dilithium Press

GET PERSONAL WITH YOUR TI-99

WILLIAM A. MANNING, LON INGALSBE

Get Personal with Your TI-99/4A

Get Personal with Your TI-99/4A

William A. Manning
and Lon Ingalsbe

dilithium Press
Beaverton, Oregon

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Ingalsbe, Lon
Get personal with your TI-99/4A.

Includes index.
1. TI-99/4A (Computer)–Programming. 2. Basic (Computer program language) I. Manning, William A. II. Title.
QA76.8.T133I54 1984 001.64'2 83-20994
ISBN 0-88056-098-3

Cover: Bruce Fleming

Printed in the United States of America

dilithium Press
8285 S.W. Nimbus
Suite 151
Beaverton, Oregon 97005

Preface

This book is designed to provide background information and programming skills for the TI-99/4A personal computer. This computer brings *computer power* to the user's fingertips for a fraction of the cost of computers in the past. The TI-99/4A comes standard with 16,000 characters of memory, color, sound, graphics capability, and a state-of-the-art processor unit. We have seen this machine configured with peripheral devices (printers, disks, tapes, etc.) to electronically process all the accounting activities in a medium-sized distribution company. This computer is not a toy! It is a powerful tool to help you with daily activities. This book intends to give you all you need—some background information and programming skills.

To enjoy this book, you do not need a computer. You can develop the programs and carry out the activities at the end of each chapter. You can imagine the color and graphics involved. If you are thinking of buying a Texas Instruments computer, this book helps you pick appropriate hardware and software combinations.

If you have a TI now, use it to work through the exercises in this book. Make your own changes, modifications, and improvements on the program listings. Can you improve on the program's logic or output quality? Can you do the problems another way? There are many ways to get the same answer in programming. Make changes and see what happens. You can always go back to the original form.

Section I of the book focuses on an introduction to personal computing and orients you to the TI-99/4A computer. It discusses available hardware and software programs and their approximate costs.

Section II addresses programming in the BASIC language for the TI-99/4A; fundamental concepts of how to design and write programs; and detailed coverage of the most common statements. As the chapters progress, more and more of the statements combine into meaningful programs. It also discusses the concept of computer files.

Introduction

The Ultimate Goal – Increasing Your Computer Literacy

Your major educational goal with computers should be to increase your literacy and help you feel more comfortable with the subject matter. While there is no well-defined definition, most experts agree that it should include at least these subjects:

1. A basic understanding of computers and what they can and cannot do.
2. A minimum level of skill in programming a computer in a popular language.
3. Some understanding of the impact computers will have on our future lives.

Table of Contents

Section I Some Helpful Background Information

Chapter 1 Welcome to Personal Computing

Chapter 2 An Introduction to the TI Home Computer

Section II An Introduction to BASIC Programming

Chapter 3 The Nature of Computers: Programmer's Perspective

Chapter 4 Algorithms–The Computer's Recipe

Chapter 5 Writing and Editing BASIC Programs

Chapter 6 TI BASIC Calculations

Chapter 7 Input and Reading Data

Chapter 8 Program Branching and Loop Structures

Chapter 9 Arrays

Chapter 10 An Introduction to Sound and Color-Graphics *or*: Fun with Your TI

Section I

Background

If you suffer from cyberphobia (fear of computers) or cyberphilia (obsession with computers), this book can cure both!

Microcomputers have become a mass-produced, consumer-oriented items. They are everywhere. Recently, at a mountain resort, a man was sunbathing by the pool, a portable microcomputer in his lap. He was preparing memos for an office meeting the following week and was obviously enjoying the best of both worlds. Technology had added another dimension to his work and play.

Welcome to the exciting world of microcomputers! How much do you know about computers right now? Take the computer literacy test in Chapter 1 and again when you complete the book. You should see the dramatic increase in your knowledge and understanding of computers and how they work! You will also be pleased with how much you have learned about the Texas Instruments home computer.

Knowledge of the microcomputer industry and how people use them increases your comfort zone with computers and gives you ideas of how to use your computer in your home and career. Turn your imagination loose as you read this chapter.

Chapter 2 introduces the Texas Instruments company and its home computer. It discusses the development and evolution of the home computer, the computer parts themselves, and some of the computer programs available for the system. This computer is expandable. As your knowledge, abilities, uses and budget grow, you can add devices to the computer console unit to help carry out those applications. Let's begin—with a brief orientation to the personal computing industry.

Chapter 1

Welcome To Personal Computing

Welcome to the most fabulous realm of life in the twenty-first century–personal computing. It is to the information revolution what the auto was to the industrial revolution and your life may never be the same. Before you begin, take the test in Figure 1.1. When you complete the book, take this test again to see how much you have learned.

THE TECHNOLOGY

What's a half inch in length, a half inch in width, as thick as a dime and has revolutionized the computer industry? The microcomputer *miracle chip*! Computers–that only 25 years ago filled rooms with vacuum tubes and mazes of wires–are reproduced today at very low cost in the postage stamp-sized integrated circuit chip. This technology has spawned a new industry–the microcomputer industry. Had the auto industry advanced as rapidly as computers, a Rolls-Royce that today costs $85,000 would instead cost $3.00, get two million miles per gallon, and deliver enough horsepower to drive a battleship. And the trend continues. The keys to the success of this industry are portability, miniaturization, and low cost.

This book specifically concerns personal computers developed by Texas Instruments, Inc.; as versatile home computers within the financial reach of our society. The *machine* will become the home computing and information center as well as a link between the home and our electronic society. It is what the telephone was to the home 100 years ago. Microcomputers and robots will no doubt cause many social dislocations–and opportunities. Author James Martin predicts that the microcomputer industry will make thousands of new millionaires.

Since Apple Computer first introduced its personal computer in 1976, the market has grown from nothing to over $6 billion in seven years. Estimates are that by 1986 sales will exceed $21 billion. At an average cost of $1000, this would mean 21 million computers in operations, worldwide. And most people feel this is only the beginning. Costs will continue to drop and capabilities will continue to increase as time and technology advance.

Answer each question; 5 = Complete Understanding . . . thru 1 = No Understanding.

Question	Now	After Reading Book
1. What is a computer?		
2. What is a computer program?		
3. What is a "canned" program?		
4. List 10 computer applications?		
5. How have computers affected our lives?		
6. What do computers do best?		
8. What are RAM and ROM?		
9. What is a "byte"?		
10. What do you know about Texas Instruments?		
11. What does a programmer do?		
12. What is a flowchart?		
13. What does BASIC stand for?		
14. What does an IF/THEN statement do?		
15. What's involved in "debugging"?		
16. What does the **CALL SOUND** command do?		
17. What is a computer file?		
18. Name some future applications of software		
19. How much do you know about your TI computer?		
20. Are you glad you bought this book?		
Total		
Average		

Figure 1.1 The computer literacy test.

Some Computer Applications

But what use do microcomputers have? Consider this:

It was the sort of situation that gets people killed.

> A troubled man had taken his wife and infant daughter hostage in a trailer house in Cottonwood, Arizona. They subsequently escaped but the man had a rifle and had threatened to kill anyone who tried to capture him.
>
> A little, one-armed robot rolled up to the trailer, and explained in a monotone voice that the man should give up. The robot spun on its wheels and left to return seconds later. It deposited a ringing telephone and announced that the police chief was calling. A half hour later that man surrendered—marking an historic first for computer technology and law enforcement.
>
> Never before in America had anyone been brought to a peaceful surrender with the aid of a robot. As the man was arrested, he commented that he thought his experience with the four-foot-tall 210-pound, six-wheeled machine was neat.
>
> "It broke the ice and took my mind off my problems," he said.

View computers as tools to help the human do his work. A good tool saves time, money, and energy. A screwdriver helps us loosen a fastner we otherwise could not remove. A pulley helps us achieve a mechanical advantage and lift a load we otherwise could not handle. Computers should give us a mental advantage or relieve us of the boredom from tasks we otherwise could or would not do—manually figure each payroll check for 1000 employees, for example. Computers should be used to complement human skills. Ideally paired, man and computer make a powerful team. Look at space travel, medical research, or medical technology. But how do we pair up? Which does which?

Figure 1.2 lists the advantages of computers and humans. Computers are extremely fast! Computer processing times are now measured in nanoseconds or billionths of a second. Within one-half second, a large computer can: a) monitor 100 heart patients and alert doctors to trouble, b) figure paychecks for 1000 employees, c) post 3000 checks to 250 different bank accounts, d) score 150,000 answers to a questionnaire and a few perform other chores!

But computers are dumb. You have to tell them what to do—and in very exact terms. They can do computations with a high degree of accuracy (take figures to many decimal places) and are very reliable. That means if a computer totals a list of 100 numbers 50 different times, it gets the same answer each time. Humans are notoriously bad at this activity. Computers can also quickly store, retrieve, and display large amounts of data. With these characteristics, computers excel in applications that require repetitive tasks where large amounts of data must be analyzed.

COMPUTERS		**HUMANS**	
Characteristics	**Applications**	**Characteristics**	**Applications**
High-speed computations	Date storage/ retrieval	Creativity	Design
Accuracy	Computations	Conceptual	Implementation
Reliability	Summarization	Sensitivity to Environment	Human Skills
Large storage capability	Classification and sorting	Mobility	Travel
		Multiple senses	Persuasion

Figure 1.2 Computers vs. Humans.

One major characteristic of computer applications is high volume of a large task done over and over, such as paychecks or a large number of requests for information from a collection of data such as airline reservations or bank account balances. In payroll, the process is called *batch processing* because the pay records are collected over a pay period and then run all at once. The bank or airline reservation system is a *query* or *on-line* application because information requests occur on a random or demand basis to a continuously updated data base.

Humans, on the other hand, identify the application for the computer, define how the computer is to solve the problem, and, ultimately, use the computer's output as part of the solution. Humans begin and end the computer application. Humans interact with other humans at home or work to design and implement the computer's output.

Computers supply the computational skills. Humans supply the management skills. The two together are greater than either one can be separately.

Whether the benefits of computerization outweigh the cost is a major consideration. Will the application pay for itself? You can estimate costs of hardware, software, personnel, supplies, furniture, and the physical space. But the benefits are much less tangible and very difficult to determine. Computers costing hundreds of thousands of dollars are often purchased with little formal evaluation of the computer's benefits.

In personal computing or desk top computing, many of the standard applications in the home or organization are already programmed and available in *canned* form for a reasonable price. Figure 1.3 lists the common personal computer applications and the software programs Texas Instruments has in those areas. If programs already exist for an application, it is usually smart to buy them rather than write an original program. Prepackaged programs can save a beginning computer user time and money.

Applications	**TI Software**
Home Management	Home Budget Management Personal Record Keeping Tax Planning
Word Processing	TI Writer
Electronic Spread Sheet	Multiplan
Education	Miliken Math Plato Scot Foresman
Family Entertainment/ Games	PARSEC Hunt the Wampus TI Invaders
Original Applications	TI BASIC Programming

Figure 1.3 Typical applications and canned software available.

WHAT MAKES A COMPUTER PERSONAL?

A computer is a general purpose, electronic machine that rapidly and reliably stores and processes data. This data may be numbers or alphabetical letters. Computers come in all sizes—some fill large rooms and cost millions of dollars, some are hand-held and sell for $49. Why such a difference?

Speed, storage capabilities and hardware accessories are the key differences. How fast a computer can transfer data from one part of the computer to another is called *access time* or *cycle time*. The faster the access time, the faster the computer. Large machines cycle in nanoseconds or billionths of a second. A nanosecond is the time it takes light to travel one foot. (Light travels at 186,000 miles per second or seven times around the world in one second.) Personal computers that transfer data in thousandths of a second are much slower.

Storage size affects two areas—word size and primary core memory. Word size is the storage area allocated to a single piece of data. The large IBM 370, model 3033 has a word size four times larger than the TI-99/4A and eight times larger than the original Apple II computer. A large machine can perform an operation in one move while a microcomputer may take several steps. Larger computers also access larger blocks of data from the general memory area and do so more rapidly.

Finally, more information can be stored in the memory of larger computers. The TI-99/4A has a 16K memory (*K* equals 1024 characters or "bytes"—one byte can store one character). Thus, approximately 16,000 characters can fit in the TI's memory at any one time. The characters may be changed during the program, but cannot exceed 16,000. With some accessories, the TI can be expanded, but the standard model is 16K. Other personal computers range in storage from 2K to 512K. Large, business and scientific computers may have millions of characters of primary memory area.

Price is the ultimate distinction. Personal computers normally cost less less than $5,000. In the $1,000 to $5,000 price range, Apple, IBM and Radio Shack are the leaders. These desk-top computers are used by small businesses and other organizations for record-keeping, word processing and company planning. The system typically consists of a keyboard, controller unit (heart of the computer), CRT (cathode ray tube) monitor, printer, and one or two disk drives. Purchased software programs run the machines although they can be programmed in BASIC (Beginners All-purpose Symbolic Instruction Code) language. Some of these systems find their way into the home for education, entertainment, and home management applications.

The market for computers costing less than $500 has exploded. Estimates are that 3.5 million units will be sold in 1983, 100 times the 1980 volume of 35,000 units sold. Texas Instruments, Timex-Sinclair, Commodore, Atari, and Radio Shack are the prime competitors. As technology improves and competition stiffens, prices will continue to tumble and capabilities continue to expand. The purchaser will continue to be perplexed by what to buy and when.

SUMMARY

The revolution in the electronic age has been phenomenal in the past 25 years. Room-sized computers shrank to desk tops, and costs shrank with them.

This put computer capabilities into the hands of the ordinary person.

Humans and computers work together. Computers do the boring work and don't complain. Humans are then freed to do the creative work. Computers complement human skills, but one becomes dependent on the other. Computers process in nanoseconds. They are dumb but accurate. They store and retrieve large amounts of data. Humans tell them what to do and manage them.

Costs vary from $500 to $5,000. The key differences that determine cost are their storage capabilities and hardware accessories.

The future looks bright for the computer industry. In the *electronic society*, more and more activities and services will be computer centered. Home computers today may be a luxury but will soon become our link to the outside world and our most necessary home appliance.

REVIEW ACTIVITIES

1. Visit a retail computer store. Ask about computer prices and capabilities. Review store brochures. Tell a friend about your experiences.
2. Read a recent issue of a computer magazine. Look at the ads and the story coverage. Can you understand the articles?
3. Watch your local newspaper for computer-related stories. Clip out a story and show it to a friend. Discuss the article and its implications for you, your friend and society.
4. List ten instances in which you could justify the cost and use of a computer in your life. Are they being done by someone else?
5. Discuss the future of computers with a friend. How do you see your life and society being changed by computers and information technology? When will it happen? What is the time frame?

Chapter 2

Introduction To the Texas Instruments Home Computer

BACKGROUND

Texas Instruments (TI) is the leading producer of semiconductor products. It develops, manufactures, and sells electronic equipment such as calculators, microprocessors, and small business and home computers. Headquartered in Dallas, Texas, Texas Instruments has 50 plants in 20 countries and was a $4.3 billion company in 1982. It employs over 80,000 people. Products include a variety of electronic devices for industrial, consumer and government markets. Over 30,000 shareholders own 23 million shares of TXN (New York Stock Exchange symbol) stock.

EVOLUTION OF THE TI HOME COMPUTER

Always a leader in electronic education, TI introduced the TI-99/4A home computer in 1979 at an original cost of over $1,000. A flat keyboard, limited software and an expensive price generated lagging sales for this unit. The 16,000 character, random access memory (RAM) machine was ultimately discounted to $650 before it was discontinued in 1981.

The TI-99/4A home computer succeeded the original version. It featured many extras—a standard keyboard wth upper and lower case lettering, an automatic key repeat function, keys designed to access specific computer functions, and expanded software applications. The price had shrunk to $525. That price continued to fall until August, 1982, when the basic unit sold for $299. TI announced a $100 factory rebate on all TI-99/4A home computer sales to bring the actual retail price to $199. With this incentive, the ensuing Christmas season, and an aggressive multimedia advertising campaign that featured Bill Cosby, sales soared from an estimated 30,000 units prior to 1982 until they passed 500,000 consoles and continued to grow. In March 1983, the company discounted the TI-99/4A again to bring the unit down to $149 after the rebate. Recently, prices have dropped to under $50.

By 1990, seven out of every 10 homes (nearly 50 million households) in the United States will have at least one personal computer.

HARDWARE FOR THE TI PERSONAL COMPUTER

Hardware refers to the physical machines or components that make up the computing system. They take up space, generate heat, and make noise, as they carry out the physical activities of computer application such as printing, storing, displaying, and transferring data.

The Console

The heart of the TI-99/4A Home Computer system is the system console. It comes standard with 16K (16,000 characters) of **RAM** (random access memory), a state-of-the-art 16 bit microprocessor and 26K of **ROM** (read only memory). A standard BASIC language premanently resides in **ROM**. The typewriter keyboard comes with upper and lower case characters. Graphics are possible in 16 different colors and sound in tones ranging from 110 Hz to 40,000 Hz can be programmed. Factory supplied with the RF modulator, the console can be attached to any color or black-and-white television set. The set serves as an output device or monitor for the console. Figure 2.1 is a photo of the internal components of the console that show the integrated circuits, the storage chips, and the keyboard module.

Some Accessories

Depending on your needs, you can configure your system with a variety of optional accessories available. Figures 2.2 and 2.3 show a complete system. Figure 2.4 explains the components you will need for various configurations.

First, you purchase the console. Then if you want to do word processing, you need a printer. If you want to store programs or data, you need an off-line storage medium like tape (regular cassette recorder and connecting cables) or diskette (requires peripheral expansion system, disk controller card, and a disk drive). Tape storage is much less expensive, but extremely slow compared to diskette storage. Because of the time and effort you put into your programs and the fact that you will lose them if they are not stored before the console is turned off, your second purchase should be a cassette recorder or disk drive system.

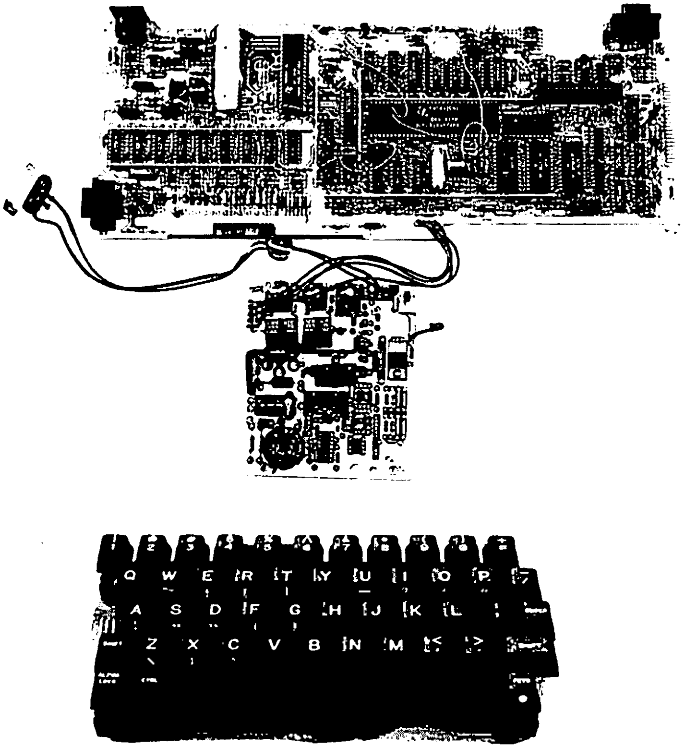


Figure 2.1 The Internal components of the TI-99/4A and the keyboard module.

Figure 2.8 shows that the peripheral expansion box, or P-box, as it is called, is the heart of any major system enlargement. This modular expansion system allows your computer to use diskette storage, printers, memory expanders, and telecommunications when combined with the device and its associated peripheral expansion card or circuit box. These expansion cards plug into the P-box.

Decide what you want your home computer to do. Then use these figures and approximate prices to configure your system and estimate its cost.

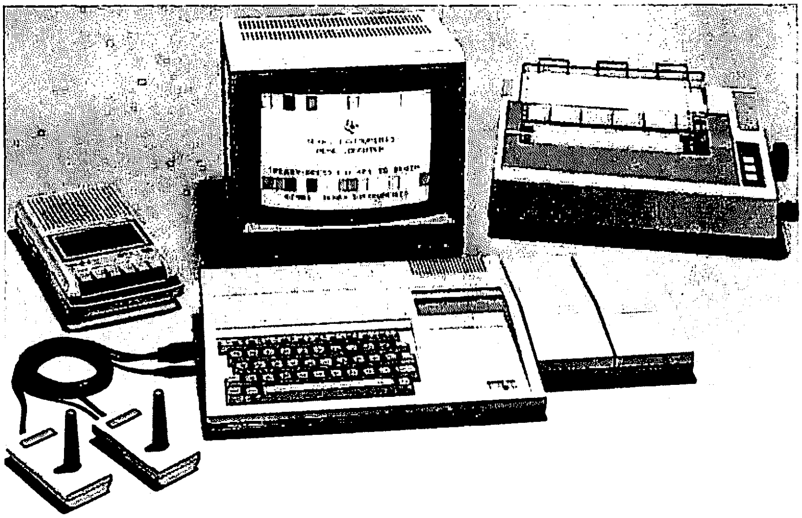


Figure 2.2 A complete system.

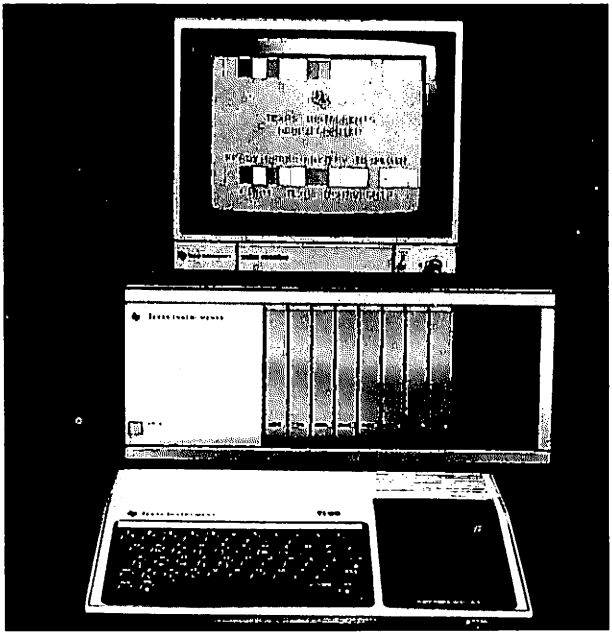
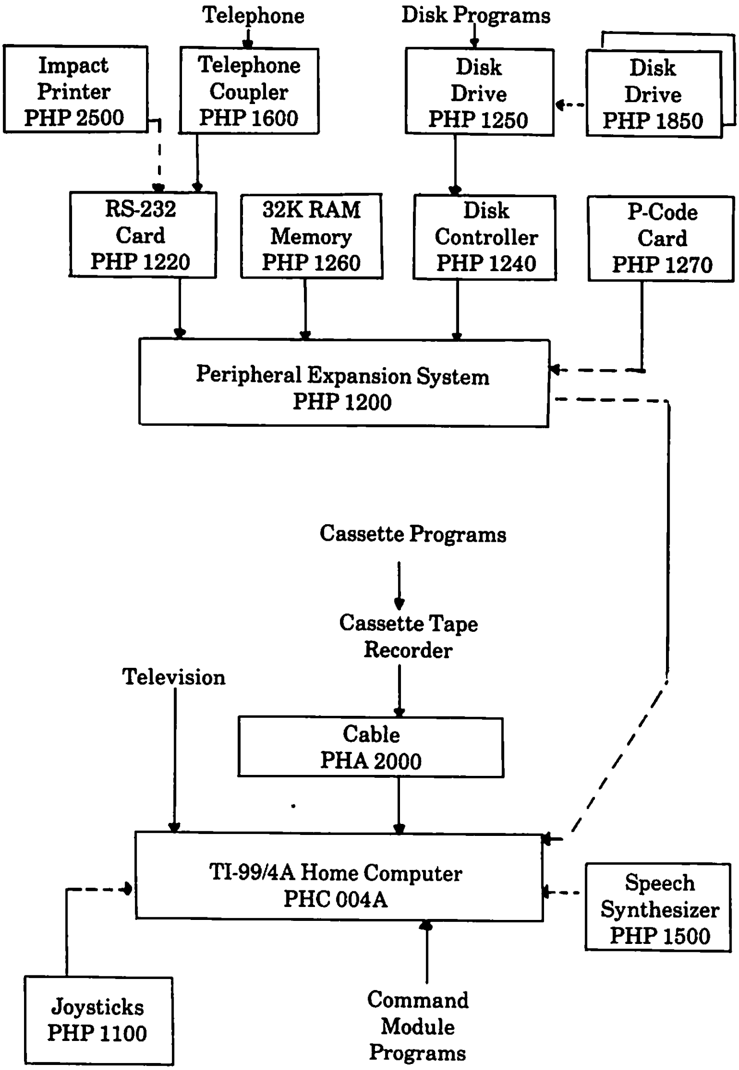


Figure 2.3 A complete system.



To use the diagram above, find the item you would like to add to your computer system. Then trace through the diagram to the computer itself. the things you go through are required for hookup. Example: If you want to connect a printer (PHP 2500) to the computer (PHC 004A), you'll need an RS-232 card (PHP 1220) and a Peripheral Expansion System (PHP 1200). Prices of the computer equipment (hardware) are listed.

Figure 2.4 Required components.

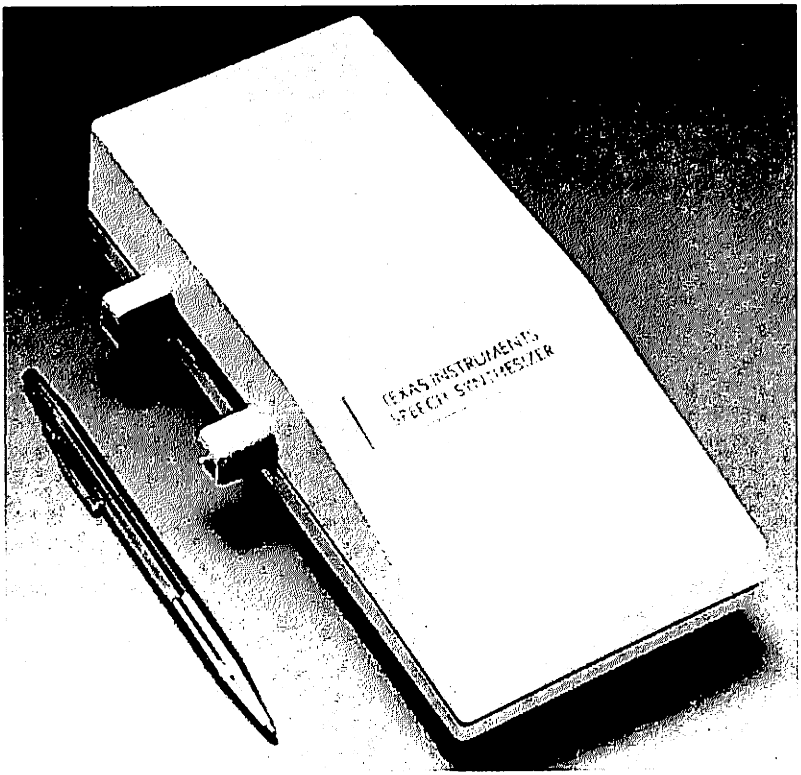


Figure 2.5 The speech synthesizer.

SOFTWARE

Programs that direct the computer to solve specific problems and oversee its operations are called software. This is the single most significant factor in a successful computer application. As a personal computer user you can either write original software for each application or buy already written *canned* software. You can use both types.

Prepackaged Software

If it can be purchased and is appropriate for the application, canned software is ideal. It may be expensive but has many benefits. Prepackaged programs are available for immediate use—no startup, no development time and no expense. The programs are usually well-documented (or explained) and error-free. User-friendly programs guide the first-time user through the entire process. You do not have to know a programming language. Popular prepackaged programs available for the TI-99/4A include *Parsec, Munchman, TI Invaders, Personal Record Keeping,* and *Multiplan.*

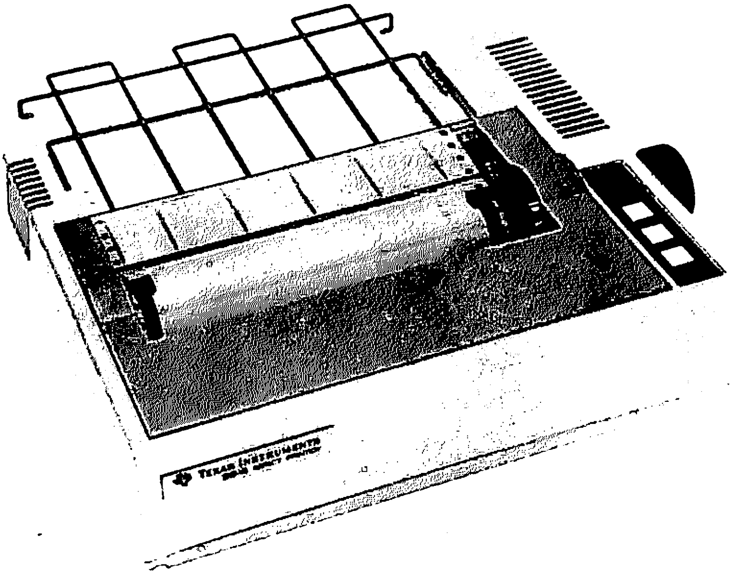


Figure 2.6 The TI impact printer.

Software Formats

Programs currently available for the TI-99/4A come in three different forms—solid-state software or command modules, cassettes, and diskettes. The right form for you depends on your type of computer hardware. If you only have the console and a TV monitor, command modules are your only choice. These are application programs in high demand that have been programmed into a ROM (read only memory) chip. When a command module is inserted in the console, the entire instruction set is available to the user. Command modules are sometimes called firmware (as opposed to hardware or software) because they are *burned* into a silicon chip and the instructions cannot be changed. Command module prices range from $19.95 to over $100. Each module comes with a booklet that documents the operation of the program. This form of software is ideal for first-time users, children, and highly structured applications.

Other prepackaged programs are in cassette or diskette form. These are typically larger programs with smaller demand. The original program is simply copied onto blank cassettes or diskettes, packaged, and sold much like stereo tapes and records. Your computer must be equipped with the required hardware devices to utilize either medium. TI offers a large supply of excellent application software in cassette and diskette format. And the number is constantly growing. Estimates are that within a year, over 1,000 application programs will be available for the TI-99/4A.

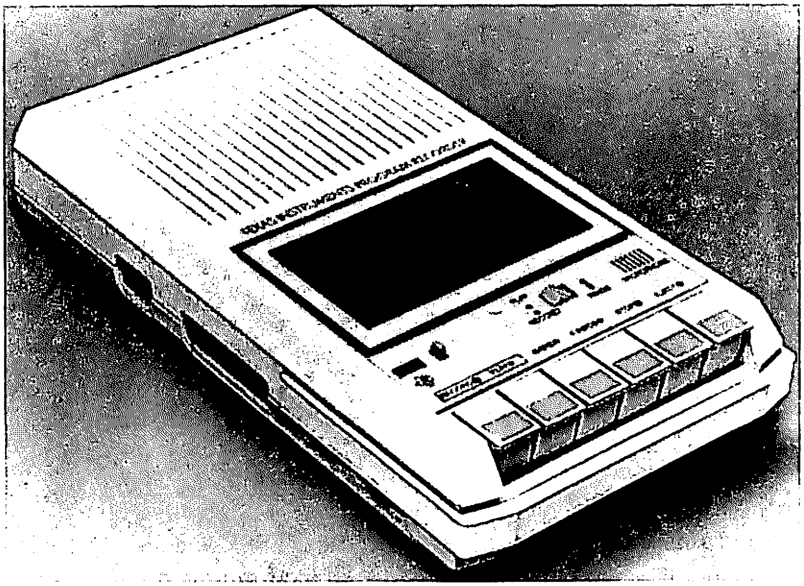


Figure 2.7 The TI program cassette recorder.

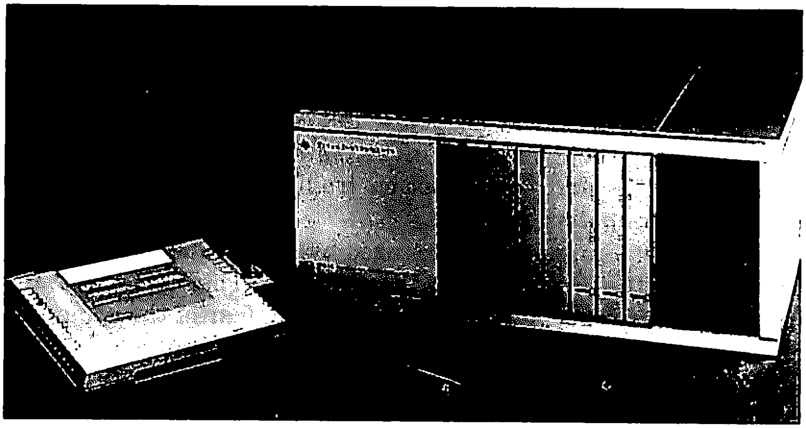


Figure 2.8 The peripheral expansion system.

Application Areas

Home computer application programs fall into three distinct classes–education, home/financial management, and home entertainment. There is a variety of software in each category for the TI-99/4A. Some of these programs were developed by Texas Instruments and others were developed by private companies for the retail market.

The educational benefits of home computers can justify their purchase. According to studies, it takes approximately 140 contact hours in the classroom to advance one grade level in most subjects at the elementary level. Computers and educational software can cut that time to 40 hours. Gifted children can advance at their own pace. Slower children get personalized attention. And the computers fill the learning environment with sound, color, voice, graphics, and positive reinforcement. We have worked with four-year-old children, teaching them fundamentals of arithmetic and reading. Watching their excitement and enthusiasm is pure joy. In time, programs will be available for more advanced subjects–calculus, physics, statistics, foreign languages, biology, etc.

TI has software to suit many daily needs. Our home, our finances, and our body can all be managed better by computers. Programs that track personal and household expenses and budgets can strengthen personal finances. Meal and diet planning programs can lead to better health. A complete exercise program on a command module (see Physical Fitness) is tailored to age and current physical condition. Time management programs help allocate our most precious resource–time. Many people use home computers for their personal investing decisions. Mortgages, financing, stocks and bonds, option trading, and accounting are some popular applications.

Home entertainment is epitomized by the arcade games. They are typically the first programs the user buys. They are a first step in computer literacy. Such games eliminate fear of the computer and show that computers can actually be fun. You can become familiar with the keyboard layout and functions. Once the games are mastered however, you should pursue more valuable uses for your computer or it will not reach its fullest potential value for you.

SOURCES OF HARDWARE AND SOFTWARE

Hardware and software are both expensive. It is important to make sure the hardware and software you buy will do the job you need done. Always try the equipment and programs before you buy them. Observe the program's ease of use or user friendliness. Does it guide you through or must you be totally familiar with the program's operations before you can start? Do you know other people who have used the equipment and program and are they satisfied? Scan through the program's documentation manual. Is it easy to read and follow? Are there examples? Does the program do what you need done? Is it worth the price? The same is true for hardware?

Retail Stores

Retail stores are the primary source of hardware and software. K-Mart, Jafco, Toys-R-Us and J.C. Penney are just a few. Regional and local stores also handle TI computers. Prices vary substantially from one store to another. Watching newspaper ads and a few phone calls will result in large savings. Visiting the retail store will allow you to try various software programs and assess their value.

Service is another consideration. If you have questions or problems with your system, a knowledgeable outlet can help. Working with computers can become frustrating and having a friend to turn to will ease these frustrations.

Mail Order

Mail order houses can offer substantial discounts on TI components and programs. They deal in volume sales on a national level. Discounts of up to 25 percent are possible. Be sure you know what you want! Once you open the package, few mail order houses will accept returned merchandise. Also make sure the firm is reputable. Most vendors require prepayment and have long delivery times. Once you send your payment, you must rely on the organization's integrity to make delivery.

Magazines and newspapers are prime sources for names of mail order firms. The *99'er Magazine*, discussed later in this chapter, is an excellent reference. Most firms publish price lists in their ads so you can do comparative shopping. Most accept checks or credit cards. Note expected delivery times and who pays the shipping charges. Small, inexpensive items may cost more to ship than they are worth. Local and national newspapers typically have a classified section dealing with computers. Advertisements cover equipment, sale items, services retailers and mail order firms.

Tronics, Inc.

Tronics Sales Corporation, located in Fort Worth, Texas, is the largest single distributor of the TI-99/4A. Founded by Jody Black, the firm uses a vertical or pyramid marketing structure to sell and distribute electronic equipment for the home through Tronics distributors. Once established, a distributor finds other people who are interested in buying equipment and software, sells to them and then they in turn, solicit others, forming the pyramid. The program has been successful. Nationally, there are over 10,000 Tronics distributors. Each may be considered an independent businessperson. Distributors are paid based on the credit volume or sales they make plus a percentage of the volume of the distributors down the line from them. Tronics is in no way affiliated with Texas Instruments, Inc., except as a distributor. For more information about the organization, write:

Tronics Sales Corporation
2563 East Loop 820 North
Fort Worth, Texas 76118

Each of these alternatives has its advantages and disadvantages. Take care in selecting the one best suited to your particular needs. You can save money by carefully shopping among these choices.

KEEPING INFORMED

Once you buy a TI Home Computer, how can you keep informed of new developments for your unit? Explore books, reference manuals, magazines, user groups and clubs to help you better utilize your new computer.

What Comes with the Computer?

When you buy your TI-99/4A, you will receive three helpful items: the *Beginner's BASIC* manual, the *User's Reference Guide* and the Reference Card. *Beginner's BASIC*, a step-by-step hands-on approach to learning the TI BASIC language, covers some introductory statements in BASIC and shows the results when they are run on the computer. Printing, computations, looping, color, and some graphic capabilities are briefly covered. The book does not cover the most difficult aspect of programming–how to write your own original program. This book provides that help.

The *User's Reference Guide* explains each individual statement in TI BASIC. The manual shows how each statement is formed (the "syntax" or structure of the statement), all the variations allowed in each statement and the variety of statements available. When you make a programming mistake, the *Reference Guide* helps you find the error. It explains 82 TI BASIC commands and error messages are discussed in detail in Section III of the Appendix.

The third helpful item in your packet of materials is the TI-99/4A BASIC Reference Card. As you become more familiar with programming, you will only need a quick refresher on the form of a particular statement. Does it need a comma or a period, an apostrophe or a quotation mark? The quick Reference Card shows this as well as the complete vocabulary of statements, color codes, and character codes for graphics. It is an excellent resource and you will use it frequently. These documents will make programming your TI-99/4A easier and more fun.

Magazines

A number of personal computer magazines are now being published. Many are directed at the more expensive computers, those over $1,000 and often used in business or organizations. These inlcude *BYTE*, *Creative Computing*, *Personal Computing*, *PC Magazine*, *PC World* and *Interface Age*. Typically the articles deal with the IBM PC, Apple, Radio Shack, and Osborne computers as well as software programs for those machines. Not much is discussed about the TI-99/4A, Commodore or Atari.

Our favorite magazine is the *99'er Home Computer Magazine* that covers only Texas Instruments home computers. Published monthly, the newsstand price is $3.50. An annual subscription is $25 in the U.S.A. Their address is:

99'er Home Computer Magazine
P.O. Box 5537
Eugene, OR 97405
(503) 485-8796

A recent issue covered such topics as reviews of TI's new CC-40 Compact Computer, robots, new computer games, LOGO, and computers and the handicapped. Many program listings for applications and games appear in the magazine. You can simply type in the listings and save them on your cassettes, or, for a nominal fee, purchase preprogrammed cassettes containing all the programs in one particular issue, which are offered by the magazine. Advertisements show the newest hardware and software available for your home computer. The articles are interesting and educational. It is a good way to keep informed.

Texas Instruments Computer Advantage Club

To help meet the challenge of the expanding need for computer awareness, Texas Instruments has established its own Computer Advantage Club. The objective is to give hands-on training with the TI-99/4A Home Computer. Training sessions are for adults adn young people ages 8 and up. Club members receive both small group and individualized instruction in computer operation and applications. The clubs are active in 27 cities throughout the nation and their goal is to be in 100 cities by the end of 1983. They charge an education fee for each participant. For more information on the club nearest you, call toll free:

Outside Texas: 1-800-858-4096
Inside Texas: 1-800-692-1318

WHY LEARN ABOUT COMPUTERS? *OR* OPPORTUNITIES!

In July 1982, William Turner, Texas Instruments assistant vice-president and consumer products marketing manager, told reporters in Lubbock, Texas, that "a child or adult who is not computer literate will be a misfit in the late 1980s and 1990s." The computer "is a one-product alternative to multiple specialty products for each application in the household," he added. These statements are indicative of the impact of the computer field on our society. Those without computer literacy will surely feel disadvantaged in tomorrow's world.

Opportunities for computer literates are unlimited. Many people have found new, exciting careers in the computer field. In their book *How to Make Money with Your Microcomputer* (dilithium Press), Carl Townsend and Merl Miller discuss how to write computer articles, how to develop and sell software, how to open your own computer store, how to make money teaching others about computers, and how to operate a computer repair business.

A recent article in the *Wall Street Journal* highlighted Paul Lutus, 37, "Oregon's millionaire oracle of the computer age." Living as a "mountain hermit" for years in the mountains of southern Oregon, Lutus writes and sells computer programs. Lutus wrote *Apple Writer*, a computer program for word processing on the Apple computer, which makes between $5,000 and $7,000 a day in royalties for Lutus, as he receives 25 percent of the wholesale price of each program sold.

One evening Lutus was typing on the keyboard of his Apple computer and the keys became stuck. Lutus, totally engrossed in his work got up from his chair and realized that the cabin temperature had dropped below freezing. The keys on his computer had begun to freeze!

William Gates, at age 19, formed Microsoft, Inc., a software development firm in Bellevue, Washington. Last year his company's sales were estimated at $40 million, and IBM asked the company to write the operating system software for IBM's new personal computer.

Similar success stories abound. As the market grows for personal computers, new opportunities open up for creative and innovative individuals in our society. "After growing wildly for decades, the computer industry now appears to be approaching its infancy," states a U.S. Government report on computing. So get started!

Section II

An Introduction to BASIC Programming

Coding the solution solves, or at least clarifies most computer problems. But before you can code the problem, you must first develop a tentative way to solve the problem—an algorithm or sequence of steps that leads to the desired solution. This book includes program segments and problems that portray concepts which will help beginning programmers and indicate the principles of good programming practice.

A series of practical programming applications in each chapter unifies previous programming concepts and statements and gives computing tools that you can apply to personal computers.

WHAT IS BASIC?

BASIC the acronym for Beginnners All-purpose Symbolic Instruction code, is the primary language for the TI and all other personal computers. It was developed at Dartmouth College to be an easy language for people who have had no experience with computers or computer language. Each statement begins with a keyword in plain English. These are such words as READ, LET, FOR, NEXT, etc. The word stands for the process which that statement performs. READ reads numbers or letters into the computer. FOR tells the computer to repeat a portion of the program, etc. This section discusses program design and writing programs in BASIC especially for the TI computer.

Chapter 3

The Nature of Computers: Programmer's Perspective

Most people initially believe that computers are extremely intelligent, artificial life-forms that they must learn to coax into working for them. The facts are:

1. Computers cannot be coaxed but must be told exactly what to do.
2. They have I.Q.s of 0 (zero) and need to have a specific set of instructions before they can do anything.
3. Computers remember instructions only for as long as they are turned on.

These sets of instructions are programs. Given a program, or set of programs, the computer becomes a tool which may act as a fast, reliable, and inexpensive source of labor or entertainment. Without programs, the computer is worthless to us.

You can write programs for the TI-99/4A then give them to the computer in a variety of ways. You need to know where the computer keeps its programs, how it may be given its programs, and understand how to write your own programs for the computer to follow.

The computer has memory, which is the space to store the computer's program, and the results, or answers (data), obtained by the computer from following the instructions in that program (executing the program).

Units called *bytes* measure this memory space. A byte of memory is roughly equal to the memory needed to hold one character, such as the letter A or the number 3. One thousand bytes make up a single *K* of memory so that a computer with 16K of memory is able to hold 16 thousand characters.

Let's say, for instance, your name is Chris and you have written a program that instructs the computer to do two things: 1) ask what your name is, and 2) remember it.

The program itself uses memory. When the computer executes your program, it asks what your name is. When you type in your name, the computer has data to put in its memory. That data will use approximately five more bytes of the computer's memory since "Chris" has five characters in it.

RAM & ROM MEMORY

There are two types of memory inside the computer:

1) a portion of memory called *Read Only Memory* (**ROM**) and,

2) the *current memory*, usually called *Random Access Memory* or **RAM**.

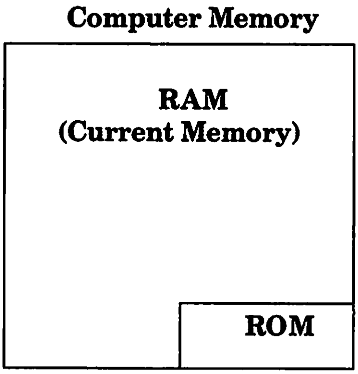


Figure 3.1 Computer memory.

ROM Memory

ROM is memory that has special computers permanently etched into it. These programs cannot be changed, added to, or erased. **ROM** can only hold programs, not data.

The programs of the **ROM** memory inside the TI-99/4A relate, for the most part, directly to the computer rather than anything that you might want the computer to do. This **ROM** holds the sets of instructions the computer needs to operate correctly. That is, many of the program in this **ROM** are there to help the computer get started when it is first turned on. They instruct the computer to check all of its parts to see if any are missing or not working, check to see what, if any, attachments are connected to it, and perform the other housekeeping chores necessary for the computer to function properly while it is being used.

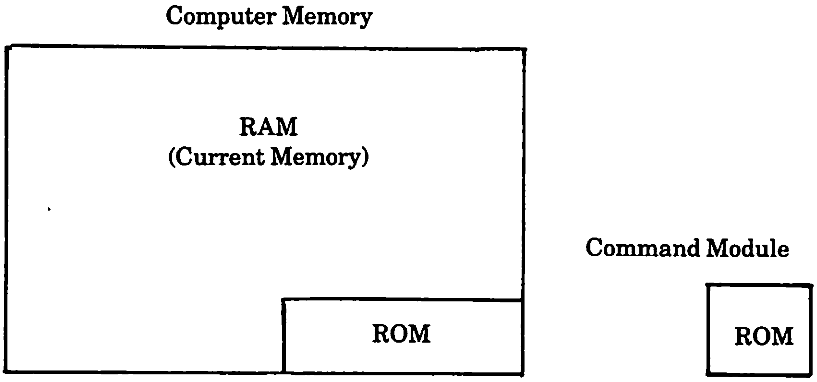


Figure 3.2 Computer memory.

The **ROM** memory inside the TI-99/4A also contains a special program called TI BASIC. This set of instructions in necessary for the computer to have while you write your own sets of instructions (BASIC programs) for the computer to execute.

ROM Software

Software is another name for programs, or sets of instructions, the computer follows in order to work for you. There are a number of Command Modules available for the TI-99/4A. These modules are actually **ROM** memory chips that may be attached to the computer by inserting them into the module outlet. These software modules expand the **ROM** of the computer to include the programs contained in the **ROM** of the Command Module.

RAM Memory

The computer uses its **RAM**, or current memory, as another place to hold its programs. When the computer is turned off, however, this part of its memory is wiped clean. Since **RAM** memory is temporary in nature, it is called the computer's current memory.

RAM is used to hold other types of software available for the TI-99/4A and the data that the computer might obtain from executing any program. **RAM** space is also used to hold your BASIC programs. **RAM** is the portion of computer memory which is extremely versatile, capable of holding a variety of programs and data. It is, however, only temporary memory.

A third type of memory, magnetic memory, must be used if you want to permanently store the programs and data that the computer's current memory temporarily holds.

Magnetic Storage

You can save your programs and data on some type of magnetic storage medium, which, in the case of the TI-99/4A, can be either a cassette recorder's tape, or a computer diskette.

By keeping programs or data stored on this magnetic storage, you overcome the problem of the computer's inability to remember the programs or data held in its **RAM** after it has been turned off. To do this, you simply give the computer the programs or data it needs from the magnetic storage each time it needs them.

Magnetic tape or diskettes act as a computer's filing cabinet to hold its programs and data while it is turned off. In fact, we refer to the individual programs and sets of data on magnetic storage as files. (Refer to Appendix I for saving BASIC programs.)

Software available for the TI-99/4A, other than **ROM** chip Command Modules, are programs written on magnetic storage of some type, either cassette tape or diskette.

Loading the Current Memory

In general, when you want the computer to have a particular set of data or a program in its current memory, you connect the device (either cassette recorder or diskette drive) to the computer, and type in the appropriate command instructions for the computer to execute. The computer then:

1. finds the data or program on the tape or diskette,
2. reads what is there, and,
3. memorizes that set of instructions or data.

We call this *loading* a program or reading a data file.

When you load a program, or have the computer read data from magnetic storage, part of the computer's current memory is used to hold a copy of that program or data. As you write your programs for the computer, the effect is the same. That is, as you type in your program, the computer's current memory holds your set of instructions as you type them in.

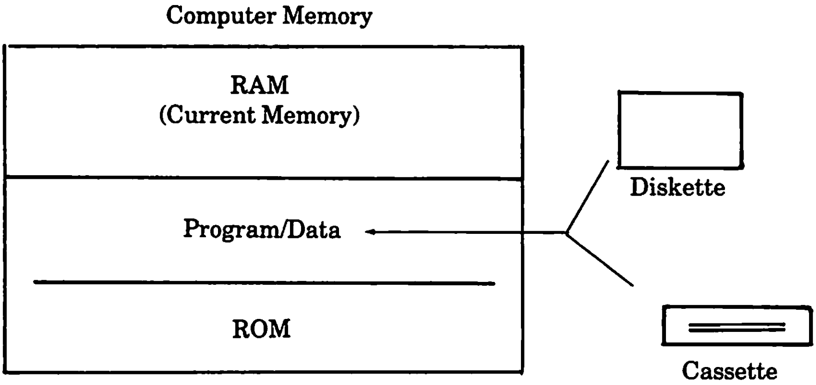


Figure 3.3 Computer memory.

PROGRAMMING LANGUAGES

A program, the set of specific instructions for the computer to follow or *execute*, must be written in a form that the computer can understand, i.e., in a programming language. Actually, the computer only understands one language, appropriately called machine language. This is an example of how we might visualize machine language:

```
11001011  01111001  01001001  10010111  11001011
01001001  10010111  01001001  01001001  11001011
01111001  01001001  10010111  01001001  01001001
10010111  10010111  01001001  01001001  10010111
```

Some people have the technical knowledge to write programs in this highly complex pattern of 1s and 0s. In fact, the programs in the computer's **ROM** and most of the TI software programs are written in this way.

Fortunately, these same people write special types of machine language programs that allow you to write your programs in a way you can read and understand.

These machine language programs are called programming languages. But from the computer's perspective, they are programs that interpret what you type on the keyboard into something the computer can understand, i.e., 1s and 0s.

Thus programming languages are simply go-between programs that aid you in your attempt to communicate your instructions to the computer. Your instructions are interpreted for the TI-99/4A with a programming language called TI BASIC.

TI BASIC is one of the more than 50 versions of BASIC in existence today. Although each version of BASIC has its individual characteristics, the similarities are far more striking than the differences.

When you first turn on the TI-99/4A, a menu screen is presented after the title screen to indicate that you should press the number 1 (one) key FOR TI BASIC. When you type 1 (one), the screen clears and the message, TI BASIC READY appears. At this point you are at the **ROM** BASIC operating level.

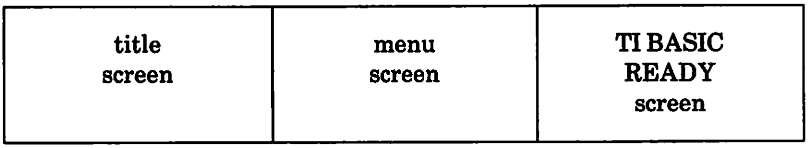


Figure 3.4 The first three screens.

OPERATING LEVELS FOR THE TI-99/4A

ROM memory holds the programming language program TI BASIC. When you press the number one key you tell the computer to follow the instructions in this program. As long as the computer is following those instructions, you can communicate with it by typing in the commands and statements of BASIC. You can also begin to write your own set of instructions, a BASIC program, for the computer to follow. When you write your BASIC program (or load one from the magnetic storage), you create another level for the computer, the BASIC program operating level. When you finish writing or loading your BASIC program and are ready to have the computer begin executing it, type **RUN**.

At this point the computer jumps from the **ROM** BASIC operating level to the BASIC program operating level. That is, the computer starts following the instructions in your own BASIC program. The computer will stay at your program's operating level until:

1. the set of instructions ends, or,
2. the computer finds a particular instruction that was written incorrectly, or,
3. you intentionally interrupt the program's execution by pressing the FCTN/4 (CLEAR)keys on the keyboard.

You know the computer has finished executing your instructions without finding a single incorrect statement when the message:

*** * DONE * ***

appears on the monitor screen.

If the computer finds an instruction that was typed incorrectly or that it simply does not understand, it will give you a different message, an error message. Several error messages may appear, and a list of them along with an explanation of what could have caused them appears in Appendix V.

If you interrupt the execution of your BASIC program by pressing the FCTN/4 (CLEAR) keys, the computer stops executing your program and gives you a BREAK POINT message that tells you what instruction it would have executed next.

In any case, when the computer stops executing the BASIC program, it drops back to the **ROM** BASIC operating level where you can either change instructions, add instructions, or store the finished program on magnetic storage.

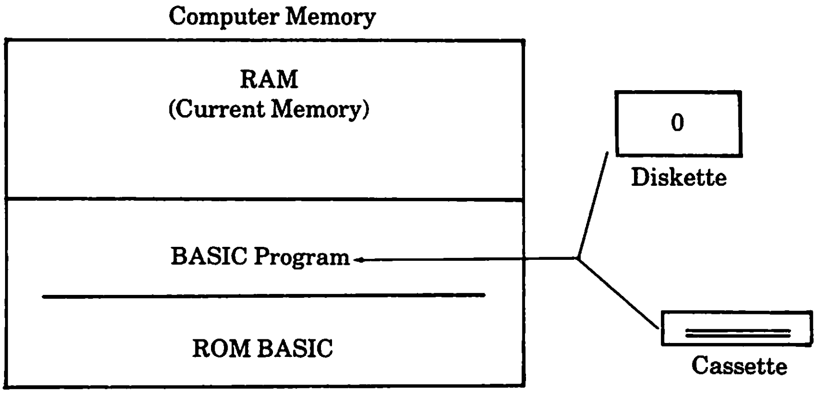


Figure 3.5 Computer memory.

MORE THAN ONE BASIC PROGRAM

You can only have one BASIC program at a time in the computer's current memory. This means that if you want to write or use another BASIC program, you first have to erase any BASIC program that is currently in **RAM**.

If you are not going to store the BASIC program, or already have stored it on magnetic storage, you can type **NEW**. This BASIC command tells the computer to erase the BASIC program from the current memory but to stay at the **ROM** BASIC operating level so you can type in a new BASIC program.

Instead of typing **NEW**, you can type **BYE**. The command **BYE** tells the computer to erase the current memory and leave the **ROM** BASIC level. **BYE** causes the computer to return to the title screen you started with. If you load a BASIC program from magnetic storage, the computer automatically erases any BASIC program it may have in its current memory before it loads the new program.

Either way (typing **NEW**, **BYE**, or loading a program from magnetic storage), any BASIC program in the current memory will be lost forever if you have not saved it on magnetic storage.

SUMMARY

You must realize the difference between intelligence and memory if you are going to understand the computer. Man has intelligence, the computer has only memory. As you learn how to use the computer, advanced as it is, you will gain a new appreciation for the "90 percent water-based grey-matter computer" you have used every day of your life.

This chapter introduced several key tems and concepts that you will be using in the remaining chapters on BASIC programming for the TI-99/4A. The following is a brief summary of some of the more important ones.

KEY TERMS AND CONCEPTS

PROGRAM	A set of specific instructions for the computer to follow in order to perform a task.
EXECUTE	The computer's process of following the instructions in a program.
DATA	Words or numbers generated during a program's execution that need to be retained

	in the computer's memory or stored on magnetic storage.
BYTE	A unit of memory space equal to one character.
K	One thousand bytes.
RAM	Random Access Memory or current memory. Used to hold programs and/or data on a temporary basis.
ROM	Read Only Memory. The portion of computer memory with programs permanently etched into it.
SOFTWARE	Another name for a program but implies that it is a purchased program.
COMMAND MODULE	A **ROM** memory chip containing software for the TI-99/4A.
MAGNETIC STORAGE	The means to save programs and data for the computer's current memory.
FILES	Sets of data or individual programs stored on a magnetic medium (cassette tape or diskette).
BASIC	A machine language program that is used to interpret English type instructions into ones and zeros.
OPERATING LEVEL	The set of instructions (program) that the computer is executing at a given moment.
RUN	The BASIC command that tells the computer to start executing the BASIC program that is in its current memory.
NEW	The BASIC command that erases only the BASIC program that is currently in the computer's current memory.
BYE	The BASIC command that instructs the computer to leave the **ROM** BASIC operating level. Any BASIC program in **RAM** is

	erased and the computer returns to the title screen.
FCTN/4	The two keys that, when pressed together, interrupt the execution of a BASIC program.
*** * DONE * ***	The message the computer gives when it has completed the program without finding an incorrect statement. Otherwise, the computer gives you an error message.

CHAPTER CHALLENGE

1. What sets of instructions does the computer follow in order to do anything?
2. When the computer is following the instructions that make up a program, what is the process called?
3. Inside the computer is memory which is simply room to store the computer's programs and the results of executing programs. What are these results called?
4. Memory space is measured in units called what? And 6.5K equals how many of these units?
5. What two types of memory are inside the computer?
6. What kind of memory has programs permanently etched into it and is used to get the computer started when it is turned on? What programming language program does it contain?
7. What kind of memory is temporary in nature and holds BASIC programs and many of the software packages available for the TI-99/4A?
8. Command Modules are actually what kind of memory chips that can be connected to the computer to give it software programs to execute?
9. What do you use to save the programs that are held in the computer's current memory? Name the two types.
10. What are programs that have been saved for the computer on magnetic storage called?
11. What is the only language that the computer really understands? What is the special program called that interprets what you type into this language?
12. What BASIC command causes the computer to start executing the BASIC program that is in its current memory?
13. At what BASIC program level will the computer be when it is executing a BASIC program?

14. How many BASIC programs can you have in the computer's current memory at a time?
15. If you type **NEW**, what kind of memory is erased? If you type **BYE**, the computer leaves what kind of operating level? Either way, what will be erased in the computer's memory?

ANSWERS TO EXERCISES

1. programs
2. executing
3. data
4. bytes; 6,500
5. **RAM** (or current) and **ROM**
6. **ROM**; BASIC
7. **RAM** (or current)
8. **ROM**
9. Magnetic storage; cassette tape and diskette
10. files
11. machine language; BASIC
12. **RUN**
13. operating
14. one
15. **RAM** (or current); **ROM** BASIC; BASIC program

Chapter 4

Algorithms – The Computer's Recipe

> *Al•go•rithm*–a rule of procedure for solving a problem
> that frequently involves repetition of an operation,
> *Merriam-Webster*, 1983

You are now ready to write BASIC programs for the TI-99/4A. First and most important, understand how the computer follows program instructions.

Even though BASIC is a programming language program that makes it easier to write a set of instructions for the computer, you cannot escape the ultra-logic of the machine. Your instructions must be absolutely precise and ordered in a logical sequence before the computer can execute them.

A set of instructions is an algorithm that conforms to the logic of the computer. It is the recipe to solve a problem. It represents the type and order of instructions that the computer will follow to execute a program that has been typed in. Algorithms are not written in the statements and commands of a programming language like BASIC but in a general form with their instructions in English. You can then translate these instructions into any programming language because the order and character of them is computer correct.

We can use the algorithm of a simple problem to demonstrate the three fundamentals of computer program execution. Imagine a robot computer that is just smart enough to understand English but needs a specific set of instructions before it can do anything. Unless the robot is instructed otherwise, it follows instructions, one after another and will not stop until you tell it to do so. The robot sits at a table with a deck of cards turned face up before it. The robot is supposed to go through the deck, pick out the four kings in the deck, and set them to its right. The robot will set any other card to the left. When it has found all four kings, it is supposed to stop. Study the following instructions for the robot to follow carefully.

```
Instruction 1–THE TOTAL KINGS SO FAR EQUALS ZERO
Instruction 2–READ THE TOP CARD
Instruction 3–IF THE TOP CARD IS A KING THEN GO TO INSTRUCTION 6 AND CONTINUE FROM THERE
Instruction 4–SET THE TOP CARD TO THE LEFT
Instruction 5–GO TO INSTRUCTION 2 AND CONTINUE FROM THERE
Instruction 6–SET THE TOP CARD TO THE RIGHT
Instruction 7–ADD ONE TO THE TOTAL KINGS SO FAR
Instruction 8–IF THE TOTAL KINGS SO FAR IS LESS THAN FOUR THEN GO TO INSTRUCTION 2 AND CONTINUE FROM THERE
Instruction 9–STOP
```

ASSIGNMENT–CONDITION–ITERATION:
Three Fundamentals of Program Execution

> *ASSIGNMENT*–instructions that tell the computer that something has a value.

The first instruction in our robot algorithm (THE TOTAL KINGS SO FAR EQUALS ZERO) is an assignment instruction that tells the robot it has found no kings. This would be obvious if you were doing this job, but that is not the case with computers.

The next instruction (READ THE TOP CARD) is also an assignment instruction. The computer is told that something (THE TOP CARD) will change in value as instructions are followed. THE TOP CARD may be any card in the deck, and the computer robot has to **READ** it to know its value.

Finally, instruction 7 (ADD ONE TO THE TOTAL KINGS SO FAR) is another assignment. Here you tell the computer robot to change the value of something (THE TOTAL KINGS SO FAR) and increase that value by one every time that instruction is executed.

In each case, the computer robot is instructed to assign a value to something so that the set of instructions can be successfully executed.

CONDITION–the decision-making ability of the computer. **IF** a condition is true, **THEN** the computer is instructed to **GO TO** another instruction. If the condition is not true, the computer moves on to the very next instruction.

In instruction 3, the robot is told that IF THE TOP CARD IS A KING THEN GO TO INSTRUCTION 6 AND CONTINUE FROM THERE. The condition is IF THE TOP CARD IS A KING. If the condition is true, then why tell it to go to instruction 6? Unless told otherwise, the robot will move to the next instruction which is 4 (SET THE TOP CARD TO THE LEFT). That is not what should be done if that card is a king. By sending the robot to instruction 6 (SET THE TOP CARD TO THE RIGHT) the cards end up either on the right or on the left, depending on their values, king or no king.

Every time the robot finds a king, it executes instruction 6, then instruction 7 (ADD ONE TO THE TOTAL KINGS SO FAR).

The next instruction, 8 (IF THE TOTAL KINGS SO FAR IS LESS THAN FOUR THEN GO TO INSTRUCTION 2 AND CONTINUE FROM THERE), is the other conditional instruction in the algorithm. This means that the robot continues to go through the deck of cards until it finds four kings. When it has found four kings, THE TOTAL KINGS SO FAR will not be less than four. It will be equal to four and the condition will no longer be true. The robot will move on to the very next instruction, 9 (STOP).

ITERATION–an instruction that causes the computer to execute one or more instructions over again.

There are two places in the set of instructions where the robot is told to GO TO INSTRUCTION 2 (READ THE TOP CARD) AND CONTINUE FROM THERE. This starts the whole process over with a new TOP CARD. When an instruction causes a repetition of a process (called a loop) it causes iteration, a very powerful characteristic of computers. The instructions could have had the computer read a card, set it to the left or the right and stop when it found all four kings, then repeat those same instructions 52 times in the algorithm. Instead, iteration uses the same set of instructions over and over until the job was done.

THE NATURE OF COMPUTER PROGRAMS

In the robot algorithm there were three elements: INPUTS, the original deck of playing cards; a PROCESS, sorting them by whether or not they were kings; and OUTPUTS, two stacks of sorted cards, one on the left and one on the right.

All computer programs share these same three elements. The three fundamentals of assignment, condition, and iteration make up the model which defines the nature of computer programs.

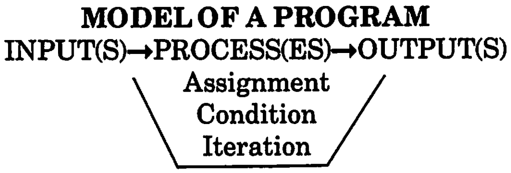


Figure 4.1 A model of a program.

This model shows that computer programs have INPUTs, which can be numbers, names, even pictures that the computer can *see* with the aid of optical scanning devices. It tells us that these INPUTs must be defined with values that can be ASSIGNED to a memory location within the computer.

Once the INPUTs are in the computer's memory, it can compare them with each other through CONDITION. It can also perform addition and other mathematical operations on them, sort them alphabetically, search for a particular INPUT, or perform a great number of PROCESS(ES), many of which will be covered in the following chapters.

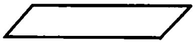
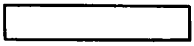
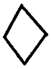
It can perform these PROCESSes over and over through ITERATION, one of the strongest features of the computer, and finally, OUTPUT the results in a form that can be understood and used.

FLOWCHARTS

Another way to define the set of instructions that were given to the robot is with a diagram.

Established symbols represent the various instructions used in computer programs. These symbols are combined to illustrate the program in a *flowchart*. Although there are many of these symbols, you need only three to flowchart most programs.

THE IMPORTANT FLOWCHART SYMBOLS

1.  The INPUT/OUTPUT symbol
2.  The PROCESS symbol
3.  The DECISION symbol

These symbols are connected together with arrows to indicate how you are to execute the program, and have short explanations printed beside them to tell more specifically to what the symbols refer. The following flowchart is for the robot computer algorithm.

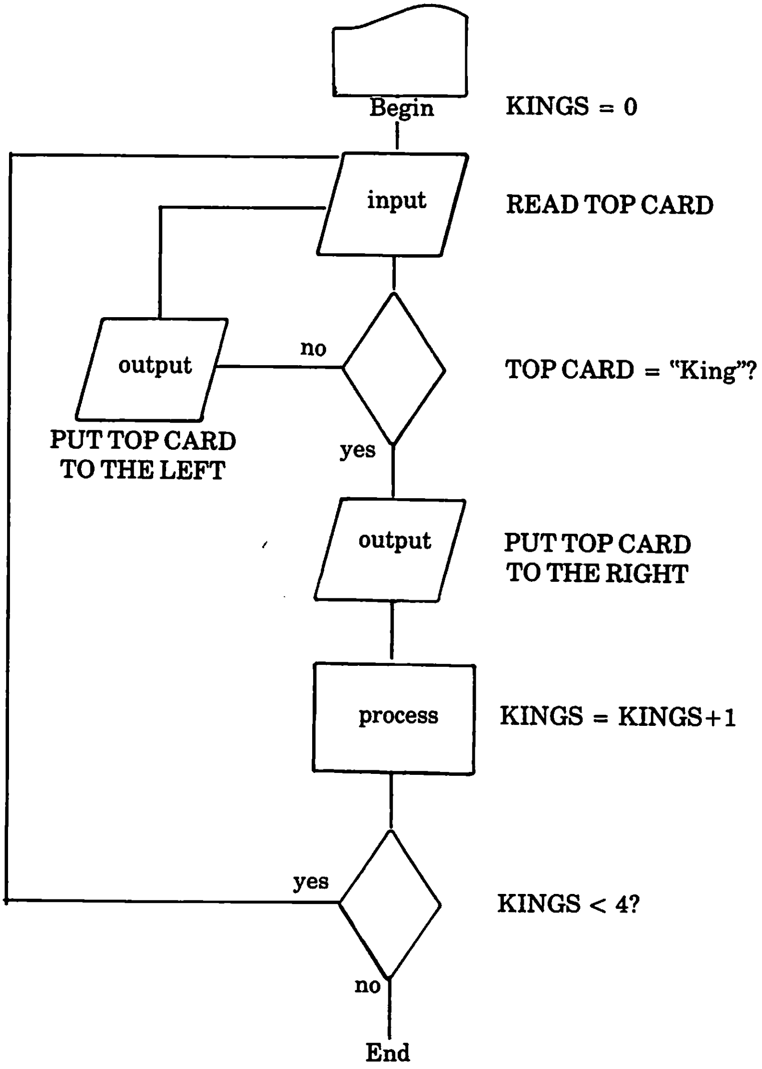


Figure 4.2 The robot computer algorithm flowchart.

THE BASIC PROGRAM

```
10 LET KINGS = 0
20 READ TOPCARD$
30    IF TOPCARD$ = "K" THEN 60
40    PRINT TOPCARD$
50 GOTO 20
60    PRINT TAB(20);TOPCARD$
70    LET KINGS = KINGS+1
80 IF KINGS < 4 THEN 20
90 DATA 2,3,4,5,6,7,8,9,10,J,Q,K,A
100 DATA 2,3,4,5,6,7,8,9,10,J,Q,K,A
110 DATA 2,3,4,5,6,7,8,9,10,J,Q,K,A
120 DATA 2,3,4,5,6,7,8,9,10,J,Q,K,A
130 END
```

This is how you type in the BASIC program as defined by its algorithm and flowchart.

The BASIC program, its flowchart, and its algorithm demonstrate several points:

1. The computer executes a BASIC program from the top-down, one instruction after another, until an instruction alters its sequence by sending it to another instruction. Then it will begin at that instruction and continue to execute the program in the same manner as before.
2. All BASIC instructions are numbered with a line number to tell the computer that one instruction follows another since its line number is greater. These line numbers do not have to be in increments of 10 as is shown in this example. However, that is standard procedure.
3. Each line has one BASIC statement on it, and that statement is always the first word on that line. The exception is a BASIC statement that is optional, that is, it is implied to be the first word on that line.

SUMMARY

There are over 80 statements, commands, and functions in the TI BASIC programming language, many of which are for advanced computer use.

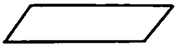
However, it is not the number of statements, commands or functions you know that makes or breaks you as a programmer. To say that you know all 80 of them, therefore you are a BASIC programmer, is like saying you know the 26 letters of the alphabet, therefore you are a writer. It is the logical relationship of the statements you use that describe your ability as a programmer.

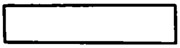
This relationship is often quite complicated. By flowcharting or writing the algorithm for programs before you type them in, you formally address the true nature of your program. In the long run, you save yourself much time and frustration by defining the problem first.

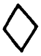
The fundamentals of program logic are important to you because you will use them as long as you write programs.

CHAPTER CHALLENGE

1. Name three fundamentals of program execution that match the following definitions:
   a. an instruction that causes the computer to execute one or more instructions over again.
   b. instructions that tell the computer that something has a value.
   c. the decision-making ability of the computer.
2. Name the three elements of all computer programs.
3. Match these three flowchart symbols with their correct names.

a.  The PROCESS symbol.

b.  The DECISION symbol.

c.  The INPUT/OUTPUT symbol.

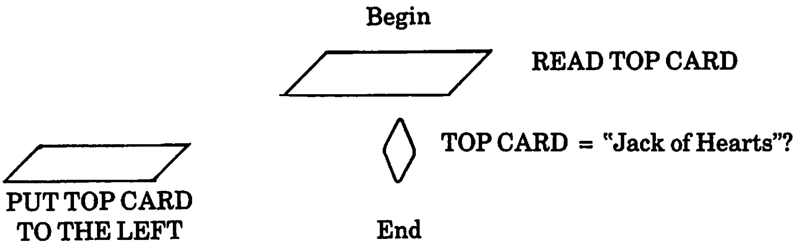
4 Every instruction in a BASIC program has at least two things. Name them.

5. Suppose you are going to write a new program for the computer robot that will instruct it to go through the same deck of cards until it finds the jack of hearts. After the robot finds this card, you want it to stop. Here is the algorithm.

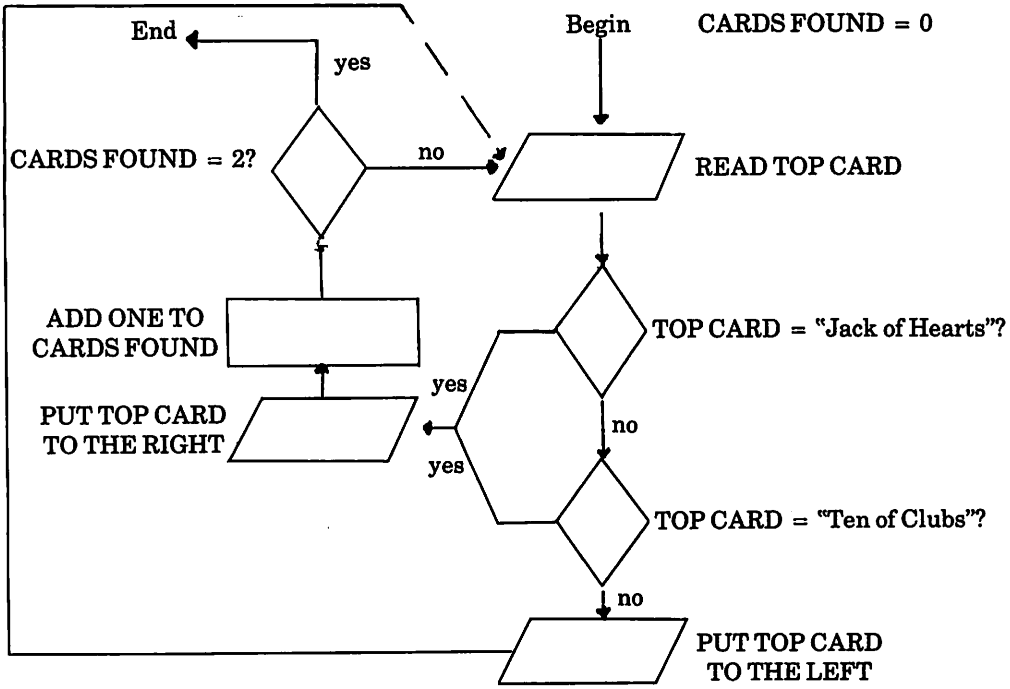
```
Instruction 1 – READ THE TOP CARD
Instruction 2 – IF THE TOP CARD IS THE JACK OF
                HEARTS THEN GO TO
                INSTRUCTION 5
Instruction 3 – THE TOP CARD TO THE LEFT
Instruction 4 – GO TO INSTRUCTION 1 AND
                CONTINUE FROM THERE
Instruction 5 – STOP
```

a. Which instruction(s) are ASSIGNMENT instruction(s)?
b. Which instruction(s) are CONDITION instruction(s)?

c. Which instruction(s) cause ITERATION?
d. Now fill in any necessary remarks and connect the symbols in the following flowchart of the algorithm.



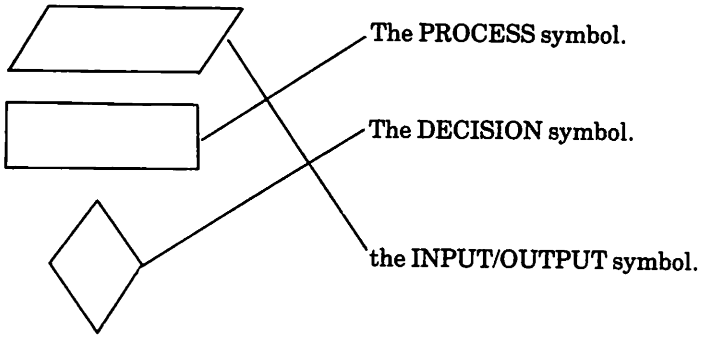
6. Suppose you want the robot to pick out both the jack of hearts and the ten of clubs, then stop. Here is the flowchart; now write the algorithm.



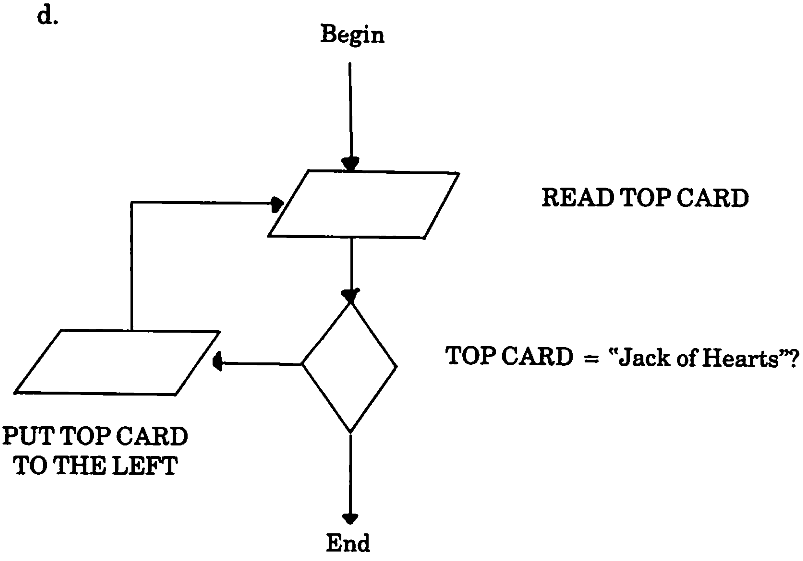
7. Why do you need to count the CARDS FOUND in this program?
8. What would happen if the deck of cards was missing the ten of clubs and the robot followed your instructions?

ANSWERS

1. a. ITERATION
   b. ASSIGNMENT
   c. CONDITION
2. INPUT(s)
   PROCESS(es)
   OUTPUT(s)
3. a.
   b.
   c.



4. A line number and a BASIC statement.
5. a. Instruction 1
   b. Instruction 2
   c. Instruction 4
   d.



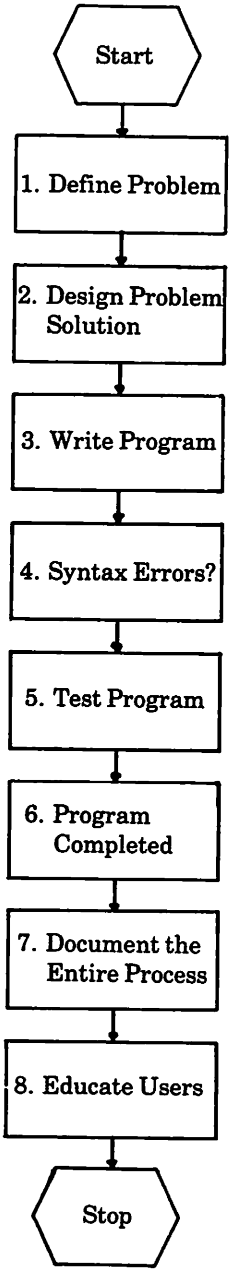
6.
   ```
   Instruction 1–CARDS FOUND EQUALS ZERO
   Instruction 2–READ THE TOP CARD
   Instruction 3–IF THE TOP CARD IS A JACK OF HEARTS
                 THEN GO TO INSTRUCTION 7 AND CON-
                 TINUE FROM THERE
   Instruction 4–IF THE TOP CARD IS A TEN OF CLUBS THEN
                 GO TO INSTRUCTION 7 AND CONTINUE
                 FROM THERE
   Instruction5–PUT THE TOP CARD TO THE LEFT
   Instruction 6–GO TO INSTRUCTION 2 AND CONTINUE
                 FROM THERE
   Instruction 7–PUT THE TOP CARD TO THE RIGHT
   Instruction 8–ADD ONE TO CARDS FOUND
   Instruction 9–IF CARDS FOUND EQUAL 2 THEN GO TO
                 INSTRUCTION 11
   Instruction 10–GO TO INSTRUCTION 2 AND CONTINUE
                  FROM THERE
   Instruction 11–STOP
   ```
7. It is necessary to stop the execution of the program after the robot has found both cards. You can do this by keeping track of the number of cards it has found, and sending it to the STOP instruction when that number equals 2.
8. The program will *blow up* on the robot when it trys to **READ** the 52nd card, as there won't be one there to **READ**. The robot may just have a nervous fit, the computer would give you an error message.

Developing Original Programs

Program development is a well-defined process. To minimize programming errors, follow the steps shown in Figure 4.4. Be as explicit as possible at each step. To remember this process use the acronym CONSTRUCT (see Figure 4.3).

Acronym	Stands for
C	Collect the facts
O	Organize the steps
N	Develop a Network of the flow
S	Eliminate Syntax errors
T	Test the program with data
Ru	Run applications of program
C	Compose a documentation file
T	Tutor others on program's use

Figure 4.3 How to construct original programs.



Recognize the problem. Determine outputs required. What inputs are necessary? Determine what calculations are needed (how can input documents be processed or transformed into required outputs?).

Divide problem into subunits. Write out sequence of activities. Use flowchart to visualize logic flow.

Code solution in BASIC language.

Enter in computer for debugging and editing.

Check logic of program using previously validated solutions.

rogram free of logic and syntax errors.

Collect results of steps 1-6. Write up the entire process. Deposit material in documentation manual.

Teach others how to use the program.

Figure 4.4 Steps in program development.

Step 1–Define the problem: You recognize the need for certain types of information about decisions you must make. What types of information do you need? List them. Try to sketch the form and content of the ideal output document from the program. What input documents would be necessary and where will they come from? Get copies of these source documents as part of the program's documentation.

Step 2–Design the Solution: Break the problem down to subtasks. Write a sentence or two about each task. Explain the task. Arrange the tasks in ascending order. You now have the beginnings of a *flowchart* or roadmap. A flowchart is a visual diagram of the activities and their sequence in a project. Flowcharts are excellent communication tools to explain and coordinate what has to be done. They document the program for you and other users. A good flowchart makes programming easier and reduces your error rate. Get into the habit of documenting your efforts through flowcharts. They are an integral part of program design.

Step 3–Write the program: With the flowchart complete, you are ready to code the solution in BASIC. Each activity in the flowchart suggests one or more BASIC statements. Figure 4.6 lists each flowcharting symbol, its meaning, and the BASIC instructions that perform that activity. How to write these programs in BASIC is the subject of the rest of Section II in this book.

Step 4–Syntax Error: If BASIC statements are improperly formed and don't conform to the rules of the language, a syntax error occurs. Misspelling a keyword like **REED** instead of **READ** is a syntax error, and the computer will display an error message when the statement is entered or when the program is **RUN**. Remove these errors by retyping the statement correctly and reentering it. Even if there is only one syntax error in the program, it will not run. If you don't understand the mistake, a listing of error messages is shown in Appendix V.

Step 5–Test Program: When the program is free of syntax errors, you must retest its logic. Is it looping and branching correctly? Input data for which you already know the results. Does the program replicate the known answers? Is the program's output reasonable? Try to test each branch or condition in the program. A payroll program might compute pay correctly for regular time but not when overtime occurs. You often hear of people who get large refunds or paychecks by mistake from a computer. Somewhere in the program an error has occurred; it can be embarrassing and expensive.

Step 6–Completed Program: The program is done and operational. It works. Now begin to use it for its intended purpose. Maybe improvements, or enhancements, will be made later.

Step 7–Document the Process: Collect the results of the previous six activities and deposit them in a folder or three-ring binder. This would include program listings, flowcharts, operating instructions, examples of outputs, source documents, file names and formats and any helpful hints about the program or the application. Now you have a user's manual for future reference.

Step 8–Educate Users: Teach others how to use the program. The user's manual will help. Having others benefit from your effort is helpful to them and rewarding to you. Perhaps you can market your program to others. Regardless, the CONSTRUCT process will make programming easier and faster for you.

Programming Errors

Every programmer makes mistakes. Making and correcting errors are excellent learning experiences. Programming is a skill and in any skill, you learn by doing. View correcting programming errors as a challenge to your mind or as a detective might view solving a complex case. The process should be fun and exciting, not tedious and frustrating. In debugging programs, we have spent up to eight hours searching for one error. But when we found it, it was exhilarating. Programming the TI comptuer will add excitement to your life.

Programming erros fall into three broad categories: syntax, execution, and logic.

Syntax errors–A BASIC statement is not formed according to the rules of the language–misspelled keyword, missing comma, undefined label, etc. Here the computer helps you find the error by printing error messages or diagnostics. These messages identify the cause and location (statement number) of the error. Simply retype the statement correctly and the error disappears.

Execution errors–After the computer translates each statement and understands it (no syntax errors), the program is executed. It is here that an execution error can occur. Examples are division by zero, a calculated value exceeds the computer's storage capability, the program runs out of data, etc. Error messages are also printed for execution errors so you can find and correct the errors.

Logic errors–Here the statement sequence or algorithm you have written does not correctly express the problem and thus the results are incorrect. Common examples might be an improper formula, branching to the wrong location, or printing the wrong variable. These are the most difficult to find, since no error messages occur. The CONSTRUCT process suggested earlier helps reduce logic errors. Step 5 in CONSTRUCT is a check for the presence of logic errors in your program.

With this background, you can begin to write programs for your TI-99/4A.

EXERCISES–CHAPTER 4

Definitions

1. What is a logic diagram?
2. What does BASIC stand for?
3. What advantages do computers have over humans?
3. What advantages do humans have over computers?
5. What are the characteristics of good computer applications?
6. List five of the most common types of applications for personal computers.
7. What does CONSTRUCT stand for?
8. What is a flowchart?
9. Name three types of programming errors.
10. Which of the three error types is hardest to find and why?

ACTIVITIES

11. Develop a flowchart showing how to bake a cake.
12. List ten applications where computers are being used today in our society.
13. List five applications where you could personally use a computer. Do prepackaged programs exist for the application? Find out.
14. Develop a flowchart for selecting a new job.
15. Think of another acronym for the eight steps in computer program development. What do each of the letters stand for?

Chapter 5

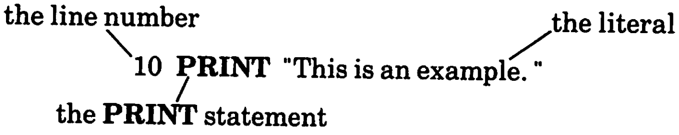
Writing and Editing BASIC Programs

The next step to writing in BASIC is to learn how to type in instructions and to understand how the computer recognizes them. Examples explain this process best, so here is your first BASIC statement: **PRINT**.

PRINT is called an *output statement.* It is an instruction that tells the computer to put something either on the monitor screen, on magnetic storage, or on the printer. The **PRINT** statement has many variations, but for the purpose of using it in these examples, you will cover only one of them.

PRINTING LITERALS

The **PRINT** statement may be used to output to the monitor screen what is called a *literal.* A literal is enclosed in quotes and follows the **PRINT** statement on a BASIC program line.



```
10 PRINT "This is an example."
```

Figure 5.1 A program line that contains a literal.

This short program tells the computer that its first and only instruction is to **PRINT**, on the screen, the words:

This is an example.

As it executes this short program, the computer will **PRINT** literally and precisely what you have typed between the quotes in the **PRINT** statement.

THE COMPUTER'S MESSAGE BOARD

As you type the BASIC program, the line you are typing appears on the monitor screen as it would on a piece of paper if you were using a typewriter. A small blinking marker called the *cursor* indicates where the next character will appear on the screen when you type it in. When you finish typing the line, press the ENTER key. This causes the cursor to jump to the beginning of the next line, just as a typewriter jumps to a new line when you press its carriage return key.

Although the computer's keyboard and monitor screen behave like a typewriter with paper in it, they are quite different. Think of the computer's monitor screen as a type of electronic message board. When you type a program line, you are writing a message on the message board for the computer. When you press the ENTER key, you send that message to the computer for it to *enter* into its current memory. Once that particular message is in the computer's memory, you can write other messages for it in the same way.

This message board works both ways. It lets the computer send messages to you. In the last example, for instance, you would type **RUN** then press ENTER to send the message the computer needs (the BASIC command) to begin executing your program. The computer would respond by sending a message (the program output) back to you by **PRINTing** it on its message board, the monitor screen.

```
>10 PRINT "This is an example."
>RUN
 This is an example.

 * * DONE * *

>
```

The > character at the far left of the screen is a prompt from the computer that says it is your turn to type in a message. When you see this prompt it means that you are at the **ROM** BASIC operating level and that the computer is waiting for you to type something, a BASIC program line or a BASIC command. When the computer sends a message, such as the output of a BASIC program or the computer's * * **DONE** * * message, it leaves off the >.

These are the messages you could expect to appear on the monitor screen as you type and **RUN** the following:

```
>10 PRINT "This is an example. "
>20 PRINT "We are going to "
>30 PRINT "PRINT literals "
>RUN
 This is an example.
 We are going to
 PRINT literals

* * DONE * *
```

CLEANING OFF THE MESSAGE BOARD

You can clean off your message board any time you want by typing:

```
>CALL CLEAR
```

Remember to press the ENTER key.

When you **CLEAR** the screen, it is important to note that the BASIC program is not cleared away. Only those messages that happen to be left on the message board are cleared away. Some of those messages were the lines of the BASIC program, but you have already sent these messages to the computer with the ENTER key. The computer has these program lines stored in its current memory.

LISTING THE PROGRAM

If you want the computer to put your BASIC program back on the screen, type **LIST** then press ENTER. This BASIC command causes the computer to put all of the lines in the BASIC program back on the screen.

```
>LIST
 10 PRINT "This is an example. "
 20 PRINT "We are going to "
 30 PRINT "PRINT literals "
>
```

You can also tell the computer to: **LIST** just one line of your program; **LIST** all of the lines from one line number through another line number; **LIST** all of the lines up to a particular line number; or, **LIST** all of the lines from a particular line number on. The following are examples of how we can use the **LIST** command.

Example	**Explanation**
>**LIST** 20	Lists line 20 only.
>**LIST** 50-100	Lists line 50 through 100 only.
>**LIST** -50	Lists lines up to 50.
>**LIST** 100-	Lists lines 100 on.

IMPORTANCE OF LINE NUMBERS

The computer will always **LIST** and **RUN** (execute) the BASIC program in order of that program's line numbers. It totally disregards the order in which you send the messages, it is only concerned with the order in which the lines numbers say the messages should be. Using the previous example, you can type in **LIST** and **RUN** in this way:

```
>30 PRINT "PRINT literals "
>20 PRINT "We are going to "
>10 PRINT "This is an example. "
>LIST
 10 PRINT "This is an example. "
 20 PRINT "We are going to "
 30 PRINT "PRINT literals "
>RUN
 This is an example.
 We are going to
 PRINT literals

* * DONE * *

>
```

INSERTING A PROGRAM LINE

To insert a program line into your BASIC program, type in and send the message (program line) to the computer using a line number that is between the number of the line which will preceed the new line and the number of the line which will follow it. For instance, if you wanted to insert a new **PRINT** statement between lines 20 and 30, you could type in a line with the line number 25. This is why program lines are usually numbered by ten. It gives you nine possible extra lines that you can insert between any two original lines. Here is an example of what messages would appear on the screen as you insert a new line into the sample program. Note: the lines with a > are the lines which you type.

```
>LIST
 10 PRINT "This is an example."
 20 PRINT "We are going to"
 30 PRINT "PRINT literals"
>25 PRINT "insert a line and"
>LIST
 10 PRINT "This is an example."
 20 PRINT "We are going to"
 25 PRINT "insert a line and"
 30 PRINT "PRINT literals"
>
```

Removing a line from a BASIC program is even simpler. Type the line number then press ENTER. The following is what you can expect to see on the monitor screen as you delete the line that you just inserted in the example program:

```
>LIST
 10 PRINT "This is an example."
 20 PRINT "We are going to"
 25 PRINT "insert a line and"
 30 PRINT "PRINT literals"
>25
>LIST
 10 PRINT "This is an example."
 20 PRINT "We are going to"
 30 PRINT "PRINT literals"
>
```

EDITING THE LINE YOU ARE TYPING

As you type a program line, there are several ways to correct any mistakes you may make before you ENTER that line. A special key, marked FCTN (FUNCTION), is used with various other keys on the TI-99/4A keyboard. This key lets you edit the line you are typing. The FCTN key is pressed and held, while a second key is struck, in order to perform the action you desire. For instance, pressing and holding FCTN, then tapping the S key will move the cursor to the left, back along the program line, one character at a time.

Most of the keys on the TI-99/4A keyboard will repeat if they are held down. This means if you hold the FCTN key down and the S key down, the cursor keeps moving to the left until you release the

S key. It also means that if you are a heavy-handed typist, you will see a lot of THISSSSSSSS on the screen from time to time.

The following describes the methods and keys you can use to edit BASIC program lines as you type them.

Retyping Characters

Use the cursor-left keys (FCTN S) to move the cursor backwards along the program line. This simply moves the cursor over the characters on that line. When the cursor is over a mistake or typo, you can retype that part and then either ENTER that line or move the cursor back to the end of the line with the cursor-right keys (FCTN D) and continue typing the line.

Inserting Characters

FCTN and the number key marked 2 cause whatever you type to be INSerted at the current position of the cursor. This means you can move the cursor back in a line (FCTN S), press the FCTN 2 keys, and insert into, rather than retype over, that part of the program line. FCTN 2 turns on an insert mode that stays on until you finish typing. When you move the cursor with a FCTN key or press ENTER, the insert mode turns off.

Deleting Characters

Next to INS (FCTN 2) is DEL (FCTN 1). This is the key that, when struck or held with the FCTN key, deletes characters from your program line. If you move the cursor back and delete a character, the rest of the characters to the right of the cursor shift one space to the left. If you hold the delete key down, the cursor looks like it is gobbling up the characters in the rest of that line.

Erasing the Entire Program Line

FCTN and the number key marked 3 will erase the program line you are currently typing. This has the effect of removing that message from the message board, which lets you start typing that entire message (including line number) over again.

After you have typed and edited the BASIC program line, press the ENTER key to send that line to the computer for it to put in its memory. The cursor can be anywhere on that line when you ENTER it. The computer will know that you are sending it that entire line.

EDITING A LINE AFTER IT HAS BEEN ENTERED

One way to change a BASIC program line after it has been entered is to type that line over again. Any time you send the computer a program line with the same number as a line it has already received, it replaces the old line with the new one. In other words, when you send a message with the same line number as one you have already sent, the computer assumes the first message was wrong.

Many times there are only one or two incorrect characters in a line. There is a fast, easy way to correct these mistakes in one or more of your program lines.

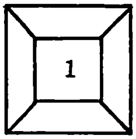
The Edit Command

Remember the screen is an electronic message board and you can cause the computer to put those messages back on the screen by typing **LIST**. Another command, **EDIT** causes the computer to do something similar. When you type **EDIT** [line number], the computer will put that line back on the screen, with the cursor at the first position of the first BASIC statement. From there you can edit that line in the same manner that you edit a line before you have ENTERed it. That is, you can retype, insert, or delete with the FCTN keys to make corrections and press the ENTER key to send that corrected program line (message) to the computer. When you press the ENTER key, the computer leaves the **EDIT** mode and you can continue writing new program lines in the normal manner.

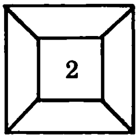
You can also **EDIT** several program lines, one after another, by using the FCTN keys: FCTN X (cursor-down) and FCTN E (cursor-up), instead of the ENTER key, when you are through editing a program line. These two FCTN keys will ENTER the corrected program line then move to the next line in the program without causing the computer to leave the **EDIT** mode. The next line can be either the line before, FCTN E (cursor-up), or the line after, FCTN X (cursor-down) the line you are on.

EDITING KEYS SUMMARY

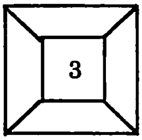
(Del)



(Ins)



(Erase)



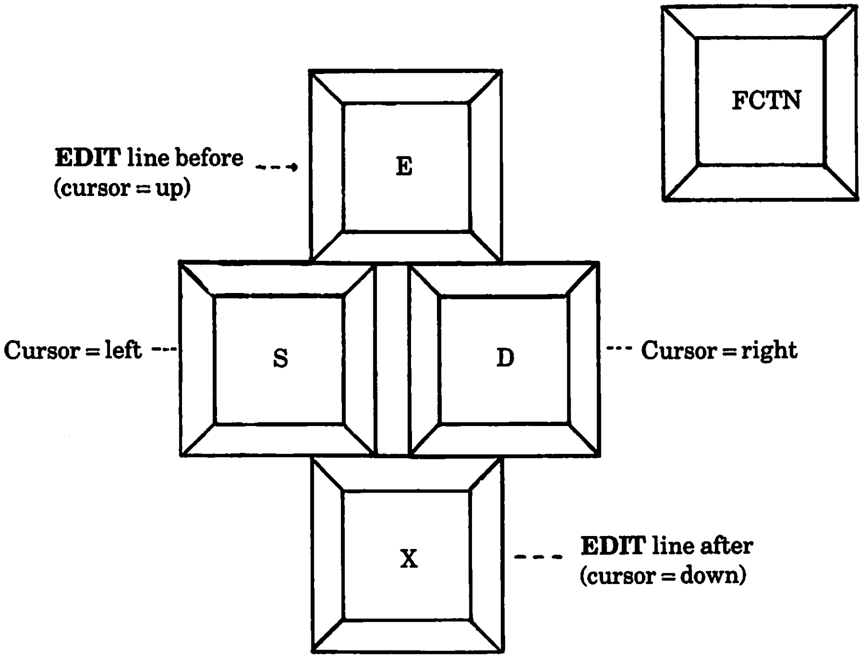

FCTN
EDIT line before (cursor = up)
E
Cursor = left
S
D
Cursor = right
X
EDIT line after (cursor = down)

Figure 5.2 A summary of the editing keys.

FCTN S	Moves cursor to the left on current line.
FCTN D	Moves cursor to the right on current line.
FCTN 1	Deletes character at current cursor position.
FCTN 2	Inserts character at current cursor position.
FCTN 3	Erases current program line.
FCTN E	Displays line before current line for **EDIT**ing.
FCTN X	Displays line after current line for **EDIT**ing.

When typing a program line you can make a TI BASIC program line four times longer than the length of the screen (112 characters vs. 28 characters). When you type in over 28 characters, the program line wraps around and continues one line down on the other side of the screen. The computer still considers this to be one program line. To avoid confusion, a BASIC program line is referred to as a logical line.

OTHER IMPORTANT PROGRAM LINE COMMANDS

Number

The computer can automatically generate line numbers while you type in the BASIC programs. To do this, type **NUM** and press ENTER. The computer then starts by setting up line 100 and will set up a new line (increased by 10), each time you ENTER a finished BASIC program line. Typing **NUM** then pressing ENTER puts the computer in the **NUMber** mode and pressing the ENTER key without typing anything on a line just generated causes the computer to leave the **NUMber** mode.

Resequence

You can **RESequence** line numbers by typing **RES** then pressing ENTER. This causes the computer to renumber the BASIC program lines. **RES** is particularly useful when you want to insert a line but can't because the two line numbers on either side of where you want to insert the new line are too close together, such as line 10 and line 11. **RESequencing** will result in the program beginning with the line number 100. Each one after that will have a line number increased by 10.

SUMMARY

What is the difference between a BASIC command and a BASIC statement? Generally, an instruction is a command when you ENTER it without a line number. **RUN**, **LIST**, and **EDIT** are examples of what are usually referred to as commands. Statements are used primarily in the BASIC programs and so have line numbers preceeding them.

Lack of distinction comes because instructions, usually called commands, are used in programs as statements, and instructions, usually called statements, are used without line numbers as BASIC commands. The cause of the confusion is the strong interrelationship between the BASIC program operating level and the BASIC operating level. It is more important to remember that these are two different operating levels than it is to worry about what is a command and what is a statement.

The following BASIC commands help jog the programmer's current memory when it needs it. They include a few commands listed that have not been discussed thus far but include brief explanations that you should understand fairly well.

BASIC COMMANDS SUMMARY

CALL CLEAR clears the screen of all messages.
EDIT 60 displays line 60 for editing.

EDIT mode keys

FCTN S moves cursor to the left on line 60.
FCTN D moves cursor to the right on line 60.
FCTN 1 (del)etes character at current cursor position.
FCTN 2 (ins)erts character at current cursor position.
FCTN 3 (erase)s line 60 from the screen (not memory).
FCTN E enters line 60 and displays line 50 for **EDITing**.
FCTN X enters line 60 and displays line 70 for **EDITing**.

LIST lists all lines of the BASIC program in current memory on the monitor screen.
  LIST 40 lists line 40.
  LIST 40-100 lists lines 40 through 100.
  LIST 100- lists lines 100 on.
  LIST -100 lists line up to 100.
NEW erases all BASIC program lines from the current memory.
  BYE erases all BASIC program lines from the current memory and returns to title screen.
NUM starts automatic line numbering with line 100 and by increments of 10.
  NUM 50 starts wtih line 50 and by increments of 10.
  NUM 200,50 starts with line 200 and by increments of 50.
  NUM ,5 starts with line 100 and by increments of 5.
RES renumbers all of the program lines, starts with the number 100 and increments by 10.
  RES 50 starts with line 50 and by increments of 10.
  RES 200,50 starts with line 200 and by increments of 50.
  RES ,5 starts with line 100 and by increments of 5.
RUN begins execution of the BASIC program in current memory.
  RUN 50 starts program execution at line 50.
  FCTN 4 (**CLEAR**) stops program execution. Exits **EDIT** and **NUM** modes.

CHAPTER CHALLENGES

The first part of this Chapter Challenage is an exercise that uses many of the BASIC commands covered in this chapter. The idea here, is to gain keyboard familiarity and a general feel for the TI-99/4A's behavior when it is given a command.

There will probably be a number of error messages the computer will send you while you do this exercise, but that is part of what you are learning: how nit-picky this machine can be.

Feel free to experiment; try different commands. If the computer complains, type in that line or command again. Remember, there is no way to hurt the computer by typing on the keyboard.

1. Turn on the computer and follow the screen instructions until the TI BASIC READY message appears. At the bottom of the screen there should be a > □
2. Now type **NUM** and press ENTER. The computer generates line numbers for the BASIC program. This is what you should see at the bottom of the screen:

```
>100 □
```

3. Type in the following BASIC program. Remember that the computer will leave the **NUMber** mode if you press the ENTER key just after a line number has been generated.

```
>100 CALL CLEAR
>110 PRINT "This program will "
>120 PRINT "instruct the computer "
>130 PRINT "to PRINT out this "
>140 PRINT "message. "
>150 [ENTER]
>
```

4. Be sure the computer is no longer in the **NUM** mode then type **RUN** and press ENTER.
5. Now type **LIST** then press ENTER.
   A. Try **LISTing** just one line of this program.
   B. Try **LISTing** lines 120 through 140 only.
   Type **CALL CLEAR** ENTER to clear the screen then **LIST** the whole program again.
6. Type in the new program lines:

```
>125 PRINT "to first clear the screen"
>126 PRINT "and then PRINT out this"
```

LIST the program again. This is what it should look like:

```
100 CALL CLEAR
110 PRINT "This program will"
```

```
120 PRINT "instruct the computer "
125 PRINT "to first clear the screen "
126 PRINT "and then PRINT out this "
130 PRINT "to PRINT out this "
140 PRINT "message. "
```

7. Now remove line 130 by simply typing 130 and pressing ENTER. **LIST** the program again. **RUN** the program. This is what the output should look like:

```
This program will
instruct the computer
to first clear the screen
and then PRINT out this
message.
```

8. Type **RES** then press ENTER and **LIST** the program again. Notice that the line numbers have changed.
   A. Try typing **RES** 10,10 and **LISTing** the program.
   B. Try typing **RES** 500,100 and **LISTing** the program.
   RES, CALL CLEAR the screen and **LIST** the program. This is what it should look like:

```
100 CALL CLEAR
110 PRINT "This program will "
120 PRINT "instruct the computer "
130 PRINT "to first clear the screen "
140 PRINT "and then PRINT out this "
150 PRINT "message. "
```

9. Now type **EDIT** 110 then press ENTER. Move the cursor to the right with the FCTN D keys until it is over the *p* in *program*. Press the FCTN 2 keys to turn on the INSert mode and type the word BASIC (add a space to separate the two words). The line 110 should look like this:

```
110 PRINT "This BASIC program will "
```

10. Instead of pressing the ENTER key, press the FCTN X keys so that line 110 will be entered without the computer leaving the **EDIT** mode. This should bring line 120 to the screen for you to **EDIT**. Move the cursor to the right until it is over the *t* in *the* and type in *my TI Computer* so that line 120 looks like this:

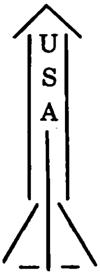
```
120 PRINT "instruct my TI Computer "
```

11. Press the FCTN X keys again so that you can **EDIT** line 130. Move the cursor over the *f* in *first* and use the FCTN 1 keys to DELete this word from this line. Now press the ENTER key. This causes the computer to leave the **EDIT** mode and you can **RUN** the **EDITed** program. This is what the output should look like:

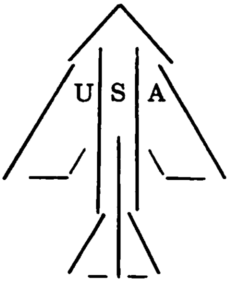
```
This BASIC program will
instruct my TI Computer
to clear the screen
and then PRINT out this
message.
```

Here are a few pictures that you can *draw* using only **PRINT** statements and keyboard characters in the BASIC program. Try writing the programs that, when executed, will **PRINT** these pictures on the screen with their titles.

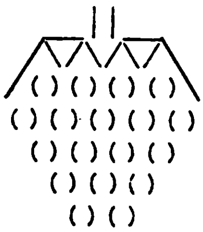
1) ROCKET SHIP



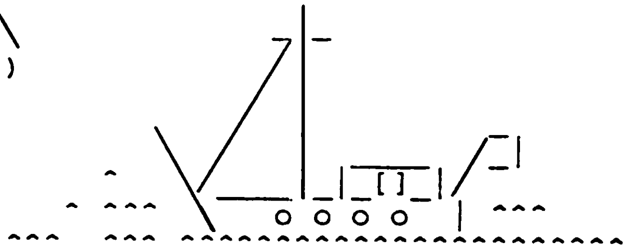
2) FIGHTER PILOT



3) STRAWBERRY



4) SHIP AT SEA



Chapter 6

TI BASIC Calculations

At times you will want the computer to perform mathematical calculations while it executes BASIC programs. The TI-99/4A is well equipped to solve problems that involve buying and selling goods, figuring percentages, analyzing finances, etc.

You may think that computers are useful only if you know algebra, calculus, or other forms of advanced mathematics. But the computer's usefulness lies in its ability to perform repetitious calculations at a very fast rate without fatigue or errors. The *volume*, not the complexity of calculating work makes computers valuable tools.

For most of us, *volumes of work* means adding up rows and rows of figures, dividing several numbers several times to find percentages, or repeatedly performing fairly simple calculations on many different sets of numbers. To perform these tasks on the computer, you need to be able to instruct the TI-99/4A to do computations.

COMPUTATIONS

There are five basic operations in mathematics: addition, subtraction, multiplication, division, and exponentiation (powers of numbers). When you do these operations by hand, you can say the same thing in several ways. For instance, 3 × 2, 3(2), and 3*2, are all different expressions for 3 times 2.

But the computer understands only one expression for each operation. Figure 6.1 is a list of the five operations and their computer formats.

3+2	three plus two
3−2	three minus two
3*2	three times two
3/2	three divided by two
3∧2	three to the second power (three squared)

V	Addition
/	Subtraction
*	Multiplication
−	Division
+	Exponentiation

Figure 6.1 Operations and their formats.

More than One Operation at a Time

In addition to knowing the format of mathematical operations, you also need to know the order of execution the computer will follow to solve a problem that includes more than one operation.

Take the simple equation, 3+5*2, for example. Three plus five is eight, and eight times two is 16. On the other hand, five times two equals ten, and if you add three to ten, the answer is 13.

So, the equation 3+5*2 could have two answers.

3+5*2=?	3+5*2=?
8 *2=16	3+ 10 =13

Is the correct answer 16 or 13? Without knowing the order in which you should execute the problem, you have no way of knowing the correct answer.

The computer handles this type of problem with a set of rules given to it by the TI BASIC language program. But all programming languages have similar instructions for the computer to follow. They are called "rules of priority", and they tell the computer to execute certain operations before others.

The following table describes which operations the computer has been instructed to execute, in order of their priority.

EXECUTION PRIORITY

1st Priority	Any operation or set of operations in parentheses ( )
2nd Priority	Exponentials ∧
3rd Priority	Multiply, Divide *, /
4th Priority	Add, Subtract +, −

Figure 6.2 Execution priority.

The computer's approach to solving this example problem, 3+5*2, would be to first perform the multiplication, 5*2=10, then perform the addition, 3+10=13. This is because multiplication has a higher execution priority than addition.

Suppose you are actually trying to calculate the total pieces of bread necessary to make three ham and five tuna fish sandwiches. If this were the case, you would first want to add three and five to get the total number of sandwiches, then multiply that times two (no triple deckers in this problem) to get the correct number of pieces of bread–16. To get the computer to come up with the correct answer, you have to modify the way you write the equation.

PARENTHESES

You can make the computer alter its order of execution by placing operations inside parentheses or sets of parentheses. The operations inside of the parentheses will then have the highest priority, that is, those operations will be executed first.

(3+5)*2

Typing in your equation as shown above instructs the computer to first add three and five (ham and tuna fish sandwiches) then multiply that by two (the pieces of bread per sandwich).

You may also use parentheses inside parentheses to control the computer's order of executing the problems. When you do this, the operations of the inside parentheses have a higher priority then the operations of the outside parentheses.

For example, suppose you take a vacation and plan to make those ham and tuna sandwiches for lunches each day. The vacation will last five days but you are going to have lunch at a restaurant on two of those days. You already know how to get the computer to compute the pieces of bread you will need each day:

(3+5)*2

and now you want to type in the equation for calculating how much bread to buy for the vacation.

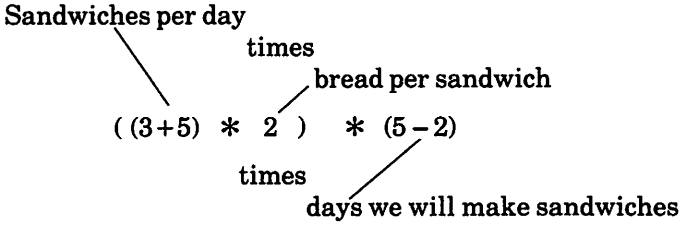


Figure 6.3 Sandwiches.

When the computer reaches this equation, it goes straight to the innermost parentheses (3+5) and performs that operation first. This will leave it with:

(8*2)*(5−2)

Then the computer will perform the two operations, 8*2 and 5−2, since these are both still inside of parentheses. The computer treats these two operations with equal priority. You don't know which one it will do first. You only know that they will both be done before anything else. This will leave the computer with:

16*3

This is the last operation the computer has to perform, and it will compute the final answer to your problem as 48, the number of pieces of bread you will need for your vacation lunches.

When there is more than one operation inside a single set of parentheses, the computer follows its standard order of execution to perform those operations until all of the operations inside that parentheses are performed. Also, parentheses are used throughout BASIC programs in several statements, commands, and functions. Generally, they are the way to tell the computer to "Do this first" or, as in the case of BASIC functions, parentheses are often used to tell the computer to "Do this only."

To summarize the discussion of execution priorities, you have an example of a more complicated equation, one that has several operations in it, and which shows the order of execution the computer will follow as it computes your answer.

Example

Equation	Computer Format	Answer
$3+\frac{7-5}{2^2}$	3+(7−5)/2∧2	3.5
parentheses 1st	(7−5)=2	leaves 3+2/2∧2
exponent 2nd	2∧2=4	leaves 3+2/4
division 3rd	2/4=.5	leaves 3+.5
addition 4th	3+.5=3.5	leaves 3.5

Figure 6.4 Example.

NUMERIC FUNCTIONS

The best way to describe functions is to say they are built-in tricks that the computer can do. Numeric functions are computations done with numbers for which it would otherwise require considerable time to write BASIC instructions.

Functions usually start with a three-letter word that stands for the type of trick that function does. The three-letter word is usually followed by a set of parentheses with something inside them. The function will perform its computation on the numeric value within the parentheses.

For instance, the function **SQR** which stands for *square root function* will find the square root of the numeric expression in its parentheses. To find the square root of 16 type:

SQR(16)

and the computer will find, or compute, the square root of 16 when it reaches this part of the BASIC program.

The number 16 is a *numeric expression.* A numeric expression can be a number or an equation. This means you can type:

SQR(4*5 – 4)

In this case, the computer first solves the equation inside the parentheses. Then it will find the square root of the answer to the equation, the square root of 16 in this example.

Whatever you put inside the function's parentheses (the first or outside set of parentheses) is called the *argument* of the function.

If you remember all of regular mathematics, you might remember that a negative number does not have a square root since a negative number times itself is a positive number. If you type in:

SQR(–16)

the computer gives you the error message, BAD ARGUMENT. This does not mean you failed to convince it to find the square root of negative 16. It means that it can't do that particular trick (**SQR**) with the information you have typed inside of the function's parentheses – its argument.

There are twelve numeric functions built-in to the TI-99/4A, but most of these are for advanced graphics, engineering, or scientific applications. Two of them, however, are useful in a number of situations.

INT (NUMERIC EXPRESSION) – INTEGER FUNCTION

An integer is a whole number. There is no fractional part attached to it, such as 2.3766. An integer is a whole number such as 2, 5, 199, 0, –3, etc.

The **INTeger** function turns its numeric expression, or argument, into a whole number by *truncating* (computer language for chopping off) anything to the right of the decimal point. This means that 2.3766 gets changed to 2 and so does 2.9999999. In other words, **INT** does not round off the number, it simply chops off the fractional part.

This is such a handy function because it gives you control over what the numbers will look like when you go to read answers, or output, on the monitor screen. To explain how you can do this, you first need to look at what the TI-99/4A comes up with when it computes an answer.

If you type 1+3 and have the computer print the answer on the monitor, it prints 4 with no problem. But, if you have the computer print the answer to 1/3 (1 divided by 3), it prints .3333333333.

Now if you were writing a program that was splitting a dollar three ways and wanted output so that it could be read as cents instead of .3333333333, you would type:

```
INT( (1/3)*100)
```

When the computer reaches this part of the program, it first goes to the innermost parentheses, (1/3), and computes the answer as .3333333333. It will then multiply that by 100 to get 33.33333333. Then it will convert that to an **INTeger** and the final answer will be 33, the number of whole cents in one-third of a dollar.

Another example would be to say you have $10.00 to split three ways and you want output in dollars and cents display (3.33) instead of 3.3333333333. To do this type:

```
INT( (10/3)*100)/100
```

Here the computer will:

First	Divided 10 by 3 to get 3.3333333333
Second	Multiply 3.3333333333 by 100 to get 333.3333333
Third	Convert that to an **INTeger** to get 333
Fourth	Divide 333 by 100 to get 3.33

Notice what this last example tells us about numeric functions. They have an execution priority just as the operations have. In fact their priority is higher than all operations unless those operations are in parentheses.

In both of these examples you control your output by first moving the decimal place in the number, converting it to an **INTeger**, and then moving the decimal place back if you wish.

Rounding with the INT Function

If you want to round off the answer, add .5 to the numeric expression after you have moved the decimal to the right as far as you want to round.

For example, take the two numbers 1.234 and 1.236 and follow how the computer rounds them off with the **INTeger** function.

```
INT(1.234*100+.5)/100    INT(1.236*100+.5)/100
INT(123.4+.5)/100        INT(123.6+.5)/100
INT(123.9)/100           INT(124.1)/100
123/100                  124/100
1.23                     1.24
```

RND–Random Number Function

A random number is meant to be a surprise. You don't know what number the computer is going to come up with when you use this function. **RND** does not use an argument to come up with a random number. You simply type in **RND** and the computer will generate a number for you.

Random numbers are useful when you want to write a program that is unpredictable even by you. Computer Aided Instruction (CAI) programs often use random numbers to give unpredictable problems. Computer games are another major area where random numbers may be desirable and finally, random numbers are often used to test programs or simulate different situations to see if a complicated equation works under a variety of conditions.

There are several ways that the computer may generate random numbers. To explain them, you will need a new BASIC statement and a simplified explanation of how the computer comes up with its random numbers.

Imagine that the computer has a long list of random numbers to use. If you simply tell the computer to **RND**, it starts at the top of this list and sends the first number on it. The problem is that the computer will send the same number every time you **RUN** your program, that is, it starts at the top of the list again.

If you want the computer to start at a different spot in this list of random numbers, you need to use the BASIC statement **RAN-**

DOMIZE in the beginning of the same program. This statement may then be followed by a number that tells the computer to start at a spot somewhere other than the top of the list:

RANDOMIZE 121

The number you type in is called a *random number seed,* and this number is what determines where the computer will start on the list of random numbers. Even though the spot will be different, the computer will still start at that same spot on the list, every time you **RUN** your program. That is unless you change your random number seed in the **RANDOMIZE** statement.

If you want the computer to start on a totally unpredictable spot on its list, each time you **RUN** the program type in the **RANDOMIZE** statement without the random number seed. That is, simply type **RANDOMIZE.**

When the computer generates a random number, that number is a 10-digit number greater than or equal to zero and less than one. A typical random number might look like this:

.5209795429

After a random number has been generated it is up to you to turn it into the type of number you want. This is another time that the **INT** function comes in handy. If you want to generate a random number between one and 100 type:

INT(RND * 100)+1

The computer will execute this numeric expression in the following manner:

```
INT(.5209795429*100)+1
INT(52.09795429)+1
52+1
53
```

MORE ON THE PRINT STATEMENT

One of the things that a **PRINT** statement can do is **PRINT**, on the monitor screen, the answer to a numeric expression. All you have to do is write the BASIC line as:

10 **PRINT** [numeric expression]

For example, you could type in and **RUN** a short program that uses some of the examples covered in this chapter.

```
>10 PRINT 3+5
>20 PRINT 1/3
>30 PRINT 3+(7-5)/2^2
>40 PRINT SQR(4*5-4)
>50 PRINT INT(RND*100)+1
>RUN
  8
  .3333333333
  3.5
  4
  53

* * DONE * *
```

Now you can combine the literals discussed earlier with numeric expressions in the **PRINT** statements to add meaning to the answers the computer comes up with. Some examples might be:

```
10 PRINT "4 divided by 5 is ";4/5
RUN
4 divided by 5 is .8

* * DONE * *

10 PRINT "The answer is $ ";INT(1/3*100)/100
RUN
The answer is $ 3.33

* * DONE * *

10 PRINT "Put ";30/5; " cows in each barn "
RUN
Put 6 cows in each barn

* * DONE * *
```

Notice that since you have the computer **PRINT** more than one thing (literals and numbers in these examples), you insert a semicolon(;) between each thing on the **PRINT** line. The semicolon used this way is called a *PRINT separator* which is necessary to tell the computer that one thing is separate from the next.

PRINT SEPARATORS

You can use three **PRINT** separators: semicolons (;), colons (:), and commas (,). Each of these separators tells the computer that one thing on the **PRINT** program line is separate from the next thing, but each of them also carries its own special message for the computer.

The semicolon (;) tells the computer to stay on the *same line* for the next thing it is going to **PRINT**.

The colon (:) tells the computer to move to the beginning of the *next line* for the next thing it is going to **PRINT**.

The comma (,) tells the computer to move to the *next print zone* for the next thing it is going to **PRINT**.

There are two **PRINT** zones on the monitor screen, one on the left- and one on the right-hand side of each line. Each line on the screen is 28 character spaces wide, and the right-hand zone begins at the fifteenth space. When you use a comma as a **PRINT** separator the next thing is printed in the next zone which will either be on the right-hand side of the same line, or on the left-hand side of the next line down.

The following examples show how you can use **PRINT** separators to control your output.

Examples: PRINT separators

```
        ";" SAME LINE

>10 PRINT "This is ";
>20 PRINT " an example "
>RUN
 This is an example

* * DONE * *

>
```

```
     "," NEXT PRINT ZONE

>10 PRINT "This ","is
     an ","example "
>RUN
 This          is an
 example

* * DONE * *

>
```

": " NEXT LINE

```
>10 PRINT "This ":"is
     an ":"example "
>RUN
This
is an
example

* * DONE * *

>
```

COMBINATIONS

```
>10 PRINT "This ","is an "::,
>20 PRINT "example "
>RUN
This          is an

              example

* * DONE * *

>
```

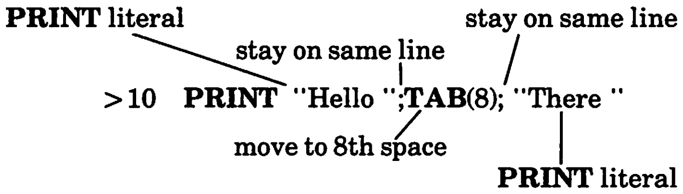
Figure 6.5 Examples: Print separators.

Notice that several commas or colons may be used at a time to send a separator's message more than once, and that the message the separator sends can be at the end of a **PRINT** program line. The computer will then remember that message until it is told to **PRINT** again.

TAB [NUMERIC EXPRESSION] FUNCTION

Another way to control output is with the **TAB** function. This function is used in a **PRINT** program line to tell the computer to move forward to a certain character space before **PRINTing** the next thing.

As mentioned, the monitor screen is 28 character spaces wide. If you want the computer to move forward to the eighth space before **PRINTing** something else, you can type **TAB**(8). The following example shows how you type this function into your **PRINT** program lines.



```
>10 PRINT "Hello ";TAB(8); "There "
```

Figure 6.6 A PRINT program line.

The **TAB** function is very useful for lining up columns of output. You can use several **TABs** in your **PRINT** line to set up these columns.

```
>10 PRINT TAB(5); "Mon ";TAB(10);40/5;TAB(15);
"Hours "
>20 PRINT TAB(5); "Tues ";TAB(10);30/5;TAB(15);
"Hours "
>30 PRINT TAB(5); "Wed ";TAB(10);25/5;TAB(15);
"Hours "
>40 PRINT TAB(5); "Thurs ";TAB(10);20/5;TAB(15);
"Hours "
>50 PRINT TAB(5); "Fri ";TAB(10);10/5;TAB(15);
"Hours "
>RUN
        Mon     8    Hours
        Tues    6    Hours
        Wed     5    Hours
        Thurs   4    Hours
        Fri     2    Hours
* * DONE * *
```

You should notice that you cannot **TAB** to a character space that the computer has already passed on that line. In other words, you cannot **TAB** backwards. You do not get an error message, but the computer will move to the next line down to **PRINT** the rest of what is in that **PRINT** program line.

SUMMARY

Although the TI-99/4A can be used as a calculator, it has much more potential. Once you learn how to type in equations to get the right answers, you can write programs to output the answers in a report form, easily read by anyone.

You have now started to build a vocabulary of computer instructions. Like building blocks, these instructions are individual parts of BASIC programs that can be put together in a variety of ways to do the same thing. For instance, all of the following program lines instruct the computer to **PRINT** the word "Hello" in the same place on the screen.

```
10 PRINT "                    Hello "
10 PRINT , "Hello "
10 PRINT TAB(15); "Hello "
```

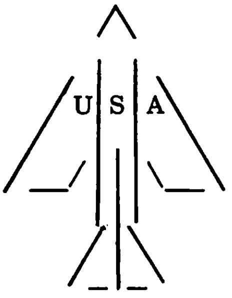
Given an equation such as;

$$\frac{16+4}{10}\times 3$$

you may find that there is more than one correct computer format for it.

```
((16+4)/10)*3
3*((16+4)/10)
```

Also like building blocks, these instructions provide you with the opportunity to be original and creative in writing your programs. For instance, in the last Chapter Challenge we had the computer **PRINT** a rocket ship made out of keyboard characters on the screen. The finished program should have looked something like this:



```
>100 PRINT "                              "
>110 PRINT "                              "
>120 PRINT "                              "
>130 PRINT "                              "
>140 PRINT "                              "
>150 PRINT "                              "
>160 PRINT "                              "
>170 PRINT "                              "
>180 PRINT "                              "
>190 PRINT "                              "
```

Figure 6.7 A program to print a rocket.

Now if you add a line

```
>200  PRINT ::::::::::::::::::::::::::::::::
```

and **RUN** this program, the rocket ship appears to be taking off as the computer starts to execute all the "move to the next line" colons in the last **PRINT** line. Actually, the screen is *scrolling* by.

The fact that the screen scrolls has not been mentioned before because it is one of many things that should become obvious as you learn to program. As you continue to learn new instructions, it should be pointed out that it would be impossible to itemize every combination into which those instructions can be put. Therefore, it becomes increasingly important that you experiment, try different combinations of instructions, and observe how the computer behaves when it reaches those program lines. This is the challenge and fun of programming and is fundamental to learning how to build programs out of the available building blocks.

CHAPTER CHALLENGE

Write the following as numeric expressions:

1. 2 times 6
2. 1 plus 5
3. 8 minus 3
4. 4 divided by 2
5. 3 squared

What are the computer's answers to the following numeric expressions?

6. 3+6/3
7. 4*2+1
8. 2∧2-6/2
9. (8+2)/5
10. 9/3+4/2
11. ((4/2+3*2)+4)/3
12. **SQR**(6-2)
13. **INT**((1/4)*100)
14. **INT**(**RND***10)+1

The following exercises contain a wide range of difficulty. For readers who have not studied math for some time and are not sure how the equation should be worked, the answers to them may be found on the far right-hand side of the page. This still does not tell how to work the equation but if you work it one way, and you get the answer shown, the odds are that you worked it correctly.

Convert these equations to their computer format then type them in as a BASIC program that **PRINTs** "The answer is" [numeric expression].

EXAMPLE

$7+\frac{2\times 3}{2}$	7+(2*3)/3	10

```
10 PRINT "The answer is ";7+(2*3)/2
```

EQUATION	COMPUTER FORMAT	ANSWER
15. $\frac{3+7}{2}$		5
16. $(2\times 5)+3$		13
17. $\frac{16+4}{10}\times 3$		6

18. $\frac{15\times3}{6+3}+7$	12
19. $\frac{(2\times12)+6}{2}-5$	10
20. $\left(\frac{3+5}{10-6}\right)^2$	4
21. $300+\frac{\frac{300\times.05}{5}}{(1+.05)^{12}}$	168.33
22. $5\times\sqrt{\frac{16\times4}{(3+5)\,(1/2)}}$	20

In these problems, combine literals with numeric expressions and **PRINT** separators with **TAB** functions to create output that is clear and easy to read.

23. Write a BASIC program that will output the number of hours, minutes, and seconds in a year.
24. You can convert degrees Fahrenheit to degrees Centigrade by subtracting 32 degrees from the Fahrenheit temperature, then multiplying that number by $\frac{5}{9}$ths. Write a BASIC program that will output a conversion of 98 degrees Fahrenheit to degrees Centigrade.
25. Write a BASIC program that converts 55 miles per hour to the number of feet per second one is traveling at that speed. There are 5280 feet in a mile and 60 seconds in a minute.

ANSWERS

1. 2*6
2. 1+5
3. 8−3
4. 4/2
5. 3∧2
6. 5
7. 9
8. 1
9. 2
10. 5
11. 4

12. 2
13. 25
14. A surprise number between 1 and 10
15. (3+7)/2
16. 2*5+3
17. ((16+4)/10)*3
18. (15*3)/(6+3)+7
19. (2*12+6)/2 – 5
20. ((3+5)/(10 – 6))2∧2
21. (300+((300*.05)/5))/(1+.05)∧12
22. 5*((16*4)/(3+5)*(1/2))∧.5

```
23. 10 CALL CLEAR
    20 PRINT TAB(7); "HOW TIME FLIES "::
    30 PRINT " ============================ "
    40 PRINT "Hours in a year =  ";TAB(20);365*24
    50 PRINT "Minutes in a year =  ";TAB(20);365*24*60
    60 PRINT "Seconds in a year =  ";TAB(20);365*24*60*60
    70 PRINT " ============================ "::::
24. 10 CALL CLEAR
    20 PRINT TAB(6); "ITS NOT THAT HOT "::
    30 PRINT "**************************** "::
    40 PRINT 98;TAB(6); "Degrees Fahrenheit "
    50 PRINT TAB(13); "is only "::
    60 PRINT INT( (85 – 32)*(5/9)*100)/100; "Degrees
       Centigrade "::
    70 PRINT "**************************** "::::
25. 10 CALL CLEAR
    20 PRINT TAB(3); "LIFE IN THE  "; " " " "; "FAST ";
       " " " "; " LANE "::
    30 PRINT "<<<<<<<<<<<<<<
       >>>>>>>>>>>>>> "::
    40 PRINT TAB(2);55; "MPH may seem slow but "::
    50 PRINT TAB(5);INT( ( 55*5280)/60)/60);
       "feet per second "::
    60 PRINT " is as fast as we should go "::
    70 PRINT ">>>>>>>>>>>>>>
       <<<<<<<<<<<<<< "::::
```

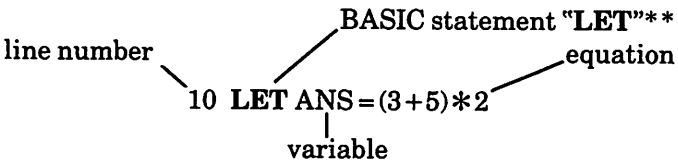
Chapter 7

Input and Reading Data

VARIABLES

While the computer can compute answers and **PRINT** them on the monitor screen, it does not automatically save those answers in its memory. It only **PRINTs** the answers on the screen. Computation instructions in BASIC programs include a way to save the answers so that you can use them later in the program. To save an answer, the computer must be told that something is to hold what the answer is. That "something" is called a *variable.*

Although variables can be used to do much more than hold answers, look for now at how you can type in a program line that uses a variable.



```
10 LET ANS=(3+5)*2
```

Figure 7.1 A program line with an ANS variable.

The **LET statement is optional with TI BASIC and will be omitted in future examples. For instance, this line 10 will be written as:

```
10 ANS=(3+5)*2
```

This tells the computer that you have made up a name for the variable (ANS), and that ANS is the name of the variable it is to use to hold the answer to this equation. From our point of view, the variable ANS acts like a "basket" for the computer to put the answer to the equation into. After this program line has been executed, the variable ANS will have taken the value of 16. That is, the value 16 will be in the basket named ANS. Here are the three steps that the computer takes when it executes this program line.

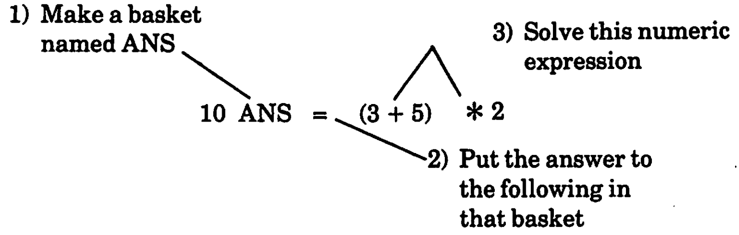

1) Make a basket named ANS
3) Solve this numeric expression
10 ANS = (3 + 5) * 2
2) Put the answer to the following in that basket

Figure 7.2 The three steps of execution.

The name variable tells you a great deal about what it is. Variables are baskets that can hold any single value at a time. In other words, once the basket ANS has been made, it can hold the value of 12, or 19, or 109923, or any number.

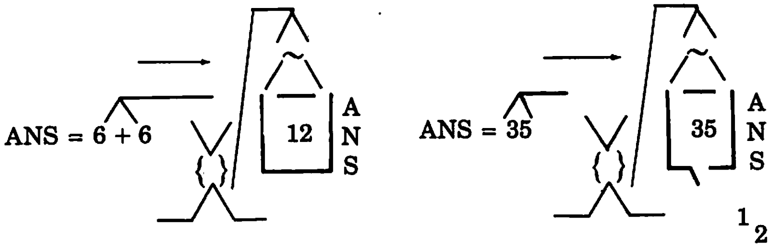


Figure 7.3 The answer variable.

You are the one who decides what to name variables used in the program. The names have certain criteria:

1. They must start with a letter (A, B, C,. . .);
2. they can be up to 15 letters or numbers only in length (no spaces, commas, etc.), and;
3. They must not be a word the computer is programmed by TI BASIC to recognize as a statement or command such as **PRINT**, **LET**, **EDIT**, **LIST**, etc. (called *reserved words*). (See Appendix I for a complete list of reserved words.)

Use of variables is fundamental to programming. They are the means by which the computer can handle volumes of work. The earlier reference to *assignment* as an instruction that tells the computer that something has a value, is a direct reference to the use of variables in computer programs, and an understanding of variables is necessary to our understanding of BASIC programming.

NUMERIC AND STRING VARIABLES

When the computer puts a value into a variable, that value can be either a number such as the answer to an equation or a number you type in. It can also be a group of words, string of letters, or characters such as the literals used in the **PRINT** program lines. The variables used to hold numbers are called numeric variables and the variables used to hold words are called string variables.

Numeric Variables

Numeric variables are the baskets that hold numbers. Before the computer puts a number into a numeric variable, that variable automatically holds the number zero. Once a numeric variable has been assigned a number, that number may be added, subtracted, etc., with the numbers in other numeric variables. The answer can then be assigned as a value to another numeric variable.

```
>10 A=5
>20 B=3
>30 C=A*B
```

In this example there are three numeric variables named: A, B, and C. As this program is executed, the first two program lines (10 and 20) tell the computer to put the numbers 5 and 3 into the variables A and B. The third line (30) tells the computer to perform a computation using the numbers in those variables and assign the answer to the third variable, C.

After line 30 is executed, the variable named C will be holding the number 15.

MORE ON VARIABLE NAMES

To develop good programming technique, take another look at the names that have been given to variables. When BASIC was a relatively new programming language, variable names could only be a single letter, A through Z. It was pretty confusing to read a program full of variables whose names didn't tell you anything about the values they held.

Newer versions of BASIC expanded the number of characters you could use, so now you can give variables names that make sense in terms of the BASIC program's purpose. For instance, if you were using the equation in the last example to compute the number of square feet on the side of a small wall, five feet long and three feet high, you might want to rename your variables as:

```
10 LENGTH=5
20 HGHT=3
30 SQFT=LENGTH*HGHT
```

There may be a few drawbacks to using these longer variable names in your BASIC programs.

1. **More opportunity for typing error.** If you mistype a variable name the computer may assume you mean *make a new basket* in which case there will be the value zero in it.

   ```
   10 LENGTH=5
   20 HGHT=3
   30 SQFT=LENTH*HGHT
   ```

   Misspelled LENGTH, the computer creates a new variable "LENTH" with the value 0 in it, then puts the answer to 0*3 in the basket named SQFT.
2. **Reserved words.** There are a number of words that have been reserved by TI BASIC. With longer variable names, you run the risk of accidentally using one of these words. The problem is easy to overcome but difficult to spot. When you use a reserved word as a variable name, the computer gives the error message INCORRECT STATEMENT or CAN'T DO THAT, both of which may mean several things. This is what makes the problem difficult to spot. After you have figured out that it is a reserved word problem, you simply add or subtract a character or two from the variable name and the computer will accept it.

   ```
   >10 NUM=25
    * CAN'T DO THAT
   >10 NUMB=25
   >
   ```

3. **Memory space.** Remember that the computer has memory space measured in bytes. One byte is roughly equivalent to one character, such as a single letter or a number. Your variable name uses memory, one byte per character. What is in that variable, its value, also uses memory. Therefore, in the case of

longer variable names, you may find that the variable's name uses more memory than the value you have in it.

This is the least pressing of the drawbacks since it only becomes a factor when your BASIC program is approaching the limits of your TI-99/4A's memory.

The tradeoff then, is program readability for the extra time and care required in the choice to use variable names longer than one or two characters. In an effort to make example programs as readable as possible, this book uses multi-character variable names more often than not.

STRING VARIABLES

String variables are the baskets that hold words or anything with characters and/or spaces. The literals used in the **PRINT** program lines are examples of what are called *strings*.

Before the computer puts a string into a string variable, that variable automatically holds a *null* string. In other words, it does hold something, but null means no characters at all.

String variables can hold numbers but, if they do, those numbers cannot be added, subtracted, etc. A telephone number or a postal zip code are good examples of numbers you might assign to string variables since normally there is no reason to do computations with these numbers.

On the other hand, the computer will not put a string into a numeric variable, so you need a way to tell the computer that your variable is meant to hold strings instead of numbers. To do this, type in your string variable names with a dollar sign ($) on the end. The dollar sign tells the computer that the variable is to hold a string instead of a number.

The computer follows these four instructions when it executes this program line:

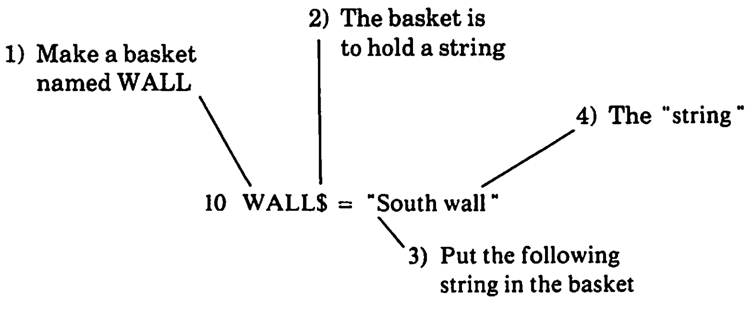

1) Make a basket named WALL
2) The basket is to hold a string
4) The "string"
10 WALL$ = "South wall"
3) Put the following string in the basket

Figure 7.4 The four instructions the computer follows.

After the computer executes this short program, the string variable WALL$ will have the value *South Wall* assigned to it.

When you assign a value to a string variable in this way, enclose the word(s) in quotes. Spaces are often included in strings and are considered to be characters by the computer. The quotes tell the computer exactly where the string begins and ends, and whether to include any leading or trailing spaces in the string. The computer does not consider the quotes themselves to be a part of the string.

MORE ON THE PRINT STATEMENT

There is another variation of the **PRINT** statement to examine. Besides **PRINTing** literals and the answers to numeric expressions, you can **PRINT** the value held by any variable on the monitor screen. To do this type:

PRINT [variable name]

For instance, using our last two examples and combining them with a literal in the **PRINT** program line (line 50), you have the following short BASIC program:

```
>10 WALL$= "South Wall"
>20 LENGTH=5
>30 HGHT=3
>40 SQFT=LENGTH*HGHT
>50 PRINT WALL$::SQFT;"square feet"
>RUN
 South Wall

 15 square feet

** DONE **
```

After you have assigned values to the variables, the computer is instructed to perform a computation (line 40) and then output the results (line 50). The **PRINT** program line can be broken down into its individual instructions as:

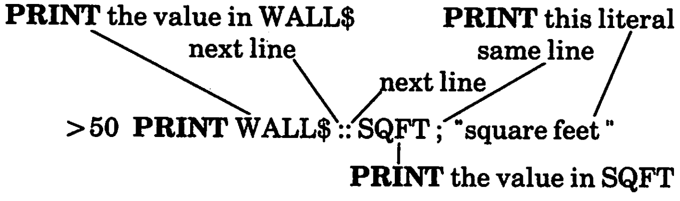


Figure 7.5 The PRINT program line.

This program can now be expanded into a program that can be used to solve a real life problem. This example program will be the basis to explain some of the many ways that variables are used in the BASIC programs.

EXAMPLE PROGRAM "BRICK IN THE WALL"

Suppose you plan to build a small wall on the south side of the hot tub in your back yard. You are going to build it of bricks that measure 2½ x 8½ inches on a side. You want to know how many bricks it is going to take to build the wall in Figure 7.6.

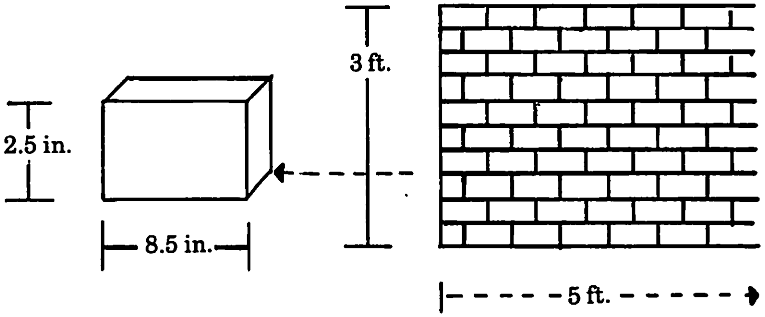


Figure 7.6 How many bricks will it take to build the wall?

Here is how you write the BASIC program.

BASIC program lines

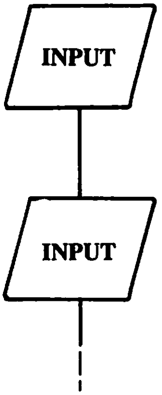
The Flowchart

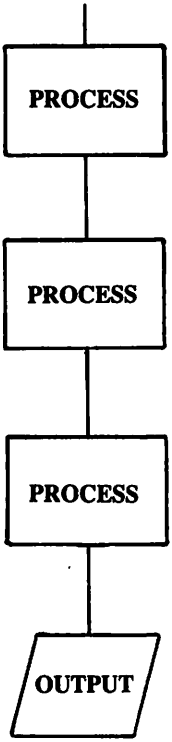
```
10 WALLS = “South Wall”
20 LENGTH = 5
30 HGHT = 3
```

Wall name, size as length and height in feet

```
40 BLEN = 8.5
50 BHGHT = 2.5
```

Brick size (on a side) as length and height in inches





```
60 SQFT = LENGTH
* HGHT
70 WINCH = SQFT *
144
```

Compute wall size in square inches (144 sq. inches/sq. foot)

```
80 BINCH = BLEN *
BHGHT
```

Compute brick size in square inches

```
90 BRKS = WINCH /
BINCH
```

Divide wall size by brick size to get number of bricks

```
100 PRINT
WALL$::“needs”;
BRKS;“bricks”
```

Output the number of bricks required

Figure 7.7 A flowchart.

This program demonstrates an important point about variables: a logical order of statements is necessary to get the program to **RUN** correctly. That is, first put the values in the variables (INPUT), then perform the computations (PROCESS) and **PRINT** the results (OUTPUT).

INPUT → PROCESS → OUTPUT

The first five lines of the program contain all of the strings and numbers that the final answer depends on. These five lines contain the inputs to your program.

```
>10 WALL$ = "South Wall "
>20 LENGTH=5
>30 HGHT=3
>40 BLEN=8.5
>50 BHGHT=2.5
```

Another way to input these five values is to use the **READ** and **DATA** statements of TI BASIC.

READ–DATA STATEMENTS

The **READ** statement tells the computer to look for a **DATA** statement and put what is there into a variable. One advantage of the **READ** and **DATA** statements is that you can assign several values to several variables in fewer program lines. For instance, the first five lines of the BRICK IN THE WALL program could be written in two lines as:

```
>10 READ WALL$,LENGTH,HGHT,BLEN,
    BHGHT
>20 DATA South Wall,5,3,8.5,2.5
```

After line 10 is executed, the five variables in line 10 will be assigned the five values in line 20 in order of what is in the two lines. Thus when the computer reaches the first **READ** statement in a BASIC program, that statement tells it to find the first **DATA** statement in that program and assign the first value it finds there to the first variable in the **READ** statement it is executing. If there is another variable in that **READ** statement, the computer puts the next value it finds in the **DATA** statement in that variable, and so on. Unless you use a special BASIC statement (**RESTORE**), the computer will not **READ** a value in a **DATA** statement more than once while it executes a BASIC program. For instance, you could write your **READ–DATA** statements for this program as:

```
>10 READ WALL$,LENGTH
>20 READ HGHT,BLEN,BHGHT
>30 DATA South Wall,5,3,8.5,2.5
```

After the computer executes line 10, WALL$ and LENGTH will have the values *South Wall* and 5 assigned to them. At this point the computer puts an invisible marker at the value 3 in the **DATA** statement, like a book marker reminds you where you have stopped reading a novel.

```
                      **marker

>30 DATA South Wall,5,3,8.5,2.5
```

When the computer reaches line 20 where it is told to **READ** again, it starts with the value 3 in the **DATA** statement and assigns that value to the first variable in the second **READ** statement (HGHT). It then continues to assign the rest of the values in the **DATA** statement to the rest of the variables in that **READ** statement, in their order of appearance.

In fact, it really doesn't matter how many **READ** or **DATA** statements you use to assign values to your variables. It only matters that the variables and the values are in the correct order and number. For instance, both of these examples will assign the right values to the right variables in the program.

```
>10 READ WALL$,LENGTH,HGHT,BLEN,
    BHGHT
>20 DATA South Wall,5,3
>30 DATA 8.5,2.5
```

```
>10 READ WALL$
>20 READ LENGTH,HGHT,BLEN,BHGHT
>30 DATA South Wall
>40 DATA 5,3,8.5,2.5
```

The correct number means that there must be at least as many values in the **DATA** statement as there are variables in the **READ** statement. If there are fewer values than variables, the **READ** statement tells the computer to **READ** something that isn't there. The computer will not like this. It will complain and give the error message:

```
* DATA ERROR
```

The statement actually executed is the **READ** statement. You must be sure these statements are in the proper logical place in your BASIC program.

DATA statements, on the other hand, are non-executable statements and can be placed anywhere in your program. The **DATA** statement is only there as a place to type in the values that you will be **READing**. Even though **DATA** statements can be scattered throughout the BASIC program, there is a rule of good programming that says to put them at the end of the program. This makes programs easier for you to read, even if the computer doesn't care.

```
>10 READ WALL$,LENGTH,HGHT,BLEN,
    BHGHT
>20 SQFT=LENGTH*HGHT
```

```
>30 WINCH=SQFT*144
>40 BINCH=BLEN*BHGHT
>50 BRKS=WINCH/BINCH
>60 PRINT WALL$:: "needs ";BRKS; "bricks "
>70 DATA South Wall,5,3,8.5,2.5
```

There are two things to notice about how you type in values when you use a **DATA** statement.

```
>70 DATA South Wall,5,3,8.5,2.5
```

1. Use commas to tell the computer that one value is separate from the next value. The last value in the **DATA** statement is not followed by a comma.
2. Since commas tell the comptuer where one value stops and another begins, you need not put quotes around your strings. If you want leading or trailing spaces in the string, put the comma, to the left or the right of that string, the correct number of spaces. If for some reason you want a comma in the string, then you can enclose the string in quotes.

```
DATA "South,Wall ",5,3,8.5,2.5
```

So far you have been able to input values into variables two different ways. Either way, when you **RUN** your program, you find out that it takes 101.6470588 bricks to build a wall five feet long and three feet high.

But suppose you are not sure just how long or high you want to build this wall. Suppose instead that you are more concerned about he fact that you have 247 bricks to use and would really rather have a wall five feet high and eight feet long. Or, you may consider building two walls, one on the south side of the hot tub and one next to the back porch. Actually, you could do a lot of things and are not really sure what you want to do with your 247 bricks.

INPUTTING A VALUE TO FILL A VARIABLE BASKET

In many cases it is desirable to have a program that allows the user to **INPUT** values (strings or numbers) from the keyboard. This lets you use the same program over and over with different values each time.

To do this, type an **INPUT** statement that: 1. tells the user what type of information to enter (prompts the user), and 2. assigns that **INPUT**to an appropriate variable.

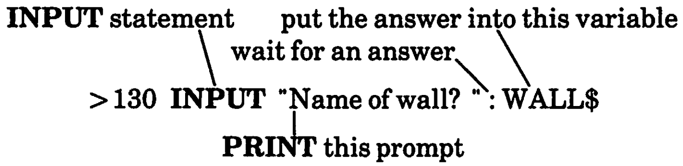

INPUT statement
put the answer into this variable
wait for an answer
>130 INPUT "Name of wall? ": WALL$
PRINT this prompt

Figure 7.8 Using the INPUT statement.

When the computer reaches an **INPUT** statement, it will print the **INPUT** prompt on the monitor screen and then wait for you to type something in from the keyboard. The variable directly following the **INPUT** prompt is the variable basket for what you type. That is, whatever you type and ENTER, your answer to the prompt, is put in that variable. After you have ENTERed your answer, the computer moves on to the next program line and continues to execute your BASIC program.

Now you are going to change three of the lines in the example program so you can ENTER, from the keyboard, a new set of values for the brick wall. By doing this you can ENTER a different set of values each time you **RUN** the program. The rest of the program stays the same as your original example.

```
>10 INPUT "Name of wall? ":WALL$
>20 INPUT "Length of wall? ":LENGTH
>30 INPUT "Height of wall? ":HGHT
```

Now when you **RUN** the program, it starts with an exchange of messages between you and the computer. What you type in response to the prompts gives the computer the information it needs to compute the number of bricks you will need.

```
>LIST
10 INPUT "Name of wall? ":WALL$
20 INPUT "Length of wall? ":LENGTH
30 INPUT "Height of wall? ":HGHT
40 SQFT=LENGTH*HGHT
50 WINCH=SQFT*144
60 BINCH=BLEN*BHGHT
70 BRKS=WINCH/BINCH
80 PRINT WALL$:: "needs ";BRKS; "bricks "
>RUN
 Name of wall? South Wall
 Length of wall? 8
```

```
     Height of wall? 5
     South Wall

   needs 271.0588235 bricks
```

You don't have enough bricks to build the wall eight feet long and five feet high, but you do have a BASIC program that will let you **INPUT** the size walls you are considering, then tell you how many bricks you need to build them.

Another method of writing an **INPUT** statement is to precede it with a prompting **PRINT** statement and not use the prompt option of the **INPUT** statement. In this example, we could type:

The semicolon keeps the **INPUT** on the same line as the prompt

```
>10 PRINT "Name of wall ";
>20 INPUT WALL$
>RUN
  Name of wall?
```

Figure 7.9 Using the PRINT statement to prompt for input.

Notice that if you use the **INPUT** statement without its prompt, it automatically gives you a question mark. Otherwise you need to include a question mark in the **INPUT** prompt as you did in the example program. Note also, when you use the **INPUT** statement this way, you do not use the colon before the **INPUT** variable as you do when you use the **INPUT** prompt.

This alternative method of writing **INPUT** statements is important because only one variable can be in an **INPUT** program line: the variable into which the user's answer will go.

There may be times however, when you will want to use a variable as part of your prompt to the user. For instance, you could write the first few program lines as:

```
>10 INPUT "Name of wall? " WALL$
>20 PRINT WALL$; " Length- ";
>30 INPUT LENGTH
>40 PRINT WALL$; " Height- ";
>50 INPUT HGHT
>RUN
  Name of wall? South Wall
  South Wall Length-?5
  South Wall Height-?3
```

In this example, your prompts use the name of the wall (WALL$) when asking for the length and height of the wall.

SUMMARY

You should be able to write significant BASIC programs now that you have been introduced to the use of variables in programs. With the two types of variables–numeric and string–and the memory to hold their values, the computer becomes much more than a calculator; it becomes a processor of both words and numbers.

There are three ways to put values into variable baskets: the direct assignment (a = 5); the **READ–DATA** statements (**READ** A, **DATA** 5); and the **INPUT** statement (**INPUT** "Enter a number":A). Each way accomplishes the same thing, it puts the value 5 into the variable A, but each of them may be a more suitable method depending on circumstances or preferences.

For instance, if you were sure of the size wall you were going to build and were trying to decide which size brick you were going to buy, you would write the program to **INPUT** the different size bricks you are considering and use direct assignment or **READ–DATA** statements for the rest of the information.

To determine cost of the project, you could write a program line to **INPUT** various prices per brick and have the computer output not only the total number of bricks required, but also the total cost of the bricks.

If you were in business and building brick walls and making bids on jobs were every day occurances, you could write a program to consider all of the costs (bricks, mortar, labor, etc.) which would let you **INPUT** those values that vary from job to job and would output an estimate of what you should bid.

The computer's full processing power will become more apparent to you as you study the statements and functions of BASIC. With what you have learned already, you can turn the computer into a valuable tool.

CHAPTER CHALLENGE

1. What will be the output of this program?

```
10 A=5
20 B=A+1
30 A=B
40 PRINT A
```

2. What will be the output of this program?

```
10 A=5
20 A=A+1
30 PRINT A
```

3. What is wrong with this program?

```
10 A=5
20 READ ANSWER$
30 READ B,C
40 ANS=A*C+B
50 PRINT ANSWER$;ANS
60 DATA 2,4,The answer is
```

4. What will be the output of this program?

```
10 A=5
20 READ ANSWER$
30 READ B,C
40 PRINT ANSWER$;A*C+B;
50 DATA The answer is,2,4
```

5. What will be the output of this program?

```
10 A=5
20 READ ANSWER$
30 READ B,C
40 ANS=A*C+B
50 PRINT ANSWER$;ANS
60 DATA The answer is,2,4
```

6. What is wrong with the logical placement of the program lines in this program?

```
10 A=5
20 ANS=A*C+B
30 PRINT ANSWER$;ANS
40 READ ANSWER$,B,C
50 DATA The answer is,2,4
```

7. What will be the output of the last program?
8. **Batting Averages**

   Your little league team would like to know the batting averages for each of its members. A batting average equals the number of hits each member has had, divided by the opportunities to get a hit (number of times at the plate minus walks and sacrifices). This number is then multiplied by one thousand to get the standard expression.

   If Bill had been to the place 60 times, had eight walks, three sacrifices and had 16 hits, Bill would be batting 327 for the game.

(16/(60−(8+3)))*1000

Write a BASIC program that will let you **INPUT**:

1. the team member's name;
2. the number of hits;
3. how many times that member came to bat;
4. the number of walks; and
5. the number of sacrifices.

The program should **PRINT** the member's name and his/her batting average in the following way:

```
Batting avg. for Chris–327
* * DONE * *
```

9. **Remodel the House**

You have decided to put new carpets and wallpaper in all of the rooms in a five room house. To keep the family budget in line, you are going to remodel one room at a time and need a program that gives a rough estimate for the cost of remodeling each room. You have the following information to work with.

1. All of the rooms in the house have eight foot ceilings.
2. The average cost of wallpaper is $0.45 per square foot of wall.
3. The average cost of carpet is $2.25 per square foot of floor.

Write a BASIC program that will allow you to **INPUT** the name of a room, the length of the room and the width of the room. The program should then **PRINT** the room's name, the number of square feet and cost of both the wallpaper and carpet needed to finish that room. It should also **PRINT** the total cost for remodeling that room.

10. **Gas Dollars**

You know that gas mileage for your car is now important, but what really concerns you is the dollar cost of driving. Suppose you have three cars in the family. They all use different types of gas (at different prices) and are driven different amounts of miles in a week. You use a credit card, 1.5% interest per month, to charge all of your gas.

How much a month is it going to cost you to pay for the gas to drive each car. You have not figured the gas mileage for these cars yet so, write a BASIC program that will let you **INPUT**:

1. beginning mileage;
2. ending mileage;
3. gallons to fill the tank;

4. cost per gallon of gas; and
5. estimated miles per week you will be driving.

The program should **PRINT** the gas mileage figure and the estimated cost of driving that car for one month (including interest). There are 4.3 weeks in a standard month.

11. **Ham and Tuna Sandwiches**

Thirty-five couples have been invited to a lawn party. Ham sandwiches and tuna sandwiches are going to be served. The portion and price figures for the ingredients of these sandwiches is given below.

Table 7.1 Portions and prices of ham and tuna sandwiches.

	Ham Sandwich	Tuna Sandwich	Cost
Ham	3 ounces	–	\$0.15/oz.
Tuna	–	2 ounces	\$0.20/oz.
Bread	2 pieces	2 pieces	\$010/pc.
Mayonnaise	½ ounce	1 ounce	\$0.05/oz.
Mustard	½ ounce	–	\$0.08/oz.
Pickles	–	½ ounce	\$0.07/oz.

Write a BASIC program that uses **READ–DATA** statements to input these portion and price figures and allows the user to **INPUT**:

1. a predicted percentage of couples attending;
2. an estimated number of sandwiches needed for each couple; and
3. an estimate of the percentage of sandwiches that will need to be ham sandwiches.

The program should then **PRINT** the total quantity of each ingredient needed to make the sandwiches and the total cost of making them.

ANSWERS

1. 6 (The variable A takes a new value in line 30.)
2. 6 (Line 20 can be read as "A takes the value of itself plus one")
3. The computer will give an error message when it tries to **READ** "The answer is" into the variable C.
4. The answer is 14. (You can **PRINT** the answer to numeric expressions that use variables.)
5. The answer is 14.

6. The **READ** statement should be before line 20. Otherwise the computer is instructed to perform computations and output with variables that have not been assigned values yet.
7. 0
8. **Batting Averages**

```
>10 CALL CLEAR
>20 INPUT "Name? ":MEMB$
>30 INPUT "Hits? ":HITS
>40 INPUT "Times at bat? ":UPS
>50 INPUT "Walks? ":WALKS
>60 INPUT "Sacrifices? ":SACS
>70 CHANCES=UPS-(WALKS+SACS)
>80 AVG=(HITS/CHANCES)*1000
>90 CALL CLEAR
>100 PRINT "Batting avg. "::MEMB$;"- ";INT(AVG+.5)
>RUN
[screen clears]
Name? Chris
Hits? 16
Times at bat? 60
Walks? 8
Sacrifices? 3
[screen clears]
Batting avg.

Chris-327

* * DONE * *
```

9. **Remodel the House**

```
>10 CALL CLEAR
>20 HGHT=8
>30 PAPER=.45
>40 RUG=2.25
>50 INPUT "Name of room? ":RM$
>60 INPUT "Length in feet? ":LGNTH
>70 INPUT "Width in feet? ":WDTH
>80 FLOORSQFT=LGNTH*WDTH
>90 WALLSQFT=(LGNTH+WDTH)*HGHT*2
>100 CALL CLEAR
>110 PRINT "*** ";RM$;" *** "::
>120 PRINT "Wall Paper- ";WALLSQFT;"Square feet "
>130 PRINT TAB(12);INT(PAPER*WALLSQFT);
     "dollars "::
```

```
>140 PRINT "Carpet      - ";FLOORSQFT; "Square feet "
>150 PRINT TAB(12);INT(RUG*FLOORSQFT); "dollars "
>RUN
[screen clears]
Name of room ? 2nd Bedroom
Length in feet ? 17
Width in feet ? 14
[screen clears]
* * * 2nd Bedroom * * *

Wall paper-496 Square feet
           223 dollars

Carpet     -238 Square feet
           535 dollars
```

* * DONE * *

10. **Gas Dollars**

```
>10 CALL CLEAR
>20 INPUT "Beginning mileage ? ":BMILE
>30 INPUT "Ending mileage ? ":EMILE
>40 INPUT "Gallons to fill ? ":GALS
>50 INPUT "Cost of gas/gallon ? ":GASCOST
>60 INPUT "Miles/week will drive ? ":MILES
>70 MPG=(EMILE-BMILE)/GALS
>80 MONTHCOST=( (MILES*4.3)/MPG)*GASCOST)*
   1.015
>90 CALL CLEAR
>100 PRINT "            *** GAS   *** "
>110 PRINT "*** MILEAGE/MONTHLY
     COST ***"::
>120 PRINT INT(MPG*100)/100; "Miles per gallon "::
>130 PRINT  "$ ";INT(MONTHCOST*100)/100; "Monthly
    gas bill "
>RUN
[screen clears]
Beginning mileage ? 347
Ending mileage ? 544
Gallons to fill ? 13.2
Cost of gas/gallon ? 1.13
Miles/week will drive ? 220
[screen clears]

            *** GAS   ***
***   MILEAGE/MONTHLY COST   ***

14.92          Miles per gallon

$  72.7 Monthly gas bill

* * DONE * *
```

11. **Ham and Tuna Sandwiches**

(Note: program sections are labeled to help reading.)

```
>5 CALL CLEAR
input values
>10 PAIRS=35
>20 READ HAM,HBRD,HMAYO,HMUST
>30 READ TUNA,TBRD,TMAYO,TPICK
>40 READ HAMCST,TUNACST,BRDCST,MAYOCST,
    MUSTCST,PICKCST

>70 INPUT "Attendance % ? ":PER
>80 INPUT "Sandwiches per couple ? ":SANDS
>90 INPUT "Ham sandwich % ? ":HPER
compute number of sandwiches
>100 TOTALSANDS=PAIRS*PER*SANDS
>110 HAMS=INT(TOTALSANDS*HPER)
>120 TUNAS=INT(TOTALSANDS-HAMS)
compute total cost
>50 HSAND=HAM*HAMCST+HBRD*BRDCST+
    HMAY*MAYOCST+HMUST*MUSTCST
>60 TSAND=TUNA*TUNACST+TBRD*BRDCST+
    TMAYO*MAYOCST+TPICK*PICKCST
>130 TOTALCST=HAMS*HSAND+ TUNAS* TSAND
compute total portions
>140 TOTHAM=HAM*HAMS
>150 TOTTUNA=TUNA*TUNAS
>160 TOTBRD=HBRD*HAMS+TBRD*TUNAS
>170 TOTMAYO=HMAYO*HMAS+TMAYO*TUNAS
>180 TOTMUST=HUMST*HAMS
>190 TOTPICK=TPICK*TUNAS
output the answers
>195 CALL CLEAR
>200 PRINT "***   PARTY SANDWICHES   *** "::
>210 PRINT HAMS; "Ham sandwiches "::
>220 PRINT TUNAS; "Tuna sandwiches "::
>230 PRINT " ======================== "::
```

```
>240 PRINT "Ham ",TOTHAM;TAB(22); "ounces "
>250 PRINT "Tuna ",TOTTUNA;TAB(22); "ounces "
>260 PRINT "Bread ",TOTBRD;TAB(22); "pieces "
>270 PRINT "Mayonaise ",TOTMAYO;TAB(22); "ounces "
>280 PRINT "Mustard ",TOTMUST;TAB(22); "ounces "
>290 PRINT "Pickles ",TOTPICK;TAB(22); "ounces ":::
>300 PRINT "TOTAL COST-$ ";TOTALCST
```

data

```
>310 DATA 3,2,.5,.5
>320 DATA 2,2,1,.5
>330 DATA .15,.2,.1,.05,.08,.07
```

```
>RUN
```

screen clears

```
Attendance % ? .75
Sandwiches per couple ? 3
Ham sandwich % ? .65
```

screen clears

```
***  Party Sandwiches  ***

51 Ham sandwiches

27 Tuna sandwiches

          ==========================
    Ham             153     ounces
    Tuna            54      ounces
    Bread            156     pieces
    Mayonaise       52.5    ounces
    Mustard         52.5    ounces
    Pickles         13.5    ounces

    TOTAL COST-$ 51.055

    * * DONE * *
```

Chapter 8

Program Branching and Loop Structures

Instructions of *assignment*, one of the three fundamentals of computer programs, was discussed in the preceeding chapter. Along with the other two fundamentals, *condition* and *iteration*, you begin to realize the full processing power of the TI-99/4A.

As the power of the programs increases, so does the difficulty in reading the program **LISTings**. Such techniques as using longer variable names are useful to help remember what they are holding. Keeping track of what value is in what variable is by far the most challenging aspect of using program variables. A technique of program documentation is the **REM** statement.

REM STATEMENTS

REM statements are the way you write notes to yourself within the program. First type in the line number then **REM**, which stands for *remark*. After **REM** you can type anything you want. **REM** tells the computer to ignore what is on that line; it is not an instruction but simply a note to yourself.

```
100  REM Program Name—Home Budget
  •
  •
  •
180  REM Input the dollar values
  •
  •
  •
250  REM Compute avg. car expense
  •
```

```
•
•
520 REM This is the output section
•
•
•
and so on
```

As your programs become longer and more complicated, you may want to include **REM** statements to remind yourself what the program is doing at that particular point. As the example programs progress, you will use them frequently.

PROGRAM BRANCHING

A program *branch* instructs the computer to alter its normal sequence of excecuting program lines one after another by telling it to **GO TO** another line number and continue from there. In its simplest form, the **GO TO** statement (which may be typed as either **GO TO** or **GOTO**) is used to create what is called an *unconditional* branch.

The **GOTO** statement The line number to **GO TO**

>60 **GOTO** 120

Figure 8.1 An unconditional branch.

When the computer reaches line 60, it will unconditionally **GO TO** line 120 and continue executing the BASIC program from there. It will skip over any lines between lines 60 and 120.

The only reason this is known as an unconditional branch is the fact that you can also have what is known as a *conditional* branch. You can tell the computer that **IF** a certain condition is true, **THEN** go to another line number.

The condition The line number to go to

>60 **IF** A=5 **THEN** 120

Figure 8.2 A conditional branch.

The **IF–THEN** statement gives the computer the ability to decide if it should go to another line number or not. This statement works by setting up a condition: **IF** the condition is met (is true), **THEN** the computer is instructed to go to another line number. Otherwise (in this case if A does not equal 5), the computer moves on as it would normally and executes the next line in the program. Notice that the **GOTO** is implied in an **IF–THEN** statement and that the statement is madeup of two words separated by the condition that you want to check.

The key to using **IF–THEN** statements in the BASIC programs is the type of conditions that you may use. These conditions almost always involve at least one variable and are typed in using the standard signs:

<	less than
>	greater than
=	equal to

Relational Operators

The signs <, >, = are known as *relational operators* since they describe a relationship (less than, greater than, or equal to). The following is a list of all the possible relationships that the variable A can have with the number 5 and shows how you can type in those conditions.

IF A<5	If A is less than 5
IF A< =5	If A is less than or equal to 5
IF A=5	If A equals 5
IF A< >5	If A is less than or greater than 5 (not equal)
IF A= >5	If A is equal to or greater than 5
IF A>5	If A is greater than 5

All of these possible relationships apply to string variables and literals as well. For instance,

IF ANS$ = "YES " **THEN** 300

is a perfectly valid expression in BASIC. The ability to compare strings is how the computer can alphabetize lists of names, etc., or check to see (as in this case) if a user **INPUT** is a particular answer,

IF ANS$ = "YES "

It is interesting to note how the computer views a condition in an **IF–THEN** statement. **IF** the condition is met (is true) then the computer assigns the value 1 to that condition. If the condition is not met (is not true) then the computer assigns the value zero to that condition.

The computer then checks to see if the value for that condition is something other than zero (0 = not true). Only if the value is something other than zero will the computer continue to the next part of the **IF–THEN** statement: **GO TO** a line number. If the value is zero then the computer ignores the **GO TO** instruction and moves on to execute the next line in that program.

This means you can check two or more conditions at the same time by either multiplying or adding the values that the computer assigns to each condition in an **IF–THEN** statement.

Logical Operators

Logical operators test two or more conditions at the same time. The two logical operators you will use are *and* and *or*.

IF A > B and C = D **THEN** [line number]
IF A < C or B = A **THEN** [line number]

You cannot use the words *and* or *or* in a program line, but you can accomplish the same thing with the mathematical operations * (multiplication) and + (addition).

IF(A > B)*(C = D) **THEN** [line number]
IF (A < C)+(B = A) **THEN** [line number]

IF Condition *and* Condition

Suppose that: A = 6, B = 4, C = 7, D = 7.

This condition is true, its value is 1

IF (A > B)*(C = D) **THEN** [line number]

This condition is true, its value is 1

1 times 1 equals 1 (true), the condition is met and the computer goes to the line number stated after the **THEN** part of this **IF–THEN** statement.

If either of the conditions in this *and* example had been false, that is had the value 0, the answer to (Condition*Condition) would have been false, i.e., 1*0 = 0 and 0*1 = 0.

Only if both conditions, Condition 1 *and* Condition 2 are true (equal to 1) does the computer go to the stated line number.

IF Condition *or* Condition

Suppose again that: A = 6, B = 4, C = 7, D = 7.

This condition is true, its value is 1

IF (A < C)+(B = A) **THEN** [line number]

This condition is not true, its value is 0

1 plus 0 equals 1 (true) and the computer goes to the line number stated in the **IF–THEN** statement.

If either condition 1 *or* Condition 2 are true (have the value 1), adding them together results in a value greater than zero and the computer goes to the stated line number.

Only if both conditions are false (0 + 0) will the computer ignore the **GO TO** instruction in the **IF–THEN** statement and move to the next line in your BASIC program.

PROGRAMS FOR MANY USERS

There are times when you want BASIC programs to handle a variety of computations or processes, depending on the particular circumstances. When this is the case, you can write programs to ask the users questions about the situation, and based on their answers, branch your program to the appropriate computations or processes for that user.

For example, you write a program that will **PRINT** out the ideal body weight for a person to have. Ideal body weight may depend on three things: the person's height, sex, and frame. If the program is to work it needs to be able to branch to the computation that fits the individual person.

When you reach this level of sophistication in the BASIC programs, you flowchart the program before you type it in. The following flowchart describes the BASIC program you are about to write.

Begin the program by asking for the person's height, then ask if they are male or female.

```
>100 INPUT "Your height in inches? ":HGHT
>110 INPUT "(M)ale or (F)emale? ":SEX$
```

So far, the user is expected to first type and enter their height in inches and then enter either an M for male or an F for female (a string value). Now use your first **IF–THEN** statement to begin branching to the appropriate places:

```
>120 IF SEX$ = "M" THEN 270
```

If the user does not enter an M in line 110, the computer will not go to line 270. This means you can write the next line (130) with the assumption that the user is a female. Actually, you will write several lines to set up the next branches, different frame sizes for females.

```
>130 PRINT "  Female Frame Size  "
>140 PRINT "------------------"
>150 PRINT "          1) Large       "
>160 PRINT "          2) Medium     "
>170 PRINT "          3) Small       "::
>180 INPUT "Enter the correct number ":FRAME
```

Conceptually, the program branches look something like this:

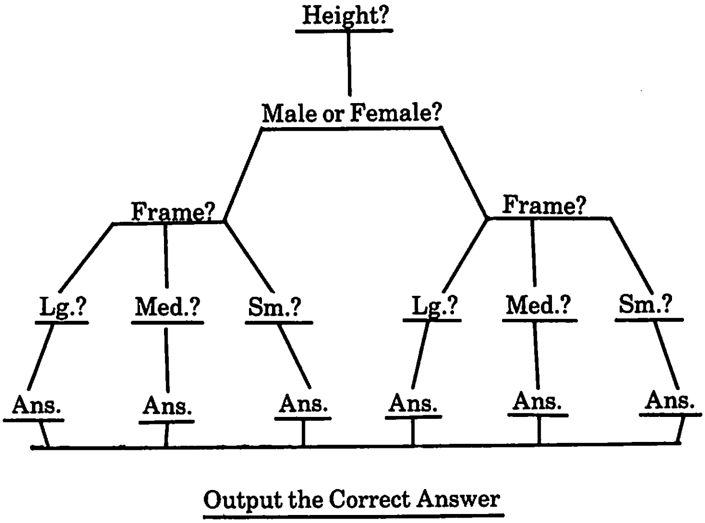


Figure 8.3 A branching model.

At this point, a female should type in the number 1, 2, or 3. Based on what number she enters, the program branches to the right computation for her.

```
>190 IF FRAME = 2 THEN 230
>200 IF FRAME = 3 THEN 250
```

```
>210 WEIGHT=HGHT*1.95
>220 GOTO 400
>230 WEIGHT=HGHT*1.70
>240 GOTO 400
>250 WEIGHT=HGHT*1.55
>260 GOTO 400
```

Immediately after doing the correct computation and putting the answer in the variable basket WEIGHT, you will branch again (**GOTO** 400), sending the computer to the **PRINT** statement that tells the user his/her ideal weight. At line 400 you will have the statement:

```
>400 PRINT "Your ideal weight is ";WEIGHT
```

What if the user is a male? Remember that you sent the computer to line 270 if the **INPUT** SEX$ = "M". Start at line 270 then and do the same type of thing you did for the female computations. The finished program looks like this:

```
>90 REM PROGRAM IDEAL WEIGHT
>100 INPUT "Your height in inches? ":HGHT
>110 INPUT "(M)ale or (F)emale? ":SEX$
>120 IF SEX$ = "M" THEN 270
>130 PRINT "  Female Frame Size  "
>140 PRINT "------------------"
>150 PRINT "        1) Large       "
>160 PRINT "        2) Medium   "
>170 PRINT "        3) Small       "::
>180 INPUT "Enter the correct number ":FRAME
>190 IF FRAME=2 THEN 230
>200 IF FRAME=3 THEN 250
>210 WEIGHT=HGHT*1.95
>220 GOTO 400
>230 WEIGHT=HGHT*1.70
>240 GOTO 400
>250 WEIGHT=HGHT*1.55
>260 GOTO 400
>270 PRINT "  Male Frame Size  "
>280 PRINT "------------------"
>290 PRINT "        1) Large    "
>300 PRINT "        2) Medium   "
>310 PRINT "        3) Small    "::
>320 INPUT "Enter the correct number ":FRAME
```

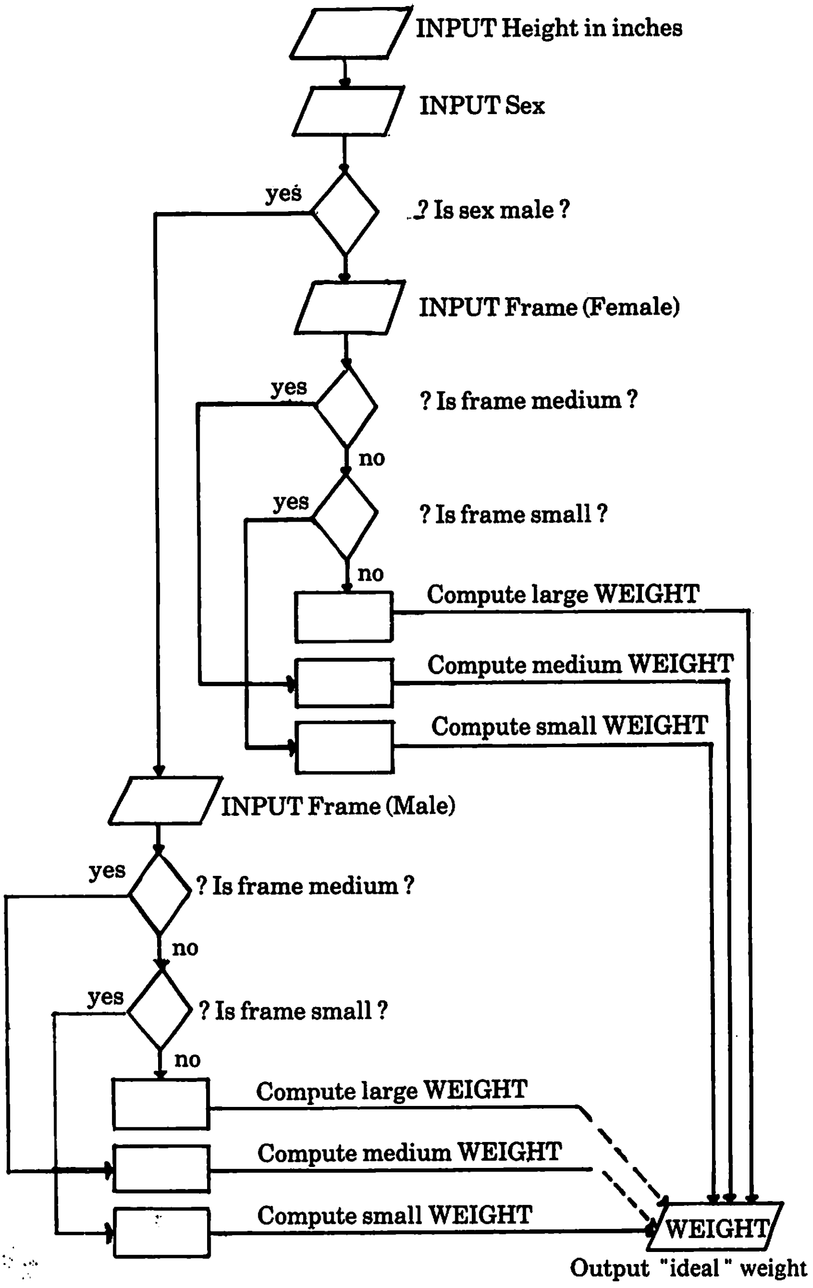


Figure 8.4 A flowchart for an ideal-weight program.

```
>330 IF FRAME=2 THEN 370
>340 IF FRAME=3 THEN 390
>350 WEIGHT=HGHT*3.20
>360 GOTO 400
>370 WEIGHT=HGHT*2.75
>380 GOTO 400
>390 WEIGHT=HGHT*2.25
>400 PRINT "Your ideal weight is ";WEIGHT
```

Note: A capital letter has a different value than the same letter in lower case. If you want you program to recognize M *or* m in line 120, type:

```
>120 IF (SEX$= "M")+(SEX$= "m") THEN 270
```

This sample program illustrates a more subtle lesson about computer programming.

GIGO stands for *Garbage In, Garbage Out.* It refers to the fact that computers depend on correct input to give correct output. If you entered height as a negative two inches (–2), sex as Z and frame as 17, by the instructions in your program the computer would assume that the user was female and her frame was large. It would then output:

```
Your ideal weight is -3.9

* * DONE * *
```

To avoid this problem of GIGO, have your programs **PRINT** out the **INPUTs** with which it is working, when it provides its final answers. For instance, you can include a few lines in the BASIC program for the following final output:

```
HEIGHT=68
SEX=F
FRAME=2

Your ideal weight is 115.6

* * DONE * *
```

On the other hand, there is the little known acronym, *GAG* which stands for *Garbage, All Garbage.* This occurs when the assumptions built into the program are incorrect or not understood by the user.

The example program, *Ideal Weight* demonstrates the idea of branching a program. Very few medical experts would recommend that you pay any attention to this program's output. Aside from the fact that many experts don't believe there is such a thing as ideal weight by height, sex, and frame, there is also the fact that the figures by which you multiply HEIGHT to get WEIGHT were made up. There is absolutely no scientific basis for the answers this program comes up with.

Even though you improve the program's reliability and reduce the chance of GIGO, there is no way to overcome the program's lack of validity and its output is Garbage All Garbage regardless of the INPUTs.

Far too many people are willing to change their lifestyles or their behavior in some way because the computer produced a report. Remember that the program produced the report and that the program is only as good as the programmer's ability or intentions.

LOOP STRUCTURES

You have heard of *loops* as something a computer program does. A loop instructs the computer to repeat the execution of one or more instructions already executed. To do this, it uses statements that branch backwards in the program. The fundamental of iteration, mentioned previously, refers directly to loops in the BASIC programs.

Loops combine variables with a process, using new values in the variable baskets, each time the process is repeated. The following example program demonstrates a simple loop.

```
>10 PRINT X;
>20 X=X+1
>30 GOTO 10
```

There are a total of three lines in this program. The last one, line 30, instructs the computer to **GOTO** line 10. This instruction sets up a loop that keeps the computer executing the program until you either turn off the machine or interrupt execution with the FCTN 4 keys. For this reason, it is known as an endless loop.

As the computer begins to execute this short program, it starts with line 10. Here it is told to **PRINT** the value in the variable basket X. Since the variable has not been assigned a value yet, it automatically holds the value of zero. So the computer will print zero. The semicolon at the end of the **PRINT** statement tells the computer to stay on the same line to **PRINT** the next thing it is told to **PRINT**.

The computer is finished with line 10 so it moves to line 20. This line instructs the computer to assign a new value to X: the value it already has, plus one. In other words, the variable X is to take a new value, the value of itself plus one. The value assigned to X is now one (X = 0 + 1).

The computer is finished with line 20 so it moves to line 30 which simply tells the computer to **GOTO** line 10, which it does. Arriving at line 10, the computer is instructed to **PRINT** the value assigned to X which is now one.

The net effect of running this program is to have the computer **PRINT** every whole number from one to infinity (or at least until it reaches a very large number). That is, the computer is instructed to print X, increase X, print X, increase X, and so on. The variable X is used as what is commonly called a counter variable. For all practical purposes, it counts the number of times the loop is completed. You can begin to appreciate the potential power of iteration when you consider that three lines of program generate, essentially, an infinite amount of output in this simple example.

This type of endless loop is not very practical. For programming purposes, you need to control the number of times a loop is executed. One way to do this is to include a condition statement inside the loop. The following uses an **IF–THEN** and **GOTO** statement to tell the computer to leave the loop when the value in X is greater than 25.

```
>10 PRINT X;
>20 X=X+1
>25 IF X>25 THEN 40
>30 GOTO 10
>40 END
>
>RUN
 0     1     2     3     4     5     6     7     8
 9      10      11      12      13      14      15
 16      17      18      19      20      21      22
 23    24    25

* * DONE * *
```

The **END** statement tells the computer it is through executing the program. It is not always necessary, but including it in your programs is generally considered to be good form.

Another way of accomplishing the same thing is to write the program using an **IF–THEN** statement as the instruction that starts and stops the loop.

```
>10 PRINT Z;
>20 Z=Z+1
>30 IF Z<26 THEN 10
>40 END
>
>RUN
 1      2      3      4      5      6      7      8
 9      10     11     12     13     14     15
 16     17     18     19     20     21     22
 23     24     25

* * DONE * *
```

In either case, you use a **GO TO** instruction to establish a loop, a counter variable to keep track of how many times the loop is executed and a condition statement to stop execution of the loop.

There is another way to set up loops within the BASIC program. It is fundamentally the same as the loops you have just covered. It too uses a value held in a counter variable as the key to starting and stopping the loop, and, it uses a type of condition statement to monitor that variable during the loop's execution. The difference is mostly in its readability and shorthand type format.

THE FOR–NEXT LOOP

FOR–NEXT Example	Explanation
>10 **FOR** X=0 **TO** 25	>10 Start X at 0 and end the loop when X=25
>20 **PRINT** X	>20 **Print** the value in X
>30 **NEXT** X	>30 Add one to X. **If** X is less than or equal to 25, **THEN GOTO** line 20. Otherwise continue on to line 40
>40 **END**	

In this type of loop a statement at the beginning sets the number of times the loop will be completed. It uses a counter variable, like the X=X+1 in the **GOTO** loop; but this counter is assigned its beginning value in the **FOR** statement and is increased by one at the **NEXT** statement. The condition statement is hidden in both the **FOR** and the **NEXT** statements. The **FOR** statement sets the limit for the variable, then the **NEXT** statement makes comparisons to see if that variable is over its limit.

The **FOR** statement can also set how much X is increased when the computer reaches the **NEXT** statement. When you want X to increase by a number other than 1, add **STEP** [the number] to the **FOR** statement as follows:

```
>10 FOR X=0 TO 25 STEP 5
>20      PRINT X;
>30 NEXT X
>40 END
```

Since X is increased by five rather than by one each time the loop is executed, the results of **RUNing** this program would be:

```
>RUN
 0    5    10    15    20    25

* * DONE * *
```

You can count the values in X backwards. The following shows how you could change this program to start X with the value of 25, decrease it by five each time through the loop and stop when X is less than zero.

```
>10 FOR X=25 TO 0 STEP -5
>20      PRINT X;
>30 NEXT X
>40 END
>
>RUN
 25    20    15    10    5    0

* * DONE * *
```

Nested Loops

Nested loops are loops within loops. You can use nested loops to repeat a loop with a new set of variables for it to work on. One of the more popular forms of nested loops are nested **FOR–NEXT** loops. A simple example shows the mechanics of how a nested loop is executed by the computer.

```
>100 FOR X=1 TO 3
>110      FOR Y=1 TO 5
```

```
>120          PRINT X*Y;
>130       NEXT Y
>140 PRINT "iteration ";X::
>150 NEXT X
>RUN
 1    2    3    4    5    iteration    1

 2    4    6    8    10    iteration    2

 3    6    9    12    15    iteration    3

** DONE **
```

The inside loop is this:

```
>110      FOR Y=1 TO 5
>120          PRINT X*Y;
>130      NEXT Y
```

It **PRINTs** X*Y five times and is repeated three times by the outside loop:

```
>100 FOR X=1 TO 3
>110
>120
>130
>140 PRINT "iteration ";X::
>150 NEXT X
```

Output for the inside loop (**PRINT** X*Y); changes as the counter variables change while the loops are being executed.

```
>RUN
 1    2    3    4    5    iteration    1

 2    4    6    8    10    iteration    2

 3    6    9    12    15    iteration    3

** DONE **
```

As the inside loop (**FOR** Y = 1 **TO** 5) is executed, Y will take the value one through five, and the inside loop is executed three times by the outside loop (**FOR** X = 1 **TO** 3). Since X takes a new value at each iteration of the outside loop, output (X*Y) will change each time the outside loop loops. The value of X changes as the output of the outside loop (**PRINT** "iteration";X::) is executed.

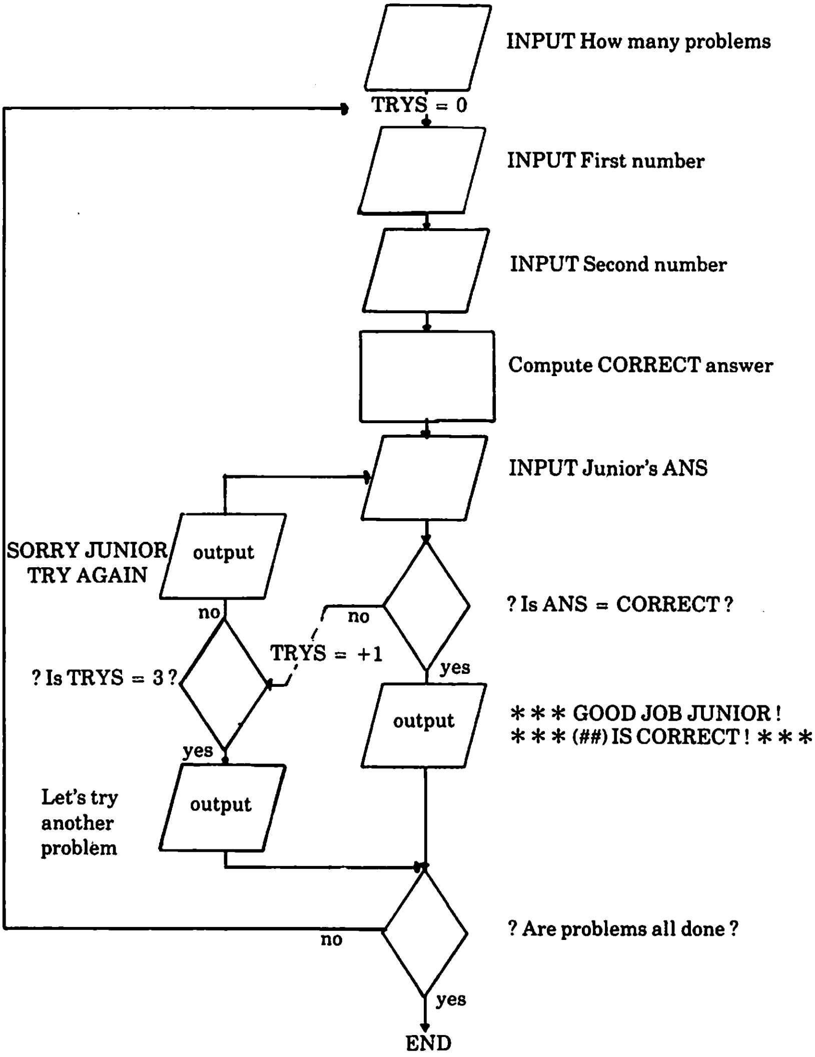


Figure 8.5 A flowchart for a math problems program.

Here's how you could write a BASIC program using nested loops to help a friend with his homework.

Homework Helper

Junior is learning his multiplication tables at school. He said that his teacher sent him home with a whole page of problems to complete. Since Junior isn't the "brightest light on the porch" when it comes to math, write a BASIC program to help him with the problems.

You are going to write a program that will start by asking Junior how many problems he has to work. Then you are going to write a program that gives Junior three chances to answer each problem. If he gets the answer right, then the program will ask Junior for the next two numbers he has to multiply.

```
90 REM Homework Helper
   100 INPUT "How many problems Jr.? ": PROBS
   110 FOR X = 1 TO PROBS
   120      TRYS = 0
   130      INPUT " First number ? ": A
   140      INPUT " Second number ? ": B
   150      CORRECT = A*B
   160           PRINT "What is the answer to "
   170           PRINT A; "times ";B;
   180           INPUT ANS
   190           IF ANS = CORRECT THEN 250
   200                TRYS = TRYS+1
   210                IF TRYS = 3 THEN 270
   220                PRINT "SORRY JUNIOR "
   230                PRINT "TRY AGAIN "
   240           GOTO 160
   250      PRINT "***   GOOD JOB JUNIOR
            !  ***"
   260      PRINT "*** ";ANS; "IS CORRECT
            !  ***"
   270 CALL CLEAR
   280 PRINT "Let's try another problem "
   290 NEXT X
```

First have Junior enter the number of problems he brought home. Put this number in the variable PROBS. Note: Use the variable PROBS to set the limit on how many times the outside loop is executed. This means that you can use the same program to do 10 or 15 or any number of homework problems.

TRYS is the counter variable for your **IF–THEN** loop (the inside loop). This is the variable that lets Junior have up to three TRYS to get the correct ANS. Since TRYS counts for your inside loop (TRYS = TRYS + 1), set it back to zero each time you execute that loop. Otherwise, Junior would only get three incorrect TRYS for the whole set of problems.

You can exit the inside loop in two ways, either Junior gets the right answer, or he uses up his TRYS (line 190 or line 210). Either way, the program will end up on **NEXT** X where the computer checks to see if the outside loop has been executed PROBS times. That is, if PROBS = 1 then once, if PROBS = 2 then twice, etc. If the outside loop has not been executed PROBS times, Junior starts all over again by entering two new numbers to multiply.

SUMMARY

Branching and loops can become quite complicated. Multiple branches and loops within loops are common in programming. A beginning programmer will need practice before program branching and loop structure become second nature.

All of the loops in this section had indented lines to show at a glance where the loop began and ended. This is a form of documentation and has nothing to do with how the program executes. In fact, the spaces in front of a program line typed on the TI-99/4A are automatically erased when that line is **LISTed**.

Note: Program branches and loops are a powerful means to instruct the computer. Also, variables are the key to efficient use of branches and loops in the BASIC programs.

This completes the introduction of the three fundamentals of program execution—assignment, condition, and iteration. By now, you should be getting a feel for the nature of programming. There are many ways to combine instructions; but, you must follow a particular structure. This is a logical structure, and as long as you stay within the structure, what you can do with your program instructions is limited only by your imagination.

CHAPTER CHALLENGE

1. What will be the output of this program?

```
>10 X=5
```

```
>20 Y=3
>30 GOTO 50
>40 PRINT "HELLO ";
>50 PRINT X*Y
>60 END
```

2. What will be the output of this program?

   ```
   >10 A$= "TEST1 "
   >20 IF A$= "TEST2 " THEN 60
   >30 PRINT A$
   >40 A$= "TEST2 "
   >50 GOTO 20
   >60 PRINT A$
   >70 END
   ```

3. What will be the output of this program?

   ```
   >10 PRINT X;
   >20 IF X*X=81 THEN 50
   >30 X=X+1
   >40 GOTO 10
   >50 END
   ```

4. What will be the output of this program?

   ```
   >10 IF NUMB=9999 THEN 80
   >20 READ NUMB
   >30 IF NUMB<20 THEN 50
   >40 GOTO 10
   >50 PRINT NUMB;
   >60 GOTO 10
   >70 DATA 15,35,12,66,22,13,7,9999
   >80 END
   ```

5. **Border Bucks**

   When you perform conversions such as feet to inches, gallons to liters, weeks to day, etc., it is often desirable to be able to perform those conversions both ways. One such conversion might be the exchange of Canadian dollars for American dollars.

   Every day you find an exchange rate listed in the newspaper for these two currencies. This rate is listed as the portion of a Canadian dollar that is equal to one American dollar. Your conversions then, would be computed as follows:

Canadian to American	American to Canadian
Can$*rate	Amer$*(1/rate)

If, for example you wanted to convert 500 American dollars to Canadian dollars at an exchange rate of .82375, your equation would be:

500 * (1/.82375)

and your answer would be $606.98 Canadian dollars.

Write a BASIC program that lets the user **INPUT** which conversion is desired and the amount of currency being converted along with the most recent exchange rate. The program should then output the amount after conversion.

6. **Miles vs Mouth**

   You should all know that your weight is a matter of the calories you take in and the calories you burn off. If they are the same, your weight stays the same.

   Suppose that jogging is your way to keep your calorie intake/burn off at a desirable level and that you know that one mile of jogging burns off 85 calories.

   What you want is a BASIC program that will tell you 1. how many miles you need to run to burn off **INPUT** X number of calories; or 2. how many calories you will burn off by running **INPUT** M number of miles.

   Write a program that allows the user to **INPUT** which computation is desired then **INPUT** the appropriate numbers. The program should output the answer with a simple word of encouragement.

7. **Travel Expense**

   Suppose you have two cars, an older Pontiac that runs on regular gas at 17 miles to the gallon and a newer Mercury station wagon that gets 11 miles to the gallon with unleaded regular, or 13 miles to the gallon with unleaded supreme gas. You want to have a program that will tell you how much the gas will cost for various trips you are planning. You know that this will depend on which car you plan to take and what type of gas you plan to use if you are planning on taking the Mercury.

   Write a BASIC program that lets the user **INPUT**:

   1. the length of the trip in miles;
   2. the car being taken;
   3. if the car is the Mercury, then what type of gas will be bought; and
   4. the price of the type of gas for the car being taken.

The program should then OUTPUT the amount of money it will cost to buy gas for the trip.

8. **More Junior**

   You are still having problems with Junior. His teacher sent home a note to express her concern that, although Junior got all the problems on his homework assignment correct, he failed to pass a test on the subject the very next day. Evidently, Junior figured that he could **RUN** the program you wrote for him over and over until he had guessed all the correct answers. But you are not discouraged!

   Write a BASIC program that will generate **RANDOM** numbers between one and 10, and that uses two of those numbers at a time to set up multiplication problems for Junior to solve. Write the program so that it sets up 10 problems and keeps track of how many times Junior answered correctly and how many times he answered incorrectly.

   If Junior gets anything less than 90 percent correct answers, i.e., nine out of 10 correct, the program should set up another 10 problems. When he does get over 90 percent, the program should **PRINT** out a message telling Junior where you hid a bag of his favorite candy for him.

   P.S. Just hope Junior doesn't figure out how to **LIST** a program.

Answers

1. 15 The program skips over line 40 **PRINT** "HELLO "
2. TEST1
   TEST2

   The order that the computer will execute this program is (by line number): 10, 20, 30, 40, 50, 20, 60, 70
3. 0 1 2 3 4 5 6 7 6 8 9

   When X*X equals 81 (X = 9), the condition is met and the computer will go to line 50.
4. 15 12 13 7

   NUMB has a new value put in it each time the comptuer is instructed to **READ** NUMB. That value is then checked with a condition and **PRINTed** if it is less than 20. The first condition **IF** NUMB = 9999 is placed there to stop the computer when it runs out of **DATA**. This is often referred to as using a sentinel or a flag and means that you have placed an unusual value at the end of your **DATA** statement as a way to tell the computer it has reached the end of the **DATA**.

5. **Border Bucks**

```
>10 CALL CLEAR
>20 PRINT " ** EXCHANGE AMOUNTS ** "::
>30 PRINT " 1) American to Canadian "::
>40 PRINT " 2) Canadian to American ":::
>50 INPUT " Number of your choice ? ":A
>60 PRINT :: "Current exchange rate ? ":RATE
>70 CALLL CLEAR
>80 IF A=1 THEN 150
>90 PRINT "CANADIAN FOR AMERICAN "::
>100 INPUT " $ Amount Canadian ? ": CAN
>110 ANS=INT(CAN*RATE*100)/100
>120 PRINT $;CAN; "Canadian ":: "      is "
>130 PRINT "$ ";ANS; "American ":::
>140 GOTO 200
>150 PRINT " AMERICAN FOR CANADIAN "::
>160 INPUT " $ Amount American ? ": USA
>170 ANS=INT(USA*(1/RATE)*100)/100
>180 PRINT "$ ";USA; "American ":: "      is "
>190 PRINT "$ ";ANS "Canadian ":::
>200 END
```

6. **Miles vs Mouth**

```
>10 CALL CLEAR
>20 PRINT " CALORIE CONTROL::
>30 PRINT " 1) Miles into Calories "::TAB(10); "or "::
>40 PRINT " 2) Calories into Miles ":::
>50 INPUT " Which one (1 or 2) ? ":A
>60 CALL CLEAR
>70 IF A=2 THEN 110
>80 INPUT "Number of miles ? ":MILES
>90 PRINT MILES; "miles burns ":MILES*85; "calories ":::
>100 GOTO 130
>110 INPUT "Number of calories ":CALS
>120 PRINT INT(CALS/85); "miles burns ";CALS;
      "calories ":::
>130 PRINT " *** Keep on jogging *** "
>140 END
```

7. **Travel Expense**

```
>10 CALL CLEAR
>20 PRINT "<<< TRIP PLANNER >>> ":::
>30 INPUT "Miles to go ? ":MILES
>40 INPUT "(P)ontiac or (M)ercury ":CAR$
```

```
>50 CALL CLEAR
>60 IF (CAR$ = "M ")+(CAR$ = "m ") THEN 100
>70 INPUT "Price of regular gas ":GASP
>80 PRINT :: "$ ";INT( (MILES/17)*GASP); "dollars for the
    trip "
>90 GOTO 180
>100 INPUT "(R)egular or (S)upreme ":TYPE$
>110 IF (TYPE$ = "S ")+TYPE$ = "s ") THEN 120
>120 INPUT "Price of reg. unleaded ":GASP
>130 PRINT :: "$ ",INT( (MILES/11)*GASP); "dollars for
     the trip "
>140 GOTO 180
>150 INPUT "Price of super unleaded ":GASP
>160 PRINT :: "$ ";INT( (MILES/13)*GASP); "dollars for
     the trip "
>170 PRINT ::: " Drive carefully "
>180 END
```

8. **More Junior**

```
>10 RANDOMIZE
>20 CALL CLEAR
>30 PRINT " GET SET "
>40 PRINT " GET READY "
>50 PRINT " GO "
>60 FOR PAUSE = 1 TO 100
>70 NEXT PAUSE
>80 WRONG = 0
>100        A = INT(RND*10+1)
>110        B = INT(RND*10+1)
>120        CORRECT = A*B
>130        PRINT "What is the answer to "::
>140        PRINT A; "times ";B;
>150      INPUT ANS
>160        IF ANS = CORRECT THEN 200
>170        WRONG = WRONG+1
>180        PRINT ::WRONG; "wrong answers so far "
>190      GOTO 210
>200        PRINT :: "Correct Jr., very good! "
>210      FOR PAUSE = 1 TO 100
>220      NEXT PAUSE
>230    CALL CLEAR
>240 NEXT PROBS
>250 IF WRONG/10<.90 THEN 20
```

```
>260 PRINT " * * * YOU DID IT. * * * "
>270 PRINT " !! JUNIOR !! "::
>280 PRINT "Your tootsie pops are "
>290 PRINT " in the closet. "
>300 END
```

Chapter 9

Arrays

Up to now a variable name could have only one value at a time. But if you wanted to store a large number of values in the computer at one time, each value needed a different variable name. This is awkward and time consuming. For example, if you wanted to find the average age of 50 different people you know, each person must supply you with their age. Now you have 50 different pieces of data. You also want the lightest and heaviest weights in the group. Until now, you could only do something like this:

```
10  READ WEIGHT1
20  READ WEIGHT2
30  READ WEIGHT3
  •
  •
  •
500 READ WEIGHT50
510 TOTAL1 = WEIGHT1+WEIGHT2+ . . . +
    WEIGHT10
520 TOTAL2 = WEIGHT11+WEIGHT12. . . +
    WEIGHT20
  •
  •
  •
560 AVERAGE = (TOTAL1+TOTAL2+ . . . +
    TOTAL5)/50
570 PRINT "AVERAGE =  ";AVERAGE
580 DATA 155,210,110,125,155,. . .
590 DATA . . . . . . . . . . . . . . . . . . . ,255
600 END
```

This will work but it is very cumbersome and inefficient. A better way is by using arrays to store the weight data. Figure 9.1 contrasts data storage using the two methods.

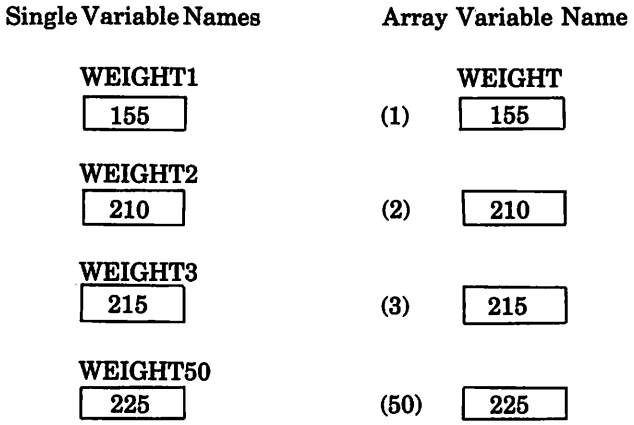


Figure 9.1 Data storage formats.

ONE-DIMENSIONAL ARRAYS

A BASIC language array is a set of data identified by a single variable name (WEIGHT in this example). To identify a particular value within the array you may reference its position within the array. In the example above, the third weight in the array WEIGHT is 215, or WEIGHT(3) = 215.

The position within the array is enclosed in parentheses following the variable name. This position indicator is called a *subscript*. The general form of an array is given below.

array name(subscript)

The subscript can be a constant or a variable. Using the information in Figure 9.1, what values would these variables generate?

WEIGHT(2) =	*Answer*: 210
WEIGHT(50) =	*Answer*: 225
WEIGHT(1) =	*Answer*: 155

Example 9.1: AGE Array

Let AGE = 12, 14, 80, 61, 15, 42, 36. Then:

AGE(1) = 12 (element in the first position of AGE array)

AGE(3) = 80 (element in the third position of AGE array)
AGE(2) = (element in the position of AGE array)
AGE(7) = (element in the position of AGE array)
AGE(4) = (element in the position of AGE array)

Now you have the capability to input, process and output large amounts of data under one name. You can even pinpoint a particular piece of data by specifying its physical location in the array. What you need is a subscript manipulator. But we already have one! Remember the **FOR–NEXT** loop! Remember the *control variable* in the **FOR** statement?

[line#] **FOR** [*control variable*] = [initial value] **TO** [final value] **STEP** [amt]

That *control variable* is the ideal subscript variable for these arrays. Watch how they work together.

Example 9.2: WEIGHT Array

```
10 REM EXAMPLE 9.2
20 REM ARRAY EXAMPLE TO READ AND PRINT
   FIVE WEIGHTS
30 REM INPUT THE FIVE WEIGHTS
40 FOR I=1 TO 5
50 READ WEIGHT(I)
60 NEXT I
70 REM PRINT OUT ALL THE WEIGHTS
80 FOR I=1 TO 5
90 PRINT WEIGHT(I)
100 NEXT I
110 DATA 155,210,215,165,195
120 CALL CLEAR
130 REM PRINT OUT THE SECOND WEIGHT
140 PRINT "WEIGHT #2 = ";WEIGHT(2)
150 END
```

Answers:

```
155
210
215
```

```
165
195
WEIGHT #2 = 210
```

The subscript **I** points to the box within the array WEIGHT in which an activity is to be performed and the subscript value comes from the **FOR** statement.

In Exercise 9.1 below, try to answer the questions following the exercise.

Exercise 9.1: Array Computations

Given the following program, write the final values for the variable shown at the end of the listing.

```
10 REM EXERCISE 9.1
20 REM PROGRAM USING ARRAYS IN
   COMPUTATIONS
30 FOR I=1 TO 6
40 READ HEIGHT(I)
50 NEXT I
60 SUM=HEIGHT(2)+HEIGHT(4)
70 AMOUNT=(HEIGHT(1)+HEIGHT(3))/2
80 DIFFER=HEIGHT(2)+HEIGHT(6)-HEIGHT(1)
90 DATA 68,72, 65,69,76,62
100 END
```

HEIGHT(2) =
HEIGHT(6) =
SUM =
AMOUNT =
DIFFER =

Answers:
HEIGHT(2) = 72
HEIGHT(6) = 62
SUM = 141
AMOUNT = 66.5
DIFFER = 66

Notes on Subscripts

The subscript's sole purpose is to point to an exact location or box within a list of array values. Subscripts (values within the parentheses) may be integer constants like 1, 40, 11, etc. They may also be variable names like I, J, K, KOUNT, POINT or SPOT. They may even be expressions like I+2, I*J or (A*B+C)/2. Expression results are rounded to the nearest whole number. That number becomes the pointer. Subscripts cannot be negative. From the list below, which subscripts are acceptable?

Exercise 9.2: Valid Subscripts

Variable Name	Acceptable Subscript
a. B(I)	(Yes)
b. B(3+4)	(Yes)
c. CDIC(I+5)	(Yes)
d. SUM(-4)	(No)
e. VALUE(K*J-3)	(Yes)
f. TEST(T/4-3*CC1+ TTACT)	(Yes)
g. DATA3(AMOUNT)	(Yes)
h. FINAL(-3-4)	(No)

Dimension Statements

Arrays allow for large volumes of data storage. Once the array size exceeds ten elements, a DIMENSION is needed. The form of this statement is:

[line#] **DIM** [array name]([integer$_1$,integer$_3$,integer$_3$])

The **DIM** statement allocates blocks of storage to the array name specified. Each position in the array requires one space in the array's dimension. **DIM** statements are normally placed at the beginning of the program and must precede the first occurrence or reference to the array name in the program. **DIM** statements act like reservations at a restaurant: "Reserve me a table for eight under the name Johnson." Space reserved in the **DIM** statement must equal or exceed the actual number of elements in the array. Allocating excessive space in the **DIM** statement is wasteful and may exhaust core storage space in larger, more complicated programs.

Arrays may have one dimension (rows only), two dimensions (rows and columns), or three dimensions (rows, columns and tiers or levels or planes). The **DIM** allows up to three subscripts. Unless specified in a **DIM** statement, all array values are automatically assigned a **DIM** value of 10. If your array will fit in 10 or less boxes, you do not need to dimension the name ahead of its use in the program. Note that subscripts in the array name in the **DIM** statement must be constants (no variables or expressions allowed). Also, more than one array name can be in a **DIM** statement. Each array name is separated by a comma.

Exercise 9.3: DIMENSION Statements

a. 10 **DIM** A(10)
b. 45 **DIM** ARRAY(10,20)
c. 36 **DIM** VALUES(40,30,20),IDATA(20,15)
d. 50 **DIM** TOTALS(20),SUMS(12),AMT(30,40)

How many total words of storage would be allocated to the following variables using the **DIM** statements shown above?

	Answers
1. A =	10
2. ARRAY =	200 (10*20)
3. VALUES =	24000 (40*30*20)
4. IDATA =	300 (20*15)
5. TOTALS =	20
6. SUMS =	12
7. AMT =	1200

Two-Dimensional Arrays

Two-dimensional arrays have rows and columns. The array VALUES(5,3) would look like Figure 9.2 after executing:

10 **DIM** VALUES(5,3)

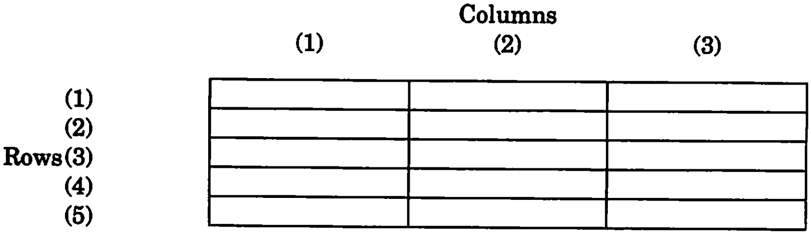


Figure 9.2 The layout of an array.

Now fill the array or matrix with data. You have asked five people (rows) their age (column one), weight (column two) and height (column 3). They reported the following:

	Age(Yrs)	Weight(#)	Height(″)
Person 1	32	165	66
Person 2	40	225	74
Person 3	16	140	64
Person 4	9	83	57
Person 5	74	136	63

Write a BASIC program to allocate space for this data and then read the data into that space. The program requires a **DIM** statement and a **READ** loop. You will fill the array by storing all the data about one person first, then go on to the second person, the third, and so on. Look at Example 9.3 to see how it will look.

Example 9.3: Two-Dimensional Array

```
10 REM EXAMPLE 9.3
20 REM THIS PROGRAM CREATES AND FILLS
   AN ARRAY
30 REM WITH DATA ABOUT FIVE PEOPLE.
40 REM FIRST, ALLOCATE ARRAY SPACE
50 DIM VALUES(5,3)
60 REM NOW READ IN PEOPLE DATA
70 FOR I=1 TO 5
80 REM NOW READ IN DATA ABOUT PEOPLE
90 FOR J=1 TO 3
100 READ VALUES(I,J)
110 NEXT J
120 NEXT I
130 REM HERE ARE THE DATA RECORDS FOR
    FIVE PEOPLE
140 DATA 32,165,66
150 DATA 40,225,74
160 DATA 16,140,64
170 DATA 9,83,57
180 DATA 74,136,63
190 END
```

The matrix VALUES is now loaded and looks like this:

	(1)	(2)	(3)
(1)	32	165	66
(2)	40	225	74
(3)	16	140	64
(4)	9	83	57
(5)	74	136	63

Figure 9.3 Matrix values.

You can identify any value in the array by specifying the person (row position) and characteristic (column position).

What would VALUES(3,3) contain? *Answer*: 64, third person's height

What would VALUES(4,2) contain? *Answer*: 83, fourth person's weight.

So far, you have only stored the data base. You have not processed it or printed it on the screen. How would you find the average age or average weight or average height of the participants?

One way would be to hold the column value constant and add all the row values in that column. The following statement would find the average age of the participants.

```
185 AVERAGE=(VALUES(1,1)+VALUES(2,1)+
        VALUES(3,1)+VALUES(4,1)+VALUES(5,1))/5
```

This could also be done for weight and height, but that would be very cumbersome if, for example, you had 100 participants and 30 characteristics. Then it would be difficult to use this approach. Could you use the control variable in the **FOR** statement to manipulate the row and column pointers? The answer is an emphatic "Yes"!

Examine this option. In statement number 185, look at what subscripts are moving and what subscripts are constant. The row values vary as you search through the people and the column values stay constant (locked onto the characteristics—age). The following program finds the average of each variable (age, weight, and height) and presents the results on the screen. Assume the data above has already been read in VALUES.

Example 9.4: Averaging Data in a Two-Dimensional Array

```
200 REM EXAMPLE 9.4
210 REM AVERAGING IN TWO-DIMENSIONAL
```

```
    ARRAYS
220 FOR J=1 TO 3
230 TOTAL=0
240 REM SEARCH THROUGH PEOPLE
250 FOR I=1 TO 5
260 TOTAL=TOTAL + VALUES(I,J)
265 NEXT I
270 AVERAGE=TOTAL/5
280 PRINT "AVERAGE FOR VARIABLE ";J;
    " = ";AVERAGE
290 NEXT J
300 END
```

Answers:

```
AVERAGE FOR VARIABLE 1 = 34.2
AVERAGE FOR VARIABLE 2 = 149.8
AVERAGE FOR VARIABLE 3 = 64.8
```

The outside loop (the J loop) moves slowly through the variables while the inside loop (the I loop) searches through all the people for the same variable.

When J=1: I=1, 2, 3, 4, 5
When J=2: I=1, 2, 3, 4, 5
When J=3: I=1, 2, 3, 4, 5

With this understanding and capability, large volumes of data can be processed quickly and efficiently. You have harnessed the two most powerful capabilities of the BASIC language—loops and arrays. Loops allow cycling through array data, and loop values serve to point to exact locations within the array itself, using subscripts as the pointers.

Example 9.5: Bank Survey

Imagine you have just completed a large survey for a bank on the attitudes of its customers. You surveyed 50 customers and asked them 20 questions about the bank's services.

Question 1: How would you rate the bank's advertising? (1=Excellent, 2=Very Good, 3=Average, 4=Below Average, 5=Poor)

Question 2: How would you rate the bank's tellers? (Same rating scale as above, 1=Excellent . . . 5=Poor)

•
•
•

Question 20: What's your overall impression of this bank? (Same rating scale as above, 1 = Excellent . . . 5 = Poor)

How many pieces of data would you have from the survey? *Answer*: 50 people * 20 questions = 1000 pieces of data

Write the program to find the average response for each of the 20 questions.

```
10 REM EXAMPLE 9.5
20 REM PROGRAM TO ANALYZE BANK SURVEY
   DATA
30 REM READ IN DATA ARRAY
40 DIM SURVEY(50,20)
50 FOR I=1 TO 50
60 FOR J=1 TO 20
70 READ SURVEY(I,J)
80 NEXT J
90 NEXT I
100 DATA 3,2,5,4,1,2,2,3,1,2,3,4,3,4,2,3,4,5,1,2
110 DATA 4,3,2,4,3,.....................4
 .
 .
 .
129 DATA 3,2,3,4,5,2,3,4,2,1,2,3,4,2,3,2,1,5,1,2
130 REM NOW COMPUTE AVERAGES
140 REM FOR EACH QUESTIONS
150 REM HOLD COLUMN POINTER
    (QUESTION #)
160 REM VARY ROW POINTER (PERSON #)
170 FOR J=1 TO 20
180 TOTAL=0
190 FOR I=1 TO 50
200 TOTAL=TOTAL+SURVEY(I,J)
210 NEXT I
220 PRINT "QUES # ";J; "AVERAGE =  ";TOTAL/50
230 NEXT J
240 END
```

In example 9.5 above, how many times are the following statement numbers executed when the program is run?

	Answers
a. #50:	50
b. #70:	1000 (50*20)

c. #180: 20
d. #200: 1000 (50*20)
e. #230: 20

Three-Dimensional Arrays

You can add a third layer to the matrix in TI BASIC by adding another dimension. Now you have rows, columns, and planes. The storage would look like a Rubics cube.

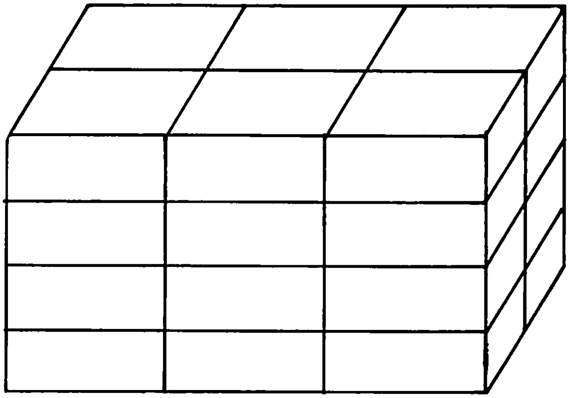


Figure 9.4 A three-dimensional array.

What if you were keeping statistics about NBA basketball teams or NFL football teams? Suppose you need to keep statistics on the starting five for each of four NBA teams. For each player you need height, weight, total playing time this season, and total points scored. The data matrix would be three-dimensional. The four teams are: Los Angeles, Philadelphia, Milwaukee, and San Antonio. One team's statistics would be kept in a matrix form similar to that shown in Figure 9.4.

Imagine four layers or planes in the above matrix, one layer for each team, and you have a three-dimensional matrix. The rows are the positions (guards, forwards, and center); the columns equal the statistics (height, weight, etc.); and the planes would be the four teams.

Example 9.6: Basketball Statistics

Write a program to read the data from **DATA** statements and store them in a three-dimensional matrix. Then have your program find the average height of the four centers.

Hint: Where are the heights of the four centers? What box locations are they?

Answers:
(3,1,1)
(3,1,2)
(3,1,3)
(3,1,4)

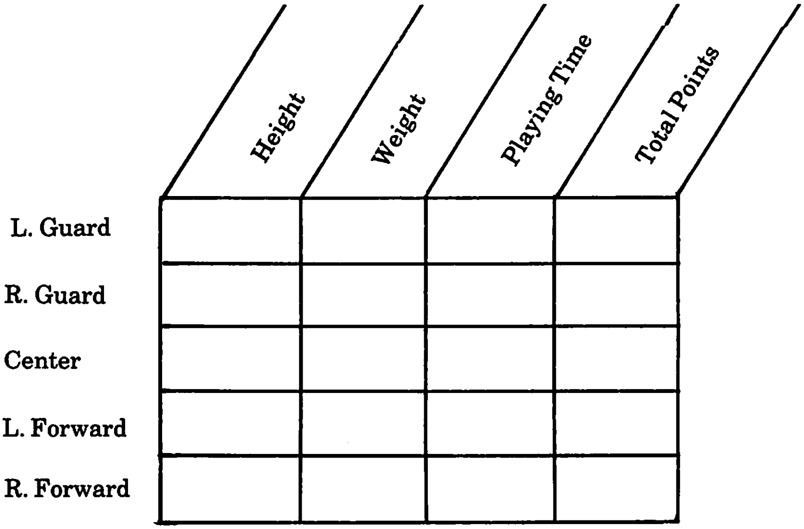


Figure 9.5 NBA team statistics matrix.

Note: which of these subscripts are constant and which vary?

Solution:

```
10 REM EXAMPLE 9.6
20 REM THIS PROGAM USES THREE-DIMENSIONAL ARRAYS
30 REM TO PROCESS NBA TEAM STATISTICS
40 REM FIRST, READ IN TEAM STATISTICS
50 DIM STATS(5,4,4)
60 FOR K=1 TO 4
70 FOR I=1 TO 5
80 FOR J=1 TO 4
90 READ STATS(I,J,K)
100 NEXT J
```

```
110 NEXT I
120 NEXT K
130 DATA 76,210,2460,1640
140 DATA 80,218,2550,1710
 •
 •
 •
920 DATA 82,235,2145,1510
930 REM NOW FIND AVERAGE HEIGHT
940 REM OF THE FOUR CENTERS
950 REM MUST LOCK ON ROW AND COLUMN
    POSITIONS
960 REM VARY THE PLANE POSITION
970 TOTALHT=0
980 FOR K=1 TO 4
990 TOTALHT=TOTALHT+STATS(3,1,K)
1000 NEXT K
1010 PRINT "AVER. CENTER HEIGHT = ";
     TOTALHT/4
1020 END
```

Exercise 9.4: NBA Stats

a. How many times is statement number 90 executed?
Answer: 80

b. How many pieces of data are stored in STATS matrix?
Answer: 80

c. What do columns stand for in the STATS matrix?
Answer: player characteristics

d. Write a series of BASIC instructions that would modify the program in Example 9.6 to find the average weight of all left guards. (Hint: Which subscripts would stay constant and which would vary?)
Answer:

```
970 TOTALWT=0
990 TOTALWT=TOTALWT+STATS(1,2,K)
1010 PRINT "AVE. WEIGHT OF L. GUARDS= ";
     TOTALWT/4
```

Exercise 9.5: Advanced NBA Statistics (Optional review)

If you really want to get fancy, see if you can compute the average points scored by the starting five players of each team.

Assume there are 82 games played during the regular season and the matrix contains the final total points scored during the 82-game season. How would the program look? (No answer is provided.)

SUMMARY

Arrays let you store large amounts of data under one variable name. To access a particular value within the array, you need the coordinates of that value within the array, referred to as the row position in a one-dimensional array, or the row and column position in the two-dimensional array or a row, column, and plane position in the three-dimensional array. Arrays of more than 10 elements must be dimensioned in the program before the first reference to the array name. The **DIM** statement defines the name of the array and allocates a block of storage in core for the array's values.

When arrays are combined with **FOR–NEXT** loops and the **FOR** statement's *control variable*, you have a very powerful computing tool. The control variable becomes a pointer to a specific location or box in the array. Array values can be sorted, summed, tested against another value, or redefined to a new value. If you find yourself naming variables in your program, i.e., AMOUNT1, AMOUNT2, AMOUNT3, and so on, you should be using arrays instead of single-valued variable names. Most powerful BASIC programs make extensive use of arrays. Try to incorporate them into your future programming applications.

REVIEW ACTIVITIES

1. Which of the following are valid array names?

	Answers
Name	**Valid?**
a. SUM(4)	(Yes)
b. SUME(4)	(Yes)
c. AMOUNT(I)	(Yes)
d. AMOUNT(I+J)	(Yes)
e. TOTAL(I*J/K,L)	(Yes)
f. TOTAL(–6)	(No)

2. Which of the following are valid **DIM** statements?

	Answers
Statement	**Valid?**
a. **DIM** A(4)	(No)
b. **DIM** B4(6)	(No)

c. 50 **DIM** A(J) (No)
d. 60 **DIM** A(4,3),B(20) (Yes)
e. 10 **DIM** SUM(30*10) (No)

3. In the following gasoline bill program, what values would be output for the variables listed?

```
5 REM REVIEW ACTIVITY 9.3: GASOLINE BILL
10 DIM DOLLAR(7)
20 FOR I=1 TO 7
30 READ DOLLAR(I)
40 NEXT I
50 DATA 11,15,6,7,3.5,12,14
60 C=DOLLAR(3)+DOLLAR(5)
70 D=C/2
80 E=C+D+DOLLAR(2)
90 F=C-D+DOLLAR(6)
100 PRINT C,D,E,F
```

	Answers
C=	9.5
D=	4.75
E=	29.25
F=	16.75

4. List the results of the output from this calorie counter program.

```
10 REM REVIEW ACTIVITY 9.4: CALORIE
   COUNTER
20 DIM CALORIE(7)
30 TOTAL=0
40 WEIGHT=160
50 FOR DAY=1 TO 7
60 READ CALORIE(DAY)
70 TOTAL=TOTAL+CALORIE(DAY)
80 NEXT DAY
90 PRINT "MONDAY = ";CALORIE(1)
100 PRINT "TOTAL = ";TOTAL
110 WEEKEND=CALORIE(6)+CALORIE(7)
120 AVECALORIE=TOTAL/7
130 POUND=TOTAL/WEIGHT
140 DATA 1580,2140,1960,2100,1800,3500,3200
150 PRINT "AVE
    CAL= ";AVECALORIE,"POUND "; POUND
160 PRINT "WEEKEND ";WEEKEND
```

	Answers
TOTAL =	16280
WEEKEND =	6700
AVECALORIE =	2325.714286
POUND =	101.75

5. Given the following list of daily meal expenses for a week, write a BASIC program to find the total and average amounts spent.

	Amount
Monday	10.40
Tuesday	14.10
Wednesday	7.85
Thursday	7.50
Friday	11.50
Saturday	18.00
Sunday	12.75

```
10 REM REVIEW ACTIVITY 9.5: EXPENSE
   ACCOUNT
20 DIM MEALS(7)
25 TOTAL=0
30 FOR I=1 TO 7
40 READ MEALS(I)
45 TOTAL=TOTAL+MEALS(I)
50 NEXT I
60 DATA 10.40,14.10,7.85,7.50,11.50,18.00,12.75
70 AVERAGE=TOTAL/7
80 PRINT "TOTAL ", "AVERAGE "
90 PRINT TOTAL,AVERAGE
100 END
```

Answers: TOTAL–82.10; AVERAGE–11.7285

6. Write a program to compute, load, and output the multiplication tables from 1 to 10. Store the results in a two-dimensional matrix. (Hint: **DIM** TABLE(10,10))

Example

```
        1 2 3 ...
      -------------
    1 I 1 I 2 I 3 I ...
      -------------
    2 I 2 I 4 I 6 I ...
      -------------
```

```
3  I 3 I 6 I 9 I ...
   ---------------
```

Answer:

```
10 REM REVIEW ACTIVITY 9.6:
   MULTIPLICATION TABLE DRILL
20 DIM TABLE(10,10)
30 FOR I=1 TO 10
40 FOR J=1 TO 10
50 TABLE(I,J)=I*J
60 PRINT I; "* ";J; " =  ";TABLE(I,J)
70 NEXT J
80 NEXT I
90 END
```

7. To extend Review Activity number 6 above, have the computer drill students in their math skills. This exercise is an example of a multiplication drill. It is called COMPUTER FLASH CARDS. With minor modifications this program can add, subtract, and divide. Speed of response can also be varied.
   Write a BASIC program to drill students in multiplication. Number values will be limited to 1-10. Then specify the length of delay in seconds before the answer is shown on the screen. Say the answer out loud before it appears on the TV screen. Use parts of Review Activity number 6 in this program.

Answer:

```
10 REM REVIEW ACTIVITY 9.7: COMPUTER
   FLASH CARDS
20 DIM TABLE(10,10)
30 FOR I=1 TO 10
40 FOR J=1 TO 10
50 TABLE(I,J)=I*J
60 NEXT J
70 NEXT I
80 CALL CLEAR
90 PRINT "WELCOME TO: "
100 PRINT "COMPUTER FLASH CARDS!! "
110 PRINT
120 PRINT
130 PRINT "# OF SECONDS ANSWER DELAY "
140 INPUT N
145 PRINT "BEGIN FIVE DRILL EXERCISE "
```

```
150 FOR K=1 TO 5
160 CALL CLEAR
170 PRINT "INPUT 1ST NUMBER(1-10) "
180 INPUT I
190 PRINT "INPUT 2ND NUMBER(1-10) "
200 INPUT J
210 PRINT "    ";I; "X ";J; " = ";
220 FOR DELAY=1 TO 400*N
230 NEXT DELAY
240 ANS=TABLE(I,J)
250 PRINT ANS
260 FOR DELAY=1 TO 2000
270 NEXT DELAY
280 NEXT K
290 END
```

Think how this program can be modified to:

a. divide
b. subtract
c. add
d. allow larger input numbers

8. Write a program to keep a check register for up to 50 checks per month. The register would look like the one shown below. Use a two-dimensional array for storage. Print a check listing. For each check in the register, record check number, day of month, amount, and category of expenditure code (1 = food, 2 = utilities, 3 = auto expense, etc.).

Hint: **DIM** REGISTER(50,4)

Possible Format:

```
  Check      Day of                 Category
  Number     Month       Amount       Code
   ====       ===        ======       ====
    1          3          10.85         3
    2          4          26.45         6
    .          .            .           .
    .          .            .           .
    .          .            .           .
```

Answer:

```
10 REM REVIEW ACTIVITY 9.8: CHECK
   REGISTER
20 DIM REGISTER(50,4)
```

```
30 REM N = NUMBER OF CHECKS
40 PRINT "INPUT NUMBER OF CHECKS IN
   CHECK REGISTER "
45 INPUT N
50 PRINT "CHECK# DAY AMOUNT CODE "
55 PRINT "====  ===  ======  ===="
60 FOR I = 1 TO N
70 FOR J = 1 TO 4
80 READ REGISTER(I,J)
90 NEXT J
100 FOR K = 1 TO 4
110 PRINT REGISTER(I,K); "    ";
120 NEXT K
130 PRINT
140 NEXT I
150 DATA 1,1,30.00,10
160 DATA 2,4,18.40,4
 •
 •
 •
500 END
```

Think of what information could be gathered from REGISTER(50,4).

a. dollars spent in each category code;
b. total dollars spent per month;
c. total dollars spent by week;
d. dollars spent in two or more category codes;
e. etc.

9. Write a program to budget next year's monthly expenses for food, shelter, utilities, auto, and recreation. Total each category and compute a grand total for the year. Your budget layout might look like the one below. Load this budget into a two-dimensional matrix called BUDGET. Print January's budget.

Item	Jan (1)	Feb (2)	Mar (3)	Dec (12)	Total (13)
Food(1)	______	______	______	______	______
Shelter(2)	______	______	______	______	______
Utilities(3)	______	______	______	______	______
Auto(4)	______	______	______	______	______
Recreation(5)	______	______	______	______	______
			Grand Total		______

Answer:

```
10 REM REVIEW ACTIVITY 9.9: HOME BUDGET
20 DIM BUDGET(5,13)
30 FOR I=1 TO 5
40 TOTCATEGORY=0
50 GRANDTOT=0
60 FOR J=1 TO 12
70 READ BUDGET(I,J)
80 TOTCATEGORY=TOTCATEGORY+
   BUDGET(I,J)
90 NEXT J
100 BUDGET(I,13)=TOTCATEGORY
110 NEXT I
120 FOR I=1 TO 5
130 GRANDTOT=GRANDTOT+BUDGET(I,13)
140 NEXT I
150 DATA 100,120,120,130,130,140,140,140,140,145,
    145,165
 .
 .
 .
200 PRINT "CATEGORY ", "AMOUNT "
210 FOR I=1 TO 5
220 PRINT I,BUDGET(I,1)
230 NEXT I
240 END
```

The program computes category totals for the entire year (contained in BUDGET(I,13)). How would you compute and print monthly totals of all the categories?

10. Time is your most important asset. Write a time management program. Estimated how many hours per week you spend in the categories below. The total must equal 168 hours (24 * 7).

Activity	Activity	Hours Per Week	Percent of Time
1	Sleeping	____________	
2	Work	____________	
3	Eating	____________	
4	TV	____________	
5	Commuting	____________	
6	Reading	____________	
7	Exercise	____________	

8	Recreation	____________
9	Visiting	____________
10	Other	____________
	Total	168

Write a program to enter these data in a matrix or array called TIME. Compute the percentage of time you spend in each activity and place that value in column two of the TIME matrix. Print the time spent and percentage of time for each category. How do they look?

Answer:

```
10 REM REVIEW ACTIVITY 9.10: TIME
   MANAGEMENT
20 DIM TIME(10,2)
30 TOTAL=0
40 FOR I=1 TO 10
50 READ TIME(I,1)
60 TOTAL=TOTAL+TIME(I,1)
70 NEXT I
80 REM COMPUTE % TIMES
90 FOR I=1 TO 10
100 TIME(I,2)=TIME(I,1)*100/TOTAL
110 NEXT I
120 DATA 50,46,10,15,4,10,11,9,5,8
130 PRINT "ACTIVITY "; "TIME "; "% TIME "
140 FOR I=1 TO 10
150 PRINT I; "    ";TIME(I,1),TIME(I,2)
160 NEXT I
170 END
RUN
```

```
    Activity        Time        % Time
       1             50        29.76190476
       2             46        27.38095238
       3             10        5.952380952
       4             15        8.928571429
       5              4        2.380952381
       6             10        5.952380952
       7             11        6.547619048
       8              9        5.357142857
       9              5        2.976190476
      10              8        4.761904762
```

Keep an actual time log for one week! See how it compares to your estimates above. Revise your program **DATA** statements and recompute the percentages. Do they differ greatly from your estimates?

11. Home inventories are important, yet few people get organized enough to take one. Let the computer be your incentive. Combine learning and value. Write a program using a three-dimensional array to collect and analyze your household belongings. List the major items you own by room. For each item, classify it according to some category codes (see below) and estimate its value.

Category Code	Category
1	Jewelry
2	Appliances
3	Furniture
4	Art
5	Coins
6	Silverware
•	•
•	•
•	•

In this example, rows stand for a particular item, columns represent characteristics of the item (category and value) and tiers or planes would be rooms (living room, bedroom, kitchen, etc.).

Write a TI BASIC program to:

1. Read the data from **DATA** statements into a three-dimensional array called INVENTORY.
2. Determine the total inventory value of each room.
3. Determine the value of all jewelry items.
4. Determine the number of jewelry items in the master bedroom.

(Optional—no solution supplied.)

Chapter 10

Introduction to Sound and Color-Graphics *or*: Fun with Your TI

Sound and color-graphics capabilities are two of the strongest features of your TI-99/4A. If you have ever played one of TI's arcade games (i.e., *Parsec*, *TI Invaders*, or *Munchman*), you have seen sound and graphics at work. Have you ever wondered how they developed those screen colors, character movements, and sound? The answer is graphics.

Graphics and sound provide excellent opportunities to present your program output in a powerful and interesting way. The program communicates better with its users if the results are captivating and enjoyable. Watch little children use the educational command modules in arithmetic or grammar. Learning becomes fun and rewarding. You need to know how to develop graphic output and use it in your programs.

Graphics comes from the word graph. Imagine the output screen as a sheet of graph paper. The graph has 24 lines or rows (horizontal to the screen) and 32 columns (vertical to the screen). You can then display any character you want, any place you want, simply by identifying the character and a specific row and column position on the screen. At the intersection of those two positions (called coordinates) on the screen, the character will appear. The screen becomes a giant scratch pad. With graphics you can draw a picture, a graph, a chess or checker board, even animated characters like Walt Disney's Mickey Mouse or Pluto. With sound you can make these pictures move with musical backgrounds.

TI has built in a number of graphic and sound capabilities in its language. All you do is request one of these subprograms, supply the command with a set of instructions or parameters, and the computer does the rest. A series of these commands can make interesting applications.

The commands include **CALL SOUND**, **CALL CLEAR**, **CALL SCREEN**, **CALL COLOR**, **CALL HCHAR**, and **CALL VCHAR**. All of these commands work in the immediate mode as well as in program statements.

CALL SOUND

You can play your TI-99/4A like a musical instrument. It will generate notes and tones over a range of several octaves. You can specify how long the tone is played (called duration), what tone is played (called frequency), and how loud the tone is played (called volume).

The format, or syntax, of the **CALL SOUND** command is:

```
[line #] CALL SOUND(duration,frequency,volume,
                    [frequency,volume])
```

where the three values stand for:

Value	**Allowable Range**
duration	1 to 4250, inclusive and –1 to –4250, inclusive (1000 = 1 second)
frequency (tone or note)	110 to 44733, inclusive or –1 to –8, inclusive (noise)
volume	0(loudest) to 30(quietest), inclusive

Duration

The duration of a note can range from one millisecond (1/1000 of one second) to 4250 milliseconds or 4¼ seconds. The duration is specified first in the **CALL SOUND** command. If the duration value is positive, the computer continues to execute the program, even if another **CALL SOUND** command is encountered. If the duration value is negative and a second **CALL SOUND** command is executed, the first command will terminate immediately and let the second **CALL SOUND** command begin.

Frequency

Frequency may either be a tone or a type of noise. Tones are measured in cycles per second or Hertz (named after Heinrich Hertz, a German scientist). One cycle per second = 1 Hertz = 1 Hz. Frequency values may range from 110 Hz, which is equal to an A note below low C on the piano keyboard, to a high of 44733 Hz, a tone well above human hearing limits. Some common notes and their frequencies are shown in Figure 10.2.

Your TI-99/4A can make noises, too. Sounds like cars idling, racing, and crashing add to the reality of games. Explosions and other types of sounds are also possible. Frequency values between the range of −1 to −8 generate those noises. The duration and volume values work with these sounds. Try these commands on your computer and you will experience these sounds that are hard to describe.

Up to three frequencies and one noise can be played at one time. This allows you to form chords and sound-action combinations. Sound is an excellent positive reinforcement for children when they have successfully completed a computer game or exercise.

Volume

Volume simply sets the loudness control on the sounds. The larger the number, the softer the sound. In this chapter, we have included a hearing test to demonstrate the volume control. Remember, the volume control on your monitor or TV set will determine the beginning volume level of the **CALL SOUND** command. Raising the TV volume raises the starting volume range for the sound control command.

Try some sounds on the TI-99/4A.

Example 10.1: The Low A Note

Write an instruction to play note A below low C on the piano keyboard. Play the note for two seconds. What's the command?

Answer: 10 CALL SOUND (2000,111,15)

Example 10.2: Can You Hear This?

Write a program to test your hearing of sounds. Play low C below middle C from softest (30) to loudest (0) in increments of two. Print out the loudness level and mark the ones you can hear. Also try middle C and C above middle C. On the chart below record the results.

Note	Sound Level
Low C	
Middle C	
C Above Middle C	

Solution

```
10 REM EXAMPLE 10.2: CAN YOU HEAR THIS
20 NOTE(1)=131
30 NOTE(2)=262
40 NOTE(3)=523
50 FOR I=1 TO 3
60 FOR J=30 TO 0 STEP -2
70 CALL SOUND(500,NOTE(I),J)
80 PRINT "HZ= ";NOTE(I)
90 PRINT "LOUDNESS LEVEL = ";J
100 PRINT "CAN YOU HERE THIS? YES/NO-
    MARK ON CHART "
110 NEXT J
120 NEXT I
130 END
RUN
```

Look at the piano keyboard below. Note the key locations and the Hz values for selected keys.

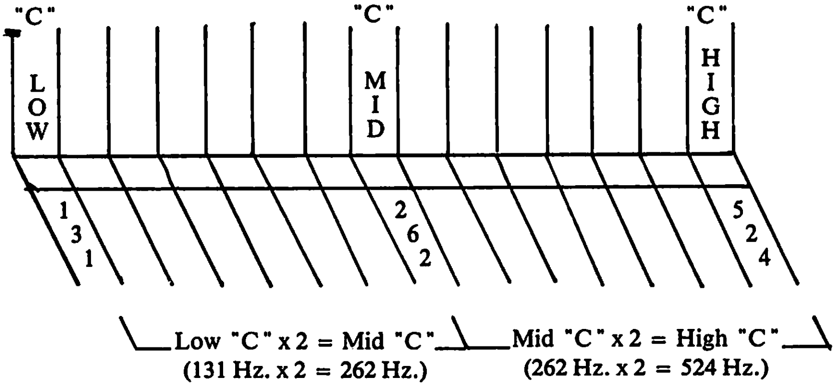


Figure 10.1 A piano keyboard.

Example 10.3: Playing As and Cs

Write a program in TI BASIC to play three A notes in three octaves beginning with low A. Then play three C notes beginning with low C. Play each note for one second, loudly.

Solution

```
10 REM EXAMPLE 10.3: PLAYING A'S AND C'S
20 REM PLAY THREE "A " NOTES, THEN THREE
   "C " NOTES
30 F=55
40 FOR I=1 TO 3
50 F=F*2
60 CALL SOUND(1000,F,1)
70 NEXT I
80 IF F< >440 THEN 110
90 F=65
100 GOTO 40
110 END
RUN
```

Hear the notes? Sing along! "Play it again, TI."

Example 10.4: Playing Chords

The TI can play up to three notes at one time. This allows you to play songs. Write a program to play a C-major chord (middle C, E and G notes). Hold the chord for two seconds. Play it loudly, middle range and softly. Note how delay loops are used to break between durations of the chord.

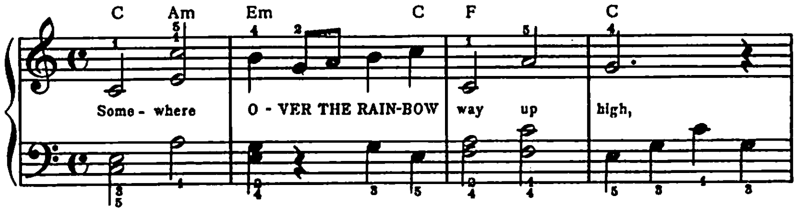
Solution

```
10 REM EXAMPLE 10.4: CHORDS
20 REM THIS PROGRAM PLAYS A C-MAJOR
   CHORD
30 REM NOTE HOW A LOOP (STATEMENTS 80-90)
   IS USED TO BREAK
35 REM DURATION OF THE CHORD!
40 FOR I=1 TO 3
50 C=262
60 E=330
70 G=392
80 CALL SOUND(2000,C,5*I,E,5*I,G,5*I)
90 FOR K=1 TO 1600
100 NEXT K
110 NEXT I
120 END
RUN
```

Hear the chord? Can you change its duration? (Hint: Change 2000 value in statement number 80.)

Example 10.5: Playing Songs (Optional, unless you read music.)

Write a program to play this song. Do you recognize the tune?



Solution

```
10 REM EXAMPLE 10.5: PLAYING SONGS
20 CALL SOUND(2000,131,15,165,15,262,15)
30 CALL SOUND(2000,220,15,330,15,523,15)
40 CALL SOUND(1000,165,15,196,15,494,15)
50 CALL SOUND(500,392,15)
60 CALL SOUND(500,440,15)
70 CALL SOUND(1000,494,15,196,15)
80 CALL SOUND(1000,523,15,165,15)
90 CALL SOUND(2000,262,15,175,15,220,15)
100 CALL SOUND(2000,440,15,262,15,175,15)
110 CALL SOUND(2000,392,15,165,15)
120 END
RUN
```

Note: To repeat this song, type:

```
115 GOTO 20
```

Noise

The TI-99/4A can produce noise as well as tones. The actual sound of the noises is hard to describe. The best way to experience them is to play with some programs that generate noise. Use the **CALL SOUND** command and instead of inserting Hertz values for the tones, insert a number from −1 through −8. These numbers produce the various noises. The following program produces all the noises. In the table below, write your description of the noise that each value generates.

Noise #	Sounds Like?
–1	
–2	
–3	
–4	
–5	
–6	
–7	
–8	

Example 10.6: Noises

Write a program to play each noise in the TI-99/4A.

Solution

```
10 REM EXAMPLE 10.6: NOISES
20 FOR I=8 TO 1 STEP -1
30 LEVEL=I-9
40 PRINT "NOISE # ";LEVEL; " SOUNDS LIKE
   THIS "
50 CALL SOUND(2000,LEVEL,1)
60 PRINT "WHAT DOES IT SOUND LIKE? "
70 FOR DELAY=1 TO 2000
80 NEXT DELAY
90 NEXT I
100 END
RUN
```

Exercise 10.1: Drag Race

Write a program using noises to simulate the sound of a dragster on the drag strip. How would it sound? Use the results from Example 10.6 to pick your sounds. Rerun Example 10.6 if necessary.

Exercise 10.2: Cannon Fire

Write a program that sounds like a cannon shooting a shell into the air. Did it explode?

CALL CLEAR

The **CALL CLEAR** command inserts blank characters in all 768 spaces (24 rows × 32 columns) on the TV screen and thus erases any instructions or output that was previously on the screen. **CALL CLEAR** gives you a fresh screen for your program's listing or output. This statement is often one of the first statements in an application. Remember, all the following statements can be used in immediate mode or with line numbers in a program.

Examples

[In immediate mode]

```
>PRINT "HOW ARE YOU? "
HOW ARE YOU
```

```
>CALL CLEAR
```

[Screen goes blank]

[In program mode (note line numbers and **RUN** command)]

```
10 PRINT "THIS IS A TEST "
20 PRINT "WILL IT CLEAR THIS MESSAGE? "
30 CALL CLEAR
40 GOTO 10
RUN
```

(Press CLEAR Function keys to halt execution.)

Make up some of your own examples. Can you develop a flashing message on the screen? (Hint: Use **FOR–NEXT** loops to delay execution of the **CALL CLEAR** instruction.)

Can you develop a digital clock for your TI-99/4A? (Hint: Time your clock with the **FOR–NEXT** loop.)

CALL SCREEN

Unless instructed otherwise, the screen color on your monitor is blue and the characters are black. When you execute a program, the screen changes to light green. But there are 16 screen colors you can use. Figure 10.2 lists the screen colors and their codes.

Color	Color Code
Transparent	1
Black	2
Medium Green	3
Light Green	4
Dark Blue	5

Color	Color Code
Light Blue	6
Dark Red	7
Cyan	8
Medium Red	9
Light Red	10
Dark Yellow	11
Light Yellow	12
Dark Green	13
Magenta	14
Gray	15
White	16

Figure 10.2 TI screen colors.

The instruction to change screen color is:

CALL SCREEN(color code)

When you execute the **CALL SCREEN**, the screen color immediately changes to the color specified by the color code. Any characters on the screen remain unchanged–only the surroundings change. Try this example to become familiar with the **CALL SCREEN** command.

Example 10.7: Screen Colors

```
10 REM EXAMPLE 10.7
20 REM CALL SCREEN COLORS
30 CALL CLEAR
40 PRINT "HOW DOES THIS MESSAGE LOOK? ";I
50 FOR I=1 TO 16
60 CALL SCREEN(I)
70 FOR DELAY=1 TO 500
80 NEXT DELAY
90 NEXT I
100 RUN
```

This will show you all 16 screen color options. Did you see 16 messages? Why or why not?

CALL COLOR

A companion instruction to **CALL SCREEN** is **CALL COLOR**. Together they become the artist's palette and paintbrush. Previously you learned that there are 768 separate spaces or addresses on the screen (24 rows × 32 columns) that you can isolate and activate. Once you get to one of these boxes or locations and want to print a character, the character consumes only a small portion of the box. This is referred to as the foreground of the box and is the actual characer itself. Surrounding the character, but still within the confines of the box is the background. This is shown in Figure 10.2 below.

Through the **CALL COLOR** command you can specify foreground and background colors. The command format is:

CALL COLOR(character-set-number,foreground-color-code,background-color-code)

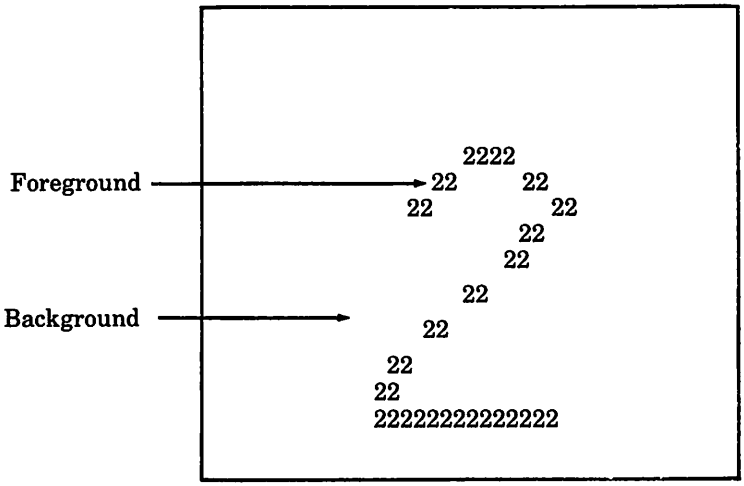


Figure 10.2a Foreground and background colors.

The Color code numbers range from 1 to 16 and are the same as those specified under the **CALL SCREEN** section above (i.e., 1 = transparent, 2 = black, etc.). These three values within the parentheses can be specified as constants, variables or expressions.

CALL HCHAR

The **CALL HCHAR** subprogram positions a row character anywhere on the screen and optionally reproduces that character a specified number of times in that row. The **HCHAR** command format is:

CALL HCHAR(row-number,column-number,
character code[,number of repetitions])

The intersection of the row and column numbers locates the first character. Character code specifies the character to be printed there and the optional repetition value tells how many times to print that character. With no repetition value, only one character is printed. As with the other graphic commands, values in the **CALL CHAR** list may be constants, variable names or expressions. The maximum row number is 24 and the maximum column number is 32.

Character codes correspond to letters, numbers, and special characters. A directory of these codes and their characters is shown in Appendix II.

Example 10.8: Printing a B.

Write a program segment to print a capital B in row 10, column 14 of your screen.

```
10 REM EXAMPLE 10.8
15 CALL CLEAR
20 CALL HCHAR(10,14,66)
30 END
RUN
```

What happens when you run this? What is the character code for B?

Example 10.9: Printing Bs

Write a program segment to print 10 Bs starting in row 8 and column 6.

```
10 REM EXAMPLE 10.9
15 CALL CLEAR
20 CALL HCHAR(8,6,66,10)
30 END
RUN
```

What happens?

Example 10.10: Repeating Characters on the Screen

Write a program segment to print three $s, in the configuration shown below, across the screen beginning in the upper left-hand corner and moving down to the lower right-hand corner.

```
$$$
 $$$
  $$$
   $$$
    ...
     ...
      ...
```

Figure 10.3 Repeating characters on the screen.

```
10 REM EXAMPLE 10.10: A DIAGONAL OF $$$'s

   15 CALL CLEAR
   20 FOR I=1 TO 24
   30 CALL HCHAR(I,I,36,3)
   40 NEXT I
   50 END
   RUN
```

Does the output look like the picture above?

Example 10.11: Question Marks

Write a program segment to fill the screen (24 rows × 32 columns) with ?. Then clear the screen.

```
5 REM EXAMPLE 10.11: QUESTION MARKS
10 FOR I=1 TO 24
20 CALL HCHAR(I,1,63,32)
30 NEXT I
40 CALL CLEAR
50 END
RUN
```

```
????????????????????????????????
????????????????????????????????
????????????????????????????????
               .
               .
               .
????????????????????????????????
????????????????????????????????
```

Figure 10.4 A full screen.

Example 10.12: A Border Test

For a variety of reasons, columns 1, 2, 31, and 32 may not appear on your screen. Thus, column displays should be limited to columns 3-30. Write a program to fill these two vertical edges with the "at" sign (@). Can you see them on your screen?

```
10 REM EXAMPLE 10.12: A BORDER TEST
15 CALL CLEAR
20 FOR I=1 TO 24
30 FOR J=1 TO 2
40 CALL HCHAR(I,J,64)
50 CALL HCHAR(I,30+J,64)
60 NEXT J
70 NEXT I
80 END
RUN
```

Another way to program this application is shown below. What are the differences between this program and Example 10.12?

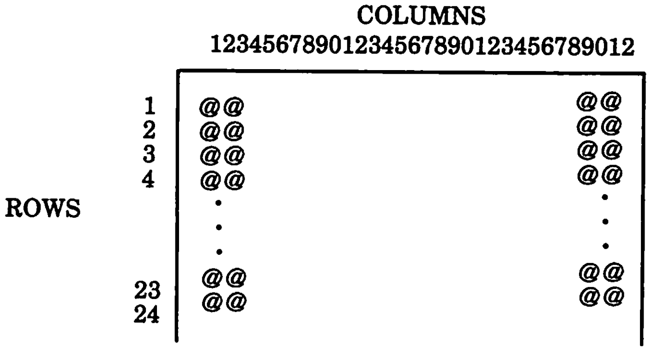


Figure 10.5 A screen with borders.

```
10 REM EXAMPLE 10.12A: A BORDER TEST (AN
   ALTERNATIVE WAY)
15 CALL CLEAR
20 FOR I=1 TO 24
30 CALL HCHAR(I,1,64,2)
40 CALL HCHAR(I,31,64,2)
50 NEXT I
60 END
RUN
```

Can you think of another way?

The **HCHAR** command allows you to pinpoint a row and column position on the screen, then fill the row from that position with one or more duplicates of the specified character. An alternative to the **HCHAR** command is the **VCHAR** command.

CALL VCHAR

This command works much like the **HCHAR** command, except that the characters fill the screen vertically from the starting point, instead of horizontally. If you run to the bottom of the screen, the string of characters will continue at the top of the next column to the right. If you run off the right side of the display, the character string will wrap-around to column 1 on the left side of the screen.

The general form of the **CALL VCHAR** command is:

CALL VCHAR(row number,column number,
character code[,number of repetitions])

Example 10.13: A Screen Full of Equals

Write a program segment to fill the screen with equal signs (=). Use the **VCHAR** command. This can be done a number of ways.

```
10 REM EXAMPLE 10.13: A SCREEN FULL OF
   EQUALS
15 CALL CLEAR
20 CALL VCHAR(1,1,61,768)
30 END
RUN
```

This program would fill the screen from position (1,1) through (24,32) with = . Why 768? Remember the screen has 24 rows and 32 columns. 24 × 32 = 768 spaces on the screen. Fill them all with equal signs.

Example 10.14: Using the CALL VCHAR Command

How many of the **CALL HCHAR** examples shown above can you convert to the **CALL VCHAR** command? Let's do one; Example 10.12, the border test. Remember, this one asks you to fill columns 1, 2, 31, and 32 with @ signs. It is actually easier to do this example using the **CALL VCHAR** command than it is with the **CALL HCHAR** command.

```
10 REM EXAMPLE 10.14: USING THE CALL
   VCHAR COMMAND
20 CALL VCHAR(1,1,64,48)
30 CALL VCHAR(1,31,64,48)
40 END
RUN
```

This example shows the wrap-around feature of the **CALL VCHAR** command. When the string of output characters reaches the bottom of the screen, the output continues on the top of the next column. The 48 in the command fills two complete columns of output with the @ sign.

Now you have all the building block commands to do some fun activities in TI sound and color graphics. You have:

CALL CLEAR	clears the screen of all output (your eraser command)
CALL SOUND	for musical backgrounds
CALL SCREEN	for backdrop colors
CALL COLOR	for character colors
CALL HCHAR and **CALL VCHAR**	for character identification, location and duplication

You also have all the BASIC language commands available too! How can you put them together in some fun and exciting applications?

Think about these ideas:

- Write a program to create a musical, digital clock.
- Animate an ant or a flea on the screen.
- Fire a missile at a target.
- Create a flashing sign with random colors.
- Draw a large heart on the screen. Can you make it beat? With a musical background?
- Draw a Christmas tree. With lights. With music.

We will investigate some of these applications in the section below.

Table 10.1 Hearing chart.

Note	Loudness Level	Can You Hear This? (Yes/No)	
		Your Answer	Friend's Answer
131	30		
	26		
	22		
	18		
	14		
	10		
	6		
	2		
262	30		
	26		
	22		
	18		
	14		
	10		
	6		
	2		
523	30		
	26		
	22		
	18		
	14		
	10		
	6		
	2		
1047	30		
	26		
	22		
	18		
	14		
	10		
	6		
	2		

SOME FUN APPLICATIONS OF TI SOUND AND COLOR GRAPHICS

Application 1: A Hearing Test

How about a program to test your hearing? The following program will do that. Enter this program in your TI-99/4A and use the chart below to test your level of hearing. Have a friend run the program and compare your results.

Program: A Hearing Test

```
10 REM APPLICATION #1-HEARING TEST
20 CALL CLEAR
30 PRINT "-SET TV VOLUME NOW- "
40 PRINT " "
50 PRINT "SET VOLUME UNTIL YOU CAN JUST
   BARELY HEAR THIS TONE "
60 CALL SOUND(4200,140,18)
70 FOR K=1 TO 900
80 NEXT K
90 NOTE(1)=131
100 NOTE(2)=262
110 NOTE(3)=523
120 NOTE(4)=1047
130 FOR I=1 TO 4
140 PRINT
150 FOR J=30 TO O STEP -4
160 CALL SOUND(1500,NOTE(I),J)
170 PRINT "Hz=  ";NOTE(I)
180 PRINT "LOUDNESS LEVEL=  ";J
190 PRINT "CAN YOU HEAR THIS? YES/NO-
    MARK ON THE CHART "
200 PRINT " "
210 PRINT " "
220  FOR KK=1 TO 1000
230 NEXT KK
240  NEXT J
250 PRINT " "
260 PRINT " "
270 FOR M=1 TO 1900
280 NEXT M
290 CALL CLEAR
300 NEXT I
310 END
RUN
```

Application 2: The Star Spangled Banner

Ever wonder what it would be like to be in battle and hear the "Star Spangled Banner"? Listen to this song and hear "the bombs bursting in air" as you enter and run this next TI BASIC program.

Program: Star Spangled Banner

```
10 REM APPLICATION #2: BATTLE AND THE
   STAR SPANGLED BANNER
20 CALL SOUND(1000,330,15)
30 CALL SOUND(1000,294,15)
40 CALL SOUND(1000,262,15)
50 CALL SOUND(1000,330,15)
60 CALL SOUND(1000,392,15)
70 CALL SOUND(3000,523,15)
80 CALL SOUND(1000,659,15)
90 CALL SOUND(1000,587,15)
100 CALL SOUND(1000,392,15)
110 CALL SOUND(1000,330,15)
120 CALL SOUND(1000,370,15)
130 CALL SOUND(2000,392,15)
140 FOR I=130 TO 1400 STEP 7
150 CALL SOUND(1,I,15)
160 NEXT I
170 FOR I=6 TO 7
180 CALL SOUND(1000,-I,9)
190 NEXT I
200 END
RUN
```

Modify this program by adding more notes from the "Star Spangled Banner".

Application 3: Your Heartbeat

In medicine, computers are used to search the human body for disease. Computers show blood flow, organs in operation, joints moving, and lungs expanding and contracting. These displays are often in color on computer monitors. You can simulate some of these activities on the TI-99/4A. How would a heart look as it beat and pumped blood through the body? How could you show it expanding and contracting? Computer graphics are the key.

This next program uses the **CALL SOUND, CALL HCHAR, CALL SCREEN, CALL CLEAR,** and **CALL COLOR** graphics command to simulate the beating of a heart. The sound is included to simulate the sound of the heart beat. Load this program, run it, and modify the instructions to make the simulation better. Use this as an example to see what changes you can make.

Note: How the screen is treated as a grid and the **CALL HCHAR** command positions the characters on the screen to represent the heart.

```
       H          H
     HHHHH      HHHHH
   HHHHHHHHH  HHHHHHHHH
 HHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHHHHHHHHHHH
 HHHHHHHHHHHHHHHHHHHHHHHH
  HHHHHHHHHHHHHHHHHHHHHH
    HHHHHHHHHHHHHHHHHH
       HHHHHHHHHHHH
          HHHHHH
            HH
```

Figure 10.6 Using characters to make a heart.

What other organs could you reproduce using TI sound and color graphics? Could you show how the leg moves? How would the eye look on the TV screen? What types of sound would you add to the movement of your human part?

Program: The Human Heart

```
10 REM APPLICATION #3: THE HUMAN HEART
20 DIM C(19),N(19),C1(19),N1(19)
30 REM READ NORMAL HEART SIZE DATA
40 M=72
50 SWITCH=1
60 FOR I=1 TO 13
70 READ C(I),N(I)
80 DATA 10,13,9,15,9,15,9,15,9,15,9,15,10,13
90 DATA 10,13,11,11,12,9,13,7,15,3,16,1
100 NEXT I
110 REM READ EXPANDED HEART SIZE
```

```
120 FOR I=1 TO 15
130 READ C1(I),N1(I)
140 DATA 7,19,7,19,7,19,7,19
150 DATA 7,19,7,19,7,19,8,17,9,15,10,13,11,11
160 DATA 12,9,13,7,15,3,16,1
170 NEXT I
180 CALL SCREEN(8)
190 CALL COLOR(6,7,7)
200 CALL CLEAR
210 CALL SOUND(1500,131,2)
220 CALL HCHAR(7,12,72,2)
230 CALL HCHAR(7,19,72,2)
240 CALL HCHAR(8,11,72,5)
250 CALL HCHAR(8,17,72,5)
260 FOR I=1 TO 13
270 CALL HCHAR(1+8,C(I),72,N(I))
280 NEXT I
290 REM SHOW EXPANDED HEART
300 CALL SOUND(1500,262,3)
310 REM SIMULATE HEART BEAT
320 CALL HCHAR(5,10,M,3)
330 CALL HCHAR(5,20,M,3)
340 CALL HCHAR(6,19,M,5)
350 CALL HCHAR(6,9,M,5)
360 CALL HCHAR(7,8,M,7)
370 CALL HCHAR(7,18,M,7)
380 CALL HCHAR(8,7,M,9)
390 CALL HCHAR(8,17,M,9)
400 FOR I=1 TO 13
410 CALL HCHAR(I+8,C1(I),M,C(I)-C1(I))
420 CALL HCHAR(I+8,C(I)+N(I),M,(N1(I)-N(I))/2)
430 NEXT I
440 CALL HCHAR(22,15,M,3)
450 CALL HCHAR(23,16,M,1)
460 SWITCH=SWITCH*(-1)
470 IF SWITCH>0 THEN 500
480 M=32
490 GOTO 510
500 M=72
510 GOTO 210
520 END
RUN
```

When this program is typed in and run, you will see a large, red heart expanding and contracting on the screen. Modify the program to improve it. Can you add veins and arteries? Can you show the chambers in the heart? Can you show the blood flow? Try some of these modifications.

Application 4: A Colorful Chime and Digital Clock

Do you own a digital clock? Do you own a TI-99/4A? Then you own a digital clock! Your TI can be programmed to become a digital clock. It will also have sound and color. The computer changes screen color and chimes on the hour. Imagine your friends' reactions when they see your computer keeping time in your home or apartment.

The following program simulates a digital clock for your TI-99/4A.

Program: Digital Clock

```
10 REM APPLICATION #4: DIGITAL CLOCK
   SIMULATION
20 CALL CLEAR
30 PRINT "SET YOUR CLOCK NOW "
40 FOR DELAY=1 TO 100
50 NEXT DELAY
60 INPUT "WHAT IS THE HOUR?(1-12) ":HOUR
70 INPUT "WHAT IS THE MINUTE?(1-59) ":MIN
80 INPUT "WHAT IS THE SECOND?(1-59) ":SEC
90 CALL CLEAR
100 CALL SCREEN(HOUR+2)
110 PRINT TAB(8);HOUR; ": ";MIN; ": ";SEC;
120 FOR I=1 TO 138
130 NEXT I
140 CALL CLEAR
150 IF SEC>58 THEN 180
160 SEC=SEC+1
170 GOTO 510
180 IF MIN>58 THEN 220
190 SEC=0
200 MIN=MIN+1
210 GOTO 510
220 IF HOUR>11 THEN 470
230 SEC=0
```

```
240 MIN=0
250 HOUR=HOUR+1
260 FOR K=1 TO HOUR
270 CALL SOUND(600,130*K,2)
280 FOR T=1 TO 135
290 NEXT T
300 NEXT K
310 SEC=SEC+HOUR
320 GOTO 510
330 HOUR=1
340 MIN=0
350 SEC=0
360 CALL SOUND(200,130,2)
370 GOTO 80
380 STOP
390 IF HOUR>12 THEN 470
400 SEC=0
410 MIN=0
420 HOUR=HOUR+1
430 FOR K=1 TO HOUR
440 CALL SOUND(200,130,2)
450 NEXT K
460 GOTO 510
470 HOUR=1
480 MIN=0
490 SEC=0
500 CALL SOUND(200,130,2)
510 GOTO 90
520 END
RUN
```

Application 5: A Christmas Tree

The following program combines music with sound and color graphics to play the song, "O Christmas Tree" as it constructs a tree on the screen. Lights blink different colors on the tip of the tree as the background color changes and the notes in the song change. It is a festive program.

```
                0
                |
                T
            0  TTT  0
            | TTTTT |
            TTTTTTTTT
      0   TTTTTTTTTTTTT   0
      | TTTTTTTTTTTTTTTTT |
      TTTTTTTTTTTTTTTTTTTTT
0   TTTTTTTTTTTTTTTTTTTTTTTTT   0
| TTTTTTTTTTTTTTTTTTTTTTTTTTTTT |
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
              TTTTT
              TTTTT
              TTTTT

" O CHRIST MAS TREE, O CHRIST MAS TREE
   HOW BEAU TI FUL AND BRIGHT. "
```

Figure 10.7 The tree displayed by the program.

Program: Christmas Tree

```
10 REM APPLICATION #5: CHRISTMAS TREE
20 REM THIS PROGRAM CONSTRUCTS A
   CHRISTMAS TREE
30 REM IT ALSO PLAYS THE SONG "O
   CHRISTMAS TREE " AND PRINTS THE WORDS
   TO THE SONG
40 CALL CLEAR
50 PRINT "THE PROGRAM PUTS A CHRISTMAS
   TREE ON THE SCREEN, PLAYS A SONG AND
   PRINTS THE WORDS TO THE SONG. "
60 PRINT
70 PRINT "THE WORDS ARE: "
80 PRINT
90 PRINT
100 PRINT
110 PRINT "O CHRIST MAS TREE "
120 PRINT "O CHRIST MAS TREE "
130 PRINT "HOW BEAU TI FUL AND BRIGHT. "
140 PRINT
150 PRINT
160 PRINT
170 FOR I=1 TO 2300
```

```
180 NEXT I
190 CALL CLEAR
200 CALL COLOR(7,13,13)
210 CALL HCHAR(3,16,84)
220 CALL HCHAR(4,15,84,3)
230 CALL HCHAR(5,14,84,5)
240 CALL HCHAR(6,13,84,7)
250 CALL HCHAR(7,12,84,9)
260 CALL HCHAR(8,11,84,11)
270 CALL HCHAR(9,10,84,13)
280 CALL HCHAR(10,9,84,15)
290 CALL HCHAR(11,8,84,17)
300 CALL HCHAR(12,7,84,19)
310 CALL HCHAR(13,6,84,21)
320 CALL HCHAR(14,5,84,23)
330 CALL HCHAR(15,15,84,3)
340 CALL HCHAR(16,15,84,3)
350 CALL HCHAR(17,14,84,5)
360 CALL HCHAR(18,14,84,5)
370 CALL HCHAR(19,14,84,5)
380 REM LOCATE LIGHT STEMS
390 CALL HCHAR(2,16,33)
400 CALL HCHAR(5,13,33)
410 CALL HCHAR(5,19,33)
420 CALL HCHAR(8,10,33)
430 CALL HCHAR(8,22,33)
440 CALL HCHAR(11,7,33)
450 CALL HCHAR(11,25,33)
460 CALL HCHAR(13,5,33)
470 CALL HCHAR(13,27,33)
480 ICOLOR = 3
490 JCOLOR = 5
500 CALL COLOR(6,ICOLOR,JCOLOR)
510 CALL HCHAR(1,16,79)
520 CALL HCHAR(4,13,79)
530 CALL HCHAR(4,19,79)
540 CALL HCHAR(7,10,79)
550 CALL HCHAR(7,22,79)
560 CALL HCHAR(10,7,79)
570 CALL HCHAR(10,25,79)
580 CALL HCHAR(12,5,79)
590 CALL HCHAR(12,27,79)
```

```
600 CALL SOUND(500,262,2,131,2,220,2)
610 GOSUB 910
620 CALL SOUND(375,262,2,349,2,175,2)
630 GOSUB 910
640 CALL SOUND(125,262,2,349,2,175,2)
650 GOSUB 910
660 CALL SOUND(500,262,2,349,2,175,2)
670 GOSUB 910
680 CALL SOUND(500,330,2,392,2,131,2)
690 GOSUB 910
700 CALL SOUND(375,349,2,440,2,175,2)
710 GOSUB 910
720 CALL SOUND(125,349,2,440,2,175,2)
730 GOSUB 910
740 CALL SOUND(750,349,2,440,2,175,2)
750 GOSUB 910
760 CALL SOUND(250,349,2,440,2,175,2)
770 GOSUB 910
780 CALL SOUND(250,175,2,330,2,392,2)
790 GOSUB 910
800 CALL SOUND(250,349,2,440,2,349,2)
810 GOSUB 910
820 CALL SOUND(500,392,2,233,2,165,2)
830 GOSUB 910
840 CALL SOUND(500,330,2,131,2,233,2)
850 GOSUB 910
860 CALL SOUND(500,294,2,392,2,175,2)
870 GOSUB 910
880 CALL SOUND(500,262,2,349,2,175,2)
890 CALL SOUND(250,4000,30)
900 GOTO 600
910 IF JCOLOR = 12 THEN 950
920 JCOLOR = JCOLOR + 1
930 ICOLOR = ICOLOR + 1
940 GOTO 970
950 JCOLOR = 3
960 ICOLOR = 5
970 CALL COLOR(6,ICOLOR,JCOLOR)
980 CALL SCREEN(ICOLOR)
990 RETURN
1000 END
RUN
```

Check your input listing if you don't see a flashing Christmas tree and hear music.

REVIEW ACTIVITIES

1. What does the **CALL SOUND** command do?
2. What are the variables you must specify in the **CALL SOUND** statement?
3. How many notes can you play within one **CALL SOUND** statement?
4. What is a *Hertz* and what does it have to do with **CALL SOUND?**
5. What is *noise* in the **CALL SOUND** statement and how can we use it?
6. List a number of other commands in TI color graphics and tell what they do.
7. How many different colors can be specified in the **CALL SCREEN** command?
8. What is the difference between the **CALL VCHAR** and the **CALL HCHAR** commands?
9. List the parameters or variables in the **CALL COLOR** command.
10. What is the purpose of the character-set number in the **CALL COLOR** command?
11. What are the dimensions of the screen? Are all positions usable?
12. Define what this command would do: **CALL COLOR**(1,2,8)
13. What other comand is often used together with the **CALL COLOR** command?
14. What color would the screen be with this command: **CALL SCREEN**(8)
15. Why is it important to program sound and color graphics into computer applications?

Chapter 11

Program and Data Files

Storing the BASIC programs and data on magnetic storage is a subject broad enough to fill a book. Saving certain types of files on cassette tape is the only area of storing a BASIC program that will be dealt with here.

The first accessory for your TI-99/4A that you should consider buying is a cassette recorder on which to store BASIC programs. You need the following equipment:

1. A cassette tape recorder with:
   a. an output jack (earphone or extension speaker jack);
   b. an input jack (microphone jack);
   c. volume control;
   d. tape counter (a necessity);
2. a cassette interface cable; and
3. a high quality blank cassette tape not longer than 60 minutes.

The cassette recorder is connected to the TI-99/4A with the interface cable so that the red tail is plugged into the microphone jack, the white tail is plugged into the earphone jack, and the other end of the cable is plugged into the back of the computer. If the cassette recorder has a remote jack, plug the black tail into it.

You may have as many as two recorders at the same time connected to the computer with device names that the computer will recognize, CS1 and CS2. Here you are going to use only one recorder, and it must be connected as CS1. This connection makes a difference only if you have the double cassette interface cable.

Before you save or record or load a program or data from tape, be sure that the recorder's

1. volume is set at about one-half;
2. treble (if there is tone control) is set at maximum; and
3. tape counter is set at the appropriate spot (000 for beginning).

SAVING A BASIC PROGRAM

When you **SAVE** a BASIC program, the computer is actually converting your program into a noise that it can understand. Once it has made this noise code copy of the program, it records it on cassette tape.

After you have written the BASIC program and are ready to **SAVE** it, type:

```
>SAVE CS1
```

Note: The letters used to type in this command must be in upper case (capitals).

This command starts a **ROM** program that you cannot interrupt without the computer leaving the **ROM** BASIC operating level which will cause you to lose the BASIC program you are trying to **SAVE**. The **SAVE** program, however, may interrupt itself. If it hits a problem it will stop execution and give you an error message. In this case you will not lose the BASIC program. You will simply have to try to correct the problem and start the **SAVE** program over again. Assuming the computer has not encountered a problem so far, it will respond with the message:

```
*REWIND CASSETTE TAPE CS1
 THEN PRESS ENTER
```

You rewind the tape to the spot where you want to start recording and write down the position given by the tape counter for future reference. Then press ENTER. Next the computer sends the message:

```
*PRESS CASSETTE RECORD CS1
 THEN PRESS ENTER
```

This means you press the button(s) that start the cassette tape recording and then press ENTER. Next you will see the message:

```
*RECORDING
```

After a short wait, you may hear the special computer to cassette code that actually records on the tape. When the computer is through recording the program on the tape it will send the message:

```
*PRESS CASSETTE STOP CS1
 THEN PRESS ENTER
```

Now push the button(s) that stop the cassette recording and press ENTER. You should write down the tape counter position so you know where the program begins and ends on the cassette tape. Then the computer will ask:

```
*CHECK TAPE (Y OR N)?
```

It is always a good idea to have the computer check the program on the cassette tape to see if it was saved without any problems. If you type a Y (without ENTER), the computer will continue by sending the message:

```
*REWIND CASSETTE TAPE CS1
 THEN PRESS ENTER
```

When you do that, it will tell you to:

```
*PRESS CASSETTE PLAY CS1
 THEN PRESS ENTER
```

At this point press the Play button(s) on the cassette recorder and press ENTER on the computer. The computer will respond with the message:

```
*CHECKING
```

After a short wait, and if everything is OK, the computer tells you:

```
*DATA OK
 PRESS CASSETTE STOP CS1
 THEN PRESS ENTER
```

After you press ENTER, the computer will stop executing this **ROM** program and you can continue to write programs in BASIC or whatever.

Note: The computer does not actually take your program and put it on tape, it copies the program onto the tape. If you now **LIST** your program, you will find that it is still in the computer's current memory.

All of the above assumes that the computer did not encounter a problem trying to **SAVE** the program. If this was not the case, here are some possible areas to check.

First make sure the cassette recorder is working; try recording your voice. Check the batteries and plug. If you can, make sure the microphone jack works. Plug in a microphone and record your voice. If the recorder has a volume setting for recording, turn it to about half volume.

Next check to seee if the remote is working. When the computer gives the message:

```
*PRESS CASSETTE RECORD CS1
  THEN PRESS ENTER
```

If the tape does not start turning after you have pressed ENTER, unplug the remote (black) tail from the remote jack and plan to start and stop the computer manually.

Next check that the computer, monitor, and recorder are not on a continuous metallic surface. Computers and TVs generate an unusual amount of electrical noise that you can't hear but the tape recorder can. The further away from these two pieces of equipment the tape recorder is, the better.

If all else fails, consult your TI-99/4A User's Guide. It has a very good section on this subject.

LOADING A BASIC PROGRAM

Loading a program is how the computer retrieves a program that you have previously saved. Like saving a program from the current memory onto tape, the computer does not actually take your program away from the tape. It makes a copy of it and puts it into its current memory. When you are ready to load a program you have saved, type in:

```
>OLD CS1
```

Remember that this must be in upper case (capital) letters. The computer will then respond by sending the message:

```
*REWIND CASSETTE TAPE CS1
  THEN PRESS ENTER
```

This tells you to set the tape recorder at the tape counter position that marks the beginning of where the program is saved. After you press ENTER, the computer will tell you to:

```
*PRESS CASSETTE PLAY CS1
  THEN PRESS ENTER
```

When you have done this, the computer will give you the message:

```
*READING
```

which will stay on the screen until it has memorized your program (or until it recognizes that there is a problem). If the program was successfully loaded, the computer will tell you:

```
*DATA OK
*PRESS CASSETTE STOP CS1
 THEN PRESS ENTER
```

at which point you turn off the recorder and press ENTER. You can then **LIST** or **RUN** your program as you please.

Since the computer does not take your program away from the tape, there is no reason to resave it after you are through with it unless you have changed (added to or **EDITed**) the program.

Chapter 12

String Functions and Subprograms

STRING FUNCTIONS

So far you have been processing primarily numbers or numeric data. You have yet to explore the area that processes characters instead of numbers. Data processing, referred to as word processing, alphabetic processing, or string processing is a big application area for computers. Some people estimate that 30 to 40 percent of all personal computers sold are purchased to do word processing. Texas Instruments has a large, powerful, packaged program which can be purchased for the home computer. It is called *TI Writer*.

How does the compute process string data? What can it do with letters? Two obvious applications of this process are correspondence and letter writing. Letters are keyed into the computer, then displayed on the screen for editing, storage, and printing. In this way you can personalize form letters and make multiple copies. Some companies even combine word and data processing applications by sending letters to customers that contain past due amounts retrieved from billing records. Word processing lets you cut-and-paste or rearrange paragraphs in a letter or speech or move paragraphs from one letter to another. Word processing is even used to analyze speeches from Russian and Chinese leaders to measure the hostility index of the speech. Key words are assigned point values and the speech is scanned for these words. When a key word is encountered, the program assigns its point value to a running total hostility value for that speech. By monitoring these speeches, the U.S. Government measures changes in attitudes within that country's leadership.

In the office of the future, word processing will be a critical concern. Letters will be stored on disks and sent electronically from one office to another or from the company to the customer.

Electronic libraries are possible where you can access the books or periodicals you would like to read. An actual copy of the book or article could be sent to your home computer for reading at your leisure. Newspapers and classified ads could work the same way. Word processing will be an integral part of life in the future.

How does it work on the TI-99/4A? Remember that alphabetic letters and special characters in the computer are stored in ASCII numeric representation. Each character converts to its preassigned numeric code. Thus letters can be processed because they are stored numerically in the computer. A collection of one or more letters is called a *string*. Finally, signal the computer that a variable name stands for a string of characters instead of a number by attaching the $ character to the end of the variable name. Figure 12.1 reviews the use of variable names.

Variable Name	Numeric or String Data
TEST	Numeric Data
TEST$	String Data
SUM	Numeric Data
SUM$	String Data
B(4)	Numeric Data
B$(A+B)	String Data
INFORMATION$	String Data

Figure 12.1 Naming variables.

Texas Instruments has supplied its home computer with a number of built-in BASIC string functions or commands. They are used to process the strings of alphabetic and special character data ($, ; ' # etc.) in the program. Typically, they evaluate the status or content of the string and return some information about the string back to the program. You can also modify or add string data together to form new strings. This is identical to what you can do with numeric data. A complete vocabulary of string commands or functions is shown in Figure 12.2.

The ASCII Function

The ASC function gives the ASCII character code of the first character in the string expression. This function converts characters to their numeric equivalent in the computer. Remember, the complete list of ASCII character codes is shown in Appendix I.

Form

ASC(string expression)

Example

```
10 B$ = "HELP "
20 PRINT "THE ASCII CODE
   FOR H = ";ASC(B$)
RUN
```

Answer: 72

The CHaRacter String Function

The **CHR$** string returns the character corresponding to the ASCII code in the numeric expression. The normal range of numeric values is between 32 and 127. The **CHR$** performs the opposite function of the **ASC** command.

Form

CHR$(numeric expression)

Example

```
100 A$ = CHR$(49)
110 B$ = CHR$&
    CHR$(39)&CHR$(109)
120 PRINT A$
130 PRINT B$
RUN
```

Answer:

```
1
M&m
```

The LENgth Function

The **LEN** function returns the number of characters in a string expression. This function answers the question: How long is the string? Blanks in the string are valid characters and add to the string's length. A null string, one with no characters or not defined, yields a zero from the **LEN** function. Note also that this function does not end with a $. Thus it returns a number rather than a string result.

Form

LEN(string expression)

Example

```
100 B$ = "5 "
110 C$ = "FIVE "
120 PRINT B$;LEN(B$)
130 PRINT C$;LEN(C$)
RUN
```

Answer:

```
5   1
FIVE 5
```

The POSition Function

Assume string1 is the source string, and string2 is the match string. The **POSition** function searches string1 to see if it contains string2. This is a very powerful function. If the second string is in the first string, the function returns the position number of the first occurence of that string in the original string. If the string is not found, **POS** returns a value of zero. Note that it stops searching after the first occurrence. If string2 occurs more than once in string1, the search must be restarted from that point. The numeric expression argument or parameter identifies the starting position for the search in string1.

Form

POS(string1,string2,
numeric expression)

Example

```
100 MESSAGE$= "WHAT IS
    IT? "
110 TEST1$= "AT "
120 PLACE=
    POS(MESSAGE$,
    TEST1$,1)
130 PRINT TEST1$;
     " OCCURS IN ";PLACE;
140 PRINT " RD ";
     " POSITION OF ";
    MESSAGE$
RUN
```

Answer: AT OCCURS IN 3 RD POSITION OF WHAT IS IT?

The SEGment Function

The **SEGment** function returns a substring or segment of a source string. The substring begins at numeric expression1 and is numeric expression2 in length. The substring can be assigned to another string variable using the **LET** statement. Imagine using the **POS** statement explained above to find a string and then using the **SEG** function to assign that string to another string variable.

Form

SEG$(string expression,
numeric expression1,
numeric expression)

Example

```
100 SOURCE$= "HELLO
    BILL "
110 N$= BSEG$
    (SOURCE$,7,4)
120 PRINT "MY NAME
    IS ";N$
```

```
130 END
RUN
```

Answer: MY NAME IS BILL

The STRing Function

The **STRing** function converts a number to a string. When a number is converted to a string, the leading and trailing blanks are eliminated. The **STRing** function, for example, would convert the number portion of an address to its string equivalent. Then the entire address could be stored as a string variable.

Form

STR$(numeric expression)

Example

```
100 VALUE = 100.5
110 NUM$ = STR$(VALUE)
120 PRINT VALUE,NUM$
130 END
RUN
```

Answer: 100.5 100.5
(Which is string, which is numeric?)

The VALue Function

The **VALue** function performs the opposite function of the **STR$** function. The **VALue** function converts string data to their numeric equivalent, and then allows numeric processing of the results. This would be used to convert string data such as "10/22/83" for sorting by date of occurrence. The string would have to be converted to its numeric equivalent. Then all dates could be sorted. An example of this is given later in the chapter.

Form

VAL(string expression)

Example

```
100 NUM$ = "100.5 "
110 VALUE = VAL(NUM$)
120 AMOUNT = VALUE*4
130 PRINT
NUM$;VALUE;AMOUNT
140 END
RUN
```

Answer: 100.5 100.5 402

SOME APPLICATIONS OF TI STRING FUNCTIONS

Consider how you can use these seven functions in various applications. In the examples below, the string functions are combined to perform the required activity. As you read and review these examples, think of how you might use these functions in your applications. Develop your programs and enter them for testing on your TI-99/4A

Example 12.1: Letters in a Word

How many S's are in MISSISSIPPI? Write a TI BASIC program to find the number of S's in any word. What functions would you need to use? How would you change the search letter?

Solution:

```
10 REM EXAMPLE 12.1
20 REM PROGRAM TO FIND S 's IN WORDS
30 COUNT=0
40 READ WORD$
50 LETTER$= "S "
60 START=1
70 CHECK=POS(WORD$,LETTER$,START)
80 IF CHECK=0 THEN 120
90 COUNT=COUNT+1
100 START=CHECK+1
110 GOTO 70
120 PRINT "THERE ARE ";COUNT;LETTER$; "'s IN  ";WORD$
130 DATA "MISSISSIPPI "
140 END
RUN
```

Answer: THERE ARE 4 S's IN MISSISSIPPI

What would you change to find the number of R's in RAILROAD TRACKS?

Answer:

```
10 REM PROGRAM FOR R 's
50 LETTER$= "R "
130 DATA "RAILROAD TRACKS "
```

Example 12.2: Reversing Name Fields

Names on forms are often written or reported as:

Last Name, First Name.

Yet, this is very impersonal. Write a program using string functions to change this form to:

First Name Last Name.

The process requires you to:

1. find a comma in the name field;
2. assign the characters to the left of the comma to LASTNAME$;
3. assign the characters to the right of the comma to FIRSTNAME$;
4. **PRINT** FIRSTNAME$, a blank and then LASTNAME$.

Solution:

```
100 REM EXAMPLE 12.2
110 REM THIS PROGRAM REVERSES
120 REM THE NAME FIELD FROM
130 REM LAST NAME, FIRST NAME TO FIRST NAME LAST
NAME
140 PRINT "INPUT NAME(LAST,FIRST)-MUST BE EN-
    CLOSED IN QUOTES"
150 INPUT NM$
160 PRINT NM$
170 SPOT=POS(NM$,",",1)
180 LAST$=SEG$(NM$,1,SPOT-1)
190 FIRST$=SEG$(NM$,SPOT+1,25)
200 PRINT FIRST$;" ";LAST$
210 END
RUN
```

Answer: HANSON, MARSHA
MARSHA HANSON
("HANSON, MARSHA" was the name entered in statement 150.)

BASIC String Function	Operation	Example
1. **ASC**(string expression)	Returns ASCII character code for the first character in the string expression.	ASC(H) is 72
2. **CHR$**(numeric expression)	Returns the character corresponsing to the ASCII code in the numeric expression.	CHR$(72) is
3. **LEN**(string expression)	Returns the number of characters in the string expression.	N$="NEW YORK LEN(N$) is 8

4. **POS**(string1, string2,numeric expression)	Returns the location of string2 within string1. Search for string1 for string2 begins at numeric expression.	M$ = "TI-99/4A
		N$ = "99 "
		POS(M$,N$,1)

5. **SEG$**(string expression,numeric expression1, numberic expression2)	Returns a portion of "string expression" beginning at numeric expression1 position and running for numeric expression2 characters.	TEST$ = "BASIC SEG$)TEST$.1 "BASIC "
6. **STR$**(numeric expression)	Changes a number to its string equivalent.	A = 400.6 STR$(A) is "
7. **VAL**(string expression)	Changes a string expression to its numeric equivalent.	A$ = "361.61 " VAL(A$) is 3

Figure 12.2 TI BASIC string functions.

Notes:

1. Function names ending in "$ " (**CHR$**, **SEG$**, **STR$**) return string values.
2. Function names without "$ " (**ASC**, **LEN**, **POS**, **VAL**) return numeric values.
3. **ASC** and **CHR$** functions are inverse functions or perform opposite functions of each other.
4. **STR$** and **VAL** functions are also inverse functions of each other.

Example 12.3: Date Conversions

Often data are supplied in an alphabetic format. It is easy for people to enter data in that format. Yet, to do processing, you need the data in its numeric form. Many documents collect the date in the form:

month/day/year
or
mm/dd/yy

If you want to sort these documents by date, you need to convert to a numeric format first and then sort. To do this imagine the data comes in the following form in **DATA** statements:

```
300 DATA "08/26/83"
```

Write a program to **INPUT** these data, convert them to their numerical equivalent and then **PRINT** them.

Solution:

```
100 REM EXAMPLE 12.3
110 REM THIS PROGRAM CONVERTS
120 REM STRING DATE DATA TO
130 REM NUMERIC DATE DATA.
140 REM ORIGINAL DATA IN THE
150 REM FORM OF MM/DD/YY.
160 REM CONVERTED DATA IN THE
170 REM FORM YY,MM,DD.
180 READ TDATE$
190 PRINT "ORIGINAL FORM "
200 PRINT TDATE$
210 START = 1
220 BREAK$ = "/"
230 DATA "10/11/83"
240 FOR I = 1 TO 2
250 SPOT = POS(TDATE$,BREAK$,START)
260 NMBR = SPOT - START
270 VALUE$(I) = SEG$(TDATE$,START,NMBR)
280 START = SPOT = 1
290 NEXT I
300 YEAR$ = SEG$(TDATE$,SPOT+1,2)
310 PRINT
320 PRINT "CONVERTED FORMS "
330 PRINT "YEAR = ";VAL(YEAR$)
340 PRINT "MONTH = ";VAL "(VALUE$(1) )
350 PRINT "DAY = ";VAL(VALUE$(2) )
360 PRINT
370 PRINT "DATA NUMBER "
380 DATEY = 10000 * VAL(YEAR$)
```

```
390 DATEM=100*VAL(VALUE$(1))
400 DATED=VAL(VALUE$(2))
410 PRINT DATEY+DATEM+DATED
420 END
RUN
```

Answer:

```
ORIGINAL FORM
10/11/83

CONVERTED FORMS
YEAR=83
MONTH=10
DAY=11

DATE NUMBER
 831011
```

Example 12.4: Search and Replace Words

Can you write a program to change the *a* word to an *an* word in the following sentence?

"What a great day!"

Change the sentence to read:

"What an great day!"

Review the following example to see how string functions are used to find and replace a word in a sentence.

Solution:

```
10 REM EXAMPLE 12.4
20 REM THIS PROGRAM SEARCHES FOR A KEY WORD
   AND REPLACES
30 REM THAT WORD WITH ANOTHER WORD.
40 REM IN THIS EXAMPLE, "A "
50 REM IS THE SEARCH WORD
60 REM AND "AN " IS THE REPLACE WORD.
70 PRINT "INPUT SENTENCE TO BE SEARCHED "
80 PRINT "(ENCLOSE SENTENCE IN QUOTES)"
90 INPUT SENTENCE$
100 PRINT "WHAT IS THE SEARCH WORD "
110 PRINT "(ENCLOSE WORD IN QUOTES-WITH LEAD-
    ING AND TRAILING BLANKS)"
```

```
120 INPUT SEARCH$
130 PRINT "WHAT IS THE REPLACE WORD? ENCLOSE IN
    QUOTES AND BLANKS "
140 INPUT REPLACE$
150 START=1
160 SPOT=POS(SENTENCE$,SEARCH$,START)
170 NSENTENCE$=SEG$(SENTENCE$,1,SPOT-1) &RE-
    PLACE$&SEG$(SENTENCE$,SPOT+3,
180 PRINT
190 PRINT "ORIGINAL= ";SENTENCE$
200 PRINT "REVISED= ";NSENTENCE$
210 END
RUN
```

Answer: ORIGINAL=WHAT A GREAT DAY!
REVISED=WHAT AN GREAT DAY!

In summary, you can write very complex programs to process string data. There are some programming langugaes used exclusively for string processing such as SNOBOL 4 and LISP. TI BASIC has seven string processing functions. They are very powerful and their use is limited only by your ability to understand how they can be included in your programs. Working through the previous examples and the exercises at the end of this chapter will help you learn these string functions. Try to think of applications for string variables. You will start to see how word processing programs manipulate their data.

Subprograms

Subprograms are separate small programs that can be written and incorporated into a larger calling or mainline program. They are used to carry out an activity that is used quite often in the program. String functions discussed in the previous section are good examples of subprograms. The seven string functions are called upon by the programmer to perform some activity or function related to string data. Subprograms may be built-in to the language like the string functions or they may be written by the programmer to do special tasks. In a payroll application, the mainline program might read in data about the employee and separate subprograms or subroutines could:

1. compute gross pay;
2. compute deductions;

3. print the paycheck; and
4. print the pay stub (showing deductions and year-to-date figures).

Numeric Functions

Functions are subprograms supplied with the BASIC language. They perform commonly-used computations on string or numeric data. Numeric functions compute such things as:

1. absolute value of a number (no sign, all positive);
2. rounding routines;
3. trigonomic functions like sine, cosine, and tangent values;
4. converting values to their **INTeger** equivalents (removes the decimal portion of the number); and
5. a variety of other activities.

Functions are independent programs which have the ability to return one computed result to the calling program. The arguments or parameters are given to the function in a list (values after the function name enclosed in parentheses). The function returns the results of its computation by assigning the answer to the function name. The function format is:

function name (argument list)

The function can be used as an expression in the right-hand side of a computation or **LET** statement like:

20 **LET** RESULT = **ABS**(–4 ✻ 16)
(Takes the absolute value of –4 ✻ 16 or 64 and assigns this value to **ABS** which in turn assigns it to RESULT.)

or

300 **LET** WHOLE = **INT**(B ✻ SUM)
(Assigns integer portion, no decimal, of B ✻ SUM computation to **INT** which in turn assigns it to WHOLE.)

Functions can also be used to stand alone such as in an output statement or a computation statement or in a decision statement like:

50 **IF** K > **INT**(C ✻ D) **THEN** 60
(Compares K to integer result of C ✻ D.)

Typically, program execution of subprograms would be like the flow diagram shown in Figure 13.3. Note the logic flow in this diagram.

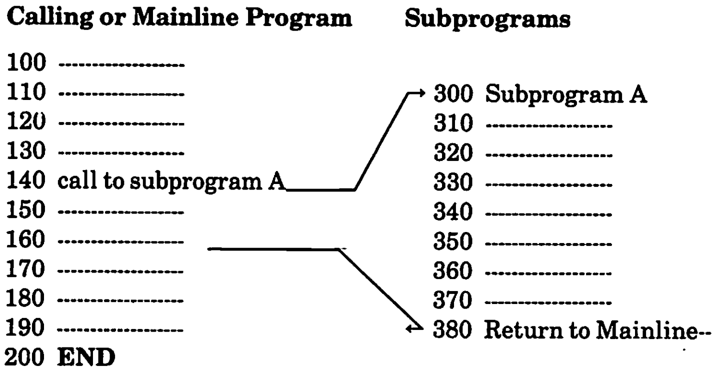


Figure 12.3 Subprogram logic flow.

The mainline program calls the subprogram. Any values computed in the subprogram are than available for use by the calling program.

Appendix VI lists the 12 built-in functions in TI BASIC. The examples below show how they can be used in programs.

Example 12.5: Generating random numbers (numbers with equal chance of occurrence).

Write a program to compute and print 10 random numbers between zero and one.

```
10 REM EXAMPLE 12.5
20 REM PROGRAM TO FIND 10 RANDOM NUMBERS
30 PRINT " RANDOM #'S "
40 PRINT " ========= "
50 FOR I = 1 TO 10
60 VALUE = RND
70 PRINT VALUE
80 NEXT I
90 END
RUN
```

Answer:

```
RANDOM #'S
==========
.5291877823
```

```
.3913360723
.5343438556
.3894551053
.2555008073
.5621974824
.2553391577
.5882911741
.7000201301
.0010849577
```

Note: All of these numbers fall within the range of less than 1 and greater than or equal to zero.

Example 12.6: Absolute and Integer Values

Absolute values are values with no signs. All values are positive. The absolute value of a positive number is the same positive number. The absolute value of a negative number is the same number, but stripped of its sign. The absolute value of 10 is 10. The absolute value of –10 is 10. Another way to write this is /–10/ is 10 where / / stands for absolute value of the number inside the slash marks. Integer values are numbers stripped of their decimal portions. In this example we find and print the absolute values and integer values of five number: 10.1, –16.3, 11.51, 143.001 and –16.51.

Solution:

```
10 REM EXAMPLE 12.6
20 REM PROGRAM TO FIND ABSOLUTE AND INTEGER
VALUES OF NUMBERS
30 PRINT "VALUE ";TAB(10); "ABSOLUTE ";TAB(20);
   "INTEGER "
40 PRINT "======================= "
50 FOR I=1 TO 5
60 READ A
70 B=ABS(B)
80 C=INT(A)
90 PRINT A;TAB(10);B;TAB(20);C
100 DATA 10.1,-16.3,11.51,143.001,-16.51
110 NEXT I
120 END
RUN
```

Answer:

```
VALUE          ABSOLUTE          INTEGER
               ==============
10.1           10.1              10
-16.3          16.3              -17
11.51          11.51             11
143.001        143.001           143
-16.51         16.51             -17
```

Example 12.7: Rounding Values

In this example you develop a program to find the *cents* equivalent of five random numbers. Remember the range for random numbers? Round these values where necessary. For example:

if random value = 0.4611348, cents = 0.46
if random value = 0.9350005, cents = 0.94

Solution:

```
10 REM EXAMPLE 13.7
20 REM ROUNDING PROGRAM
30 PRINT " RANDOM # CENTS "
40 PRINT "=========
50 FOR K = 1 TO 5
60 A = RND
70 CENTS = INT( (A+0.005)*100)/100
80 PRINT A,CENTS
90 NEXT K
100 END
RUN
```

Answer:

```
RANDOM #       CENTS
==========
.5291877823      .53
.3913360723      .39
.5343438556      .53
.3894551053      .39
.2555008073      .26
```

Example 12.8: Trigonometry Functions

Angles can be measured by a number of trigonometric functions. TI BASIC has many of these functions included in its language. They include sine, cosine, tangent, etc. Write a program to find the sine and cosine of 0 degrees, 30 degrees, 60 degrees, and 90 degrees.

Remember, the argument for **SIN** and **COS** must be in radians. To convert degrees to radians, multiply degrees by pi/180 where pi = 3.14159 or 22/7. Note in the example that both these values for pi are used.

Solution:

```
10  REM EXAMPLE 12.8
20  REM PROGRAM TO FIND THE
30  REM SINE AND COSINE OF
40  REM 0, 30, 60, AND 90 DEGREES
50  PRINT  "DEGREES ";TAB(10); "SINE ";TAB(20); "COSINE "
60  PRINT  " ========== "
70  FOR I = 1 TO 4
80  READ DEGREES
90  DATA 0,30,60,90
100  SVALUE = SIN(DEGREES*(22/7/180) )
110  SAVLUE = INT(SVALUE*1000+0.5)*0.001
120  CVALUE = COS(DEGREES*3.14159)/180)
130  CVALUE = INT(CVALUE*1000+0.5)*0.001
140  PRINT DEGREES;TAB(10);SVALUE;TAB(20);CVALUE
150  NEXT I
160  END
RUN
```

Answer:

```
DEGREES   SINE      COSINE
          ==========
0         0         1
30        .5        .866
60        .866      .5
90        1         0
```

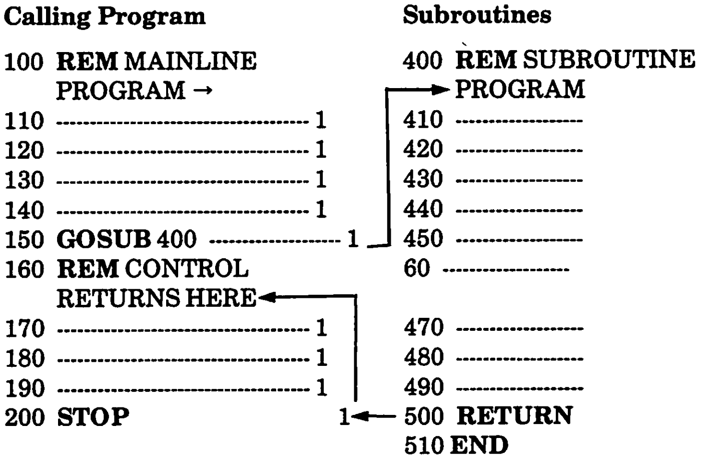
Think of other applications for these functions. Think of ways they can be put together. How many functions can you put in one statement and still determine the output? Try it. Now let's move on to subroutines.

Subroutines

Subroutines can return one or more values to the calling program. Any value computed and assigned to a variable in a subroutine is available to the calling program after the value has been computed. One subroutine may even call another subroutine. Key statements in subroutines are the **GOSUB** statement, the

RETURN statement, the **STOP** statement and the **END** statement. The following figure shows how control and execution are passed from the calling program to the subprogram and back again.

In Figure 12.4, the **GOSUB** statement in line 150 passes execution to line 400, the first statement in the subroutine. Sequential execution continues in the subroutine until the **RETURN** statement in line 500 is encountered. Control then passes back to the next statement beyond the **GOSUB** in the calling program, in this case line 160. Sequential execution continues in the calling program until statement 200, the **STOP** statement, is encountered. This terminates execution of the entire program. Note the **END** statement in line 510 of the subroutine. This is the highest line number in the program and it must contain the final record in the program file–the **END**statement.



```
Calling Program                          Subroutines

100 REM MAINLINE                         400 REM SUBROUTINE
    PROGRAM →                              ──► PROGRAM
110 --------------------------- 1        410 --------------
120 --------------------------- 1        420 --------------
130 --------------------------- 1        430 --------------
140 --------------------------- 1        440 --------------
150 GOSUB 400 ----------------- 1        450 --------------
160 REM CONTROL                          60 --------------
    RETURNS HERE◄──
170 --------------------------- 1        470 --------------
180 --------------------------- 1        480 --------------
190 --------------------------- 1        490 --------------
200 STOP                        1◄──     500 RETURN
                                         510 END
```

Figure 12.4 Subroutine format and program control.

Now look at some subroutine examples. Subroutines are very useful in BASIC programming. If you find yourself repeating a series of BASIC code more than once in a program, you should be using a subroutine and a calling program configuration.

Example 12.9: Payroll Computation

Write a mainline program to read in four employees' hourly wage rate and hours worked this week. Print out the employee number and gross pay. Use a subroutine to compute gross pay.

Solution:

```
10 REM EXAMPLE 12.9
20 REM PROGRAM TO COMPUTE EMPLOYEES PAY
30 PRINT "EMPLOYEE ";TAB(10); " HOURS "; " GROSS PAY "
40 FOR EMPLOYEE = 1 TO 4
50 READ RATE,HOURSWORKED
60 GOSUB 110
70 PRINT EMPLOYEE;TAB(12);HOURSWORKED; TAB(20);
   GROSS
80 NEXT EMPLOYEE
90 DATA 8,40,7,36,10,40,6.50,35.5
100 STOP
110 REM GROSS PAY SUBROUTINE
120 GROSS = RATE * HOURSWORKED
130 RETURN
140 END
RUN
```

Answer:

```
EMPLOYEE HOURS      GROSS PAY
1        40         320
2        36         252
3        40         400
4        35.5       230.75
```

Note:

1. The **STOP** statement halts mainline program execution. Otherwise the mainline program would run into the subroutine. There may be more than one **STOP** statement in a program.

2. The **END** statement is always physically the last statement in a program (as determined by its line number). The **END** statement terminates the program listing. There is always only one **END** statement in a program.

Example 12.10: Sorting

Sorting is a major activity of computer programs. Often you may need an alphabetical listing of people or cities or products. Or you need to sort part numbers, dates, or purchase amounts in ascending or descending order. BASIC programs can do this for you.

Write a calling program to read the weights of six people into an array called WEIGHT. Then call a subroutine to find the heaviest weight in the group. Finally, have the calling program print the weight of the heaviest person.

Solution:

```
10 REM EXAMPLE 12.10
20 REM PROGRAM TO FIND LARGEST NUMBER IN AN ARRAY
30 DIM WEIGHT(6)
40 HEAVIEST = 0
50 PRINT "BEGINNING WEIGHTS "
60 FOR I = 1 TO 6
70 READ WEIGHT(I)
80 PRINT TAB(6);WEIGHT(I)
90 NEXT I
100 DATA 165,145,315,185,210,261
110 GOSUB 150
120 PRINT
130 PRINT "LARGEST WEIGHT =  ";HEAVIEST
140 STOP
150 REM SUBROUTINE TO FIND LARGEST NUMBER IN AN ARRAY
160 FOR I = 1 TO 6
170 IF HEAVIEST > WEIGHT(I) THEN 190
180 HEAVIEST = WEIGHT(I)
190 NEXT I
200 RETURN
210 END
RUN
```

Answer:

```
 BEGINNING WEIGHTS
          165
          145
          315
          185
          210
          216

LARGEST WEIGHT = 315
```

Example 12.11: More Sorting

Write a program to sort the six weights in Example 12.10 above into ascending order (small to large). Use a subroutine to sort and a mainline routine to input the data and output the results.

Solution:

```
10 REM EXAMPLE 12.11
```

```
20 REM PROGRAM TO SORT ARRAY DATA FROM SMALL
TO LARGE
30 DIM WEIGHT(6)
40 FOR INDEX = 1 TO 6
50 READ WEIGHT(INDEX)
60 NEXT INDEX
70 DATA 165,145,315,185,210,261
80 GOSUB 150
90 PRINT "RANK ", "WEIGHT "
100 PRINT "=====", "====="
110 FOR INDEX = 1 TO 6
120 PRINT INDEX,WEIGHT(INDEX)
130 NEXT INDEX
140 STOP
150 REM SORT ROUTINE
160 FOR 1 = 1 TO 6
170 FOR J = 1 TO 6
180 IF WEIGHT(I)< = WEIGHT(J) THEN 220
190 TEMP = WEIGHT(J)
200 WEIGHT(I) = WEIGHT(J)
210 WEIGHT(J) = TEMP
220 NEXT J
230 NEXT I
240 RETURN
250 END
RUN
```

Answer:

```
RANK      WEIGHT
=====     =====
  1        145
  2        165
  3        185
  4        210
  5        261
  6        315
```

Note: Look closely at the activity in this subroutine to understand sorting. Follow the subscript values (set from the indexes in the two nested loops) and array values as the program executes. The following table might be helpful.

Since WEIGHT(J) is less than WEIGHT(I) you must switch the values of these two. This is explained below:

Step #	I Value	J Value	WEIGHT(I)	WEIGHT(J)	Is WEIGHT(I) <=WEIGHT(J)?
1	1	1	165	165	YES
2	1	2	165	145	NO

Figure 12.5 shows that when a new, smaller weight is found, the two elements in the array must change places. Statement 190 in the subroutine temporarily stores its contents in TEMP. In the second step, statement 200 shifts its contents into the first position of the array WEIGHT. Remember, this is a smaller number than what was previously stored there, and your objective is to sort small to large. Now statement 210 shifts the contents of TEMP back into the second position of the array WEIGHT.

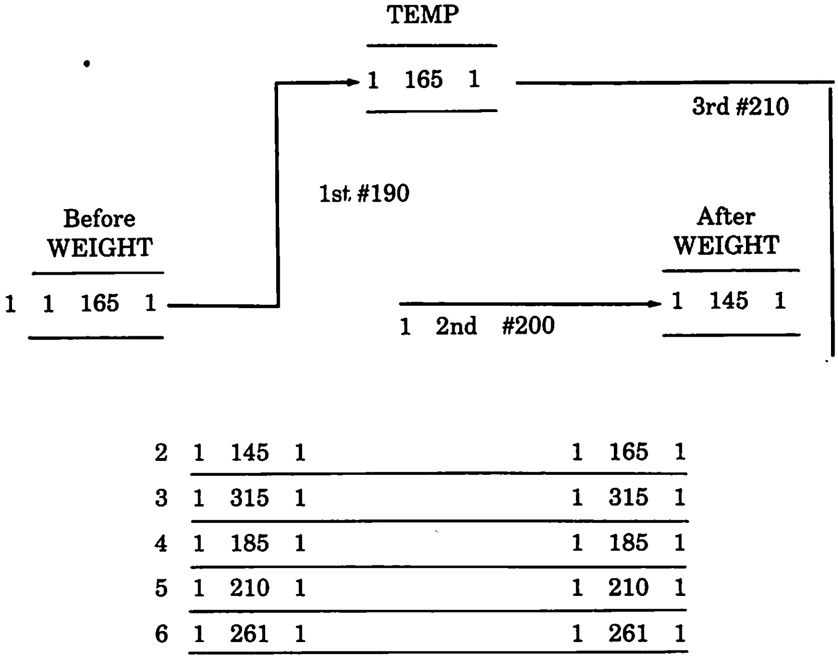


Figure 12.5 The switch routine when sorting.

With this shift complete, the process continues. Now WEIGHT(1), the new, smallest weight will be compared to WEIGHT(3) through WEIGHT(6) to see if any of these are smaller than WEIGHT(1). If any one is, the switch routine will be implemented again and a new smallest weight will be placed in WEIGHT(1). Eventually the smallest value in the array will bubble-up to the first location and remain there. When I changes to 2 (in statement 160), the sort will begin to sort for the second smallest number. Eventually that value will bubble-up to WEIGHT(2). The process continues until all values are arranged from small to large.

Question: How would you change this program to sort from large to small?

Answer: Change statement 180 to read:

```
180 IF WEIGHT(I)> =WEIGHT(J) THEN 220
```

A careful review of Example 12.11 will add greatly to your understanding of sorting.

Example 12.12: Alphabetical Sorts

Remember that letters are sorted using ASCII codes for each character. A has a lower code than B. Strings are stored as long numeric values. Thus you can ask if STRING1$>STRING2$ and get an answer. Look at this example of an alphabetical sort routine for names.

Write a calling program to read four names, print them out unordered (last name, first name). Then call a subroutine to sort those names alphabetically. Finally, have the mainline routine print the sorted names.

Solution:

```
10 REM EXAMPLE 12.12
20 REM ALPHABETIC SORT ROUTINE
30 PRINT "NAME BEFORE SORTING"
40 PRINT "========= "
50 DIM NAMES$(4)
60 DATA "PHONES, BILL ", "MARSH, HENRY ", "MARS,
   JIM "
70 DATA "WINSLOW, MARSHA "
80 FOR I1=1 TO 4
90 READ NAMES$(I1)
100 PRINT NAMES$(I1)
110 NEXT I1
```

```
120 GOSUB 200
130 PRINT
140 PRINT "NAMES AFTER SORTING "
150 PRINT " ========== "
160 FOR K = 1 TO 4
170 PRINT NAMES$(K)
180 NEXT K
190 STOP
200 REM ALPHA SORT ROUTINE
210 FOR I = 1 TO 4
220 FOR J = I TO 4
230 IF NAMES$(I)< =NAMES$(J) THEN 270
240 TEMP$ = NAMES$(I)
250 NAME$(I) = NAMES$(J)
260 NAMES$(J) = TEMP$
270 NEXT J
280 NEXT I
290 RETURN
300 END
RUN
```

Answer:

```
NAMES BEFORE SORTING
==========
PHONES, BILL
MARSH, HENRY
MARS, JIM
WINSLOW, MARSHA

NAMES AFTER SORTING
==========
MARS, JIM
MARSH, HENRY
PHONES, BILL
WINSLOW, MARSHA
```

SUMMARY

Subprograms can be called by a mainline program and return values to the calling program. Two types of subprograms are functions and subroutines. Functions return a single value to the calling program. TI BASIC comes with a number of built-in functions, some processing string data and others used for numeric values. Parameters or arguments, contained in parentheses just after the function's name, describe what the function should process. The functions supplied by Texas Instruments in their BASIC language perform a number of tasks that are often needed when programming.

If a function is not available to do your task, subroutines are the answer. They are written by the programmer to do a specific task. A subroutine may return a number of values to the calling program and may be called as many times as needed, often being supplied with new data to analyze at each call. **GOSUB** and **RETURN** statements control the program flow in and out of a subroutine. If you find yourself repeating similar segments of code in your program, you should probably be using a function or subroutine programming format.

CONCLUSION

With the purchase of a home computer and this book, you have taken the first step toward life in the 21st century. You are aware of what personal computers can do and in what areas. You have learned how to communicate with computers using BASIC programming, the most popular computer language today. You have used the most popular home computer on the market, the TI-99/4A. You have read about Texas Instruments and its technology. You have taken a giant leap toward computer literacy. You are aware of the impact computers might have on life in the future. Your home computer may well become your communications link with society's networks.

Not all computer applications are beneficial or even cost effective. The computer is only a tool for human decision-makers. Computer power is real. Capturing and directing this power is a function of the user. It can be used for freedom or control. The ultimate limitation is the imagination, character, and ingenuity of the user. You have the opportunity and the responsibility to become as effective as possible in the application of this technology in your life and work. Continue to read and keep informed in the area. Search out opportunities for advancement. As the old scout motto says, "Be prepared." Get ready for life in the next century. You can be certain of only one thing–it will be different!

Good luck and Godspeed on your journey into the computer age!

Appendix I

Reserved Words in TI BASIC

There are some reserved, or keywords, in TI BASIC that you may not use as variable names in your computer programs. Some of these are used as function names, some are system commands, and some are keywords in BASIC instructions. To use these names as variables in your program will cause errors. A list of TI BASIC reserved words is given below.

ABS
APPEND
ASC
ATN
BASE
BREAK
BYE
CALL
CHR$
CLOSE
CON
CONTINUE
COS
DATA
DEF
DELETE
DIM
DISPLAY
EDIT
ELSE
END
EOF
EXP
FIXED
FOR
GO
GOSUB
GOTO
IF
INPUT
INT
INTERNAL
LEN
LET
LIST
LOG
NEW
NEXT
NUM
NUMBER
OLD
ON
OPEN
OPTION

OUTPUT
PERMANENT
POS
PRINT
RANDOMIZE
READ
REC
RELATIVE
REM
RES
RESEQUENCE
RESTORE
RETURN
RND
RUN
SAVE
SEG$
SEQUENTIAL
SGN
SIN
SQR
STEP
STOP
STR$
SUB
TAB
THEN
TO
TRACE
UNBREAK
UNTRACE
UPDATE
VAL
VARIABLE

Appendix II

ASCII Character Codes

All characters that print to the screen–letters, numbers, and symbols are assigned a character code. The standard characters are represented by character codes ranging from 32 through 127. A list of the characters and their codes is given bleow. Capital letters are represented by the codes 65 through 90. Small letters are represented by the codes 97 through 122. These character codes are used with the graphics and string commands.

ASCII Code	Character
32	[blank] (space)
33	! (exclamation point)
34	" (quote)
35	# (number or pound sign)
36	$ (dollar)
37	% (percent)
38	& (ampersand)
39	' (apostrophe)
40	((open parenthesis)
41	) (close parenthesis)
42	* (asterisk)
43	+ (plus)
44	, (comma)
45	– (minus)
46	. (period)
47	/ (slant)
48	0
49	1
50	2

51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	: (colon)
59	; (semicolon)
60	< (less than)
61	=(equals)
62	> (greater than)
63	? (question mark)
64	@ (at sign)
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[(open bracket)

92	\(reverse slant)
93	] (close bracket)
94	∧ (exponent)
95	_ (underscore)
96	` (grave)
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{ (left brace)
124	| (vertical line)
125	} (right brace)
126	~ (tilde)
127	DEL (appears on screen as a blank)

Appendix III

Color Codes and Set Numbers

The following two tables show the color codes and set numbers that are used with the color-graphics commands. The codes are contained here for reference. A complete description of their use is contained in Chapter 10, "An Introduction to Sound and Color-Graphics".

The character codes shown in Appendix II are grouped into 12 sets for use by color-graphics programs. The sets and their codes are shown below.

Set Numbers

Set #	Character Codes	Set #	Character Codes	Set #	Character Codes
1	32-39	5	64-71	9	96-103
2	40-47	6	72-79	10	104-111
3	48-55	7	80-87	11	112-119
4	56-63	8	88-95	12	120-127

Two additional characters are preset on the TI-99/4A. The cursor is assigned ASCII code 30 and the edge character is assigned ASCII code 31.

Color Codes

Color	Code #	Color	Code #
Transparent	1	Light Blue	6
Black	2	Dark Red	7
Medium Green	3	Cyan	8
Light Green	4	Medium Red	9
Dark Blue	5	Light Red	10

Dark Yellow	11	Magenta	14
Light Yellow	12	Gray	15
Dark Green	13	White	16

Appendix IV

Musical Note Frequencies

Frequency	*Note*	*Frequency*	*Note*
110	A	440	A (above middle C)
117	A#,B♭	466	A#,B♭
123	B	494	B
131	C (low C)	523	C (high C)
139	C#,D♭	554	C#,D♭
147	D	587	D
156	D#,E♭	622	D#,E♭
165	E	659	E
175	F	698	F
185	F#,G♭	740	F#,G♭
196	G	784	G
208	G#,A♭	831	G#,A♭
220	A (below middle C)	880	A (above high C)
233	A#,B♭	932	A#,B♭
247	B	988	B
262	C (middle C)	1047	C
277	C#,D♭	1109	C#,D♭
294	D	1175	D
311	D#,E♭	1245	D#,E♭
330	E	1319	E
349	F	1397	F
370	F#,G♭	1480	F#,G♭
392	G	1568	G
415	G#,A♭	1661	G#,A♭
		1760	A

This Appendix shows the frequencies of four octaves of musical notes including sharps (#) and flats (b). Remember, the frequencies go well above 1760Hz. Use values from this Appendix with the CALL SOUND command.

Appendix V

TI Error Messages

Error messages or diagnostics occur when the computer encounters something in your program it cannot do. This error message will help you identify the type of error where it has occurred. Expect to make mistakes and try to learn something from each error. Every programmer makes mistakes. View each error as a mental challenge or treasure hunt. Search, find, and remove the elusive bug!

THREE TYPES OF ERRORS

The TI-99/4A checks for errors at three different times while it processes your program. The first check occurs as you enter each line of code from the keyboard. If the statement does not conform to the rules of BASIC, the computer beeps and prints an error message on the screen. This is called an entry error or Type I error or syntax error.

When all the statements in the program are entered correctly, but before running the program, the computer assigns space for the variables, checks to see if branching statements connect, if arrays are dimensioned, etc. It does this through a symbol table. Now the computer may find errors that were not obvious when individual lines were entered. Here, the computer prints an error message and the line number containing that error. A **FOR** statement without a matching **NEXT** would be caught in the symbol table. Type II errors are printed while the screen is still blue. Once all Type II errors are corrected, the program **RUN** begins, and the screen turns to light green on a color TV. Any errors printed on the light green screen are Type III errors and occur during the actual running of your program. To advance this far, your program must be free of Type I and Type II errors.

If the program passes the symbol table check and begins **RUNing**, it may encounter an execution error. Here you have asked the computer to do something it cannot do. Division by zero or not enough space in a dimensioned variable are two examples. An error message and statement number are printed after the **RUN** command.

Below are listed the common error messages for each type of error.

TYPE I ERRORS OR ENTRY ERRORS

Bad Line Number. Line numbers must be greater than zero and less than 32768. Also, if the **RESEQUENCE** command generates a value greater than 32767, this error occurs.
Remedy: Retype line numbers within acceptable limits.

Bad Name. Variable name exceeds 15 characters.
Remedy: Retype line numbers within acceptable limits.

Can't Do That. This message is caused by a number of different errors. Key BASIC statements such as **FOR, NEXT, RETURN** entered without line numbers generate this error. Or if you use system commands such as **RUN, LIST, NEW, OLD, SAVE** with line numbers, the error message is CAN'T DO THAT.
Remedy: Check your line numbers. Make sure they are used only with BASIC instructions.

Incorrect Statement. Common causes of this error include: unmatched quotation marks around literal output messages; no valid separator (colon, comma, quote mark) between variable names in a **READ, INPUT** or **PRINT** statement; invalid print separator between numbers in system commands (**LIST, NUMBER, RESEQUENCE** or **RUN**).
Remedy: Check punctuation in input/output statements. Check punctuation in system commands.

Line Too Long. More than three lines of code entered in one statement. This exceeds the size of the input buffer.
Remedy: Break large statements into smaller ones.

Memory Full. The last line you entered causes the program to exceed the available core storage.
Remedy: Rework your program to reduce or streamline the lines of code.

TYPE II ERRORS OR SYMBOL TABLE ERRORS

These errors occur immediately after the **RUN** statement is entered, but before execution begins. The screen color is still blue.

Bad Value. Dimensions for an array exceed 32769 (machine capacity), or an array dimension may be assigned zero when the **OPTION BASE** = 1 command is in effect.

Remedy: Check array dimension sizes. Check the status or condition of the **OPTION BASE** command.

Can't Do That. There is more than one **OPTION BASE** command in your program or the **OPTION BASE** statement occurs after (higher line number) the **DIM** statement.
Remedy: Check the status and locations of the **OPTION BASE** commands.

For–Next Error. There are unmatched **FOR–NEXT** statements (one is occurring without another) in your program. Also, you cannot branch into a **FOR–NEXT** loop unless you start at the front door–at the **FOR** statement in the loop.
Remedy: Check for balanced **FOR–NEXT** loops. Also, check all branches to **FOR–NEXT** loops.

Incorrect Statement. A **DIM** statement in your program has more than three dimensions or no dimensions or a variable is used in a **DIM** subscript. Also, **OPTION BASE** command may not be formed correctly–no zero or one argument or a missing key word.
Remedy: Check all **DIM** statement values. Look at the argument values in **OPTION BASE** commands.

Memory Full. Array size exceeds the available core memory or your program and data exceed space available.
Remedy: Review and reduce dimensioned array space. Streamline program code.

Name Conflict. Two arrays have the same name or an array has the same name as a single-valued variable.
Remedy: Check names of all arrays. Resolve duplicate names.

TYPE III OR PROGRAM EXECUTION ERRORS

This error halts execution of the program. As part of the error message, the number of the line causing the error will be printed. This helps pinpoint the problem. The screen color is green.

Bad Argument. A built-in function has a bad argument (value inside parentheses). Remember that **VAL** and **ASC** arguments must be defined.

Remedy: Check to see if arguments for **VAL** and **ASC** are defined and that **VAL** arguments are strings.

Bad Line Number. This means you are trying to branch to a line that does not exist.

Remedy: Check your branching statement to see if the branch line number exists.

Bad Name. Subprogram name in a **CALL** statement is invalid.

Remedy: Check spelling on subprogram name. Retype.

Bad Subscript. Subscript value exceeds array size specified in **DIM** statement or subscript value is zero when **OPTION BASE** 1 was specified.

Remedy: Check subscript computation and size of array arguments against its dimensioned size.

Bad Value. This occurs when some value in the statement exceeds its acceptable limit. Screen and color statements cannot exceed 16 color choices. The sound command is confined to loudness and frequency limits. **CHAR** statement arguments must be valid ASCII codes. If a **STEP** is in the **FOR** statement, the **STEP** must be defined and not zero.

Remedy: Check limit values for all arguments in the error statement. If the arguments are computed, check their formulas. The computer can't work outside its specified limits for each type of statement.

Can't Do That. This is caused by an unmatched pair of statements such as a **RETURN** with no **GOSUB** or a **NEXT** without a **FOR**. The looping or branching operation cannot be performed.

Remedy: Check for a companion, matching statement for the flagged error statement. Add or modify statements to make this match.

Data Error. Here, you have run out of data in **DATA** statements before the **READ** statement was completed.

Remedy: Compare the number of pieces of data in the **DATA** statement against the number of variables in the **READ** statement. Look for missing commas in the **DATA** statement list. Be especially careful of dimensioned variables. Do they have the right number of pieces of data?

File Error. This occurs when you try to close a file that was not opened or reopen a previously opened file. Improper reading from or writing to a file will also cause this error.

Remedy: Check the status and condition of the file referenced in the error statement. Modify open or closed status.

Incorrect Statement. This common error is caused by an improper use of BASIC. The computer has encountered an instruction it cannot process. Some common causes are: missing or unmatched parentheses; missing comma; expression or arithmetic operation using +, −, *,/ improperly formed; missing key word; string expression assigned to a numeric value, or the reverse, or missing punctuation.

Remedy: Review the BASIC format of the statement. You have either too much in the statement or have left something out.

Input Error. Data entered from the keyboard is either too long, or alphabetic data is entered when a numeric value is expected. It can also occur when more data is entered (separated by commas) than is expected.

Remedy: Prepare a screen message for the user to guide and explain the data entry format required, such as:

```
200 PRINT "SEPARATE DATA WITH COMMAS "
```

The I/O error generates a two-digit code that helps explain the error. For example, I/O ERROR 66 IN 300, means error 66 occurred

I/O Error. This is caused by a problem with a peripheral device, typically a cassette recorder. The message may occur when you try to save or retrieve a program from the recorder. Either the program does not exist, the file name is incorrect, the recorder is not on or is disconnected, or you have an illegal input/output command.

Table A.1 I/O Error Codes

	First "X"		Second "Y"
Number	Operation	Number	Error
0	**OPEN**	0	Invalid device or file name
1	**CLOSE**	1	Write Protect in effect
2	**INPUT**	2	Bad open attribute
3	**PRINT**	3	Illegal operation
4	**RESTORE**	4	Insufficient space on storage medium

5	**OLD**	5	File past end
6	**SAVE**	6	Peripheral device not insatalled, defective or not activated
7	**DELETE**	7	File does not exist

in line 300. The 66 actually consists of two values, XY, where X represents the I/O operation that caused the error and Y represents the kind of error that occurred. The range of X and Y is 0 through 7. The following table lists the possible values and meanings of X and Y in the I/O ERROR message.

Remedy: Check condition of cassette recorder. Is it on; cable connected correctly; cassette tape mounted? Is the cassette recorder compatible with the TI-99/4A? Is the device correctly specified in the **DELETE**, **SAVE**, or **OLD** command? Make the necessary corrections.

Memory Full. Program contains too many subroutine branches with no **RETURN** command or a **GOSUB** statement branches to its own line number for example:

```
450 GOSUB 450
```

Also a relational, string, or numeric expression may be too long.

Remedy: Check for matching **GOSUB** and **RETURN** commands. Check all **GOSUB** statements. Divide large expressions into two separate statements and then combine.

Number Too Big. (Warning only – the computer substitutes the largest or smallest value possible on the TI-99/4A and continues execution. Obviously, this is not the proper solution to your problem.)

A numeric operation has produced a result that the computer cannot store. This is called overflow. It is often caused by dividing a number by zero.

Remedy: Check the values of each variable in the expression. Is each defined? Is it typed correctly?

String-Number Mismatch. This is caused by attempting to assign a number to a string variable or a string to a numeric variable. It may also occur when a string variable is used in a numeric function like **TAB(A$)** or **LOG(B$)**. Finally, using a string for a file number in an **OPEN** or **CLOSE** statement will generate this error.

Remedy: Check for compatability of variable and its value or a function and its argument in the error statement. Correct as necessary.

Appendix VI

Built-in Numeric Functions

Section A of this appendix is a summary of the 12 functions, including their name and format. This may be enough information for some users. Section B gives more detail about each function. Use whichever section helps you the most.

SECTION A

Summary of the Built-in Functions

Name	Format
1. **ABS**–Absolute Value	**ABS**(numeric expression)
2. **ATN**–Arctangent	**ATN**(numeric expression)
3. **COS**–Cosine	**COS**(numeric expression)
4. **EXP**–Exponential	**EXP**(numeric expression)
5. **INT**–Integer	**INT**(numeric expression)
6. **LOG**–Natural Logarithm	**LOG**(numeric expression)
7. **RANDOMIZE** Statement	**RANDOMIZE**[seed]
8. **RND**–Random Number	**RND**
9. **SGN**–Sign Function	**SGN**(numeric expression)
10. **SIN**–Sine	**SIN**(numeric expression)
11. **SQR**–Square Root	**SQR**(numeric expression)
12. **TAN**–Tangent	**TAN**(numeric expression)

SECTION B

Detailed Review of Built-in Functions

TI BASIC Numeric Function	Description	Example
1. **ABS**(numeric expression) Absolute value function	Returns absolute value of the argument.	ABS(–21) is 21 ABS(4+10) is 14
2. **ATN**(numeric expression) Arctangent function	Returns the arctangent of the argument. Result is in radians.	ATN(0.44) is 0.4145
3. **COS**(numeric expression) Cosine function	Returns the cosine of the argument when the argument is expressed as an angle in radians. To convert an angle to radians, multiply the angle by pi/180 where pi = 22/7.	COS(1.047) is 0.5
4. **EXP**(numeric expression) Exponential function	Returns the value of ∧(x) where x is the argument and e = 2.71828.	EXP(9) is 8103.08
5. **INT**(numeric expression) Integer function	Returns the largest integer (whole number, no decimal part) that is not greater than the numeric expression. For positive values, the decimal is dropped. For negative arguments, the next smallest integer is used.	INT(–4.64) is –5 INT(3.1619) is 3 INT(4.991) is 4

6. **LOG**(numeric expression) Natural logarithm function	Returns the natural log of the argument. The argument must be greater than zero. The natural log of X is log(base e) of X.	**LOG**(3.5) is 1.2528 **LOG**(−1) is "BAD ARGUMENT "
7. **RANDOMIZE** [seed] Randomize statement	This optional statement is used in combination with the **RND** function. Without **RANDOMIZE**, the random numbers generated by **RND** will always be the same series. When **RANDOMIZE** is used, without a *seed*, each call of **RND** generates a different set of random numbers. The optional seed value defines the starting position for the random number table. If two **RANDOMIZE** seed values are the same, the random values will be the same.	**RANDOMIZE** gives a new set of random numbers each time. **RANDOMIZE** 14 gives the same set of random numbers for each program run.
8. **RND**	Returns a pseudo-random number between 0< =RN<1. The sequence of random numbers for each program run will be the same unless **RANDOMIZE** is used.	A = **INT**(10 * **RND**) +1 is 6

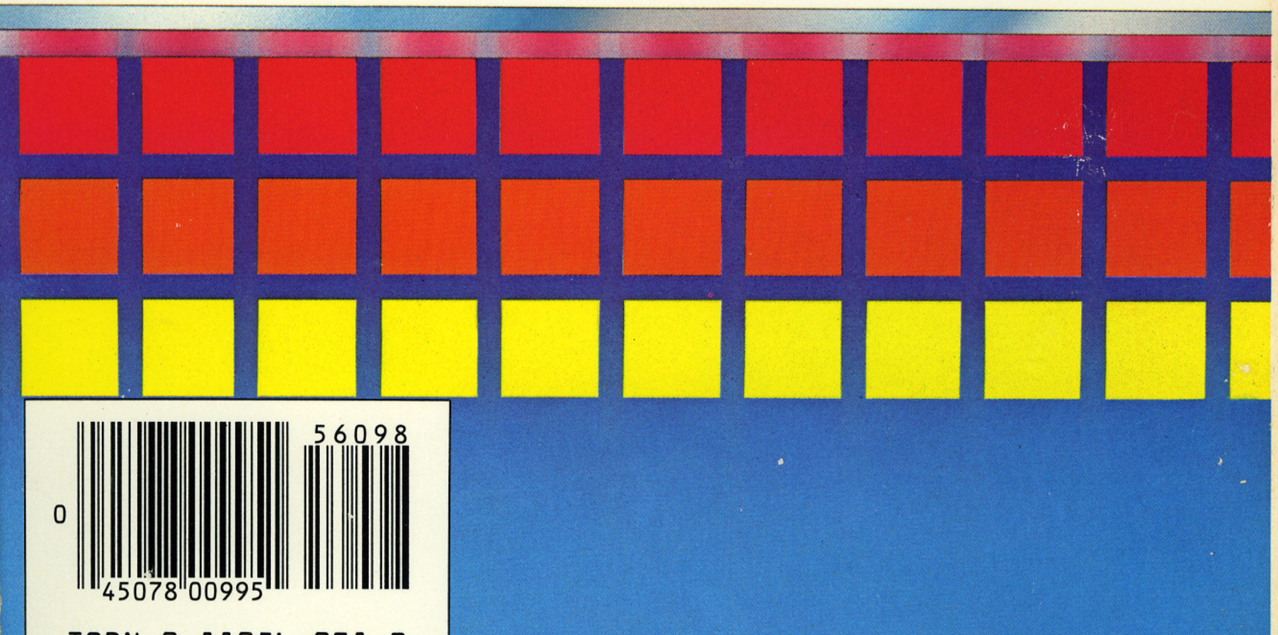
9. **SGN**(numeric expression) Sign function	Returns the algebraic sign of the argument. If the numeric expression is: < 0, then **SGN** = −1 = 0, then **SGN** = 0 > 0, then **SGN** = 1	**SGN**(−11) is −1 **SGN**(0) is 0 **SGN**(413) is 1
10. **SIN**(numeric expression) Sine function	Returns the sine value of the argument when the angle is expressed in radians. If the angle is in degrees, multiply the degrees by pi/180 where pi = 22/7, to get the angle in radians.	**SIN**(30 * 22/7/180) is 0.5 **SIN**(0.523598776) is approximately 0.5
11. **SQR**(numeric expression) Square root function	Returns the square root of the positive argument. If the argument is negative, "BAD ARGUMENT " error message occurs on the screen and the program halts.	**SQR**(81) is 9 **SQR**(−6) is "BAD ARGUMENT "
12. **TAN**(numeric expression) Tangent function	Returns the tangent of the argument when the argument is expressed in radians. To change angle in degrees to radians, multiply by pi/180 where pi = 22/7.	**TAN**(0.7854) is approximately 1 **TAN**((22/7)/180 * 45) is 1

Index

Your TI-99/4A is a powerful tool that can help you with daily activities. Here's a book that shows you how! **GET PERSONAL WITH YOUR TI-99/4A:**

- Gives you background information on computers and teaches you TI BASIC programming techniques.
- Provides you with over 100 example programs and exercises.
- Tells you the fundamentals of how to design and write programs.
- Introduces you to sound and color graphics.
- Concludes each chapter with activities that help you make your own changes and improvements to the example programs.

This book gets you involved with your computer and teaches you good programming skills. **GET PERSONAL WITH YOUR TI-99/4A!**



0 45078 00995 56098

ISBN 0-88056-098-3

>>$9.95

dilithium Press
GENERAL INTEREST