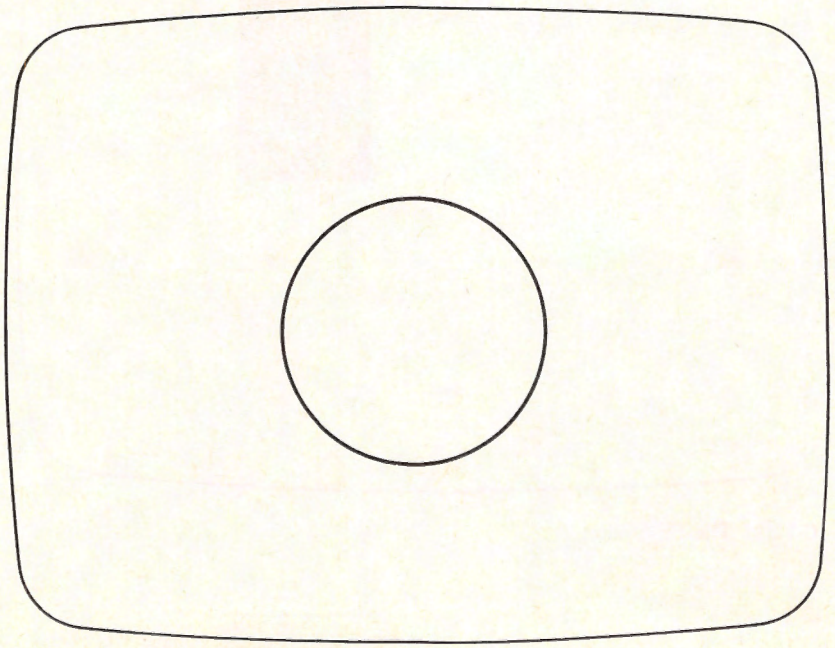


Figure 10-12.

Note that the circles on the screen are not smooth, but have a "ragged" appearance. This is due to the limited resolution of the screen. If you use high-resolution mode, you will notice that the appearance of your circles improves greatly.

Circular arcs are somewhat more complicated to draw since their description is based on the radian system of angle measurement. Let's take a few moments to describe radian measurement.

Recall the number pi from high school geometry. Pi is a number, denoted by the Greek letter π, that is approximately equal to 3.1415926... (the decimal expansion goes on forever). Ordinarily, angles are measured in degrees, with 360 degrees equaling one complete revolution. In radian measurement, there are 2*pi radians in a revolution. That is:

2*pi radians = 360 degrees

1 radian = 360/(2*pi) degrees

If you use the value of pi and carry out the arithmetic, you find that 1 radian is approximately 57 degrees. When describing angles to the computer, you must always use radians.

To draw a circular arc, you use the following variation of the CIRCLE statement:

```
CIRCLE (xcenter,ycenter),radius,color,startangle, endangle
```

where startangle and endangle are measured in radians. For example, to draw a circular arc for the above circle, corresponding to an angle of 1.4 radians, beginning at angle .1 radians, we may use the command

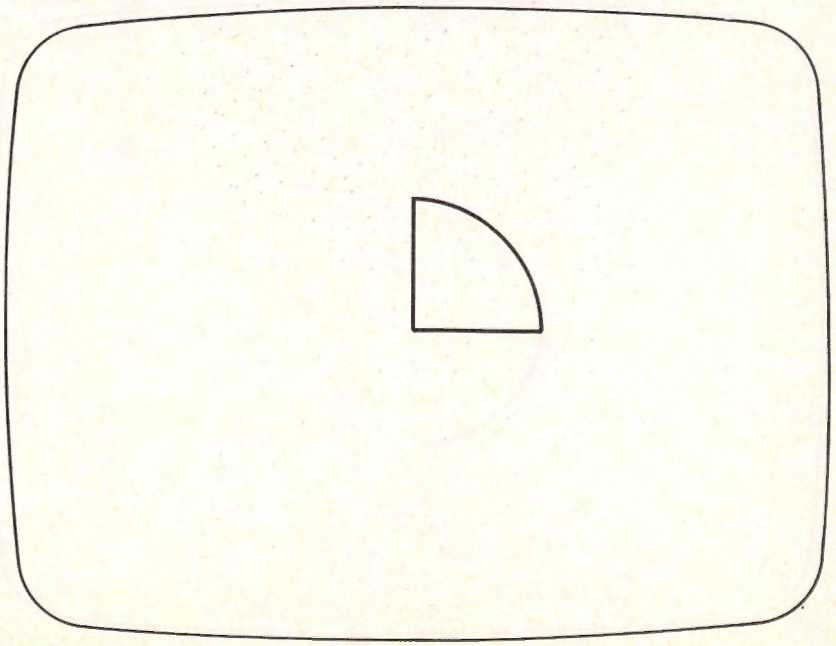


Figure 10-13A.

```
CIRCLE (100,100),50,1,.1,1.5
```

The resulting angle is pictured in Figure 10-13A.

Note that Figure 10-13A does not include the sides of the sector. To include a side on a circular arc, put a minus sign on the corresponding angle. (We can't use -0, however. See below.) For example, to include both sides in Figure 10-13A, we may use the statement:

```
CIRCLE (100,100),50,1,-.1,-1.5
```

The resulting arc will look like the one in Figure 10-13B.

TEST YOUR UNDERSTANDING 3 (answer on page 254)

a. Draw a circular arc with radius 60, center (125,75) and going from a starting angle of .25 radians to and ending angle of .75 radians.

b. Draw the same circular arc as in a., but with sides included.

If you have an angle 0 and wish to include a side, just note that the angle 0 and the angle 2*pi are the same. So just replace 0 by 2*pi = 6.28... , and put a minus sign on this new angle!

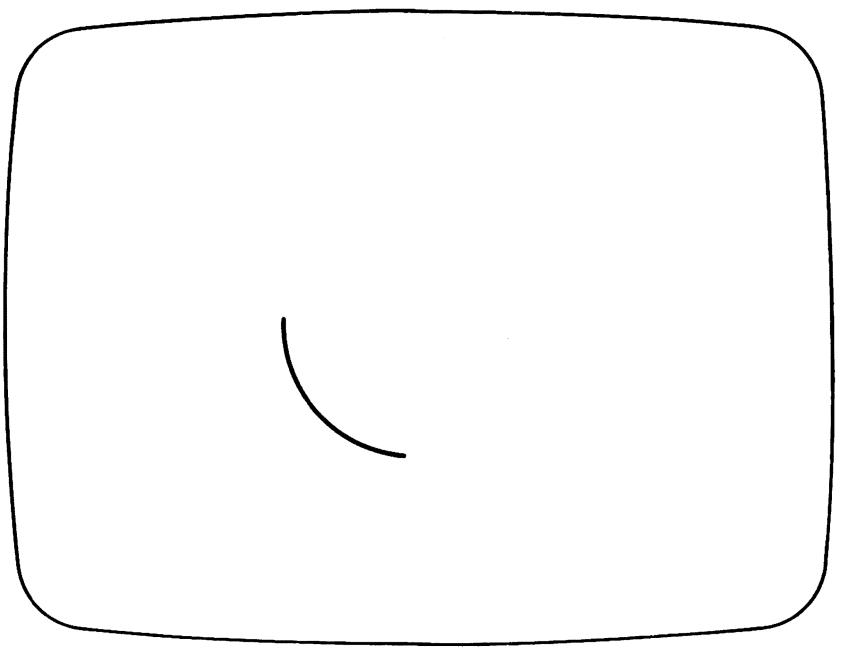


Figure 10-13B.

Aspect Ratio The CIRCLE statement has an added complication we haven't yet mentioned, namely the aspect ratio. Usually, when you plot circles on graph paper, you use the same scale on the x-axis as on the y-axis. If, for example, a unit on the x-axis is larger than a unit on the y-axis, your circle will appear as an ellipse, stretched out in the x-direction. Similarly, if the unit on the y-axis is larger than the unit on the x-axis, the circle will appear as an ellipse stretched out in the y-direction. So, like it or not, the geometry of circles is intimately bound up with that of ellipses. For this reason, the CIRCLE statement may also be used to draw ellipses.

Consider the following example in high-resolution graphics mode:

```
CIRCLE (300,100),100,,,,.5
```

This statement plots an ellipse with center (300,100). The extra commas are placeholders for the unspecified attribute, beginning angle and ending angle. The x-radius is 100. The number .5 is called the **aspect ratio**. It tells us that the y-radius is .5 times the x-radius, or 50.

Similarly, consider the statement

```
CIRCLE (300,100),100,,,,1.5
```

Here the aspect ratio is 1.5, which is larger than 1. In this case, BASIC assumes that the radius 100 is the y-radius. The x-radius is 1.5 times the y-radius, or 45. The corresponding ellipse is shown in Figure 10-15.

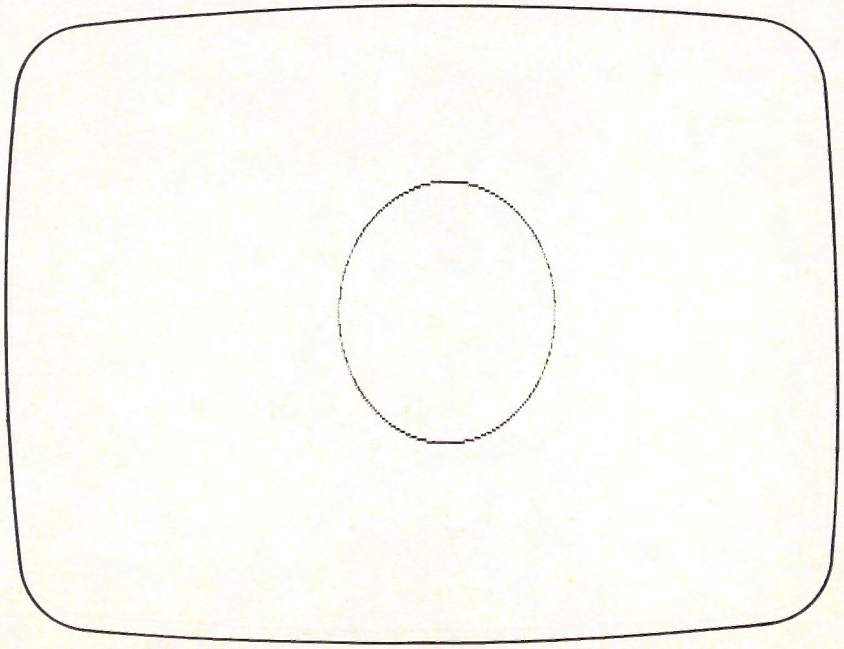


Figure 10-14. The Ellipse CIRCLE (300,100),100,,,,.5.

What is the aspect ratio for a circle? Well, that's a tricky question. On first glimpse, you probably would guess that the aspect ratio is 1. And indeed it is if you are looking for a mathematical circle. However, if you draw a circle with an aspect ratio of 1, you will get an ellipse. The reason is that the scales on the x- and y-axes are different. Let's consider high-resolution graphics mode. The screen is 640 x 200 pixels. The ratio of width to height is 200/640, or 5/16. So to achieve a circle, you would expect to have to multiply the x-radius by 5/16 to get the proper y-radius; that is, an aspect ratio of 5/16. Well, not quite! TV screens are not square. The ratio of width to height is 4/3. So in order to achieve an ellipse that is visually a circle, we must multiply by 5/16 **and** by 4/3. In other words, the aspect ratio is:

(5/16) * (4/3) = 5/12

Strange, but true. In medium-resolution mode, the aspect ratio giving a visual circle is 5/6. If you use the CIRCLE statement without any aspect ratio, then BASIC assumes an aspect ratio of 5/6 in medium-resolution mode and 5/12 in high-resolution mode. With these aspect ratios, circles look like circles. However, the y-radius is quite different from the x-radius!

You can get even finer grained control over circles and ellipses if you apply some mathematics. Suppose that an ellipse (or circle) has its center at the point with coordinates (x0,y0). Suppose that the horizontal half-axis has length A and the vertical half-axis has length B. Then a typical point (x,y) on the ellipse takes the form:

x = x0 + A*cos(t)
y = y0 + B*sin(t)

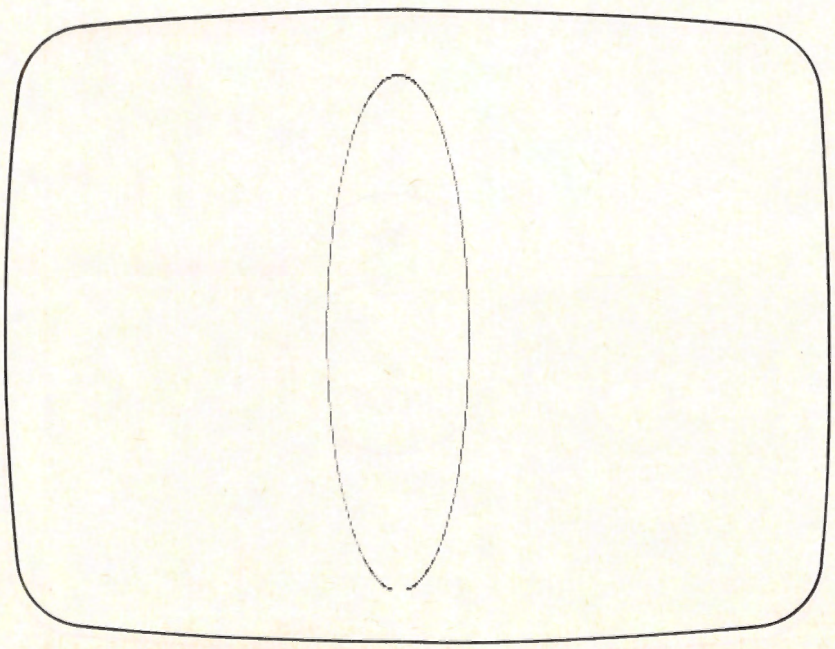


Figure 10-15. The Ellipse CIRCLE (300,100),100,,,,1.5.

where t is an angle between 0 and 2*pi radians. The geometric meaning of the angle t is shown in Figure 10-16. The above equations are called the **parametric equations for the ellipse**. They are very useful in drawing graphics.

For example, here is a program that draws an ellipse with center (320,100) (the center of the screen in high-resolution mode) by plotting dots in a "sweep" fashion (see Figure 10-17). This graph may be used to simulate the motion of a planet around the sun.

```
5 'planetary orbit
10 SCREEN 2:CLS:KEY OFF
20 FOR T=0 TO 6.28 STEP .05
30     X=320+200*COS(T):Y=100+30*SIN(T)
40     PSET (X,Y)
50     FOR K=1 TO 25:NEXT K
70 NEXT T
```

Note that line 50 provides a delay between plotting of consecutive dots.

Exercises (answers on page 371)

Write BASIC instructions to draw the following:

1. A line connecting (20,50) and (40,100).
2. A line in color 2 connecting the current cursor position and the point (250,150).
3. A line in color 1 connecting (125,50) to a block 100 blocks to the right and 75 units down from it.
4. A rectangle with corners at (10,20), (200,20), (200,150), and (10,150).
5. The rectangle of Exercise 4 with its sides and interior in color 3.
6. A circle with radius 20 and center (30,50).
7. A circular arc of the circle of Exercise 6 with a starting angle 1.5 and ending angle 3.1.
8. The circular arc of Exercise 7 with sides.
9. Write a program that simulates the movement of a sweep second hand around the face of a clock.

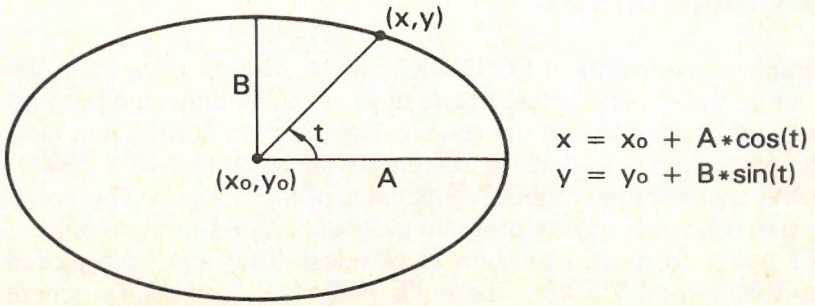


Figure 10-16. An Ellipse in Parametric Form.

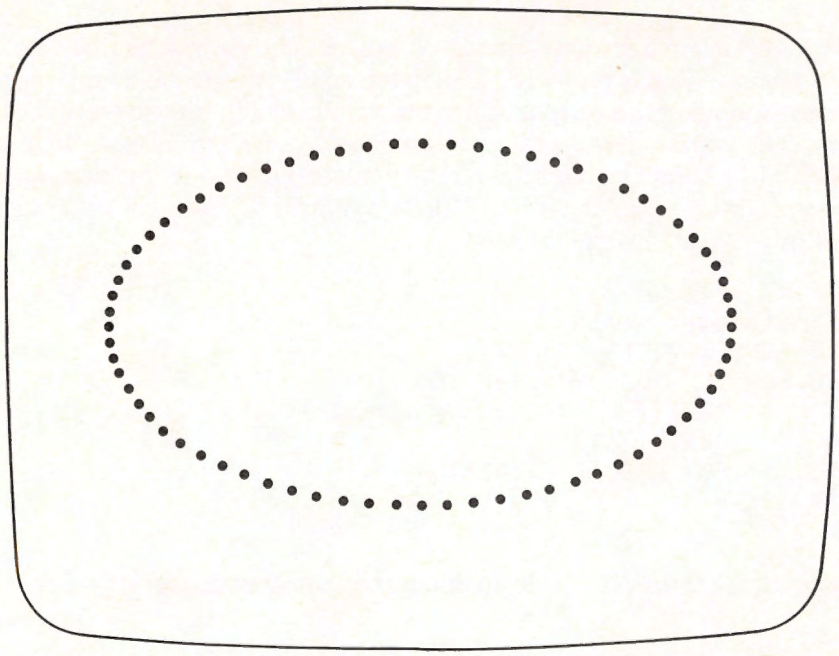


Figure 10-17. Simulating a Planetary Orbit.

ANSWERS TO TEST YOUR UNDERSTANDING

```
1:  a. 10 LINE (0,100)-(50,75),2
    b. 10 LINE (0,0)-(50,50)
       20 LINE (50,50)-(100,30)
       30 LINE (100,30)-(0,0)
2:  a. 10 LINE (10,10)-(50,100),,B
    b. 10 LINE (10,10)-(50,100),2,B
3:  a. 10 CIRCLE (125,75),60,,.25,.75
    b. 10 CIRCLE (125,75),60,,-.25,.75
```

10.4 Computer Art

The graphics statements of PC BASIC may be used to draw interesting computer art on the screen. As just a taste of what can be done, the program below draws random polygons on the screen. The program is written in high-resolution graphics mode, so that the screen has dimensions 640 x 200. The program first chooses the number N% of sides of the polygon. The polygon may have up to 6 sides. Next, the program picks out N%+1 random points (it takes N%+1 points to draw a polygon of N sides). The points are stored in the arrays X%(J%) and Y%(J%), where J% = 0, 1, 2, ..., N%. To generate only closed polygons, we define the point (X%(N%+1),Y%(N%+1)) to be the initial

point (X%(0), Y%(0)). The program then draws lines between consecutive points. Figure 10-18 shows a typical polygon.

The program then erases the polygon and repeats the entire procedure to draw a different polygon. The program draws 50 polygons.

```
10 'Computer art
20 SCREEN 2:CLS:KEY OFF
30 RANDOMIZE VAL(RIGHT$(TIME$,2))
40 FOR M%=1 TO 50
50      C%=1:GOSUB 90      'Draw random polygon
60      C%=0:GOSUB 190     'Erase polygon
70 NEXT M%
80 END
90 'Draw random polygon
100     'Determine number of sides
110     N%=INT(5*RND(1) + 1)    'N=# sides <= 6
120     'Compute coordinates of certices
130     FOR J%=0 TO N%
140          X%(J%)=INT(640*RND(1))
150          Y%(J%)=INT(200*RND(1))
160     NEXT J%
170     X%(N%+1)=X%(0):Y%(N%+1)=Y%(0)
180     'Draw sides
190     FOR J%=1 TO N%+1
200         LINE (X%(J%-1),Y%(J%-1))-(X%(J%),Y%(J%)),C%
210     NEXT J%
220 RETURN
```

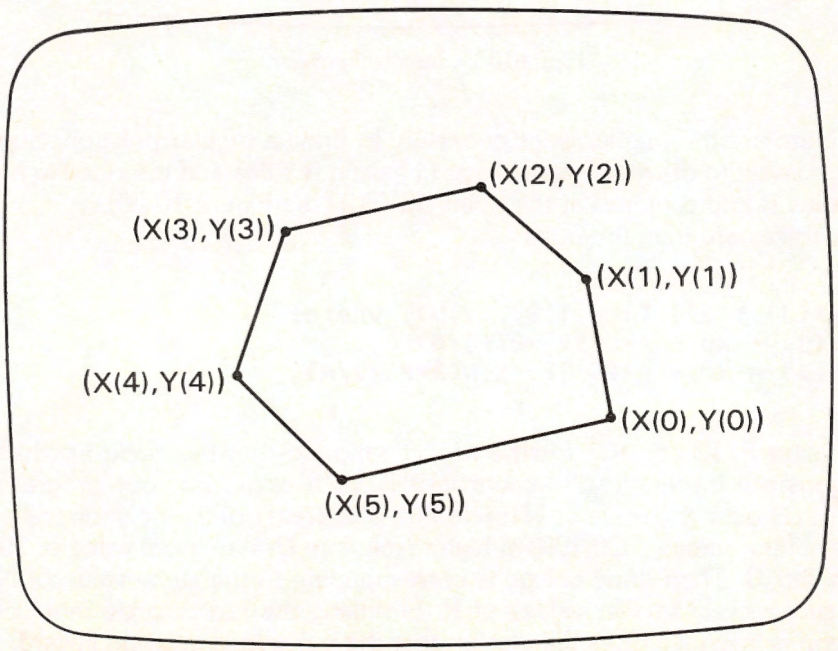


Figure 10-18. A Typical Polygon.

Here is a second program that draws a regular polygon (one with sides of equal length) and then draws inscribed replicas of the original polygon, each of smaller size, until the interior of the original polygon is filled with the inscribed replicas. (See Figure 10-19.)

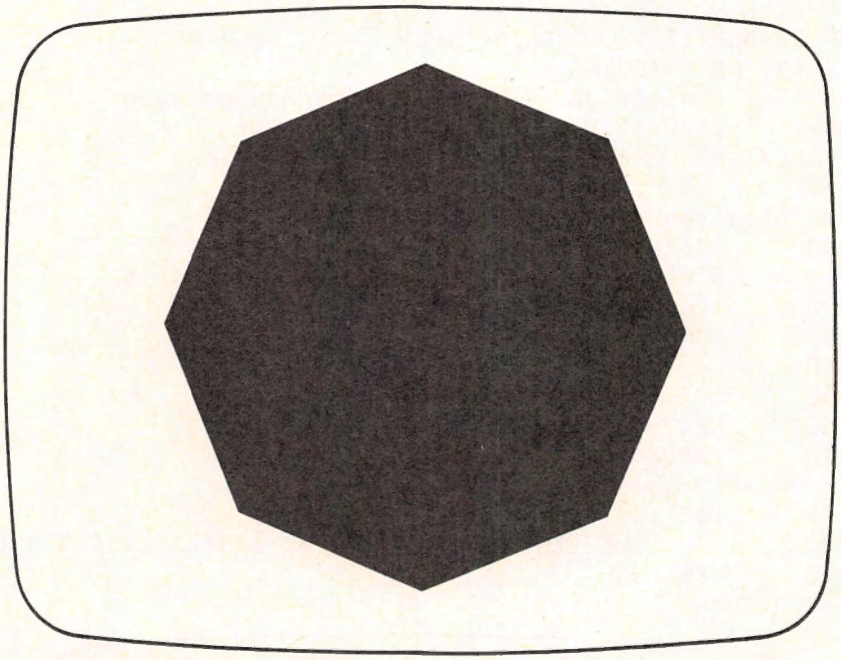


Figure 10-19. Inscribed polygons.

Here are the mathematics necessary to draw a regular polygon. Suppose that you wish to draw a regular polygon having N sides and inscribed in a circle of radius R and centered at the point (X0,Y0) (see Figure 10-20).
The vertices are then the points

```
(X(J),Y(J)) (J=0,1,2,...,N), where:
X(J) = X0 + R*COS(2*PI*J/N)
Y(J) = Y0 + R*(5/12)*SIN(2*PI*J/N)
```

The factor 5/12 corrects for the aspect ratio, so that the circle in which the polygons are inscribed will appear visually as a circle. For our program, the user will choose the value of N (up to 20). The center of the polygon will be the center of the screen (320,100) in high resolution. Use an initial value of 100 for the radius R. Then draw polygons corresponding to the same value of N, but with successively smaller values of R. Shrinking the radius circle in which the polygon is inscribed gives the illusion that the polygon is growing inward. Here is the program:

```
100 DIM X%(21),Y%(21)
110 INPUT "NUMBER OF SIDES";N%
120 IF N%>20 THEN 110
130 SCREEN 2:CLS:KEY OFF
140 PI=3.14159
150 FOR R%=100 TO 0 STEP -4
160       GOSUB 190
170 NEXT R%
180 END
185 'Calculate vertices
190     FOR J%=0 TO N%
200             X%(J%)=320+R%*COS(2*PI*J%/N%)
210             Y%(J%)=100+R%*(5/12)*SIN(2*PI*J%/N%)
220     NEXT J%
230     X%(N%+1)=X%(0):Y%(N%+1)=Y%(0)
235     'Draw polygon
240     FOR J%=0 TO N%
250             LINE (X%(J%),Y%(J%))-(X%(J%+1),Y%(J%+1))
260     NEXT J%
270 RETURN
```

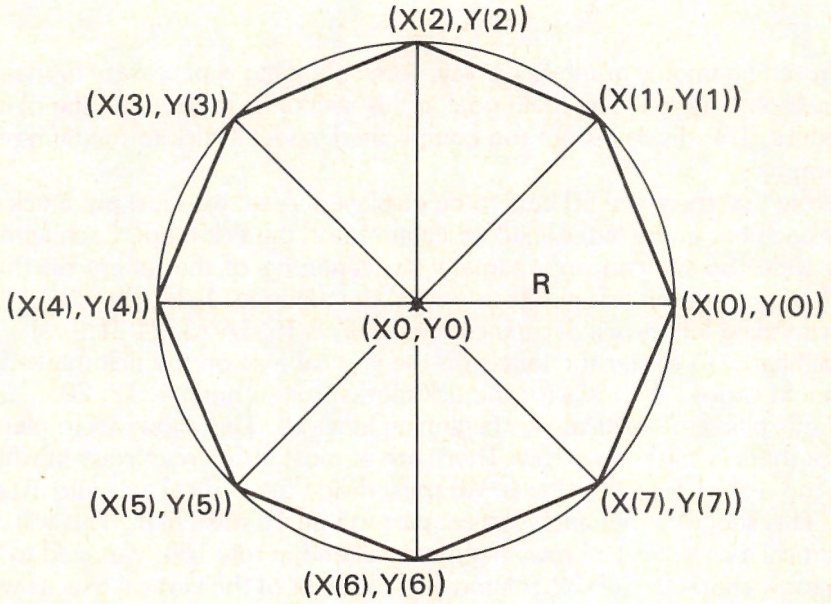


Figure 10-20. An Inscribed Polygon.

10.5 Drawing Barcharts

In this section, we'll apply what we have just learned about drawing lines and rectangles to draw the bar chart shown in Figure 10-21.

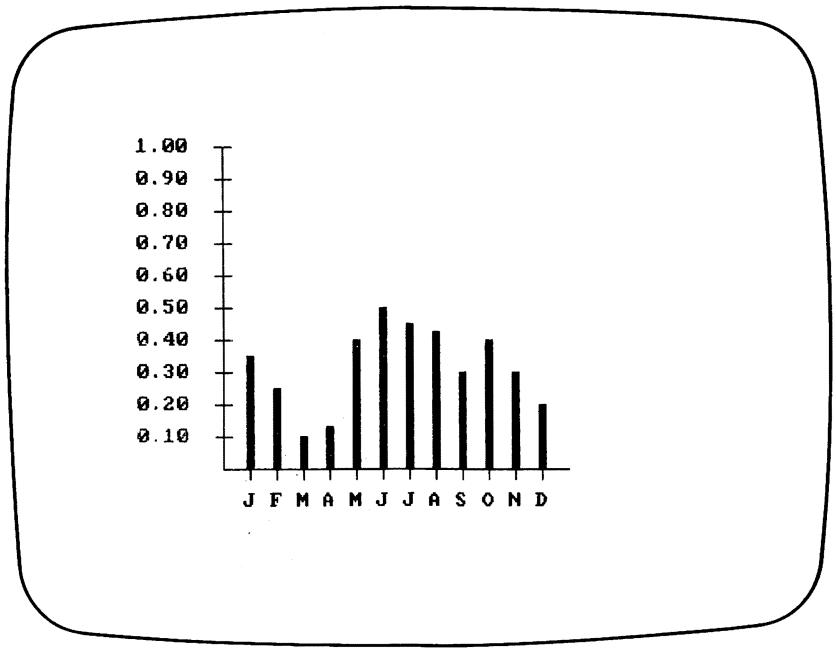


Figure 10-21. A Barchart.

In setting up any graphics display, some planning is necessary to make the display look "pretty." The main goal in this section is to illustrate the planning procedure. This display is not too complicated, so let's stick to medium-resolution graphics.

Note that there are 10 bars to be displayed. Also, we must put a tick mark under each bar and a letter lined up centered on the tick mark. Each letter is 8 pixels wide. So we can approximate the centering of the letters on the tick marks by placing the tick marks in one of the columns 4, 12, 20, 28, ... (The corresponding letters will occupy columns 0-7, 8-15, 16-23, 24-31, ...)

Similarly, to center the labels on the vertical axis on the tick marks there, we should choose the rows for the tick marks from among 4, 12, 20, ...

Let's place the vertical axis beginning in row 4. This allows us to place the top tick mark in the proper row. There are at most 195 screen rows in which to place the rest of the vertical axis. We must divide the vertical axis into 10 equal parts. This suggests that each vertical part will be 16 rows high. This will cause the vertical axis to be 160 rows high and will end in row 164. We need to leave room for 4 characters (=32 columns) to the right of the vertical axis as well as the tick marks. And let's not push the labels too far to the left. Finally, the vertical axis must be in one of the columns 4, 12, 20, 28, ... One possibility is to put the vertical axis in column 52. It turns out that this gives a reasonable looking display.

The horizontal axis will begin at the point (52,164). The horizontal axis is divided into 13 equal parts. Let's make each part two characters (=32 columns) wide. This means that the right endpoint of the horizontal axis is (52+13*16,164).

Here is the section of the program to draw the two axes.

```
100 'Draw axes
110 LINE (52,164)-(52+16*13,164)
120 LINE (52,164)-(52,4)
```

Next, let's draw the tick marks and print the labels. For the vertical axis, we use a PRINT USING statement to format the labels to contain one digit to the right of the decimal point. For the horizontal axis, we read the labels into a string array A$(). That is, A$(1)="J", A$(2)="F",... . To print the labels on the horizontal axis, we then print the various string array entries. Here is the program segment that draws the tick marks and labels the axes:

```
200 'Draw tick marks
210 FOR J=1 TO 10
220       LINE (47,164-16*J)-(57,164-16*J)
221       LOCATE 21-2*J,1
222       PRINT USING "#.##";J/10;
230 NEXT J
240 FOR J=1 TO 12
250       LINE (52+16*J,164)-(52+16*J,169)
260       LOCATE 23,(52+16*J)/8
270       PRINT A$(J);
280 NEXT J
```

Note the positioning of the labels. The labels on the vertical axis are in rows 1, 3, 5, 7, ..., 19. However, the first label in is row 19, the tenth in row 1. To achieve the correct positioning, we locate the cursor in row 21-2*J. So when J is 1, the label is put in row 21-2*1 = 19, and when J is 10, the label is put in row 21-2*10=1. (The labels start from row 21 and back up two rows at a time.)

Similarly, the position of the Jth horizontal label is gotten by dividing the column position, namely 52+16*J, by 8 (since a character occupies 8 columns). Note that this division always leaves a remainder of 4. The LOCATE statement drops any fractional part, so the character is positioned at the character position which starts just to the right of the tick mark. This is how the positioning was set up.

Now we have drawn everything but the bars. We store the height of the Jth bar in the variable BAR(J). The scale on the vertical axis is from 0 to 1, and the axis is 160 rows high. So the height of the Jth bar is BAR(J)*160. So the Jth bar runs from row 164 to row 164-BAR(J)*160. Let's make the bar extend for 5 columns, two on either side of the tick mark. This means that the Jth bar starts in column:

```
52+16*J - 2 = 50+16*J .
```

Similarly, the Jth bar ends in column:

```
52+16*J + 2 = 54+16*J .
```

Here are the instructions to draw the bars.

```
300 'Draw bars
310 FOR J=1 TO 12
320     LINE (50+16*J,164)-(54+16*J,164-BAR(J)*160),,BF
330 NEXT J
```

Finally, we assemble the various pieces into a single program:

```
10 DIM A$(12),BAR(12)
20 CLS:SCREEN 1
30 KEY OFF
40 FOR J=1 TO 12
50     READ A$(J)
60 NEXT J
70 FOR J=1 TO 12
80     READ BAR(J)
90 NEXT J
100 'Draw axes
110 LINE (52,164)-(52+16*13,164)
120 LINE (52,164)-(52,4)
200 'Draw tick marks
210 FOR J=1 TO 10
220     LINE (47,164-16*J)-(57,164-16*J)
221     LOCATE 21-2*J,1
222     PRINT USING "#.##";J/10;
230 NEXT J
240 FOR J=1 TO 12
250     LINE (52+16*J,164)-(52+16*J,169)
```

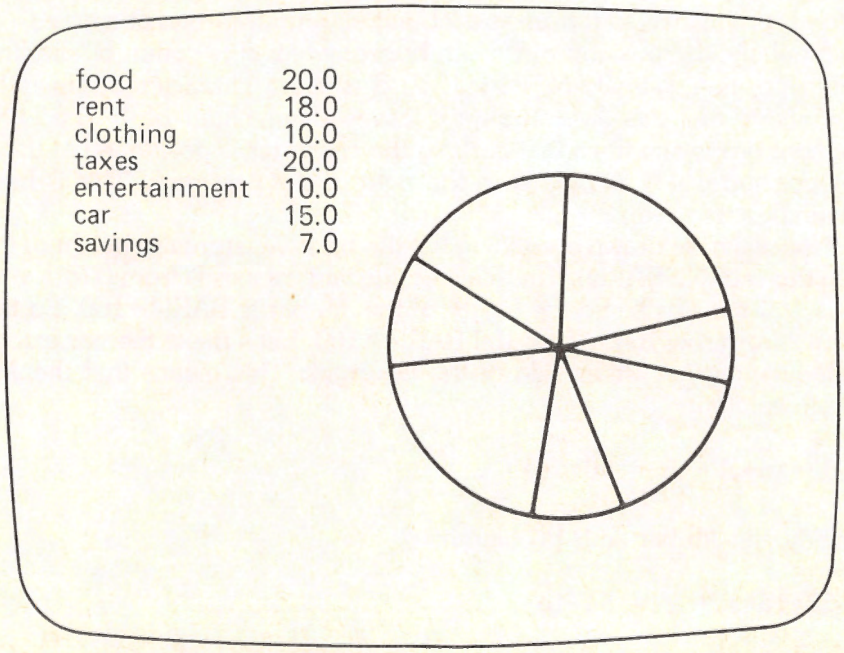


Figure 10-22.

```
260     LOCATE 23,(52+16*J)/8
270     PRINT A$(J);
280 NEXT J
300 'Draw bars
310 FOR J=1 TO 12
320     LINE (50+16*J,164)-(54+16*J,164-BAR(J)*160),,BF
330 NEXT J
1000 DATA J,F,M,A,M,J,J,A,S,O,N,D
1010 DATA .35,.25,.10,.13,.40,.50,.45,.425,.30,.40,.30,.20
2000 GOTO 2000
```

10.6 Drawing Pie Charts

As an application of the CIRCLE command, let's draw the pie chart shown in Figure 10-22.

To draw this pie chart, let's begin by creating an array to contain the various data and to list the data as shown on the left. We put the category names (Food, Clothing, and so forth) in an array B$( ). The numerical quantities we put in an array A( ). The first part of our program then consists of reading the data from DATA statements and setting up the two arrays. Also, we perform screen initialization by choosing SCREEN 2 (high-resolution graphics mode), and turning the function key display off. Here is the section of the program that accomplishes all these tasks:

```
100 'Program initialization
110 DIM A(7), B$(7),ANGLE(7)
120 DATA food, .20, rent, .18, clothing, .10, taxes, .20
130 DATA entertainment, .10, car, .15, savings, .07
140 FOR J=1 TO 7
150     READ B$(J), A(J)
160 NEXT J
170 SCREEN 2:           'high resolution
180 KEY OFF:            'turn off function keys
```

Our next step is to create the left portion of the display. This requires some care and planning. Let's skip the top 4 lines and begin the display in the 5th line. We set up the numbers in our data statements as decimals rather than percentages since the computation of angles that follows is more conveniently carried out in terms of decimals. However, to display percentages, we multiply each number A(J) by 100. To get a formatted display, we use the PRINT USING statement. Let's put the category description in print zone 1 and the percentage in print zone 2. Here are the instructions corresponding to this section of the program. Pay particular attention to the print statements in lines 240 and 250.

```
200 'Display listed data
210 CLS
220 PRINT: PRINT:PRINT:PRINT
230 FOR J=1 TO 7
```

```
240     PRINT B$(J),;: 'print and move to 2nd print field
250     PRINT USING "##"; 100*A(J);"%"
260 NEXT J
```

Finally, we come to the section of the program in which we draw the pie. The Jth data item corresponds to the proportion A(J) of the total pie. In angular measure, this corresponds to A(J)*(2*PI) (recall that 2*PI radians corresponds to the entire pie). The first slice of the pie begins at angle ANGLE(0), which we set at 0; it ends at ANGLE(1)=A(1)*(2*PI). The second slice begins where the first slice ends; namely, at ANGLE(1). It ends at ANGLE(1)+A(2)*(2*PI). And so forth. Here is the section of the program that draws the various pie slices. Notice that each of the sides of the pie slices is drawn twice, once in each of the slices to which it belongs. This does no harm.

```
300 'Draw Pie
310  ANGLE(0)=0
320  PI=3.14159
330  FOR J=1 TO 7
340     T=A(J)*(2*PI):    'T=angle for current data item
350     ANGLE(J)=ANGLE(J-1)+T
360     CIRCLE (450,100),100,,-ANGLE(J),ANGLE(J-1)
370  NEXT J
```

Note that in line 360, we did not specify a color. Nevertheless, we left space for the color parameter by inserting an extra comma. (The space for the color is an imaginary one between the two commas.) If BASIC calls for a parameter in a certain place, you usually may omit the parameter as long as you leave a place for it. If you don't, BASIC can't understand your statement.

You might wonder how we chose the center of the circle at (450,100) and the radius of 100. Well, it was mostly trial and error. We played around with various circle sizes and placements and chose one that looked good! In graphics work, you should not be afraid to let your eye be your guide.

For convenience, we now assemble the entire program into one piece.

```
100 'Program initialization
110  DIM A(7), B$(7),ANGLE(7)
120  DATA food, .20, rent, .18, clothing, .10, taxes, .20
130  DATA entertainment, .10, car, .15, savings, .07
140  FOR J=1 TO 7
150     READ B$(J), A(J)
160  NEXT J
170  SCREEN 2:              'high resolution
180  KEY OFF:               'turn off function keys
200 'Display listed data
210  CLS
220  PRINT: PRINT:PRINT:PRINT
230  FOR J=1 TO 7
240     PRINT B$(J),;: 'print and move to 2nd print field
250     PRINT USING "##.#"; 100*A(J)
```

```
260  NEXT J
300 'Draw Pie
310  ANGLE(0)=0
320  PI=3.14159
330  FOR J=1 TO 7
340     T=A(J)*(2*PI): 't=angle for current data item
350     ANGLE(J)=ANGLE(J-1)+T
360     CIRCLE (450,100),100,,-ANGLE(J),ANGLE(J-1)
370  NEXT J
```

This program is subject to a number of enhancements, some of which will be suggested in the exercises.

Exercises

1. Alter the program above so that it accepts the data from the keyboard. Allow it to keep asking for data until it receives a data name "@". Allow for up to 20 data items.
2. Modify the above program so that the pie is drawn in color 2 of palette 1. (This will involve some respacing, since you are now in medium-resolution and 40-character width.)

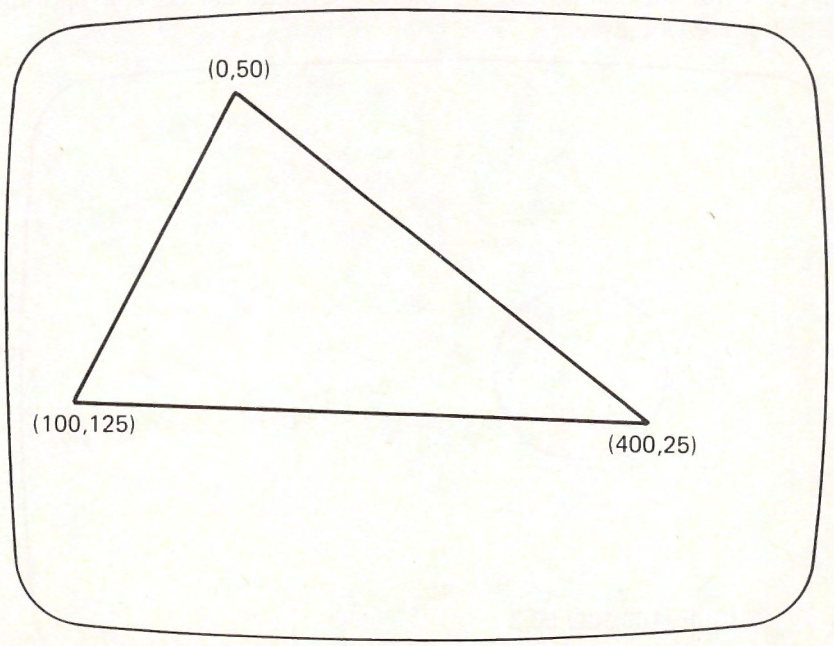


Figure 10-23. A Triangle.

10.7 Painting Regions of the Screen

Using the graphics commands of PCjr Cartridge BASIC, it is possible to draw a tremendous variety of shapes. For example, Figure 10-23 shows a triangle you may draw using several LINE statements. Figure 10-24 shows a circle drawn using the CIRCLE statement. Underneath each shape is a statement to draw the shape. The boundary lines of each shape are specified in the graphics statements used to draw it. The triangle is drawn in attribute 2. No color is indicated in the case of the circle, so it is drawn in attribute 3.

The PAINT statement allows you to color the "inside" of a region, just as if the region were in a coloring book and you used a crayon to color it. For example, we may use the PAINT command to paint the interiors of the triangle of Figure 10-23 and the circle of Figure 10-24.

The format of the PAINT command is:

```
PAINT (x,y),color,boundary
```

Here (x,y) is a point of the region to be painted, **color** is the color paint to use, and **boundary** is the color of the boundary. PAINT starts from the point (x,y) and begins to paint in all directions. Whenever it encounters the boundary color, it stops PAINTING in that direction.

For example, consider the triangle in Figure 10-23. The point (75,75) lies inside the triangle. And the triangle itself is drawn in attribute 2. Suppose that we wish to color the interior of the triangle in attribute 3. The appropriate PAINT statement would be:

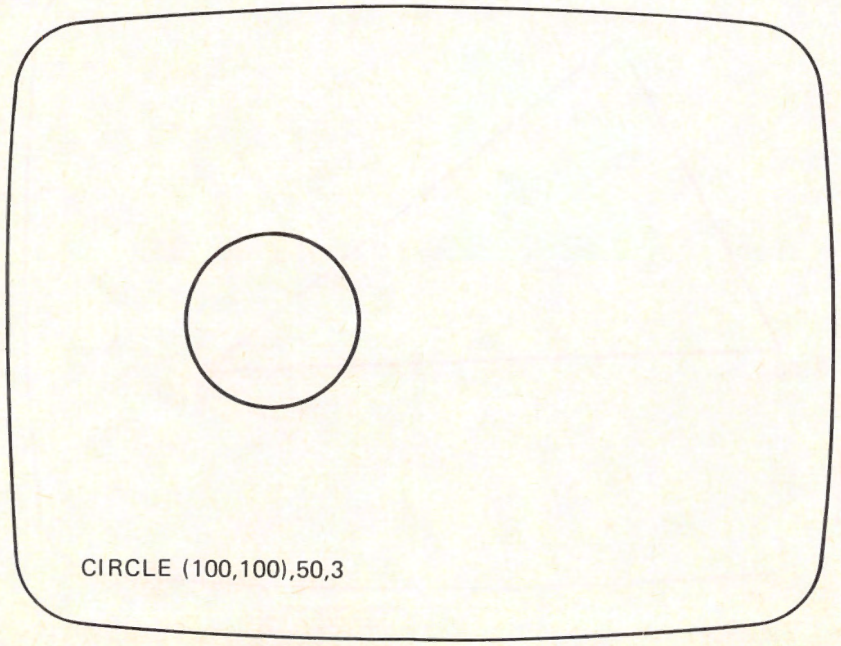


Figure 10-24. A Circle.

```
PAINT (75,75),3,2
```

TEST YOUR UNDERSTANDING 1 (answer on page 274)
Write a statement which will color the interior of the circle of Figure 10-24 in color 1.

PAINT is a very straightforward statement to understand. The main difficulty, however, is in specifying a point within the region. Or, to put it more precisely, if we are given a region, how do we specify a point within it? Well, that's a mathematical question. And I just happen to be a mathematician! So I can't resist explaining a little mathematics at this point.

Let's begin by considering the case of the rectangle (x1,y1)-(x2,y2). The center of the rectangle is at the point ((x1+x2)/2, (y1+y2)/2); that is, to obtain the coordinates of the center of the rectangle, we average the values of the coordinates of the opposite corners. See Figure 10-26.

Another way of getting the same answer is to average the values of the coordinates of all 4 corners: (x1,y1), (x1,y2), (x2,y2), (x2,y1). Now there are 4 x-coordinates to add up, but we must divide by 4: We obtain (2*x1+2*x2)/4 = (x1+x2)/2, and similarly for the y-coordinate.

Let's now consider a triangle with vertices (x1,y1), (x2,y2), (x3,y3). Suppose that you average the coordinates to obtain

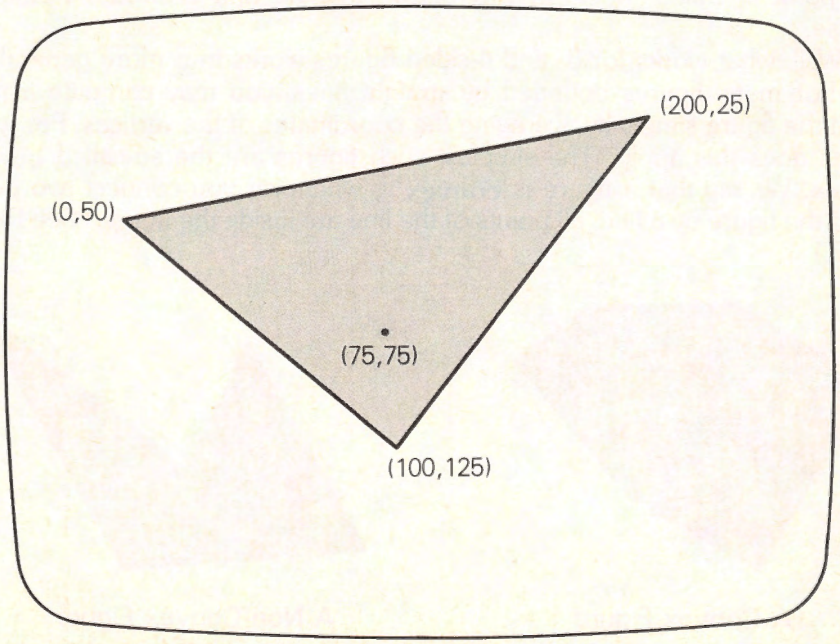


Figure 10-25. PAINTing the Interior of the Triangle.

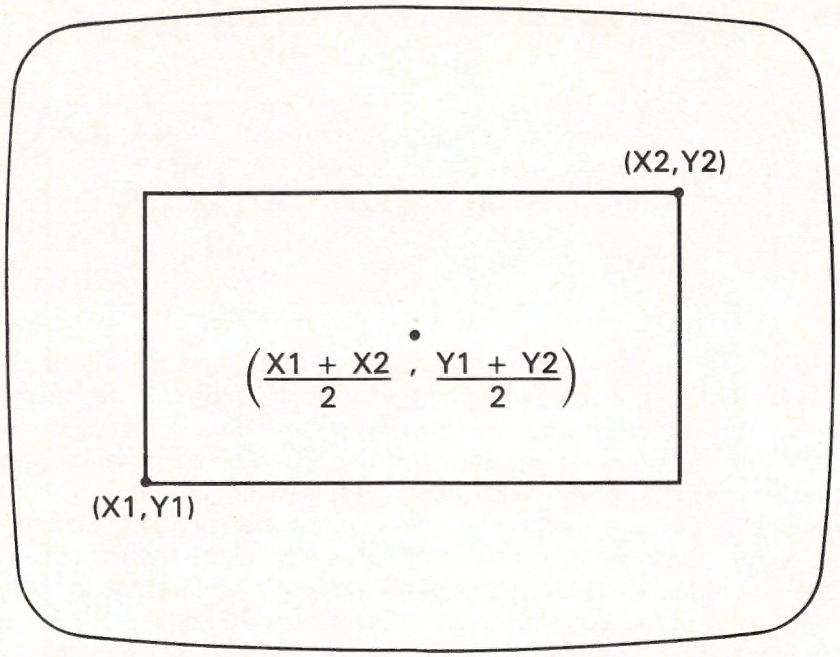
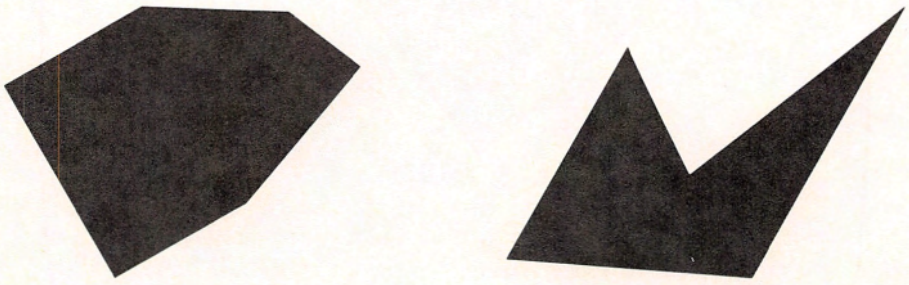


Figure 10-26. The Center of a Rectangle.

```
( (x1+x2+x3)/3, (y1+y2+y3)/3 ).
```

This point is called the **centroid** of the triangle and is always inside the triangle.

Well, what works for 3- and 4-sided figures works in a more general setting. For many figures bounded by straight lines, you may compute a point within the figure simply by averaging the coordinates of the vertices. For which figures does this apply? The simplest such figures are the so-called **convex bodies**. We say that a figure is **convex** if, whenever you connect two points within the figure by a line, all points of the line are inside the figure. (See Figure 10-27.)



A Convex Figure A Non-Convex Figure

Figure 10-27. Convex and Non-Convex Figures.

A convex figure bounded by line segments is a type of polygon. Suppose that the vertices of such a polygon are (x1,y1), (x2,y2), ... , (xn,yn). Then the point

```
( (x1+...+xn)/n, (y1+...+yn)/n )
```

obtained by averaging the x- and y-coordinates is called the **centroid** of the polygon. And the centroid is always inside the polygon.

So if you wish to PAINT a convex polygon, just compute the centroid. And this will give you the point to use in the PAINT statement!

10.8 The Graphics Macro Language*

Using the various statements of PC BASIC, you may draw some very complex screen images. However, the programs can become rather complex. Many drawings consist only of straight lines, in various positions on the screen. Such drawings may be concisely described and drawn using the Graphics Macro Language, as implemented in the DRAW statement.

To understand the DRAW command, it helps to think of an imaginary pen you may use to draw on the screen. The motion of the pen is controlled by a graphics language used by DRAW. The format of the DRAW command is:

```
DRAW <string>
```

Here <string> is a sequence of commands from the graphics language.

In giving commands, you will refer to points on the screen. The action of many of the commands will depend on the **last point referenced**. This is the point most recently referred to in a graphics command associated with DRAW. The CLS and RUN statements both set the last point referenced to the center of the screen (this is (160,100) in medium-resolution graphics and (320,100) in high-resolution graphics).

The graphics commands associated with DRAW are indicated by single letters. The most fundamental is the M command:

```
DRAW "M x,y"
```

which draws a straight line from the last point referenced to the point with coordinates (x,y). After the statement is executed, the point (x,y) becomes the last point referenced.

Here are two variations on the M command:

1. If M is preceded by N, then the last point referenced is not changed. For example, here is a DRAW command to draw an angle, as in Figure 10-28. (The vertex of the angle is at (360,100) and the computer is assumed to be in SCREEN 2.)

*Graphics Macro Language is a registered trademark of Microsoft Corporation.

```
DRAW "M 500,100 NM 200,50"
```

2. If M is preceded by B, then the last referenced point is changed, but no drawing takes place. The BM command is used to relocate the pen. For example, here is a command to draw the angle of Figure 10-29, with the vertex located at (300,110):

```
DRAW "BM 300,110 M 500,100 NM 200,50"
```

TEST YOUR UNDERSTANDING 2 (answer on page 274)
Use the DRAW command to draw the triangle of Figure 10-23.

Using Relative Coordinates In our above discussion, all of our coordinates were **absolute**; that is, we specified the actual coordinates. However, you also may use this form of the M command:

```
M +r,+s
```

It will draw a line from the last reference point to the point that is r units to the right and s units down. (Down is in the direction of increasing y coordinates!) In a similar fashion, we may use the commands:

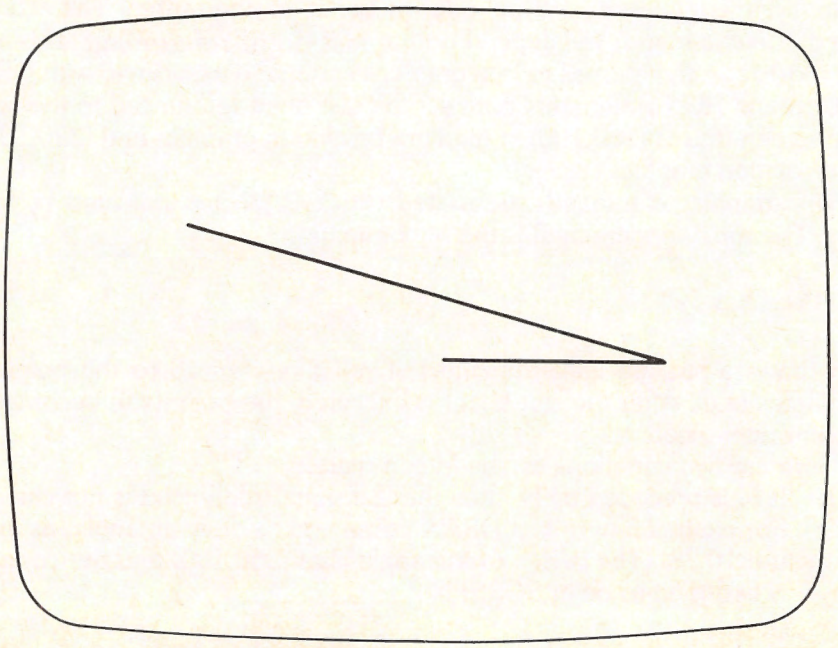


Figure 10-28. An Angle.

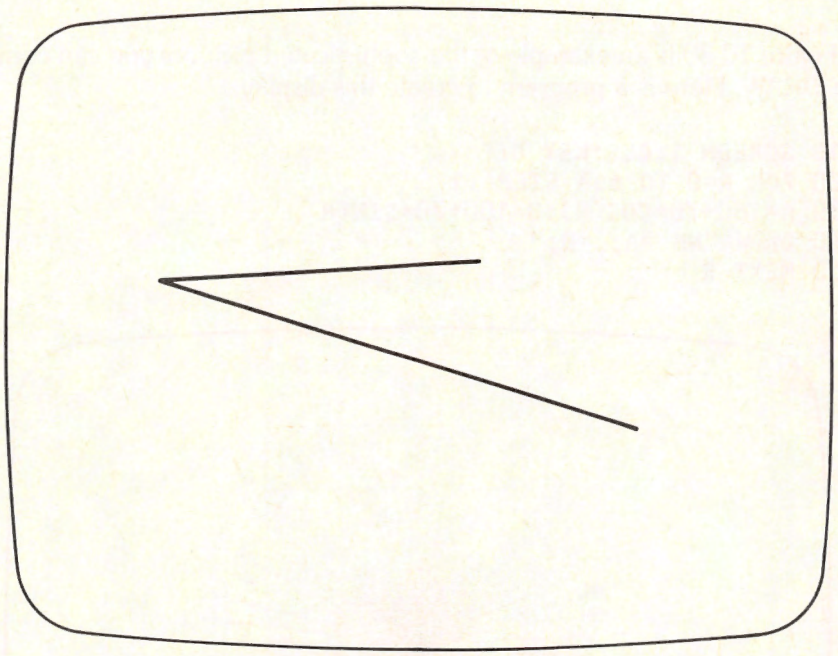


Figure 10-29. Another Angle.

```
M -r,+s
M -r,-s
M +r,-s
```

Specifying Coordinates Using Variables The coordinates in an M command may be specified by variables <variable1> and <variable2>, respectively. Here is the form of the command:

```
M =<variable1>;,=<variable2>;
```

Note the semicolons and the comma. You need these. For example, to draw a line from the last referenced point to the point specified by the values of the variables A and B, we could use the command:

```
M =A;,=B;
```

By preceding = signs with a + sign, we may use variables to specify a relative coordinate position. For example, to draw the line to the point that is A units to the right and B units down, we could use the command:

```
M +=A;,+=B;
```

Note that the signs of A and B give the actual direction of motion. For example, if A is negative, then the motion will be ABS(A) units to the left.

Figure 10-30 is an example of the sophisticated pictures you can compose using DRAW. Here is a program to create this display.

```
10 SCREEN 1:CLS:KEY OFF
20 FOR R=0 TO 6.3 STEP .1
30 A=160+70*COS(R):B=100+70*SIN(R)
40 DRAW "NM =A;,=B;"
50 NEXT R
```

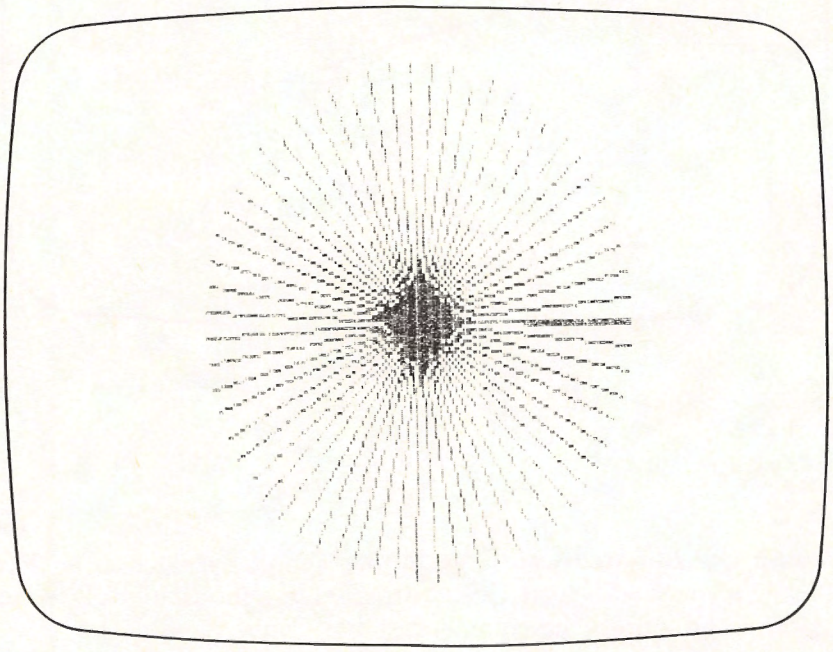


Figure 10-30. A Complex Display.

TEST YOUR UNDERSTANDING 3 (answer on page 274)
Use the random number generator to generate 50 pairs of random points. Use DRAW to draw a line associated with each pair.

More About Relative Motions In most drawing, coordinates are given in relative rather than absolute form. To shorten the lengths of the strings involved in describing such motions, DRAW includes the following commands:

```
U n - Move n units up
D n - Move n units down
L n - Move n units left
R n - Move n units right
E n - Move n units northeast
      (n units to the right, n units up)
```

```
F n - Move n units southeast
      (n units to the right, n units down)
G n - Move n units southwest
      (n units to the left, n units down)
H n - Move n units northwest
      (n units to the left, n units up)
```

The effect of these commands is illustrated in Figure 10-31.

You may use the N and B options with the commands U-G. For example, the command:

```
DRAW "NU 50"
```

will draw a line from the last referenced point upward for 50 units. However, the last referenced point is not updated. Similarly, the command:

```
DRAW "BU 50"
```

will update the last referenced point to the point 50 units up from the current point. However, no line is drawn.

You also may use variables in connection with the commands U-G. For example, consider the command:

```
DRAW "U =A;"
```

It will draw a line from the last referenced point A units upward. (If the value of A is negative, then the motion will be downward.)

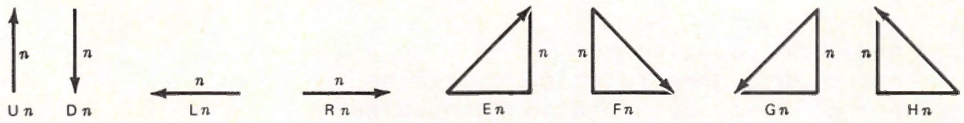


Figure 10-31. The Relative Motion Commands U-G.

Color You may specify color within DRAW by using the command:

```
C n
```

Here n is 0, 1, 2 or 3 and refers to a color in the current palette.

Here is a program to draw the sailboat of Figure 10-32.

```
10 SCREEN 1,0: CLS: KEY OFF
20 COLOR 7,0
30 DRAW "C1 L60 E60 D80 C2 L60 F20 R40 E20 L20"
```

The background is white, the sail green, and the boat red.

Angle You may rotate a figure through an angle that is a multiple of 90 degrees. Just precede the draw string (describing the figure in unrotated form)

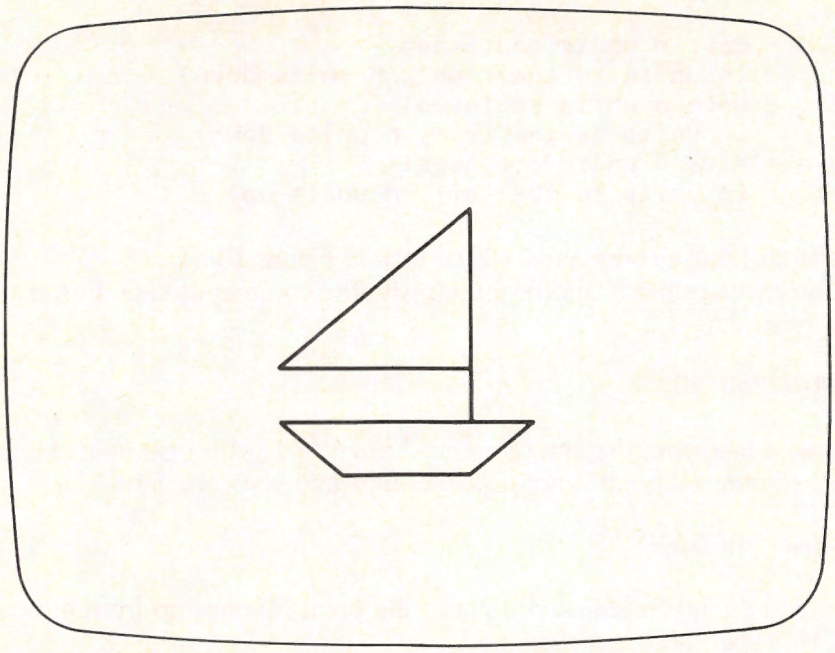


Figure 10-32. A Sailboat.

with the command:

```
A n
```

Here:

```
n=0 : no rotation
n=1 : 90-degree rotation clockwise
n=2 : 180-degree rotation clockwise
n=3 : 270-degree rotation clockwise
```

For example, here is a program that illustrates the sailboat of Figure 10-32 rotated through the various possible angles. (See Figure 10-33.)

```
10 CLS: SCREEN 1: KEY OFF: PSET (160,100)
20 INPUT "ANGLE (0-3)";N
30 DRAW "A=N; BU40 L30 E30 D40 L30 F10 R20 E10 L10"
```

Scale You may automatically scale figures (make them larger or smaller) using the command:

```
S n
```

All line lengths will be multiplied by n/4. Here n is an integer in the range 1 to 255.

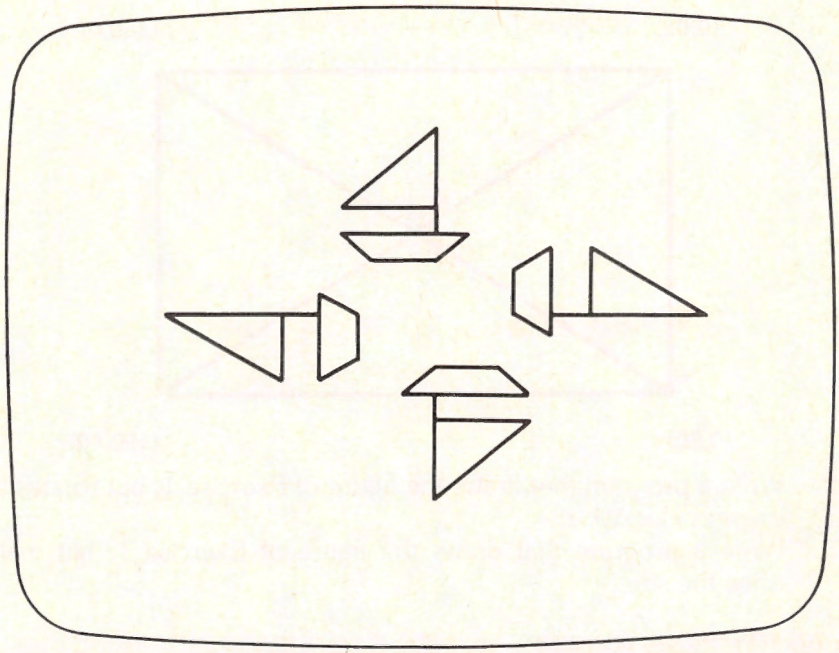


Figure 10-33. Rotated Sailboats.

TEST YOUR UNDERSTANDING 4 (answer on page 274)
Write a command to draw the sailboat of Figure 10-33, but at half scale.

Substrings You may define a string, A$, outside a draw statement and then use it in the form:

```
DRAW A$
```

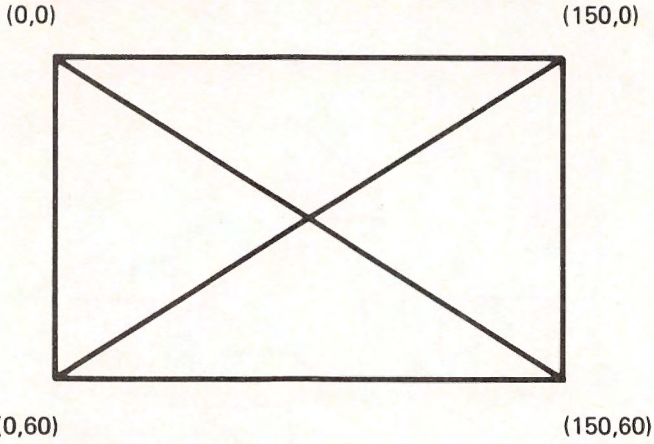
Often, you will wish to use one string several times within a single picture. (This is convenient, for example, if you wish to draw the same figure in several parts of the screen.) You may incorporate a string A$ within a larger string by preceding it with the letter X. For example, here is a statement that draws A$, moves up 50 units, and draws A$ again:

```
DRAW "XA$; BU50; XA$"
```

Note that X commands are separated from adjacent commands with semicolons.

Exercises

1. Write a program to draw the following figure:



2. Write a program that draws the figure of Exercise 1, but rotates it 270 degrees clockwise.
3. Write a program that draws the figure of Exercise 1, but makes it twice the size.

ANSWERS TO TEST YOUR UNDERSTANDING

```
1: PAINT (100,100),1,3
2: DRAW "BM 0,50 M 100,125 M 400,25 M 0,50"
3: 10 DIM X(50),Y(50)
   20 CLS: SCREEN 2: KEY OFF
   30 FOR J=1 TO 50
   40     X(J)=INT(RND*620)): X*(J)=INT(RND*620))
   50     Y(J)=INT(RND*200)): Y*(J)=INT(RND*200))
   60     DRAW "BM =X(J); =Y(J); M =X*(J);=Y*(J);"
   70 NEXT J
4: DRAW "S2 C1 L60 E60 D80 C2 L60 F20 R40 E20 L20"
```

10.9 Saving and Recalling Graphics Images

Cartridge BASIC contains commands that allow you to save and recall the contents of any rectangle on the screen. This is extremely convenient in many graphics applications, particularly animation.

Let's begin this discussion with a description of the image to be saved. The image must consist of a rectangular portion of the screen. The rectangle in question may start and end anywhere, and may contain text characters, portions of text characters, or a graphics image. You specify the rectangle by giving the coordinates of two opposite vertices: either the upper-left and lower-right, or the lower-left and upper-right. Thus a rectangle is specified in the same way as in using the LINE statement to draw a rectangle. Here are some specifications of rectangles:

```
(0,0)-(100,100)
(3,8)-(30,80)
```

In specifying rectangles, remember to indicate the coordinates in terms of the current graphics mode (either medium- or high-resolution). In either graphics mode, text characters occupy 8 x 8 rectangles. For example, the character in the upper left corner of the screen occupies the rectangle (0,0)-(7,7). (Lines of text are always 8 pixels high.)

TEST YOUR UNDERSTANDING 1 (answer on page 277)
Specify the rectangle consisting of the second text line of the screen. (Assume that you are in the medium-resolution graphics mode.)

The GET statement allows you to store the contents of a rectangle in an array. You may use any array as long as it is big enough. Suppose that the rectangle is x pixels long and y pixels high. Then the size of the array must be at least:

```
4 + (2*x+7)*y/32
```

in medium-resolution and:

```
4 + (x+7)*y/32
```

in high-resolution. (Recall that the size of the array is specified in a DIM statement.) For example, suppose that the array is 10 pixels wide and 50 pixels high and is in medium-resolution. Then the array required to store the rectangle must contain at least:

```
4 + (2*10+7)*50/32 or 46
```

elements. We could use an array A() defined by the statement:

```
DIM A(46)
```

Once a sufficiently large array has been dimensioned, you may store in it the contents of the rectangle using the GET statement, which has the form:

```
GET (x1,y1)-(x2,y2), arrayname
```

For example, to store the rectangle (0,0)-(9,49) (this rectangle is 10 by 50) in the array A, we could use the statement:

```
GET (0,0)-(9,49), A
```

To summarize: To store the contents of a rectangle in an array, you must:

1. Use a DIM statement to define a rectangle of sufficient size.
2. Execute a GET statement.

You may redisplay the rectangle at any point on the screen by using the PUT statement. For example, to redisplay the rectangle stored in A, you could use the statement:

```
PUT (100,125), A
```

This particular statement would redisplay the rectangle in A, with the upper left corner of the rectangle at the point (100,125).

To see GET and PUT in action, let's examine the following program:

```
10 SCREEN 1
20 DIM LETTER(9)
30 LOCATE 1,1
40 PRINT "A"
50 GET (0,0)-(7,7),LETTER
60 CLS
70 PUT (100,100),LETTER
```

Line 10 puts BASIC in medium-resolution graphics mode. We are out to store an 8 x 8 array, so we use the above formulas to calculate the required array size, which works out to 9. In lines 30-40, we print a letter "A", and in line 50, we store the image in the array LETTER. We then clear the screen. Line 70 recovers the image from the array and places it with its upper left corner at the point (100,100).

Don't erase the screen yet. Type:

```
PUT (100,100),LETTER
```

and press ENTER. Note that the letter A at (100,100) disappears. If you type the same line again, the A reappears. This feature may be used to create the illusion of motion across the screen. Suppose that you wish to create the illusion that the letter A is moving across the screen. Merely display it and erase it from consecutive screen positions. The screen creates the displays faster than the eye can view them. What you see is a continuous motion of the letter across the screen. Here is a program to create this animation:

```
10 SCREEN 1
20 DIM LETTER(9)
30 LOCATE 1,1
40 PRINT "A"
50 GET (0,0)-(7,7),LETTER
60 CLS
70 FOR XPOSITION = 0 TO 311
80   PUT (XPOSITION,0),LETTER
90   PUT (XPOSITION,0),LETTER
100 NEXT XPOSITION
```

Note that the XPOSITION runs from 0 to 311. Although the screen is 319 pixels wide, the variable XPOSITION specifies the upper left corner of the rectangle, which is 8 x 8. So 311 is the largest possible value of the variable.

Animation is the backbone of all the arcade games that have become so popular in recent years.

Saving a Screen Image on Diskette

Storing large graphics images (such as the entire screen) takes a great deal of memory. To store the entire screen takes more than 16,000 bytes. Compare this with the fact that BASIC can use a maximum of 65,536 bytes. Because graphic images tend to use such large amounts of memory, it is often necessary to save the screen image on diskette. Here is a program for saving the current screen image on diskette under the filename "SCREEN".

```
10 DEF SEG = &HB800
20 BSAVE "SCREEN",0,&H4000
```

To recall the stored image to the screen, use the program:

```
10 DEF SEG = &HB800
20 BLOAD "SCREEN",0
```

Exercises

1. Specify the rectangle of length 80 and height 40 whose upper left corner is at (10,10).
2. Specify the rectangle that consists of the first two text columns of the screen.
3. Write a dimension statement for the rectangle specified in Exercise 1.
4. Write a dimension statement for the rectangle specified in Exercise 2.
5. Store the current screen contents on diskette.
6. Clear the screen and recall the screen contents stored in Exercise 5.
7. Store a happy face (ASCII character 2) in an array.
8. Display the happy face at the following points:
   a. (0,0)
   b. (50,50)
   c. (0,100)
9. Construct an animation that moves the happy face across the screen in text line 10.
10. Construct an animation that moves the happy face diagonally across the screen from the upper left to the lower right corner.

ANSWER TO TEST YOUR UNDERSTANDING

1: (0,8)-(319,15)

10.10 VIEW and WINDOW

In this section, we will discuss the VIEW and WINDOW statements—two of the very powerful graphics enhancements provided in Cartridge BASIC.

The WINDOW statement allows you to define your own coordinate system on the screen. For example, consider the statement:

```
WINDOW (-2,0)-(2,100)
```

It causes the screen coordinates to be redefined, as shown in Figure 10-34. Note that the lower left corner becomes the point (-2,0) and the upper right corner becomes the point (2,100). The x-coordinates of the screen run from -2 on the left to 2 on the right. The y-coordinates run from 0 at the bottom to 100 on the top. The point in the middle of the screen is (0,50).

After using a WINDOW command, all graphics commands work with the new coordinates. For example, suppose that we execute the above WINDOW statement. The statement:

```
PSET (0,50)
```

turns on the pixel at the center of the screen.

TEST YOUR UNDERSTANDING 1 (answer on page 282)
Assume that the screen coordinates are defined by the WINDOW command of Figure 1. Describe the location of the points:
a. (1,75) b. (-1,100) c. (2,10)

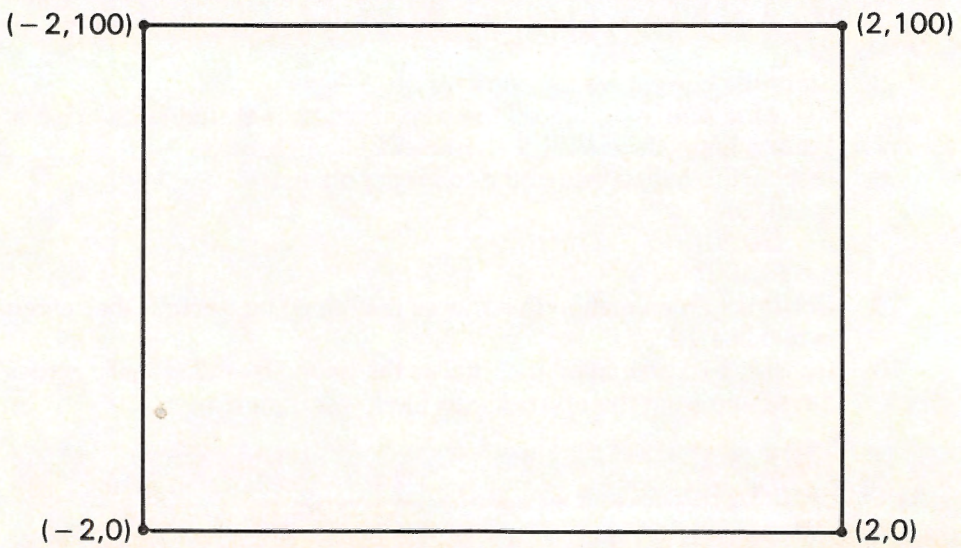


Figure 10-34. Cartesian Coordinates (-2,0)-(2,100).

The WINDOW statement does not disturb the contents of the screen, so you may use several different coordinate systems within a single program. Moreover, the placement of text is still governed by the usual text coordinate system (lines 1-25, columns 1-40 or 80), so you can mix text and graphics determined by a WINDOW command.

The WINDOW statement automatically reorders the values of the extreme x- and y-coordinates so that the lesser x-coordinate is on the left, the greater on the right, the lesser y-coordinate is at the bottom and the greater is on the top. Thus, for example, the following WINDOW statements are all equivalent:

```
WINDOW (-1,1)-(1,-1)
WINDOW (1,1)-(-1,-1)
WINDOW (-1,-1)-(1,1)
WINDOW (1,-1)-(-1,1)
```

Note that the above statements turn the screen into a portion of a Cartesian coordinate system, of the same type used in graphing points and equations in algebra. Note also that increasing values of the y-coordinate correspond to moving up the screen. This is the exact opposite of the normal graphics coordinates, in which the pixel rows are numbered from 0 (top of screen) to 199 (bottom of screen). A coordinate system in which increasing values of the y-coordinate correspond to moving **down** the screen are called **screen coordinates**. You may use the WINDOW statement to create a set of screen coordinates using the SCREEN option. For example, the statement:

```
WINDOW SCREEN (-2,0)-(2,100)
```

creates a coordinate system as shown in Figure 10-35. Note that y-coordinate 0 is now at the top of the screen.

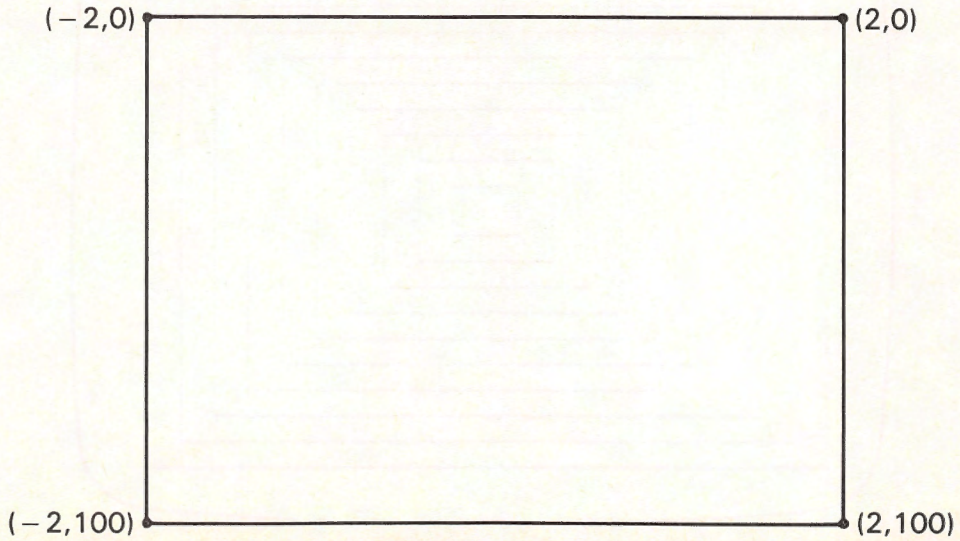


Figure 10-35. Screen Coordinates (-2,0)-(2,100).

Example 1. Use the WINDOW command to draw an expanding family of rectangles beginning at the center of the screen.
Solution. Let's use a single line statement, namely:

```
LINE (-.1,-.1)-(.1,.1),,B
```

to draw a rectangle with center (0,0). But let's use a sequence of WINDOW commands to redefine the coordinate system so that the radius .1 corresponds to successively larger distances on the screen. That is, we will let the Jth coordinate system be generated by the statement

```
WINDOW (-1/J,-1/J)-(1/J,1/J)
```

for J=1, 2, ... ,10. For the first coordinate system, the screen corresponds to (-1,-1)-(1,1). So the distance .1 seems small. (It corresponds to only .05 of the way across the screen). On the other hand, for J=10, the coordinate system corresponds to (-.1,-.1)-(.1,.1), so .1 is halfway across the screen. Here is our program:

```
10 SCREEN 2:KEY OFF
20 CLS
30 FOR J=1 TO 10
40       WINDOW (-1/J,-1/J)-(1/J,1/J)
50       LINE (-.1,-.1)-(.1,.1),,B
60 NEXT J
```

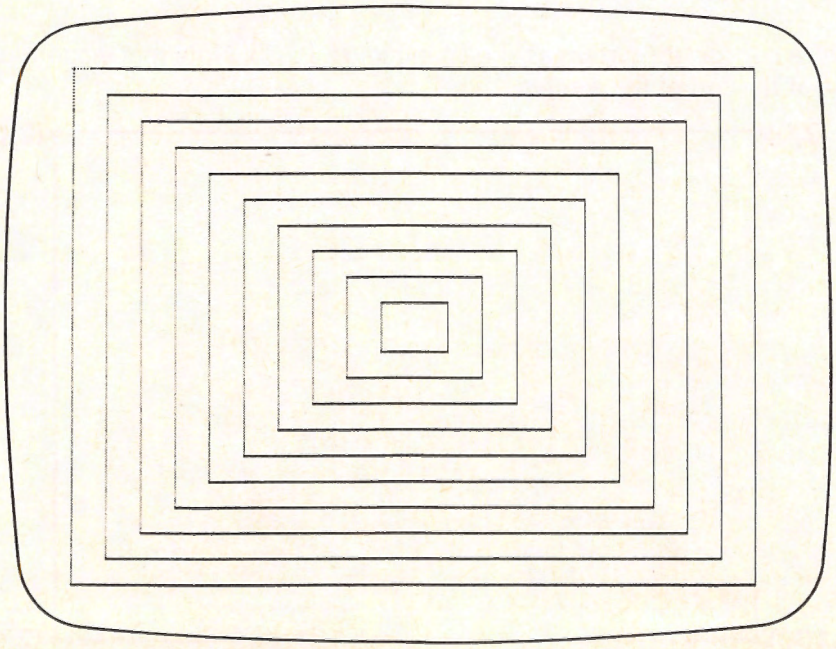


Figure 10-36. Expanding Rectangles.

The output of the program is shown in Figure 10-36.

The WINDOW statement ignores points corresponding to positions off the screen. This procedure is known as **clipping**.

RUN, SCREEN, and WINDOW with no parameters disable any previous WINDOW command.

The VIEW statement allows you to restrict screen activity to a portion of the screen. For example, to restrict all screen activity to the rectangle (20,10)-(100,200), we would use the statement

```
VIEW SCREEN (20,10)-(100,200)
```

This statement turns the rectangle (20,10)-(100,200) in a **viewport**. While a viewport is in effect, you may not plot any points outside the viewport. For example, if a circle statement refers to a circle that lies partially outside a viewport, then only the portion within the viewport will be drawn.

If you execute CLS while a viewport is in effect, you will erase **only** the inside of the viewport.

Note that the viewport applies only to graphics commands. Text commands may apply to any position on the screen, even though a viewport is in effect. Thus, for example, you may use LOCATE and PRINT as if the viewport were not present.

The full form of the VIEW statement is:

```
VIEW [SCREEN] (x1,y1)-(x2,y2),[color],[boundary]
```

The [color] option allows you to fill in the viewport with a particular attribute. The boundary option allows you to put a rectanglular boundary around the viewport. The value of [background] determines the attribute of the bounding rectangle.

For example, the statement:

```
VIEW SCREEN (10,20)-(200,100),3,2
```

defines a viewport colored in attribute 3 with a boundary in attribute 2, and the statement:

```
VIEW SCREEN (10,20)-(200,100),,2
```

defines a viewport with a boundary in attribute 2. The interior of the viewport is the background color.

You may omit the SCREEN parameter to obtain plotting relative to the viewport. For example, consider the statement

```
VIEW (10,20)-(200,100)
```

It defines the same viewport as above. However, the point (x,y) in a graphics statement will be interpreted to mean (x+10,y+20). In other words, the upper

left corner of the viewport is considered as the corner of the screen. The same clipping rule as for VIEW SCREEN applies: If a point (as computed relative to the viewport) lies outside the viewport, then it is not plotted.

You may disable a viewport using the statement:

```
VIEW
```

Similarly, using RUN or SCREEN will cancel a viewport.

You may combine VIEW and WINDOW. For example, consider the statements:

```
10 VIEW (80,16)-(559,167),,3
20 WINDOW (0,0)-(20,100)
```

They define a viewport in the rectangle (80,16)-(559,167) and then redefine the coordinates **within the viewport** as the Cartesian coordinates (0,0)-(20,100), so (0,0) corresponds to the lower left corner of the viewport and (20,100) to the upper right corner.

On the other hand, consider the statements:

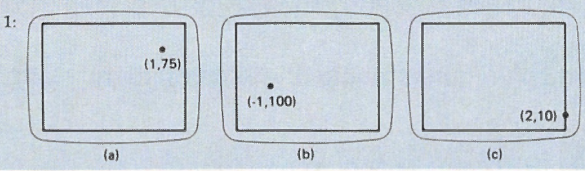
```
10 VIEW SCREEN (80,16)-(559,167),,3
20 WINDOW (0,0)-(20,100)
```

Now the WINDOW command refers to the entire screen. (0,0) corresponds to the lower left corner of the screen and (20,100) to the upper right corner of the screen. The viewport serves as a mask to clip off all points that (in the coordinates specified by WINDOW) land outside the viewport.

As we'll see in the next section, viewports are ideal for generating business graphics. We'll use a combination of VIEW and WINDOW to create a custom coordinate system on which to draw a bar graph.

ANSWER TO TEST YOUR UNDERSTANDING

1:



10.11 Sound and Music on the PCjr

The PCjr has a speaker located in the system unit. You may use this speaker to introduce sound and music into your programs. There are three sound commands on the PCjr - BEEP, SOUND, and PLAY. Let's survey the capabilities of these commands.

BEEP

The **BEEP** command is the simplest of the sound commands. It allows you to sound the speaker for 1/4 second. This command gives you no control

over the pitch or the duration of the sound. Here is an example of BEEP in a subroutine that responds to a mistake in input.

```
80 PRINT "YOU MADE A MISTAKE, TRY AGAIN!"
90 BEEP
100 RETURN
```

You also may use a **BEEP** statement within other statements, as in:

```
10 IF X=100 THEN BEEP
```

Professional programs employ sophisticated input routines that subject user input to a number of tests to determine if the input is acceptable. (Is the length correct? Does the input employ any illegal characters?) Here is a simple subroutine of this type. The main program assigns a value to the variable LENGTH, which gives the maximum length of an input string. The subroutine illuminates a box, beginning at location (1,1) (top left corner of the screen) to indicate the maximum field size for the input. The routine then allows you to input characters and to display them in the appropriate position in the illuminated field. For each character displayed, part of the illumination disappears. Moreover, using the backspace key restores one character space of illumination. If you attempt to input characters beyond the illuminated field, the routine beeps the speaker.

```
5000 'Input routine
5001 'LENGTH is the maximum number of characters in
     input string
5002 'COUNT is the current cursor position in the input
     field
5010 COUNT=1
5020 CLS
5030 LOCATE 1,1
5040 PRINT ""
5050 LOCATE 1,1
5060 FOR I=1 TO LENGTH
5070      LOCATE 1,I:PRINT CHR$(219);
5080 NEXT I
5090 LOCATE 1,1
5100 A$=INKEY$
5110 IF A$="" THEN 5100
5120 IF A$=CHR$(8) THEN 5200
5130 IF A$= CHR$(13) THEN 5240
5140 IF COUNT= LENGTH+1 THEN 5180
5150 LOCATE 1,COUNT:PRINT A$;
5160 COUNT=COUNT+1
5170 GOTO 5100
5180 BEEP
5190 GOTO 5090
5200 COUNT=COUNT-1
5210 IF COUNT=0 THEN BEEP:COUNT=COUNT+1
```

```
5220 LOCATE 1,COUNT:PRINT CHR$(219);
5230 GOTO 5090
5240 RETURN
```

Sound

The second speaker command is called **SOUND**. This handy little command enables you to access any frequency between 37 and 32767 Hertz (cycles per second, also abbreviated Hz). The duration of the sound is measured in clock ticks, and there are 18.2 clock ticks per second. A numeric expression in the range 0 to 65535 (that's slightly over one hour) is used. To produce a sound at 500 Hz and make it last for 40 ticks of the clock, we would use this statement:

```
SOUND 500, 40
```

Here is an elementary graphics program that has been enhanced by the SOUND command. It draws fixed triangles and random circles and blinks them in a manner suitable for illuminating rock concert. SOUND provides some audio accompaniment.

```
10 KEY OFF
15 'turns the key line off
20 SCREEN 1
25 ' switches from text mode to graphics mode
30 FOR I=1 TO 100
40   CIRCLE (RND*250, RND*200), 30
45    '  draws a circle with random coordinates and a
         diameter of 30
50 SOUND RND*1000+37, 2
55   ' creates a random sound from 37 to 1037 Hz. with a
       duration of 2 clock ticks
60   CLS
70   DRAW "E15; F15; L30"
75   ' draws a triangle
80   SOUND RND*1000 + 37, 2
90   CLS
100 NEXT I
110 END
```

Music on the PCjr

Next on the level of sound sophistication is the **PLAY** command. It enables you to turn your PCjr into a piano and play musical compositions as simple or as complex as you like. (There is even an arrangement of Beethoven's **Moonlight Sonata** for the PCjr!)

A few musical facts will help you a great deal in your programming:

1. Just like a piano, the PCjr uses 7 octaves, numbered 0 to 6. Each octave starts with C and goes to B.

2. Octave 3 starts with middle C.
3. The tempo of a song is the speed at which it is played. On the PCjr, tempo is measured by the number of quarter notes per second. The tempo may range from 32 to 255.
4. The PCjr allows you to style your notes as normal, legato, or staccato. Normal means that notes are held down for 7/8 of their defined length. Legato means that each note will play for the full time period that you set it to play, while staccato means that each note is held for only 3/4 of the time specified. Legato notes sound "smooth," whereas staccato notes are "crisp."

PLAY

The **PLAY** command allows you to show your creative musical genius, even if you can't play a comb. It uses a language that allows you to write music in the form of strings. Once the music has been transcribed, the PLAY statement allows you to play it on the speaker.

In order to use the PLAY command:

1. Code the desired musical notes as a string.
2. Use the PLAY command in the form

```
PLAY <string>
```

For example, consider this program. Why not type it in and listen to the results:

```
10 A$="O3L4EDCDEE"
20 PLAY "XA$; E2DDD2EGG2; T255XA$; EEDDEDP2C1"
```

The program probably makes the music look quite mysterious. However, the musical language is quite simple. Here is a summary.

NOTES Notes are indicated by the letters A to G with an optional #, +, or -. The # or + after a letter indicates a sharp, while - indicates a flat.
For example, the note "A sharp" is written A#, whereas "G flat" is written G-.

LENGTH L defines the LENGTH of a note. L1 is a whole note, L2 is a half note, L4 is a quarter note, ..., L64 is a 64th note. The L command defines the length of all subsequent notes until another L command is given. For example, to play the string of notes CDEFG in quarter notes, use this string:

```
L4 CDEFG
```

If you wish to define the length of a single note, omit the L and put the number indicating the length after the note. For example, C4 would indicate C held for a quarter note. Subsequent notes would be held for an amount defined by the most recent L command.

As in musical notation, a dot after a note indicates that the note is to be held for one-and-one-half times its usual length.

REST P1 is a whole note rest, P2 a half note, and so forth.

OCTAVE Initially, all notes are taken from octave 4 (the octave above the one beginning with middle C). The octave is changed by giving the O command. For example, to change to octave 2, the command would be O2. After you give an octave command, all notes are taken from the indicated octave unless you change the octave or temporarily overrule the octave (see below). Another method of specifying the octave is by using the symbols > and <. The symbol > means to go up one octave, and the symbol < means to go down one octave.

TEST YOUR UNDERSTANDING 1 (answer on page 287)
Write a string that plays an ascending C major scale in eighth notes, pauses for a half-note, and then plays the same scale descending.

TEST YOUR UNDERSTANDING 2 (answer on page 287)
Write a string that plays the scale of TEST YOUR UNDERSTANDING 1 in octave 5.

TEMPO Tempo is the speed at which a composition is played. Tempo is measured in terms of quarter beats per second. Unless you specify otherwise, the tempo is set at 120. You may set the tempo using the T command. For example, to set the tempo to 80, use the command T80. The tempo remains unchanged until you give another T command. The tempo may range from 32 to 255.

TEST YOUR UNDERSTANDING 3 (answers on page 287)
Write a string that plays the scale of TEST YOUR UNDERSTANDING 1 at a tempo of 80; at a tempo of 150.

STYLE You may select the style of notes from among: normal, legato, or staccato. The respective commands are MN, ML, and MS. The style chosen remains in effect until it is canceled by another style selection.

CHORDS You may PLAY up to three strings simultaneously, thereby creating harmony. For example, to play the C-major chord consisting of the notes C, E and G, you could use the statement:

```
PLAY "C","E","G"
```

Similarly, to simultaneously play the strings A$ and B$, you could use the statement:

```
PLAY A$, B$
```

Of course, it's up to you to code the strings so that the correct notes occur with one another. Chords may be used only if you are using an external speaker, such as the speaker in a TV set. Before playing chords, you must turn on the external speaker using the command:

```
SOUND ON
```

TEST YOUR UNDERSTANDING 4 (answer on page 287)
Write a string that plays the scale of TEST YOUR UNDERSTANDING 1 with legato style; with staccato style.

Ordinarily, the PLAY command will cause BASIC to stop while the specified notes are played. However, you also may use the PLAY command in **background mode**. In this mode, while the speaker plays the notes, BASIC continues executing the program, beginning with the statements immediately after the PLAY statement. The background mode may be started with the command MB. You may return to normal mode (also called **foreground mode**) with the command MF.

In coding music, you may wish to use the same string a number of times. This would occur, for example, in the case of a refrain. The X command allows you to repeat a string without retyping it. Just store the desired string in a string variable, say A$. Whenever the string is required, type:

```
XA$;
```

Exercises

1. Choose a piece of piano music and transcribe it for the PCjr.

ANSWERS TO TEST YOUR UNDERSTANDING

```
1:  A$="CDEFGAB O5 C O4 P2 CBAGFEDC"
2:  A$="O5CDEFGAB O6 C P2 C O5 BAGFEDC"
3:  A$="T80 CDEFGAB O5 C P2 C O4 BAGFEDC"
    A$="T150 CDEFGAB O5 C P2 C O4 BAGFEDC"
4:  A$="ML CDEFGAB O5 C P2 C O4 BAGFEDC"
    A$="MS CDEFGAB C O5 P2 C O4 BAGFEDC"
```

11

WORD PROCESSING

11.1 What is Word Processing?

Microcomputers are causing an office revolution. As microcomputers become cheaper and easier to use, they are finding their way into every aspect of business. Nowhere does the revolutionary impact of microcomputers promise to be greater than in the area of word processing. In brief, a **word processor** is a device made by combining the traditional typewriter with the capabilities of the computer for storing, editing, retrieving, displaying, and printing information. It is no exaggeration to say that the traditional typewriter is now as obsolete as a Model T. Over the next decade or so the typewriter will be completely replaced by increasingly sophisticated word processors.

Basically, a word processor uses the microcomputer as a typewriter. However, instead of using paper to record the words, a word processor uses the computer memory. First, the words are stored in RAM. When you wish to make a permanent record of them, you store them on disk as a data file. As you type, the text can be viewed on the video display. This part of word processing is not revolutionary. The true power of a word processor doesn't come into play until you need to edit the data in a document. Using the power of the computer, you can perform the following tasks quickly and with little effort: Move to any point in the document; add words, phrases, sentences, or even paragraphs; delete portions of the text; move a block of text from one part of the document to another; insert "boiler plate" information (standard pieces of text such as resumes or company descriptions) from another data file (for example, you could add a name and address from a mailing list); selectively change all occurrences of one word (say, "John") to another (say, "Jim"); or print the contents of a file according to a requested format.

In this chapter, we will discuss the characteristics of the various word processing programs which you can purchase for your PCjr. Then, to give you a taste of word processing proper, we will build a rudimentary word processor which you can use to prepare letters, term papers, memos, or other documents.

11.2 Using Your Computer As a Word Processor

A word processing system is a computer program for creating, storing, editing, and printing text.

At its most basic level, you use a word processing system like you would use a typewriter. Suppose that you wish to prepare a document. You would turn on the computer and run the word processing program. The program first asks for the type of work you would like to perform. Possibilities include: Type in a new document, edit an old document, save a document on diskette, or print a document. Select the first option. Next describe various format parameters to the word processor: line width, number of characters per inch, number of lines per page, spacing between lines, and so forth.

Then type the document exactly like you would on a typewriter. There are several huge exceptions, however! First of all, don't worry about carriage returns. The word processor takes care of forming lines. It accepts the text we type, decides how much can go on a line, forms the line, and displays it. Any text left over is automatically saved for the next line. The only function of the carriage return is to indicate a place where you definitely want a new line, such as at the end of a paragraph.

A second advantage of a word processor is in correcting errors. To correct an error, move the cursor to the site of the mistake, give a command to erase the erroneous letters or words, and type in the replacements. Of course, such action generally will destroy the structure of the lines. (Some lines now may be too long and others too short.) By using a simple command, it is possible to "re-form" the lines according to the requested format.

Typically, a word processor has commands which enable you to scroll through the text of a document to look for a particular paragraph. Some word processors even allow you to mark certain points so that you may turn to them without a visual search.

When the document is finally typed to your satisfaction, you give the computer an instruction which saves a copy of it on diskette. At a future time, you may recall the document and add to it at any point (even within the body of a paragraph!). Typically, word processors have certain "block operations" which allow you to "mark" a block and then either delete it, copy it, or move it to another part of the document. You also may insert other documents into the current document. This is convenient, for example, in adding boiler plate material, such as resumes, to your document. You may even use the block operations to alter boiler plate material to fit the special needs of the current document.

You may construct your document in as many sessions as you wish. When your diskette finally contains the document as you want it, you give the instruction to print. Your printer will produce a finished, error-free copy of the document.

As if the above were not enough of an improvement over the conventional typewriter, the typical word processor can do even more. The features available depend, of course, on the word processor selected. Here are some of the goodies to look for:

Global Search and Replace. Suppose that your document is a proposal. Further suppose that, at some later time, you wish to resubmit your proposal to Acme Energetics. In your original proposal, you included numerous references to the original company, Jet Energetics. A global search and replace

feature allows you to instruct the computer to replace every occurrence of a particular phrase with another phrase. For example, you could replace every occurrence of "Jet Energetics" with "Acme Energetics." Global search and replace can be even more sophisticated. In some systems, the word processor can be instructed to ask you whether or not to make each individual change. Another variation is to instruct the word processor to match any capitalization in the phrases replaced.

Centering. After typing a line you may center it using a simple command.

Boldface. You may print certain words in darker type.

Underscore. You may underline portions of text.

Subscripts and Superscripts. You may print subscripts (as in a_1) and superscripts (as in a^2). This is extremely useful for scientific typing and footnoting.

Justification. You may instruct the word processor to "justify" the right-hand margins of your text, so that the text always ends exactly at the end of a line. This is possible only if you have a printer which is capable of spacing in increments smaller than the width of a single letter.

Spelling Correction. There are a number of spelling correction programs which compare words of your document against a dictionary (sizes range from 20,000 to 70,000 words). If the program doesn't find a match, it asks you if the word is spelled correctly and gives you an opportunity to add the word to the dictionary. In this way the output of a word processor can be proofread by computer.

Footnotes and Indexing. Some of the more elaborate word processing programs keep track of footnotes and place each one on the proper page when the document is printed. In addition, some word processing packages allow you to designate terms to be included in an index of your document.

As of this writing, several word processing programs are available for the PCjr, practically all with sufficient power to handle all but the most demanding tasks. You should plan on adding one of them to your personal software library as soon as possible. If you don't, you will be missing out on one of the most powerful applications of your PCjr.

11.3 A Do-It-Yourself Word Processor

It is quite impractical for you to build your own word processor. For one thing, such a program is long and complicated. Moreover, if you write in BASIC, the operation of the program will tend to be rather slow. An efficient word processor almost always is written in machine language. Nevertheless, to acquaint you with a few of the virtues of word processing, let's ignore what I just said and build a word processor anyway!

Our word processor will be line-oriented: You type each line just as if you are typing it on a typewriter. At the end of each line, you will give a carriage return by typing ENTER. The Jth line will be stored in the string variable A$(J). Assume that you have 32K of memory available for document storage. This allows us to store and edit a document of about five double-spaced, typed pages. Our word processor will have five modes. In the first mode, we input

text. This operation will proceed exactly as on a typewriter. At the beginning of each line, the word processor will display a ?. Type your line after the question mark. Terminate the line with ENTER. To indicate that you don't wish to type any more lines, type % followed by **ENTER**.

A second mode allows us to save a document. The program saves your document as a data file under a file name requested by the program. The first item in a document file always will be the number of lines in the document. This quantity will be denoted by the variable L. Next are the lines of the document: A$(1), A$(2), ... , A$(L) .

A third mode lets you produce a draft version of the document. In this mode, the document is printed with each line preceded by its line number. The line numbers allow you to identify lines with errors. In order to print a document, you first must save it on the disk.

A fourth mode allows for document editing. To correct errors, you identify the line by number and retype the line. To end the edit session, type % followed by **ENTER**. This will bring you back to the beginning of the program, but you still will be working on the same document. After ending an edit session, your next action should be to save the document. The fifth and final mode allows you to print a final draft of a document.

When the word processor is first run, you will see the following prompt:

```
WORD PROCESSING PROGRAM
CHOOSE ONE OF THE FOLLOWING MODES
     INPUT TEXT(I)
     PRINT DRAFT (PD)
     PRINT FINAL DRAFT (PF)
     SAVE FILE (S)
     EDIT (E)
     QUIT (Q)
```

In response, you type I, PD, PF, S, E, or Q, followed by ENTER. If you choose I, the screen will be cleared and you may begin typing your document. For the other modes, there are prompts to tell you what to do. Here is a listing of the program.

You should use this program to type a few letters. You will find it a big improvement over a conventional typewriter. Moreover, this probably will whet your appetite for the more advanced word processing features described in the preceding section.

```
100 'Main Menu
110 CLS
120 DIM A$(150)
130 PRINT "WORD PROCESSING PROGRAM"
140 PRINT "CHOOSE ONE OF THE FOLLOWING MODES"
150 PRINT,"INPUT TEXT(I)"
160 PRINT ,"PRINT DRAFT(PD)"
170 PRINT, "PRINT FINAL DRAFT(PF)"
180 PRINT, "SAVE FILE(S)"
190 PRINT, "EDIT(E)"
```

```
200 PRINT, "QUIT(Q)"
210 INPUT X$
220 IF X$="I" THEN 300
230 IF X$="PD" THEN 390
240 IF X$="PF" THEN 480
250 IF X$="S" THEN 590
260 IF X$="E" THEN 670
270 IF X$="Q" THEN 830
280 GOTO 140
290 'Document Entry
300 L=1
310 PRINT "After each line of document, type ENTER"
320 LINE INPUT A$(L)
330 IF A$(L)="%" THEN L=L-1:GOTO 130
340 L=L+1
350 IF L <= 150 THEN 320
360 IF L>150 THEN PRINT "DOCUMENT TOO LARGE"
370 GOTO 130
380 'Print a draft copy
390 INPUT "DOCUMENT NAME";Y$
400 OPEN Y$ FOR INPUT AS #1
410 INPUT #1,L
420 FOR K=1 TO L
430     INPUT #1, A$(K)
440     LPRINT K;">";TAB(7) A$(K)
450 NEXT K
460 CLOSE 1
470 GOTO 130
480 INPUT "DOCUMENT NAME";Y$
490 'Print final copy of document
500 OPEN Y$ FOR INPUT AS #1
510 INPUT#1,L
520 FOR K=1 TO L
530     INPUT#1, A$(K)
540     LPRINT A$(K)
550 NEXT K
560 CLOSE 1
570 GOTO 130
580 'Save current document
590 INPUT "DOCUMENT NAME";Y$
600 OPEN Y$ FOR OUTPUT AS #1
610 WRITE#1,L
620 FOR K=1 TO L
630     WRITE#1,A$(K)
640 NEXT K
650 CLOSE 1
660 GOTO 130
670 INPUT "DOCUMENT NAME"; Y$
680 'Edit document
690 OPEN Y$ FOR INPUT AS #1
700 INPUT #1, L
```

```
710 FOR K=1 TO L
720     INPUT #1,A$(K)
730 NEXT K
740 INPUT "NUMBER OF LINE TO EDIT";Z
750 CLS
760 PRINT A$(Z)
770 PRINT "TYPE CORRECTED LINE"
780 LINE INPUT A$(Z)
790 IF A$(Z) <> "%" THEN 740
800 CLOSE 1
810 GOTO 130
820 'Exit program
830 END
```

Exercises

1. Modify the word processor to allow input of line width. (You will not be able to display lines longer than 80 characters on a single line. However, string variables may contain up to 255 characters.)
2. Modify the word processor so that you may extend a line. This modification should let your corrected line spill over into the next line of text. The program should then correct all of the subsequent lines to reflect the addition.
3. Modify the word processor to allow deletions from lines. Subsequent lines should be modified to reflect the deletion.

12

SOME ADDITIONAL PROGRAMMING TOOLS

In this chapter we will present four additional programming tools.

12.1 The INKEY$ Variable

The Keyboard Buffer

Many programs depend on input from the operator. We have learned to provide such input using the INPUT and LINE INPUT statements. When the program encounters either of these statements, it pauses and waits for input. The program will not proceed unless valid input is provided. The INKEY$ function provides an alternative method of reading the keyboard.

When a key is pressed, BASIC interrupts what it is doing and places the corresponding ASCII code in a reserved section of memory called the **keyboard buffer**. The keyboard buffer has space to record a number of keystrokes. The process of recording information in the keyboard buffer usually proceeds so that you don't even realize that the keyboard buffer is there. For instance, in typing program lines, BASIC is constantly reading the keyboard buffer and displaying the corresponding characters on the screen. In a similar fashion, an INPUT statement reads the keyboard buffer and displays the corresponding characters on the screen. A carriage return (generated by ENTER) tells the INPUT statement to stop reading the buffer.

As characters are read from the buffer, the space they occupy is released. If the buffer is full and you attempt to type a character, you will hear a beep on the speaker. This is to inform you that, until the buffer is read, further typed characters will be lost.

Note that you may type on the keyboard while a program is running. Even though BASIC is busy executing a program, it will pause to place your typed characters in the keyboard buffer and then return to execution. When the buffer is next read, it will read the characters in the order they were typed. In this way, you may "type ahead" of required program input.

The INKEY$ Variable

The INKEY$ function allows you to read one character from the keyboard buffer. When the program reaches INKEY$, it will read the "oldest" character

in the keyboard buffer and return it as a string. This procedure counts as reading the character, so that the character is removed from the buffer. If there is no character in the keyboard buffer, INKEY$ will equal the empty string.

INKEY$ has many uses. For example, suppose that you wish your program to pause until some key is pressed. Here is a statement which accomplishes this task:

```
100 IF INKEY$ = "" THEN 100
```

The program will continually test the keyboard buffer. If there is no character to be read, the test will be repeated, and so on until some key has been pressed.

Caution: We have explained the operation of INKEY$ in terms of the keyboard buffer so that you could understand the following trap: If the keyboard buffer is not empty, a reference to INKEY$ will remove a character. If you use INKEY$ a second time, you will be referring to the keyboard buffer anew and the value of the first INKEY$ will be lost. Moral: If you wish to use the value of INKEY$ again, store the value in a string variable, as in the statement:

```
10 A$ = INKEY$
```

Exercises (answers on page 371)

Suppose that the keyboard buffer is originally empty and you type A followed by F followed by c.

1. What is the value of INKEY$?
2. Suppose that the INKEY$ of Exercise 1 has been executed. Suppose that it is followed by the statement:

   ```
   IF INKEY$ <> "" THEN PRINT INKEY$
   ```

   What letter will be displayed on the screen?
3. Write a program which tests the keyboard and displays the keys pressed. It should display them in a single line, with no spaces between consecutive characters.

12.2 The Function Keys and Event Trapping

The function keys are the 10 keys labeled F1 through F10 on the top of the PCjr keyboard and are used in combination with Fn.

The Function Keys as User-Defined Keys

Each function key may be assigned a string constant containing as many as 15 characters. When a function key is pressed, the corresponding string is input to BASIC. In this way, you may reduce typing standard inputs to single keystrokes. This tends to eliminate errors in typing. For example, suppose that

an input statement asked for a response of HIGH, LOW, or AVERAGE. You could define function keys F1, F2, and F3 to be, respectively, the strings:

```
F1: HIGH <carriage return>
F2: LOW <carriage return>
F3: AVERAGE <carriage return>
```

Then pressing F1, for example, would be equivalent to responding to the input statement with the string HIGH followed by ENTER.

Setting Function Keys. You may assign strings to the function keys in either command or execution mode. To assign <string> to the function key n, use the statement:

```
KEY n, <string>
```

Suppose that you wish to assign key F1 the string:

```
LIST <carriage return>,
```

This may be done by the statement:

```
10 KEY 1, "LIST"+CHR$(13)
```

Subsequently, whenever you press key F1 (Fn-1) the desired string will be input to BASIC. In particular, if you happen to be in the immediate mode, inputting the string will cause the current program to be listed. In effect, you have customized the F1 key to a special application. In a similar fashion, you may customize other keys with commands or keystroke sequences which come up often in your work.

If you assign a null string to a function key, this will disable the key.

To display the current function key string assignments, use the command:

```
KEY LIST
```

The current string assignments will be displayed on the usual text area of the screen (lines 1-24).

In writing or running a program, it is often convenient to have a reminder of the various key string assignments on the screen at all times. This may be accomplished by giving the command:

```
KEY ON
```

The first 6 characters of each function key string will then be displayed in line 25 of the screen. (In case of a line width of 40, only the first 5 function key strings are displayed.) To turn off the function key display in line 25, use the command:

```
KEY OFF
```

TEST YOUR UNDERSTANDING 1 (answers on page 301)

a. Write commands to assign the following strings to function keys 1-3.

```
F1 - "ADDITION"
F2 - "SUBTRACTION"
F3 - "MULTIPLICATION"
```

Disable all other function keys.

b. Display the function key assignments in line 25.

Event Trapping (Cartridge BASIC Only)

For this section, you must use the cartridge version of PCjr BASIC.

We have described how to input data using INPUT, LINE INPUT, and INKEY$. All of these input methods have the following feature in common: The program decides when to ask for the input. You may use the function keys for a very different form of input.

Suppose that you wish the program to watch function key F1. The instant F1 is pressed, you wish the program to go to the subroutine in line 1000. This may be accomplished first by turning on event trapping for key F1 using the statement:

```
10 KEY(1) ON
```

which tells the program to examine F1 after every program statement is executed. Next, we tell the program that whenever F1 is pushed, go to the subroutine starting in line 1000:

```
20 ON KEY(1) GOSUB 1000
```

The program will inspect the keyboard buffer at the end of each program statement. When it detects that F1 has been pushed, it will execute GOSUB 1000.

You may use event trapping to implement a menu, as illustrated in the following example.

Example 1. Write a program to test addition, subtraction, and multiplication of two-digit numbers. Let the user select the operation with function keys F1 through F3. Let function key F4 end the program.

Solution. We create four subroutines, corresponding to addition, subtraction, multiplication, and END. These four subroutines begin in lines 1000, 2000, 3000, and 4000, respectively. What is of most interest to us, however, are lines 10-210. We first clear the screen and define the strings associated with function keys F1 to F4 as ADD, SUBTR, MULT, and EXIT. Then we disable the rest of the function keys. In lines 140-170, we set up the event-trapping lines for function keys F1-F4. In lines 180-200, we turn the event trapping on.

In line 210, we select the two numbers to use in our arithmetic. In line 220 we set up an infinite loop which continuously goes from 220 to 210 and back to 220. Note that this infinite loop accesses different random numbers in each repetition. Thus, the problem you get will depend on how long you take to press one of the function keys. Therefore, it is really unnecessary to use the randomize command to guarantee non-repeatability. The program keeps executing the loop until one of the function keys is pressed. Then it goes to the appropriate subroutine. Notice that the strings attached to F1-F4 are displayed in line 25 of the screen. This is accomplished in line 130.

```
10  'Initialize function keys
20   CLS
30   KEY 1, "ADD"
40   KEY 2, "SUBTR"
50   KEY 3, "MULT"
60   KEY 4, "END"
70   KEY 5,""
80   KEY 6,""
90   KEY 7,""
100  KEY 8,""
110  KEY 9,""
120  KEY 10,""
130  KEY ON
140  ON KEY(1) GOSUB 1000
150  ON KEY(2) GOSUB 2000
160  ON KEY(3) GOSUB 3000
170  ON KEY(4) GOSUB 4000
180  FOR J=1 TO 4
190     KEY(J) ON
200  NEXT J
210  X=INT(100*RND):Y=INT(100*RND)
220  GOTO 210
1000 'Addition
1010       CLS
1020       PRINT "ADDITION"
1030       PRINT "PROBLEM"
1040       PRINT X;" +";Y;" EQUALS?"
1050       INPUT ANSWER
1060       IF ANSWER=X+Y THEN 1070 ELSE 1090
1070       PRINT "CORRECT"
1080       GOTO 1100
1090       PRINT "INCORRECT. THE CORRECT ANSWER IS";X+Y
1100 RETURN
2000 'Subtraction
2010       CLS
2020       PRINT "SUBTRACTION"
2030       PRINT "PROBLEM"
2040       PRINT X;" -";Y;" EQUALS?"
2050       INPUT ANSWER
2060       IF ANSWER=X-Y THEN 2070 ELSE 2090
```

```
2070        PRINT "CORRECT"
2080        GOTO 2100
2090        PRINT "INCORRECT. THE CORRECT ANSWER IS";X-Y
2100 RETURN
3000 'Multiplication
3010        CLS
3020        PRINT "MULTIPLICATION"
3030        PRINT "PROBLEM"
3040        PRINT X;" *";Y;" EQUALS?"
3050        INPUT ANSWER
3060        IF ANSWER=X*Y THEN 3070 ELSE 3090
3070        PRINT "CORRECT"
3080        GOTO 3100
3090        PRINT "INCORRECT. THE CORRECT ANSWER IS";X*Y
3100 RETURN
4000 'Exit
4010        CLS
4020        KEY OFF
4030 END
```

There may be certain sections in the program where you want to disallow trapping of function key n. This may be done using either of the statements:

```
KEY(n) STOP
KEY(n) OFF
```

You may resume trapping of function key n using the statement

```
KEY(n) ON
```

If function key n is pressed while a STOP is in effect, the event will be remembered. When trapping is turned on, the program will jump to the appropriate subroutine. If you use a KEY(n) OFF statement, then function keys are not remembered.

In addition to the function keys, you may trap the cursor motion keys. (These are the four keys on the numeric keypad with arrows pointing in the four possible directions of cursor motion.) The commands for trapping these keys are

```
ON KEY(n) GOSUB
KEY(n) ON
KEY(n) OFF
KEY(n) STOP
```

where n = 11 corresponds to cursor up, n = 12 to cursor left, n = 13 to cursor right, and n = 14 to cursor down.

Exercises (answers on page 372)

1. Write a statement which disables function key F5.

2. Write a statement which assigns function key F1 the string "LIST"<carriage return> .
3. Write a program which causes function key F1 to erase the screen and start a new program.
4. Modify the program of Example 1 to disallow function key trapping during the subroutines beginning in lines 1000, 2000, and 3000.

ANSWER TO TEST YOUR UNDERSTANDING

1: a.

```
10 DATA ADDITION,SUBTRACTION,MULTIPLICATION
20 FOR J=1 TO 3
30     READ A$(J)
40 NEXT J
50 FOR J=1 TO 10
60     KEY J,A$(J)
70 NEXT J
```

b. `KEY ON`

12.3 Error Trapping

At the moment, our programs have only a single way to respond to an error: The program stops and an error message is displayed. Sometimes the program stops with good cause, since a logical error prevents BASIC from making any sense of the program. However, there are other instances in which the error is rather innocent: The printer is not turned on, the wrong data diskette is in the drive, or the user provides an incorrect response to a prompt. In each of these situations, it is desirable for the program to report the error to the user and wait for further instructions. Let's learn how to make the program take such action.

Ordinarily, the response to an error is to halt the program. However, an alternative is provided by the

```
ON ERROR GOTO <line number>
```

statement. If your program contains such a statement, BASIC will go to the indicated line number as soon as an error occurs. For example, suppose your program contains the statement:

```
ON ERROR GOTO 5000
```

Whenever an error occurs, the program will go to line 5000. Beginning in line 5000, you would program an **error-trapping routine**, which would:

1. Analyze the error
2. Notify the user of the error

3. Resume the program and/or wait for further instructions from the user.

The ON ERROR GOTO is called an **error-trapping statement**. It may occur anywhere in the program. After you type RUN, BASIC scans your program for the presence of an error-trapping statement. If BASIC finds an error-trapping line, it sets up code to send your program to the desired program line, should an error occur. In order to minimize BASIC's time to search for an error-trapping statement, you should place an error-trapping statement at the beginning of the program.

To see how an error-trapping routine is constructed, let's consider a particular example. Suppose that your program involves reading a data file, which must be on the diskette in the current drive. The program user may place the wrong diskette in the drive or may not insert any diskette at all. Let's write an error-trapping routine to respond to these two types of errors.

Let's place our error-trapping routine beginning in line 5000. We begin our program with the error-trapping line:

```
10 ON ERROR GOTO 5000
```

When an error occurs, BASIC makes a note of the line number in the variable ERL (error line) and the error number in ERR. It then goes to line 5000. The values of the variables ERL and ERR are at our disposal, just like the values of any other variables.

In our particular example, there are two types of errors to look out for: **File Not Found** (error number 53) and **Disk Not Ready** (error number 71). The first error occurs when the file requested by the program is not on the indicated disk. The second error occurs when either the diskette drive door is open or no diskette is in the drive. The error numbers were obtained from either the list of errors on the summary card at the back of the book or in Appendix A of the BASIC Reference Manual. In the case of each error, the error-trapping routine should notify the user and wait for the situation to be corrected. Here is the routine:

```
5000 'Error-trapping routine
5010 IF ERR=53 PRINT "File Not Found"
5020 IF ERR=71 PRINT "Disk Not Ready"
5030 IF ERR<>53 AND ERR<>71 THEN PRINT "Unrecoverable
     Error": END
5040 PRINT "CORRECT DISKETTE. PRESS ANY KEY WHEN READY."
5050 IF INKEY$="" THEN 5060
5060 RESUME
```

Several comments are in order. Notice that the error-trapping routine only allows recovery in the case of errors 53 and 71. If the error is any other type, line 5040 will cause the program to END. Line 5050 tells the operator to correct the situation. In line 5060, the program waits until the operator signals that the situation has been corrected. The RESUME in line 5070 clears the error

condition and causes the program to resume execution with the line that caused the error.

Note that we analyzed our errors using ERR. Note that ERL returns the line number, which is the same for both errors. So ERL doesn't allow us to tell the errors apart.

The RESUME statement has several useful variations:

`RESUME NEXT` — causes the program to resume with the line immediately after the line which caused the error.

`RESUME <line number>` — causes the program to resume with the indicated line number.

In designing and testing an error-trapping routine, it is helpful to be able to generate errors of a particular type. This may be done using the ERROR statement. For example, to generate an error 50 (field overflow) in line 75, just replace line 75 with

```
75 ERROR 50
```

When the program reaches line 75, it will simulate error 50. The program then will jump to the error-trapping routine to be tested.

Exercises (answers on page 372)

1. Write an error-trapping routine which allows the program to ignore all errors.
2. Write an error-trapping routine which allows detection of a **Type Mismatch** error in line 500. The response should be to display the error description and go to line 600.

12.4 Chaining Programs

The CHAIN instruction allows you to call a BASIC program from within an operating program. For example, the statement:

```
2000 CHAIN "SQUARES"
```

will cause the program to load and execute the program "SQUARES". The current program will be lost, as will the values of all its variables. BASIC will begin execution of "SQUARES" with its first line.

You may begin execution of "SQUARES" at line 300 by using the statement:

```
2000 CHAIN "SQUARES",300
```

You may carry ALL of the variables of the current program over into "SQUARES" and begin with the first statement of "SQUARES" by using the statement:

```
2000 CHAIN "SQUARES",,ALL
```

To carry forward all of the variables of the current program and to begin "SQUARES" at line 300, use the statement:

```
2000 CHAIN "SQUARES",300,ALL
```

A CHAIN statement is useful if a particular program is too large for memory. You may break the program into subprograms and use CHAIN statements to link them into a single program. In the interest of saving memory, you may wish to carry over only some of the variables of the current program. You may do this with the COMMON statement. For example, to pass the variables A, B, and C$ to "SQUARES", we would include the following statement in the chaining program:

```
10 COMMON A,B,C$
```

If, in addition, you wish to pass the values of the array SALARY(), the COMMON statement should be in the form:

```
10 COMMON A,B,C$,SALARY()
```

You may include as many COMMON statements as you wish. However, a variable may appear in only one of them. COMMON may appear anywhere in a program, but it is a good idea to place it at the beginning.

Be careful in using the CHAIN statement. It has the following significant effects:

1. There is no way to pass user-defined functions to the chained program.
2. Any variable types that have been defined by the statements DEFINT, DEFSNG, or DEFDBL will not be preserved. (See Chapter 14 for a discussion of variable types.)
3. Any error-trapping line number will not be preserved.
4. All files are closed.

The CHAIN statement completely eliminates the current program. You may keep a portion (or all) of the current program by using the CHAIN MERGE statement. For example, the statement:

```
CHAIN MERGE "SQUARES",300
```

will merge the program "SQUARES" with the current program and resume execution at line 300. The lines of "SQUARES" will be interleaved with the lines of the current program. If a line number in "SQUARES" duplicates a line number in the current program, then the line in the current program will be deleted in favor of the corresponding line in "SQUARES".

The program to be MERGEd must have been stored in ASCII format. (This is the format created by the command **SAVE,A**.) Otherwise BASIC will report a **Bad File Mode** Error.

In some applications, you may wish to delete a section of the current program before MERGEing. For example, the statement:

```
CHAIN MERGE "SQUARES",300,DELETE 300-1000
```

will first delete lines 300-1000 of the current program, merge "SQUARES" with the current program, and resume execution at line 300 of the resulting program.

CHAIN MERGE leaves files currently open and preserves variables, variable types, and user-defined functions.

Exercises (answers on page 372)

1. Write a statement to merge the program "L" into the current program and begin at the first line of the resulting program.
2. Write a program to run the programs "A", "B", and "C" one after the other.

13

COMPUTER GAMES

In the last few years, computer games have captured the imaginations of millions of people. In this chapter, we will build several computer games that use both the random number generator and the graphics capabilities of the PCjr. In many games, we need a clock to time moves. We will start by learning to tell time with the computer.

13.1 Telling Time With Your Computer (Cartidge BASIC With DOS Only)

The PCjr Disk Operating System has a built-in clock (a **real-time clock** in computer jargon) that allows your programs to take into account the time of day (in hours, minutes, and seconds) and the date (month, day, and year). You can use this feature for many purposes, such as timing a segment of a program (see Example 1).

Reading the Real-Time Clock

The real-time clock keeps track of six pieces of information in the following order:

Month (01-12)
Day (01-31)
Year (1980-2099)
Hours (00-23)
Minutes (00-59)
Seconds (00-59)

The date is displayed in the following format:

```
2-15-1984
```

The time is displayed in the following format:

```
14:38:27
```

The above displays correspond to February 15, 1984, at 27 seconds after 2:38 pm. Note that the hours are counted using a 24-hour clock, with 0 hours corresponding to midnight. Hours 0-11 correspond to am, and hours 12-23 correspond to pm. Also note that the year must be in the range 1980-2099.

The clock is programmed to account for the number of days in a month (28, 30, or 31), but it does not recognize leap years.

In BASIC, time is identified as TIME$. To display the current time on the screen, use the command:

```
10 PRINT TIME$
```

If it is currently 5:10, pm the computer will display the time in the format:

```
17:10:07
```

(The :07 denotes 7 seconds past the minute.)

BASIC identifies the date as DATE$. To display the current date of the screen, use the command

```
20 PRINT DATE$
```

If it is currently December 18, 1984, the computer will display

```
12-18-1984
```

TEST YOUR UNDERSTANDING 1 (answer on page 311)
Display the current time and date.

Setting the Clock

You have an opportunity to set the clock when starting the Disk Operating System. Recall that the initial DOS display asks you for the date. If you accurately answer this question, the computer will keep the correct date as long as it is operating continuously. Note, however, that the computer will lose track of this data as soon as it is turned off. You also may use **TIME$** and **DATE$** to set the time and date as follows: Suppose that the time is 12:03:17 and the date is 10/31/1984. You would then type the commands:

```
TIME$ = "12:03:17"
DATE$ = "10-31-1984"
```

These commands may be typed whenever the computer is not executing a program and they are typed without a line number. These commands also may be used within a BASIC program (with a line number, of course). For example, to reset the time to 00:00:00 within a program, you would use the statement

```
10 TIME$ = "00:00:00"
```

In setting the date, two variations are acceptable. First, you may replace some or all of the dashes in the date by slashes. All of the following are acceptable forms of the date:

```
10/31/1984 10-31-1984
10/31-1984 10-31/1984
```

Second, you may input the year as two digits. For example, you could input 1984 as 84. The computer will automatically supply the missing 19.

TEST YOUR UNDERSTANDING 2 (answer on page 311)
Write instructions which set the hours of the clock to 2 pm and the date to January 1, 1984.

TEST YOUR UNDERSTANDING 3 (answer on page 311)
Set the clock with today's date and time. Check yourself by printing out the value of the clock.

TEST YOUR UNDERSTANDING 4 (answer on page 311)
Write a program which continually displays the correct time on the screen.

Calculating Elapsed Time

The real-time clock may be used to measure elapsed time. You could ask the computer to count 10 seconds or three days. In such measurements, it is convenient to have the components (that is, the hours, minutes, seconds, and so on) of the time and date available individually. Next, let's discuss a method for determining these numbers.

Begin with the string **TIME$**. Suppose that **TIME$** is now equal to:

```
"10:07:32"
```

To isolate the seconds (the 32), we must chop off the initial portion of the string, namely "10:07:". We may do this using the statement **RIGHT$**:

```
RIGHT$(TIME$,2)
```

forms a string out of the rightmost two digits of the string **TIME$**. This is the string "32". In most applications, we will require the 32 as a number rather than as a string. To convert a string consisting of digits into the corresponding numeric constant, we may use the **VAL** function:

```
VAL("32") = 32,
VAL(" -15") = -15
```

and so forth. To obtain the SECONDS portion of the time as a numeric constant, use the statement:

```
10 SECONDS = VAL(RIGHT$(TIME$,2))
```

In a similar fashion, we may calculate the HOURS portion of the time by extracting the left two characters of the time and converting the resulting string into a numeric constant. The statement to accomplish this is:

```
20 HOURS = VAL(LEFT$(TIME$,2))
```

Finally, to calculate the MINUTES portion of the time, we must extract from TIME$ a string of two characters in length beginning with the fourth character. For this purpose, we use the **MID$** statement as follows:

```
30 MINUTES = VAL(MID$(TIME$,4,2))
```

To calculate the MONTH, DAY, and YEAR portions of the date as numeric constants, we use the statements:

```
40 MONTH = VAL(LEFT(DATE$,2))
50 DAY = VAL(MID$(DATE$,4,2))
60 YEAR = VAL(RIGHT$(DATE$,4))
```

The ON TIMER statement is even more convenient for calculating elapsed time. Consider the following two statements:

```
10 ON TIMER(10) GOSUB 200
20 TIMER ON
```

The first statement tells the computer that whenever the timer is turned on, BASIC should count 10 seconds and then go to 200. The timer may be turned on anywhere within the program using a statement like that on line 20.

Example 1. In Example 6 of Section 5.2, we developed a program to test mastery of addition of two-digit numbers. Redesign this program to allow 15 seconds to answer the question.

Solution. Let us use the real-time clock. After a particular problem has been given, we will start the seconds portion of the clock at 0 and perform a loop until 15 seconds have elapsed. After 15 seconds, the program will print out, "TIME'S UP. WHAT IS YOUR ANSWER?" Here is the program. Lines 50 and 60 contain the loop.

```
10 FOR J=1 TO 10:      'LOOP TO GIVE 10 PROBLEMS
20   INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30   PRINT "WHAT IS THEIR SUM?"
40   ON TIMER(15) GOSUB 100
50   TIMER ON
60   GOTO 60: 'WAIT 15 SECONDS
100  INPUT "TIME'S UP! WHAT IS YOUR ANSWER";C
120  IF A+B=C THEN 200
130  PRINT "SORRY. THE CORRECT ANSWER IS",A+B
```

```
140  GOTO 500:       'GO TO THE NEXT PROBLEM
200  PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210  R=R+1:          'INCREASE SCORE BY 1
220  GOTO 500:       'GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

TEST YOUR UNDERSTANDING 5 (answer on page 311)
Modify the above program so that it allows you to take as much time as you like to solve a problem, but keeps track of elapsed time (in seconds) and prints out the number of seconds used.

Exercises (answers on page 372)

1. Set the clock with today's date and the current time.
2. Print the current time on the screen.
3. Write a program which prints the date and time at one-second intervals.
4. Write a program which prints the date and time at one-minute intervals.

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
10 PRINT TIME$ : PRINT DATE$
20 END
RUN
```

2:
```
TIME$ = "14:00:00"
DATE$ = "1/1/84"
```

4:
```
10 PRINT TIME$
20 CLS
30 FOR J=1 TO 500
40 NEXT J: 'DELAY
50 GOTO 10
60 END
```
Note: This program is an infinite loop and will need to be terminated by pressing the key combination Ctrl-Break.

5: Delete lines 45-60. Add these lines:
```
100 INPUT "WHAT IS YOUR ANSWER";C
110 MINUTES = VAL(MID$(TIME$,4,2))
111 SECONDS = VAL(RIGHT$(TIME$,2))
112 PRINT "YOU TOOK",60*MINUTES+SECONDS,
"SECONDS"
```

13.2 Blind Target Shoot

The object of this game is to shoot down a target on the screen by moving your cursor to hit the target. The catch is that you only have a two-second look at your target! The program begins by asking if you are ready. If so, you press any key. The computer then randomly chooses a spot to place the target. It lights up the spot for two seconds. The cursor then moves to the upper left position of the screen (the so-called "home" position). You then must move the cursor to the target, based on your brief glimpse of it. You have five seconds to hit the target. (See Figure 13-1.)

Your score is based on your distance from the target, as measured in terms of the moves it would take to get to the target from your final position. Here is the list of possible scores:

Distance From Target	Score
0	100
1 or 2	90
3 to 5	70
6 to 10	50
11 to 15	30
16 to 20	10
over 20	0

Move the cursor using the cursor motion keys on the numeric keypad. We will use event trapping to interrupt the program while it is running.

Here is a sample session with the game. The underlined lines are those you type.

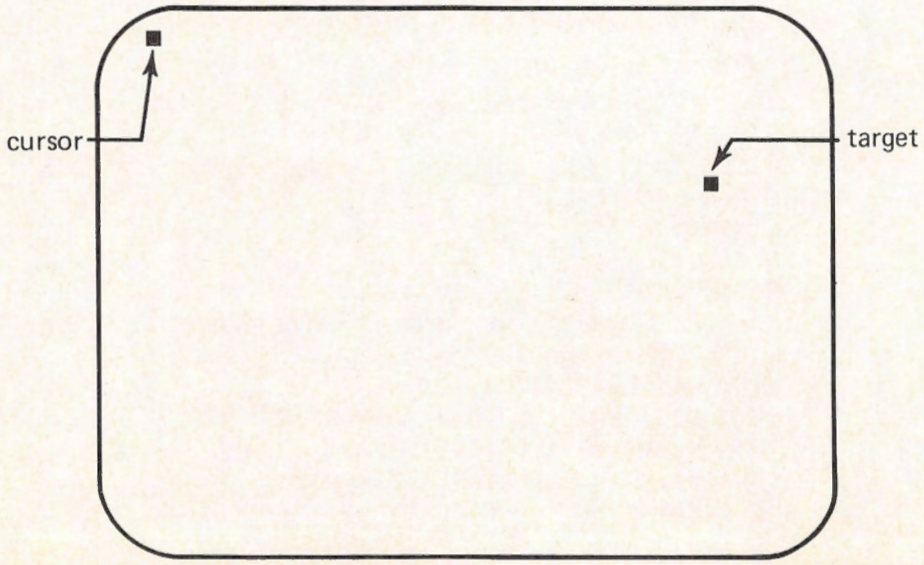


Figure 13-1. Blind Target Shoot.

```
RUN

BLIND TARGET SHOOT
TO BEGIN GAME, PRESS ANY KEY
```

Press any key. The screen clears. The target is displayed. See Figure 13-2.

The screen is cleared and the cursor is moved to the home position. See Figure 13-3a. The cursor is then moved to the remembered position of the target. See Figure 13-3b. Time runs out. See Figure 13-3c.

The score is calculated. See Figure 13-4.

Here is a listing of the program:

```
100 'Title Screen
110      CLS
120      KEY OFF
130      SCREEN 0: WIDTH 40
140      RANDOMIZE VAL(RIGHT$(TIME$,2))
150      PRINT "BLIND TARGET SHOOT"
160      PRINT "TO BEGIN GAME, PRESS ANY KEY"
170      IF INKEY$="" THEN 170
180      CLS
190 'Initialization
200     TIME$ = "0:0:0":          'Reset Clock
210     LOCATE ,,0:               'Turn off cursor
220 'Choose target location (targrow,targcol)
230     TARGCOL = INT(40*RND)+1
240     TARGROW = INT(25*RND)+1
```

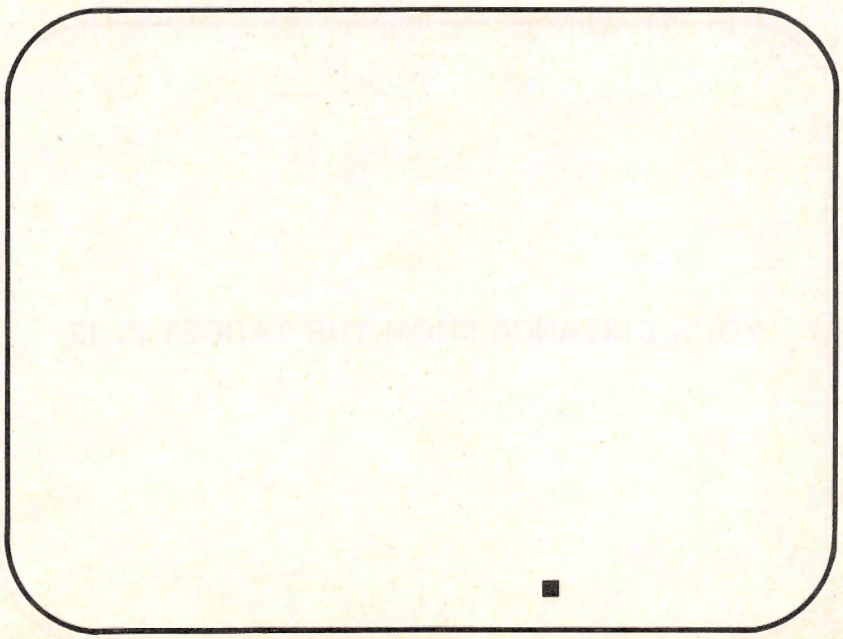


Figure 13-2.

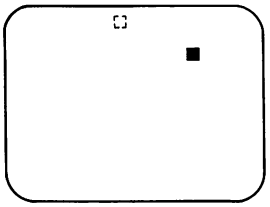


Figure 13-3a

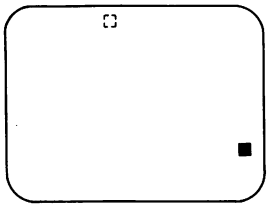


Figure 13-3b

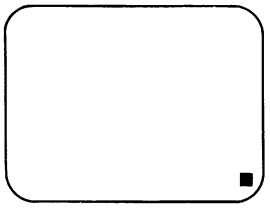


Figure 13-3c

```
250     LOCATE TARGROW,TARGCOL
260     PRINT CHR$(219):          'Display target
270 'Look at target
280     SECONDS = VAL(RIGHT$(TIME$,2))
290     IF SECONDS = 2 THEN 310 ELSE 280
300 'Two seconds elapsed
310     LOCATE TARGROW,TARGCOL
320     PRINT " ";:               'Blank out target
330     LOCATE ,,1,0,7
340     PRINT CHR$(11)
350     TIME$ = "0:0:0"
360     'Reset clock to 0
370 'Turn on cursor key trapping
380     ON KEY(11) GOSUB 480
390     ON KEY(12) GOSUB 520
400     ON KEY(13) GOSUB 560
410     ON KEY(14) GOSUB 600
420     KEY(11) ON
430     KEY(12) ON
440     KEY(13) ON
```

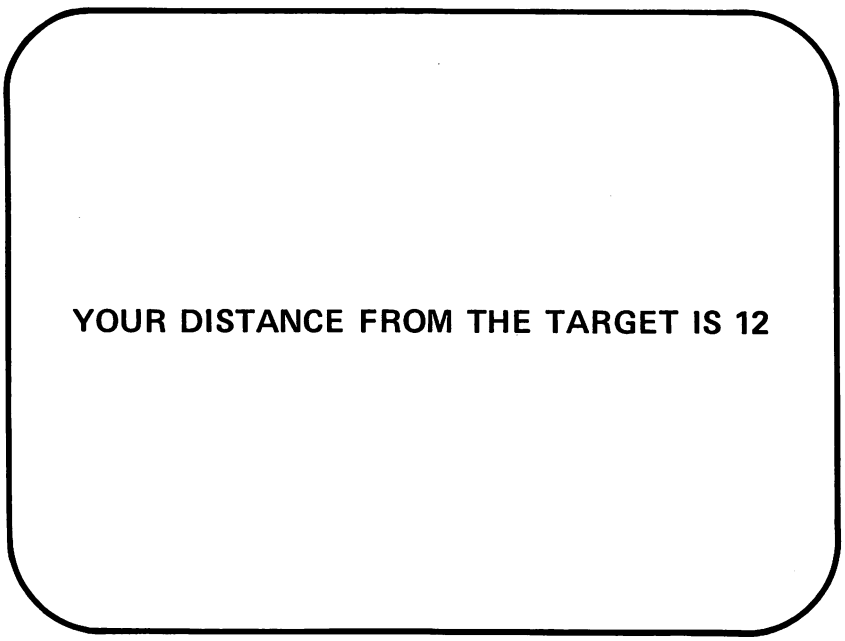


Figure 13-4.

```
450     KEY(14) ON
460     SECONDS = VAL(RIGHT$(TIME$,2))
470     IF SECONDS = 5 THEN 700 ELSE 370
480 'Cursor Up
490     GOSUB 640
500     PRINT CHR$(30);
510 RETURN
520 'Cursor Left
530     GOSUB 640
540     PRINT CHR$(29);
550 RETURN
560 'Cursor Right
570     GOSUB 640
580     PRINT CHR$(28);
590 RETURN
600 'Cursor Down
610     GOSUB 640
620     PRINT CHR$(31);
630 RETURN
640 'Turn off cursor motion trapping
650     KEY(11) OFF
660     KEY(12) OFF
670     KEY(13) OFF
680     KEY(14) OFF
690 RETURN
700 'Compute score
710    D = ABS(POS(0)-TARGCOL)+ABS(CSRLIN-TARGROW)
720    CLS
730    PRINT "YOUR DISTANCE FROM THE TARGET IS";D
740    IF D=0 THEN PRINT "CONGRATULATIONS"
750    IF D=0 THEN PRINT "YOU HIT THE TARGET!"
760    SC = 100
770    IF D>0 THEN SC=SC-10
780    IF D>2 THEN SC = SC-20
790    IF D>5 THEN SC = SC-20
800    IF D>10 THEN SC=SC-20
810    IF D>15 THEN SC = SC-20
820    IF D>20 THEN SC = SC-10
830    PRINT "YOUR SCORE IS",SC
840    INPUT "DO YOU WISH TO PLAY AGAIN(Y/N)";B$
850    IF B$ = "Y" OR B$="y" THEN 180 ELSE 860
860    END
```

Exercises (answer on page 373)

1. Experiment with the above program by making the time of target viewing shorter or longer than one second.
2. Experiment with the above program by making the time for target location shorter or longer than five seconds.
3. Modify the program to keep a running total score for a sequence of 10 games.

4. Modify the program to allow two players, keeping a running total score for a sequence of 10 games. At the end of ten games, the computer should announce the total scores and declare the winner.

13.3 Shooting Gallery

In this section, we develop a game called Shooting Gallery, which simulates the shooting galleries of carnivals. The player has a gun to fire at a moving target. (See Figure 13-5.) The program keeps track of the hits. The game shows 20 moving targets during one play.

The design of this game incorporates most of what we know. Let's begin by enabling event trapping of the cursor motion keys: up, down, right, and left. The right and left motions will tell the program that we wish to move the gun to the right or left. The cursor-up key will fire the gun. Lines 10-80 turn on the appropriate trapping.

This program will be in the medium-resolution graphics mode. The gun initially will be in the center of the last text row of the screen. The first position of the bullet after being fired will be in row 185. We will keep track of the horizontal position of the gun in the variable GUNPOSITION and the vertical and horizontal positions of the bullet in the variables BULLETROW and BULLETCOL, respectively. Line 90 initializes GUNPOSITION and BULLETROW.

For the gun, we will use the small house-shaped figure (ASCII character 127). The bullet will be a vertical arrow (ASCII character 24), and the target will be a happy face (ASCII character 2). All of these figures are to be animated, so

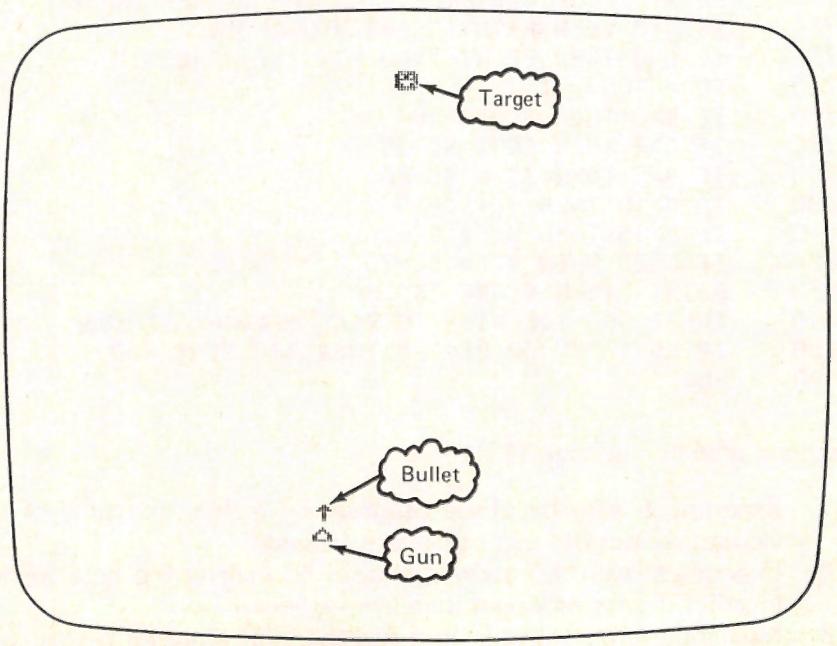


Figure 13-5. The Game of Shooting Gallery.

it is necessary to GET all of them in appropriate arrays A%, B%, and C%. This is done in lines 100-210. The use of % means that the arrays will contain integers. Limiting the type of number that the arrays can contain will speed up program execution. (This is a concern, since animations tend to run slowly in BASIC.)

Line 220 places the gun in its initial position. The main program is in lines 230-330. There is an outer loop for 20 targets and an inner loop, each step of which moves the target two columns across the screen and the bullet (if any have been fired) 8 rows up the screen. If you fire the gun (using the cursor-up key), the program is interrupted and the gun-firing routine is called. This displays the bullet in its initial position. All subsequent motion of the bullet is controlled by the main loop. The bullet disappears when it reaches the row of the target. The target disappears when it hits the right edge of the screen. If the bullet and the target are at the same place at the same time, both disappear and you are credited with a hit.

The BEEP command is used to sound the speaker when you score a hit. Also note the use of the function ABS in line 670. ABS(X) is just X with its sign removed. For example, ASB(+5) = 5, whereas ABS(−5) = 5.

```
10 'Initialization
20      KEY OFF
30      ON KEY(11) GOSUB 590
40      ON KEY(12) GOSUB 530
50      ON KEY(13) GOSUB 470
60      KEY(11) ON
70      KEY(12) ON
80      KEY(13) ON
90      GUNPOSITION=160:BULLETROW=185
100     DIM A%(100),B%(100),C%(100)
110     SCREEN 1
120     CLS
130     PRINT CHR$(2)
140     GET (0,0)-(7,7),A%
150     CLS
160     PRINT CHR$(127)
170     GET (0,0)-(7,7),B%
180     CLS
190     PRINT CHR$(24)
200     GET (0,0)-(7,7),C%
210     CLS
220     PUT (GUNPOSITION,185),B%
230 'Main program loop
240     FOR TARGET=1 TO 20
250             PUT (0,8),A%
260             FOR COLUMN=2 TO 312 STEP 2
270                     GOSUB 340:          'Move target
280                     GOSUB 380:          'Move bullet
290             NEXT COLUMN
300             IF COLUMN=316 THEN 320
310             PUT (312,8),A%
```

```
320     NEXT TARGET
330     END
340 'Move target
350     PUT (COLUMN-2,8),A%
360     PUT (COLUMN,8),A%
370 RETURN
380 'Move bullet
390     IF BFLAG=0 THEN 460
400     PUT (BULLETCOL,BULLETROW),C%
410     BULLETROW=BULLETROW-8
420     IF BULLETROW<10 THEN GOSUB 670 ELSE 450
430     BFLAG=0
440     GOTO 460
450     PUT (BULLETCOL,BULLETROW),C%
460 RETURN
470 'Move gun 8 steps to right
480     PUT (GUNPOSITION,185),B%
490     GUNPOSITION=GUNPOSITION+8
500     IF GUNPOSITION>311 THEN GUNPOSITION=311
510     PUT (GUNPOSITION,185),B%
520 RETURN
530 'Move gun 8 steps to left
540     PUT (GUNPOSITION,185),B%
550     GUNPOSITION=GUNPOSITION-8
560     IF GUNPOSITION<0 THEN GUNPOSITION=0
570     PUT (GUNPOSITION,185),B%
580 RETURN
590 'Shoot gun
600     IF BFLAG=1 THEN 650
610     BFLAG=1
620     BULLETCOL=GUNPOSITION
630     BULLETROW=177
640     PUT (BULLETCOL,BULLETROW),C%
650 RETURN
660 'Determine if target is hit
670     IF ABS(BULLETCOL-COLUMN)<7 THEN GOSUB 690
680 RETURN
690 'erase target and bullet
700     PUT (COLUMN,8),A%
710     BEEP
720     SCORE=SCORE+1
730     LOCATE 1,1
740     PRINT "SCORE";SCORE;" hits";
750     COLUMN=314
760 RETURN
```

Exercises (answers on page 373)

1. Run the above program to get a feel for its operation.
2. Modify the above program so that the bullet speed is increased by a factor of two. (This makes the game easier!)

3. Modify the above program so that the bullet speed is divided by a factor of two.
4. Modify the above program so that every fifth target is a sun (ASCII code 15). Modify the scoring so that hitting a sun counts for five hits.

13.4 Tic Tac Toe

In this section, we present a program for the traditional game of tic tac toe. We won't attempt to let the computer execute a strategy. Rather, we will let it be fairly stupid and choose its moves randomly. We will also use the random number generator to "flip" for the first move. Throughout the program, you will be "O" and the computer will be "X". Figures 13-6 through 13-9 illustrate a sample game.

```
LOAD "TICTAC"
READY
RUN
```

Figure 13-6.

TEST YOUR UNDERSTANDING 1 (answer on page 324)
How can the computer toss to see who goes first?

The computer draws a tic tac toe board in Figure 13-8.
The computer displays your move and makes a move of its own in Figure 13-9.
The computer will now make its move and so on until someone wins or a tie game results.

```
TIC TAC TOE
YOU WILL BE O;THE COMPUTER WILL BE X
THE POSITIONS OF THE BOARD ARE NUMBERED
AS FOLLOWS:

        1       2       3
        4       5       6
        7       8       9

THE COMPUTER WILL TOSS FOR FIRST.
YOU GO FIRST.
WHEN READY TO BEGIN TYPE 'R'
R
```

Figure 13-7.

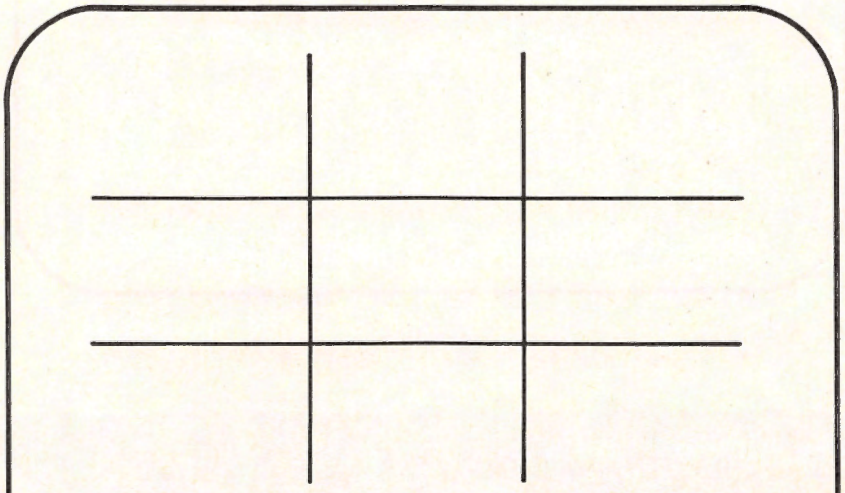


Figure 13-8.

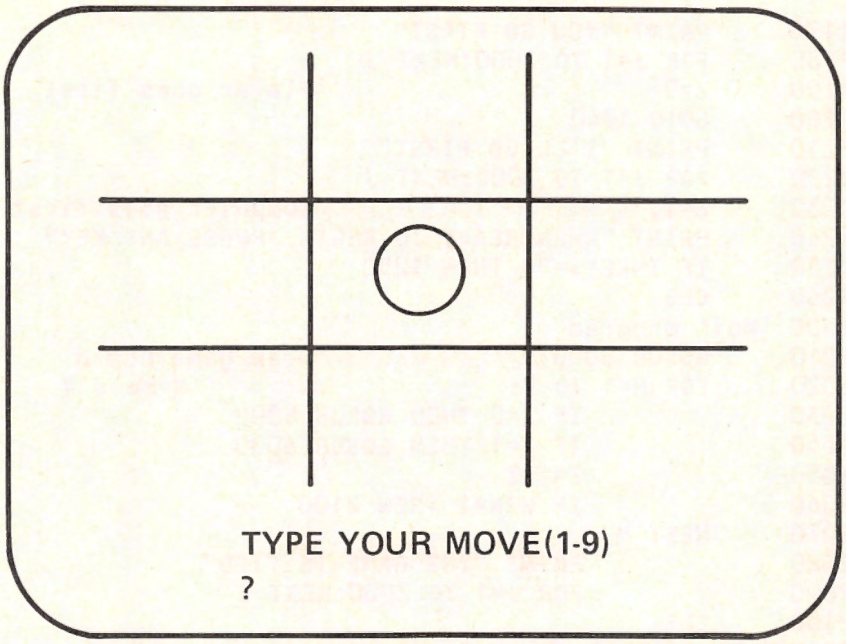


Figure 13-9.

Here are the variables used in the program:

Z = 0 if it's your move and Z=1 if it is the computer's.

A$(J) (J=1, 2, ..., 9) contains O, X, or the empty string, indicating the current status of position J.

S = the position of the current move.

M = the number of moves played (including the current one).

We used a video display worksheet to lay out the board, and to determine the coordinates for the lines and the X's and O's.

Here is a listing of our program.

```
1000 'Initialization
1010     CLEAR:KEY OFF
1020     SCREEN 1
1030     RANDOMIZE VAL(RIGHT$(TIME$,2))
1040     DIM A$(9)
1050     DIM B(9)
1060     CLS
1070     PRINT "TIC TAC TOE"
1080     PRINT "YOU WILL BE O; THE COMPUTER WILL BE X"
1090     PRINT "THE POSITIONS ON THE BOARD ARE NUMBERED"
1100     PRINT "AS FOLLOWS"
1110     PRINT "1";TAB(8) "2";TAB(16) "3"
1120     PRINT "4";TAB(8) "5";TAB(16) "6"
1130     PRINT "7";TAB(8) "8";TAB(16) "9"
1140     PRINT "THE COMPUTER WILL TOSS FOR FIRST"
1150     FOR J=1 TO 2000:NEXT J
1160     IF RND(1) > .5 THEN 1170 ELSE 1210
```

```
1170     PRINT "YOU GO FIRST"
1180     FOR J=1 TO 2000:NEXT J
1190     Z=0:                          'Player goes first
1200     GOTO 1240
1210     PRINT "I'LL GO FIRST"
1220     FOR J=1 TO 2000:NEXT J
1230     Z=1:                          'Computer goes first
1240     PRINT "WHEN READY TO BEGIN, PRESS ANY KEY"
1250     IF INKEY$="" THEN 1250
1260     CLS
2000 'Main program
2010     GOSUB 3000:                   'Draw game board
2020     FOR M=1 TO 9:                        'M=move #
2030             IF Z=0 THEN GOSUB 5000
2040             IF Z=1 THEN GOSUB 6000
2050             Z=1-Z
2060             IF WIN=1 THEN 2100
2070     NEXT M
2080             PRINT "THE GAME IS TIED"
2090             FOR J=1 TO 2000:NEXT J
2100     CLS
2110     LOCATE 1,1
2120     INPUT "ANOTHER GAME(Y/N)";R$
2130     IF R$="Y" OR R$="y" THEN 1060 ELSE END
3000 'Draw TIC TAC TOE Board
3010    CLS
3020    LINE (103,8)-(103,191)
3030    LINE (206,8)-(206,191)
3040    LINE (8,70)-(311,70)
3050    LINE (8,132)-(311,132)
3060    RETURN
4000 ' Display current game status
4010    LOCATE 5,7: PRINT A$(1);
4020    LOCATE 5,20: PRINT A$(2);
4030    LOCATE 5,33: PRINT A$(3);
4040    LOCATE 14,7: PRINT A$(4);
4050    LOCATE 14,20: PRINT A$(5);
4060    LOCATE 14,33: PRINT A$(6);
4070    LOCATE 21,7: PRINT A$(7);
4080    LOCATE 21,20: PRINT A$(8);
4090    LOCATE 21,33:PRINT A$(9);
4100 RETURN
5000 'Player's Move
5010             LOCATE 1,1
5020             INPUT "TYPE YOUR MOVE(1-9)";S
5030             IF A$(S) = "" THEN 5100
5040             LOCATE 1,1
5050             LINE (0,0)-(319,7),0,BF:'Blank out first row
5060             PRINT "ILLEGAL MOVE"
5070             FOR J=1 TO 2000:NEXT J
5080             GOTO 5000
```

```
5090            GOSUB 7000
5100            A$(S) = "O"
5110            GOSUB 7000:     'Is game over?
5120            LINE (0,0)-(319,7),0,BF:'Blank out first row
5130            GOSUB 4000:     'Display move
5140 RETURN
6000 'Computer's Move
6010    LOCATE 1,1
6020    PRINT "Here's my move!";
6030    GOTO 8000:              'Is there a winning move?
6040    'If not, choose random move
6050    S = INT(9*RND+1)
6060    IF A$(S) = "" THEN 6070 ELSE 6050
6070    A$(S) = "X"
6080    FOR J=1 TO 2000:NEXT J: 'Delay
6090    GOSUB 7000:             'Is game over?
6100    GOSUB 4000:             'Display move
6110 RETURN
7000 'Is the game over?
7010    IF Z = 0 THEN C$ = "O" ELSE C$ = "X"
7020    IF A$(1) = A$(2) THEN 7030 ELSE 7050
7030    IF A$(2) = A$(3) THEN 7040 ELSE 7050
7040    IF A$(3) = C$ THEN 7260
7050    IF A$(1) = A$(4) THEN 7060 ELSE 7080
7060    IF A$(4) = A$(7) THEN 7070 ELSE 7080
7070    IF A$(7) = C$ THEN 7260
7080    IF A$(1) = A$(5) THEN 7090 ELSE 7110
7090    IF A$(5) = A$(9) THEN 7100 ELSE 7110
7100    IF A$(9) = C$ THEN 7260
7110    IF A$(2) = A$(5) THEN 7120 ELSE 7140
7120    IF A$(5) = A$(8) THEN 7130 ELSE 7140
7130    IF A$(8) = C$ THEN 7260
7140    IF A$(3) = A$(6) THEN 7150 ELSE 7170
7150    IF A$(6) = A$(9) THEN 7160 ELSE 7170
7160    IF A$(9) = C$ THEN 7260
7170    IF A$(4) = A$(5) THEN 7180 ELSE 7200
7180    IF A$(5) = A$(6) THEN 7190 ELSE 7200
7190    IF A$(6) = C$ THEN 7260
7200    IF A$(7) = A$(8) THEN 7210 ELSE 7230
7210    IF A$(8) = A$(9) THEN 7220 ELSE 7230
7220    IF A$(9) = C$ THEN 7260
7230    IF A$(3) = A$(5) THEN 7240 ELSE 7320
7240    IF A$(5) = A$(7) THEN 7250 ELSE 7320
7250    IF A$(7) = C$ THEN 7260 ELSE 7320
7260    GOSUB 4000
7270    LOCATE 1,1
7280    PRINT SPACE$(80);
7290    LOCATE 1,1
7300    PRINT C$ , "WINS THIS ROUND":WIN=1
7310    FOR J=1 TO 2000:NEXT J
7320 RETURN
```

```
8000 'Look for a winning move
8010    COUNT = 0
8020    FOR I=1 TO 9
8030            IF A$(I) = "X" THEN B(I) = 1
8040            IF A$(I) = "" THEN B(I) = 0
8050            IF A$(I) = "O" THEN B(I) = -1
8060    NEXT I
8070    READ I,J,K
8080    COUNT = COUNT+1
8090    IF COUNT = 8 THEN 8180
8100    S = B(I)+B(J)+B(K)
8110    IF S = 2 THEN 8120 ELSE 8070
8120    IF B(J) = 0 THEN A$(J) = "X" ELSE 8140
8130    GOTO 8310
8140    IF B(K) = 0 THEN A$(K) = "X" ELSE 8160
8150    GOTO 8310
8160    IF B(I) = 0 THEN A$(I) = "X" ELSE 8070
8170    GOTO 8310
8180    RESTORE
8190    COUNT = 0
8200    READ I,J,K
8210    COUNT = COUNT + 1
8220    S = B(I)+B(J)+B(K)
8230    IF COUNT = 8 THEN 8320
8240    IF S=-2 THEN 8250 ELSE 8200
8250    IF B(J) = 0 THEN A$(J) = "X" ELSE 8270
8260    GOTO 8310
8270    IF B(K) = 0 THEN A$(K) = "X" ELSE 8290
8280    GOTO 8310
8290    A$(I) = "X"
8300    GOTO 8310
8310    RESTORE : GOTO 6080
8320    RESTORE:GOTO 6040
8330    DATA 1,2,3,4,5,6,7,8,9,1,4,7,2,5,8,3,6,9,1,5,9,3,5,7
```

Exercises

1. Modify the above program so that you and the computer may play a series of ten games. The computer should decide the champion of the series.
2. Modify the above program to play 4x4 tic tac toe.

ANSWER TO TEST YOUR UNDERSTANDING

1: See lines 120-170 of the tic tac toe program.

14

DIFFERENT KINDS OF NUMBERS IN BASIC

In this chapter we will discuss the various types of numbers used by BASIC and the library of "built-in" mathematical functions which you may use.

14.1 Single- and Double-Precision Numbers

Up to this point, we have used the computer to perform arithmetic without giving much thought to the level of accuracy of the numbers involved. However, when doing scientific programming, it is absolutely essential to know the number of decimal places of accuracy of the computations. Let's begin this chapter by discussing the form in which BASIC stores and uses numbers.

Actually, BASIC recognizes three different types of numeric constants: integer, single-precision, and double-precision.

An **integer constant** is an ordinary integer (positive or negative) in the range from -32768 to +32767. (32768 is 2 raised to the fifteenth power. This number is significant to the internal workings of the PCjr.) Here are some examples of integer numeric constants:

7, 58, 3712, -15, -598

Integer constants may be stored very efficiently in RAM. Moreover, arithmetic with integer constants takes the least time. Therefore, in order to realize these efficiencies, PCjr BASIC handles integer constants in a special way.

A **single-precision constant** is a number with seven or fewer digits, which is not an integer. Some examples of single-precision constants are:

5.135, -63.5785, 1234567, -1.467654E12

Note that a single-precision constant may be expressed in "scientific" or "floating point" notation, as in the final example shown here. In such an expression, however, you are limited to seven or fewer digits. In PCjr BASIC, single-precision constants must lie within these ranges: Between $-1x10^{38}$ and $-1x10^{-38}$; Between $1x10^{-38}$and $1x10^{38}$. This limitation seldom is much of a limitation in practice. After all, 1×10^{-38} equals:

.00000000000000000000000000000000000001

(37 zeros followed by a 1), which is about as small a number as you are ever likely to encounter! Similarly, 1×10^{38} equals:

100,000,000,000,000,000,000,000,000,000,000,000,000

(a 1 followed by 38 zeros), which is large enough for most practical calculations.

A **double-precision constant** is a number containing more than seven digits. Here are some examples of double-precision numbers:

2.0000000000, 3578930497594, −3946.635475495

Scientific notation also may be used to represent double-precision numbers: Use the letter D to precede the exponent. For example, the number:

2.7575757575D−4

equals the double-precision constant:

.00027575757575

The number:

1.3145926535D+15

equals the double-precision constant:

1,314,159,265,350,000

A double-precision constant may have up to 17 digits. Double-precision constants are subject to the same range limitations as single-precision constants.

Single-precision constants occupy more RAM than integer constants. Moreover, arithmetic with single-precision constants proceeds slower than integer arithmetic. Similarly, double-precision constants occupy even more memory, and arithmetic proceeds even slower than with single-precision constants. PCjr BASIC recognizes each of the three types of numerical constants and uses only as much arithmetic power as necessary.

Here are the rules for determining the type of a numerical constant:

1. Any integer in the range −32768 and 32767 is an integer constant.
2. Any number with seven or fewer digits that is not an integer constant is a single-precision constant. Any number in scientific notation using E before the exponent is assumed to be a single-precision constant. If a number has more than seven digits in scientific notation but uses an E, it will be interpreted as a double-precision constant. For example, the number:

1.23456789E15

will be interpreted as the double-precision constant:

1.23456789D

3. A number with more than seven digits will be interpreted as a double-precision constant. If more than 17 digits are specified, then the number will be truncated after the seventeenth digit and written in scientific notation. For example, the number:

123456789123456789

will be interpreted as the double-precision constant:

1.2345678912345678D17

The type of a numeric constant may be specified by means of a **type declaration tag**. For instance, a numeric constant followed by % will be interpreted as an integer constant. For example, 1% will be interpreted as the integer constant 1. A % sign in a number containing a decimal will be ignored. For example, the number:

1.85%

will be interpreted as the single-precision constant:

1.85

If the constant containing a % is too large to be an integer constant (that is, not in the range -32768 to +32767), an OVERFLOW error will occur. A numeric constant followed by ! will be interpreted as a single-precision constant and rounded accordingly. For example, the constant:

1.23456789!

will be interpreted as:

1.234567

The constant:

123456789!

will be truncated to seven significant digits and written in scientific notation as:

1.2345678E9

A # serves as a type declaration tag to indicate a double-precision constant. For example, the constant:

1.2#

will be interpreted as the 17-digit double-precision constant:

1.2000000000000000.

In scientific notation, the letter D serves as a type declaration tag.

TEST YOUR UNDERSTANDING 1 (answers on page 330)
Write out the decimal form of the following numbers:

a. −7.5%
b. 4.58923450183649E+12
c. 270D−2
d. 12.55#
e. −1.62!

A type declaration tag supersedes rules 1-3 in determining the type of a numeric constant.

Let's discuss the way BASIC performs arithmetic with the various constant types. The variable type resulting from an arithmetic operation is determined by the variable types of the data entering into the operation. For example, the sum of two integer constants will be an integer constant, provided that the answer is within the range of an integer constant. If not, the sum will be a single-precision constant. Arithmetic operations among single-precision constants will always yield single-precision constants. Arithmetic constants among double-precision constants will yield a double-precision result. Here are some examples of arithmetic:

5% + 7%

The computer will add the two integer constants 5 and 7 to obtain the integer constant 12.

4.21! + 5.2!

The computer will add the two single-precision constants 4.21 and 5.2 to obtain the single-precision result 9.41.

3/2

Here the two constants 3 and 2 are integers. However, since the result, 1.5, is not an integer, it is assumed to be a single-precision type.

The result of:

```
1/3
```

is the single-precision constant .3333334. Similarly, the result of the double-precision calculation:

```
1#/3#
```

is the double-precision constant .33333333333333333.

TEST YOUR UNDERSTANDING 2 (answers on page 330)
What result will the computer obtain for the following problems?

a. 2/5 + 1/3
b. .4% + .3333333333333333333%
c. .4# + .3333333333333333333#
d. .4! + .3333333333333333333!

It is important to realize that if a number does not have an exact decimal representation (such as 1/3 = .333...) or if the number has a decimal representation which has too many digits for the constant type being used, the computer then will be working with an approximation of the number rather than the number itself. The built-in errors caused by the approximations of the computer are called **round-off errors** Consider the problem of calculating:

```
1/3 + 1/3 + 1/3
```

As we have seen above, 1/3 is stored as the single-precision constant .3333334. The computer will form the sum as

```
.3333334 + .3333334 + .3333334 = 1.0000002
```

The sum has a round-off error of .0000002.

PCjr BASIC displays up to seven digits for a single-precision constant. Due to round-off error, the answer to any single arithmetic operation is guaranteed accurate to only six places, however. Double-precision constants are displayed rounded off to 16 digits. For a single arithmetic operation, the computer's design guarantees that a double-precision answer will be accurate to 16 digits. If you perform many such operations, it is possible that cumulative round-off error will make the sixteenth or earlier digits inaccurate.

Exercises(answers on page 374)

For each of the constants below, determine the number stored by the computer.

1. 3
2. 2.37

3. 5.78E5
4. 2#
5. 3!
6. -4.1!
7. -4.1%
8. 3500.6847586958658!
9. 2.176D2
10. -5.94E12
11. 3.58695040038372653 74
12. -234542383746.21
13. -2.367D20
14. 457000000000000000!

For each of the arithmetic problems below, determine the number as stored by the computer.

15. 1 + 45
16. 2/4
17. 3#/5#
18. 3!/5! + 1
19. 2#/3#
20. 2#/3# + .53#
21. 2/3
22. 2/3 + .53
23. .5E4 - .37E2
24. 1.75D3 - 1.0D-5
25. For each of exercises 15 through 24, determine how the computer will display the result.
26. Calculate 1/3 + 1/3 + 1/3 + ... + 1/3 (1000 1/3's) using single-precision constants. What answer is displayed? Is this answer accurate to six digits? If not, explain why.
27. Answer the same question as Exercise 26, but use double-precision constants and 17 digits.

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. -7.5
   b. 4,589,235,000,000
   c. 2.7000000000000000
   d. 12.550000000000000
   e. -1.620000
2: a. .7333333
   b. 0
   c. .73333333333333333
   d. .7333333

14.2 Variable Types

In the previous section we introduced the various types of numerical constants: integer, single-precision, and double-precision. There is a parallel set of types for variables.

A **variable of integer type** takes on values which are integer type constants. An integer type variable is indicated by the symbol % after the variable name. For example, here are some variables of integer type:

A%, BB%, A1%

In setting the value of an integer type variable, the computer will round any fractional parts to obtain an integer. For example, the instruction

```
10 A% = 2.54
```

will set the value of A equal to the integer constant 3. Integer type variables are useful when keeping track of integer quantities, such as line numbers in a program.

A **variable of single-precision type** is one whose value is a single-precision constant. A single-precision type variable is indicated by the symbol ! after the variable name. Here are some examples of single-precision variables:

K!, W7!, ZX!

In setting the value of a single-precision variable, all digits beyond the seventh are rounded. For example, the instruction

```
20 A! = 1.23456789
```

will set A! equal to 1.2345678.

If a variable is used without a type designator, the computer will then assume that it is a single-precision variable. All of the variables we have used until now have been single-precision variables. These are, by far, the most commonly used variables.

A **double-precision variable** is a variable whose value is a double-precision constant. Such variables are useful in computations where great numerical accuracy is required. A double-precision variable is indicated by the tag # after the variable name. Here are some examples of double-precision variables:

B#, C1#, EE#

In setting the values of double-precision variables, all digits after the seventeenth digit are rounded.

Note that the variables A%, A!, A#, and A$ are four distinct variables. You could, if you wish, use all of them in a single program. (But it probably would be very confusing and probably would produce errors if you did.)

TEST YOUR UNDERSTANDING 1 (answer on page 333)
What values are assigned to each of these variables?

a. A# = 1#
b. C% = 5.22%
c. BB! = 1387.5699

Using the type declaration tags %, !, and # is a nuisance since they must be included whenever the variable is used. There is a way around this tedium. The instructions **DEFINT**, **DEFSNG**, and **DEFDBL** may be used to define the types of variables for an entire program, so that type declaration tags need not be used. Consider the instruction

```
100 DEFINT A
```

It specifies that every variable which begins with the letter A (such as A, AB, or A1) should be considered as a variable of integer type. Here are two variations of this instruction:

```
200 DEFINT A,B,C
300 DEFINT A-G
```

Line 200 defines any variables beginning with A, B, or C to be of integer type. Line 300 defines any variables beginning with any of the letters A through G to be of integer type. The **DEFINT** instruction usually is used at the beginning of a program, so that the resulting definition is in effect throughout the program.

The instruction **DEFSNG** works exactly like **DEFINT** and is used to define certain variables to be single-precision. The instructions **DEFDBL** and **DEFSTR** work the same way for double-precision and string variables, respectively.

Note that type declaration tags override the DEF instructions. For example, suppose that the variable A was defined to be single-precision using a DEFSNG instruction at the beginning of the program. It would be legal to use A# as a double-precision variable, since the type declaration tag # would override the single-precision definition.

WARNING: Here is a mistake that is easy to make. Consider the following program:

```
10 LET A# = 1.7
20 PRINT A#
30 END
```

This program seems harmless enough. We set the double-precision variable A# to the value 1.7 and then display the result. You probably expect to see the display:

```
1.700000000000000
```

If you actually try it, the display will read:

```
1.700000047683716
```

What went wrong? Well, it has to do with the way the internal logic of the computer works and the way in which numbers are represented in binary notation. Without going into details, let us merely observe that the computer interprets 1.7 as a **single-precision constant**. When this single-precision constant is converted into a double-precision constant (an operation which makes use of the binary representation of 1.7), the result coincides in its first 16 digits with the number given above. Does this mean that we must worry about such craziness? Of course not! What we really should have done in the first place is to write

```
10 A# = 1.7#
```

The display then would be exactly as expected.

Exercises (answers on page 374)

Calculate the following quantities in single-precision arithmetic.

1. (5.87 + 3.85 - 12.07)/11.98
2. (15.1 + 11.9)^4/12.88
3. (32485 + 9826)/(321.5 - 87.6^2)
4. Rework Exercise 1 using double-precision arithmetic.
5. Rework Exercise 2 using double-precision arithmetic.
6. Rework Exercise 3 using double-precision arithmetic.
7. Write a program to determine the largest integer less than or equal to X, where the value of X is supplied in an INPUT statement.

Determine the value assigned to the variable in each of the following exercises.

8. A% = -5
9. A% = 4.8
10. A% = -11.2
11. A! = 1.78
12. A# = 1.78#
13. A! = 32.653426278374645237
14. A! = 4.25234544321E21
15. A! = -1.23456789E-32
16. A# = 3.283646493029273646434
17. A# = -5.74#

ANSWERS TO TEST YOUR UNDERSTANDING

1: a. 1.0000000000000000
   b. 5
   c. 1387.570

14.3 Mathematical Functions in BASIC

In performing scientific computations, it is often necessary to use a wide variety of mathematical functions, including the natural logarithm, the exponential, and the trigonometric functions. PCjr BASIC has a wide range of these functions "built-in." In this section we will describe these functions and their use.

All mathematical functions in BASIC work in a similar fashion. Each function is identified by a sequence of letters (**SIN** for sine, **LOG** for natural logarithm, and so forth). To evaluate a function at a number X, we write X in parentheses after the function name. For example, the natural logarithm of X is written LOG(X). The program uses the current value of the variable X and will calculate the natural logarithm of that value. For example, if X is currently 2, then the computer will calculate LOG(2).

Instead of the variable X, we may use any type of variable: integer, single-precision, or double-precision. We also may use numerical constants of any

type. For example, SIN(.435678889658595) asks for the sine of a double-precision numerical constant. Note that unless you take special precautions (see below), all BASIC functions return a single-precision result accurate to six digits. For example, the above value of the sine function will be computed as:

```
SIN(.435678889658595) = .422026
```

To obtain double-precision values for the various built-in functions, you must request BASIC using the /D option. That is, when you type BASICA to start BASIC, use a command line of the form:

```
BASICA /D
```

BASIC lets you calculate a function of any expression. Consider the expression X^2 + Y^2 -3*X. It is perfectly acceptable to call for calculations such as:

```
SIN(X^2 + Y^2 - 3*X)
```

The computer will first evaluate the expression X^2 + Y^2 - 3*X using the current values of the variables X and Y. For example, if X = 1 and Y = 4, then X^2 + Y^2 -3*X = $1^2 + 4^2 - 3*1 = 14$. So the above sine function will be calculated as SIN(14) = .9906074.

Trigonometric Functions

PCjr BASIC has the following trigonometric functions available:

SIN(X) = the sine of the angle X
COS(X) = the cosine of the angle X
TAN(X) = the tangent of the angle X

Here the angle X is expressed in terms of radian measure. In this measurement system, 360 degrees equal two times pi radians. Or one degree equals .017453 radians, and one radian equals 57.29578 degrees. If you want to calculate trigonometric functions with the angle X expressed in degrees, use these functions:

SIN(.017453*X)
COS(.017453*X)
TAN(.017453*X)

The three other trigonometric functions, SEC(X), CSC(X), and COT(X), may be computed from the formulas:

SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = SIN(X)/COS(X)

Here, as above, the angle X is in radians. To compute these trigonometric functions with the angle in degrees, replace X by:

```
.017453*X
```

PCjr BASIC only has one of the inverse trigonometric functions, namely the arctangent, denoted ATN(X). This function returns the angle whose tangent is X. The angle returned is expressed in radians. To compute the arctangent with the angle expressed in degrees, use the function:

```
57.29578*ATN(X)
```

TEST YOUR UNDERSTANDING 1 (answer on page 338)
Write a program which calculates sin 45°, cos 45°, and tan 45°.

Logarithmic and Exponential Functions

BASIC allows you to compute e^X using the exponential function:

```
EXP(X)
```

Furthermore, you may compute the natural logarithm of X via the function:

```
LOG(X)
```

You may calculate logarithms to base b using the formula:

```
LOG_b (X) = LOG(X)/LOG(b)
```

Example 1. Prepare a table of values of the natural logarithm function for values X = .01, .02, .03, ..., 100.00. Output the table on the printer.
Solution. Here is the desired program. Note that our table has two columns with a heading over each column.

```
10 LPRINT "X", "LOG(X)"
20 J = .01
30 LPRINT J, LOG(J)
40 IF J = 100.00 THEN END
50 J = J + .01
60 GOTO 30
100 END
```

TEST YOUR UNDERSTANDING 2 (answer on page 338)
Write a program which evaluates the function f(x) = sin x/ (log x + e^x) for x = .45 and x = .7.

Example 2. Carbon dating is a technique for calculating the age of ancient artifacts by measuring the amount of radioactive carbon-14 remaining in the artifact, as compared with the amount present if the artifact were manufactured

today. If r denotes the proportion of carbon-14 remaining, then the age A of the object is calculated from the formula

```
A = -(1/.00012)*LOG(r)
```

Suppose that a papyrus scroll contains 47% of the carbon-14 of a piece of papyrus just manufactured. Calculate the age of the scroll.

Solution. Here r = .47 so we use the above formula.

```
10 LET R = .47
20 LET A = -(1/.00012)*LOG(R)
30 PRINT "THE AGE OF THE PAPYRUS IS", A, "YEARS"
40 END
```

Powers

PCjr BASIC has a square root function, denoted SQR(X). As with all the functions considered so far, this function will accept any type of input and will output a single-precision constant. For example, the instruction

```
10 LET Y = SQR(2.00000000000000000)
```

sets Y equal to 1.414214.

Actually, the exponentiation procedure we learned in Chapter 4 will work equally well for fractional and decimal exponents, and, therefore, provides an alternate method for extracting square roots. Here is how to use it. Taking the square root of a number corresponds to raising the number to the 1/2 power. We may calculate the square root of X as:

```
X^(1/2)
```

Note that the square root function, SQR(X), operates with greater speed and is, therefore, preferred. The alternate method is more flexible, however. For instance, we may extract the cube root of X as:

```
X^(1/3)
```

or we may raise X to the 5.389 power, as follows:

```
X^5.389
```

Greatest Integer, Absolute Value, and Related Functions

Here are several extremely helpful functions. The greatest integer less than or equal to X is denoted **INT(X)** . For example, the largest integer less than or equal to 5.46789 is 5, so:

```
INT(5.46789) = 5
```

Similarly, the largest integer less than or equal to -3.4 is -4 (on the number line, -4 is the first integer to the left of -3.4). Therefore:

```
INT(-3.4) = -4
```

Note that for positive numbers, the INT function throws away the decimal part. For negative numbers, however, INT works a little differently. To throw away the decimal part of a number (positive OR negative), we use the function **FIX(X)** . For example:

```
FIX(5.46789) = 5
FIX(-3.4) = -3
```

The absolute value of X is denoted ABS(X). Recall that the absolute value of X is X itself if X is positive or 0, and is -X if X is negative. Thus:

```
ABS(9.23) = 9.23
   ABS(0) = 0
ABS(-4.1) = 4.1
```

Just as the absolute value of X "removes the sign" of X, the function SGN(X) throws away the number and leaves only the sign. For example:

```
  SGN(3.4) = +1
SGN(-5.62) = -1
```

Conversion Functions

PCjr BASIC includes functions for conversion of a number from one type to another. For example, to convert X to integer type, use the function **CINT(X)** . This function will round the decimal part of X. Note that the resulting constant must be in the integer range of -32768 to 32767 or an error will result.

To convert X to single-precision, use the function CSNG(X). If X is of integer type, then CSNG(X) will cause the appropriate number of zeros to be appended to the right of the decimal point to convert X to a single-precision number. If X is double-precision, then X will be rounded to seven digits.

To convert X to double-precision, use the function CDBL(X). This function appends the appropriate number of zeros to X to convert it to a double-precision number.

Exercises (answers on page 375)

Calculate the following quantities:

1. e^1.54
2. e^-2.376
3. log 58
4. log .0000975
5. sin 3.7
6. cos 45°

7. arctan 1
8. tan .682
9. arctan 2 (expressed in degrees)
10. $\log_{10}18.9$
11. Make a table of values of the exponential function exp(x) for x = -5.0, -4.9, ... , 0 , .1, ... , 5.0.
12. Evaluate the function

$$3x\text{^}(1/4)\log(5x) + \exp(-1.8x)\tan x$$

for x = 1.7, 3.1, 5.9, 7.8, 8.4, and 10.1.

13. Write a BASIC program to graph the function y = sin x for x from 0 to 6.28. Use an interval of .05 on the x-axis.
14. Write a BASIC program to graph the function y = ABS(x).
15. Write a program to calculate the fractional part of x. (The fractional part of x is the portion of x which lies to the right of the decimal point.)

ANSWERS TO TEST YOUR UNDERSTANDING

1:
```
10 A = .017453
20 PRINT SIN(45*A), COS(45*A), TAN(45*A)
30 END
```

2:
```
10 DATA .45,.7
20 FOR J=1 TO 2
30    READ A
40    PRINT SIN(A)/(LOG(A)+EXP(A))
50 NEXT J
60 END
```

14.4 Defining Your Own Functions

In mathematics, functions usually are defined by specifying one or more formulas. For instance, here are formulas which define the functions f(x), g(x), and h(x):

$$f(x) = (x^2\text{-}1)^{1/2}$$

$$g(x) = 3x^2\text{-}\ 5x\ \text{-}\ 15$$

$$h(x) = 1/(x\text{-}1)$$

Note that each function is named by a letter, namely f, g, and h, respectively. PCjr BASIC allows you to define functions like these and to use them by name throughout your program. To define a function, we use the DEF FN instruction. This instruction is used before the first use of the function in the program. For example, to define the function f(x) above, we could use the instruction:

```
10 DEF FNF(X) = (X^2 - 1)^(1/2)
```

To define the function g(x) above, we use the instruction:

```
20 DEF FNG(X) = 3*X^2 - 5*X - 15
```

Note that in each case, we use a letter (F or G) to identify the function. Suppose that we wish to calculate the value of the function G for = 12.5. Once the function has been defined, this calculation may be described to the computer as FNF(12.5). Such calculations may be used throughout the program and will save the effort of retyping the formula for the function in each instance.

You may use any valid variable name as a function name. For example, you may define a function INTEREST by the statement:

```
10 DEF FNINTEREST(X) = ...
```

Moreover, in defining a function, you may use other functions. For example, if FNF(X) and FNG(X) are as defined above, then we may define their product by the instruction:

```
30 DEF FNC(X) = FNF(X)*FNG(X)
```

All of the functions above were functions of a single variable. However, BASIC allows functions of several variables as well. They are defined using the same procedure as above. To define the function $A(X,Y,Z) = X^2 + Y^2 + Z^2$, use the instruction:

```
40 DEF FNA(X,Y,Z) = X^2 + Y^2 + Z^2
```

You may even let one of the variables be a string variable. Consider the following function:

```
50 DEF FNB(A$) = LEN(A$)
```

This function computes the length of the string A$.

Finally, functions may produce a string as a function value. The name for such a function must end in $. Consider the following function:

```
60 DEF FND$(A$,J) = LEFT(A$,J)
```

This function of the two variables A$ and J will compute the string consisting of the J leftmost characters of the string A$. For example, suppose that A$ = "computer" and J = 3. Then:

```
FND$(A$,J) = "com"
```

Exercises (answers on page 375)

Write instructions to define the following functions.

1. x^2 - 5x
2. 1/x - 3x
3. 5exp(-2x)
4. x log(x/2)
5. tan x / x
6. cos(2x) + 1
7. The string consisting of the right two characters of C$.
8. The string consisting of the four middle characters of A$ beginning with the Jth character.
9. The middle letter of the string B$. (Assume that B$ has an odd number of characters.)
10. Write a program which tabulates the value of the function in Exercise 3 for x = 0, .1, .2, .3, .4, ... , 10.0.

15

COMPUTER-GENERATED SIMULATIONS

15.1 Simulation

Simulation is a powerful analysis tool which lets you use your computer to perform experiments to solve a wide variety of problems which might be too difficult to solve otherwise.

To describe what simulation is, let us use a concrete example. Assume that you own a computer software store. At the moment, you have only one salesperson behind the counter, but you are considering adding a second. Your question is: Should you hire the extra person? Being an analytical person, you have collected the following data. Traffic through your store varies by the hour. However, you have kept a log for the past month and are able to estimate the average number of potential customers arriving in the shop according to the following table:

9-10	10
10-11	15
11-12	15
12-1 pm	40
1-2	30
2-3	10
3-4	10
4-5	8
5-6	25
6-7	50
7-8	45
8-9	30

You have observed that you are currently paying a penalty for not having a second salesperson: If there is too long a wait, a customer will go somewhere else to buy software. A certain percentage of people entering the shop will leave, depending on the size of the line. Here are the results of your observations:

line size	fraction leaving
0	0
1	.2
2	.2
3	.3

4	.3
5	.4
6	.4
7	.4
8	.5
9	.6
10	.65
11	.7
12	.75
13	.75
14	.75
15 or more	.75

The average time to wait on a person is two minutes and the size of the average purchase is $30. The cost of hiring the new salesperson is $300 per week. Assuming that the salesperson works continuously while the shop is open, what action should you take?

This problem is fairly typical of the problems which arise in business. It is characterized by data accumulated from observations and unpredictable events (When will a given customer arrive? Will he or she encounter a long line? Will he or she be the impatient sort who walks out?). Nevertheless, you must make a decision based on the data you have. How should you proceed?

One technique is to let your computer "imitate" your store. Let the computer play a game which consists of generating customers at random times. These customers enter the "shop" and, on the basis of the current line, decide whether or not to stay. The computer will keep track of the line, the number of customers who leave, the revenue generated, and the revenue lost. The computer will keep up the simulated traffic for an entire "day" and present you with the results of the daily activity. But, you might argue, the computer data may not be valid. Suppose that it generates a "non-typical" day. Its data might be biased. This could, indeed, happen. In order to avoid this pitfall, we run the program for many simulated days and average the results. The process we have just described is called **simulation** .

In this chapter we will provide a glimpse of the power of simulation and provide you with enough of an idea to build simple simulations of your own.

First, let us handle some of the mathematical ideas we will need in the next section. The required notions center around the following question: How do we make the computer imitate an unpredictable event? Consider the irate customer who arrives to encounter a line of four people ahead of him. According to the above table, the customer will leave 30 percent of the time and remain in line 70 percent of the time. How do you let the computer make the decision for the customer?

Easy! Use the random number generator. Recall that RND generates a random number between 0 (included) and 1 (excluded). Suppose we ask how often RND is larger than .30. If, indeed, the numbers produced by the random number generator show no biases, approximately 70 percent of the numbers produced will lie in the given interval since this interval occupies 70 percent of the length of the interval from 0 to 1. We let our customer decide as follows: If

RND > .30 then the customer joins the line; otherwise, the customer walks out in a huff. We will employ this simple idea several times in designing our simulation.

15.2 Simulation of a Computer Store

Let us build a simulation to solve the problem stated in the preceding section. We must decide on techniques for imitating each of the important aspects of the problem.

Since the problem calls for analysis of actions as time passes, we must somehow measure the passage of (simulated) time. To do this, we will use the variables HOUR and MINUTES to keep track of the current time. In order to avoid a problem with am and pm, let's use the military time system. In this system the pm hours are denoted as 13 through 24. For example, 1:15 pm is shown as 13:15. As our unit of simulated time, let's use four minutes, the time it takes to serve a customer. Our program will then look at time in four-minute segments. During each four-minute segment, it will take certain actions and then advance to the next time segment by adjusting HOUR and MINUTES.

Let us store the statistical data on customer arrivals in the array TRAFFIC(J) (J = 9, 10, ..., 20). TRAFFIC(9) will equal the number of customers arriving between 9 and 10 am, TRAFFIC(10) the number arriving between 10 and 11 am, ... , TRAFFIC(20), the number arriving between 8 and 9 pm. The first action of the program is to set up the array TRAFFIC().

The next step will be to read in the customer "impatience data." Let WALKOUT(K) be the percentage of customers who leave when the line is K people long. We next read the given data into this array.

Our program will keep track of the following variables:

CASHFLOW = total sales for the day
CUSTOMERSERVED = number of customers served for day
LOSTCASH = cash lost to impatient customers
MAXLINE = maximum line length during the day

At the beginning of each hour, the program will schedule the arrival of the customers. For the Jth hour, it will schedule the arrival of TRAFFIC(J) customers. Each customer will be given a time of arrival in minutes past the hour. The computer will choose this arrival time using the random number generator. In the absence of any other information, let's assume that the customers spread themselves out in a random but uniform manner over the hour. The way we'll handle things inside the computer is as follows. At the beginning of each simulated hour, we set up an array CUST(SEG) with 15 entries, one for every four-minute segment in the hour. This array will indicate how many customers arrive in each four-minute interval. For example, if CUST(10)=4, then four customers will arrive between 36 and 40 minutes past the hour (that is, in the tenth four-minute interval in the hour). The program will randomly place each of the TRAFFIC(J) customers in four-minute intervals using the random number generator.

The program progresses through the simulated hour in four-minute segments. For the Tth four-minute segment, it causes CUST(T) customers to

arrive at the store. The computer lets these customers each look at the line and decide whether to leave or stay. If a customer decides to stay, then he or she is added to the line. If the customer decides to go, the computer makes a note of the $30 cash flow lost. After the customers are either in line or have left, the salesperson services two customers (remember, two customers are serviced every four minutes) and $30 is added to the cash flow. Finally, the time is updated and the entire procedure is repeated for the next four-minute segment. Let's be rather hard-hearted. If there are any customers left in line at closing time, we don't wait on them and add their business to that lost. This rather odd way of doing business is appropriate since we are analyzing the need for more personnel, and any overtime should be considered in that analysis.

Here is our program.

```
10 'Initialization
20    DIM TRAFFIC(20),WALKOUT(15),CUST(15)
30    RANDOMIZE VAL(RIGHT$(TIME$,2))
40   'Read arrival data
50      DATA 10,15,15,40,30,10,10,8,25,50,45,30
60      FOR HOUR=9 TO 20
70           READ TRAFFIC(HOUR)
80      NEXT HOUR
90    'Read walkout data
100     DATA .2,.2,.3,.3,.4,.4,.4,.5,.6,.65,.7,.75,.75,
            .75,.75
110     FOR LNE=1 TO 15
120             READ WALKOUT(LNE)
130     NEXT LNE
140   'Initialize Variables
150     LNE=0
160     MAXLINE = 0
170     LOSTCASH = 0
180     CASHFLOW = 0
190     CUSTSERVED = 0
200   '*************Main Program**************
210   CLS:PRINT "SIMULATING. PLEASE WAIT."
220   FOR HOUR = 9 TO 20
230     FOR MINUTE = 0 TO 56 STEP 4
240        'Update clock
250         IF MINUTES = 0 THEN GOSUB 570: 'Plan hour
260         SEG = MINUTES/4 + 1
270        'Simulate arrivals for current 4 minute segment
280         FOR J=1 TO CUST(SEG)
290             GOSUB 390
300         NEXT J
310        'Simulate customers served
320         GOSUB 490
330     NEXT MINUTE
340   NEXT HOUR
350   'Compute daily statistics
```

```
360       GOSUB 660
370 END
380 '**************Subroutines**************
390 'Arrival of one customer
400    IF LNE > 15 THEN L=15 ELSE L=LNE
410    IF RND>WALKOUT(L) THEN 420 ELSE 460
420    'Customer stays
430            LNE=LNE+1
440            IF LNE>MAXLINE THEN MAXLINE=LNE
450            GOTO 480
460    'Customer leaves
470            LOSTCASH=LOSTCASH+30
480 RETURN
490 'Wait on Customers
500 FOR J=1 TO 2
510            IF LNE=0 THEN 550
520     LNE=LNE-1
530     CASHFLOW=CASHFLOW+30
540     CUSTSERVED=CUSTSERVED+1
550 NEXT J
560    RETURN
570 'Plan Customer Arrivals for Next Hour
580    FOR SEGMENT=1 TO 15
590            CUST(SEGMENT) = 0
600    NEXT SEGMENT
610    FOR I=1 TO TRAFFIC(HOUR)
620        X = INT(15*RND)+1
630        CUST(X) = CUST(X) + 1
640    NEXT I
650 RETURN
660 'Print Summary of Day
670     CLS
680     PRINT
690     PRINT "CASH FLOW"; TAB(30) CASHFLOW
700     PRINT "CUSTOMERS SERVED"; TAB(30) CUSTSERVED
710     PRINT "CUSTOMERS NOT SERVED";
        TAB(30) 288-CUSTSERVED
720     PRINT "LOST CASH"; TAB(30) LOSTCASH
730     PRINT "MAXIMUM LINE"; TAB(30) MAXLINE
740 RETURN
```

Here are the results of four runs of the program:

RUN #1:

```
CASH FLOW              6900
CUSTOMERS SERVED        230
CUSTOMERS NOT SERVED    58
LOST CASH              1620
MAXIMUM LINE            9
```

RUN #2:

```
CASH FLOW               6780
CUSTOMERS SERVED         226
CUSTOMERS NOT SERVED     62
LOST CASH               2160
MAXIMUM LINE             9
```

RUN #3:

```
CASH FLOW               7020
CUSTOMERS SERVED         234
CUSTOMERS NOT SERVED     54
LOST CASH               2040
MAXIMUM LINE             13
```

RUN #4:

```
CASH FLOW               7320
CUSTOMERS SERVED         244
CUSTOMERS NOT SERVED     44
LOST CASH               2070
MAXIMUM LINE             10
```

We observe several interesting facts about the output. First note that the runs are not all identical. This is because the RANDOMIZE instruction creates new random customer arrival patterns for each run. Second, note the small percentage error in the data from the various runs. We seem to have discovered a statistical pattern which persists from run to run.

Finally, and most significantly, note that we are losing several thousand dollars per day in business because of our inability to service customers. At $300 per week, the additional salesperson is a bargain! Even a single day's lost sales is enough to pay the salary. It appears that we should add the extra salesperson. Actually, a bit more caution is advisable. We were dealing with cash flow rather than profit. In order to make a final decision, we must compute the profit generated by the additional sales. For example, if our profit margin is 40 percent then the profit generated by the extra sales will clearly amount to more than $300 per week and the extra salesperson should be hired.

The above example is fairly typical of the way in which simulation may be applied to analyze even fairly complicated situations in a small business. We will present some further refinements in the exercises.

Exercises

1. Run the above program for ten consecutive runs and record the data. Does your data come close to the data presented above? (Remember: Due to the RANDOMIZE instruction, you cannot expect to duplicate the given results exactly, only within statistical error.)

2. Suppose that customers become more impatient, and the likelihood of leaving is doubled in each case. Rerun the experiment to determine the lost cash flow in this case.
3. Suppose that customers become more patient, and the likelihood of leaving is cut in half in each case. Redo the experiment to determine the lost cash flow in this case.
4. Consider the original set of experimental data. Now assume that the second salesperson has been hired. Rerun the experiment to determine the average lost cash flow and the average line at closing.
5. Modify the given program so that you may calculate the average waiting time for each customer.

ANSWERS TO SELECTED EXERCISES

CHAPTER 3

Section 3.1 (page 39)

1. Valid
2. Valid
3. > is an illegal character
4. The colon is an illegal character
5. This is legal (believe it or not!)
6. Use only one period
7. Extensions may only have 3 characters.
8. Names may only have 8 characters.

Section 3.2 (page 41)

1. `COM1:ALICE.3`
2. `LPT1:MESSAGE`
3. `A:*.*`
4. `*.*`
5. `A:?.*`
6. `A:RALPH.*`

Section 3.4 (page 46)

1. `COPY B:TEST LPT1:`
2. `COPY A:*.COM B:`
3. `ERASE A:TEST`
4. `COPY A:TEST B:TEST3`
5. `COPY A:D???????.* B:`

CHAPTER 4

Section 4.2 (page 61)

1. Locates the cursor in row 3 and column 4.
2. Locates the cursor in row 12 and column 8.
3. Locates the cursor in row x and column y.
4. Displays 18.
5. Displays 18.
6. Displays the product of x and y.
7. Displays 2.
8. Displays 0.
9. Displays 2.
10. Displays the integer quotient of x divided by y.

Section 4.3 (page 69)

1.
```
10 PRINT 57+23+48
20 END
```
2.
```
10 PRINT 57.83*(48.27 -12.54)
20 END
```
3.
```
10 PRINT 127.86/38
20 END
```
4.
```
10 PRINT 365/.005 + 1.02^5
20 END
```
5. 2.3E7 6. 1.7525E2
7. -2E+08 8. 1.4E-04 9. -2.75E-10 10. 5.342E+16
11. 159,000
12. -20,345,600 13. -.000000000007456
14. .00000000000000000239456 15. 3 16. 2 17. 1
18. 1 19. 1

Section 4.5 (page 76)

1.
```
10 PRINT 2^1,2^2,2^3,2^4
20 PRINT 3^1,3^2,3^3,3^4
30 PRINT 4^1,4^2,4^3,4^4
40 PRINT 5^1,5^2,5^3,5^4
50 PRINT 6^1,6^2,6^3,6^4
60 END
```
2.
```
10 PRINT "CAST REMOVAL",45
20 PRINT "THERAPY",35
30 PRINT "DRUGS",5
40 PRINT
50 PRINT "TOTAL",45+35+5
60 PRINT "MAJ MED", .8*(45+35+5)
70 PRINT "BALANCE", .2*(45+35+5)
80 END
```
3.
```
10 PRINT "THACKER", 698+732+129+487
20 PRINT "HOVING", 148+928+246+201
30 PRINT "WEATHERBY",379+1087+148+641
40 PRINT "TOTAL VOTES", 698+732+129+487+148+928+
   246+201+379+1087+148+641
50 END
```
Note that line 40 extends over two lines of the screen. To type such a line just keep typing and do not press ENTER until you are done with the line. The maximum line length is 255 characters.
4. -2
5.
```
 SILVER     GOLD     COPPER     PLATINUM
  327        448       1052          2
```
6.
```
          GROCERIES   MEAT   DRUGS
MON         1,245     2,348  2,531
TUE           248     3,459  2,148
```

Section 4.6 (page 85)

1. 10 2. 0 3. 50 4. 9 -7 18
5. JOHN JONES AGE 38 6. 22 57 7. A can only assume numeric constants as values.
8. Nothing 9. A$ can only assume string constants as values. 10. No line number. String constant not in quotes.
11. Nothing 12. A variable name must begin with a letter.
13.
```
10 LET A=2.3758:B=4.58321:C=58.11
20 PRINT A+B+C
30 PRINT A*B*C
40 PRINT A^2+B^2+C^2
50 END
```
14.
```
10 LET A$="Office Supplies":B$="Computers":C$="Newsletters"
20 LET RA=346712:RB=459321:RC=376872
30 LET EA=176894:EB=584837:EC=402195
40 PRINT ,A$,B$,C$
50 PRINT "REVENUE",RA,RB,RC
60 PRINT "EXPENSES",EA,EB,EC
70 LET PA=RA-EA:PB=RB-EB:PC=RC-EC
80 PRINT "PROFIT",PA,PB,PC
90 PRINT
100 PRINT "TOTAL PROFIT",PA+PB+PB
```

Section 4.7 (page 93)

2. Type CLS and press ENTER. Type LIST and press ENTER.
3. Type SAVE "PROGRAM1" and press ENTER. Type NEW and press ENTER.
4. Type LOAD "PROGRAM1" and press ENTER. Type LIST and press ENTER.
5. Type the new line and press ENTER.
6. Type SAVE "PROGRAM2" and press ENTER.
7. Type LOAD "PROGRAM2" and press ENTER. Type DELETE 20 and press ENTER. Type RUN and press ENTER.
8. Type RENUM 100,,100 and press ENTER.
9. Type RENUM 2000,20,5 and press ENTER.

CHAPTER 5

Section 5.1 (page 109)

1.
```
10 S=0
20 FOR J=1 TO 25
30     S=S+J^2
40 NEXT J
```

```
   50 PRINT S
   60 END
2. 10 S=0
   20 FOR J=0 TO 10
   30     S=S+(1/2)^J
   40 NEXT J
   50 PRINT S
   60 END
3. 10 S=0
   20 FOR J=1 TO 10
   30     S=S+J^3
   40 NEXT J
   50 PRINT S
   60 END
4. 10 S=0
   20 FOR J=1 TO 100
   30     S=S+1/J
   40 NEXT J
   50 PRINT S
   60 END
5. 10 PRINT "N","N^2","N^3","N^4"
   20 FOR N=1 TO 12
   30     PRINT N,N^2,N^3,N^4
   40 NEXT N
   50 END
6. 10 PRINT "MONTH","INTEREST","BALANCE"
   20 B=4000,P=125.33
   30 FOR J=1 TO 12
   40     I=.01*B:'I=THE INTEREST FOR MONTH
   50     R=P-I:'R=REDUCTION IN BALANCE FOR MONTH
   60     B=B-R: NEW BALANCE
   70     PRINT J,I,B
   80 NEXT J
   90 END
7. 10 PRINT "END OF YEAR", "BALANCE"
   20 B=1000
   30 FOR J=1 TO 15
   40     B=B+1000+.10*B :'ADD DEPOSIT AND INTEREST
   50     PRINT J,B
   60 NEXT J
   70 END
8. 10 S=3.5E7: P = 5.54E6
   20 PRINT "END OF YEAR", "SALES", "PROFITS"
   30 FOR J=1 TO 3
   40     S = 1.2*S: P = 1.3*P
   50     PRINT J,S,P
   60 NEXT J
   70 END
```

Section 5.2 (page 123)

1.
```
10 J=1
20 IF J^2 >= 45000 THEN 100 ELSE 30
30 PRINT J,J^2
40 J=J+1
50 GOTO 20
100 END
```

2.
```
10 PI=3.14159
20 R=1
30 IF PI*R^2 <= 5000 THEN 40 ELSE 100
40 PRINT R,PI*R^2
50 R=R+1
60 GOTO 30
100 END
```

3.
```
10 PRINT "SIDE OF CUBE","VOLUME"
20 S=1
30 V = S^3
40 IF V <175000 THEN 50 ELSE 100
50 PRINT S,V
60 S=S+1
70 GOTO 30
100 END
```

4.
```
10 FOR J = 1 TO 10 : 'LOOP TO GIVE 10 PROBLEMS
20      INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30      INPUT "WHAT IS THEIR PRODUCT";C
40      IF A * B = C THEN 200
50      PRINT "SORRY. THE CORRECT ANSWER IS",A*B
60      GOTO 500 : 'GO TO THE NEXT PROBLEM
200     PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210     LET R = R+1 : 'INCREASE SCORE BY 1
220     GOTO 500 : 'GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

5.
```
10 FOR J = 1 TO 10 : 'LOOP TO GIVE 10 PROBLEMS
15      PRINT "CHOOSE OPERATION TO BE TESTED:"
16      PRINT "ADDITION (A), SUBTRACTION (S), OR
        MULTIPLICATION (M)"
17      INPUT A$
20      INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
21      IF A$ = "A" THEN 30
22      IF A$ = "S" THEN 130
23      IF A$ = "M" THEN 230
30      INPUT "WHAT IS THEIR SUM";C
40      IF A + B = C THEN 400
50      PRINT "SORRY. THE CORRECT ANSWER IS",A+B
```

```
60      GOTO 500 : 'GO TO THE NEXT PROBLEM
130     INPUT "WHAT IS THEIR DIFFERENCE";C
140     IF A-B = C THEN 400
150     PRINT "SORRY. THE CORRECT ANSWER IS",A-B
160     GOTO 500 : 'GO TO THE NEXT PROBLEM
230     INPUT "WHAT IS THEIR PRODUCT";C
240     IF A*B = C THEN 400
250     PRINT "SORRY. THE CORRECT ANSWER IS",A+B
260     GOTO 500 : 'GO TO THE NEXT PROBLEM
400     PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
410     LET R = R+1 : 'INCREASE SCORE BY 1
420     GO TO 500 : 'GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

6. See the answer to Exercise 8.
7. See the answer to Exercise 9.
8.
```
10 INPUT "NUMBER OF NUMBERS";N
20 FOR J=1 TO N
30     INPUT A
40     IF J=1 THEN B=A
50     IF A>B THEN B=A
60 NEXT J
70 PRINT "THE LARGEST NUMBER INPUT IS",B
80 END
```
9. Replace line 50 in Exercise 8. by:
```
50 IF A<B THEN B=A
```
10.
```
10 A0=5782:A1=6548:B0=4811:B1=6129:C0=3865:C1=4270
20 D0=7950:D1=8137:E0=4781:E1=4248:F0=6598:F1=7048
30 FOR J=1 TO 6
40    IF J=1 THEN A=A0:B=A1
50    IF J=2 THEN A=B0:B=B1
60    IF J=3 THEN A=C0:B=C1
70    IF J=4 THEN A=D0:B=D1
80    IF J=5 THEN A=E0:B=E1
90    IF J=6 THEN A=F0:B=F1
100   I = B-A
110   IF I>0 THEN PRINT "CITY",J,"HAD AN INCREASE OF", I
120   GOTO 200
130   IF I<0 THEN PRINT "CITY",J,"HAD A DECREASE OF",A-B
140   GOTO 300
200   IF I>500 THEN PRINT "CITY",J,"MORE THAN 500 INCREASE"
300 NEXT J
400 END
```
11.
```
10 PRINT "THIS PROGRAM SIMULATES A CASH REGISTER"
20 PRINT "AT THE QUESTION MARKS, TYPE IN THE PURCHASE"
30 PRINT "AM'TS. TYPE -1 TO INDICATE THE END OF THE ORDER"
```

```
40 INPUT "TYPE 'Y' IF READY TO BEGIN"; A$
50 IF A$ = "Y" THEN 60 ELSE 10
60 CLS
70 INPUT "ITEM"; A
80 IF A = -1 THEN 200 ELSE 90
90 T = T+A: T IS THE RUNNING TOTAL
100 GOTO 70
200 PRINT "THE TOTAL IS", T
210 S = .05*T:'S=SALES TAX
220 PRINT "SALES TAX", S
230 PRINT "TOTAL DUE", S+T
240 INPUT "PAYMENT GIVEN";P
250 PRINT "CHANGE DUE", P-(S+T)
300 END
```

12.
```
10 INPUT "CASH ON HAND"; C1
20 PRINT "INPUT ACCOUNTS EXPECTED TO BE RECEIVED IN NEXT
   MONTH."
30 PRINT "TO INDICATE END OF ACCOUNTS TYPE -1."
40 INPUT "ACCOUNTS RECEIVABLE";A
50 IF A=-1 THEN 100
60 C2 = C2+A:'C2=RUNNING TOTAL OF ACCOUNTS RECEIVABLE
70 GOTO 40
100 PRINT "INPUT ACCOUNTS EXPECTED TO BE PAID IN NEXT
    MONTH."
110 PRINT "TO INDICATE END OF ACCOUNTS TYPE -1."
120 INPUT "ACCOUNTS PAYABLE";A
130 IF A=-1 THEN 200
140 C3 = C3+A:'C3=RUNNING TOTAL OF ACCOUNTS PAYABLE
150 GOTO 120
200 PRINT "CASH ON HAND",C1
220 PRINT "ACCOUNTS RECEIVABLE",C2
230 PRINT "ACCOUNTS PAYABLE",C3
240 PRINT "NET CASH FLOW", C1+C2-C3
300 END
```

Section 5.4 (page 133)

1.
```
1000 ' 10 Asterisks
1010   LOCATE L,1
1020   PRINT "**********"
1030   RETURN
```

2.
```
1000 ' M Asterisks
1010   LOCATE L,1
1020   FOR J=1 TO M
1030       PRINT "*";
1040   NEXT J
1050   RETURN
```

3.
```
1000 ' M Asterisks; General Position
1010   LOCATE L,K
1020   FOR J=1 TO M
1030       PRINT "*";
1040   NEXT J
1050   RETURN
```
4.
```
10 K=5:L=3:M=30
20 GOSUB 1000
30 K=4:L=5:M=35
40 GOSUB 1000
50 K=8:L=7:M=12
60 GOSUB 1000
70 END
1000 ' M Asterisks; General Position
1010   LOCATE L,K
1020   FOR J=1 TO M
1030       PRINT "*";
1040   NEXT J
1050   RETURN
```
5. The program execution goes to line:
(a) 200 (b) 500 (c) The line after 10 (d) The line after 10 (e) Error. Program terminates.
6. Lines 200 and 50.

CHAPTER 6

Section 6.1 (page 141)

1. `DIM A(5)` 2. `DIM A(2,3)` 3. `DIM A$(3)`
4. `DIM A(1,3)`
5. `DIM A$(4),B(4)`
6.
```
10  DIM A$(3),B(3,3),C$(3)
20 PRINT ,,"Receipts"
30 C$(1) = "Store #1":C$(2) = "Store #2": C$(3) = "Store
   #3"
40 A$(1) = "1/1-1/10":A$(2) = "1/11-1/20":A$(3)="1/21-1/31"
50 B(1,1)= 57385.48:B(1,2)=89485.45:B(1,3)=38456.90
60 B(2,1)=39485.98:B(2,2)= 76485.49:B(2,3)=40387.86
70 B(3,1)=45467.21:B(3,2)=71494.25:B(3,3)=37983.38
100 PRINT ,C$(1),C$(2),C$(3)
200 FOR J=1 TO 3
220     PRINT A$(J),B(J,1),B(J,2),B(J,3)
230 NEXT J
300 END
```
7. Add these instructions:
```
5 DIM D(3)
240 FOR J=1 TO 3
```

```
250     D(J) = B(1,J)+B(2,J)+B(3,J)
260 NEXT J
270 PRINT "TOTALS",D(1),D(2),D(3)
```

8. Move the END to 400 and add the following instructions.

```
6 DIM E(3)
300 FOR J=1 TO 3
310     E(J) = B(J,1)+B(J,2)+B(J,3)
320 NEXT J
330 PRINT
340 PRINT "PERIOD", "TOTAL SALES"
350 FOR J=1 TO 3
360     PRINT A$(J) , E(J)
370 NEXT J
400 END
```

9.

```
10 DIM A$(4), B$(5), C(5,4)
20 A$(1) = "Store #1",A$(2) = "Store #2", A$(3) = "Store
   #3"
21 A$(4) = "STORE #4"
30 B$(1) = "REFRIG.",B$(2)="STOVE",B$(3)="AIR COND."
40 B$(4) = "VACUUM", B$(5) = "DISPOSAL"
50 PRINT "INPUT THE CURRENT INVENTORY"
60 FOR J=1 TO 4
70      PRINT A$(J)
80      PRINT
90      FOR I=1 TO 5
100          PRINT B$(I)
110          INPUT C(I,J)
120     NEXT I
130 NEXT J
200 REM REST OF PROGRAM IS FOR INVENTORY UPDATE
210 PRINT "CHOOSE ONE OF THE FOLLOWING"
220 PRINT "RECORD SHIPMENTS(R) "
230 PRINT "DISPLAY CURRENT INVENTORY(D)"
240 INPUT "TYPE R OR D";D$
250 IF D$="R" THEN 300
260 IF D$="D" THEN 600 ELSE CLS:GOTO 200
300 CLS
310 PRINT "RECORD SHIPMENT"
320 INPUT "TYPE STORE#(1-4)";J
330 PRINT "ITEM SHIPPED"
340 PRINT "REFRIG=1,STOVE=2,AIR
    COND.=3,VACUUM=4,DISPOSAL=5"
350 INPUT I
360 INPUT "NUMBER SHIPPED";S
370 B(I,J) = B(I,J)-S
380 GOTO 200
600 CLS
610 PRINT A$(1),A$(2),A$(3),A$(4)
```

```
620 FOR I=1 TO 5
630    FOR J=1 TO 4
640         PRINT B(I,J);
650    NEXT J
660 NEXT I
670 GOTO 200
1000 END
```

Note that this program is really an infinite loop. For this type of program, this is a good idea. You don't want to accidentally end the program, thereby erasing the current inventory figures! End this program using Ctrl-Break.

Section 6.2 (page 147)

1. `A(1)= 2,A(2)=4,A(3)=6,A(4)=8,A(5)=10,A(6)=12,A(7)=14, A(7)=16,A(8)=18,A(9)=20`
2. `A(0) = 1.1, A(1) = 3.3, A(2) = 5.5, A(3) = 7.7, B(0)=2.2, B(1) = 4.4, B(2) = 6.6, B(3) = 8.8`
3. `A(0) = 1, A(1) = 2, A(2) = 3, A(3) = 4, B$(0) = "A", B$(1) = "B", B$(2) = "C", B$(3) = "D"`
4. `A(0) = 1,B(0)=2,A(1)=3,B(1)=4,A(2)=1,B(2)=2,A(3)=3,B(3)=4`
5. `A(1,1)=1,A(1,2)=2,A(1,3)=3,A(1,4)=4,A(2,1)=5,A(2,2)=6, A(2,3)=7,A(2,4)=8,A(3,1)=9,A(3,2)=10,A(3,3)=11,A(3,4)=12`
6. `A(1,1)=1,A(2,1)=2,A(3,1)=3,A(1,2)=4,A(2,2)=5,A(3,2)=6, A(1,3)=7,A(2,3)=8,A(3,3)=9,A(1,4)=10,A(2,4)=11,A(3,4)=12`
7. Out of DATA in 30
8. 2 Type Mismatches in 30. (Attempt to set a numeric variable equal to a string and a string variable equal to a numeric constant.) Also, Out of Data in 30.
9. Set F(J) equal to the federal withholding tax for employee J, N(J) = the net pay, and add the following lines.

```
280 PRINT "EMPLOYEE", "WITHHOLDING","NET PAY"
290 FOR J = 1 TO 5
300  IF D(J) <= 200 THEN F(J) = 0
310  IF D(J) <= 210 THEN F(J) =29.10
320  IF D(J) <= 220 THEN F(J)=31.20
330  IF D(J) <= 230 THEN F(J)=33.80
340  IF D(J) <= 240 THEN F(J)=36.40
350  IF D(J) <= 250 THEN F(J)=39.00
360  IF D(J) <= 260 THEN F(J)=41.60
370  IF D(J) <= 270 THEN F(J)=44.20
380  IF D(J) <= 280 THEN F(J)=46.80
390  IF D(J) <= 290 THEN F(J)=49.40
400  IF D(J) <= 300 THEN F(J)=52.10
410  IF D(J) <= 310 THEN F(J)=55.10
420  IF D(J) <= 320 THEN F(J)=58.10
430  IF D(J) <= 330 THEN F(J)=61.10
```

```
440  IF D(J) <= 340 THEN F(J)=64.10
450  IF D(J) <= 350 THEN F(J)=67.10
500  N(J) = D(J)-E(J)-F(J)
600  PRINT B$(J),F(J),N(J)
700 NEXT J
```

10.
```
 5 DIM A(25)
10 DATA 10,10,9,9,8,11,15,18,20,25,31,35,38,39,40,40,42,38
20 DATA 33,27,22,18,15,12
30 FOR J=0 TO 23
40       READ A(J)
50       S=S+A(J)
60 NEXT J
70 PRINT "AVERAGE 24 HOUR TEMP.", S/24
100 PRINT "TO FIND THE TEMPERATURE AT ANY PARTICULAR HOUR"
110 PRINT "TYPE THE HOUR IN 24-HOUR NOTATION: 0-12=AM"
120 PRINT "13-24=PM"
130 PRINT "TO END THE PROGRAM, TYPE 25"
140 INPUT "DESIRED HOUR";A
150 IF A=25 THEN 200
160 PRINT "THE QUERIED TEMPERATURE WAS",A(J),"DEGREES"
170 GOTO 100
200 END
```

Section 6.3 (page 157)

1.
```
10 PRINT "THE VALUE OF X IS",5.378
20 END
```

2.
```
10 PRINT "THE VALUE OF X IS";TAB(22) 5.378
20 END
```

3.
```
10 PRINT "DATE";TAB(7) "QTY";TAB(13) "@";TAB(18) "COST";
20 PRINT TAB(26) "DISCOUNT";TAB(38) "NET COST"
30 END
```

4.
```
10 X=6.753:Y=15.111:Z=111.850:W=6.702
20 PRINT USING "###.###"; X
30 PRINT USING "###.###"; Y
40 PRINT USING "###.###"; Z
50 PRINT USING "###.###"; W
60 PRINT "____"
70 PRINT USING "###.###"; X+Y+Z+W
80 END
```

5.
```
10 X=12.82:Y=117.58:Z=5.87:W=.99
20 PRINT USING "$###.##"; X
30 PRINT USING "$###.##"; Y
40 PRINT USING "$###.##"; Z
50 PRINT USING "$###.##"; W
60 PRINT USING "$###.##"; W
60 PRINT "____"
```

```
70 PRINT USING "$###.##"; X+Y+Z+W+W
80 END
```

6.
```
10 PRINT TAB(46) "DATE";TAB(53) "3/18/81"
20 PRINT
30 PRINT "Pay to the Order of";
40 PRINT TAB(27) "Wildcatters, Inc."
50 PRINT
60 PRINT "The Sum of";TAB(41) "*********$89,385.00"
```

7.
```
10 X=5787:Y=387:Z=127486:W=38531
20 PRINT USING "###,###"; X
30 PRINT USING "###,###"; Y
40 PRINT USING "###,###"; Z
50 PRINT USING "###,###"; W
60 PRINT "____"
70 PRINT USING "###,###";X+Y+Z+W
80 END
```

8.
```
10 X=385.41:Y=17.85
20 PRINT " ";:PRINT USING "$$###.##";X
30 PRINT "-";
40 PRINT USING "$$###.##";Y
50 PRINT " ____"
60 PRINT " ";:PRINT USING "$$###.##";X-Y
70 END
```

9.
```
10 INPUT "NUMBER TO BE ROUNDED";X
20 PRINT USING "######";X
30 END
```

10. Modify the program of Example 2 of Section 5.4 by substituting PRINT USING "$####.##"; statements.

11. Put the computer into 40-character-per-line mode by typing WIDTH 40 followed by ENTER. Then RUN the program of Exercise 4.

Section 6.4 (page 165)

1. `100*RND` 2. `100+RND` 3. `INT(50*RND+1)`
4. `INT(4+77*RND)` 5. `2*INT(25*RND+1)` 6. `50+50*RND`
7. `3*INT(9*RND+1)` 8. `1+3*INT(7*RND+1)`

10. Add the following instructions:
```
132 IF C(J) > A(J) THEN 135 ELSE 140
135 PRINT "BET INVALID:NOT ENOUGH CHIPS PURCHASED"
137 C(J)=0
139 GOTO 120
```

11. Change line 132 in Exercise 10 to read:
```
132 IF C(J) > A(J)+100 THEN 135 ELSE 140
```

12.
```
10 PRINT "CHOOSE OPERATION TO BE TESTED"
20 PRINT "ADDITION(A),SUBTRACTION(S),MULTIPLICATION(M)"
30 INPUT A$
40 A = INT(10*RND):B=INT(10*RND)
```

```
50 IF A$ = "A" THEN 100
60 IF A$ = "S" THEN 200
70 IF A$ = "M" THEN 300
100 CLS
110 PRINT "WHAT IS ";A;"+";B;"?"
120 INPUT C
130 D = A+B
140 GOTO 400
200 CLS
210 PRINT "WHAT IS ";A;"-";B;"?"
220 INPUT C
230 D = A-B
240 GOTO 400
300 CLS
310 PRINT "WHAT IS ";A;"X";B;"?"
320 INPUT C
330 D = A*B
340 GOTO 400
400 IF C=D THEN 410 ELSE 420
410 PRINT "YOUR ANSWER IS CORRECT"
415 GOTO 430
420 PRINT "INCORRECT.THE CORRECT ANSWER IS", D
430 INPUT "ANOTHER PROBLEM(Y/N)";B$
440 IF A$ = "Y" THEN 10
450 END
```

13. Put your names in a series of DATA statements located in lines 1000-1010.

```
 5 DIM A$(10)
10 FOR J=1 TO 10
20    READ A$(J)
30 NEXT J
40 FOR J=1 TO 4
50    PRINT A$(INT(10*RND))
60 NEXT J
70 END
```

CHAPTER 7

Section 7.1 (page 170)

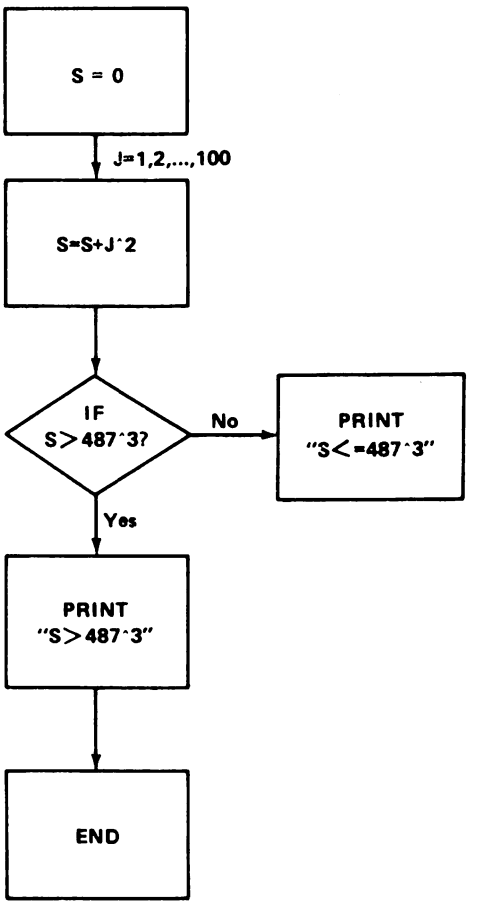


Figure A-1.

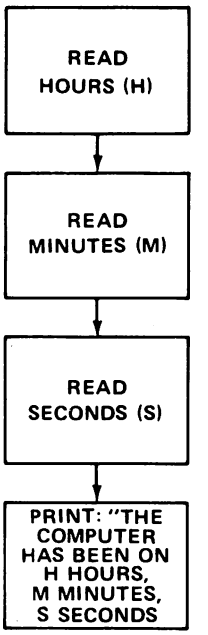


Figure A-2.

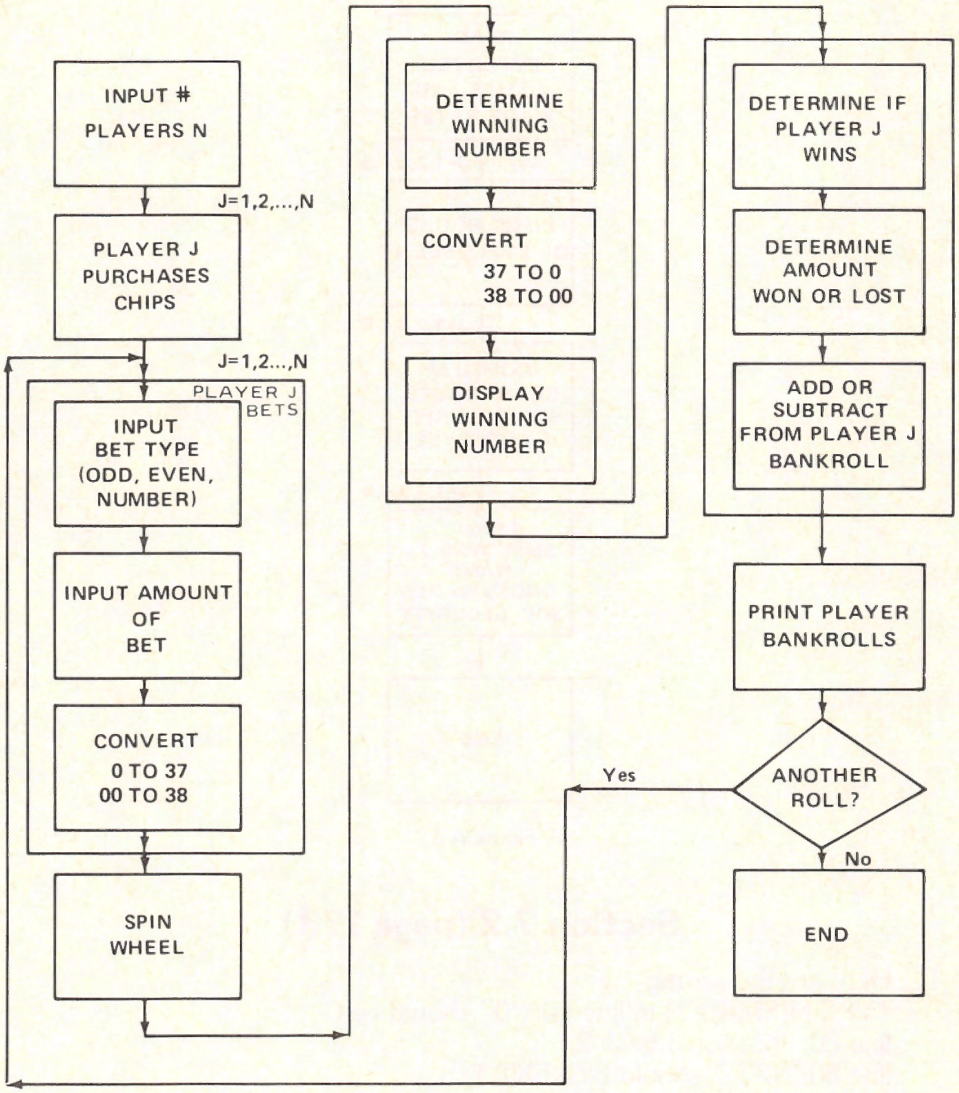

INPUT #
PLAYERS N
J=1,2,...,N
PLAYER J
PURCHASES
CHIPS
J=1,2...,N
PLAYER J
BETS
INPUT
BET TYPE
(ODD, EVEN,
NUMBER)
INPUT AMOUNT
OF
BET
CONVERT
0 TO 37
00 TO 38
SPIN
WHEEL
DETERMINE
WINNING
NUMBER
CONVERT
37 TO 0
38 TO 00
DISPLAY
WINNING
NUMBER
DETERMINE IF
PLAYER J
WINS
DETERMINE
AMOUNT
WON OR LOST
ADD OR
SUBTRACT
FROM PLAYER J
BANKROLL
PRINT PLAYER
BANKROLLS
ANOTHER
ROLL?
Yes
No
END

Figure A-3.

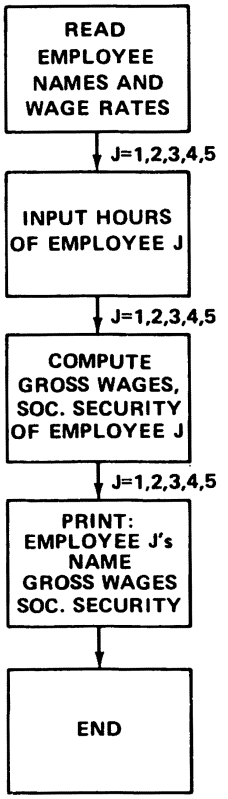


Figure A-4.

Section 7.2 (page 173)

1. Here are the errors:
   TYPE MISMATCH in line 10: "0" should be 0
   line 30: J(2 should be J^2
   line 80: NXT T should be NEXT T
   line 90: should be deleted
   line 100: ST should be S*T
   line 110 quotes around "THE ANSWER IS"
2. Line 30 should read: PRINT "THE FIRST N EQUALS",N
   Need a line 40 stating GOTO 200

CHAPTER 8

Section 8.2 (page 187)

1.
```
10 DATA 5.7,-11.4,123,485,49
20 OPEN "NUMBERS" FOR OUTPUT AS #1
30 FOR J=1 TO 5
40     READ A
```

```
    50     WRITE #1, A
    60 NEXT J
    70 CLOSE 1
    80 END
2.  10 OPEN "NUMBERS" FOR INPUT AS #1
    20 FOR J=1 TO 5
    30     INPUT #1,A
    40     PRINT A
    50 NEXT J
    60 CLOSE 1
    70 END
3.  10 OPEN "NUMBERS" FOR APPEND AS #1
    20 DATA 5, 78, 4.79, -1.27
    30 FOR J=1 TO 4
    40     READ A
    50     WRITE#1,A
    60 NEXT J
    70 CLOSE 1
    80 END
4.  10 OPEN "NUMBERS" FOR INPUT AS #1
    20 FOR J = 1 TO 9
    30     INPUT #1, A
    40     PRINT A
    50 NEXT J
    60 CLOSE 1
    70 END
5.  10 OPEN "CHECKS" FOR OUTPUT AS #1
    20 PRINT "TYPE CHECK DATA ITEMS REQUESTED."
    30 PRINT "FOLLOW EACH ITEM BY A CARRIAGE RETURN."
    40 OPEN "CHECKS" FOR OUTPUT AS #1
    50 INPUT "CHECK #";A
    60 INPUT "DATE";B$
    70 INPUT "PAYEE";C$
    80 INPUT "AMOUNT(NO $)";D
    90 INPUT "EXPLANATION";E$
    100 WRITE #1,A,B$,C$,D,E$
    110 INPUT "ANOTHER CHECK(Y/N)";F$
    120 CLS
    130 IF F$ = "Y" THEN 20
    140 CLOSE
    150 GOTO 1000
    1000 END
6.  10 OPEN "CHECKS" FOR INPUT AS #1
    20 IF EOF(1) THEN GOTO 500
    30 INPUT #1, A,B$,C$,D,E$
    40 S=S+D
    50 GOTO 30
    100 CLOSE
```

```
110 PRINT "TOTAL OF CHECKS IS",S
120 GOTO 1000
1000 END
```

Section 8.3 (page 191)

1. "MY","DOG","SAM", 1234<ENTER>
2. MYDOGSAM1234<ENTER>
3. MY,DOG,SAM,1234<ENTER>
4. "MY DOG, ",SAM,1234<ENTER>
5. MY
6. "MY","DOG","SAM", 1234
7. MYDOGSAM1234
   MY
   MY DOG,
8.
```
10 INPUT#1,E$
20 PRINT E$;
30 INPUT#1,E$
40 PRINT " ";E$
50 INPUT#1,E$
60 PRINT E$
```

Section 8.4 (page 197)

1.
```
 10 OPEN "TELEPHON" AS #1 LEN=72
 20 FIELD #1,  20 AS NAME$,  25 AS ADDRESS$,  10 AS
    CITY$, 2 AS STATE$, 5 AS ZIP$, 10 AS TELEPHONE$
 30 CLS
 40 INPUT "NAME";A$
 50 INPUT "STREET ADDRESS";B$
 60 INPUT "CITY";C$
 70 INPUT "STATE";D$
 80 INPUT "ZIP CODE";E$
 90 INPUT "TELEPHONE NUMBER";F$
100 LSET NAME$=A$
110 LSET ADDRESS$=B$
120 LSET CITY$=C$
130 LSET STATE$=D$
140 LSET ZIP$=E$
150 LSET TELEPHONE$=F$
160 PUT #1, LOF(#1)+1
170 INPUT "ANOTHER ENTRY (Y/N)";G$
180 IF G$="Y" THEN 20 ELSE 200
190 CLOSE #1
200 END
```

2.
```
10 OPEN "SALES" AS #1 LEN=16
20 FIELD #1, 4 AS NUM1$, 4 AS NUM2$, 4 AS NUM3$, 4 AS
   NUM4$
30 FOR J=1 TO 20
40    GET #1, J
50    PRINT CVS(NUM1$), CVS(NUM2$), CVS(NUM3$),
   CVS(NUM4$)
60 NEXT J
70 CLOSE #1
80 END
```

Section 8.7 (page 212)

1. (a)
```
10 S=0
20 FOR J=1 TO 50
30       S=S+J^2
40 NEXT J
50 PRINT S
60 END
```
(b) Type SAVE "SQUARES",A

2. (a)
```
100 S = 0
110 FOR J=1 TO 30
120      S=S + S+J^3
130 NEXT J
140 PRINT S
150 END
```
(b) Type MERGE "SQUARES" (c) Type LIST

(d) DELETE 60 (This is the END of SQUARES.) Type RUN.

(e) Type SAVE "COMBINED",A (The A is optional.)

3. Type LOAD 'COMBINED'
4. Type ERASE 'SQUARES'

CHAPTER 9

Section 9.1 (page 218)

1.
```
10 PRINT ASC("A")
20 PRINT ASC("a")
30 PRINT ASC("B")
40 PRINT ASC("b")
50 PRINT ASC("C")
60 PRINT ASC("c")
70 PRINT ASC("D")
80 PRINT ASC("d")
```
2. The ASCII codes of upper- and lowercase letters differ by 32.
3.
```
10 PRINT "He said to me,"+CHR$(ASC("))+"The IBM is an
excellent computer"+CHR$(ASC("))+"."
```

Section 9.2 (page 225)

2.
```
10 A$ = "15+48+97=160"
20 B$(1) = LEFT$(A$,2)
30 B$(2) = MID$(A$,4,2)
40 B$(3) = MID$(A$,7,2)
50 B$(4) = RIGHT$(A$,3)
60 FOR J=1 TO 4
70     B(J) = VAL(B$(J))
80 NEXT J
90 FOR J=1 TO 3
100    PRINT USING "###", B(J)
110 NEXT J
120 PRINT "____"
130 PRINT USING "###", B(4)
140 END
```

3.
```
10 INPUT A$:INPUT B$
20 A1$= RIGHT$(A$,7):B1$= RIGHT$(B$,7)
30 A2 = VAL(A1$): B2 = VAL(B1$)
40 PRINT USING "$####.##",A2
50 PRINT USING "$####.##",B2
60 PRINT "____"
70 PRINT USING "$####.##",A2+B2
80 END
```

Section 9.3 (page 228)

1.
```
10 PRINT "HELLO"
20 PRINT CHR$(28)+CHR$(28)+CHR$(28)+CHR$(28)+CHR$(28);
```

2.
```
100 INPUT S$
110 IF S$="S" THEN PRINT CHR$(29);
120 IF S$="D" THEN PRINT CHR$(28);
130 IF S$="X" THEN PRINT CHR$(31);
140 IF S$="E" THEN PRINT CHR$(30);
150 END
```

3.
```
10 INPUT LONGSTRING$
20 L=LEN(LONGSTRING$)
30 S$=LEFT$(LONGSTRING$,1)
40 LONGSTRING$=RIGHT$(LONGSTRING$,L-1)
50 GOSUB 100
60 IF L=1 THEN 80
70 GOTO 20
80 END
100 'Effect motion for S$
110 IF S$="S" THEN PRINT CHR$(29);
120 IF S$="D" THEN PRINT CHR$(28);
130 IF S$="X" THEN PRINT CHR$(31);
140 IF S$="E" THEN PRINT CHR$(30);
150 RETURN
```

5.
```
10 C=POS(0)
20 LOCATE 25,POS(0)
```
6.
```
10 LOCATE CSRLIN,1
```

CHAPTER 10

Section 10.1 (page 235)

1.
```
10 FOR J=1 TO 80
20     LOCATE 18,J: PRINT CHR$(196);
30 NEXT J
40 END
```
2.
```
10 FOR J=1 TO 25
20     LOCATE J,17: PRINT CHR$(179);
30 NEXT J
40 END
```
3.
```
10 FOR J=1 TO 80
20    LOCATE 13,J: PRINT CHR$(196);
30 NEXT J
40 FOR J = 1 TO 25
50     LOCATE J,40: PRINT CHR$(179);
60 NEXT J
65 LOCATE 13,40: PRINT CHR$(197)
70 END
```
4.
```
10 CLS
20 FOR J=1 TO 2
30     FOR K=1 TO 80
40          LOCATE 8*J,K: PRINT CHR$(196);
50     NEXT K
60 NEXT J
70 FOR J=1 TO 2
80     FOR K=1 TO 25
90          LOCATE K,26*J: PRINT CHR$(179)
100    NEXT K
110 NEXT J
112 LOCATE 8,26: PRINT CHR$(197);
113 LOCATE 8,52: PRINT CHR$(197);
114 LOCATE 16,26:PRINT CHR$(197);
115 LOCATE 16,52: PRINT CHR$(197);
120 END
```
5.
```
10 FOR J=1 TO 24
20     LOCATE J,30: PRINT CHR$(219);
30     LOCATE J,31: PRINT CHR$(219);
40 NEXT J
50 END
```
6.
```
10 FOR J=1 TO 24
20     LOCATE J,J: PRINT CHR$(219);
```

```
30 NEXT J
40 END
```

7.
```
10 FOR J=1 TO 80
20     LOCATE 12,J: PRINT CHR$(219);
30 NEXT J
40 FOR K=1 TO 3
50     LOCATE 11,20+K*20:PRINT CHR$(219);
60     LOCATE 13,20+K*20: PRINT CHR$(219);
70 NEXT K
80 END
```

8.
```
10 FOR J=1 TO 25
20     LOCATE J,40:PRINT CHR$(219);
30 NEXT J
40 FOR J=0 TO 4
50     LOCATE 5+5*J,39: PRINT CHR$(219);
60 NEXT J
70 END
```

9. Suppose that the name to be displayed is "JOHN JONES".

```
10 LOCATE 2,2
20 PRINT "JOHN JONES"
30 LOCATE 1,1
40 FOR J=1 TO 12
50     LOCATE 1,J: PRINT CHR$(42);
60     LOCATE 2,J: PRINT CHR$(42);
70 NEXT J
80 LOCATE 2,1: PRINT CHR$(42);
90 LOCATE 2,12: PRINT CHR$(42);
100 END
```

10.
```
10 FOR J=1 TO 80
20 LOCATE 16,J: PRINT CHR$(219);
30 FOR J=0 TO 10
40   LOCATE 15,8*J:PRINT CHR$(219);
50   LOCATE 17,8*J:PRINT CHR$(219);
60   LOCATE 8*J,18
70   PRINT 8*J
80 NEXT J
90 END
```

11.
```
10 INPUT "ASCII GRAPHICS CODE",A
20 PRINT CHR$(A)
30 END
```

12.
```
10 CLS
20 LOCATE 1,1
30 PRINT "COST"
40 LOCATE 1,2
50 PRINT "PRICE"
60 LOCATE 1,3
70 PRINT "INDEX"
80 FOR J=1 TO 25
```

```
90    LOCATE J,6: PRINT CHR$(219)
100 NEXT J
110 FOR J=1 TO 80
120    LOCATE 22,J: PRINT CHR$(219)
130 NEXT J
140 DATA J,F,M,A,M,J,J,A,S,O,N,D
150 FOR J=1 TO 12
160   READ A$
170   LOCATE 21,6*J: PRINT CHR$(219)
180   LOCATE 23,6*J: PRINT CHR$(219)
190   LOCATE 6*J,24
200   PRINT A$
210 NEXT J
220 LOCATE 72,25
230 PRINT "MONTH"
240 END
```

Section 10.2 (page 242)

1.
```
10 COLOR 5,1
```

2.
```
10 COLOR 12,0
```

3.
```
10 PSET (200,80), 1
```

4.
```
10 COLOR 3,0
20 PSET (100,100), 2
```

5.
```
10 PSET STEP (-200,-100)
```

6.
```
10 PSET STEP (100,0)
```

7.
```
10 DIM P(15)
20     FOR J=0 TO 15
30     P(J)=15-J
40 NEXT J
50 PALETTE USING P(0)
```

Section 10.3 (page 253)

1.
```
10 LINE (20,50)-(40,100)
```

2.
```
10 LINE -(250,150),2
```

3.
```
10 LINE (125,50)-STEP(100,75),1
```

4.
```
10 LINE (10,20)-(200,150),,B
```

5.
```
10 LINE (10,20)-(200,150),3,BF
```

6.
```
10 CIRCLE (30,50),20
```

7.
```
10 CIRCLE (30,50),20,,1.5,3.1
```

8.
```
10 CIRCLE (30,50),20,,-1.5,3.1
```

Chapter 12

Section 12.1 (page 296)

1.
```
"A"
```

2.
```
"c"
```

3.
```
10 A$=INKEY$
20 IF A$<>"" THEN PRINT A$;
```

```
30 GOTO 10
```

Section 12.2 (page 300)

1. `KEY 5, ""`
2. `KEY 1, "LIST"+CHR$(13)`
3. `10 KEY 1, "CLS"+CHR$(13)+"NEW"+CHR$(13)`
4. Let the first line of each subroutine be:
   `FOR J=1 TO 4:KEY(J) STOP :NEXT J`
   Let the last line of each subroutine be:
   `FOR J=1 TO 4:KEY(J) ON :NEXT J`
5. `KEY(11) ON`

Section 12.3 (page 303)

1.
```
10 ON ERROR GOTO 1000
1000 RESUME
```
2.
```
10 ON ERROR GOTO 1000
1000 IF ERR=13 THEN 30 ELSE END
1010 IF ERL=500 THEN 40 ELSE END
1020 PRINT "Type Mismatch Error in Line 500"
1030 RESUME 600
```

Section 12.4 (page 305)

1. `CHAIN MERGE "L"`
2. End program "A" with the statement: CHAIN "B" . End program "B" with the statement CHAIN "C" .

CHAPTER 13

Section 13.1 (page 311)

1. Type:
   `DATE$="mm/dd/yy"`
   Press ENTER. (Here mm is the month, dd is the day and yy is the year.)
   Type:
   `TIME$="hh:mm:ss"`
   Press ENTER. Here hh is the hour, mm is the minutes, ss the seconds.
2. Type:
   `PRINT TIME$`
   Press ENTER.
3.
```
10 A$=TIME$
20 S$=RIGHT$(TIME$,2)
30 IF S$<>S1$ THEN 40 ELSE 20
40 S1$=S$
50 CLS
```

```
60 PRINT TIME$
70 GOTO 10
```

4.
```
10 A$=TIME$
20 S$=MID$(TIME$,4,2)
30 IF S$<>S1$ THEN 40 ELSE 20
40 S1$=S$
50 CLS
60 PRINT TIME$
70 PRINT DATE$
80 GOTO 10
```

Section 13.2 (page 315)

1. Modify line 290 of the program.

Section 13.3 (page 318)

1. `(10,10)-(89,49)`
2. `(0,0)-(639,15)` (in high resolution mode)
3. `DIM Z%(212)`
4. `DIM Z%(327)`
5. Use the program given in the text.
6. Use the program given in the text.
7.
```
10 SCREEN 0
20 CLS
30 PRINT CHR$(2);
40 DIM Z%(9)
50 GET (0,0)-(7,7),Z%
```
8.
```
60 SCREEN 1
70 PUT (0,0), Z%
80 PUT (50,50), Z%
90 PUT (0,100), Z%
```
9.
```
10 SCREEN 0
20 CLS
30 PRINT CHR$(2);
40 DIM Z%(9)
50 GET (0,0)-(7,7),Z%
60 SCREEN 1
70 FOR J=0 TO 312
80     PUT (J,72), Z%
90     PUT (J,72), Z%
100 NEXT J
```
10.
```
10 SCREEN 0
20 CLS
30 PRINT CHR$(2);
40 DIM Z%(9)
50 GET (0,0)-(7,7),Z%
```

```
60 SCREEN 1
70 FOR J=0 TO 312
80      PUT (J,200*J/320), Z%
90      PUT (J,200*J/320), Z%
100 NEXT J
```

CHAPTER 14

Section 14.1 (page 329)

1. 3 2. 2.370000 3. 578,000.0
4. 2.00000000000000000 5. 3.000000 6. -4.100000
7. -4 8. 3500.684 9. 217.60000000000000
10. -5,940,000,000,000 11. 3.5869504003837265
12. -2.345423837461E10
13. -236,700,000,000,000,000,000
14. 4.570000E18 15. 46 16. .5000000
17. .60000000000000000 18. 1.600000
19. .66666666666666666
20. 1.1966666666666666 21. .6666666 22. 1.196666
23. 4963.000 24. 1749.9999900000000
25. 46, .5, .6#, 1.6#, .6666666666666666, 1.196666666666666, .666666, 1.19666, 4963, 1749.99999#
26. 3.333332, accurate to six digits.
27. 3.3333333333333330, round-off error occurs in the seventeenth place.

Section 14.2 (page 333)

1.
```
10 PRINT (5.87 + 3.85 - 12.07)/11.98
20 END
```
2.
```
10 PRINT (15.1 + 11.9)^4/12.88
20 END
```
3.
```
10 PRINT (32485 + 9826)/(321.5 - 87.6^2)
20 END
```

4.-6. Place # after all constants in the above programs.

7.
```
10 INPUT X%
20 IF X% < 0 THEN X% = X% - 1
30 PRINT X%
40 END
```

8. -5 9. 5 10. -11 11. 1.780000
12. 1.7800000000000000
13. 32.65343 14. 4.252345E21 15. -1.234568E-32
16. 3.2836464930292736 17. -5.7400000000000000

Section 14.3 (page 337)

1.
```
10 PRINT EXP(1.54)
20 END
```

2.
```
10 PRINT EXP(-2.376)
20 END
```

3.
```
10 PRINT LOG(58)
20 END
```

4.
```
10 PRINT LOG(9.75E-5)
20 END
```

5.
```
10 PRINT SIN(3.7)
20 END
```

6.
```
10 PRINT COS(.017453*45)
20 END
```

7.
```
10 PRINT ATN(1)
20 END
```

8.
```
10 PRINT TAN(.682)
20 END
```

9.
```
10 PRINT 57.29578*ATN(2)
20 END
```

10.
```
10 PRINT LOG(18.9)/LOG(10)
20 END
```

11.
```
10 FOR X=-5.0 TO 5.0 STEP .1
20    PRINT X, EXP(X)
30 NEXT X
40 END
```

12.
```
10 DATA 1.7, 3.1, 5.9, 7.8, 8.4, 10.1
20 FOR J=1 TO 6
30    READ X
40    PRINT X, 3*X^(1/4)*LOG(5*X) + EXP(-1.8*X)*TAN(X)
50 NEXT J
```

15.
```
10 INPUT X
20 PRINT "THE FRACTIONAL PART OF",X,"IS", ABS(X-FIX(X))
30 END
```

Section 14.4 (page 340)

1. `10 DEF FNA(X) = X^2 - 5*X`
2. `10 DEF FNA(X) = 1/X - 3*X`
3. `10 DEF FNA(X) = 5*EXP(-2*X)`
4. `10 DEF FNA(X) = X*LOG(X/2)`
5. `10 DEF FNA(X) = TAN(X)/X`
6. `10 DEF FNA(X) = COS(2*X) + 1`
7. `10 DEF FNA$(C$) = RIGHT$(C$,2)`
8. `10 DEF FNA$(A$,J) = MID$(A$,J,4)`
9. `10 DEF FNA$(B$) = MID$(B$,INT(LEN(B$)/2)+1,1)`
10.
```
10 DEF FNA(X) = 5*EXP(-2*X)
20 FOR X=0 TO 10 STEP .1
30    PRINT X, FNA(X)
40 NEXT X
50 END
```

Index

absolute coordinates, 268
absolute value, 336
acoustic modem, 19
adding data to a file, 186
alphabetizing strings, 224
Alt, 32
Alt key, 232
AND, 112
animation, 276, 277
APPEND, 186
applications of loops, 104
arithmetic, 62
arithmetic practice, 121
array, 135, 139
array, two-dimensional, 136
arrays, deleting, 141
ASC, 217
ASCII code, 215
ASCII codes, table of, 215
ASCII control codes, 226
ASCII control codes, table of, 227
aspect ratio, 251
attribute, 241
AUTO, 92
autoexec.bat file, 54
axes, 235

background color, 240
background music, 287
backslash, 68
backspace, 225
Backspace key, 31
backup, 28
Bad File Mode, 304
barchart 257, 258
BASIC compiler, 207
BASIC editor, 95
BASIC file commands, 209
BASIC interpreter, 207
BASIC prompt, 22, 27, 70
BASIC Reference Manual, 173
batch file, 53, 54
batch file parameters, 55
baud, 19
BEEP, 225, 282, 283
Beethoven, 284
Blind Target Shoot, 312
blinking a display, 108
boldface, 291
boundary (for PAINT), 264
Break, 71
BSAVE, 277
bubble sort, 204, 205
bubble sort, modified, 208
byte, 5, 192, 192

cable, 10
cable connection for keyboard, 8
Caps Lock key, 31
carbon dating, 335
carriage return, 188, 216, 226
cartesian coordinates, 278
Cartridge BASIC, 9, 59, 248, 298
Cartridge BASIC, starting, 22
cash register, 129
cassette, 3, 15, 90
Cassette BASIC, 9, 59, 89
cassette BASIC cartridge BASIC, 59
cassette connector, 10
cassette recorder, 5
CDBL(X), 337
centering, 291
central processing unit, 2
centroid, 266, 267
CHAIN, 303
CHAIN MERGE, 304
checking, 120
CHR$, 217, 232
CIN(X), 337
CIRCLE, 61, 243, 248, 249
circuit board, 12
circular arc, 249
CLEAR, 139, 238
clipping, 281
clock, 307
clock, setting, 307
CLOSE, 180, 181
closing a file, 180

CLS, 60, 105, 232
COLOR, 60, 242, 264
color display, IBM, 17
colors, 240
colors, choosing, 242
command mode, 73
command, BASIC, 86, 87
commands, 41
COMMON, 304
communications, 18
communications parameters, 19
COMP, 52
compressed format for saving programs, 211
computer art, 254
concatenation of strings, 219
connector, 9, 9, 11
connector, cassette, 10
connector, color display, 9
connector, joystick, 10
connector, keyboard, 10
connector, light pen, 10
connector, modem, 10
connector, power, 9
connector, printer, 10
connector, television, 9
constant, 62
CONT, 105
control characters, 225
Control key, 31
conversion functions, 337
convex figure, 266
COPY, 44
copying diskettes, 29, 46
correcting errors in programs, 95
COS(X), 334
COT(X), 334
CPU, 2
creating a diskette file, 44
CSC(X), 334
CSNG(X), 337
CSRLIN, 226
Ctrl, 32
current drive, 28
cursor, 22, 27, 226
cursor down, 226
cursor left, 226
cursor motion keys, 33
cursor right, 226
cursor up, 226

Dartmouth College, 59
DATA, 143, 146
data file, 37, 179
data, entering, 118
data, inputting, 121, 121, 143
database management program, 199
DATE, 51
DATE$, 307
debugging, 170
debugging hints, 175
DEF FN, 338
DEF SEG, 277
default drive, 28
DEFINT, 331
DEFSNG, 331
Del, 32
Del key, 96
delay loops, 107
delays, 107
DELETE, 89
Delete key, 33
deleting program lines, 88
delimiter, 188
device names, 40
dice, 160
DIM, 138, 139
DIR, 39
direct-connect modem, 19
directory, 38, 45, 209
directory entry, 45
Disk Not Ready, 302
DISKCOMP, 52
DISKCOPY, 29, 29
diskette, 3, 5, 6, 9, 15, 22, 23, 90
diskette drive, 5, 9, 15, 22
diskette drive adapter, 16
diskette drive adapter board, 14
diskette, cautions in handling, 24
diskette, soft-sectored, 46
division, integer, 68
DOS prompt, 26
DOS Supplementary Programs, 25
double-precision constant, 326
double-precision variable, 331

doubly-subscripted variable, 136
DRAW, 267
duplicate definition, 140

easing programming frustrations, 167
echo, 16, 16
editing, 97
editor, 95
elapsed time, 308
ellipse, 251, 252
ellipse, parametric equations for, 252
End, 33, 33
enhanced PCjr system, 8
ENTER, 26, 32, 72
ENTER key, 31
entering data, 118
ERASE, 141
erasing a program, 91
erasing files, 210
erasing the screen, 33
ERROR, 303
error messages, 172
error messages, table of, 174
error number, 175
error trapping, 301
error trapping routine, 301
error trapping statement, 302
errors, 170
escape key, 8, 31
event trapping, 296, 298
execute mode, 73, 74
executing commands, 41
executing programs, 42
EXP(X), 335
expansion slot, 14
exponent, 67
exponential format, 66
exponentiation, 67
external commands, 43
external commands, DOS, 52

FIELD, 192, 193, 194
field of a random access file, 193
Field Overflow, 197
file, 37, 179
file buffer, 191
file buffer, random access, 197
file commands, 209
file name, 37
File Not Found, 302
file specification, 39, 40
file, adding data to a, 186
file, appending, 186
file, writing a sequential, 182
FILES, 70, 209
files, setting number of simultaneous, 181
FIX(X), 337
floppy diskette, 6
flowchart, 167
flowcharting, 167
Fn-Break 117, 235
footnotes, 291
FOR, 99
FOR without NEXT, 102
FOR...NEXT, 115
foreground color, 241
foreground music, 287
form feed, 226
FORMAT, 46
FORMATting diskettes, 46
formatting numbers, 152
formatting output, 149
full-screen editor, 95
function key display, 35
function, 333
Function key, 31, 296
function, user-defined, 338

gambling, 158
GET, 194, 194, 275
global search and replace, 290
GOSUB, 128, 129
GOTO, 113
graphics, 229
graphics characters, 232
graphics characters, table of, 233
graphics coordinates, 238, 239
graphics images, saving and recalling, 274
Graphics Macro Language, 267
graphics, high-resolution, 238
graphics, low-resolution, 238

graphics, medium-resolution, 238
greatest integer, 336

hard copy, 3, 84
Herz, 284
high-resolution graphics, 14
high-resolution graphics mode, 17
Home, 33, 226
horizontal tabbing, 151

IBM 80 cps Graphics Printer, 17
IBM color display, 17
IBM compact printer, 17
IF...THEN, 110, 111, 113, 115, 119
IF...THEN...ELSE, 110, 111, 113, 115
immediate mode, 59, 70, 73
indexing, 291
infinite loops, 117
infrared optical link, 8
INKEY$, 295, 296
input, 7, 118, 119, 122
Input Past End, 184
input routine, 283
input unit, 3
INPUT#, 184, 189
INPUT$, 123
Ins key, 96
insert key, 33
insert mode, 96
INSTR, 221, 222
INT, 160
INT(X), 336
integer constant, 325
integer division, 68
integer type variable, 330
interest, 81
internal command, DOS, 50
internal commands, 43
internal modem, 14, 18, 19

joystick connector, 10
justification, 291

Kemeny, John, 59
KEY LIST, 297
KEY n, 297
KEY OFF, 35, 297
KEY ON, 35, 297
KEY(n) OFF, 300
KEY(n) ON, 300
KEY(n) STOP, 300
keyboard, 6, 7, 8, 29
Keyboard Adventure, 8
keyboard buffer, 295
keyboard cord connector, 10
KILL, 91, 210
Kurtz, Thomas, 59

last point referenced, 239, 267
left justification, 193
LEFT$, 220
LEN, 220
length of a string, 220
LET, 78, 80, 81
light pen connector, 10
line, 243, 244
line feed, 188, 188, 216, 226
LINE INPUT, 122, 122
LINE INPUT#, 190
line number, 92
line width, 34, 229
LIST, 71, 72, 87, 89, 97
list manager program, 199
LOAD, 70, 91
loan, 81, 105
LOC, 195
LOCATE, 129, 226, 229, 239
LOF, 195
LOG(X), 335
loop, 99, 100
loop variable, 100
loops, applications of, 104
loops, making readable, 100
LPRINT, 85
LPRINT USING, 155
LSET, 193

mailing list, 184, 185
making decisions, 110
master DOS diskette, 25
mathematical function, 333
memory, 3
memory chip, 13
memory expansion and display adapter, 16

MERGE, 211, 212
merging programs, 211
microprocessor, 3, 6
MID$, 220
mixing text and graphics, 245
MKD$, 193
MKI$, 193
MKS$, 193
MOD, 68, 69
modem connector, 10
modem, acoustic, 19
modem, direct-connect, 19
monitor, direct drive, 16
monitor, RGB, 16
monitor, video composite, 16
Moonlight Sonata, 284
music, 282, 284
Music Macro Language, 60

NAME, 210
nested loops, 102, 103
nested subroutines, 132
NEW, 73
NEXT, 99, 99
null string, 62
numbers, formatting, 152
numeric constant, 62
numeric variable, 82

ON ERROR GOTO, 301
ON KEY(n) GOSUB, 298
ON TIME GOSUB, 310
ON...GOSUB, 132
OPEN 180, 192
opening a file, 180
operating system, 24
operations on strings, 219
OPTION BASE, 141
OR, 112
order of operations, 63
order relations among strings, 223
output unit, 3
output, formatting, 149

PAINT, 264
palette, 241
PALETTE USING, 241
parallel interface, 18
parameters, batch file, 55
parametric equations of an ellipse, 252
parentheses, 63
payroll, 145
PCjr, 2, 5
PCjr basic model, 9
PCjr Internal Modem, 19
PCjr keyboard, 8, 29, 31
PCjr, starting with DOS, 24
personal computing, 1
PgDn, 33
PgUp, 33
Pi, 249
pie chart, 261
pixel, 238
pixels, 242
PLAY, 60, 284, 285
polygon, 255
polygon, inscribed, 256
POS(0), 226
power, 67
power connector, 9
powers, 336
PRESET, 242
PRINT, 63, 75, 79
PRINT USING, 152
print zones, 75
PRINT#, 189
printer, 17, 84
printer connector, 10
printer, IBM 80 cps Graphics, 17
printer, IBM compact, 17
printing the screen, 33
program, 37, 70, 179
program file, 37, 179
program name, 90
programming tips, 94
protected format for saving programs, 211
PSET, 242
PUT, 193, 276

RAM, 5, 13
RAM cassette, 3
random access file, 191
random access file, file buffer of, 197

random access file, length of, 195
random access file, location in, 195
random access file, reading, 194
random access file, writing, 193
random number seed, 159
random numbers, 158, 165
RANDOMIZE, 159
READ, 143, 146
read-write window, 23
reading a random access file, 194
reading a sequential file, 184
real time clock, 307
recalling a program, 91
recalling graphics image, 274
record, 191, 192
rectangle, 243, 245
rectangles, expanding, 280
relative coordinates, 239, 268
REM, 84
remarks, 84
RENAME, 51
renaming files, 210
RENUM, 93
repetitive operations, 99
reserving graphics memory, 238
reset, 32
RESTORE, 146
RESUME, 303
RESUME NEXT, 303
RETURN, 128, 129
right justification, 193
RIGHT$, 220
RND, 158
ROM, 3, 14
roulette, 161, 163
round-off errors, 329
RS232-C interface, 20
RSET, 193
RUN, 70, 72, 73, 74

saving a program, 90
saving a screen image on diskette, 277
saving graphics image, 274
saving programs, 211
saving programs, compressed format, 211
saving programs, protected format for, 211
saving programs, tokenized format for, 211
scientific notation, 66
SCREEN, 60, 238, 240
screen coordinates, 279
scrolling, 31, 32
search and replace, 290
SEC(X), 334
semicolon, 149
sequential file, 180, 182, 191
sequential file, reading, 184
serial port, 19
SGN(X), 337
shift key, 31
Shooting Gallery, 313
simulation, 341
SIN(X), 334
single-precision, 82
single-precision constant, 325, 332
single-precision type variable, 331
sorting, 204
sound, 282, 284
space bar, 31
SPACE$, 129
SPC, 151
special characters, table of, 233
spelling, 291
starting PCjr (without DOS), 21
statement, BASIC, 59
STEP, 109, 109, 239
STOP, 105
STR$, 221
straight line, 243
string addition, 219
string concatenation, 219
string constant, 62
string variable, 82
STRING$, 234
string, length of a, 220
structuring solutions to problems, 125
subroutine, 127, 128
subscript, 135, 291
Subscript out of range, 138
subscripted variable, 135
superscript, 291

SWAP, 82
system reset, 32
system unit, 6, 8, 12, 13

TAB, 151, 155, 220, 225
Tab key, 31
TAN(X), 334
target diskette, 50
telephone directory program, 183
telephone directory, random access version, 196
text mode, 229
Tic Tac Toe, 313
TIME, 51
TIME$, 162, 307, 308
TIMER ON, 310
tokenized format for saving programs, 211
trace, 170
triangle, 243, 244
TROFF, 172
TRON, 171
TYPE, 51
type declaration tag, 327
type mismatch, 147, 189, 303
type of a variable, 330

underscore, 291
user-defined function, 338
user-defined keys, 296

VAL, 132, 162, 221, 308
variable, 77
variable of double-precision type, 331
variable of integer type, 330
variable of single-precision type, 331
variable type, 330
variable initialization, 127
video display, 3
video monitor, 3
VIEW, 278, 281
VIEW SCREEN, 281
viewport, 281

WEND, 115
WHILE...WEND, 115
WIDTH, 34, 229
wild card character, 40
wild card characters, 45
WINDOW, 278
WINDOW SCREEN, 279
Wirth, Niklaus, 209
word processing, 289
word processor, 292
write protect notch, 23
WRITE#, 182, 188
writing a random access file, 193
writing a sequential file, 182
writing BASIC programs, 71

Documentation For Optional Program Diskette

Seventy of the major programs in this book are available on diskette. You may order this diskette at your local bookseller or via the attached order envelope. Using the diskette, you may use the programs in the book without going through the somewhat painful tasks of typing and debugging.

USING THE PROGRAM DISKETTE

1. If you are using the Program Diskette for the first time, make a backup copy just as you would for any master diskette. (Follow the backup procedure outlined on pages 29-39.)
2. Start your computer, load Cartridge BASIC by typing BASICA and obtain the BASIC Ok prompt. Follow the procedure on page 22 (if you are not using DOS) or page 25-27 (if you are using DOS). Note that the Program Diskette does not contain DOS. You must start the PCjr with your own copy of DOS. Moreover, the BASIC cartridge must be in one of the cartridge slots when starting the computer.
3. The programs on the diskette are listed below by program name and page number. To run a program, first insert the Program Diskette. You may use either disk drive. If you insert the Program Diskette into the current drive, type:

   ```
   RUN "<program name>"
   ```

   and press ENTER. For example, to run the program TICTAC type:

   ```
   RUN "TICTAC"
   ```

   and press ENTER.
4. When the program is finished, BASIC will redisplay the Ok prompt. You may then rerun the same program by typing RUN and pressing ENTER or you may run another program by giving a command as described in 3.
5. To interrupt a program, simultaneously press Fn and Break.

PROGRAM DISKETTE CONTENTS

Name	Pg	Name	Pg
SUMPROD1	80	LOAN1	104
LOOP1	99	LOAN2	105
LOOP2	101	DELAY1	107
SUMPROD2	102	SECURITY	108
LOOP3	103	LUMBER	114

CARPET 115
CARPET2 116
ELECTION 116
GRADES 119
CHECKING 120
ARITH1 121
CASHREG 130
ARRAY1 139
ARRAY2 140
ELECTRIC 144
PAYROLL 145
ARRAY3 146
FORMAT1 152
FORMAT2 153
DICE 160
ROULET1 161
ROULET2 163
FILE1 182
FILE2 182
TELEINPT 183
COUNT 185
TELESRCH 185
LABEL 186
TELEAPND 187
TELERND 196
LIST 200
BUBBLE1 206
BUBBLE2 207
BUBBLE3 208
BUBBLE4 208
CENTER 220
ALPHABET 224
LINE 234
BLINK 234
TRNGL1 244
TRNGL2 245
PLANET 253
ART 255
POLYGON 257
BARCHART 260
PIECHART 262
BOATS1 271
BOATS2 272
LETTER 276
ANIMATE 276
WINDOW 280
BEEP 283
INPUT 283
ROCKCONC 284
MUSIC 285
WORDPROC 292
ARITH2 299
ARITH3 310
TARGET 313
SHOOT 317
TICTAC 321
LOG 335
CARBDAT 335
STORE 344

Now—The Critics' Choice!

IBM PCjr: INTRODUCTION, BASIC PROGRAMMING AND APPLICATIONS

Larry Joel Goldstein

Now—for the first time ever—a text specifically designed for novices, potential buyers, and existing owners of the IBM PCjr! Here is a self-study guide that gives beginning PCjr users a clear, complete, and up-to-date account of the programming capabilities and applications of this remarkable new machine. Contains all the information you need to know about the IBM PCjr, from turning it on to programming and why!

In this book you will find:

- A clear, concise outline of what the PCjr is and how it works
- A thorough treatment of DOS and PC BASIC with helpful tips on easing the frustrations of programming
- A detailed discussion of graphics, sound, and file handling
- Immediate applications to business, games, and word processing
- Extensive coverage on structuring, planning, and debugging programs
- Plus—comprehensive tables, charts, appendices, and much more!

Whether you plan to use your PCjr to run programs you buy or to run programs you write, this is the book for you!

CONTENTS

A First Look At Computers / Using Your PCjr For The First Time / An Introduction To DOS / Getting Started In BASIC / Controlling The Flow Of Your Program / Working With Data / Easing Programming Frustration / Your Computer As A File Cabinet / String Manipulation / An Introduction To Computer Graphics / Word Processing / Some Additional Programming Tools / Computer Games / Different Kinds Of Numbers In PC BASIC / Computer Generated Simulations / Answers To Selected Exercises / Index

ISBN 0-89303-539-4