SAMS

The Waite Group's BASIC Programming Primer

Mitchell Waite
Michael Pardee



Second Edition

BASIC Programming Primer

Second Edition

BY

Mitchell Waite
Michael Pardee

Cartoons by Robert Gumpertz
Photography by John Werner & Bruce Brower

HOWARD W. SAMS & COMPANY
A Division of Macmillan, Inc.
11711 North College, Suite 141, Carmel, IN 46032 USA

Copyright © 1978 and 1982 by Mitchell Waite and Michael Pardee

SECOND EDITION
NINTH PRINTING—1990

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-22014-8
Library of Congress Catalog Card Number: 82-51042

Edited by: *Lou Keglovits*
Illustrated by: *William D. Basham*

Printed in the United States of America.

Preface

This new expanded second edition of *BASIC Programming Primer* was written to show beginners that learning BASIC is fun. By using a friendly, down-to-earth style and lucid, exciting examples, this book will help you learn BASIC quickly, gently, and without tears. Although most books on BASIC today are overly pedantic and lack a sense of humor, *BASIC Programming Primer* has a witty manner, clever approach, meaningful graphics and is designed especially to keep up interest. *BASIC Programming Primer* is the only book on BASIC you may ever need.

What is BASIC? It is an extremely popular computer language, taught today in schools around the world, and incorporated into almost all personal computers on the market. Why learn it? Well, by understanding how BASIC works you are in the position to maximize the capacity of your home computer. With BASIC you can write powerful programs for fun or for profit, you can design and program your own computer games, business programs, teaching systems, household helpers, etc. And because BASIC has become a de facto standard among personal computers, knowledge of it puts you in a perfect position to control not one, but all personal computers today. Furthermore, the job market for programmers is growing at an incredible clip and experts agree there are no better employment opportunities today than in programming.

This book is written in such a way that readers at many different levels of computer involvement can learn BASIC in a few hours. In order to concentrate on the language we will consider the computer as a "black box" which responds to commands that we issue using BASIC. The black box has a keyboard like a typewriter, and a screen like a tv with which it can "communicate" with the real world (although its internal world may at times seem real too). Sound like a space drama? Well, it can be if you program it that way. Or, what if business programs are what you are looking for? Add a simple hard-copy printer to the black box, and print mountains of reports. That is how versatile the BASIC language really is.

Like the first edition of *BASIC Programming* Primer, this expanded edition covers everything you need to start using BASIC today. Still included are the things that made this book a best seller for five years, including five action-packed chapters covering BASIC input/output, program control statements, arrays, functions and interesting variations such as PEEK/POKE, CALL, etc. Also there are these demo programs people love: the loan program, metric conversion, and a TIC-TAC-TOE game you can program.

The new edition has features culled from years of feedback from all types of readers (how many books can say that?). First we have added a new chapter called "Advanced BASIC." This chapter will teach you how to code your BASIC programs just like the professionals of computing. It covers all those special BASIC statements that you find in the more expensive disk based languages; in fact, the chapter covers the BASIC statements found in the new IBM Personal Computer BASIC (a Microsoft variant) except graphics and disk (we are saving those for a separate book—watch for it).

Included in this new chapter are the powerful structures like WHILE . . . WEND and IF THEN ELSE. These allow you to code loops that are simple and easy to "see." For defining your own custom functions we cover the defined function statement (DEF FN). Also covered are special input/output commands like LINE INPUT, PRINT USING, INPUT$, and INKEY$. Advanced string functions like INSTR, MID$ on the left, STRING$, SPC(n), and SPACE$ are covered. Error checking, a subject woefully explained in almost all BASIC texts, is explained in depth so that you can prevent your programs from "bombing-out" in midflight. Finally, we discuss a most often confused subject of "variable types" in BASIC, showing you how to define and keep track of the accuracy of your calculations with the powerful double-precision variable types, as well as integers and single precision. As an added bonus we'll show you how to peek into the inner workings of BASIC and discover how it "ticks" with the statement VARPTR, and how to convert from one number system to another within the program.

There are two new programs in this chapter: Micro Space Invaders and Word Scrambler. The Micro Space Invaders lets you play that famous game on your own computer, and at the same time teaches you how the advanced string functions work. The Word Scrambler shows you how to produce all the possible arrangements of letters of a word you type in. You can use this to come up with neat names based on the initial capital letters of a longer name. Through this program you'll discover a most powerful technique called "recursion."

Another new chapter in the book, "Rubik's Cube Program," takes all the concepts you have learned in the previous chapters and puts them together in a neat game/learning program called Rubik's Cube. We show you how a program is conceived, organized, structured and coded, from beginning to end. In case you never heard of it before, Rubik's Cube is a very popular game/device that is based on a colored cube made up of little cubes that can be rotated and moved about. The sides of the cube start out different colors

but in an organized fashion. A few twists and the cube becomes hopelessly scrambled. The object is to return the cube to its virgin state . . . which is a major feat. Since over 7 million cubes have been sold in the last year it is clear that people love this puzzle. With our program you can simulate the twisting and turning of the cube on your own computer and at the same time learn how a program is "put together." We teach you how to flowchart, and how to use "top down," "bottom up," and "middle out" thinking. By the time you finish this chapter you'll be an expert in BASIC.

We've also added a quiz to the end of each chapter, along with answers in an appendix so you can test your understanding.

Through the use of many example programs, and our own recipe for presentation, we hope to arouse your interest, sharpen your wit, and hopefully open your eyes to how to write a BASIC program (if we can put a grin on your face while you read, we'll consider it a bonus). If you think there are things we left out, you're right. It was hard but we deliberately omitted several of the features of BASIC we felt would limit the book. For example, we do not cover disk statements because they are different on different machines. Same thing for graphics. Special books are out that explain graphics for the different computers (for example, see *Computer Graphics Primer* by Mitchell Waite, Howard W. Sams & Co.), so we'll let these do that job. For disk information, consult the manual that comes with your computer. However, we think that we have covered all of the important bases and that you will be quite pleased when you discover you can learn to program a computer just by reading a book.

Once you have read the book, you will only need to refer to it occasionally to make sure how some of the commands work. Realizing this, we have included a removable reference card which has condensed examples of all of the BASIC language. It is a lot easier to carry around than the book, and many people prefer to pin the card next to their computers, for easy look-up of BASIC statements and commands.

So then, carry on, and enjoy your journey into this exciting and fun-filled BASIC "vacation." One warning: Be careful of spending too much time at the terminal. BASIC, it turns out, can be addictive!

The authors would also like to take this opportunity to sincerely thank Robert Lafore, who, with Mitchell, revised portions of the text and wrote Chapters 6 and 7. Much of the wit and humor in these new chapters came from Robert's clever hand. Robert is well known for his "Interactive Fiction" programs.

We have very much enjoyed writing this book and hope you will enjoy reading it. We would love to hear how you felt about the book, what improvements you would like to see, etc. Please send them in care of the publisher.

Mitchell Waite
Michael Pardee

About the Authors



Mitchell Waite, president of The Waite Group, is an experienced programmer, fluent in a variety of computer languages. He has studied nuclear engineering, built biofeedback machines, written poetry, and raced motorcycles. Mr. Waite is the coauthor of several Sams/Waite titles, including *UNIX Primer Plus*, *CP/M Primer*, and *Soul of CP/M*.



Michael Pardee is Managing Editor for the Waite Group, a San Rafael, California corporation specializing in the writing and packaging of personal computer books. Mr. Pardee, one of the earliest writers in the personal computer industry, is coauthor, along with Mitchell Waite, of the books *Microcomputer Primer*, *Your Own Computer*, and the best-selling *BASIC Programming Primer*. Mr. Pardee has extensive experience in business data processing and computerized typesetting for the newspaper industry, and is knowledgable in several high-level and microprocessor based languages. Mr. Pardee is also an accomplished musician fluent in the guitar, piano and organ. His personal interests include sailing and camping.

Contents

CHAPTER 1 13

Basic BASIC

What is BASIC? 13
Many BASIC Dialects
What Can Be Done?—BASIC Applications 15
Business Applications—Entertainment Applications—Educational Applications—Scientific Applications
Getting Started 21
Your First Program 22
Line Numbers 23
LIST 25
RUN Line Number 26
STOP and END 27
Numbers 29
PRINT With Commas 30
Double Commas
PRINT With Semicolons 33
The Direct Mode 35
Figuring in BASIC 36
Exponentiation 37
Little Variable Boxes 37
Variable Names 41
String Variables 42
Concatenation
INPUT 44
TAB 48
BASIC No-No's 50
Reserved Words
Putting it all Together—The Loan Program 52
Expanding a Program 55
Documenting Your Programs (REM) 56
Review Questions 58

CHAPTER 2 **63**

Program Control

The Decision Making Statements 63
FOR...NEXT, the Programmed Loop—IF...Then, Conditional Branches—GOTO, Unconditional Branch—The Endless Loop—ON...GOTO, Indexed Branching
Enhancing the Loan Program 72
Metric Conversion Program and the Menu Concept 76
Simple Function Statements 81
ABS, Absolute Value Function—INT, Integerize Function—FIX—RND, Randomize Function—RANDOMIZE—Combining Functions
The Subroutine Concept 87
ON...GOSUB, Indexed Subroutine Branching
Numeric "Round-Off" Subroutine 90
Error-Checking Subroutine 97
Review Questions 101

CHAPTER 3 **105**

Getting Organized

Arrays and Subscripted Variables 105
Dimensions With DIM 108
Game BASIC: TIC-TAC-TOE 110
Remember the Rules?—Representing the Board—Representing the Markers—Overall Flow of the Game—Displaying the Board—Check for a Tie—Check for a Winner—Input Players Move—Computer Move Algorithm—Finishing the Program
Review Questions 131

CHAPTER 4 **135**

Adding More Power

DATA and READ Statements 135
Metric Conversion Program Enhanced 139
Numeric Functions 142
AND, OR, and NOT—SIN, COS, and ATN—EXP and LOG—SQR Function—SGN Function—MOD Function
String Functions 152
ASC and CHR$—LEFT$, MID$ and RIGHT$—STR$ and VAL—LEN Function
Review Questions 159

CHAPTER 5 161

Variations

PEEK and POKE 161
Controlling the External World With PEEK and POKE—Manipulating Screen Data With PEEK and POKE—Setting Controls and Options With PEEK and POKE
INP, OUT, and WAIT 168
CALL (or USR) 171
What Call Can Do
CSAVE and CLOAD 177
BASIC Command Aids 180
NEW or SCRATCH—AUTO—CLEAR—FRE(o)—CONT—DELETE—EDIT—RENUM or REN—Debug Commands—TRACE and NOTRACE (TRON and TROFF)—DSP
Video Plotting With TAB 188
What Is Your SIN?
Using a Lineprinter 192
LLIST—LPRINT
Review Questions 193

CHAPTER 6 197

Advanced BASIC

Program Structure 198
WHILE...WEND–Easy Looping—ELSE–An Alternative to THEN—Finding Prime Numbers—DEFN–Adding to BASIC's Vocabulary
Advanced Input/Output 207
LINE INPUT–Typing Anything You Like—PRINT USING–Super Output Formatter—INKEY$–Finer Control of the Keyboard—INPUT$–Reading More Keystrokes—POS–Where Is the Cursor?
Advanced String Functions 219
INSTR–Searching for Strings—A Substitute for INSTR—SPACE$–All Spaced Out—Micro Space Invaders—SPC–Another Way to Space Out—STRING$–Stringing Things Out—MID$ (on the left)–Easy String Modification
Dealing With Errors—Creatively 229
ON ERROR GOTO and RESUME—Confronting Your Errors—ERR and ERL–What Was the Error, and Where Did it Happen—Stirring Up Trouble—ERROR Used as an Extended Form of GOSUB—When to Use an Error-Trapping Routine
Variable Types—Integers, Single and Double Precision, Strings 238
When Should You Use Integers, and When Single Precision? —Type Declaration Characters—Double-Precision Variables—When Six Digits Are not Enough—DEFINT, DEFSNG, DEFDBL and DEFSTR–Defining Variable Types—Word Scrambling Program—CINT, CSNG, and CDBL–Converting Variable Types

Miscellaneous, but Useful 258
VARPTR–BASIC Looks Into Memory—HEX$, OCT$, &H and &O—Hexadecimal and Octal Conversions—OPTION BASE–Saving Array Space—SWAP–Exchanging Two Variables
In Conclusion 268
Review Questions 269

CHAPTER 7 **273**

Rubik's Cube Program

Naming the Parts of the Cube 274
Initial Planning Decisions 275
The Picture on the Screen—Representing the Cube With an Array—Representing the Cube With Strings—"Bottom Up" Program Design—Flowcharting and "Top Down" Program Design
Writing the Program 291
The Main Program—The Move List—The Display Routine—Getting Ready to Rotate—The Rotation Routines

APPENDIX A **309**

Numbering Systems

The Binary System 310
Counting—Binary Addition and Subtraction—Binary Multiplication and Division—Representing Binary Numbers—Pulse Representation of Binary Numbers—Serial and Parallel Transmission—Negative Numbers—The Use of Complements—Two's Complement—Binary-Coded Number Representation
Octal Number System 332
Addition in Octal—Octal Subtraction—Octal Multiplication—Octal Division
The Hexadecimal System 326
Hexadecimal Addition—Subtraction in Hexadecimal—Hexadecimal Multiplication

APPENDIX B **331**

Numerical Conversion

Decimal-Hexadecimal-Octal-Binary-ASCII Numerical Conversion 331
ASCII Definitions 333

APPENDIX C 335

When Programs Get Too Big—Space-Slashing Hints

Storage Allocation in Microsoft BASIC 336

APPENDIX D 339

When Programs Get Too Slow—Speed Hints

APPENDIX E 341

BASIC Language Reference

APPENDIX F 351

Exponential (Scientific) Notation

APPENDIX G 357

Answers To Review Questions

Index 365


ZZZ
BASIC

CHAPTER

1

Basic BASIC

WHAT IS BASIC?

BASIC is a programming language for computers. It is especially easy to learn and has quickly become the most popular language among computer users in the world today. The reason for the popularity of BASIC has to do with its origins. The language was developed at Dartmouth College so students with no background in programming could learn to use a computer. You don't need to become a computer scientist to learn how to use a computer. The BASIC language is designed for students and professionals alike. It has all the power of a computer language but is inherently simple to learn and to use.

BASIC has been called the "all purpose" computer language. This is because it can be applied in hundreds of areas ranging from the sciences to the arts to industrial control. Unlike other computer languages, BASIC can be learned outside normal teaching environments. With the recent appearance of "home" computers people have begun to learn BASIC in their homes, or at computer learning centers across the country.

BASIC is not the only language for computers being used today. Many other languages exist and each has its own particular merits and shortcomings. FORTRAN, for example, is a scientific language, used to perform advanced calculations and laboratory work. COBOL is used best in business data processing applications. And then there are SNOBOL, FOCAL, RPG, APL, PILOT, PL/M, LISP, and even versions called SUPER BASIC which are nothing more than enhanced implementations of the standard BASIC language. Don't let the exotic abbreviations confuse you. They each mean something special about the language, but it is not necessary to know them to learn BASIC. And furthermore, none of these other languages can offer the simplicity and gentleness that makes BASIC so easy to learn.

Many BASIC Dialects

Just as there are variations of a language when spoken in different areas of a country, there are several variations in the structure and content of the BASIC



language. We call these differences "dialect" variations. Most of the differences are minor, and mainly due to the fact that there is no fixed "standard" for the elements of the BASIC language. Most other computer languages have been created around a government standard called the ANSI standard (American National Standards Institute) but BASIC somehow missed the boat in this area, and only a minimum report has been suggested.

Since its creation at Dartmouth College, BASIC has undergone several modifications but it still maintains its overall structure. What this means to the user of BASIC is that each manufacturer will add features to its BASIC that are not necessarily found in another version of the language. For example, manufacturer A may include commands that make it possible to use graphics with BASIC, while manufacturer B may include complex mathematical functions and leave graphics out. Or manufacturer A may allow variable names to be up to 31 characters in length while manufacturer B may only allow two-character variable names. On top of this are the differences caused by the particular computer the BASIC language is designed for. A BASIC language designed for an 8080 based micro-computer may not work on a 6502 based computer and vice-versa. And it is not usually possible to take a program written on machine A and get it to work immediately on machine B. We call this characteristic "portability." BASIC is *not* portable, like a standardized language such as FORTRAN. Another way to look at this is to say that BASIC is almost always a "machine-dependent" language.

But for the most part the differences between the various BASICs are such that they can be easily identified and worked around. To make learning BASIC as simple as possible we have set aside Chapter 5, called Variations, to ex-

plore these differences in detail. The bulk of the first four chapters deals with those features of BASIC which are most common to all those on the market today.

WHAT CAN BE DONE?—BASIC APPLICATIONS

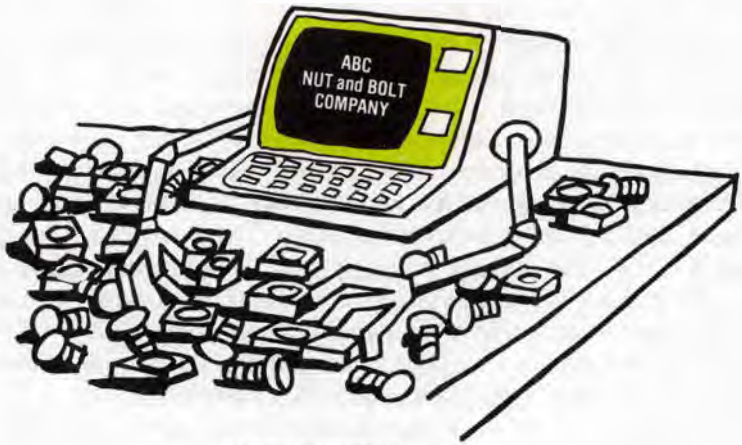
The list of things we can program in BASIC is almost endless, but for clarity we can lump applications into four broad categories: Business, Entertainment, Education, and Science. In this section we will examine some of the types of programs we can create in each of these areas. Later in the book we will explore some of these in more detail.

Business Applications

Although BASIC was never specifically intended to be a business data processing language, it has nonetheless become a popular language for creating business solutions with the computer. Some business program applications include:

General Ledger
Order Processing
Inventory Control
Customer Billing
Mailing Label Preparation

A *General Ledger* (or G/L for short) is a program or a group of programs that automates all the accounting needs of a company on the computer. A G/L replaces the error-prone manual method of bookkeeping. Business transactions are entered into the computer just as they would be entered into a book. They are "posted" to certain accounts, and then summarized into meaningful financial statements or reports. Financial transactions can be entered from



data on check stubs, invoices, deposit slips, or may be noncash entries such as depreciation. A G/L in BASIC helps a company keep track of where its money is being spent and how to make the best use of its financial resources. It also helps to prepare income tax forms at the end of the year. These G/Ls are often sold by independent software vendors in a form that can be adapted to a particular type of business.

Order Processing programs allow the user to enter customer orders into the computer and to make up shipping and billing invoices for certain products or services. A computerized Order Entry program can check inventory stock almost instantly and shipments can be made on a "same day" basis.

Inventory Control programs keep track of the number of items a company has in stock, and allow specific information about the movement of stock items to be displayed. The programs help a business maintain tight control over the volume of merchandise stored in the company warehouses. This in turn helps to increase the inventory turnover rate and save the company money.

Customer Billing programs are used to generate invoices and billing forms for a product or service. They maintain a list of customers and produce bills for these accounts on a periodic basis.

Mailing Label Preparation programs are used to produce mailing addresses for sending bills, invoices, and other data to customers or clients. A mailing label program usually involves some sorting of addresses according to ZIP code, and so on. Enormous amounts of time are saved by having the computer print the mailing labels instead of having employees do it manually.

Entertainment Applications

Even if you purchased your computer for a "serious" purpose such as a business or scientific application, the chances are that sooner or later you will find yourself using it for entertainment. A computer's fast information processing ability and the dazzling graphics displays it is capable of (in color on many computers) make it a natural medium for many different kinds of games. There are literally thousands of game programs currently available. All you need to do with these is buy them, load them into your computer, and enjoy them. It can also be very rewarding, and sometimes even more fun, to write your own game programs. Although they may lack some of the sophistication of a commercial game, you will have the satisfaction of having created them from scratch, and you can "customize" them any way you want to. Because BASIC is the primary language supplied with most personal computers, and because it is easy to learn and to use, it is a natural tool for writing game programs. Many commercial game programs are written in BASIC (although those that require very high speed graphics displays are often written in a much more complicated language called "assembler").

Many traditional games have been adapted for personal computers including chess, checkers, backgammon, Othello, Tic-Tac-Toe, hangman, and Nim.



ENTERTAINMENT I

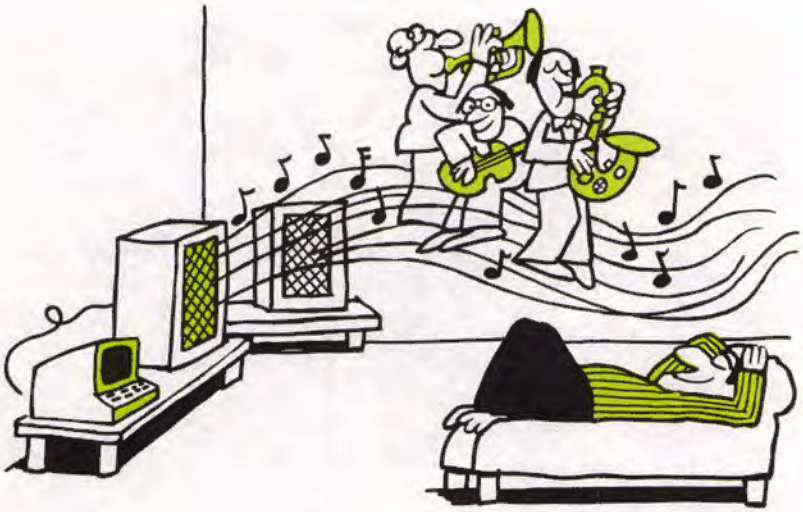
There is also a variety of card games such as blackjack and bridge. In some versions of these games you play against the computer, and in others you play against another human while the computer keeps track of the moves and the score.

Several whole new kinds of games have also appeared, based on the computer's particular abilities. Some of these games make use of the computer's arithmetic speed to "model" or imitate complex physical situations. Your computer can, for example, simulate an airplane, which you "fly" using the keyboard and the video screen. Or you can command a submarine, or a spaceship (as in a wide variety of Startrek games), or try to land a "lunar lander" on the moon without crashing.

Some games are designed to test your manual dexterity. You may have to drive a car along a winding road, escape from dangerous creatures in a maze, or shoot down approaching hordes of alien space ships. Many of these games which are available for personal computers, are versions of the games popular in video arcades across the country.

Other games involve you in a variety of different fantasy situations. In the famous "adventure" games you find yourself in a mazelike castle, or haunted house, from which you must find your way out, while avoiding various dangers, such as trap doors and evil dwarfs. And, in war simulation games, you command an army or a fleet, and reenact famous historical battles.

There is an incredible variety of games to choose from, and new ones are appearing all the time. You can buy them, or, using BASIC, you can write your own version of your favorite game, or perhaps create an entirely new game!



ENTERTAINMENT 2

Educational Applications

Educational applications are becoming increasingly important in the microcomputer field, as more and more educators learn to appreciate the advantages of computers in the classroom. Computers can produce animated color graphics which are almost guaranteed to capture a student's attention; they can provide a student with individual instruction at exactly the level he or she needs; they can create their own questions as they go along and provide instant feedback if a student doesn't understand something; and, above all, they are very patient and do not become annoyed or upset, no matter how many mistakes the student makes.

For very young children there is a variety of educational programs which are so much like games that often a child can be learning without even realizing it. Such programs may use color graphics and a simplified keyboard to teach the difference between left and right, or the recognition of patterns. For slightly older children the alphabet and simple arithmetic can be taught with graphics displays featuring the child's favorite cartoon characters, or with educational versions of popular computer games.

For grade school and high school, computer programs have been devised for almost every subject: math, spelling, grammar, geography, typing, foreign languages—the list can go on and on. There is also a variety of programs to help teachers prepare exams (sometimes tailoring them for each individual student) and grade the results. In more advanced courses computers can be used for creating pictures of DNA molecules, for analyzing the stresses on a bridge, and for a myriad other applications.

As with games, educational programs can either be purchased or written by the user. Many teachers create their own BASIC language educational pro-

grams for use in special learning situations. In this way they can produce a program which is tailored to the specific needs of a group of children in a particular learning situation.

BASIC is also the perfect language for teaching programming, because it is so easy to write—in just a few lines—a program which can actually perform useful tasks. Even a one-line program can do something interesting, as you will see in the next few pages. Many other computer languages, such as Pascal, require every program to start with a group of complicated statements to define what variables are to be used and to perform other functions. This makes them harder to learn. In BASIC you can write your first program almost instantly!



EDUCATION

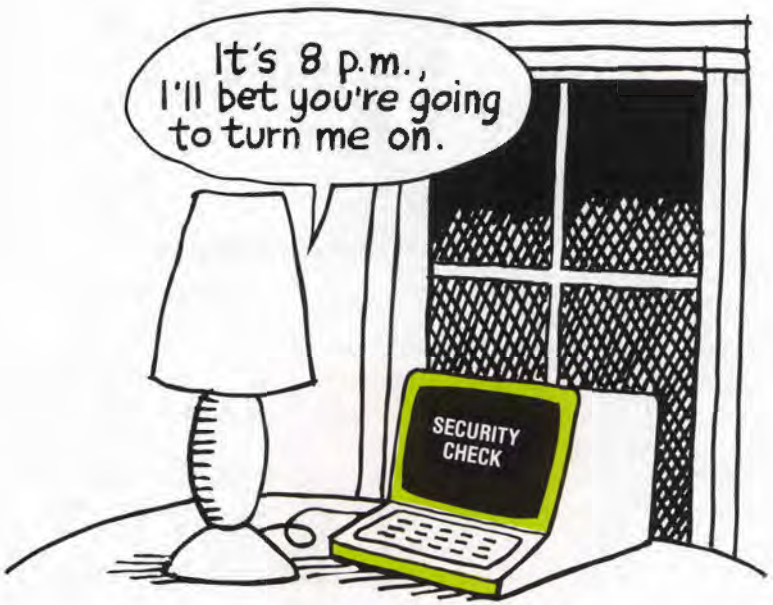
Scientific Applications

Computers programmed in the BASIC language have been used in scientific and laboratory environments for quite some time. Traditionally, BASIC programs are used in scientific and lab applications to perform complicated mathematical functions in a specific logical sequence to produce solutions to equations that would take years to solve by hand. BASIC programs can model the natural laws of hundreds of physical processes, and the model can be changed and studied by simply changing the program.

In what are called "real time" applications, computers programmed in BASIC can monitor, sense, and control external events such as the regulating of temperatures, the weighing of materials, or the movement of mechanical devices. Such applications usually involve some sort of "hardware interface,"

a device that converts signals from the "real world" to a form that the computer can deal with. You don't really need to understand the interface device to use it. As an example, a BASIC program can be designed to measure and keep track of the rotational velocity of a wind anemometer (a spinning device that measures wind speed) if a hardware interface is attached to it that produces pulses of electricity each time the anemometer turns. The computer program can count these pulses at predetermined intervals throughout the day, it can scale them to represent wind speed in miles per hour or feet per second, and store the values inside the memory of the computer for later analysis. What is more, the program can be made to work in such a way that each time the measurement is made it takes only a few seconds of computer time and leaves the computer free to do other things.

On the other hand, a BASIC program can be made to control external events rather than simply to monitor them. Lamps can be turned on and off at predefined intervals throughout the day. Valves can be opened and closed based on some internal or external event. Motors can be turned in particular directions for specific periods of time. In fact, just about anything you can imagine that contains some kind of switch or electric power source can be controlled by the computer. Again a hardware interface device is needed but in some cases the device may be extremely simple, such as a single relay or a single lamp. You can see that the computer is a universal kind of "programmed regulator." You can use it to regulate the flow of water to your



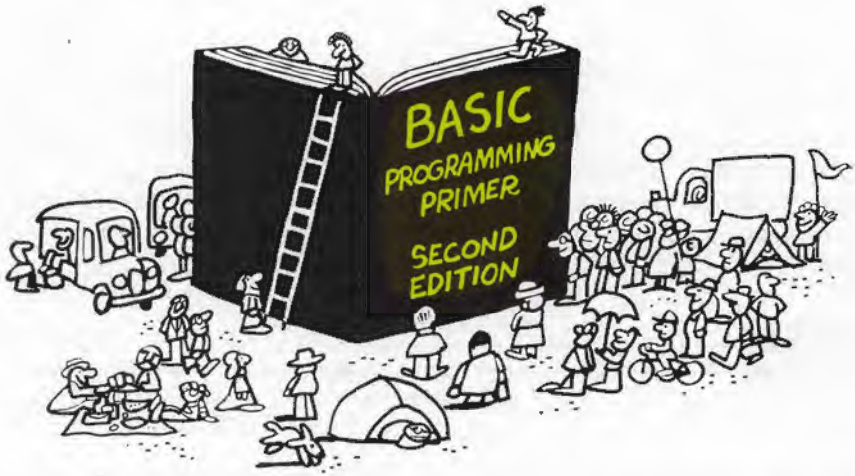
SCIENTIFIC

house plants, or to regulate the flow of chemicals to an industrial process cycle.

We have seen that these four BASIC applications span an enormous range of possibilities, and BASICally the sky's the limit. In the next sections we will begin to explore how to program in BASIC.

GETTING STARTED

Let us assume that you are sitting down at the computer. In front of you is the computer keyboard. It looks very much like a standard typewriter keyboard. Above the keyboard is the screen of the computer. The screen is called a cathode ray tube, or crt for short. The crt is where all the information appears. A crt takes the place of typewriter paper.



If you are new to this game of computers you'll have to make sure that a couple of things are in order before you start:

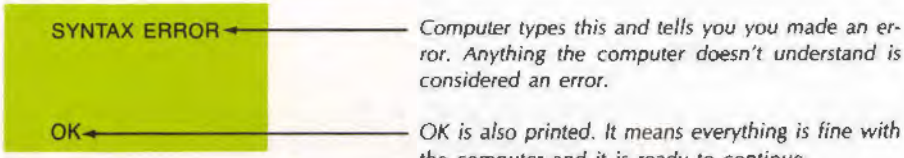
1. Is the computer turned on (or, as they say, is the computer "up and running")?
2. Is BASIC ready to go and inside your computer? In some computers you have to "call" or "load" or somehow "initialize" BASIC. The instructions for doing this can usually be found inside the manual that comes with the computer.

Let us say your computer is up and running some version of BASIC. Now to get the feel of the way a computer works we can simply type in something to the computer and see what happens. Remember, BASIC is in the computer now. Suppose you type:

```
HI COMPUTER
```

When you finish typing, press the Carriage Return key. Sometimes this key is labeled RETURN, CR, or ENTER.

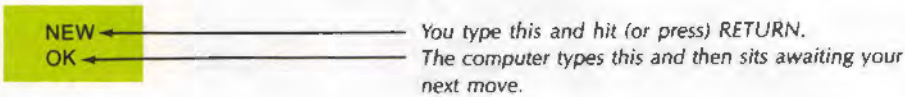
What happens now is that the computer will type out an error message in response to what you typed. It might look like this:



The computer typed SYNTAX ERROR because it didn't understand what we wanted it to do. The point is that computers have not yet been designed that understand plain English. So to make the computer do something we use a computer "language," like BASIC, to enter our instructions to the computer. The word OK means just that everything is all right, and the computer is ready for more work.

Understand that a SYNTAX ERROR doesn't mean you have done any harm to the computer. The computer simply ignores you until you type in something it understands, something in the vocabulary of the BASIC language.

Now let us get serious. Usually we start a session in BASIC by typing in the word:



Now any previous instructions that were in the computer memory have been "cleared" or "scratched" and the computer is ready to accept NEW instructions. The memory is the place where all the work in the computer occurs. It is where the BASIC program is kept. The memory can be erased by typing NEW so do not type NEW unless you wish to erase your program. Some computers use SCR or READY to clear out programs.

YOUR FIRST PROGRAM

Let us begin by typing in something the computer language can understand. Suppose we typed:

```
10 PRINT "ISN'T THIS NEAT"
```

When you are done, hit the RETURN key.

Now type in the word RUN and then press the RETURN key again. Here is what happens:

```
RUN
ISN'T THIS NEAT

OK
```

You type RUN and then press RETURN.
The computer PRINTs this and then says OK, which means "I'm done RUN-ning your program."

Congratulations are now in order. You have just "entered" or typed in a "computer program" and caused the computer to RUN or "execute" your program. The program is kind of small and doesn't do anything great, but nonetheless, it is a program.

What we have done here is tell the computer to PRINT what was inside the quotation marks. Our program was the statement 10 PRINT "ISN'T THIS NEAT." We caused the program to execute once by typing RUN. The computer responded by typing ISN'T THIS NEAT and then OK to tell us it was done. The word OK is printed out after every normal execution of a BASIC program. Abnormal programs causes SYNTAX ERROR to be printed.

LINE NUMBERS

Line numbers are used so the computer can tell the order in which it is supposed to execute "statements." When you tell the computer to RUN a program, it starts by following or executing the statement or instruction with the smallest "line number." The instruction you gave to the computer is called a PRINT statement. When you typed RUN and hit RETURN, the computer responded by printing the information between the quotation marks. The information between the quotation marks is called a "string."

Here is what the pieces of the print statement are called:

Line number *String to be printed*

```
10 PRINT "ISN'T THIS NEAT"
```

Quotation marks tell the computer where the string begins and ends.

In a BASIC program, an "indirect" statement such as

```
10 PRINT "ISN'T THIS NEAT"
```

always begins with a line number. A line number may be any positive number between 0 and 65529 although on some computers line numbers are

limited to between 1 and 9999 or 0 and 32767. Thus to make it possible for your program to run on many different versions of BASIC use line numbers between 1 and 9999. In our example we used 10 as the line number.

In BASIC, "direct" statements such as

```
NEW or RUN
```

are not part of the "program and therefore do not require line numbers. Most of the statements we will learn about are "indirect" statements, the kind with line numbers. Direct statements are more in the area of programming "aids" and so we will save the bulk of them for the last chapter. Sometimes direct statements are called BASIC "commands" because they immediately command the computer to do something for us, whereas with indirect statements we must wait for the computer to RUN them.

A program is made up of several statements, each with its own unique line number. The statement with the lowest line number gets executed first. When it is finished doing or executing the first statement, it goes on to the next statement in numerical order.

For an example of how line numbers work, type in this program (remember to type NEW since this is a new program) and then RUN it:

```
NEW
OK
10 PRINT "OPEN"
20 PRINT "THE"
30 PRINT "DOOR"
RUN
OPEN
THE
DOOR

OK
```

Hit RETURN after the end of each statement in the program, and also after you type RUN.

First the computer executes line 10 and PRINTs OPEN. Next it goes to statement 20 and prints THE and then to the highest line number 30 and prints DOOR.

Now let us examine line numbers. First of all, to type in a program you don't have to enter in line number order. That is, you can enter line numbers in any order you want, say 30 first then 20 then 10, and the computer will put them in the proper "line number order" for you. This means we can easily insert more statements in a program already stored in the computer's memory.

Note how we numbered the statements by 10s. This clever idea makes it easy for us to add more statements between existing line numbers, i.e., up to nine more statements can be added between lines 10 and 20 and so on.

For an example of this, type in this new program. Notice how the line numbers are not entered in order:

```
NEW

OK
30 PRINT "YOU"
10 PRINT "I"
20 PRINT "LOVE"
RUN

I
LOVE
YOU

OK
```

We entered line 30 first, then line 10, and finally line 20.

When we RUN the program the lines were executed in the order 10, 20, 30.

As you can see, the computer ran the program in the order dictated by the line numbers, not the order we entered them.

LIST

To see how BASIC has rearranged the program statements we can use the direct statement:

```
LIST
```

What LIST does, is just what it says—it causes the program in the memory of the computer to be displayed on the screen. Try it:

```
LIST

10 PRINT "I"
20 PRINT "LOVE"
30 PRINT "YOU"
```

Note that LIST listed each statement in the program in line number order. You can LIST a program anytime the computer is not running a program. Don't forget to hit return after typing LIST.

To see how easy it is to add statements, type in the following (don't type NEW this time because we are adding to our current program):

```
15 PRINT "DON'T"
```

Now list the program to see what happens to statement 15:

```
LIST

10 PRINT "I"
15 PRINT "DON'T"
20 PRINT "LOVE"
30 PRINT "YOU"

RUN
I
DON'T
LOVE
YOU

OK
```

See how BASIC inserted line 15 between lines 10 and 20. Easy, huh?

See how easy it is? Removing statements from your program is just as simple. To remove a statement type the line number of that statement and then press RETURN. This removes that line from the program. For example, to remove line 30 (PRINT "YOU") from our program type:

```
30 (CR)
```

The CR stands for CARRIAGE RETURN and is just to remind you to press the key. Don't type CR!

Now if we list the program with LIST, look what has happened:

```
LIST

10 PRINT "I"
15 PRINT "DON'T"
20 PRINT "LOVE"

OK
```

RUN LINE NUMBER

Sometimes we would like to RUN the program so it starts somewhere other than at the beginning line number. This comes up when you are "debugging" or testing a program. (A "bug" is a term left over from the days of World War II and secret codes. It means that something is wrong with the way the program works. Debugging is removing bugs. No DDT is required!)

In order to make a program RUN at a specific line number we would type:

```
RUN line number
```

Where "line number" appears is the line where we want the program to start. For example, in our last program we could make it start at line 15 by typing:

```
RUN 15
DON'T
LOVE
OK
```

As you can see, the program started with line 15, then executed line 20. You can RUN at any line that actually exists in the program. But if you type RUN 50 and the line 50 does not exist, this will happen:

```
RUN 50
UNDEFINED LINE NUMBER
OK
```

Since there is no line 50 in our current program in memory the computer told us so and stopped.

STOP AND END

So far we've seen how to start a program, but what about stopping the program? In all our previous examples the program stopped after it executed the last statement in the last line number. Most BASICs require that you end all programs with the statement called END. In Microsoft BASIC the END statement is not mandatory, but is still useful.

END causes the computer to stop and return to the command level. (Remember, the command level is when the computer has printed out OK, and is ready for you to command it.) The END statement should usually be at the end of your program, but it can be used elsewhere in the program.

No matter where the END statement is located, as soon as it is executed the computer stops dead in its tracks. To see how END works, first LIST the current program in memory (or reenter it if it is not still in memory):

```
LIST
10 PRINT "I"
15 PRINT "DON'T"
20 PRINT "LOVE"
OK
```

Now add this statement (remember you simply type in the new line number followed by the new statement):

```
16 END
LIST
10 PRINT "I"
15PRINT "DON'T"
16 END
20 PRINT "LOVE"
OK


RUN
I
DON'T
OK
```

Here is the new END statement.

The computer PRINTed out lines 10, 15, and then ended at line 16. LOVE never got executed.

Notice what happened here. The computer executed line 10 and PRINTed "I," executed line 15 and PRINTed "DON'T," and then found line 16 which said "END" and so the computer ended the program. This is evident by the "OK" that was printed. In general, if you come upon a program that has been running and the last thing printed on the screen was OK, then the computer is probably done and in the command mode. In fact, some computers return to the command mode and print the words READY or DONE, or just show some kind of character like > or [. Whatever, the point is that END may appear anywhere in the program and it will cause the computer to end the program when it is executed. Normally END is at the end of the program.

There is another useful statement for terminating program execution. Enter this in place of the END statement and then RUN the program:

```
16 STOP
RUN
I
DON'T
BREAK IN 16
OK
```

What happened here is the computer found the STOP statement and, like the END statement, it stopped the computer program. However, STOP also made the computer print out "where" it stopped. BREAK IN 16 means "I stopped running your program at line 16," or in other words, the computer took a "break."

Use STOP whenever you wish the program to end prematurely. For example, during debugging you may wish the computer to stop somewhere in the middle of the program so you can isolate the part that is not working right. STOP will help you here. Or as we will see later, you can make the computer execute a STOP statement "only" if it gets into trouble.

For now keep STOP in mind as another form of END.

NUMBERS

Now we have learned that PRINT "message" causes the string enclosed by the quote marks to be printed on the screen. BASIC can also handle numbers with considerable ease. Enter this program and RUN it:

```
NEW
OK

10 PRINT "12+12"
20 PRINT 12+12
RUN

12+12
24
OK
```

This says PRINT the string "12+12."

This says PRINT the sum (+) of the numbers 12 and 12.

This is the string being PRINTed. Fantastic! BASIC computed 12+12 and PRINTed the answer 24.

Note in this program we have done something new. The statement PRINT 12+12 has NO quote marks around it. When the statement was executed it calculated 12+12 and printed out 24. Or as is said in computer talk, the computer "evaluated the expression" and PRINTed the results.

BASIC can perform the kind of arithmetic found on a standard pocket calculator. Here are some rules about BASIC arithmetic:

- To ADD on the computer use +
- To SUBTRACT on the computer use −
- To MULTIPLY on the computer use *
- To DIVIDE on the computer use /

In programming we call these four symbols (+ − * /) the BASIC "arithmetic operators." The operators operate on numbers. Try typing in this program:

```
NEW
OK

10 PRINT 12+12
20 PRINT 12-12
30 PRINT 12*12
40 PRINT 12/12
RUN
 24
 0
 144
 1

OK
```

There are several things to notice about this program. First BASIC printed out the results of four expressions, one after the other. Note that the numbers are printed with a space in front of each. This can be seen by the fact that the answers are shifted one space to the right of the RUN command and the word OK. This space contains the "sign" of the number, but because the numbers are all positive, the + sign is not printed. The only time the sign of the number actually gets printed is when the number is negative. For example, add this statement and RUN the program:

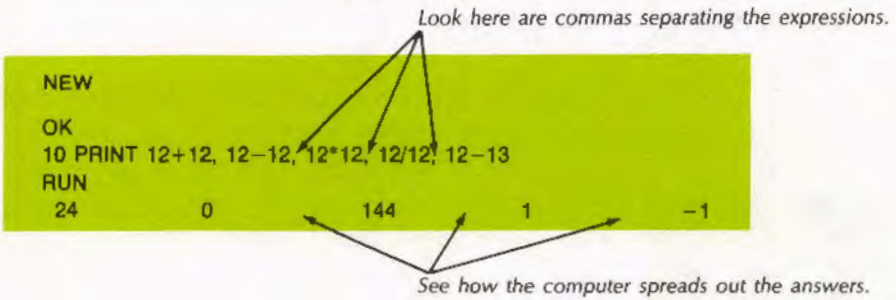
```
50 PRINT 12-13
RUN

 24
 0
 144
 1
-1
```

Here the answer to PRINT 12 − 13 is −1 and the − sign got printed. The rule is: if the number is positive (greater than one) then a space is made for an "invisible" plus (+) sign. On the other hand if the number is negative (less than one) then a minus (−) sign is printed in front of the number.

PRINT WITH COMMAS

Let us take our knowledge a little further with PRINT and calculations. Try typing in this program and RUN it:



This time we entered five expressions all in a single PRINT statement, using "commas" to separate them. You can see from the RUN of the program that the commas caused the results to be spread across the screen with lots of space being inserted between the answers.

Commas make our answers look pretty by spreading them across the page neatly and precisely. The computer takes care of the spacing for us. Take a whack at this program, type it in, and RUN it.

```
NEW

OK
10 PRINT
20 PRINT "TIMES SEVEN TABLE"
30 PRINT 7*1, 7*2, 7*3, 7*4, 7*5, 7*6, 7*7, 7*8, 7*9
RUN
TIMES SEVEN TABLE
   7          14          21          28          35
  42          49          56          63
OK
```

There are several new things to notice about this program and the results that were printed. As you can see, using commas to separate items of expressions in a single PRINT statement gives you five "columns," "places," or "print zones" across the line.

Also when we exceed the number of columns in a line, the computer simply overflows the additional information onto the columns on the next line.

On most computers a "line" is defined as having a fixed number of "character positions," where a character is any number, letter, symbol, or single space. Some computer terminals have 80 character lines, some have 64, some 40, and so on. In any case, you can consult your manual to find out what is true for your computer.

In Microsoft BASIC on an 80 column terminal a line contains 72 character positions, so that is the value we will use in this book.

Now a little detail about the five columns, or "print zones." Since there are 72 character positions and five columns, each column is exactly 14 character positions wide. BASIC calls the first character position 0 and the last 71. The last two positions (70 & 71) are not used.

To get a feeling for the spacing that commas cause, here is the output of our previous program printed with a row of numbers underneath it so you can see where the commas cause the answers to be PRINTed. Read the numbers we printed up and down, like this:
2 means 20, 5 means 56, and so on.
0 6
Here is the output of the program:

```
 7          14          21          28          35
 42         49          56          63
01234567891111111111222222222233333333334444444445555555556666666667 7
          01234567890123456789012345678901234567890123456789012345678901
```

Col one Col two Col three Col four Col five

As you can see by the numbers below the output, the columns start in positions 0, 14, 28, 42, and 56. When you look at how the actual numbers get printed you can see that the number always has a leading blank space in front of it. The blank is the space for the assumed plus sign.

Note how numbers start in the left hand edge of the column, and expand to the right as the number of digits increase. In computer talk we say the numbers are "left justified."

Now we know a little about making our numeric answers look pretty. But guess what . . . commas work with strings as well as numbers.

Type in this new program:

```
NEW

OK
10 PRINT "JOHN","HARRY","JOE","FRANK","BILL"
RUN
JOHN          HARRY         JOE           FRANK         BILL

OK
```

See how the computer spread out the strings. Like numbers, the strings begin in successive columns. The first word starts in print position 0, the next at 14, and so on.

Does this ability to columnize suggest anything to you? How about using strings with commas to label the columns and then use expressions in a PRINT statement with commas so the values come out under the desired string labels. Try this program:

```
NEW

OK
10 PRINT
20 PRINT "RATE=$11 PER HOUR"
30 PRINT "$/HR","$ PER DAY","$ PER WEEK","$ PER MONTH","$ PER YEAR"
40 PRINT 11, 11*8, 11*40, 11*40*4, 11*40*52
RUN

RATE=$11 PER HOUR
$/HR          $ PER DAY     $ PER WEEK    $ PER MONTH   $ PER YEAR
 11            88            440           1760          22880

OK
```

Pretty neat but not very useful yet. This does give us an idea of how we can "columnize" the results of our programs. Notice statement 10 (10 PRINT). This makes the computer move down one line and puts space between the word RUN and the beginning of the program output. If you wish more spacing simply add more PRINT statements, like this:

```
11 PRINT
12 PRINT
   .
   .
  etc
```

Line 20 (20 PRINT "RATE=$11 PER HOUR") causes the computer to print out the message about the amount of the hourly rate used in the rest of the program. Line 30 prints out the titles or labels for the columns. Line 40 evaluates the five expressions based on the $11 hourly rate and prints out the results under the proper columns. That's it!

Double Commas

You can use all the commas you want in a PRINT statement and each will simply cause the computer to move a column to the right before displaying. Try this:

```
10 PRINT, , "I FOUND IT!"
RUN
                              I FOUND IT!
OK
```

Or how about this:

```
10 PRINT , "GOOD", , "BYE"
RUN
          GOOD                BYE
OK
```

PRINT WITH SEMICOLONS

Commas are used to put our output in precise predefined columns. The semicolon has a different but similar use. Type in and RUN this new program:

```
NEW

OK
10 PRINT "LARRY";"MOE";"CURLY JOE"
RUN
LARRYMOECURLY JOE
OK
```

Here we typed in three famous names with semicolons between them instead of commas.

The results show the computer simply printed them with no intervening spaces.

As you can see in our program, the semicolon was used instead of the comma. Instead of printing the results in columns, the computer PRINTed the three names as one string of characters. Try changing the program like this:

```
10 PRINT " LARRY"; " MOE"; " CURLY JOE"
RUN
  LARRY    MOE    CURLY JOE
OK
```

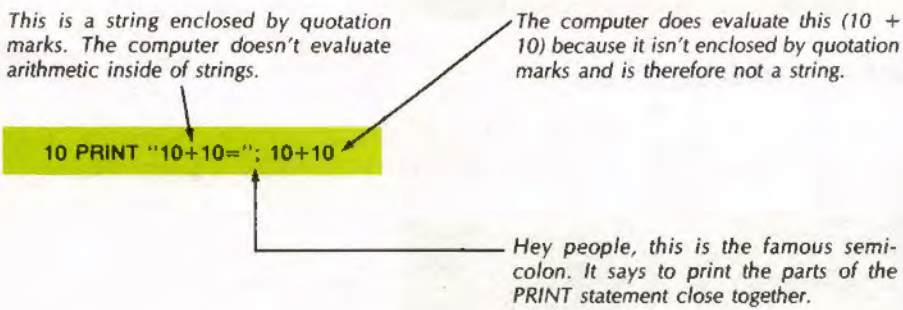
In this example we changed line 10 so that each name contains three blank spaces before the name. Since the semicolon forces the output to be squeezed together, the blank spaces are counted as characters and separate the individual names so we can read them more easily now.

We can mix strings with numbers. Try typing in and RUNning this program:

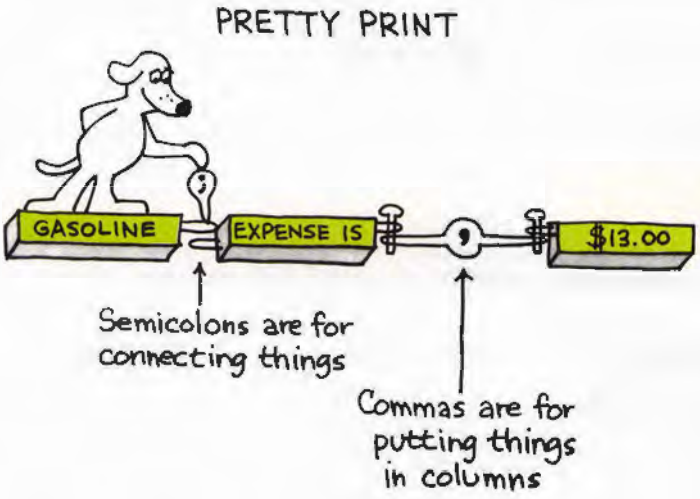
```
NEW

OK
10 PRINT "10+10="; 10+10
RUN
10+10=20
OK
```

Can you figure out how this works? Here is a diagram of the statement:



That is enough about semicolons for now. Throughout the book we will see how they can come to the rescue to make our programming life "more better."



THE DIRECT MODE

So far we have been using BASIC in the indirect or program mode. That is, we enter a statement with a line number, then type RUN to execute the program statement. Another way to make statements work is to just type the statement in without a line number. Then it will do what you typed immediately after you hit the RETURN key. We call this the immediate or direct mode. Try this example:

```
PRINT "HI IM A COMPUTER"
HI IM A COMPUTER
OK
```

Notice there is no line number. When we hit RETURN the string HI IM A COMPUTER printed immediately; note we didn't type RUN.

Now try this:

```
PRINT 2*2*2*2
 16
OK
```

Again we entered the statement without a line number. BASIC understood this to mean "execute the expression 2*2*2*2 immediately and display the result."

We can use the direct mode often in our programming. A good example of its use is to see how the BASIC arithmetic operators work together in figuring out an expression.

FIGURING IN BASIC

Try this program in the "direct mode":

```
PRINT 3+3*4
 15
OK
```

Direct mode has no line number.

Now try this slightly different version of the same expression:

```
PRINT (3+3)*4
 24
OK
```

There are a few things to note here. In the first statement, the computer multiplied before it added, even though the + sign came before the * sign. In the second statement we used left and right parentheses to make the addition happen before we multiply. We say multiplication has precedence over addition and subtraction. Parentheses override these precedences of BASIC. Try this:

```
PRINT 3+3/2-1
 3.5
OK
```

and then this:

```
PRINT (3+3)/2-1
 2
OK
```

Like multiplication, division (/) occurs before addition or subtraction. Here is a final example:

```
PRINT 4*4/2/2
 4
OK
```

Here the operators are * and /. These operators have "equal" precedence, and in this case BASIC evaluates the expression from left to right. In our

example BASIC multiples 4*4, divides by 2 to get 8 and divides by 2 again to get 4. That's it!

EXPONENTIATION

There is a special arithmetic operator in BASIC for raising a number to a power. Try these two programs in the direct mode:

```
PRINT 2*2*2*2*2
 32
OK

PRINT 2∧5
 32
OK
```

The two statements do the same thing. The symbol is called the circumflex character. On some computers it is an "up-arrow" symbol. As you can see in the second example, it means multiply a number by itself as many times as specified by the number after the symbol. Now try this:

```
PRINT 3∧3
 27
OK
```

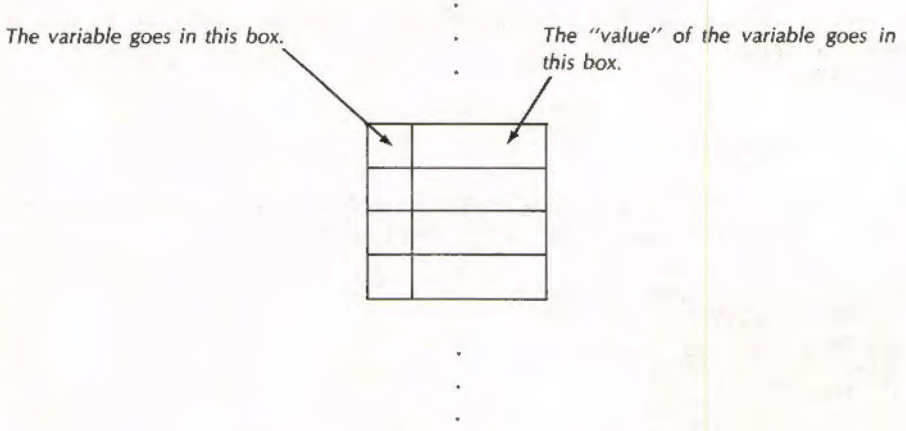
Obviously this is the same as 3*3*3, or 27. We call the operation caused by the operator "exponentiation," or raising a number to a power. In this last example we say 3 raised to the third power. Here is 4 raised to the second power:

```
PRINT 4∧2
 16
OK
```

We will see more of the exponentiation operator later.

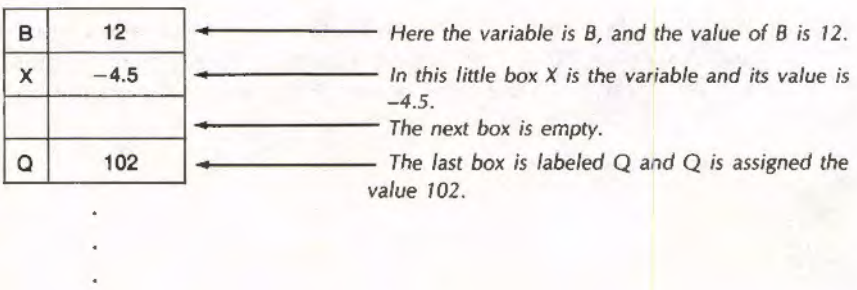
LITTLE VARIABLE BOXES

Imagine for a moment that way down inside the computer are stacks of tiny boxes. Each box has two compartments. The right side of the box can hold a number called the "value of the variable." Things look like this:



The dots here and above mean the memory continues, and this is just four of many compartments.

Take a look at the little boxes below. Some of the boxes have the variable labels filled in, and with values "assigned" to the variables. We have LET variable B have the value 12. That means the value 12 is placed in the box called B, or in simple terms B=12. Variable X is assigned the value of −4.5, or X=−4.5. And similarly the variable Q is assigned the value of 102 in the last box.



Now we will learn a way to get the computer to fill in the little boxes, that is, to assign values to variables.

Try this program:

```
10 LET A=64
20 PRINT A
```

```
RUN
 64
OK
```

When we ran the program a little box was set up by the LET statement (line 10) and the statement assigned, or "Let," the variable A have the value 64. Now the box looks like this:

A	64

Change statement 10 to this:

```
10 LET A=128
RUN
 128
OK
```

Now see by changing the value assigned to the variable A to 128 we also changed the value in the little box and when we went PRINT A the new value of 128 got printed. The box looks like this now:

A	128

You can have one value in the box at a time. The new value replaces the old value.

Let us try a little program that shows off how LET is used. Enter and RUN this program:

```
NEW

OK
10 LET A=5
20 LET B=10
30 LET C=15
40 LET D=(A+B+C)/3
50 PRINT "THE AVERAGE IS"; D
RUN
THE AVERAGE IS 10
OK
```

Here we used three LET statements to assign values to the three variables A, B, and C. Statement 40 added them together and divided the sum by 3 to obtain the average and then the LET statement assigned the result to the variable D. Statement 50 printed out the message THE AVERAGE IS and printed the value assigned to D. The boxes look like this before line 40 is executed:

A	5
B	10
C	15

Remember these are just four of the many boxes we can use for storing variables and values.

After we calculate the average and a value for the variable D, things look like the following example:

A	5
B	10
C	15
D	10

Hey, look at this, here is our new variable D and its calculated value 10!

As you can see D has its own little box now and the value we calculated and assigned to D is in the box (10).

Here is another practice using LET:

```
10 LET X=2
20 LET Y=10
30 LET Z=X∧Y
40 PRINT "X", "Y", "X∧Y"
50 PRINT X, Y, Z
RUN
X          Y          X∧Y
 2          10          1024
OK
```

Here we used the LET statement to create and assign values to the variables X and Y. Statement 30 (LET Z=X∧Y) multiplies X by itself Y times. In this example since X equals 2 and Y equals 10, the value of X∧Y is:

*2*2*2*2*2*2*2*2*2*2 or 1024

Can you see why LET Z=X∧Y makes things easier for us?

Here is another fact to know about variables. Sometimes in our programs we wish to assign several variables a special value. We can do it like this:

```
10 LET P=3.14159
20 LET C=P
30 LET D=P
40 LET E=P
50 PRINT C, D, E
RUN
 3.14159        3.14159        3.14159
OK
```

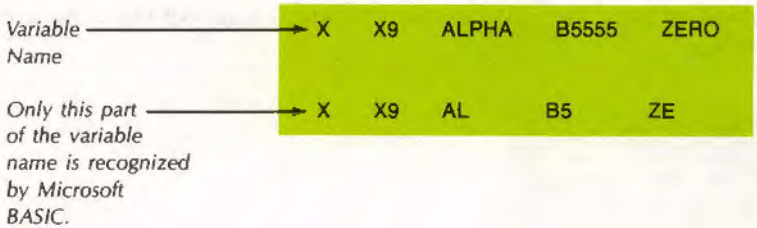
Here we are assigning the same value to three variables, and the value we are assigning is itself a variable. In this the value is 3.14159 (which is pi).

VARIABLE NAMES

Thus far we used single letters to stand for variables. BASIC "lets" you use more than a single letter. Some BASICs allow a single letter and a single number for the numeric variable.* (We say numeric because, as you may have wondered, there are variables for words too called string variables—which will be discussed later.) For example, these are legal numeric variables:

```
A     A1     Q     Z9     I5     B0
```

Other BASICs, including the Microsoft BASIC we are using here, allow any length variable up to 255 characters, although only the first two characters have any significance to the computer. The first character must always be a letter and the second can be a letter or a number. The rest of the characters in the name can be letters or numbers, but the BASIC doesn't use them at all. Here are some legal numeric variables in Microsoft BASIC:



*As defined by ANSI, American National Standard for the Programming Language Minimal BASIC, X3.60-1978, American National Standards Institute, New York.

To be sure about what is allowed in your version of BASIC, consult your BASIC manual. If you want your programs to work on all versions of BASIC, stick with single characters, or, at most, a letter and a number for variable names. Longer names will help you tell what the program does and how it works but not all BASICs allow them.

Compare the two programs. Which is easier to understand?

Version #1: Single letter variables

```
10 LET A=25
20 LET B=78
30 LET C=55
40 LET D=(A+B+C)/3
50 PRINT "THE CLASS AVERAGE IS";D
```

Version #2: Multiple letter variables

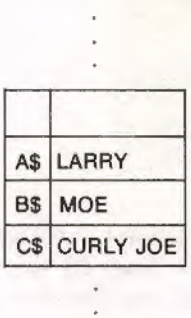
```
10 LET AL=25
20 LET BILL=78
30 LET CATHY=55
40 LET AVERAGE=(AL+BILL+CATHY)/3
50 PRINT "THE CLASS AVERAGE IS"; AVERAGE
```

STRING VARIABLES

As you may have already suspected there is more to variables than just putting numbers in boxes. Instead of a number, a variable box can contain a string of letters or characters. This type of variable is called a string variable. In order for BASIC to know we are dealing with a string variable and not a numeric variable, the label or variable name ends with a $; for example, X$ or A1$. The string that is being assigned to the string variable is enclosed by quotes, just as in the PRINT statement. Remember our program that printed out the names of three famous comedians? Here is how we could do it with string variables:

```
10 LET A$="LARRY"
20 LET B$="MOE"
30 LET C$="CURLY JOE"
40 PRINT A$; B$; C$
RUN
LARRYMOECURLY JOE
OK
```

Just as in the case of numeric variables, string variables can be imagined as being held in little boxes like this:



The only thing wrong with our program is that there is no separation between the three names. Can you think of a remedy? Here are two answers:

```
40 PRINT A$, B$, C$

RUN
LARRY     MOE     CURLY JOE
```

Simple, huh? Just changed the semicolons to commas.

This works better:

```
40 PRINT A$; "    ";B$; "    AND    ";C$
RUN
LARRY    MOE    AND    CURLY JOE
```

In this trick we printed out A$ followed by two spaces, followed by B$, followed by two spaces, the word AND, two more spaces, and then C$. Whew! In computer talk you are imbedding your variables within strings—a creative approach to displaying results.

Concatenation

Believe it or not, strings can be joined together as one happy unit with the plus (+) symbol. No, this doesn't mean addition; it is just another way of saying join two things together. Probably the inventors of BASIC could think of no better symbol than +. In computer talk we call joining of two strings concatenation (pronounced con-cat-ten-a-shon). Here's an example:

```
10 LET A$="EVERYBODY "
20 LET B$="NEEDS "
30 LET C$="SOMEBODY "
40 LET D$=A$+B$+C$
50 PRINT D$
RUN
EVERYBODY NEEDS SOMEBODY
OK
```

What happened is that we defined three separate strings to three separate string variables and then in line 40 we "concatenated" them with A$+B$+C$ and called this D$ and PRINTed it out. Neat, huh!

INPUT

There is another way of putting labels in little boxes and assigning them values. The INPUT statement allows us a way to get values and variables into the computer after the program is started by RUN. Try this example:

```
NEW

OK
10 PRINT "ENTER YOUR NAME"
20 INPUT N$
30 PRINT
40 PRINT "YOUR NAME IS "; N$
RUN
ENTER YOUR NAME
? []
```

The computer is sitting and waiting for some input from you.

(The [ ] shows where the cursor is.)

Notice the question mark. This was caused by the INPUT statement which says "INPUT something." In this case that something is a string of characters because the variable after the INPUT is a string variable. (Some BASICs require a semicolon after the INPUT, like this: INPUT ;N$.)

The computer is waiting for you to INPUT so it can put something in the little box inside the computer marked N$. You must respond by typing in a name (or any string) and then hitting the RETURN key, like this:

```
ENTER YOUR NAME
$ GYRO GEARLOOSE

YOUR NAME IS GYRO GEARLOOSE
OK
```

Here the computer made N$="GYRO GEARLOOSE" and we didn't even need quote marks like most strings do. Our little box looks like this:

N$	GYRO GEARLOOSE

The only time we need quotes around our string INPUTs is when there is a comma in the string or when we want to have spaces at the end of a string, like this:

```
RUN
ENTER YOUR NAME
? "HUN WING SING CHOY "

YOUR NAME IS HUN WING SING CHOY
OK
```

This last space made the use of quotes necessary

or

```
RUN
ENTER YOUR NAME
? "EINSTEIN, ALBERT"

YOUR NAME IS EINSTEIN, ALBERT
OK
```

Note in our program we used a PRINT statement to tell us what the INPUT statement wanted, in this case a string name. The question mark by itself doesn't tell us much. We can eliminate the PRINT statement in the program by putting it inside the INPUT statement like this:

```
10 INPUT "ENTER YOUR NAME"; N$
20 PRINT
30 PRINT "YOUR NAME IS "; N$

RUN
ENTER YOUR NAME? HARCOMB MUD

YOUR NAME IS HARCOMB MUD
OK
```

Note the space and the semicolon. Not all BASICs require the semicolon.

We call this an INPUT statement with a "Prompt" string. It makes our programming simpler and eliminates an extra PRINT statement. You can also INPUT numbers into little boxes with INPUT, like this enhancement of our previous program:

```
10 INPUT "ENTER YOUR NAME"; N$
20 INPUT "ENTER YOUR AGE IN YEARS"; Y
30 LET D=Y*365.25
40 PRINT "YOU'RE"; D; "DAYS OLD"; N$
RUN
ENTER YOUR NAME? MITCH
ENTER YOUR AGE IN YEARS? 32
YOU'RE 11656 DAYS OLD MITCH
OK
```

Here we used INPUT in line 20 to ask the user to enter his age in years. This number is then assigned to the variable Y. Next in line 30 we multiply Y times the number of days in a year (365.25) and assign this value to D. In line 40 we PRINT out the value of D, a short string and the name of the person (N$). Here is how the boxes look:



Sometimes you may wish to input several variables at once. As it is right now, we must use separate INPUT statements to do this. Here's another way:

```
10 INPUT "ENTER YOUR NAME, A COMMA, AND YOUR AGE IN YEARS"; N$, Y
20 LET D=Y*365.25
30 PRINT "YOU'RE"; D; "DAYS OLD"; N$
RUN
ENTER YOUR NAME, A COMMA, AND YOUR AGE IN YEARS? PLUTO,4
YOU'RE 1461 DAYS OLD PLUTO
OK
```

Notice the comma that separates the two inputs (PLUTO and 4).

In this example you can see that BASIC allows several variables to be defined and assigned values in a single INPUT statement. We use commas to separate them in the INPUT statement itself, and in the answer we give to the computer. You can also use the carriage return key instead of the comma to separate the information:

```
10 INPUT "ENTER YOUR NAME AND AGE"; N$,A
RUN
ENTER YOUR NAME AND AGE? BRUCE(CR)
 ?1
YOU'RE 365.25 DAYS OLD BRUCE
OK
```

CR means we pressed RETURN.

Here we typed in a name (BRUCE) and when we hit carriage return the next? appeared to tell us BASIC is ready for the next variable (Y). This approach may be used better for an application like this:

```
NEW

OK
10 PRINT "ENTER FIVE NUMBERS"
20 INPUT A, B, C, D, E
30 LET F=(A+B+C+D+E)/5
40 PRINT "THE AVERAGE OF THE NUMBERS IS"; F
RUN
ENTER FIVE NUMBERS
? 10
?? 20
?? 30
?? 40
?? 50
THE AVERAGE OF THE NUMBERS IS 30
OK
```

Notice the two?? are printed after we enter the first value of the INPUT statement. This is how BASIC tells us that an INPUT statement is requesting more input.

What about making mistakes using INPUT, what happens? Try running the previous program but answer the questions as shown below.

```
RUN
ENTER FIVE NUMBERS
? HI
? REDO FROM START
? 1
?? " $½&
?? REDO FROM START
?? 2
?? 3
?? 4
?? 5
```

Here we entered a string for a numeric variable and the computer barfed and said REDO FROM START which means type the number in again.

Sometimes when BASIC is requesting input for an INPUT statement you may not wish to answer the question. To abort the INPUT and return BASIC to the command mode (direct mode) type CONTROL C (C). This will terminate the INPUT and BASIC will print BREAK and then OK, like this:

```
RUN
ENTER FIVE NUMBERS
?
BREAK
OK
```

Although you can't see it, we pressed the CONTROL key and the C key here.

A simple way to remember the rule for input is: you can enter numbers into a string variable but you can't enter letters into a numeric variable. This is because a string variable will accept any character. An example would be a phone number or street address . . . you would not want to add these numbers together so you would use a string variable.

TAB

TAB is a keyword always used with PRINT. Its purpose is to cause the "cursor" to move horizontally a specific number of positions before printing the information in the PRINT statement. The form of TAB is:

```
PRINT TAB(var) var,var, . . .
```

You are already familiar with the PRINT statement. When TAB appears in a PRINT statement the variables (var,var, . . .) after TAB are printed out on the terminal starting at the horizontal column position specified by the value of variable *var* in the TAB parentheses. It is assumed that the variable *var* is greater than zero and less than the width of your screen. In some BASICs if the value of *var* exceeds the screen width, the information is printed on the next line until the number of column positions specified by *var* has been reached. In essence, TAB is like the TAB on a typewriter; it moves the print head to the right by so many spaces. In the case of the computer, we have a special control of the movement of the cursor (which is analogous to the print head on the typewriter) through the variable *var*. It is also assumed that *var* is an "integer" variable because there is no such thing as a fractional cursor position.

As an example, consider that you wish to print out a heading for a program so that it is centered on the terminal or screen page. There are two ways to do this. One way is to simply place the correct number of blanks inside the quote marks around the string you wish to display. The blanks precede the message

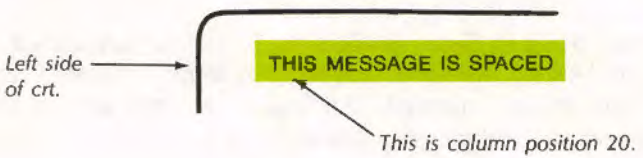
and cause the message to be printed to the right of the first column position, like this:

```
PRINT "                    THIS MESSAGE IS SPACED"
```

This is 20 spaces in front of the message.

This is the message.

When the PRINT statement is executed in BASIC the message appears like this:



We can accomplish the same thing and shorten our statement by using TAB and making the value of the variable *var* 20, like this:

```
PRINT TAB(20) "THIS MESSAGE IS SPACED"
```

This statement makes the message begin at column position 20. If we wanted to move the message frequently or test its spacing at various positions, we could do this:

```
10 INPUT I
20 PRINT TAB(I) "THIS MESSAGE IS SPACED"
RUN
?10
          THIS MESSAGE IS SPACED
RUN
?5
      THIS MESSAGE IS SPACED
RUN
?1
 THIS MESSAGE IS SPACED
```

The program allows us to INPUT a TAB value and see how the string THIS MESSAGE IS SPACED gets printed out. Note that when we entered a TAB value of 1, the string began one space over from the left side of the screen. This shows us that there is a column position "0," and the first column is column zero.

The use of TAB is not restricted to string messages only. We can force a TAB on any item in a PRINT statement. For example, consider the case where you wish to print out several numbers in column positions that are different than the ones set by the comma:

```
100 A=10
101 B=20
102 C=30
110 PRINT TAB(5) A;" ";B;" ";C
RUN
          10   20   30
```

Note that the entire group of three numbers (A, B, and C) was shifted five spaces to the right by TAB(5). The semicolon and the string " " were used to place two spaces between each number. TAB affects the group of characters A, B, and C, but the spacing relationship between the values of A, B, and C remain fixed.

Let us step back for a moment and take a look at what we've learned. Computers operate using three main functions:

1. They INPUT information.
2. They PROCESS the information.
3. They OUTPUT the results of the process.

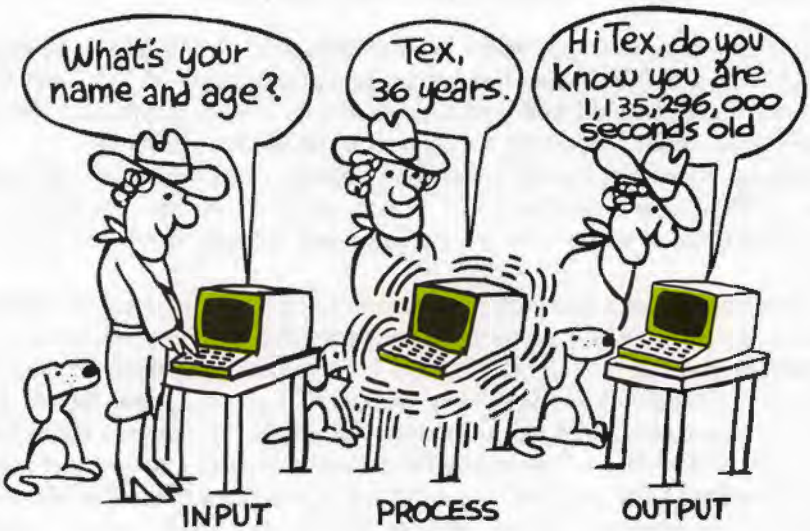
All three of these functions have their own variations which provide you with a huge variety of program techniques.

BASIC NO-NOs

As we mentioned earlier, some BASICs allow variables to be given names longer than one or two characters. For example, we can call a variable XVALUE in some BASICs and only X in others. This longer name option is really nice except for a subtle danger. The danger is that you cannot name a variable with a name that is already used in your version of the BASIC language for some statement command or function.

For example, if you call a variable RUN5, the computer will think you mean to "run" or execute line 5. Or if you have a statement: FORMAT=10 when BASIC executes it, it will think that this is a FOR . . . NEXT loop because it sees the letters FOR; thus, you will get an error of some kind.

In order to avoid these problems, your BASIC manual should come with a list of "reserved words." These are all the words that BASIC reserves for its own purposes. You may not use any of these words in your variable names. They may not appear anywhere *inside* a variable name either.



Reserved Words

Here is a list of reserved words for the industrial standard Microsoft Altair BASIC

AND	IF	READ
AUTO	INPUT	REM
CLEAR	LET	RENUM
CONSOLE	LINE	RESET
CONT	LIST	RESUME
DATA	LLIST	RETURN
DEF	LOAD	RUN
DEFDBL	LPRINT	SPACE
DEFINT	MOD	SPC
DEFSNG	NAME	STOP
DEFSTR	NEW	STRING$
DELETE	NEXT	SWAP
DIM	NOT	SYSTEM
EDIT	NULL	TO
ELSE	ON	TAB
END	OR	THEN
FN	OUT	USR
FOR	PEEK	WAIT
GOSUB	POKE	
GOTO	PRINT	

PUTTING IT ALL TOGETHER—THE LOAN PROGRAM

We now have, with just three simple keywords of the BASIC language, INPUT, LET, and PRINT, and five arithmetic operators (+ − * / ^), the ability to perform complicated and useful programs by simply combining them in some logical order, according to a predetermined plan we devise.

Although there are literally hundreds of things you could think up to program in BASIC, the ones that BASIC is most suited for involve operations that would be difficult to do with a calculator and almost impossible to do by hand.

As an example of a practical program for which you could use the computer consider the everyday problem of borrowing money. Everyone at one time or another has had to take out a loan from a bank, credit union, or other lending institution. Have you ever wondered how the bank figures your monthly payment? Or did you ever wish you could check to see if the figure was right, or if you could determine the payment on your own without having to call the bank? The program we are about to introduce pulls together everything we have learned so far to calculate the total monthly payment of a loan. The program illustrates the three laws of computing, and the way that a complex formula can be simplified into a LET statement.



The way banks figure out your monthly payment for a loan is to use special tables of figures that chart out the various payment schedules for certain loan amounts, spread over a specified number of years or months, and a certain interest rate. There is also a special formula that can be used to compute the loan payment for any combination of loan amount, term or length, and interest.

The special formula is called an *amortization loan* formula. Amortization refers to the fact that the monthly payment is based on interest computed each month on the unpaid balance. A *simple loan* formula on the other hand would just take a percentage of the total borrowed amount, add it to the amount borrowed, and divide the total by the number of months of the loan, called the term.

The amortization formula looks like this when written down on paper:

$$\text{Mon. Payment} = \frac{\text{Loan Amount} \times \text{Interest}}{1 - (1 + \text{Interest})^{-\text{months}}}$$

What the formula says is take the total amount borrowed, the interest rate and the number of months the loan is spread over and perform the calculations to obtain the monthly payment. The interest rate is the monthly interest, not the yearly, and is obtained by dividing the yearly interest rate (called annual rate) by the number of months in a year (12). Months is the total number of months of payments, and is found by multiplying the number of years of the term by the number of months in a year (12).

Now we can see that the program must require three steps to work. We must INPUT three parameters to the program, PROCESS them to compute a result called the monthly payment, and OUTPUT the result for us to read.

Thus, to start the program requires we INPUT three parameters and assign them variable names that sort of explain what each stands for in the formula:

The amount of the loan (L)
The annual interest rate (R)
The term of the loan in years (Y)

This can be accomplished quite easily with the INPUT statement:

```
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
```

We now must convert the annual interest rate (R) to a monthly interest rate which we will call I, like this:

```
1210 LET I=R/12
```

Next we must convert the term of the loan in years (Y) to the number of months this represents which we will call M, like this:

```
1230 LET M =Y*12
```

Now we are ready to convert the loan formula into a BASIC statement using LET. Notice the formula uses exponentiation, raising the product of interest times loan amount to a power determined by the length of the loan in months. This is how it is done:

```
1310 LET A=(I*L)/(1-(I+1)^-M)
```

And finally we must print out the result (A) like this:

```
1320 PRINT "THE MONTHLY PAYMENT IS"
1330 PRINT A
```

Do not let the formula scare you. You do not really have to understand why it works, but you do have to make sure it is entered into the LET statement properly so the program executes it just like it should. The parentheses are used to make the formula operations occur in the proper order. Notice for a moment that the values in the formula cause a small number (I) to be raised to a large power (M). Doing this on the calculator would be quite a complex process, but in BASIC it is easy.

Now we are ready to list the entire program and RUN it.

```
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1210 LET I=R/12
1230 LET M=Y*12
1310 LET A=(I*L)/(1-(I+1)^-M)
1320 PRINT "THE MONTHLY PAYMENT IS"
1330 PRINT A
1400 END


RUN
ENTER THE AMOUNT OF THE LOAN? 5000.00
ENTER THE ANNUAL INTEREST RATE? .09
ENTER THE TERM IN YEARS? 4
THE MONTHLY PAYMENT IS
124.427
```

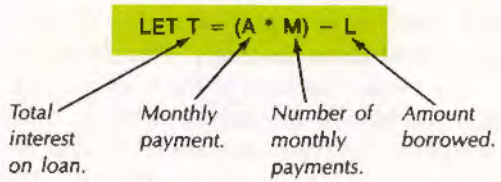
In this first case we entered the amount we wanted to borrow for an automobile loan ($5000.00), the interest rate annually was 9% (.09), and the term of the loan was 4 years. The program quickly figured out that the loan monthly payment was 124 dollars and 43 cents.

We could use the program to figure the monthly payment on a home we want to borrow for. For example, if you want to buy a house that costs $100,000 and the bank will finance 80% of this ($80,000) for 30 years at 11% then the user would type in the following:

```
ENTER THE AMOUNT OF THE LOAN? 80000.00
ENTER THE ANNUAL INTEREST RATE? .11
ENTER THE TERM IN YEARS? 30
THE MONTHLY PAYMENT IS
761.86
```

EXPANDING A PROGRAM

There is an old saying that the closer a program is to being done, the more things you will discover to add to it. Adding more functions to a BASIC program is actually quite a simple task, because BASIC was made to be expandable. All we have to do is figure out where the new statements go and type them in. For example, consider an enhancement of the Loan program that computes the actual total interest we are paying on the amount borrowed. In order to arrive at this figure we must first find the total amount we are to pay by multiplying the monthly payment (A) times the number of months of payments (M). Then we subtract the actual principal we borrowed (L) from this, and the result is the amount of the loan going toward the interest. In the form of a LET statement we can compute the interest total like this (calling the total interest (T)):



Once the amount of interest (T) is computed we can print it out with:

```
1420 PRINT "TOTAL INTEREST ON LOAN IS"
1430 PRINT T
```

Now our program looks like this:

```
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
```

```
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1210 LET I = R/12
1230 LET M = Y*12
1310 LET A = (I*L)/(1-(I+1)^-M)
1320 PRINT "THE MONTHLY PAYMENT IS"
1330 PRINT A
1410 LET T = (A*M)-L
1420 PRINT "TOTAL INTEREST ON LOAN IS"
1430 PRINT T
1500 END


RUN
ENTER THE AMOUNT OF THE LOAN? 5000.00
ENTER THE ANNUAL INTEREST RATE? .09
ENTER THE TERM IN YEARS? 4
THE MONTHLY PAYMENT IS
124.427
TOTAL INTEREST ON LOAN IS
972.512
```

See how easy it was to expand the program? Later in the book we will learn how to even further enhance the program; for now what is important is that you see that a program can be added to by simply inserting new logical statements in the proper locations.

Notice that $972.512 in interest represents almost 20% of the $5000 we originally borrowed. This clearly shows how an 11% amortized loan actually means the interest is much more than 11% of $5000, and a larger amount is paid in interest than would seem necessary. This shows how loan companies make their profits. The longer the term of the loan and the larger the amount borrowed, the larger the interest payment will be for a fixed interest.

On the practical side, the interest total can be used as a number to subtract from your income tax (if you itemize on the long form). Without the program you couldn't easily figure out ahead of time how much of the loan was going toward the interest.

DOCUMENTING YOUR PROGRAMS (REM)

Suppose you wrote a program in 1978 and then in 1981 came back to it and attempted to modify its design. Because few of us have photographic memories what we have to do is "relearn" how the program works. Although BASIC is a simple language, one cannot just look at a program listing and immediately tell how it works. Furthermore the program lacks information which points out what it actually accomplishes when it is run and, unless you

are really sharp, deciding what statements perform what functions can be very difficult.

In order to alleviate this problem, BASIC provides a special statement, called REM, which stands for REMark. What REM does is allows us to add program "documentation" information, which appears in the program listing but has no effect on the way the program works. REMs are called nonexecutable statements because when the computer encounters one in the program flow it simply ignores it and goes to the next line.

The form of the REM statement is like this:

```
line REM a message goes here about the program . . .
```

For example, in our Loan program, we can use a REM statement right away to tell us the name of the program, like this:

```
1000 REM LOAN AMORTIZATION PROGRAM
```

Now when we come back to the program at a later date and read the listing, the first REM statement will tell us the name and purpose of the program. The REM statement is therefore used throughout the program to explain to you, or to an outsider, what the statements in the program actually do to complete the total job.

Keep in mind that the REM statements do not get executed by the computer. When one is encountered the computer knows that it is to ignore the statement and execute the next statement after the REM. But when we list the program with the LIST command the information in the REM statement will be displayed for us.

One thing you will certainly discover as your programming experience grows is that "the other guy" never seems to document his program well enough for you to understand. In fact, the secret of good programming is super documentation, but most programmers are weak in these areas. A program without adequate REM statements is like a machine with no operator's manual. If you get stuck or if you want to change something, you must reinvent what the original maker had in mind when it was designed.

So to be an especially accomplished programmer, strive to keep your programs as full of REMs as you can. The only negative thing about REMs is that they consume extra memory without adding any computing power to the program. In the case of excessively long programs the additional memory consumption by the REMs can be a problem. So at worst you should at least place REMs in the critical parts of the body of the program.

As an example of good documentation, here is our loan program rewritten to include REM statements. See if you think it is easier to understand now:

```
   0 REM PROGRAM BY MITCHELL WAITE AND MICHAEL PARDEE
   1 REM WRITTEN: JULY 1978
   2 REM USED AS EXAMPLE IN THE BOOK: BASIC PRIMER
   3 REM ALL RIGHTS RESERVED WORLDWIDE
1000 REM -LOAN AMORTIZATION PROGRAM
1010 REM -INPUT SECTION
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1200 REM -CHANGE ANNUAL RATE TO MONTHLY RATE
1210 LET I = R/12
1220 REM -CHANGE YEARS TO MONTHS
1230 LET M = Y*12
1300 REM -COMPUTE THE MONTHLY PAYMENT AND PRINT
1310 LET A = (I*L)/(1-(I+1)^-M)
1320 PRINT "THE MONTHLY PAYMENT IS "; A
1400 REM -COMPUTE THE TOTAL INTEREST AND PRINT
1410 LET T = (A*M)-L
1420 PRINT "THE TOTAL INTEREST IS "; T
1500 END
```

As you can see we added several REM statements and to make them easy to find with the eye, a preceding dash (–) was added to the message in the REM to flag that this is a nonexecutable statement that is for the purpose of documentation only.

Now we have a good idea of the three most simple BASIC keywords—INPUT, LET, and PRINT—as well as some special purpose characters for formatting our output (semicolons and commas). We also know a little about numbers and variables. Now we are ready to take a bigger step into BASIC and learn how programs can be increased greatly in power and flexibility by some new BASIC keywords called program control statements.

REVIEW QUESTIONS

1. Which of the following BASIC statements are correct?

   (a) 10 PRINT"HELLO"
   (b) –5 PRINT"HOW"
   (c) 69000 PRINT"ARE"
   (d) 100.5 PRINT"YOU"

2. Which of these command-mode BASIC statements are correct? (On some computers you may use commas instead of dashes.)

   (a) LIST
   (b) LIST 10
   (c) LIST 50–100
   (d) LIST –1000
   (e) LIST 500–

3. If you want to print a book title with one space between each word, which of the following programs will do it?

   (a)
   ```
   10 A$="BASIC"
   20 B$="PROGRAMMING"
   30 C$="PRIMER"
   40 PRINT A$,B$,C$
   ```

   (b)
   ```
   10 A$="BASIC"
   20 B$="PROGRAMMING"
   30 C$="PRIMER"
   40 PRINT A$;B$;C$;
   ```

4. Which of the following commands will NOT result in BASIC printing the number 2?

   (a) PRINT 2+2/2
   (b) PRINT 2/(2/2)
   (c) PRINT 2*2/2
   (d) PRINT (2+2)/2

5. Which of the following are legal variable names?

   (a) FL
   (b) F2
   (c) 2F
   (d) FLAG
   (e) F*

6. Is this a valid BASIC statement?

   ```
   10 PRINT"THE TITLE IS "PRIMER," ISN'T IT?"
   ```

7. If the statement

   ```
   40 Q=3
   ```

   appears in your program, but you have not yet RUN the program, does Q equal 3?

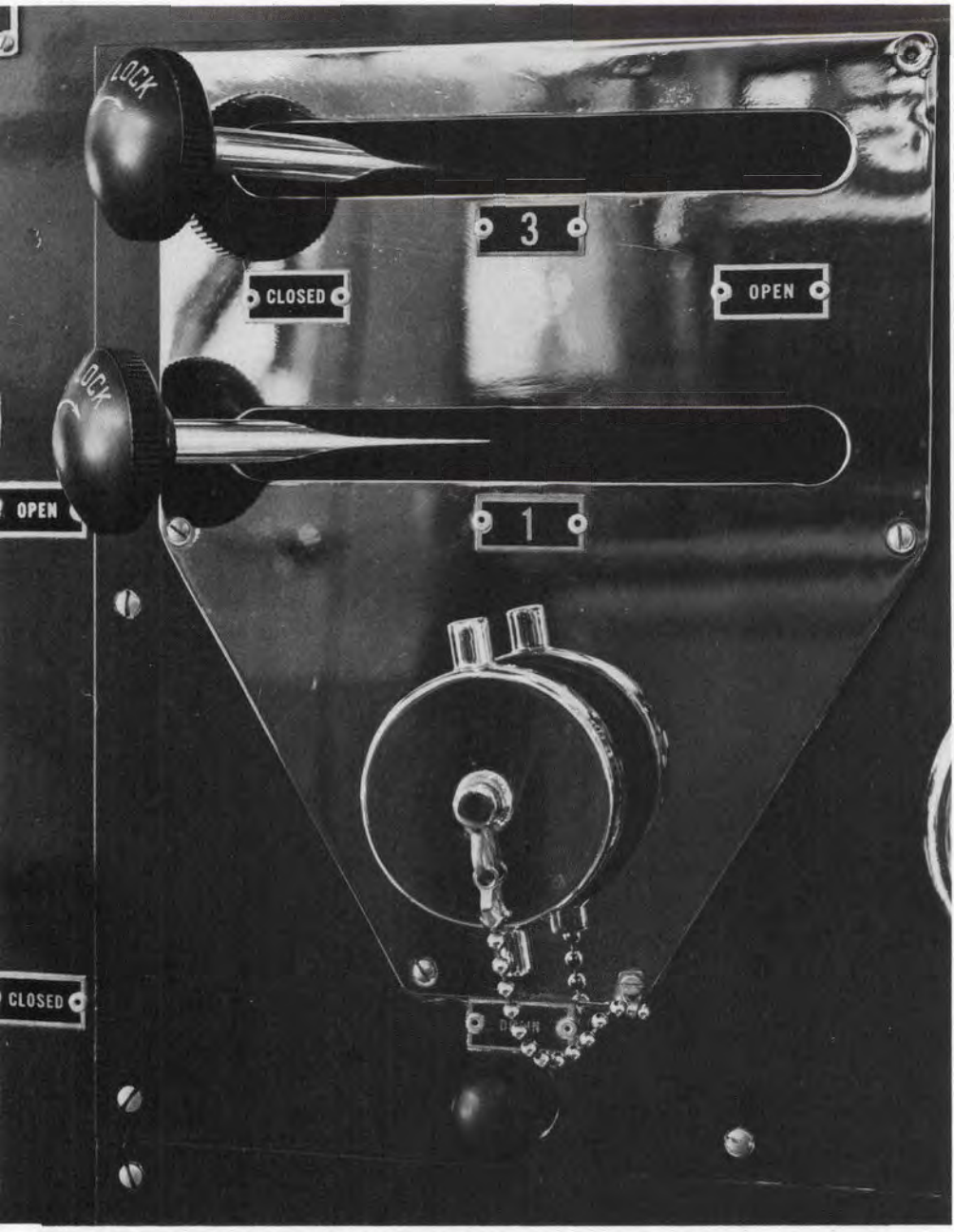
8. Is this a valid statement? Explain your answer.

   10 3=Q

9. Which of the following statements will cause an error message?

   (a) 10 LET IF=2.4
   (b) 20 LET A$="IF"
   (c) 30 LET IF$="A"
   (d) 40 LET IF$=2.4

10. Exercise: Write a program that will let you type in 5 words and will then print them out as a sentence, but in the reverse order to the way you typed them in. That is, the last word you typed in will be printed first, and so on.


LOCK
3
CLOSED
OPEN
LOCK
1
OPEN
CLOSED

CHAPTER

2

Program Control

Very often, we are confronted with certain situations which require us to make a decision of some sort. We have to take a good look at all of the information which we have available, consider all of the alternatives, and then most likely go right ahead and make an emotional decision. Yes, objectivity is a very difficult quality to maintain during any decision making process that affects us personally. We are involved in decision making during a good part of our daily lives. With the great degree of "free will" which we humans have, and the tremendous variety of choices, we are almost constantly making decisions of one sort or another.

The computer differs from man in this area in that it is never emotionally involved in any decision. As long as sufficient information is available to the computer, it can evaluate it and make a completely unbiased decision. This decision making power allows the computer to follow alternate instructions based upon the value of a number, or the sequence of letters that are entered. In this section we will discuss the BASIC statements which provide the program with a means of doing a variety of different things, all based upon the evaluation of some information. This concept is what really makes the computer a powerful tool. It is not "locked in" to doing only one thing over and over again. Rather, it can change its processing according to the nature of the information.

THE DECISION MAKING STATEMENTS

There are several BASIC statements that come under the category of decision making statements. Each has a specific use and function. In many cases these statements can be combined in such a way as to offer even greater decision making power to the computer.

FOR...NEXT, the Programmed Loop

In most programming applications there will arise a need to do the same steps over and over again. This process is known in computer talk as "iteration." It is also referred to quite often as "looping," since it usually involves

setting up the program so that it will loop back and repeat itself. If a program is written in such a way, it might easily get stuck in the loop and keep on repeating itself forever. For this reason, there is usually some means of keeping track of just how many times the program is supposed to repeat itself. In BASIC language, all of these considerations are taken care of by the FOR and NEXT statements.

The FOR statement sets up the beginning of the loop and determines how many times it will be repeated. The NEXT statement marks the end of the loop, and causes the program to go back and repeat itself if necessary. To use a simple example: Let us suppose that what we want our program to do is to INPUT four numbers from the keyboard, add them together and PRINT the total on the screen. Now, the four numbers could easily be entered through the use of just one INPUT statement but, for the sake of the example, let us say that the numbers are to be entered separately. The program looks like this:

```
10 LET T = 0
20 FOR I = 1 to 4
30 INPUT N
40 LET T = T + N
50 NEXT I
60 PRINT T
70 END
```

This is the beginning of a FOR...NEXT loop from 1 to 4.

NEXT I represents the end of the loop.

This PRINTS the total of the four numbers you INPUT.

Now, let us take a closer look at this program, statement by statement. First of all, at statement 10 we begin by setting the variable T equal to zero. This is the variable that we will use to accumulate the total of the four numbers to be INPUT. At statement 20 we see the FOR statement. It controls how many times the following statements will be performed. It specifies that the "index" I will be initialized to the value of 1, and the loop will continue until I has reached a value of 4. Statement 30 lets us INPUT a number into the variable called N. At statement 40, another LET statement is used to replace the current value of T (the running total), with the sum of T and the number N. Now, at statement 50, the NEXT I statement defines the end of the loop, and causes the value of I to be increased by 1. At this point, if I is greater than 4, the next statement in the program will be performed. That would be the PRINT statement at 60 which causes the total of the four numbers stored in the variable T to be PRINTed on the screen. If however, I is less than or equal to 4 after the NEXT I statement has increased I by one, the program will return to the first statement following the FOR statement where the loop begins. Another number will be read from the keyboard into the variable N. It will then be added to the current value of the total stored in the variable T, and so on.

There is a more powerful version of the FOR statement which allows the increment that is added to the "index" I to be some value other than one. This advanced FOR statement looks like this:




```
FOR I = 1 to 50 STEP 3
```

Notice the addition of the word STEP to the statement. This statement would begin by initializing the index I to one. Then when the NEXT I statement was encountered later in the program, the value of three would be added to I instead of one. Then, as before, the value of I would be checked to see if it had exceeded the termination value of the loop, 50.

Let us suppose that we want to write a program that will add up all of the odd numbers between 0 and 100, and print the answer. The program could be done like this:

```
10 LET T = 0
20 FOR I = 1 TO 100 STEP 2
30 LET T = T + I
40 NEXT I
50 PRINT T
```

"Step 2" makes I go 1, 3, 5 ...

Quite a simple program really. First, at line 10, we set up a variable called T and initialize it to zero. The next line is the beginning of the FOR...NEXT loop which starts at 1, and goes to 100 by 2s. Each time through the loop, the value of I is added to the total being accumulated in T. Finally, when the value of I exceeds 100, the answer is printed.

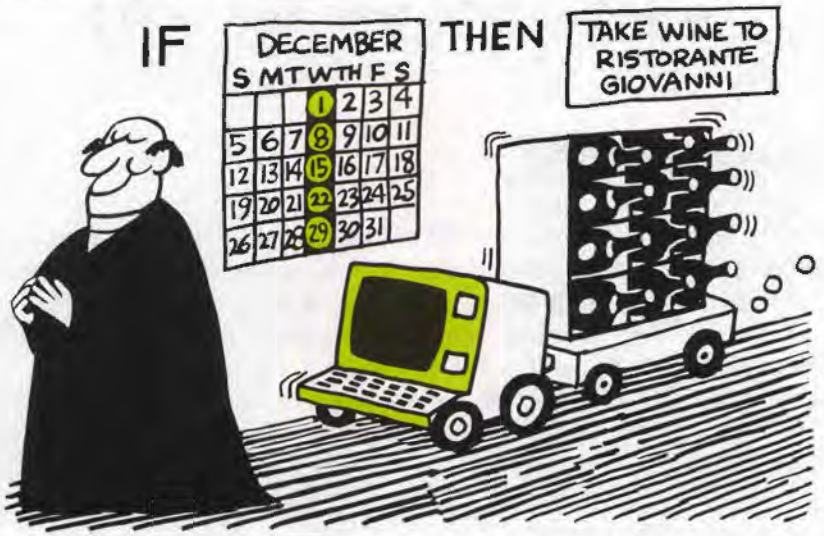
If you think about it, you will see that I will never be equal to 100. It goes, 1, 3, 5, . . . 95, 97, 99. When 2 is added to the value 99, the result is 101 which is greater than 100, and therefore causes the termination of the loop.

IF...THEN, Conditional Branches

These keywords are used to provide the BASIC language with its real decision making powers. There are many forms of the IF... THEN statement, ranging from very simple data evaluation, to quite complex combinations of conditions. To get a quick idea of what this statement is like, use a simple analogy.

IF today is Wednesday THEN take out the garbage

Here we have set up a conditional decision. If the premise is true (that it is Wednesday), then the result of the decision causes the action of taking out the garbage. Of course, the converse situation (that it is NOT Wednesday) would cause no action to be taken. It is only when the premise is true that the part following the word THEN is carried out.



IF WEDNESDAY THEN TAKE WINE

In BASIC, things work the same way. A sample of a BASIC statement using IF...THEN is the following:

```
IF A = 5 THEN PRINT "END OF JOB"
```

Here, the current value of the variable A is tested to see if it is equal to five. If it is, the program will output the message "END OF JOB." Again, if A is not equal to five, the message will not be printed. Notice that the condition being tested in this statement is the value of the variable A. This is done by using one of the arithmetic operators (=) to indicate that it is equality that is being tested. The variety of different expressions that may be included as part of the

premise is quite great. For example, the following are some other forms of the IF...THEN statement.

```
IF A+3 = B THEN LET C = 0
IF 2*B-5 = C+2 THEN PRINT "MATCH"
```

Other relational operators that may be used in the IF...THEN statement are

< less than
> greater than
<= less than or equal to
>= greater than or equal to
<> not equal to (sometimes the # symbol)

The following are some examples of these relational IF...THEN statements:

```
IF A<3 THEN GOTO 100
IF 2*B+4>20 THEN LET A = 1
If B+C+6<>0 THEN PRINT "NOT EQUAL"
```

There are variations on the "THEN" side of the IF...THEN statement as well. We have already seen that we may cause something to be printed if the premise of the IF statement is true. We may also cause other things to happen as the following examples so readily illustrate.

```
IF. . . . .THEN LET A = 125
IF. . . . .THEN LET B = A + C
IF. . . . THEN INPUT N
```

So there are many things that the program can do based upon the premise of the IF statement being true. Once again, if the premise is NOT true, the object of the statement (that part which follows the word THEN) will not be performed. Instead, the program will just proceed to the next statement in sequence.

To make this statement even more powerful, there are variations which use what are known as "logical operators" to form compound IF...THEN statements. Some examples include:

```
IF A=3 and B<4 THEN LET C=10
IF B>2 OR C<2 THEN PRINT "INVALID"
```

In the first example, the object of the IF statement will only be executed if both conditions are true. That is, "A must be equal to 3," and "B must be less

than 4." If either of these premises is not true, then the object will not be performed. In the second example, if either one of the two premises is true, then the message "INVALID" will be printed. That is, if either "B is greater than 2," or "C is less than 2," the message will be printed. We can see that by using this type of IF...THEN statement, many conditions may be tested at once, and at the same time very complex logical decisions can be made.

GOTO, Unconditional Branch

The GOTO statement is not really a decision making statement. It is used to force the program to go to some other statement and perform the program



GOTO Hades

there. It is however used most commonly in conjunction with the decision making statements and so has been included in this section. By itself, the GOTO statement will simply cause the program to "branch" to another statement. For example:

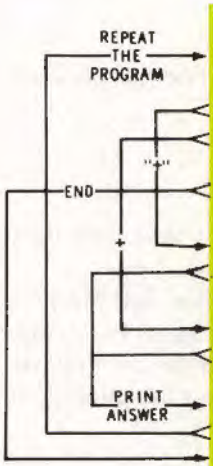
```
GOTO 1200
```

This statement would cause the program to branch to the statement bearing the number 1200 and to execute it and then continue on with the next statement after 1200. This is known as an "unconditional branch," that is, it occurs no matter what.

The GOTO statement is often used as the object of an IF...THEN statement to allow the program to be controlled by the condition of some data. For example,

```
IF A= 5 THEN GOTO 100
```

This is known as a "conditional branch" since it only occurs if the premise of the IF...THEN statement is true. If the current value of the variable A is equal to 5, the branch to statement 100 will occur; otherwise, the next statement after the IF...THEN statement will be executed. Let us use a simple example program to illustrate the use of both the IF...THEN statement and the GOTO statement. This program will INPUT two numbers, and a single character string variable that will direct the program to either "A" add the two numbers together and print out the result, or "D" divide the first number by the second number and print out the result.




```
10 REM -FIRST INPUT THE TWO NUMBERS AND THE COMMAND
20 INPUT X,Y,A$
30 REM -SEE WHICH OPERATION IS TO BE DONE
40 IF A$ = "A" THEN GOTO 90
50 IF A$ = "D" THEN GOTO 120
60 REM -IF NEITHER, END THE PROGRAM
70 GOTO 170
80 REM -ADD THE NUMBERS TOGETHER
90 LET Z = X + Y
100 GOTO 150
110 REM -DIVIDE FIRST NUMBER BY SECOND
120 LET Z = X/Y
130 GOTO 150
140 REM -PRINT THE ANSWER AND GO DO IT AGAIN
150 PRINT "THE ANSWER IS "; Z
160 GOTO 20
170 END
```

From the above example, we can see how the IF...THEN and GOTO statements can be used in conjunction to provide extensive program control.

The Endless Loop

There will be times when you are testing your new program and things just don't seem to operate properly. There are many reasons why programs don't work properly, and one of them is an "endless loop." The endless loop can be brought about in a number of ways, the simplest of which is a GOTO statement that inevitably causes the program to repeat a sequence of statements indefinitely. Something like:

```
10 LET X=1
20 GOTO 10
30 PRINT X
```

No doubt about it, this program is going to get caught up in an endless loop. The PRINT statement at line 30 will never be executed. When you tell your computer to RUN this program, it will appear to "fall asleep." The prompting "OK" will never be seen again in the program.

To provide the programmer with some alternative in these situations, most BASICs will respond to the entry on the keyboard of a "control-C sequence." This is done by holding down the key labeled "CNTL," and pressing the "C" key at the same time. Some computers have a special "BREAK" key which sends a control-C to the computer. Control-C or BREAK will cause the BASIC program to be interrupted, and the "OK" prompt returned to the screen. Also, the line at which the program was when the control-C was entered is displayed on the screen:

```
BREAK IN 20
OK
```

At this point, the programmer can examine the program to find the problem and then continue the execution of the program by typing:

```
CONT
```

Note that if the program is altered in any way (i.e., typing a new line) then CONT cannot be used to restart. Instead, RUN must be used.

In general, the control-C sequence may be used at any time that the programmer wants to interrupt a program. If a program does a lot of PRINTing, and the programmer wishes to stop the program before all of the printing has been completed, the control-C sequence may be used to abort the printing in progress and return control to BASIC.

ON...GOTO, Indexed Branching

A special form of the GOTO statement allows the programmer to specify



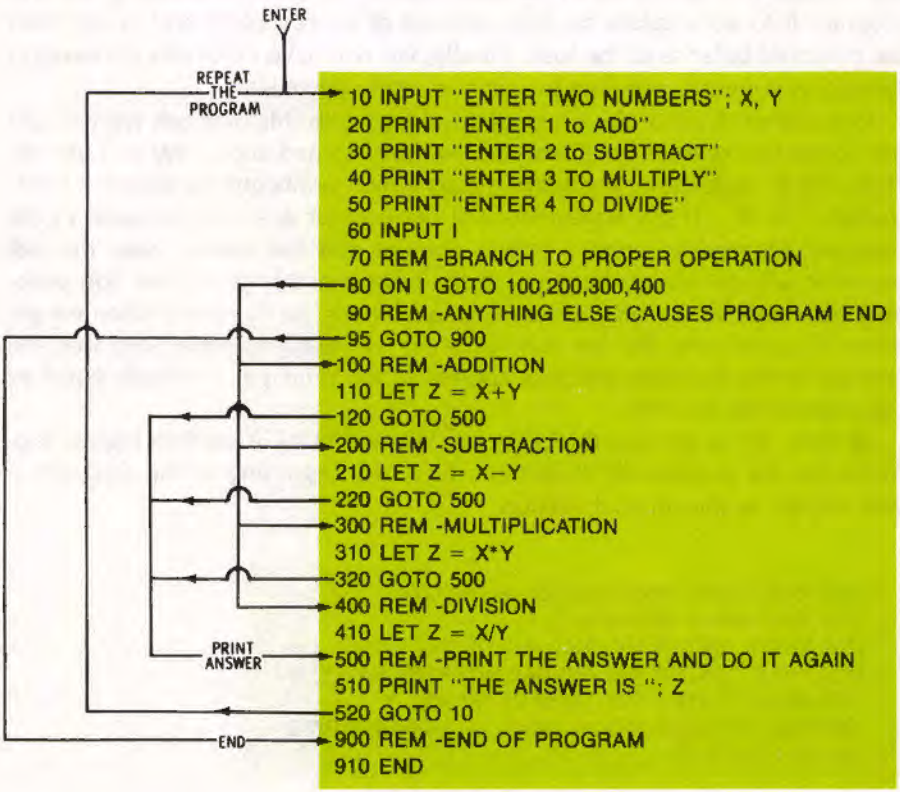
several possible destinations of the branch, and make the selection of the one to be taken based upon the value of a particular variable. The form of this type of GOTO statement is:

```
10 ON I GOTO 100, 200, 300, 400, 500 etc.
```

Here, the variable I is used as the index for the ON...GOTO statement. If the value of I is 1, then the GOTO will cause a branch to line 100. If I is equal to 2, then the branch will be to line 200, and so on.

Notice that the value of the index to the ON...GOTO statement must be an integer, a whole number. It must also be greater than zero, and not greater than the total number of lines listed after the GOTO. If the index is zero, or greater than the total number of lines in the list, the next statement after the ON...GOTO statement will be executed. If it is negative, an error will result.

The ON...GOTO statement lends itself very well to making organized program branches such as are encountered when a function or operation is selected by the programmer by entering a number through the keyboard to the program. The program can then use the number as the index for the ON...GOTO statement, as in the following example:



The ON...GOTO statement located at line 80 of this program serves to direct the program to the proper sequence of statements based upon the selected operation entered at line 60.

ENHANCING THE LOAN PROGRAM

Remember the Loan program we used in Chapter 1 to illustrate the use of the three laws of computing? Well, now we are going to modify that program to provide some additonal information. As we recall, the program required as INPUT the following:

The amount of the Loan L
The annual interest rate R
The term of the loan in years Y

By using the formula programmed into the LET statement, we were able to compute the monthly payment A. Now, we will enhance the program so that *for each payment*, it will compute the amount that is paid as interest, and also the amount that is applied to the principal of the loan. In addition, we will program it to accumulate the total amount of interest paid, and to compute the principal balance of the loan. Finally, we will have all of this information printed in columns, one line for each monthly payment.

To accomplish all of these wonderful things with this program we will use the "program control" statements that we just learned about. We will use the FOR...NEXT statements to control a loop which prints out the monthly information. The IF...THEN statements will be used for a special purpose in the program. Depending upon the loan amount and the interest rate, the last payment will not always be equal to the other monthly payments. You probably have noticed this on your own personal loans. So therefore, when we get down to computing the last payment, we will want to make sure that the amount of the payment which is applied to the principal is exactly equal to the outstanding balance.

So then, let us get into this program. We will break it up into logical segments for the purpose of illustration. First, the beginning of the program is very similar to the original version.

```
1000 REM -LOAN AMORTIZATION PROGRAM
1010 REM -INPUT SECTION
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1200 REM -CHANGE ANNUAL RATE TO MONTHLY RATE
1210 LET I = R/12
```

```
1220 REM -CHANGE YEARS TO MONTHS
1230 LET M = Y*12
1300 REM -COMPUTE THE MONTHLY PAYMENT ANO PRINT
1310 LET A = (I*L)/((1-(I+1)^-M))
1320 PRINT "THE MONTHLY PAYMENT IS $"; A
```

The next thing that the program must do is prepare to print the amortization schedule. We will need to establish a variable for the current loan balance (B) after a payment has been made. We will also need a variable to accumulate the total interest paid (T). To dress up the output, we will also print some column headings.

```
1400 REM -INITIALIZE FOR AMORTIZATION SCHEDULE
1410 LET B = L
1420 LET T = 0
1430 PRINT "NUM", "INTEREST", "PRINCIPAL", "PRIN BAL", "TOT INT"
```

Now comes the payment loop. We will use the variable J as the "index" to the loop, and the number of monthly payments M as the "terminator" of the loop. For each payment, we will compute the amount which is paid as interest I1, and the amount which is applied to the principal P. Also, we will subtract the amount applied to the principal from the loan balance B, and accumulate the total interest paid T.

```
1500 REM -DO THE LOOP FOR THE NUMBER OF PAYMENTS
1510 FOR J = 1 to M
1520 REM -COMPUTE THE INTEREST
1530 LET I1 = B*I
1540 REM -COMPUTE THE PRINCIPAL
1550 LET P = A-I1
1560 REM -FORCE OUT THE LAST PAYMENT
1570 IF J = M THEN LET P = B
1600 REM -UPDATE THE LOAN BALANCE
1610 LET B = B-P
1620 REM -UPDATE THE TOTAL INTEREST
1630 LET T = T+I1
```

Now, using the automatic column function of the PRINT statement, we will print all of the information for the monthly payment and then end the loop.

```
1700 REM -PRINT THE MONTHLY PAYMENT DETAIL
1710 PRINT J,I1,P,B,T
1720 NEXT J
```

At this point, we will add some statements that will provide the option either to go back to the beginning of the program to process another loan, or to end the program. See if you can figure out how this works.

```
1800 REM -SEE IF DONE
1810 INPUT "PROCESS ANOTHER LOAN (YES/NO)"; A$
1820 IF A$ = "YES" THEN GOTO 1100
9999 END
```

Notice that we are using a "string variable" here. Remember that this is denoted by the "$" following the variable name. So, what is happening here is that we are inputting a string of characters at statement 1810, and if that string happens to be the word "YES" then the program will go back to statement 1100 where the entire process starts. If the input string is not the word "YES," then the program will proceed to the next statement which is the END statement at 9999. This causes the program to terminate.

Now you have a very useful little program. You can use it to find out what your monthly payment would be on a particular loan, and also to see just how much monthly interest you will pay.

Table 2-1 represents an example of a printout for a sample loan amortization. At the top of the table, the numbers that are underscored would have to be entered by the user as the program requested the input information. In this example, we are amortizing a loan of $5000.00, at an annual interest rate of 9% (entered as a decimal fraction .09), over a term of 4 years. The program computed the monthly payment to be $124.427 and printed that out first. Then the program proceeds to print out the amortization schedule according to the format that we decided upon earlier. Notice that the column headings are printed first, followed by one line for each monthly payment. We can see that at first the amount paid as interest is relatively large, and decreased as the loan balance decreases. Likewise, the amount applied to the principal is relatively small at first, and then gets larger as the amount paid as interest decreases.

There are a few other things that we should notice at this time. One is that if we sum the amount paid as interest with the amount applied to the principal for each month, we do not always get the monthly payment, 124.427, as the answer. This is because the numbers printed here have not been rounded off after the computer performed the mathematics. For example, when the program converts the annual interest rate of 9% to a monthly interest rate, it comes out to a decimal fraction of .0075. (.09/12). Now, when the program computes the amount paid as interest on the second payment, the math actually comes out like this:

```
MONTHLY RATE × LOAN BALANCE = INTEREST AMOUNT
  .0075      ×     4913.07     =36.848025
```

Table 2-1. Example of Printout for a Loan Amortization

ENTER THE AMOUNT OF THE LOAN? 5000.00
ENTER THE ANNUAL INTEREST RATE? .09
ENTER THE TERM IN YEARS? 4
THE MONTHLY PAYMENT IS $124.427

Num.	Interest	Prin. Pay	Prin. Bal.	Int. to Date
1	37.5	86.9273	4913.07	37.5
2	36.8481	87.5792	4825.49	74.3481
3	36.1913	88.236	4737.26	110.539
4	35.5296	88.8977	4648.36	146.069
5	34.8629	89.5644	4558.8	180.932
6	34.1912	90.2361	4468.56	215.123
7	33.5145	90.9128	4377.65	248.638
8	32.8327	91.5946	4286.05	281.47
9	32.1458	92.2815	4193.77	313.616
10	31.4537	92.9736	4100.8	345.07
11	30.7565	93.6708	4007.13	375.826
12	30.054	94.3733	3912.75	405.88
13	29.3463	95.081	3817.67	435.227
14	28.6333	95.7941	3721.88	463.86
15	27.9149	96.5125	3625.37	491.775
16	27.1911	97.2362	3528.13	518.966
17	26.4619	97.9654	3430.16	545.428
18	25.7272	98.7001	3331.46	571.155
19	24.987	99.4403	3232.02	596.142
20	24.2413	100.186	3131.84	620.383
21	23.49	100.937	3030.9	643.873
22	22.7331	101.694	2929.21	666.606
23	21.9704	102.457	2826.75	688.577
24	21.2021	103.225	2723.52	709.779
25	20.428	103.999	2619.52	730.207
26	19.6481	104.779	2514.74	749.855
27	18.8623	105.565	2409.18	768.717
28	18.0707	106.357	2302.82	786.788
29	17.2732	107.154	2195.67	804.061
30	16.4696	107.958	2087.71	820.531
31	15.6602	108.767	1978.94	836.191
32	14.8445	109.583	1869.36	851.035
33	14.0228	110.404	1758.96	865.058
34	13.1949	111.232	1647.72	878.253
35	12.3608	112.066	1535.66	890.614
36	11.5205	112.907	1422.75	902.134
37	10.6739	113.753	1309.00	912.808
38	9.8209	114.606	1194.39	922.629
39	8.96162	115.466	1078.93	931.591
40	8.09581	116.331	962.594	939.687
41	7.22375	117.204	845.391	946.91
42	6.34518	118.082	727.309	953.256
43	5.45965	118.968	608.341	958.715
44	4.56831	119.859	488.482	963.283
45	3.66998	120.757	367.725	966.953
46	2.76479	121.663	246.062	969.718
47	1.85405	122.573	123.489	971.572
48	.9375	123.49	-1.09863E-03	972.51

According to the table, the interest amount on payment two is 36.8481 (to six significant digits). This apparent discrepancy is due to the accumulated error inherent in the least significant digit of the arithmetic processes of the computer.

METRIC CONVERSION PROGRAM AND THE MENU CONCEPT

Here is an example of a very useful program written in BASIC (Fig. 2-1). We have all heard of the intended adoption of the metric system of measurement by the world at large. Many manufacturers of prepared foods are printing both the anglo and metric quantities on their products. Although this transition may

```
2000 REM -METRIC CONVERSION PROGRAM
2010 REM -DISPLAY THE MENU
2020 PRINT
2030 PRINT "METRIC CONVERSION PROGRAM"
2040 PRINT
2041 PRINT "1 - INCHES TO CENTIMETERS"
2042 PRINT "2 - POUNDS TO KILOGRAMS"
2043 PRINT "3 - QUARTS TO LITERS"
2049 PRINT "9 - TO END THE PROGRAM"
2050 PRINT
2060 REM -GET THE SELECTION
2070 INPUT "ENTER YOUR SELECTION"; S
2080 REM -DECIDE WHERE TO BRANCH
2081 IF S = 1 THEN GOTO 2100
2082 IF S = 2 THEN GOTO 2200
2083 IF S = 3 THEN GOTO 2300
2089 IF S>3 OR S<1 THEN GOTO 9999
2100 REM -CONVERT INCHES TO CENTIMETERS
2110 PRINT
2120 INPUT "ENTER LENGTH IN INCHES"; I
2130 LET C = 2.54*I
2140 PRINT I; "INCHES IS EQUAL TO";C; "CENTIMETERS"
2150 GOTO 2000
2200 REM -CONVERT POUNDS TO KILOGRAMS
2210 PRINT
2220 INPUT "ENTER WEIGHT IN POUNDS "; P
2230 LET K = .4536*P
2240 PRINT P;" POUNDS IS EQUAL TO ";K;" KILOGRAMS"
2250 GOTO 2000
2300 REM -CONVERT QUARTS TO LITERS
2310 PRINT
2320 INPUT "ENTER VOLUME IN QUARTS"; Q
2330 LET L = .9463*Q
2340 PRINT Q;" QUARTS IS EQUAL TO ";L;" LITERS"
2350 GOTO 2000
9999 END
```

Fig. 2-1. Listing of Metric Conversion program.

seem to be difficult, there are many advantages to the metric system, which is always based upon the number 10. The anglo systems are all different. Length can be based upon a scale of 12 inches, or 3 feet, or 5,280 feet in a mile. The task of converting a length from yards to miles requires that two conversion factors be used. First, we must convert the length to feet, and then finally convert it to miles. The metric system, based upon ten, allows very simple conversion of lengths either measured in meters, or kilometers. All we need to do is to multiply or divide by a power of ten (10, 100, 1000, etc.) to perform the conversion. This then requires only that the decimal point be moved either to the right for multiplication, or to the left for division. For example, let us convert 500 meters to kilometers. First of all, we need to know the fact that 1 kilometer is equal to 1,000 meters. This is conveniently indicated by the prefix "kilo" meaning thousand. Therefore, to convert meters to kilometers, we simply divide by 1,000, or move the decimal point three places to the left. So, 500 meters is equal to .500 kilometers, or exactly half a kilometer.

During the world struggle to convert from the anglo system to the metric system, the major problem is that the new units do not have the inherent "sense" that the old ones did. How long is half a kilometer? Being conditioned to the odd concepts of feet, yards, and miles, we find it hard to conceive of half a kilometer. But this is only due to our being unfamiliar with the metric terms. With a little practice, we can easily adapt to the metric system. In the meantime, we can use BASIC to write a simple program that converts anglo measurements, which are now using, to metric measurements.

The program has three major sections which we will discuss separately. First, the program will provide us with a "menu" from which we can select the conversion we want. Then, the program will have to interpret our request and branch to the proper statements to affect the conversion. Lastly, the desired conversion will be accomplished, and the answer printed out. Here is the first part of the program:

```
2000 REM -METRIC CONVERSION PROGRAM
2010 REM -DISPLAY THE MENU
2020 PRINT
2030 PRINT "METRIC CONVERSION PROGRAM"
2040 PRINT
2041 PRINT "1 -INCHES TO CENTIMETERS"
2042 PRINT "2 -POUNDS TO KILOGRAMS"
2043 PRINT "3 -QUARTS TO LITERS"
2049 PRINT "9 -TO END THE PROGRAM"
2050 PRINT
2060 REM -GET THE SELECTION
2070 INPUT "ENTER YOUR SELECTION"; S
```

Note how we left room for you to add other conversion routines (4-8).

This is relatively straightforward. Using PRINT statements, the program displays the "menu" of the available conversion routines. In this example, we



only included three conversion routines, but as we will see, it would be very easy to add others. Notice that we have made provision for ending the program from the menu by entering a selection of "9." Also, notice the PRINT statements on lines 2020, 2040, and 2050. These cause a blank line to be displayed which is like double-spacing on the typewriter.

The next section of the program uses the INPUT variable S to determine which conversion formula is to be used. This is accomplished using several IF...THEN statements.

```
2080 REM -DECIDE WHERE TO BRANCH
2081 IF S = 1 THEN GOTO 2100
2082 IF S = 2 THEN GOTO 2200
2083 IF S = 3 THEN GOTO 2300
2089 IF S>3 OR S<1 THEN GOTO 9999
```

Could you replace the three IF...THENs with a single ON...GOTO?

Remember that if the premise is true in the IF statement then the result will be that the program performs the right side of the IF...THEN statement. If the premise is not true, the program proceeds on to the next statement. So, the program checks the value of S to see if it is equal to 1, 2, or 3 and performs a GOTO branch to the proper statement for the selected conversion. Notice that if S is not equal to 1, 2, or 3, the program will end up at line 2089. Here, using a compound IF...THEN statement, the program will branch to line 9999 if S is greater than 3, *or* if S is less than 0. So, if anything other than 1, 2, or 3 is entered, the program will branch to line 9999, the end of the program. In this section again, we can see that it would be very easy to add additional IF...THEN statements to select more than three conversion routines.

The next section of the program is where the conversions actually are done. There are actually three "subsections," each consisting of the BASIC state-

ments necessary to perform one of the three conversions. Let us look at the whole section at once, and notice how the three subsections are very similar.

```
2100 REM -CONVERT INCHES TO CENTIMETERS
2110 PRINT
2120 INPUT "ENTER LENGTH IN INCHES "; I
2130 LET C = 2.54*I
2140 PRINT I;" INCHES IS EQUAL TO ";C;" CENTIMETERS"
2150 GOTO 2000

2200 REM -CONVERT POUNDS TO KILOGRAMS
2210 PRINT
2220 INPUT "ENTER WEIGHT IN POUNDS "; P
2230 LET K = .4536*P
2240 PRINT P;" POUNDS IS EQUAL TO ";K;" KILOGRAMS"
2250 GOTO 2000

2300 REM -CONVERT QUARTS TO LITERS
2310 PRINT
2320 INPUT "ENTER VOLUME IN QUARTS "; Q
2330 LET L = .9463*Q
2340 PRINT Q;" QUARTS IS EQUAL TO ";L;" LITERS"
2350 GOTO 2000

9999 END
```

Notice that each subsection begins with a blank PRINT statement to effect the double-spacing. Then an INPUT statement is used to get the particular anglo quantity to be converted. Next, a LET statement is used to perform the arithmetic conversion of the anglo quantity to a metric quantity. Notice that the conversion factors are "numeric constants" built into the statements. These can be found in any metric conversion table. Then, a PRINT statement is used to print out the answer in a sentence form. Notice how the variables representing both the anglo and metric values are imbedded right into the output line. Finally, after each subsection, a GOTO statement causes the program to branch back to the beginning where the menu is displayed again. This procedure is repeated as many times as desired until a value other than 1, 2, or 3 is entered as the desired selection, at which time the program is ended.

The following is an example of how the program runs:

```
METRIC CONVERSION PROGRAM

1 - INCHES TO CENTIMETERS
2 - POUNDS TO KILOGRAMS
3 - QUARTS TO LITERS
9 - TO END THE PROGRAM
```

```
ENTER YOUR SELECTION? 1

ENTER LENGTH IN INCHES? 6
6 INCHES IS EQUAL TO 15.24 CENTIMETERS

METRIC CONVERSION PROGRAM

1 - INCHES TO CENTIMETERS
2 - POUNDS TO KILOGRAMS
3 - QUARTS TO LITERS
9 - TO END THE PROGRAM

ENTER YOUR SELECTION? 2

ENTER WEIGHT IN POUNDS? 155
155 POUNDS IS EQUAL TO 70.308 KILOGRAMS

METRIC CONVERSION PROGRAM

1 - INCHES TO CENTIMETERS
2 - POUNDS TO KILOGRAMS
3 - QUARTS TO LITERS
9 - TO END THE PROGRAM

ENTER YOUR SELECTION? 3

ENTER VOLUME IN QUARTS? 4
4 QUARTS IS EQUAL TO 3.7852 LITERS

METRIC CONVERSION PROGRAM

1 - INCHES TO CENTIMETERS
2 - POUNDS TO KILOGRAMS
3 - QUARTS TO LITERS
9 - TO END THE PROGRAM

ENTER YOUR SELECTION? 9
```

Notice that each time a conversion is completed, the program returns to display the entire menu again. Since once would probably be sufficient, we might want to modify the program so that only the message "ENTER YOUR SELECTION?" would be displayed after the first time through the program. This can easily be done by replacing all of the GOTO 2000 statements following the conversions with GOTO 2070.

In a later chapter, we will use this same application for another example of a more powerful metric converson program that will provide a method for having many more conversion routines. The important things to learn from this example are the "menu" concept, and the use of the IF...THEN statement to selectively branch to different parts of the same program.

SIMPLE FUNCTION STATEMENTS

Many programming applications require that some operations be performed frequently. There are several of these functions that can be used by many different types of programs. For this reason, there are special "key words" used in BASIC to invoke these functions. There are quite a few different functions, some for use with numeric data, and others for use with string data. At this point, we will take a look at three of the most simple numeric functions. These are easy to understand, and can be used right away.

ABS, Absolute Value Function

The ABS function is used to get the "absolute value" of a number. This is defined as the actual numeric value of the number, ignoring the sign. The most common form of this statement looks like this:

```
LET A = ABS(X)
```

Here, the variable (A) is replaced with the "absolute value" of the variable (X). For example:

```
ABS(5) = 5
ABS(13.25) = 13.25
ABS(-25) = 25
ABS(0) = 0
ABS(5-12) = 7
```

Notice that the absolute value of a number can never be negative. It is always a positive number equal to the actual numeric value of the contents within the parentheses. This can be a numeric constant, a numeric variable, or an expression of numeric terms such as:

```
LET A = ABS(12*X-6)
```

or,

```
LET B = ABS(I+J+(K/3))
```

This function is useful for finding the actual difference between two numbers without being concerned as to whether they are positive or negative.

INT, Integerize Function

The INT function is used to derive the closest whole number to a numeric value. Really, all it does (assuming the number is greater than zero) is to truncate (cut off) all the digits that are to the right of the decimal point. For example:

```
INT(25.035) = 25
INT(.95) = 0
INT(100.9999) = 100
INT(-12.2) = -12
```

So, this provides a way of getting rid of unwanted decimal positions. The common form used for the INT statement is:

```
LET A = INT(X)
```

where (X) is either a numeric constant, a numeric variable, or a numeric expression such as:

```
LET A = INT(12*B-2)
LET B = INT(.5*N)
```



TRUNCATE THAT NUMBER
INT (345.897)

One common usage of the INT statement is to allow for the rounding off of numeric variables to the nearest whole number. This is done very easily in BASIC as follows:

```
LET X = INT(X + .5)
```

For example, if;

X = 3.49999, then
X + .5 = 3.99999, so
INT (X + .5) = 3

However, if:

X = 3.50001, then
X + .5 = 4.00001, so
INT(X + .5) = 4

Using this technique, we can always guarantee that an answer that we are going to print out will be printed as a whole number.

A more accurate explanation of what INT(X) does is to say that it "returns the largest integer that is less than (or equal to) X." For positive values of X this is the same as chopping off the digits to the right of the decimal point; but for negative values of X it means chopping off the digits and then subtracting one. For example:

```
PRINT INT (-4.222)
-5
PRINT INT(-13.75)
-14
```

FIX

In some versions of BASIC there is another function, much like INT, but which chops off the digits to the right of the decimal place for both negative and positive numbers. This is the FIX function. For example:

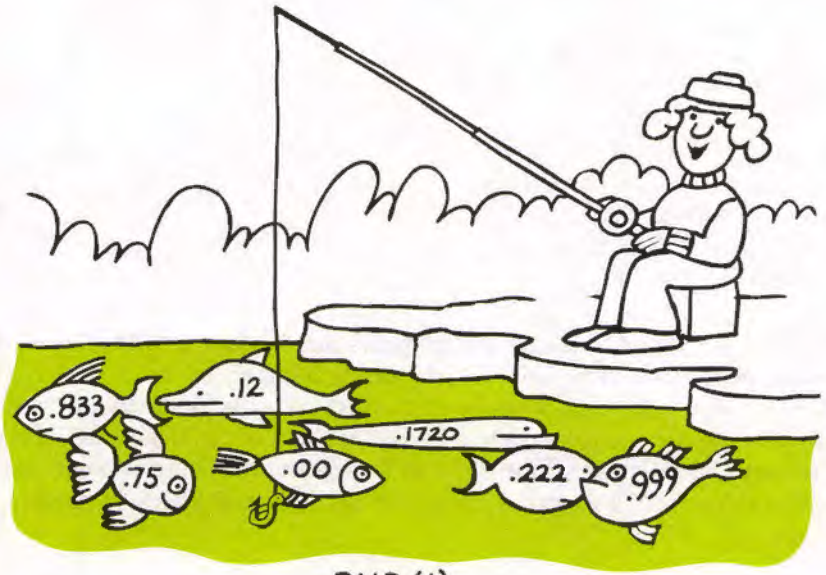
```
PRINT FIX (4.222)
4
PRINT FIX (-4.222)
-4
```

Compare this with the example of INT used with a negative number.

RND, Randomize Function

This function allows the programmer to generate a random number. This is sort of like cutting a deck of cards, or spinning the wheel of fortune, or even

throwing the dice. There is no way of knowing just what the result of the RND function will be. It will however always be returned as a number between 0 and 1. It might be .573 or .00941, or .0003 or any other number between 0 and 1. This random number can then be used to generate other numbers in



RND(1)

other ranges simply by using multiplication. For example, if we wanted to program a deck of cards being cut, we would want the result of the RND function to be a number from 1 to 52. This can be done by multiplying the number returned by the RND function by the numeric constant 52 and then adding a numeric 1 to that. The general form of the RND function is as follows:

```
100 LET A = RND(X)
```

In this statement the X serves as a "dummy" variable if it is a positive number (greater than 0); that is, no matter what value of X you use, the RND function will do the same thing—return a random number between 0 and 1. If X is 0, RND will return the same random number it did before. (In some versions of BASIC, using values of X larger than 1 will cause random numbers larger than 1 to be generated. You should consult the manual for your own version of BASIC to see exactly how it words.)

Try typing in and running the following program:

```
10 LET A = RND(1)
20 PRINT A
30 GOTO 10
RUN
0.523889
0.837612
0.39604
0.756826
```

If you let this program run, it will go on forever printing out numbers similar to these.

Randomize

This function allows you to "reseed" the random number generator used in the RND function. What does "reseed" mean? Well, normally, every time you RUN your program, the RND function will start over generating the exact same sequence of random numbers. This isn't so good if, for example, you are using RND in a game to deal you a hand of cards; you don't want to get the same hand every time you RUN the program! RANDOMIZE (RANDOM on some computers) allows you to start over with a different sequence of random numbers each time you execute it: this is called "reseeding" the random number generator. RANDOMIZE has the form

```
RANDOMIZE S
```

where S is a number between 32767 and −32768. For every different value of S you use in the RANDOMIZE statement, the RND statement will create a different sequence of random numbers. (In some versions of BASIC, using RND(X) with a negative value of X has the same effect as RANDOMIZE. In other versions, there is no variable S after the RANDOMIZE function—it makes up its own value of S each time you use it, thus creating a different seed automatically.)

Combining Functions

In order to write even more sophisticated programs, it is possible to combine these functions within one statement. For example, we can write a statement like the following:

```
LET A = INT(2*ABS(X-1))
```

or,

```
LET B = ABS(INT(X)-INT(Y))
```

Let us say that what we really want is to get a random number between 1 and 52 to simulate a deck of cards being cut. We can use a combination of the RND and INT functions like this:

```
LET C = INT(52*RND(-1))+1
```

The RND function would be done first because of the way that the parentheses are arranged, so that a random number between 0 and 1 would be generated. Then, the multiplication would follow, yielding a decimal number between 0 and 52. Next, the digits to the right of the decimal point would be stripped off using the INT function. This would result in an integer with a value ranging from 0 through 51. Finally, we will add one to the result so that we will end up with a number from 1 to 52.

We can use the same technique to simulate a throw of the dice as follows:

```
LET T = INT(6*RND(-1))+1
```

or, we could "flip a coin" like this:

```
LET F = INT(2*RND(-1))+1
```

Notice that we are always using the value of −1 to control the RND function so that we will get a new random sequence. Once a sequence has been started, it "seeds" itself, and we could change the argument of the RND function to 1 which would then continue to give us the next random number in the particular seeding sequence.

Another entertaining application of the RND function is in a little program that plays a number guessing game with you. The RND function is used to generate a number from 1 to 1,000. Then you have to guess what the number is and enter your guess through the keyboard. The program will then let you know if your guess was high or low, and give you another chance if you didn't guess it right. It also keeps track of how many chances it took for you to guess the number, and prints the total out at the end.

```
10 REM -NUMBER GUESSING GAME USING RND
20 LET N = INT(1000*RND(-1))+1
```

```
 30 LET T = 0
 40 PRINT "OK, START GUESSING"
 50 REM -INPUT A GUESS AND COUNT IT
 60 INPUT G
 70 LET T = T+1
 80 REM -CHECK FOR A HIT
 90 IF G = N THEN GOTO 160
100 REM -CHECK FOR HIGH GUESS
110 IF G>N THEN PRINT "YOUR GUESS IS HIGH"
120 REM -CHECK FOR LOW GUESS
130 IF G < N THEN PRINT "YOUR GUESS IS LOW"
140 GOTO 60
150 REM -YOU GUESSED THE NUMBER
160 PRINT "YOU GUESSED THE NUMBER IN "; T; " TRYS"
170 PRINT "DO YOU WANT TO PLAY AGAIN? (Y/N)"
180 INPUT A$
190 IF A$ = "Y" THEN GOTO 20
200 END

RUN
OK, START GUESSING
?500
YOUR GUESS IS HIGH
?250
YOUR GUESS IS LOW
?375
YOUR GUESS IS HIGH
?300
YOU GUESSED THE NUMBER IN 4 TRYS
DO YOU WANT TO PLAY AGAIN? (Y/N)
?N
OK
```

THE SUBROUTINE CONCEPT

There are many occasions during the writing of a program on which it would be desirable to "chop up" a program into smaller parts. Sometimes, it is just too difficult to write one big program that takes care of everything. Other times, there is one series of operations that must be performed several times throughout the entire program. It would be tedious to write this series into the program every place that it is needed, and the resulting program would be unnecessarily large. These are two very good reasons for the development of the "subroutine." As its name implies, a subroutine is a "subpart" of a program. It is a routine which is performed at some specified time, and does the same thing each time. The subroutine need only be "written" once, somewhere within the program, and then "called" by the program when it's execution is necessary. When the subroutine has completed its operation, it "returns" to the main program at the point right after the line where the main



GOSUB French Horn Solo

program called the subroutine. This is especially useful in situations where a simple GOTO statement just doesn't quite handle it.

Subroutines are composed of BASIC statements just like any other program. The statements all have line numbers and must follow all of the rules that apply to any program statement. The subroutine is invoked through the use of a special BASIC statement called GOSUB. The form of the GOSUB statement is as follows:

```
100 GOSUB 2000
```

Directly following the word GOSUB must be a valid line number that represents the first line of the part of the program that is to be considered the subroutine. When the GOSUB statement is executed, the program branches to the line number specified in the statement. All of the subroutine statements are then executed until another special BASIC statement known as RETURN is encountered. At that time, the program branches back to the line directly following the GOSUB statement that caused the original branch. The following is a general example of the subroutine concept.

```
100 REM -BEGINNING OF THE PROGRAM
110 (statement)
120 (statement)
130 (statement)
140 (statement)
```

```
150 GOSUB 1000

                    1000 REM -SUBROUTINE
160 (statement)     1010 (statement)
170 (statement)     1020 (statement)
180 (statement)     1030 (statement)
                    1040 RETURN
190 GOSUB 1000

200 (statement)
210 (statement)
220 END
```

This is the subroutine. Note that it would appear at the end of the program because it starts at line 1000.

Statements 100 through 140 are normal BASIC statements. At line 150 we see the first GOSUB statement which references line 1000. This is where the subroutine begins and at line 150 it is desired to branch to that subroutine and perform whatever statements are there. The subroutine is located at lines 1000 through 1040. On line 1040 we see the RETURN statement which causes the program to branch back to the line directly following the point at which the branch to the subroutine occurred. That will cause the program to go back to line 160 in the main program and continue executing statements there. Again at line 190, we see that there is another GOSUB statement which will cause a branch to line 1000 to occur. Again, the subroutine is executed until the RETURN statement at line 1040 is once again executed. The program then goes back to the next line after the GOSUB, which is line 200 this time. From here, the program keeps running until the END statement at line 220 is encountered.

In this particular example, we can see that if the statements that comprise the subroutine are found only once within the program and that if we can access these statements from several places in the main program, then the overall program has been made smaller through the use of the subroutine concept. Without the subroutine ability, we would have had to incorporate the same statements at two places in the main program, once at line 150, and again at line 190. In a sense, the GOSUB statement is like an unconditional branch with a return address telling it where to come back to when the RETURN statement is executed.

ON...GOSUB, Indexed Subroutine Branching

The ON...GOSUB statement works just like the ON...GOTO statement does. That is, there is a variable that is used as the index for the ON...GOSUB statement, and there is a list of possible line numbers to which the subroutine branch is to be made depending upon the value of the index. The common form of the statement looks like this:

```
100 ON K GOSUB 1000,2000,3000,4000,. . . .
```

Again, the value of the index variable at the time that the ON...GOSUB statement is executed must be interpreted as an integer whole number greater than zero, and less than the number of line numbers in the list which follows the GOSUB part of the statement. In the example above, if the value of K is equal to one, the statement will cause a subroutine branch to line 1000. If the value is two, the branch will be made to line 2000, and so on. Again, if the value of K is zero or greater than the number of line numbers in the list, the program will ignore the ON...GOSUB statement completely, and resume execution on the next line in the program. If the value of K is negative, an error will result. After the subroutine has finished execution, the RETURN statement at its end will cause the program to come back to the next statement after the ON...GOSUB which caused the branch in the first place.

Here again, as with the ON...GOTO statement, the major advantage is that several different subroutines may be called depending upon the value of a variable in the program. This variable can be entered via an INPUT statement, or it may be computed in the program itself from some other data. In any case, its value at the time that the ON...GOSUB statement is executed will cause a controlled branch to a subroutine.

In the following pages, we will examine the use of the subroutine concept in two very common instances. The first involves the problem of getting the results of arithmetic operations to come out in whole pennies so that we can print an organized listing. The second example of subroutine usage provides a means of validating input data to make sure that it is within certain limits.

NUMERIC "ROUND-OFF" SUBROUTINE

If we remember the Loan program, there was a small problem encountered whenever we did certain types of arithmetic operations and then tried to print out the results. Because of the way that BASIC controls the PRINT operation, if we try to print a variable which has a current value of 7.50015, *that is exactly what will be printed*, even if we are dealing with dollars and cents and the amount should really be printed as 7.50. One way to overcome this small problem is to "round off" the results of any arithmetic operation which is to be printed. The procedure for doing this is quite simple. It uses one of the simple functions which we learned in the last section. The INT function, which allows us to integerize a variable, is used to get rid of any trailing decimal places. But, since we are using numbers representing dollars and cents, there must be two places to the right of the decimal point in all of the numbers. Also, we want to make sure that the number that is left after the trailing decimal places are removed is rounded to the nearest cent. To accomplish all of these things, we can use the following procedure:

ADD .005 to the number
MULTIPLY the number by 100
INTegerize the new number
DIVIDE the integerized number by 100

Let us go through a few examples of this procedure to see how it works. We will start with the number we used a few lines back.

Original number	7.50015
ADD .005 to the number	7.50515
MULTIPLY by 100	750.515
INTegerize the number	750
DIVIDE by 100	7.5

Well, it still doesn't read 7.50 does it? This is because BASIC doesn't print trailing zeros after numbers. But, let us accept this for now. At least it doesn't print out 7.50015. Let us take another example that will work—a number like 3.466666.

Original number	3.466666
ADD .005 to the number	3.471666
MULTIPLY by 100	347.1666
INTegerize the number	347
DIVIDE by 100	3.47

There, now that is better. Next, let us take a look at how we go about writing a subroutine to do this operation. We could use a LET statement for each of the numeric operations that must be done, where the variable X represents the original number. It would look like this:

```
5000 REM -ROUND OFF SUBROUTINE
5100 LET X = X+.005
5200 LET X = 100*X
5300 LET X = INT(X)
5400 LET X = X/100
5500 RETURN
```

Well, that seems to work just fine. However, the whole thing could be done with just one statement like this:

```
5000 REM -ROUND OFF SUBROUTINE
5100 LET X = (INT(100*(X+.005)))/100
5200 RETURN
```

This approach uses "nested parentheses" to accomplish the same thing as the previous example. Because of the way that the parentheses are arranged, first the addition of .005 will be done then the multiplication by 100, then the INTegerization function and finally, the division by 100 to get back to the

right decimal place, and then the variable X is replaced with the new value.

So then, how would we go about using this subroutine? Let us take a look at some examples. First of all, we would have to set up the variable X to contain whatever number we wanted to round off. This could be done with just a LET statement. Then we would have to GOSUB to the subroutine to perform the rounding operation. When the subroutine is finished, the RETURN statement will bring the program right back to the next line after the GOSUB. This would probably be a PRINT statement since we now have the number rounded off for printing. One ideal place to use this new subroutine would be in the Loan program. Let us see how we might modify the program to incorporate this subroutine. Fig. 2-2 is a reproduction of the entire Loan program with some additional notes.

```
1000 REM -LOAN AMORTIZATION PROGRAM
1010 REM -INPUT SECTION
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1200 REM -CHANGE ANNUAL RATE TO MONTHLY RATE
1210 LET I = R/12
1220 REM -CHANGE YEARS TO MONTHS
1230 LET M = Y*12
1300 REM -COMPUTE THE MONTHLY PAYMENT AND PRINT
1310 LET A = (I*L)/((1-(I+1) ∧.-M))
1320 PRINT "THE MONTHLY PAYMENT IS $"; A
1400 REM -INITIALIZE FOR AMORTIZATION SCHEDULE
1410 LET B = L
1420 LET T = 0
1430 PRINT "NUM", "INTEREST", "PRINCIPAL", "PRIN BAL", "TOT INT"
1500 REM -DO THE LOOP FOR THE NUMBER OF PAYMENTS
1510 FOR J = 1 to M
1520 REM -COMPUTE THE INTEREST
1530 LET I1 = B*I
1540 REM -COMPUTE THE PRINCIPAL
1550 LET P = A-I1
1560 REM -FORCE OUT THE LAST PAYMENT
1570 IF J = M THEN LET P = B
1600 REM -UPDATE THE LOAN BALANCE
1610 LET B = B-P
1620 REM -UPDATE THE TOTAL INTEREST
1630 LET T = T+I1
1700 REM -PRINT THE MONTHLY PAYMENT DETAIL
1710 PRINT J,I1,P,B,T
1720 NEXT J
1800 REM -SEE IF DONE
1810 INPUT "PROCESS ANOTHER LOAN (YES/NO)"; A$
1820 IF A$ = "YES" THEN GOTO 1100
9999 END
```

Fig. 2-2. Listing of Loan program.

Here is one place where the round-off subroutine will come in very useful. At this point, the monthly payment A could be equal to a variety of different decimal places, depending upon the number of decimals in the interest rate R, and what happens to all of these numbers when the arithmetic operations at line 1310 are performed. The monthly payment amount might be as simple as 332.60 or it might come out as weird as 124.12507. So this is the first place in the Loan program where we are about to PRINT out the monthly payment, and we don't want it to come out with five places to the right of the decimal. Let us add a few lines of program right here then, to GOSUB to a subroutine to round off the monthly payment amount. It looks like this:

```
1312 LET X = A
1314 GOSUB 5000
1316 LET A = X
```

Line 5000 is where the round-off subroutine starts.

Notice that we can add lines into the program by using line numbers that are available because we planned ahead when we assigned the original line numbers. Here again, we are trying to leave a little slack in the line numbering scheme in the event that we might need to further modify this section of program in the future. We need to add three lines of program here in order to do the rounding operation. First, since the round-off subroutine always rounds off the variable called X, we must make X equal to the number that we want to be rounded. Therefore, we use the LET statement at line 1312 to get X equal to the monthly payment amount A. Next comes the GOSUB statement that will cause the program to branch to line 5000 where the round-off subroutine is located. After the subroutine has finished, the program will branch back to the next line after the GOSUB statement. That would be line 1316, where we must now replace the value calculated for A with the rounded off version which the subroutine has left stored in the variable X. Now, we can let the program go ahead and PRINT the new rounded off value of A.

Now, you may be wondering just how effective it is to use three lines of additional program just to be able to use the subroutine. Why not just insert the round-off formula as *one* line of program at the same place in the program? In this case, it is probably a trade-off. The amount of memory needed to accommodate the round-off formula is probably about the same as is required to store the three additional statements needed to use the subroutine. But, in other cases, this is not always true. We have chosen a rather simple example to illustrate how subroutines work. The same principles would apply to much more complex subroutines.

```
1320 PRINT "THE MONTHLY PAYMENT IS $"; A
1400 REM -INITIALIZE FOR AMORTIZATION SCHEDULE
```

```
1410 LET B = L
1420 LET T = 0
1430 PRINT "NUM", "INTEREST", "PRINCIPAL", "PRIN BAL", "TOT INT"
1500 REM -DO THE LOOP FOR THE NUMBER OF PAYMENTS
1510 FOR J = 1 TO M
1520 REM -COMPUTE THE INTEREST
1530 LET I1 = B*I
```

Recall here we use a FOR . . . NEXT statement (FOR J = 1 TO M) to make a loop for each month of the schedule. The index J will vary from 1 to M (M = the term of the loan in months). The NEXT J statement for this loop is coming up later, so don't worry. Now let us continue with the loan program.

This is also another place in the Loan program where the result of the arithmetic operation at line 1530 is undependable. The amount of interest for the particular monthly payment may come out to be some funny looking number with lots of digits to the right of the decimal point. So, to eliminate this problem, we will insert three more lines of program here to allow the round off subroutine to take care of the extra digits.

```
1532 LET X = I1
1534 GOSUB 5000
1536 LET I1 = X
```

Line 5000 is the round-off subroutine. I1 is the interest for a particular month J.

Here again, we follow the same procedure. First we set the variable X equal to the value of I1 which was just computed. Then, using the GOSUB statement we branch to the subroutine. Upon return from the subroutine, we must replace I1 with the new "rounded off" version.

From here on, there is no further need for rounding things off. The results of addition or subtraction operations will always yield a number with the same decimal places as the number with the most decimals which was involved in the arithmetic. For example, if we subtract

$$\begin{array}{r} 1{,}500.00 \\ -125.35 \\ \hline 1{,}474.65 \end{array}$$

the answer has two digits to the right of the decimal, just like both of the numbers involved in the subtraction. Likewise,

$$\begin{array}{r} 800.0 \\ -57.25 \\ \hline 742.75 \end{array}$$

Again, the answer has the same number of digits to the right of the decimal point as that of the number with the largest that was involved in the arithmetic. So, since all the rest of the arithmetic operations in the Loan program are either addition or subtraction, there will be no further need for the round-off subroutine.

Here is the rest of the program. Recall line 1550 calculates the principal part of the monthly payment (P=A−I1), line 1570 forces the last payment (when J=M) to be equal to the actual remaining amount on the loan (P=B), line 1610 calculates the new principal balance B, line 1630 accumulates the total interest paid, and line 1710 prints out the five items in the loan schedule (NUM, INTEREST, PRINCIPAL, PRIN BAL, TOT INT) in neat columns across the page. The NEXT J statement at line 1720 causes the process to repeat for the next month J. The statements 1800-1820 check if you wish to process another loan.

```
1540 REM -COMPUTE THE PRINCIPAL
1550 LET P = A-I1
1560 REM -FORCE OUT THE LAST PAYMENT
1570 IF J = M THEN LET P = B
1600 REM -UPDATE THE LOAN BALANCE
1610 LET B = B-P
1620 REM -UPDATE THE TOTAL INTEREST
1630 LET T = T+I1
1700 REM -PRINT THE MONTHLY PAYMENT DETAIL
1710 PRINT J,I1,P,B,T
1720 NEXT J
1800 REM -SEE IF DONE
1810 INPUT "PROCESS ANOTHER LOAN (YES/NO)"; A$
1820 IF A$ = "YES"THEN GOTO 1100
```

The last thing we do is insert the round-off subroutine. There is one other thing that we need to insert just before the subroutine. At line 1820, if the result of the IF statement is false, the branch to line 1100 will not be taken, and the *next line* will be executed. Since we don't want to execute the subroutine when we get here, the first thing that we must add is an unconditional branch to the END statement at line 9999. Here is the rest of the program.

```
1830 GOTO 9999
5000 REM -ROUND OFF SUBROUTINE
5010 LET X = (INT(100*(X+.005)))/100
5020 RETURN
9999 END
```

Fig. 2-3 shows the complete listing of the enhanced loan program with the numeric round-off subroutine.

```
1000 REM -LOAN AMORTIZATION PROGRAM
1010 REM -INPUT SECTION
1100 INPUT "ENTER THE AMOUNT OF THE LOAN"; L
1110 INPUT "ENTER THE ANNUAL INTEREST RATE"; R
1120 INPUT "ENTER THE TERM IN YEARS"; Y
1200 REM -CHANGE ANNUAL RATE TO MONTHLY RATE
1210 LET I = R/12
1220 REM -CHANGE YEARS TO MONTHS
1230 LET M = Y*12
1300 REM -COMPUTE THE MONTHLY PAYMENT AND PRINT
1310 LET A = (I*L)/((1-(I+1) ^ -M))
1312 LET X = A
1314 GOSUB 5000
1316 LET A = X
1320 PRINT "THE MONTHLY PAYMENT IS $"; A
1400 REM -INITIALIZE FOR AMORTIZATION SCHEDULE
1410 LET B = L
1420 LET T = 0
1430 PRINT "NUM", "INTEREST", "PRINCIPAL", "PRIN BAL", "TOT INT"
1500 REM -DO THE LOOP FOR THE NUMBER OF PAYMENTS
1510 FOR J = 1 to M
1520 REM -COMPUTE THE INTEREST
1530 LET I1 = B*I
1532 LET X = I1
1534 GOSUB 5000
1536 LET I1 = X
1540 REM -COMPUTE THE PRINCIPAL
1550 LET P = A-I1
1560 REM -FORCE OUT THE LAST PAYMENT
1570 IF J = M THEN LET P = B
1600 REM -UPDATE THE LOAN BALANCE
1610 LET B = B-P
1620 REM -UPDATE THE TOTAL INTEREST
1630 LET T = T+I1
1700 REM -PRINT THE MONTHLY PAYMENT DETAIL
1710 PRINT, J,I1,P,B,T
1720 NEXT J
1800 REM -SEE IF DONE
1810 INPUT "PROCESS ANOTHER LOAN (YES/NO)/"; A$
1820 IF A$ = "YES" THEN GOTO 1100
1830 GOTO 9999
5000 REM -ROUND OFF SUBROUTINE
5010 LET X = (INT(100*(X+.005)))/100
5020 RETURN
9999 END
```

Fig. 2-3. Listing of the enhanced Loan program with numeric round-off subroutine included. Old parts of the program are shaded with color.

ERROR-CHECKING SUBROUTINE

Another simple use of the subroutine concept can be illustrated by the example of checking for errors in INPUT data. There is an old programmer's proverb that says, "Garbage in, garbage out." This simply means that if you enter invalid information into a computer program, you cannot depend on the results. In some situations, the program might be "faked out" by the invalid information, and go right ahead and process it anyway. This usually yields an invalid answer, or output. In other cases, the INPUTing of invalid information may cause a program error to occur which might result in the total malfunction of the program. This is especially true if the INPUT information is to be used in a program-control statement such as a FOR . . . NEXT loop where the use of an invalid loop parameter could cause the loop to go on forever.

This subroutine provides the most basic error-checking capability and can be used in both the Loan program and the Metric Conversion program. What it will do is:

1. Check for a positive, nonzero number.
2. Check that the number does not exceed some predetermined maximum value.
3. Display an error message if either of these conditions is not correct and re-INPUT the data.

The subroutine looks like this:

```
6000 REM -NUMERIC ERROR CHECKING SUBROUTINE
6100 IF X > 0 AND X < =Z THEN RETURN
6200 INPUT "INVALID ENTRY, TRY AGAIN"; X
6300 GOTO 6100
```

The real work here is done by the IF . . . THEN statement on line 6100. The first thing that happens here is that it checks to see that the value of the variable X is greater than zero. It also checks to see that X is less than or equal to another variable called Z. Remember how the IF . . . THEN statement works? If *both* of the premises are true, then the result is also true; this means that the RETURN statement is executed, taking the program back to where the subroutine branch was made because X passed the test. Here again, we are using a variable called X which must be set up in the main program to be equal to the current value of the INPUT data to be checked. Also, the variable Z must be set up by the main program prior to the branch to the subroutine to be equal to the maximum value that the INPUT data can have. If either of the two premises in the IF . . . THEN statement is *not* true, the next line following the IF . . . THEN statement is executed. This is the INPUT statement which

causes an error message to be output and then the data to be INPUT again. The GOTO on line 6300 takes the subroutine back to check the new value.

We will use this subroutine to modify the Metric Conversion program so that we can check the INPUT data as desired. Here is the program listing up to line 2120.

```
2000 REM -METRIC CONVERSION PROGRAM
2010 REM -DISPLAY THE MENU
2020 PRINT
2030 PRINT "METRIC CONVERSION PROGRAM"
2040 PRINT
2041 PRINT "1 - INCHES TO CENTIMETERS"
2042 PRINT "2 - POUNDS TO KILOGRAMS"
2043 PRINT "3 - QUARTS TO LITERS"
2049 PRINT "9 - TO END THE PROGRAM"
2050 PRINT
2060 REM -GET THE SELECTION
2070 INPUT "ENTER YOUR SELECTION"; S
2080 REM -DECIDE WHERE TO BRANCH
2081 IF S = 1 THEN GOTO 2100
2082 IF S = 2 THEN GOTO 2200
2083 IF S = 3 THEN GOTO 2300
2089 IF S > 3 OR S <1 THEN GOTO 9999
2100 REM -CONVERT INCHES TO CENTIMETERS
2110 PRINT
2120 INPUT "ENTER LENGTH IN INCHES "; I
```

Here is a likely place to use the error-check subroutine. At line 2120 the program has just INPUT a value for the variable I which represents the number of inches that is to be converted to centimeters. If we are going to use the error-check subroutine to test this value, we must first establish the allowable maximum value for I. For the purpose of demonstration, let us suppose that we don't want to convert anything greater than 36 inches. Since we will have to get the variable X equal to the value that was INPUT for I, why not INPUT X directly by changing line 2120 also? The new statements look like this:

```
2120 INPUT "ENTER LENGTH IN INCHES "; X
2122 LET Z = 36
2124 GOSUB 6000
2126 LET I = X
```

Notice that when the subroutine returns to the main program we must finally set I equal to the INPUT value X (line 2126). In a similar manner, we can insert statements in the other conversion sections of the program in order to check the INPUT data and choose other values for the allowable maximum. Here is how the program continues to line 2220.

```
2130 LET C = 2.54*I
2140 PRINT I;" INCHES IS EQUAL TO ";C" CENTIMETERS"
2150 GOTO 2000
2200 REM -CONVERT POUNDS TO KILOGRAMS
2210 PRINT
2220 INPUT "ENTER WEIGHT IN POUNDS "; P
```

We will set a maximum weight of 2000 pounds to convert to kilograms and add a GOSUB to our error-checking subroutine.

```
2220 INPUT "ENTER WEIGHT IN POUNDS "; X
2222 LET Z = 2000
2224 GOSUB 6000
2226 LET P = X
```

Notice again, that we have replaced the INPUT statement with one which INPUTs the value of X directly. Then, once the value is determined to be correct, we set P equal to X. Continuing with the program we have:

```
2230 LET K = .4536*P
2240 PRINT P;" POUNDS IS EQUAL TO ";K;" KILOGRAMS"
2250 GOTO 2000
2300 REM -CONVERT QUARTS TO LITERS
2310 PRINT
2320 INPUT "ENTER VOLUME IN QUARTS "; Q
```

Here is the last part to be inserted into the program. The allowable maximum liquid volume to be entered will be 100 quarts.

```
2320 INPUT "ENTER VOLUME IN QUARTS "; X
2322 LET Z = 100
2324 GOSUB 6000
2326 LET Q = X
2330 LET L = .9463*Q
2340 PRINT Q;" QUARTS IS EQUAL TO ";L;" LITERS"
2350 GOTO 2000
```

Here is where we will insert the subroutine itself.

```
6000 REM -ERROR CHECKING SUBROUTINE
6100 IF X > 0 AND X < =Z THEN RETURN
6200 INPUT "INVALID ENTRY, TRY AGAIN "; X
6300 GOTO 6100
```

```
2000 REM -METRIC CONVERSION PROGRAM
2010 REM -DISPLAY THE MENU
2020 PRINT
2030 PRINT "METRIC CONVERSION PROGRAM"
2040 PRINT
2041 PRINT "1 - INCHES TO CENTIMETERS"
2042 PRINT "2 - POUNDS TO KILOGRAMS"
2043 PRINT "3 - QUARTS TO LITERS"
2049 PRINT "9 - TO END THE PROGRAM"
2050 PRINT
2060 REM -GET THE SELECTION
2070 INPUT "ENTER YOUR SELECTION"; S
2080 REM -DECIDE WHERE TO BRANCH
2081 IF S = 1 THEN GOTO 2100
2082 IF S = 2 THEN GOTO 2200
2083 IF S = 3 THEN GOTO 2300
2089 IF S > 3 OR S < 1 THEN GOTO 9999
2100 REM -CONVERT INCHES TO CENTIMETERS
2110 PRINT
2120 INPUT "ENTER LENGTH IN INCHES "; X
2122 LET Z = 36
2124 GOSUB 6000
2126 LET I = X
2130 LET C = 2.54*I
2140 PRINT I;" INCHES IS EQUAL TO ";C;" CENTIMETERS"
2150 GOTO 2000
2200 REM -CONVERT POUNDS TO KILOGRAMS
2210 PRINT
2220 INPUT "ENTER WEIGHT IN POUNDS "; X
2222 LET Z = 2000
2224 GOSUB 6000
2226 LET P = X
2230 LET K = .4536*P
2240 PRINT P;" POUNDS IS EQUAL TO ";K;" KILOGRAMS"
2250 GOTO 2000
2300 REM -CONVERT QUARTS TO LITERS
2310 PRINT
2320 INPUT "ENTER VOLUME IN QUARTS "; X
2322 LET Z = 100
2324 GOSUB 6000
2326 LET Q = X
2330 LET L = .9463*Q
2340 PRINT Q;" QUARTS IS EQUAL TO ";L;" LITERS"
2350 GOTO 2000
6000 REM -ERROR CHECKING SUBROUTINE
6100 IF X > 0 and X <=Z THEN RETURN
6200 INPUT "INVALID ENTRY, TRY AGAIN "; X
6300 GOTO 6100
9999 END
```

Fig. 2-4. Metric Conversion program with error-checking subroutine and code added. Old parts of program are shaded with color.

Fig. 2-4 shows the complete listing of the Metric Conversion program with the error-checking subroutine and code added.

We've learned a large number of powerful new BASIC statements and a couple of useful program methods. The next thing we will learn about will allow your programming to jump to a higher level of sophistication through the use of the "array" concept—so let's get moving.

REVIEW QUESTIONS

1. What will the following program do?

```
10 FOR J=100 TO 50
20 PRINT J
30 NEXT J
```

2. What will this program do?

```
10 FOR J=0 TO 10 STEP 0.5
20 PRINT J
30 NEXT J
```

3. What will this program do, when you run it?

   (a) if you type in 0?
   (b) if you type in 1?
   (c) if you type in 2?

```
10 INPUT J
20 ON J GOTO 20,40,50
30 PRINT"NEVER";
40 PRINT"SAY";
50 PRINT"DIE.";
60 STOP
```

4. Is this a valid expression?

```
10 IF 2=XX THEN PRINT"TRUE"
```

5. What will the following program do?

```
100 GOSUB 200
110 PRINT"T";
120 STOP
200 GOSUB 300
210 PRINT"A";
220 RETURN
300 PRINT"C";
310 RETURN
```

6. Is ABS(INT (−3/2)) the same as INT(ABS(−3/2))?

7. What does this program do?

```
10 FOR K=1 TO 15
20 FOR J=1 TO 30*RND(1)
30 PRINT"*";
40 NEXT J
50 PRINT
60 NEXT K
```

8. What will this program do?

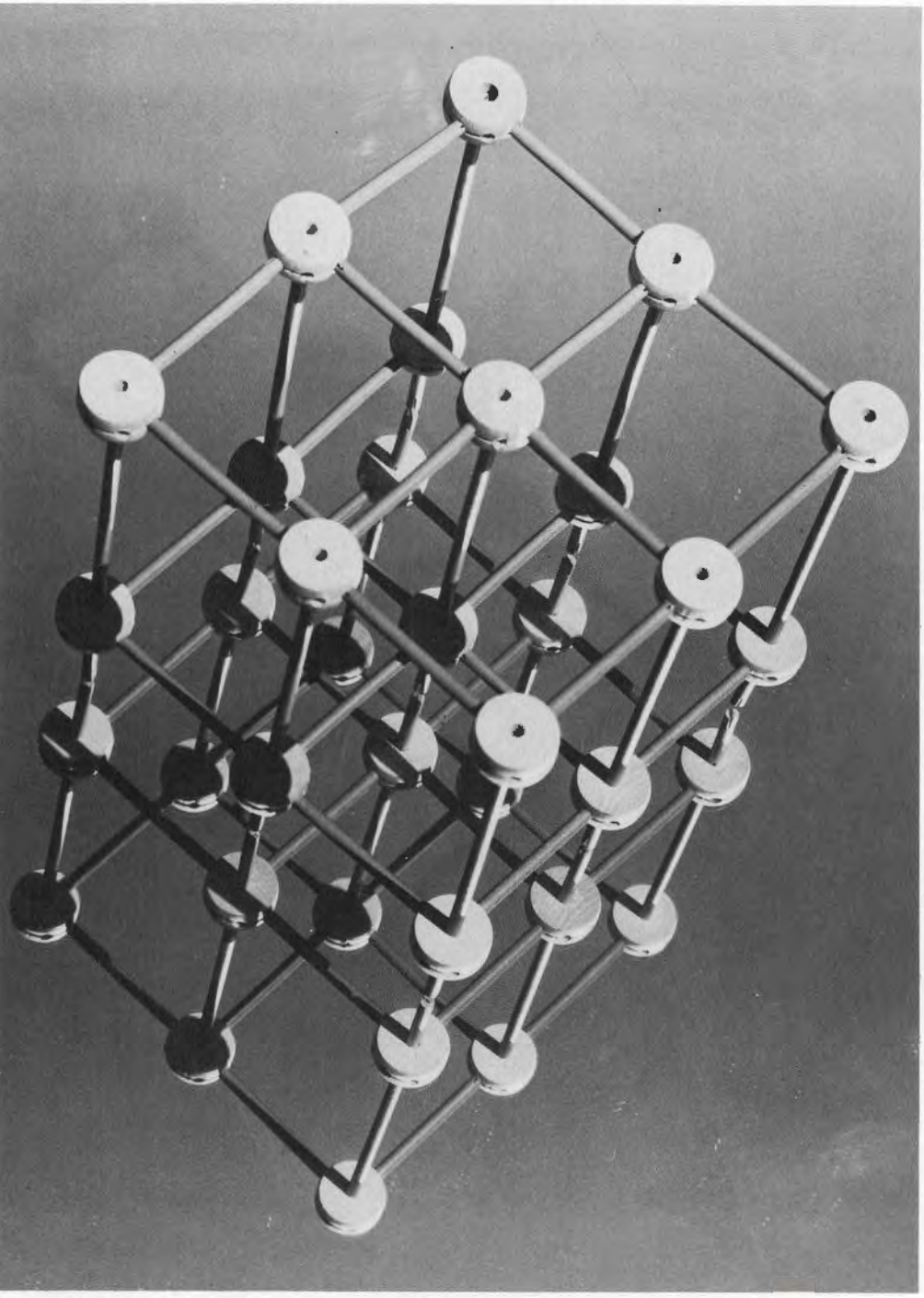
```
10 IF RND (1) <> 0.5 THEN GOTO 10
20 PRINT"DONE" : END
```

9. What will happen if you run this program?

```
10 GOSUB 100
20 STOP
100 I=I+1
110 PRINT I
120 GOTO 10
130 RETURN
```

10. How many times will this program execute lines 30 and 40?

```
10 W$="0"
20 S$="B"
30 S$=S$+W$
40 GOTO 30
```



CHAPTER

3

Getting Organized

So far in our study of BASIC we have learned how to deal with information which is made up of a single part. We can manipulate numbers by giving each a variable name, or groups of letters by giving each string a string-variable name. In order to make BASIC programming more powerful, the originators decided to allow you to take a single variable name and use it to represent information that is made up of several parts. In this section we will explore the concept of "arrays and subscripted variables," a method whereby information may be divided into several sub-elements or parts, and each part can be organized and referenced by your program with ease. Such a method means we can represent information inside the computer in a more organized manner than is possible with single-value variables. As a way of illustrating the array idea we will show how to make a program that plays TIC-TAC-TOE against you by using the array concept throughout its operation. As a by-product of the program we will see how a computer game is made, the steps involved in converting a popular game into a series of program steps called "flowcharting," and how a game "algorithm" is formed which gives it the machine intelligence to figure out how to win.

ARRAYS AND SUBSCRIBED VARIABLES

Arrays are items of data arranged and stored under a single variable name. An array has a simple name like regular variables but instead of that variable only representing a single value, an array represents several values. The parts that make up an array are called its "elements." Each element can be a number or a string. An array element is identified by an array name followed by an integer number which can be as low as 0 or as large as 255.* The number is enclosed in parentheses and follows the array name. The number

*For versions of Microsoft BASIC you can have up to 10 elements in an array before you must use the DIM statement. We will describe this shortly.

serves to identify a unique element in the array. We call a variable with a referencing number following it a "subscripted variable." The general form of a subscripted variable looks like this:

```
ARRAYNAME (integer constant)
```

The ARRAYNAME part of a subscripted variable is any legal BASIC variable name as described in Chapter 1. The "subscript" is the integer constant following the ARRAYNAME and surrounded by parentheses. The subscript tells us which item of the variable we are referencing.

In order to make this concept of arrays more lucid consider this "loaf of bread" analogy. Imagine that a loaf of bread is a variable in a program. The loaf has a name, say L, and we can talk about the variable loaf just like it is a BASIC variable. Now without arrays we can only talk about a single loaf of bread when we use the letter L. We cannot indicate a particular piece of the bread, like the center of the loaf, or the heels, or any other part of the loaf. Further, we cannot tell if the bread is wheat, white or rye unless we make up some new variable to stand for these different items of information. To talk about a loaf of wheat bread we must create a new variable, say L2, or W for Wheat. If the bread was sliced and sold in a varying number of slices in a loaf, it would be complicated to represent the different slices of different breads. (What slice is LL or B4?)

But, if we represent the loaf as an array, things become much more efficient and organized. What we can do is divide the loaf into several slices and use the array element to reference a certain slice in the loaf. The first slice would be L(1), the second L(2), the third L(3), and so on. The last slice would be determined by the largest numbered element which for now we will say is L(10). With subscripted variables unique for each slice, we can add more information about the bread. For example, if the loaf was raisin bread, we could specify how many raisins were in each slice, rather than just the total number of raisins in the loaf. If slice number one has 7 raisins in it we could represent this in BASIC as:

```
L(1) = 7
```

This is read as "L sub 1 equals 7" and means the value of element 1 of the array L is 7.

Suppose we wanted to set up the bread array (L) so that each slice contained exactly five raisins. We could do this two ways: (1) with explicit LET statements for each element, or (2) with a FOR . . . NEXT loop. The LET approach is rather straightforward, but requires quite a bit of code. Assuming we have five slices in the loaf, the LET approach looks like this:

```
100 LET L(1)=5
110 LET L(2)=5
120 LET L(3)=5
130 LET L(4)=5
140 LET L(5)=5
```

Now each element of the array L contains the constant 5.

The other approach is to use a FOR . . . NEXT loop and is possible because we can make the subscript (the value between parentheses) an integer variable. The integer variable subscript can be the index of the loop and each time it is incremented we are locating or referencing the specified element, like this:

```
100 FOR I=1 TO 5
110 LET L(I)=5
120 NEXT I
```

In this example, the array subscript is a variable called I. As I is incremented by the FOR . . . NEXT statement, the array changes to the next element, which is then set to the constant 5 by statement 110. As you can see, this approach requires fewer statements to accomplish the same thing as did the five LET statements.

In a similar manner we can make the elements of the array be displayed on the screen or terminal device by using another FOR . . . NEXT loop with an imbedded PRINT statement in it like this:

```
200 FOR I=1 TO 5
210 PRINT L(I),
220 NEXT I
```

These statements make the elements of the array L appear on the screen in columns because of the comma following the PRINT statement on line 210.

We can also make the subscript of the array an expression, as long as the results of the expression evaluate to an integer. (An array subscript must be an integer because there is no such thing as a fractional element—L(1.5) just doesn't exist.) An example of an expression for a subscript would be when we want to reference the elements in some special order, such as just the last 5 elements of a 10 element array, like this

```
300 FOR I=1 TO 5
310 LET L(I+5)=5
320 NEXT I
```

Obviously array and subscripted variables make our programming job much easier and more flexible. But arrays are not limited to just a series of linear parts. We can further divide an array with the aid of a concept called "dimension."

DIMENSION WITH DIM

Arrays and subscripts are not limited to a single element. We can, for example, consider organizing our bread analogy into even more detail. Suppose you wanted to not just specify the particular slice in the loaf, but wanted to organize the loaf into sections, say a front, a middle and an end. Now we have to specify two items: section and slice. How is this done? The answer is to consider that the loaf has two "dimensions." The first dimension is the slice. The second dimension is the section. BASIC accomplishes this concept



DIMension

by allowing you to specify a second independent subscript between the parentheses. These two subscripts must be separated by a comma, like this:

```
L(I,J)
```

Here the subscript I indicates the slice in the loaf L, and the subscript J indicates the section in the loaf L. For each section J there is a number of slices I which depends on the size of the array. We can call a dual sub-

scripted array, a *two dimensional* array, because it has the equivalent of dimensions, just like height and width describe a room.

Now when we specify the loaf we talk about a particular section and a particular slice in that section. For example, to specify that we set the middle section (section 2) and the third slice in that section we would write:

```
LET L(3,2)=5
```

To help visualize the array we can imagine that the elements are arranged in the computer like this (assuming there are three sections and five slices per section):

```
L(1,2) L(2,1) L(3,1) L(4,1) L(5,1)
L(1,2) L(2,2) L(3,2) L(4,2) L(5,2)
L(1,3) L(2,3) L(3,3) L(4,3) L(5,3)
```

For each of the five slices there is one particular section it is associated with, (1-3). Thus we now have a total of 5 × 3 or 15 slices in the loaf. We say that array L is an I by J array in BASIC.

What is nice about BASIC arrays is that we can keep adding information, or dimensions, to the array by increasing the number of subscripts. For example, if we wanted to specify a particular section, slice, and *bakery* that the bread is found in we could add a third subscript, like this:

```
L(I,J,K)
```

Here the subscript K stands for the bakery that the bread is found in and the subscripts I and J are as previously described. We now have a three dimensional array, or an I by J by K array.

Visualizing three dimensions is certainly not as easy as two, but for convenience you can imagine that the third dimension is like the length of a space, and the other two stand for height and width.

In order for BASIC to set aside the proper amount of memory space for holding the element values of the array we need to specify this before the array is used. BASIC provides a special keyword called DIM for this purpose. DIM appears at the beginning of a program that uses an array, and describes to the computer how many dimensions are in the array and how many elements are in each dimension. The general form of DIM is:

```
DIM arrayname(I,J,K,. . . ), arrayname(I,J,K,. . . ),. . .
```

where arrayname is the symbolic name of the array which follows the convention for variable names and I, J, and K stand for the *maximum elements* of the array dimensions. For example, the two-dimensional array in the loaf analogy with three sections and five slices per section would be dimensioned as:

```
DIM L(5,3)
```

The 5 indicates that the first dimension has a maximum of five elements in it and the second dimension has a maximum of three elements. Thus the computer sets aside 15 memory locations (or more if the values take up more than one byte each) for the array L.

For a three-dimensional array we would write:

```
DIM L(5,3,10)
```

where the 10 indicates that there are 10 bakeries each of which contains a loaf with 3 sections and 5 slices per section. Obviously the array needs 3 × 5 × 10 elements or 150 total elements to fully describe all combinations.

As the general form of the DIM statement shows, we can have several arrays dimensioned on the same DIM statement by simply separating each with a comma. This saves us having to write separate DIM statements for each array we wish to dimension.

Is there any limit to the number of dimensions an array can have? The answer is yes, but the actual maximum number of dimensions is a function of which BASIC you are using. The simple BASICs such as the 4K and integer BASICs, only allow a single dimensional array. The 8K and extended BASICs allow a maximum number of dimensions usually determined by the longest line you can use. Thus the maximum dimensions may range up to 127, depending on how many elements are specified for each dimension. The maximum number of elements for a particular dimension is usually limited to about 255 in most BASICs.

Is the element 0 a legal element? In most cases the zero element means the same thing as the simple variable without dimension, i.e., L(0) is the same as L. In general you should consult your BASIC manual to make sure this is the way your version actually works.

GAME BASIC: TIC-TAC-TOE

Arrays, as we have learned, are most useful for accessing information that is ordered in a special way. Games, particularly those employing a board and men or marks that move on the board, lend themselves well to simulation in



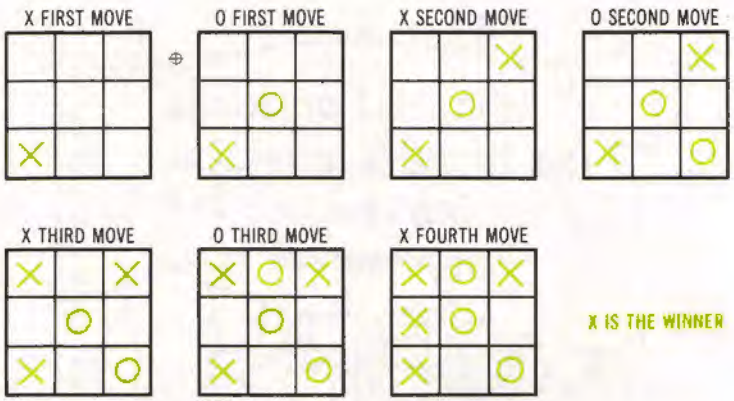
BASIC using an array. In fact, games in BASIC have quickly become most popular for learning how to program a computer, offering the proverbial battleground between man and machine.

In this section we will learn how the game of TIC-TAC-TOE can be represented in BASIC. The program will enable you to play against the computer. The computer will take care of everything—displaying the game board, moving the men, checking for ties or a winner, and allowing you to input your move. A special "move algorithm" which allows the computer to decide what move to make will be developed. But before we get into the program, let us review the rules of the game. Then, we will see how to translate these rules into program steps and how to expand them for an improved game.

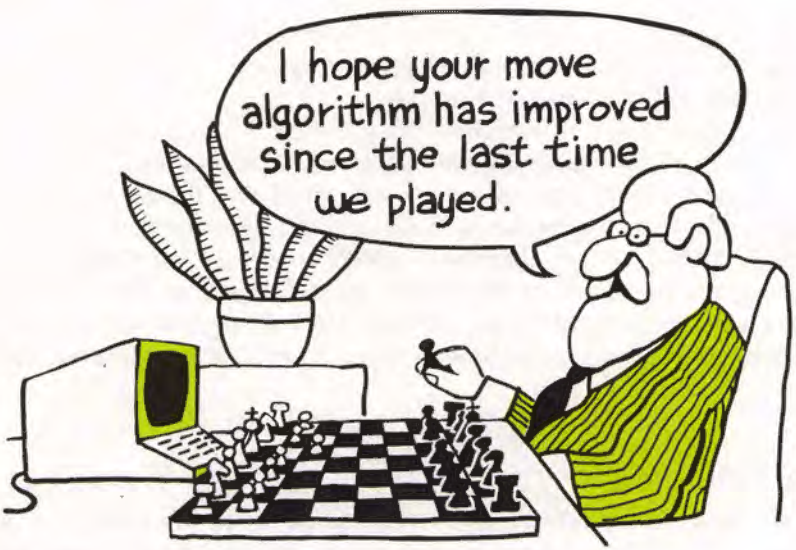
Remember The Rules?

For review, there are two players in the game of TIC-TAC-TOE. One player uses an "X" marker and the second player (in this case the computer) uses an "O" marker. The playing board consists of a square divided into nine smaller squares, and the object of the player is to get three of his own markers in a straight line by filling a row, line, or diagonal. Plays are made on the board in alternate moves. If no player gets a line and all squares are filled the game is a tie, or "cat."

The moves of a typical game may go like this:



There are a few simple strategies you can employ to make your chances of winning more likely. If you move first, take the center square. If you move second take the center square if it is unoccupied. If the center is already occupied, take a corner square. Then play to either block your opponent, or to win a line.



Representing the Board

Our first problem is to represent the board. A simple way to do this is to use a 9-element two-dimensional array called A(3,3). Since there are nine elements in the array, and there are three columns and three rows, the array will be dimensioned as:

```
DIM A(3,3)
```

The squares of the game board are referenced in the array like this:

A(1, 1)	A(1, 2)	A(1, 3)
A(2, 1)	A(2, 2)	A(2, 3)
A(3, 1)	A(3, 2)	A(3, 3)

Representing the Markers

We also need some way to represent the two different players' marks, i.e., the X and the O. One way to do this allows the program to make decisions rapidly. We will use the number −1 to stand for the computer's marker, and the number +1 (or just 1) to stand for the player's (opponent against the computer) marker. The value of any element thus tells us who is in that particular square. An empty square will be represented by a 0 (zero) value. Thus, if A(2,2) = −1 then the computer has a marker on the center square. If A(3,3)=1 then the player has a marker on the lower right corner of the board.

Overall Flow of the Game

In order to make the methods of converting a game like TIC-TAC-TOE to play on the computer we need to accomplish several steps. One way to visualize these steps is to use what is called a "flowchart." A flowchart is simply a diagram of the logic of the game, showing us what happens as the game proceeds and what decisions must be made. There are several special flowchart symbols for doing this and we will learn these as we develop the flowchart. A flowchart can have several levels of meaning, each of which gives more information about the action of the game itself. The most general flowchart explains the overall flow of the program and is shown in Fig. 3-1.

In this flowchart the boxes indicate the processes the game follows. The triangle symbol indicates a decision-making step. The flowchart shows that the game starts by displaying the rules of the game to the user, and then requests that the user select who goes first. After this the computer makes a move (or the player if he goes first) and then the board is displayed, the game checked for a tie or a winner, and the next player proceeds with a move. This generalized flowchart helps us visualize the overall logic of the game, but to use the flowchart we must expand it.

Fig. 3-2 shows the flowchart for the game in more detail. Processes that involve input and output are illustrated with a parallelogram symbol. Deci-

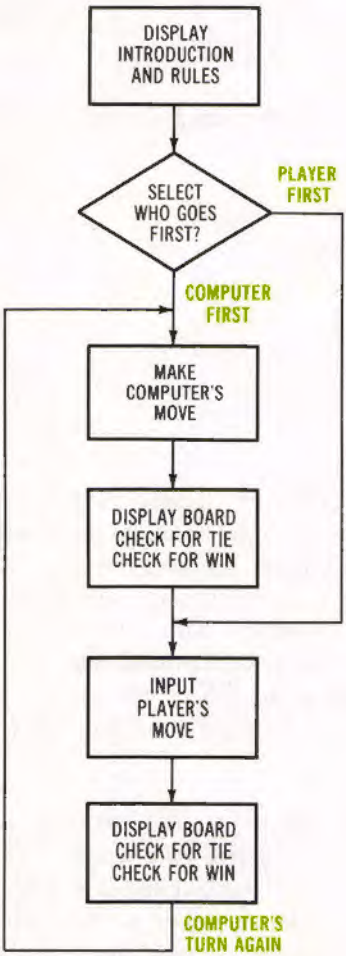


Fig. 3-1. Overall flowchart for TIC-TAC-TOE.

sions are made in triangle symbols and subroutines are set up as rectangles with pointed sides.

As you can see in the flowchart we have represented three important functions—display the board, check for a tie, and check for a winner as subroutines. Let us take a look at how they work.

Displaying the Board

One of the most important things the computer program must do is display the board after each player has made a move. This is accomplished in a simple subroutine that simply increments through each array element and prints it out on the screen.

To increment through the array we use the subscript I to stand for the row element and the subscript J to stand for a column element. Then I and J are used as the indexes of two FOR . . . NEXT loops. I is incremented after J has

been incremented three times, and a PRINT statement makes the array elements appear on the screen just like the real board would. The codes for the subroutine for displaying the game board looks like this:

```
6997 REM
6998 REM        DISPLAY THE BOARD
6999 REM
7000 PRINT: PRINT
7005 FOR I=1 TO 3: PRINT TAB(20): FOR J=1 TO 3: PRINT A(I,J);"    ";
7010 NEXT J: PRINT: NEXT I: PRINT:PRINT: RETURN
```

The first thing to notice about this program is that we have more than one statement on some lines. Most BASICS allow multiple statement lines by using a "continuation" or "separator" character between the statements. In our BASIC we use the colon symbol (:) to separate statements with the same line number. For example, line 7000 (PRINT:PRINT) simply means two PRINT statements are executed before line 7005 gets executed. Line 7005 starts a FOR . . . NEXT loop, does a PRINT and a TAB, starts a second loop, then PRINTs the referenced element of array A, and finally executes line 7010. Some BASICs use a reverse backslash symbol (/) to indicate continuation of a line.

Notice the way the PRINT statement is set up. The quote marks contain three blank spaces which cause the elements of the board to be separated so you can easily see what values they are. The TAB statement spaces the board over 20 spaces on the screen so it is in the approximate middle of the display area. The flowchart for the display board subroutine looks like Fig. 3-3.

Notice also that we are placing the REM statements before the actual subroutine with line numbers that are less than the line numbers of the subroutine itself. By keeping the line numbers for the REM statements out of the main body of the subroutine we can easily remove them later to save precious memory space.

Assuming that the elements of the board array are all zero when we start the game (you may have to zero them yourself if your BASIC does not guarantee this), we can display the board by running the subroutine:

```
RUN 7000

                    0   0   0
                    0   0   0
                    0   0   0
```

Notice how the display moved over 20 spaces before printing out. Later we will see how to add lines so the squares are easier to see, but now you can just imagine they exist.

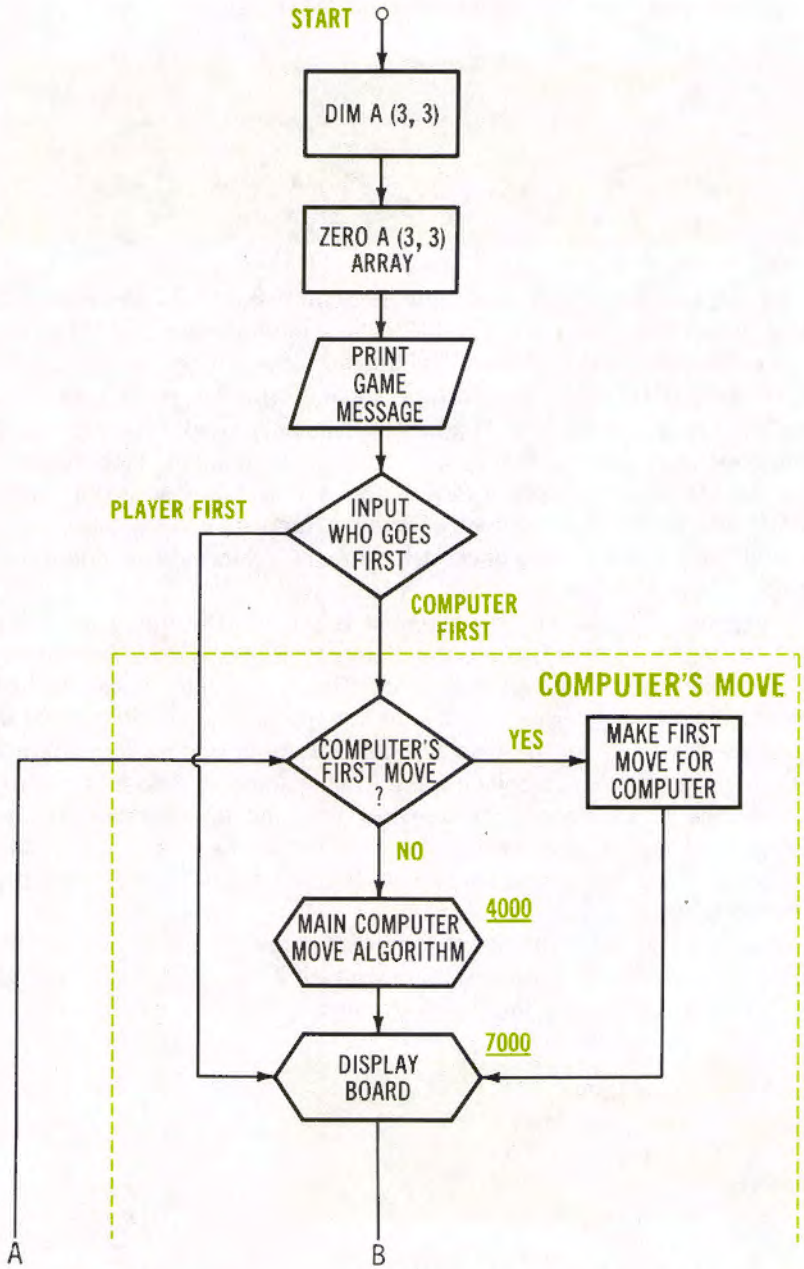
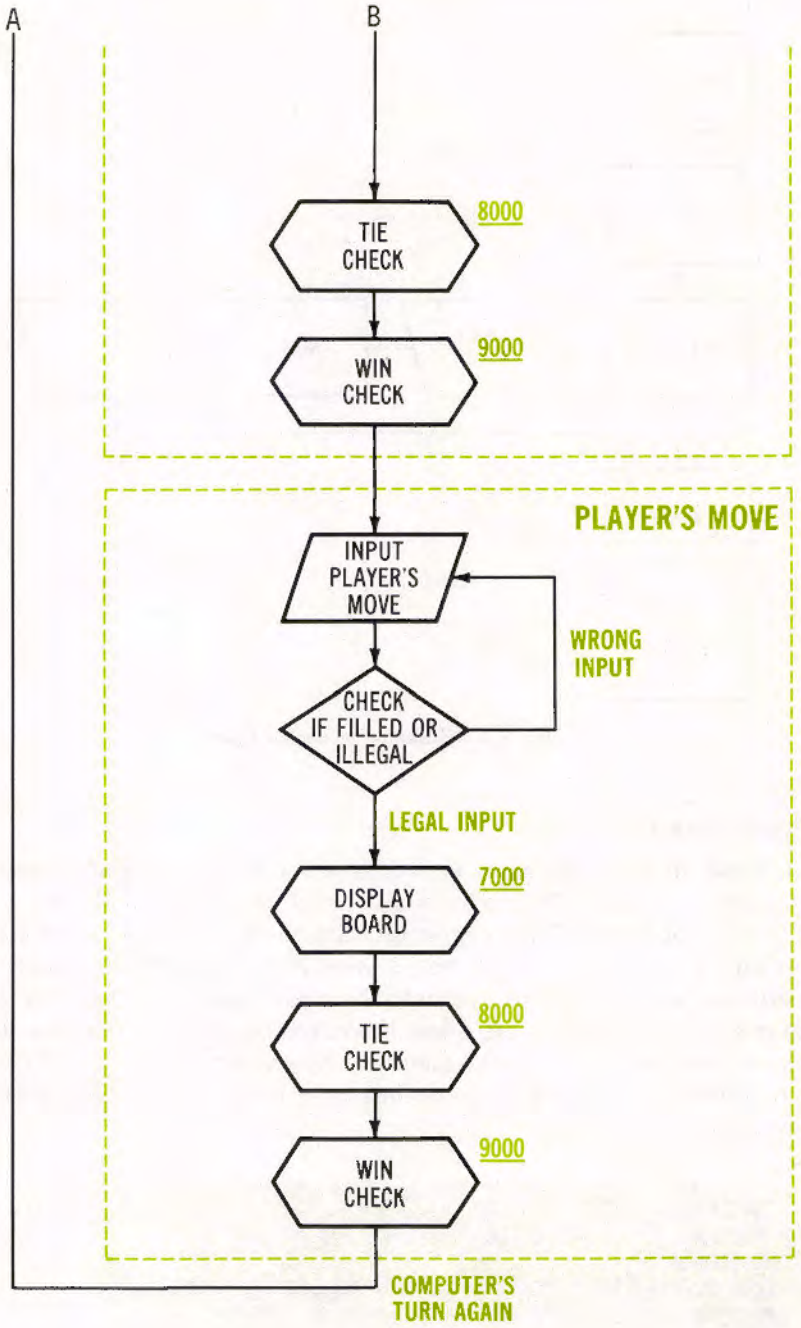


Fig. 3-2. Detailed flowchart



for TIC-TAC-TOE.

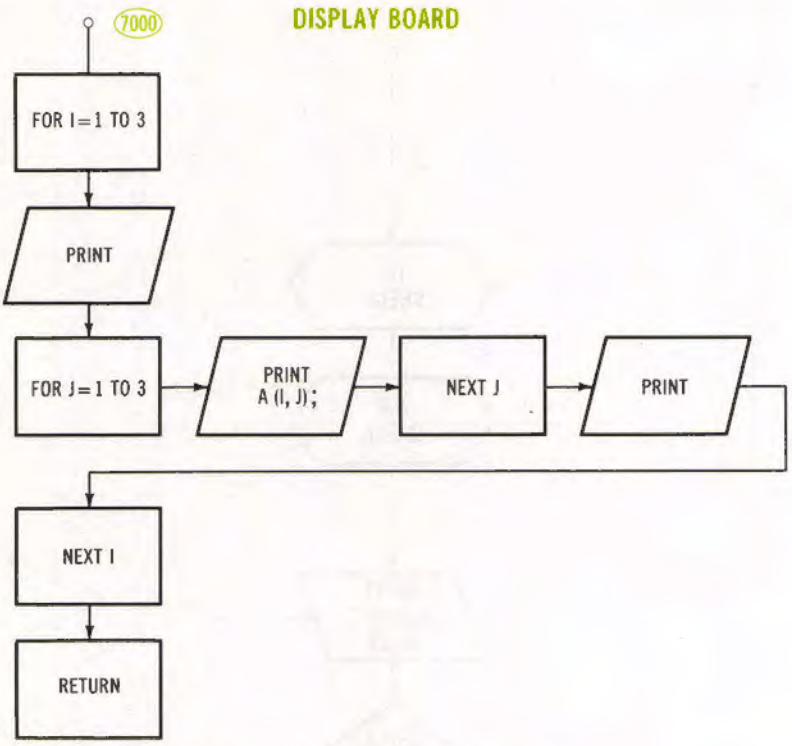


Fig. 3.3. Subroutine to display board.

Check for a Tie

In order to tell if the game is over or not we use a second subroutine to check for a tie game. The code for this function is quite simple. All we need to do is to loop through the elements of the board array looking for a zero (0) element. If we find a zero it means there is still an unfilled square on the board. In this case we must return to the main program because the game is not over until all squares are filled. If *no* zero squares are found by the subroutine then the board must be filled and we cause the message "TIE GAME" to be printed, and a new game begins. Here is the code for the tie-checking subroutine:

```
7997 REM
7998 REM          TIE CHECK
7999 REM
8000 FOR I=1 TO 3: FOR J=1 TO 3: IF A(I,J)=0 THEN 8020
8010 NEXT J: NEXT I: PRINT TAB(20) "TIE GAME": GOTO 20
8020 RETURN
```

The flowchart for the tie-checking subroutine looks like Fig. 3-4.

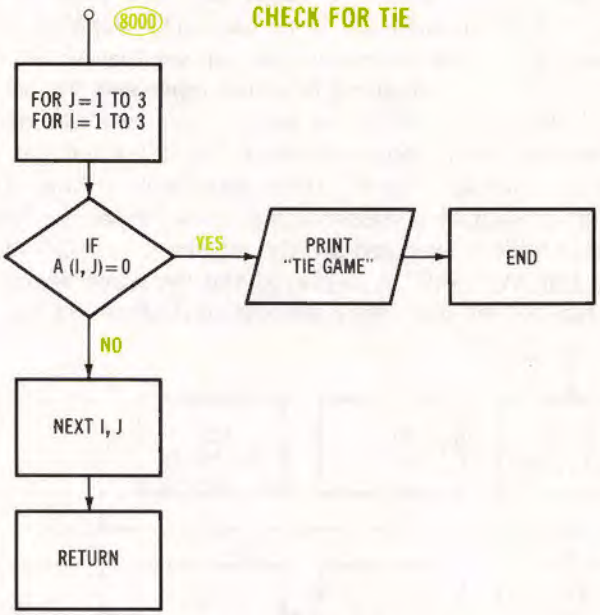


Fig. 3-4. Subroutine to check for tie game.

Check for a Winner

Finally our TIC-TAC-TOE program must be able to tell if a player has won, and end the game. To accomplish this we must be able to tell if the computer has won or the opponent has won. We can tell if a player has won by adding up all the columns, rows, and diagonals and seeing if the sum of any of these is +3. We do the same thing for the computer, but a sum of −3 indicates that it has won. The code for doing this is shown below. R is used to hold the sum of a row, C holds the sum of a column, and D1 and D2 hold the sums for the two diagonals of the board. Here is the program:

```
8997 REM
8998 REM          WIN CHECK
8999 REM
9000 LET D1=0: LET D2=0
9010 FOR I=1 TO 3: LET D1=A(I,I)+D1: LET D2=A(I,4-I)+D2
9020 LET R=0: LET C=0
9030 FOR J=1 TO 3: LET R=A(I,J)+R: LET C=A(J,I)+C: NEXT J
9040 IF D1=-3 OR D2=-3 OR R=-3 OR C=-3 THEN 9070
9050 IF D1=3 OR D2=3 OR R=3 OR C=3 THEN 9080
9060 NEXT I:RETURN
9070 PRINT TAB(20) "COMPUTER WINS": GOTO 20
9080 PRINT TAB(20) "YOU'RE THE WINNER": GOTO 20
```

Notice in line 9010 that we add up the diagonal D1 with D1=A(I,I)+D1. If I varies from 1 to 3 then we make D1 the sum of the squares A(I,I)+A(2,2)+A(3,3). Likewise D2 is set up so D2=A(I,4−I)+D2. If you follow the values for I in this expression you will see that we are adding up the squares like this A(1,3)+A(2,2)+(3,1) which represents the second diagonal on the board. This nicely illustrates the way we can use arrays by simply making expressions inside the parentheses. The rows and columns are summed in a similar manner. The IF...THEN statements in lines 9040 and 9050 are used to tell if we have a winner and if so the respective PRINT statement in lines 9070 or 9080 is executed and the message "THE COMPUTER WINS" or "YOU'RE THE WINNER" is displayed and the game begins over.

The flowchart for the win check subroutine is shown in Fig. 3-5.

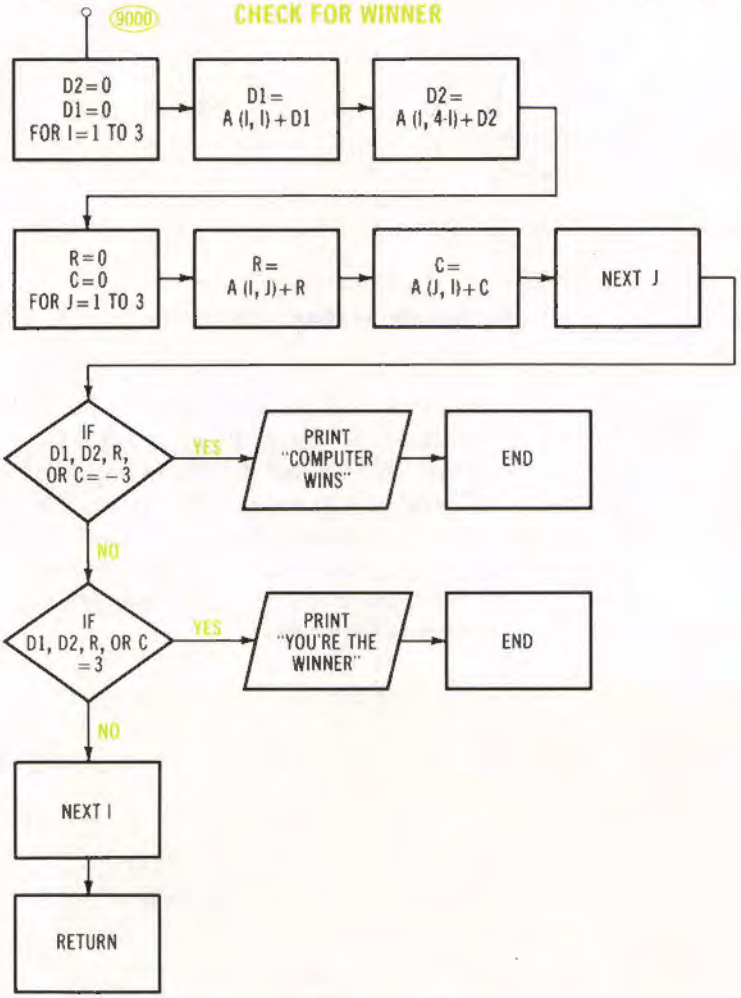


Fig. 3-5. Subroutine to check for winner.

Input Players Move

What we have now are the three simple subroutines for "housekeeping" the game. Looking at our flowchart again we now need some code that allows the user to enter his or her move. The computer must check the entry for legality and make sure the numbers are in the proper range and that the square isn't already filled. Then the square must be filled (or the player is requested to make the entry over because it is an illegal move) and the three subroutines we developed must be called.

Here is the code for inputting the player's move. (Note that because the input part of the program is not a subroutine we will place it in the beginning part of the program, with line numbers in the hundreds.)

```
298 REM            INPUT PLAYERS MOVE
299 REM
300 PRINT: INPUT "ENTER THE ROW, COLUMN FOR YOUR MOVE"; I,J
310 PRINT
320 IF I < 1 OR I > 3 OR J < 1 OR J > 3 THEN 360
325 IF A(I,J) < > 0 THEN 365
330 LET A(I,J)=1
335 GOSUB 7000     'DISPLAY BOARD
340 GOSUB 8000     'CHECK FOR TIE
345 GOSUB 9000     'CHECK FOR WINNER
350 GOTO 60       'COMPUTERS TURN
360 PRINT TAB(20) "YOUR NUMBERS ARE ILLEGAL": GOTO 300
365 PRINT TAB(20) "SORRY THAT SQUARE IS FILLED": GOTO 300
```

In this section of the program the desired square must be entered as two numbers between 1 and 3. The numbers must be separated by commas. The first number is the row and the second is the column of the square you wish to fill. After checking with statements 320 and 325, if an out of range number or filled square is found in the element the player specified, the computer prints out the respective message and the INPUT statement is repeated. If the numbers are legal, then statement 330 sets the desired element in the board array to a 1 and executes the three subroutines so the board is displayed and the game is checked for a tie or a winner.

Note that we also used a special form of remark statement. The single quote character on a line after a statement indicates the following message after the quote is a comment or remark. This sometimes can make our programs easier to read and can save us from writing a separate line for each remark.

Now, even though the program is hardly finished, it is good practice to run it and see if the code you have written checks out and works correctly. In our case we have the three subroutines and the input from the player sections done, so we can at least check that the computer puts the players' marks (1s) in the correct squares. To check the subroutines that look for a tie or a winner we can add some statements so the program branches back to the input sec-

tion and allows us to fill the board array. When we get three marks in a row the subroutine that checks for wins should operate and print out the message "YOU'RE THE WINNER."

To do this program checkout we temporarily add a line:

```
60 GOTO 300
```

which causes the computer to go back to the input routine after it has implemented all the subroutines.

Here is a run of the program so far:

```
RUN
ENTER ROW, COLUMN FOR YOUR MOVE ? 3,3
                              0  0  0
                              0  0  0
                              0  0  1
ENTER ROW, COLUMN FOR YOUR MOVE ? 2,2
                              0  0  0
                              0  1  0
                              0  0  1
ENTER ROW, COLUMN FOR YOUR MOVE 2,2

                  SORRY THAT SQUARE IS FILLED
ENTER ROW, COLUMN FOR YOUR MOVE ? 0,0
                  YOUR NUMBERS ARE ILLEGAL
ENTER ROW, COLUMN FOR YOUR MOVE ? 1,1
                              1  0  0
                              0  1  0
                              0  0  1
                  YOU'RE THE WINNER
```

The computer found that the square was filled.

The computer found that the numbers were illegal.

The WIN CHECK subroutine worked.

As we can see in the program output, everything seems to check out properly. The next logical step in the development of the program is the computer's move algorithm.

Computer Move Algorithm

In order for the computer to make an intelligent move it must be able to analyze the current board array and decide the best move to make in light of

the squares that are occupied. In essence the computer program must mimic the way we would go about making a decision to move. If you examine the thought process carefully around playing TIC-TAC-TOE you can discover that a certain series of steps, almost like a formula, can be determined. One way to express the series of steps so we can devise a program around it is in "algorithmic" form. First, we must identify the specific steps involved in making a good TIC-TAC-TOE move.

1st step Is there any line (row, column, or diagonal) that has two of your men in it and the third square in the line empty? If yes, fill it to win; if no, go to 2nd step.

2nd step Is there any line which has two of the opponent's men in it and an empty square in the line? If yes, then fill it to block the move from winning. If there is more than one line in this form, take any move because you can't win unless a mistake is made by the opponent. If no line in this form exists, go to the 3rd step.

3rd step Is there any line with one of your men and two empty squares? If yes, fill one of the squares to develop a potential winning line. If no, go to 4th step.

4th step Is there any line with one of the opponent's men in it and two empty squares? If yes, fill one of the squares to block. If no, go to 5th step.

5th step If you are this far, find an empty square and fill it with your man.

Now, to implement this series of steps as an algorithm in BASIC we must figure out a way to determine the various conditions for the lines in the program and, based on what we find, fill the respective empty square. One way to do this is to look at the board array mathematically. That is, for each line condition in the five steps there is a unique mathematical sum to the elements in the lines. For example, in step 1 a line meeting this condition would have a sum, if you added up all the elements in that line, of +2. The only way we could get the sum of the elements in a line to add up to +2 (assuming the men are either +1 or −1) is for two squares to contain +1s and one square to contain a zero (0), meaning it is empty. Thus we can loop through all the lines on the board (that is three rows, three columns, and two diagonals) and if we find any line equal to 2, then we only need to find and fill the empty square in that line.

Look at step two. It is just like step 1 except we are looking for a sum of −2 to block a potential loss. The only way we could get a line to add up to −2 is if two squares are −1 and one square is empty.

The third step involves searching for a sum of +1. This condition could occur if two squares were empty and one square was a +1, or it could happen if two squares were +1 and one was a −1. We could still use the same

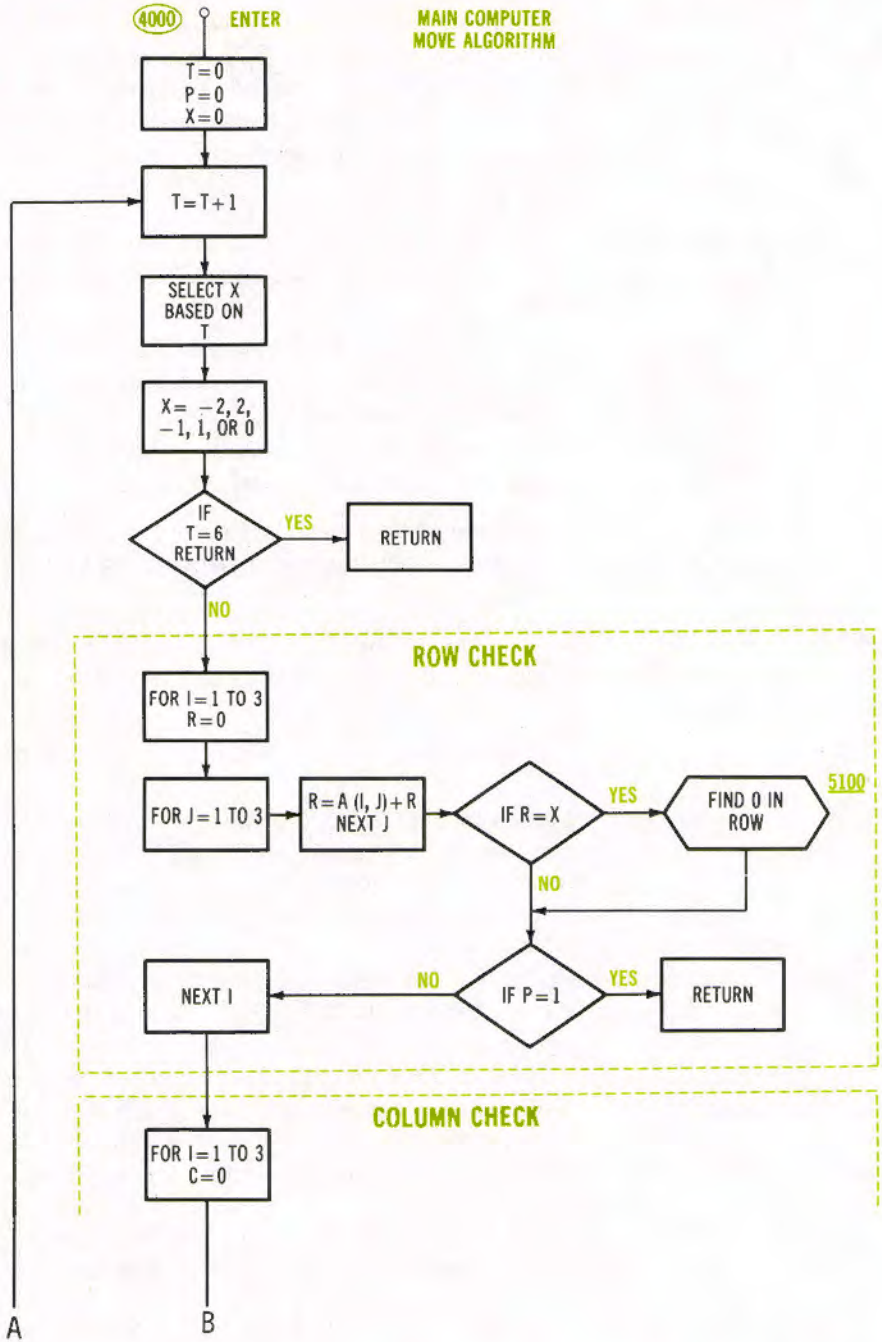
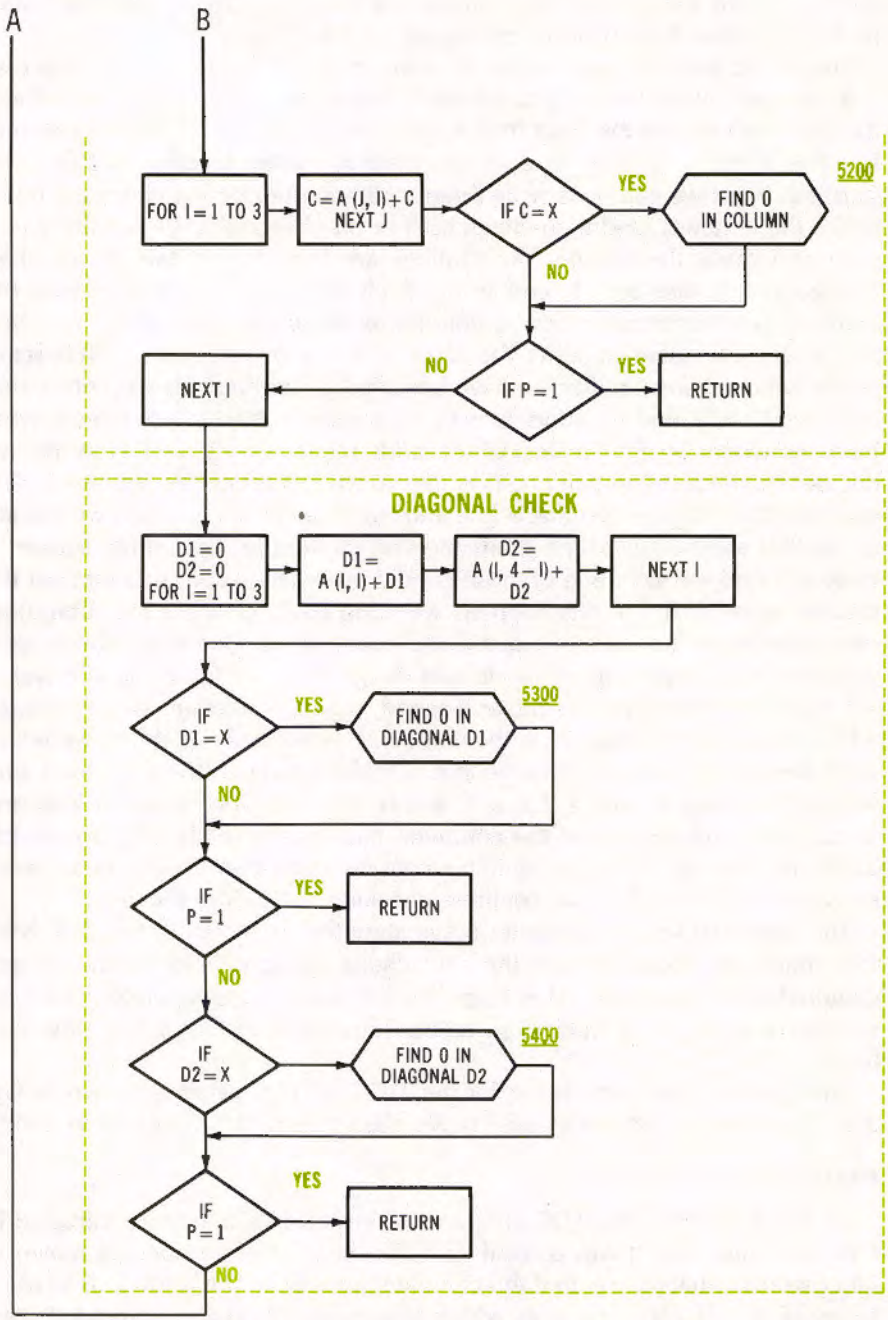


Fig. 3-6. Main computer



move algorithm.

kind of mathematical process, except we must arrange to continue after finding the condition of no empty square to fill.

The fourth and fifth steps are easily seen to be just the same logic process.

In summary what we want is a routine that allows us to check sequentially through each step of the algorithm, looking for each of the conditions (a sum in a line of +2, −2, +1, −1, or 0) and making a move based on the first one satisfied. What we can do is write three separate routines that check the rows, check the columns, and then check both of the diagonals. We put these in a loop and make the number the routines are looking for start at −2, then change to +2, then to −1, and so on. Each of the routines will contain the ability to search through a row, a column, or the diagonals looking for a line that adds up to some value X. The value of X will be set by IF...THEN statements which change it after we have executed all the routines with one value of X and have found no conditions to fill a square. Also to consider is what happens when we find a line which adds up to 2, +2, etc? How do we decide which square in that line is empty so we can fill it? The solution is that once we have determined that a line adds up to a certain number, we branch to another subroutine which scans the line looking for the empty square. If none is found we have one of those conditions where a sum was met but the squares were all full. If this happens we jump back to where the subroutine sent us to check for an empty square and continue with the move checking. If we do find an empty square we do two things. First, we fill the square with a −1 mark for the computer's move. Second, we set a software switch (a flag), which is a variable called P, to the value 1. P is normally 0. When we return from the routine now, the routine that is looking through the lines for a sum will check to see if switch P is a 1. If it is, then we don't need to look any further for a move because the computer has already made one. So we RETURN to the main program and implement the steps after the computer move subroutine. If P is still 0 we continue checking for a valid move.

The flowchart for the computer move algorithm is shown in Fig. 3-6. Note that there are boxes around the row check portion, column check, and diagonal check portions. This helps us see how things operate. The subroutines used by it for finding an empty square are in Fig. 3-7 in flowchart form.

The complete program listing for the TIC-TAC-TOE game is shown in Fig. 3-8. The code for the computer's move algorithm is in lines 4000 to 5900.

Finishing the Program

To finish the TIC-TAC-TOE program we must add a few more things to it. First, we must have a way to deal with the situation of an opening move; in other words, what do we do if the computer goes first? The solution is to use a series of IF...THEN statements which (assuming this is the computer's first turn) cause the favored strategies of grabbing the center square first to be implemented. We also must consider the condition of the computer's second move if the player has occupied the center square first. In this case the

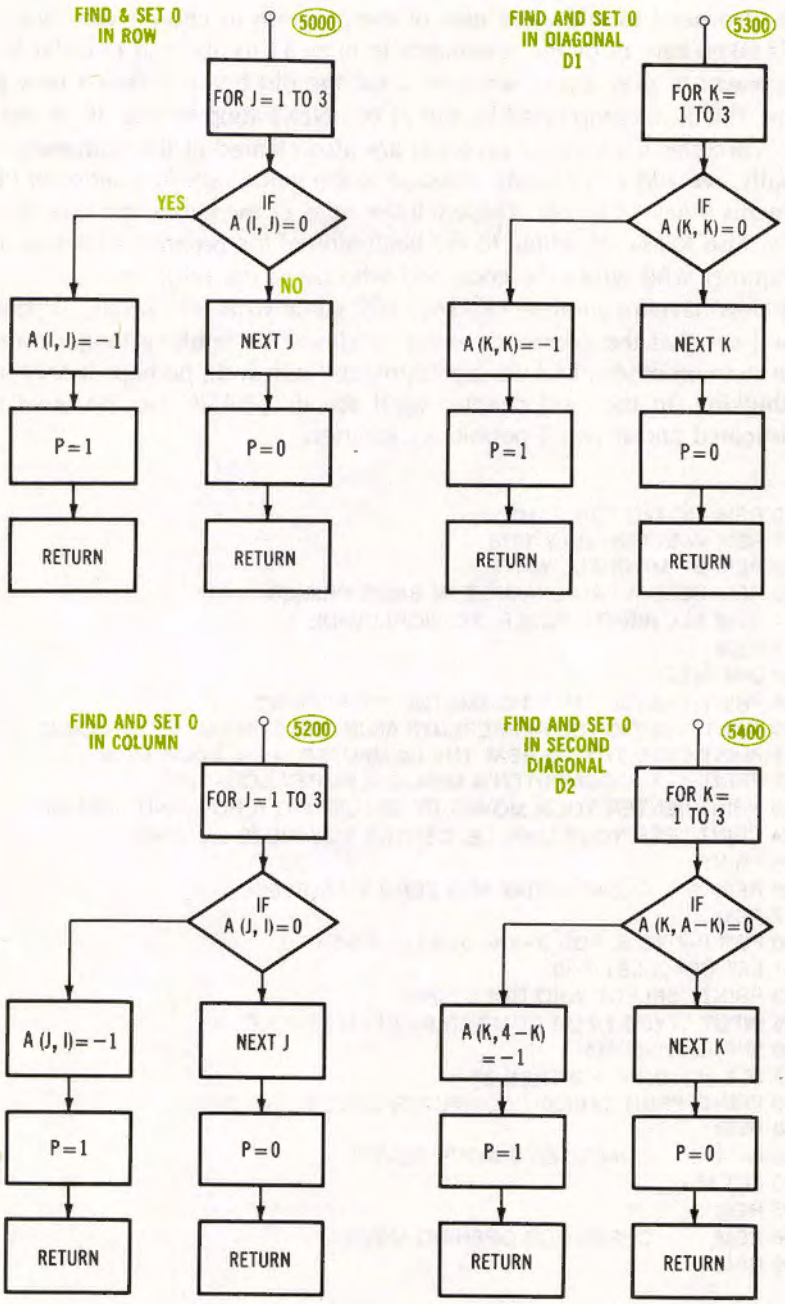


Fig. 3-7. Subroutines to find empty square and fill it (part of computer move algorithm).

IF...THENs must occupy a corner square. Lines 100 to 150 in the listing in Fig. 3-8 take care of this.

We also need to allow the user of the program to choose who goes first. This is taken care of by the statements in lines 31 to 50. And in order for our game player to play again, we must clear the old board before a new game begins. This is accomplished by the FOR...NEXT loop in line 30 of the program. Variables used in the program are also cleared at this statement.

Finally, we add a beginning message to the game user in a series of PRINT statements (lines 15 to 24). These tell the rules of the game and how to enter moves. Also REMs are added to the beginning of the program to define to the programmer who wrote the code and who owns the program.

We now have a complete TIC-TAC-TOE game written in BASIC. If you play it you'll see that the computer seems to almost be thinking longer on some moves than on others. Did we say "thinking?" Oh well, perhaps it only seems like thinking. In the next chapter we'll see that BASIC can be even more sophisticated and if you'll permit . . . smarter.

```
 0 REM TIC-TAC-TOE GAME
 1 REM WRITTEN: JULY 1978
 2 REM BY MITCHELL WAITE
 3 REM USED AS AN EXAMPLE IN BASIC PRIMER
 4 REM ALL RIGHTS RESERVED WORLDWIDE
 5 REM
10 DIM A(3,3)
15 PRINT TAB(20) "***** TIC-TAC-TOE *****":PRINT
20 PRINT "TIC TAC TOE REPRESENTS AN INTERESTING GAME OF LOGIC.":
21 PRINT "YOU TRY TO BEAT THE COMPUTER. +1 = YOUR MEN."
22 PRINT "-1 = COMPUTER'S MEN. 0 = EMPTY SQUARE"
23 PRINT "ENTER YOUR MOVES BY SPECIFYING A ROW AND COLUMN"
24 PRINT "FOR YOUR MAN, I.E. CENTER SQUARE IS 2,2":PRINT
25 PRINT
26 REM       CLEAR ARRAY AND ZERO VARIABLES
27 REM
30 FOR I=1 TO 3: FOR J=1 to 3: A(I,J)=0:NEXT J,I
31 LET C9=0: LET F=0
32 PRINT "SELECT WHO GOES FIRST"
35 INPUT "TYPE 1 FOR COMPUTER, 2 FOR YOU ";F
40 IF F=2 THEN 215
45 IF F < 1 OR F > 2 THEN 30
50 PRINT: PRINT TAB(20) "COMPUTER GOES FIRST":PRINT
58 REM
59 REM       COMPUTER'S MOVE STARTS
60 LET M=0
97 REM
98 REM       CHECK FOR OPENING MOVE
99 REM
```

Fig. 3-8. Complete program listing for TIC-TAC-TOE game.

```
100 IF A(2,2) < > 0 THEN 115
110 LET A(2,2)=-1: GOTO 200
115 IF C9 > =1 THEN 155
120 IF A(1,1) < > 0 THEN 130
125 LET A(1,1)=-1: GOTO 200
130 IF A(1,3) < >0 THEN 140
135 LET A(1,3)=-1: GOTO 200
140 IF A(3,3) < > 0 THEN 150
145 LET A(3,3)=-1: GOTO 200
150 LET A(3,1)=-1: GOTO 200
152 REM
153 REM       CALL COMPUTER MOVE ALGORITHM
154 REM
155 GOSUB 4000
200 PRINT "COMPUTERS MOVE": LET C9=C9+1
213 REM
214 REM       GO DISPLAY THE BOARD
215 GOSUB 7000
229 REM       CHECK FOR A TIE
230 GOSUB 8000
239 REM       CHECK FOR A WINNER
240 GOSUB 9000
298 REM       INPUT PLAYERS MOVE
299 REM
300 PRINT :INPUT "ENTER ROW,COLUMN FOR YOUR MOVE ":I,J
310 PRINT
320 IF I < 1 OR I > 3 OR J < 1 OR J > 3 THEN 360
325 IF A(I,J) < > 0 THEN 365
330 LET A(I,J)=1
335 GOSUB 7000     'DISPLAY BOARD
340 GOSUB 8000     'CHECK FOR TIE
345 GOSUB 9000     'CHECK FOR WINNER
350 GOTO 60        'COMPUTERS TURN
360 PRINT TAB(20) "YOUR NUMBERS ARE ILLEGAL": GOTO 300
365 PRINT TAB(20) "SORRY THAT SQUARE IS FILLED": GOTO 300
3997 REM
3998 REM       COMPUTER MOVES ALGORITHM
3999 REM
4000 LET T=0: LET P=0: LET X=0
4010 LET T=T+1: IF T=1 THEN LET X=-2
4015 IF T=2 THEN LET X=2
4020 IF T=3 THEN LET X=-1
4025 IF T=4 THEN LET X=1
4030 IF T=5 THEN LET X=0
4035 IF T=6 THEN RETURN
4097 REM
4098 REM       ROW CHECK FOR -2,+2,-1,+1,0
4099 REM
4100 FOR I=1 TO 3: LET R=0
4110 FOR J=1 TO 3: LET R=A(I,J)+R: NEXT J: IF R=X THEN GOSUB 5100
```

Fig. 3-8—cont. Complete program listing for TIC-TAC-TOE game.

```
4120 IF P=1 THEN 5900
4130 NEXT I
4197 REM
4198 REM       COLUMN CHECK FOR -2,+2,-1,+1,0
4199 REM
4200 FOR I=1 TO 3: LET C=0
4210 FOR J=1 TO 3: LET C=A(J,I)+C: NEXT J: IF C=X THEN GOSUB 5200
4220 IF P=1 THEN 5900
4230 NEXT I
4297 REM
4298 REM       DIAGONAL CHECK FOR -2,+2,-1,+1,0
4299 REM
4300 LET D1=0: LET D2=0
4310 FOR I=1 TO 3: LET D1=A(I,I)+D1: LET D2=A(I,4-I)+D2: NEXT I
4320 IF D1=X THEN GOSUB 5300
4430 IF P=1 THEN RETURN
4450 IF D2=X THEN GOSUB 5400
4460 IF P=1 THEN RETURN
4470 GOTO 4010
4999 REM
5000 REM       FIND IF A 0 EXISTS IN THE "I-TH" ROW AND MAKE -1
5001 REM
5100 FOR J=1 TO 3: IF A(I,J)=0 THEN 5120
5110 NEXT J: LET P=0: RETURN
5120 LET A(I,J)=-1: LET P=1: RETURN
5197 REM
5198           FIND IF A 0 EXISTS IN THE I-TH COLUMN, ETC.
5199 REM
5200 FOR J=1 TO 3: IF A(J,I)=0 THEN 5220
5210 NEXT J: LET P=0: RETURN
5220 LET A(J,I)=-1: LET P=1: RETURN
5297 REM
5298 REM       FIND IF A 0 EXISTS IN THE DIAGONAL D1, ETC.
5299 REM
5300 FOR K=1 TO 3: IF A(K,K)=0 THEN 5320
5310 NEXT K: LET P=0: RETURN
5320 LET A(K,K)=-1: LET P=1: RETURN
5397 REM
5398 REM       FIND IF A 0 EXISTS IN THE DIAGONAL D2, ETC.
5399 REM
5400 FOR K=1 TO 3: IF A(K,4-K)=0 THEN 5420
5410 NEXT K: LET P=0: RETURN
5420 LET A(K,4-K)=-1: LET P=1: RETURN
5430 PRINT "STOPPED AT 5430": END
5900 RETURN
6997 REM
6998 REM       DISPLAY BOARD
6999 REM
7000 PRINT:PRINT
7005 FOR I=1 TO 3:PRINT TAB(20): FOR J=1 TO 3:PRINT A(I,J);" ";
```

Fig. 3-8—cont. Complete program listing for TIC-TAC-TOE game.

```
7010 NEXT J: PRINT: NEXT I: PRINT: PRINT: RETURN
7020 RETURN
7997 REM
7998 REM        TIE CHECK
7999 REM
8000 FOR I=1 TO 3: FOR J=1 TO 3: IF A(I,J)=0 THEN 8020
8010 NEXT J: NEXT I: PRINT TAB(20) "TIE GAME": PRINT: GOTO 20
8020 RETURN
8997 REM
8998 REM        WIN CHECK
8999 REM
9000 LET D1=0: LET D2=0
9010 FOR I=1 TO 3: LET D1=A(I,I)+D1: LET D2=A(I,4-I)+D2
9020 LET R=0: LET C=0
9030 FOR J=1 TO 3: LET R=A(I,J)+R: LET C=A(J,I)+C: NEXT J
9040 IF D1=-3 OR D2=-3 OR R=-3 OR C=-3 THEN 9070
9050 IF D1=3 OR D2=3 OR R=3 OR C=3 THEN 9080
9060 NEXT I: RETURN
9070 PRINT TAB(20) "COMPUTER WINS": PRINT: GOTO 20
9080 PRINT TAB(20) "YOU'RE THE WINNER": PRINT: GOTO 20
```

Fig. 3-8—cont. Complete program listing for TIC-TAC-TOE game.

REVIEW QUESTIONS

1. Explain why a DIM statement is necessary.

2. Will the following program cause an error?

   ```
   10 INPUT L,M
   20 DIM A(L,M)
   ```

3. How many elements are there in array A, after you execute the statement DIM A(4,4,4)?

4. What will happen if you execute the following statement?

   ```
   20 DIM A(20,20,20,20)
   ```

5. What will happen if you execute this statement?

   ```
   10 DIM A(20,-20)
   ```

6. If you execute the following statement, a) how many strings can you store in array A$, and b) how long will they be?

   ```
   DIM A$(2,2)
   ```

7. Write a program which will put the numbers from 1 to 24 into the elements of the array defined by the following statement. You can put

the numbers into the array in any arrangement that you find convenient. (Don't use the zero subscripts, however.)

DIM X(4,3,2)

8. Write a program which will read the numbers from 1 to 24, in order, out of the array X(4,3,2) which you set up in question 7, and PRINT them out on the screen.


DUNLOP

CHAPTER

4

Adding More Power

Because BASIC is used by so many beginners in the computing field, it is often thought of as a "simple language" that is not capable of doing very sophisticated tasks. This is far from the truth. Although BASIC lends itself very well to simple programs, it can also be used for quite complex programs as we shall learn in this chapter. In fact, today there are many computers being used in all kinds of application areas which have been programmed in BASIC. Generally, what happens is that a particular computer manufacturer will modify the BASIC language for his computer so that it is even better at doing the specific kind of job required. There are several forms of "business BASIC" languages available today. These are versions of BASIC which lend themselves especially well to processing large amounts of information, and producing long report printouts for business accounting purposes.

Ordinary garden variety BASIC has quite a bit of computing power in itself. In this chapter we will discuss some of the more commonly used capabilities of the language. They are:

DATA and READ statements which allow information to be defined by the program.

Numeric functions which can provide extensive mathematical capability.

String functions for manipulating characters and words.

DATA AND READ STATEMENTS

Many times it is necessary or advantageous to include some type of fixed information right within the body of the program. This information may be too lengthy to be INPUT into the program through the normal manner, and might always be the same anyway. This would mean that each time the program was used, the same information would have to be INPUT at the beginning so

that the program could operate properly. Examples of fixed information include such things as tax tables, prices of merchandise, dates, names, instructions, and the like. In order for information to be used by any program, it must be stored with a variable name of some sort. Remember that numeric information is stored using one format, while string information is stored using another. If there are only a few items of fixed information which must be included in a program, then these might be easily set up using LET statements. However, when the volume of information starts to increase, there has to be a better way.

The purpose of the DATA and READ statements is to allow large volumes of fixed information to be set up within a program with relative ease. The information is included as part of the DATA statement, and it is transferred into a variable by the READ statement when the program is run. Once this has been done, the variable can be used throughout the program. The format for the DATA statement is as follows:

```
DATA (information separated by commas)
```

The format of the READ statement is like this:

```
READ (list of variables separated by commas)
```

Below are some examples of various types of DATA and READ statements.

```
DATA 1,2,3,4,5
READ A,B,C,D,E
```

This pair of statements would cause the variables A, B, C, D, and E to be initialized to the values of 1, 2, 3, 4, and 5 respectively. The same thing could have been done using the following DATA and READ statements.

```
DATA 1,2,3,4,5
READ A,B,C
READ D,E
```

This is because BASIC uses an internal pointer to keep track of where in the DATA statement to get the next element of data. This pointer is set to 1 at the beginning of the program and then is incremented each time that a variable is READ from a DATA statement. After the first READ statement above has set up the variables A, B, and C, the pointer is pointing at the next data element in

the DATA statement. When the next READ statement is encountered by the program, the variables D and E are READ from the same DATA statement beginning at the fourth data element. This procedure can go on and on as long as there is enough data in the DATA statements to fill up the variables that are to be read. If all of the data gets used up and another READ is attempted, an error will result.

There is one other BASIC statement associated with the DATA and READ statements. It is the RESTORE statement. What it does is to reset the internal pointer which BASIC uses to sequentially get the information from the DATA statement. After a RESTORE statement is executed, the next READ statement will access the first DATA statement in the program regardless of where the internal pointer happened to be pointing prior to the RESTORE.

Here's an example of the use of the RESTORE statement:

```
10 DATA 1,2,3,4,5
20 READ A,B,C
30 RESTORE
40 READ D,E,F
50 PRINT A;B;C;D;E;F
RUN
 1 2 3 1 2 3
```

This program reads the first three values from the DATA statement into variables A,B, and C. Then, the RESTORE statement moves the internal pointer back to the beginning of the data, so that READ D,E,F again reads the first three values from the DATA statement, this time into D,E and F. When A,B,C,D,E, and F are printed out in line 50, the result is the first three values of the DATA statement, repeated twice.

Let us look at some other DATA and READ combinations.

```
DATA 12,14.5,18,75.66,100,500,1000
READ A1,A2,A3,A4,A5,A6,A7
```

In this example, the variable A1 is initialized to a value of 12, A2 is set to a value of 14.5, A3 is set to 18, and so on.

```
DATA "JAN","FEB","MAR","APR","MAY","JUN"
READ A$,B$,C$,D$,E$,F$
```

In this example, the string variables A$ through F$ are initialized by the READ statement to the names of the months which are described in the DATA statement.

Probably the most useful form of the DATA and READ statements is that of initializing arrays. As we learned in the previous chapter, an array is a group of variables which usually have something in common, and are all referenced by the same variable name. A specific element of an array may then be referenced through the use of a subscript, such as A(5), M$(2) and so on.

It should be noted here that there are some differences in the way that different versions of BASIC handle string arrays. Some require that the length of each string within the array be defined in the DIM statement. For example,

```
DIM A$(12)
```

would define a string containing 12 characters. The same DIM statement in other BASICS would define a string array that contained 12 strings, each with a dynamic length determined by the actual number of characters stored in it at any given time, up to a maximum of 255. For the sake of the examples in this book, we will adhere to the latter of these conventions. That is:

```
DIM M$(2)
```

defines an array of two strings of dynamic length. In order to use DATA and READ to initialize an array, we must first make sure that the array has been defined with a DIM statement. Then, there are two ways of getting the information from the DATA statement into the array variables. We will discuss both of them below.

```
10 DIM M$(12)
20 DATA "JAN","FEB","MAR","APR","MAY","JUN"
30 DATA "JUL","AUG","SEP","OCT","NOV","DEC"
40 READ M$(1),M$(2),M$(3),M$(4),M$(5),M$(6)
50 READ M$(7),M$(8),M$(9),M$(10),M$(11),M$(12)
```

First of all, notice here that we have split up the twelve months into two DATA statements, and we have also split up the twelve elements of the array M$ into two READ statements. This is possible due to the fact that when the first READ statement in a program is encountered, the information is gotten from the first DATA statement found in the program. The variables in the READ statement are initialized until they have all been taken care of. When the next READ statement is encountered, the variables in its list are initialized beginning with the next DATA statement in the program. Notice that the first six elements of the array M$ are initialized, and then the last six are initialized on line 50.

This method will work just fine. However, it does not take full advantage of the convenience of arrays. In the next example, we will achieve the same thing, but in a different way.

```
10 DIM M$(12)
20 DATA "JAN","FEB","MAR","APR","MAY","JUN"
30 DATA "JUL","AUG","SEP","OCT","NOV","DEC"
40 FOR I = 1 to 12
50 READ M$(I)
60 NEXT I
```

Notice the difference here? Instead of spelling out each and every array element, we have used a FOR...NEXT loop to cause the index I to go from 1 to 12. Then the READ statement at line 50 reads only one element of the array M$ from the DATA statements. Notice also here that there are two DATA statements and only one READ statement with only one variable name in its list. However, the READ statement is in a loop so that it is actually executed 12 times. Each time that it is executed, it gets the next element of information from the DATA statements.

Both of the above methods will work just fine. The first method will probably work better when there are only a few variables which must be initialized. But when the number of variables starts to increase, the second method using the FOR...NEXT loop will definitely be easier to program. Imagine having to initialize a hundred or more variables using the first method.

METRIC CONVERSION PROGRAM ENHANCED

Earlier in the book, we used as an example a program that converts anglo units to metric units. In order to explain some of the beginning concepts of BASIC, the approach that we used to write this program was quite simple. Now, we will use the same application as an example of the DATA and READ statements. You will notice that this will make the program much more flexible and general-purpose usable. We will use the same conversion formulas that we used in the previous example, but in a completely new way. What we are going to do is to store all of the "literal" or string information in string arrays. Also, we will store the conversion factors themselves in a numeric array. Then, we can access the string and numeric information which we will need through the use of a subscript to the arrays. So, let us set up the beginning of the program.

```
1000 REM -METRIC CONVERSION PROGRAM
1010 REM -USING DATA AND READ
1020 REM -INITIALIZE THE ARRAYS
```

```
1030 LET N = 3
1040 DIM M$(N),Q$(N),A$(N),B$(N),F(N)
```

The variable N is used to determine how many different conversions the program is capable of doing. This is set to 3 for our example, since we will use the same three conversions that the old program used, namely, "INCHES TO CENTIMETERS," "POUNDS TO KILOGRAMS," and "QUARTS TO LITERS." Others can easily be added as we shall see. At line 1040 we see the DIM statement which sets up the size of the various arrays which are going to be used in this program. They are all set up to the length defined by the variable N, and are:

N—number of functions.

M$—a string array containing the menu.

Q$—a string array containing the "input question."

A$—a string array containing the first part of the answer.

B$—a string array containing the last part of the answer.

F—a numeric array containing the conversion factors.

The next part of the program consists of the DATA statements which will be used to READ the arrays dimensioned above.

```
1090 REM -DATA FOR ARRAYS
1100 DATA "1 - INCHES TO CENTIMETERS", "ENTER LENGTH IN INCHES",
     " INCHES "," CENTIMETERS",2.54
1200 DATA "2 - POUNDS TO KILOGRAMS", "ENTER WEIGHT IN POUNDS",
     " POUNDS "," KILOGRAMS", .4536
1300 DATA "3 - QUARTS TO LITERS","ENTER VOLUME IN QUARTS",
     " QUARTS "," LITERS ",.9463
```

Here we have three DATA statements. Each one contains all of the information necessary for one of the conversion routines. The different pieces of information are separated by commas in the DATA statement. First, there is the string information which will be used to display the menu. Then, there is the question which is asked just before the quantity to be converted is entered. Next comes the two string parts of the answer which is printed, and finally there is the actual numeric conversion factor itself. Now, let us write the statements to READ this information into the proper arrays.

```
1980 REM -READ THE ARRAYS
1990 FOR I = 1 TO N: READ M$(I),Q$(I),A$(I),B$(I),F(I): NEXT I
```

This is how the information in the DATA statements is finally READ into the arrays. We use a FOR...NEXT loop to cycle the program through the DATA statements in such a way as to make sure that all of the information pertinent to each of the conversion routines is READ into its proper place in the arrays. Now, we can go ahead with the rest of the program.

```
2000 REM -DISPLAY THE MENU
2100 PRINT: PRINT "METRIC CONVERSION PROGRAM": PRINT
2200 FOR I = 1 TO N: PRINT M$(I): NEXT I
2300 PRINT "9 - TO END THE PROGRAM": PRINT
```

First, at line 2100 we PRINT the heading. Line 2200 consists of the FOR...NEXT loop which PRINTs the menu itself. Lastly, on line 2300 we must PRINT the option to end the program. At this point, we are ready to INPUT our selection.

```
3000 REM -ENTER SELECTION
3100 INPUT "ENTER YOUR SELECTION "; S
3200 IF S < 1 OR S > N THEN GOTO 9999
```

This part of the program is similar to the original version except that we are not going to need all the IF...THEN statements to decide where in the program to branch to do the actual work. Here, all we need to do is to see if the value of S is outside of the range of defined selections, and if it is, we branch to the end of the program. The next part of the program is where the actual conversion takes place.

```
4000 REM -DO THE CONVERSION
4100 PRINT: PRINT Q$(S): INPUT X
4200 PRINT X; A$(S); "IS EQUAL TO "; X*F(S); B$(S)
4300 GOTO 2000
```

And of course,

```
9999 END
```

See how it works? First, on line 4100 we PRINT the appropriate question according to the selected conversion routine. Then, we enter the value to be converted, X. On line 4200, we accomplish the same thing that the previous program did. First, we PRINT back the value to be converted, followed by the string A$ which contains the name of the units that we are converting. Then, we PRINT the literal data "IS EQUAL TO," followed by the value of the converted quantity which is arrived at by simply multiplying X by the appropriate numeric factor contained in the array F. Lastly, we PRINT the string data which contains the description of the new units.

This program works exactly the same as the one in the previous chapter. Notice, however, that it can be expanded very easily to include many other conversion routines just by changing the LET statement in the beginning

```
1000 REM -METRIC CONVERSION PROGRAM
1010 REM -USING DATA AND READ
1020 REM -INITIALIZE THE ARRAYS
1030 LET N = 3
1040 DIM M$(N),Q$(N),A$(N),B$(N),F(N)

1090 REM -DATA FOR ARRAYS
1100 DATA "1 - INCHES TO CENTIMETERS", "ENTER LENGTH IN INCHES",
     " INCHES "," CENTIMETERS",2.54
1200 DATA "2 - POUNDS TO KILOGRAMS","ENTER WEIGHT IN POUNDS",
     " POUNDS "," KILOGRAMS",.4536
1300 DATA "3 - QUARTS TO LITERS","ENTER VOLUME IN QUARTS",
     " QUARTS "," LITERS",.9463

1980 REM -READ THE ARRAYS
1990 FOR I = 1 TO N: READ M$(I),Q$(I),A$(I),B$(I),F(I): NEXT I

2000 REM -DISPLAY THE MENU
2100 PRINT: PRINT"METRIC CONVERSION PROGRAM": PRINT
2200 FOR I = 1 TO N: PRINT M$(I): NEXT I
2300 PRINT "9 - TO END THE PROGRAM": PRINT

3000 REM -ENTER SELECTION
3100 INPUT "ENTER YOUR SELECTION "; S
3200 IF S < 1 OR S > N THEN GOTO 9999

4000 REM -DO THE CONVERSION
4100 PRINT: PRINT Q$(S): INPUT X
4200 PRINT X; A$(S); "IS EQUAL TO "; X*F(S); B$(S)
4300 GOTO 2000

9999 END
```

Fig. 4-1. Listing of enhanced Metric Conversion program.

which defines the value of N, and by adding more DATA statements in the form that we have designed here. This has been a very good example of how arrays, together with the DATA and READ statements, can make a simple program into a powerful program (Fig. 4-1).

NUMERIC FUNCTIONS

Earlier in this book, we introduced three simple numeric functions: ABS, INT, and RND. In this section we will introduce some additional numeric functions which will be useful in many different programs. Most of these functions are common to all the varieties of the BASIC language. Some particular languages may have more of them, depending upon the general nature of the language. Some of the functions are strictly for use in situations of a mathematical nature, like the trigonometric functions and the square root functions. Others may be useful in a variety of different applications, including business programs and even games. We will discuss these functions rather briefly here, and show only simple examples. In actual application, you should consult the BASIC manual of your computer for a more detailed explanation of how it operates.

AND, OR, and NOT

These three functions are sometimes called the Boolean logic functions because they perform the truth-table operations on each bit of the bytes as arguments. (See Appendix A for more information about binary and other numbering systems.) The general form of their usage follows:

```
100 LET X = A AND B
200 LET X = C OR D
300 LET X = NOT Z
```

Notice that for the AND and OR functions that there are two variables involved in the operation, while for the NOT function there is only one. Let us take a look at the truth table for each of these functions. Remember that the operation is carried out at the bit level. Corresponding bits of the two arguments are used in the following way to produce the answer.

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

C	D	C OR D
1	1	1
1	0	1
0	1	1
0	0	0

Z	NOT Z
1	0
0	1



In the first truth table, notice that for the AND function, both bits must be "1" in order for their ANDed value to be "1." If either of them is "0" then the result will be "0." The OR function on the other hand will yield a "1" for a result if either of the argument bits is a "1." Only when they are both "0" will the result become "0." The NOT function, as you can see produces the complement of the argument bit. Now, let us take a look at some examples using complete 8-bit bytes as arguments.

```
1 1 0 1 0 1 1 0
0 1 0 0 1 1 1 0
---------------
0 1 0 0 0 1 1 0
```

AND function yields "1" only when both corresponding bits are "1."

```
1 1 0 1 0 1 1 0
0 1 0 0 1 1 1 0
---------------
1 1 0 1 1 1 1 0
```

OR function yields "1" if either of the corresponding bits is "1."

NOT 1 1 0 1 0 0 1 0 equals 0 0 1 0 1 1 0 1

The NOT function simply reverses all of the bits in the argument byte.

These functions AND and OR are the same ones that we used to produce compound IF...THEN statements. Their operation in that usage is not readily

apparent, and in most cases need not be a concern. Used as we have described here, however, these functions can be very useful in dealing with data at the "bit level." For example, if we are only interested in the last four bits of a given byte of data, we can use the AND function to strip off the first four bits as follows:

```
100 LET X = X AND 15
```

Suppose that X is the variable that we wish to strip down. Looking at this function at the bit level, we see that

```
    1 0 1 1 0 1 1 0   <-- Value of X to be stripped.
AND
    0 0 0 0 1 1 1 1   <-- Binary value of 15.
    ---------------
    0 0 0 0 0 1 1 0   <-- ANDed value leaves only last 4 bits.
```

Another convenient example of the use of the AND function is regarding the use of a technique known as "bit-coding." This provides a means of economically storing the status of various conditions, which can have a value of either "true" or "false." Such conditions might include the status of various programs in a business data processing system. Other examples may be whether or not a file has been updated, or a particular program run in the right sequence. One way of doing these things would be to use a whole variable for each status indicator that is needed. If there are only a few indicators required, then this will be no problem. If, however, there are to be many indicators, then the use of a whole byte to store a "1" or a "0" is going to cost memory space. To save this space, we can use individual bits in a byte to store the condition of many different things. We can then use the AND function to interrogate the bit in question, and the OR function to change the bit. Let us suppose that the third bit of a byte of data indicates the update status of a file. A "1" means that the file has been updated, and "0" means that it has not been updated. We will call the byte S and interrogate it as follows:

```
100 IF S AND 32 = 32 THEN PRINT "FILE IS UPDATED"
200 END
```

Here is how this works. Let us assume that the file has been updated. Then the bits look like this.

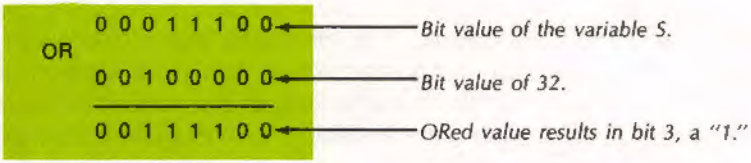


The result of the AND function yields the value of 32 which will then cause the message "FILE IS UPDATED" to be printed. If the third bit was a "0" then the result of the AND function would have yielded all "0s" and the message would not have been printed.

Now suppose that the indicator is a "0," indicating that the file has not been updated, and we wish to change it to a "1" without disturbing any of the other bits in the variable. For this we can use the OR function as follows:

```
100 LET S = S OR 32
```

The bits then look like this:



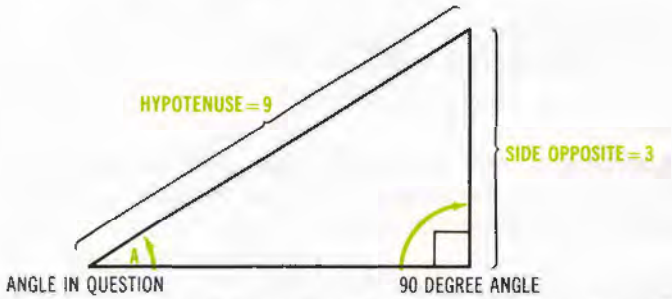
SIN, COS, and ATN

If you ever took trigonometry in high school or college then you probably remember vaguely that there were some special relationships between the sides of a right triangle and the angles themselves. These "laws of trigonometry" are very useful in many problems involving distances and relative angles. We will briefly review the basic definitions of three of the trig functions here.

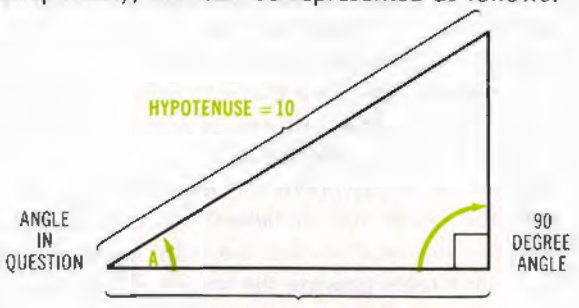
The SINE of an angle is defined as the ratio of the side of the triangle which is opposite the angle in question, divided by the hypotenuse of the right triangle. Remember that the hypotenuse is the side that is opposite the right angle. Graphically, the SINE looks like this:

$$\text{SINE(A)} = \frac{\text{side opposite}}{\text{hypotenuse}} = \frac{3}{9} = \frac{1}{3} = .3333$$



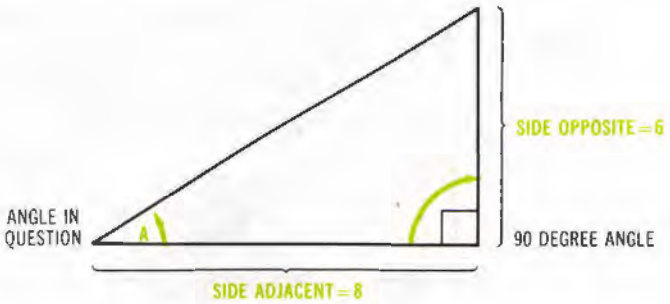


Similarly, the COSINE of an angle is defined as the ratio of the side of the triangle which is adjacent to the angle in question, divided by the hypotenuse. Graphically, this can be represented as follows:



$$\text{COS(A)} = \frac{\text{side adjacent}}{\text{hypotenuse}} = \frac{8}{10} = .8$$

The ARCTANGENT is a little different from the SINE and COSINE functions Both of them used an angle as the argument to the function in order to derive the particular ratio of the sides of the triangle in question. The ATN function works in reverse. Given the TANGENT of an angle, the ATN function will produce the angle itself. First, we should take a quick look at what the TANGENT of an angle is. It is defined as the ratio of the side opposite the angle in question, divided by the side adjacent to the angle. This looks like the following:



$$\text{TAN(A)} = \frac{\text{side opposite}}{\text{side adjacent}} = \frac{6}{8} = \frac{3}{4} = .75$$

Now, the arctangent (ATN) is just the opposite of this definition. Given that we already know what the tangent is, we can then find out what the angle is by using the ATN function like this:

The angle A = ATN(.75)

This usually returns the angle measured in radians rather than degrees. Since there are 2 pi radians in a complete circle, one radian is equal to 360 degrees divided by 2 pi, or about 57.3 degrees per radian.

In actual usage, these trig functions would be very cumbersome if we had to draw little right triangles every time that we wanted to know the sine or cosine of an angle. That is why there are books which contain large tables of these values which can be used to get the required information. If we have a computer at our disposal, which happens to support these trig functions, then we don't even need the books full of tables. We can simply program the computer to give us the information that we want using the trig functions. Most of all the BASIC languages provide the three trig functions that we have described here. Some even offer more, but these are the most commonly used, and the other trig functions can be derived from them. Using BASIC, the form of the usage of these functions is as follows:

```
100 LET X = SIN(A)
200 LET X = COS(A)
300 LET A = ATN(X)
```

Where A is the angle in radians.
Again A is the angle in radians.
Where X is the known tangent, and A will be returned as the angle in radians.

EXP and LOG

These two functions are related in the sense that they both have to do with natural logarithms. If you remember your high school math, you will recall that the logarithm of a number is the power to which the "base" must be raised to obtain the number. Natural logarithms use a special number for the base known as "base e." The value of this base is approximately 2.718 which may seem like an unusual value to base a system of logarithms on, but proves very useful in higher mathematics. To further clarify the logarithm principle:

$$\text{If } Y = e^x, \text{ then } LOG(Y) = X$$

So, the log of a number is the power to which the base (e) must be raised in order to obtain the number. This is just how the LOG function is used in a program. The typical form is:

```
100 LET X = LOG(Y)
```

The LOG function may also be used in any of the usual combinations such as imbedded in a PRINT statement, or compounded with other functions.

The EXP function is sort of the reverse of the LOG function. It will produce the result of raising the base (e) to some power. In other words:

EXP(X) is the same as e^x

The typical usage of the function is as follows:

```
100 LET Y = EXP(X)
```

One very useful application of the LOG function is regarding the conversion from one logarithm base to another. The general form of this relationship is as follows:

$$LOG_b(X) = \frac{LOG_e(X)}{LOG_e(b)}$$

That is, the logarithm of a number X in base b is equal to the natural log base (e) of the number, divided by the natural log of the base. To find the base 10 log of 100 we would write:

```
100 LET Y = LOG(100)/LOG 10
```

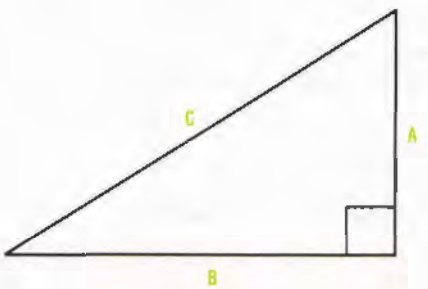
SQR Function

The SQR function is used to produce the square root of a number. Of course the number must be positive, since the square root of a negative number is not real and cannot be dealt with by this function. Remember that the square root of a number is a number which when multiplied by itself yields the original number. The general form of the SQR function is:

```
100 LET Y = SQR(X)
```

One common example of the use of the square root function has to do with some more geometry known as the Pythagorean theorem. This describes the relationship of the sides of a right triangle. Simply stated, it says that the length of the hypotenuse of a right triangle is equal to the square root of the sum of the squares of the other two sides. Using the figure below, the formula is expressed as:

$$C = \sqrt{A^2 + B^2}$$



Translating this formula into BASIC, we would use the following:

```
100 LET C = SQR(A*A + B*B)
```

Notice that to square the values of the two sides A and B, we have simply multiplied them by themselves. Some BASICs also include a numeric function

that allows a number to be raised to a power. In that case, we could use the appropriate arithmetic operator and write the expression as follows:

```
100 LET C = SQR(A^2 + B^2)
```

This would yield the same results as the statement above.

SGN Function

The function SGN which stands for "sign" is very handy for encoding the arithmetic sign of a number (+ or −). The general usage form is:

```
100 LET Y = SGN(X)
```

The variable Y will be set equal to 1 if X is positive, 0 if X is zero, or −1 if X is a negative number.

One very common use of this function is in rounding off a number to the nearest whole number. If we are going to be dealing with only positive numbers, then they could be rounded off using the following statement:

```
100 LET X = INT(X+.5)
```

By adding 0.5 to any positive number and then integerizing it with the INT function, we will obtain the nearest whole number. This does not work, however, when the number is negative. Adding 0.5 to a negative number will actually reduce its value toward zero. Therefore, we can use the SGN function as follows:

```
100 LET X = INT(X+.5*SGN(X))
```

Since multiplication takes precedence over addition, the 0.5 will be multiplied by either 1 if X is a positive number, −1 if X is a negative number, or by 0 if X is zero. This will then be added to the value of X that is then INTegerized to produce the nearest whole number.

MOD Function

Occasionally, it is useful to know what the value of the remainder of an arithmetic division is. This is the purpose of the MOD function, which is short for "modulus." The general form of the MOD function is as follows:

```
100 LET C = A MOD B
```

In the preceding example, A is divided by B, and the remainder is stored in the variable C. Some examples:

12 MOD 6=0
47 MOD 13=8
5 MOD 2=1

One use of the MOD function involves the case where several code numbers have been consolidated into one number for the sake of memory savings. For example, a date such as April 21, 1945 can be abbreviated 04/21/45. This can be represented as a number 42145. Now suppose that we want to extract the year portion of this number. We can use the MOD function as follows:

```
100 LET Y = X MOD 100
```

In this case, the variable X will contain the composite date 42145. When the MOD operation is performed, dividing by 100, the remainder 45 is stored in the variable Y. This is then the year part of the date. The rest of the date can be extracted in a similar manner. Here is the whole program:

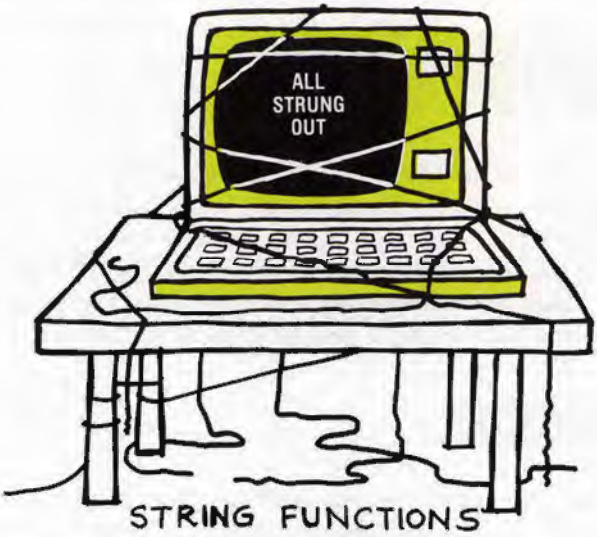
```
100 LET X = 42145
110 LET Y = X MOD 100
120 REM -SHIFT THE DIGITS RIGHT
130 LET X = INT(X/100)
140 LET D = X MOD 100
150 REM -ONE MORE TIME
160 LET M = INT(X/100)
```

STRING FUNCTIONS

Most BASIC languages provide several functions for manipulating string variables, and converting string variables to numeric variables and vice-versa. This section deals with some of the more common string functions.

ASC and CHR$

These two functions are somewhat related in that they are both used to convert between string and numeric variables. The ASC function produces the ASCII decimal value of the first character of a string variable which is given as the argument to the function. Remember that ASCII (pronounced "ass-key") is the name of a special type of coding used to represent the characters of the



alphabet and the numerals, as well as several special characters. The name stands for "American Standard Code for Information Interchange." There is a table of all of the ASCII values listed in Appendix B.

The common form of the ASC function is as follows:

```
100 LET X = ASC(A$)
```

It should be noted that the ASC function only produces the ASCII decimal value of the first character in the string. The other characters are ignored. Let us look at an example.

```
100 LET A$ = "SAN FRANCISCO"
200 LET X = ASC(A$)
300 PRINT X
```

Can you figure out what will be printed for the value of X? From the table in Appendix B, we can see that the ASCII decimal value for an upper case 'S" is 83. Since only the first character of the string A$ is used by the ASC function, the value which is printed is the ASCII decimal value for the "S" which is 83.

The ASC function is useful in situations when the program must deal with characters which are not part of the normal alphabet. For example, most computer keyboards include a key called "ESCAPE." This key usually has some special use, and does not create a regular character when it is pressed. It does however create an ASCII value of 27. If we had a program that was expecting to be INPUTing a string variable, and we wanted to see if the ESCAPE key was pressed, we could use the following statements:

```
100 INPUT A$
200 IF ASC(A$) = 27 THEN GOTO 9000
```

The above statements would first INPUT a string variable called A$ and then test the first character of that string to see if it was equal to the ESCAPE character. If it is, then the program will unconditionally branch to line 9000.

The CHR$ function operates just the reverse of the ASC function. It takes an ASCII decimal value as the argument and converts it into a one character string. The common form of the CHR$ function is as follows:

```
100 LET A$ = CHR$(X)
```

The CHR$ function is also useful for dealing with characters which are not part of the normal character set consisting of the alphabet, the numerals, and a few special characters. Most terminals have some form of audible alarm such as a bell, horn, buzzer, beeper, etc. This can be used to attract the operator's attention when displaying error messages and the like. This alarm can be activated by sending a special ASCII character to the terminal. This character cannot be enclosed in quotes like a normal string variable, so it must be generated using the CHR$ function like this:

```
100 PRINT CHR$(7)
```

When the CHR$ function converts the decimal value of 7 into an ASCII character and sends it to the terminal, the result will be that the terminal recognizes it as a command to beep the horn. You should notice that the string which is created by the CHR$ function is only one character in length.

LEFT$, MID$ and RIGHT$

These three string functions are all used to break up string variables into substrings. The LEFT$ function allows the program to move a certain number of characters from the left end of a string variable into another string. Likewise, the RIGHT$ function allows the program to move characters from the right end of a string into another string. The MID$ function, as we might guess from its name, allows characers to be moved out of the middle of one string, and into another string. Let us take a closer look at each of these functions.

The general form of the LEFT$ function is:

```
100 LET B$ = LEFT$(A$,N)
```

Notice that there are two arguments inside the parentheses of the LEFT$ function. In the first, A$ is the string variable upon which the function is to be performed. The variable N is used to define just how many characters on the left end of A$ will be moved into the string B$. For example:

```
100 LET A$ = "MILL VALLEY"
200 LET N = 4
300 LET B$ = LEFT$(A$,N)
400 PRINT B$
```

Can you figure out what will be printed by the PRINT statement at line 400? Since the LEFT$ function will only move the leftmost N characters of the string A$ into the new string B$, only the word 'MILL' will be printed. By the way, the characters in the string A$ are not disturbed. They are just the same after the LEFT$ function as they were before.

There are some restrictions on the variable N with this function. The value of N must never be negative, or greater than 255. If N is greater than the current length of the string argument, then the entire string will be moved. If N is equal to zero, the result will be a string with no length.

The RIGHT$ function works just the opposite of the LEFT$ function. That is, it moves N characters from the right end of the argument string. The general form is:

```
100 LET B$ = RIGHT$(A$,N)
```

Here again, the same rules apply as for the LEFT$ function. For example:

```
100 LET A$ = "GRAND TOTAL"
200 LET N = 6
300 LET B$ = RIGHT$(A$,N)
400 PRINT B$
```

Since this function will move the rightmost N characters from the argument string, we would expect that after the function was performed, the string B$ would contain "TOTAL." Notice that since the rightmost 6 characters are moved, the blank preceding the word TOTAL is also moved.

The MID$ function operates just a bit differently from the LEFT$ and RIGHT$ functions. Basically, it is still used to move characters from one string into another string, but the difference is that it takes them from the middle of a string. In order to do this, the MID$ function requires one additional argument. The general form of the MID$ function is as follows:

```
100 LET B$ = MID$(A$,N,M)
```

Here, the variable N is used to define the starting position within the string A$ from which characters will be moved. This is relative to the left end of the string. The variable M is used to define the number of characters to be moved. For example:

```
100 LET A$ = "GOLDEN GATE BRIDGE"
200 LET N = 8: LET M =4
300 LET B$ = MID$(A$,N,M)
400 PRINT B$
```

Because of the values of the variables N and M, the MID$ function will begin with the eighth character in the string A$ and move four characters into the string B$. After this is done, the string B$ will contain just the word "GATE." The same restrictions apply to the variable N as for the other functions LEFT$ and RIGHT$. The variable M also has the same restrictions of value.

Here again, we must stop and recognize that due to the fact that some versions of BASIC require that the string variables have a declared length, the LEFT$, MID$, and RIGHT$ functions are not permitted. Therefore, there are other means of extracting part of a string. For example,

```
LET B$ = LEFT$(A$,N)
```

would be accomplished by;

```
LET B$ = A$(1,N)
```

This moves the characters from A$ to B$ beginning with the first character, and including the Nth character. Likewise,

```
LET B$ = RIGHT$(A$,N)
```

is done using;

```
LET B$ = A$(LEN(A$)-N,+1,LEN(A$))
```

The LEN function shown in the above example produces the current length of the string given as the argument. We will explore the LEN function later in this chapter. Finally,

```
LET B$ = MID$(A$,N,M)
```

is written in some BASICs as

```
LET B$ = A$(N,N+M-1)
```

In general, these alternate forms of the string manipulation statements all use the same type of arguments, namely:

```
LET B$ = A$ (first character position, last character position)
```

STR$ and VAL

These two functions are very useful to the BASIC programmer, especially when programming a business application. They are used to convert string variables into numeric variables, and numeric variables into strings. The form of the STR$ function is:

```
100 LET A$ = STR$(X)
```

What this function does is evaluate the numeric variable X and put the characters into the string A$ which represents the value of X. For example:

```
100 LET X = 357
200 LET A$ = STR$(X)
300 PRINT A$
```

The string A$ is set up by the STR$ function to contain the characters "357" as a result of evaluating the numeric variable X. Now, the string can be printed out in any way that the programmer desires instead of leaving the choice to the BASIC way of printing numeric variables.

The VAL function is used to accomplish the opposite of the STR$ function. It takes a string variable containing characters which represent a numeric value, and evaluates them so as to create a numeric variable. The general form of the VAL function is:

```
100 LET X = VAL(A$)
```

An example of the operation of the VAL function is:

```
100 LET A$ = "25.79"
200 LET X = VAL(A$)
300 PRINT X
```

After the VAL function is complete, the variable X will contain the numeric value 25.79. The original string A$ is not disturbed, and still contains the original characters. Negative values may be converted by placing a minus sign in the first position of the string as follows:

```
100 LET A$ = " -62"
```

An error will occur if there are any characters in the argument string which are not numerals, a decimal point, or a minus sign in the first position of the string.

LEN Function

This is one of the most useful of the string functions. It is used to determine the "current length" of a string variable. Each time that a string variable is redefined with new characters, its current length changes. Especially when a string is INPUT, the length is quite variable depending upon how many characters are input. The form of the LEN function is:

```
100 LET N = LEN(A$)
```

If the string A$ happened to contain the characters "TYPEWRITER" then the LEN function would return a value of 10 in the variable N. Remember that this will only be the current length of the string and that if it is changed by the program, or by an INPUT statement, then the length will change also.

We have now explored the major bulk of the instructions that make the BASIC language capable of manipulating letters and numbers in powerful ways. The next two chapters deal with the more specialized BASIC statements not found in all versions.

REVIEW QUESTIONS

1. What will this program do? Will it cause an error message?

```
10 READ A,B,C
20 PRINT A,B,C
30 STOP
40 DATA 1,2,3
```

2. Is the following a valid statement? Is it useful?

```
10 IF 0 OR 1 THEN PRINT "TRUE"
```

3. Will this program print the word "TRUE" when you run it?

```
10 A=1:B=0:C=0
20 IF A OR B AND C THEN PRINT"TRUE"
```

4. If you execute the statement

   PRINT EXP(LOG(100))

   what will the computer print?

5. If you execute the statement

   PRINT ASC(CHR$(70))

   what will the computer print? What about this statement?

   PRINT CHR$(ASC("F"))

6. Explain the difference between CHR$ and STR$.

7. Explain the difference between ASC and VAL.

8. Suppose you have the following statement in your program:

   10 A$="SHE SAID TO."

   Add a statement to this program, using LEFT$ and RIGHT$, and the word "NEVER" (but not "SHE" or "SAID" or "TO") which will cause the computer to print

   SHE NEVER SAID TO.

9. Exercise: Write a program, using trig functions, to find the answer to the following problem: How long does a ladder need to be to just reach a window 12 feet above the ground if the bottom of the ladder is placed 6 feet from the wall?

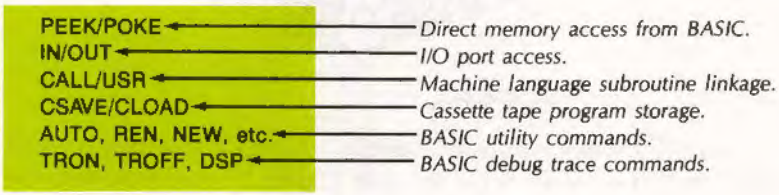
0. Solve the same problem (in 9, above) using only the SQR function.

CHAPTER

5

Variations

Chapters 1 through 4 dealt with those features of the BASIC language most common to the variety of computers on the market today. In this chapter we will explore the more advanced BASIC language features. The keywords that control these features are not always available in all versions of BASIC, but are nonetheless extremely useful in your programs. Chances are that you will find at least 50% of these keywords in any particular version of BASIC. The disk and extended BASICs usually incorporate all these topics and more. The particular keywords and topics we will cover include:



PEEK AND POKE

PEEK and POKE (EXAM and FILL in some BASICS) allow direct control over individual memory locations from BASIC. Recall that a memory location is made up of 8 bits, which is commonly referred to as a "byte." An 8-bit byte can represent a number between 0 and 255. Thus, a single memory byte can hold a number between 0 and 255. Also recall that the computer memory can have up to approximately 64,000 of these memory locations.

In BASIC we can place a number (0-255) in any of these memory locations with POKE. POKE works like this:

```
POKE A,D
```

Valid memory location.

Data to be POKEd into A (0-255).



PEEK

Here A is a positive integer expression, variable, or constant which is equal to the memory location we wish to put the data D into. It varies from 0 to the highest memory location in your computer. D is a number or expression between 0 and 255 that you want to go into location A.

For example:

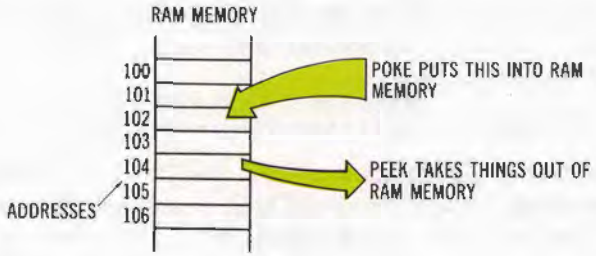
```
POKE 800,127
```

causes memory location 800 to be filled with the value 127. The previous contents of location 800 are lost. And consider this:

```
100 FOR I=800 TO 1000: POKE I,15: NEXT I
```

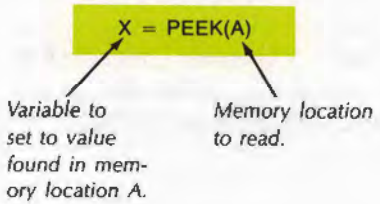
This example causes memory locations 800 to 1000 to be filled with the number 15. We can say that POKE *writes* data into memory.

We can visualize the operation of POKE like this:



The memory is visualized as a huge ladder of memory cells or locations. We can see that each location has a fixed address and can be made to contain any number between 0 and 255.

The keyword PEEK does the complementary function of POKE. PEEK is used to *read* data from memory. PEEK is a *function* and therefore is used like this:



Here the variable X is set to the value found in memory location A by PEEKing at A. Being a function, PEEK *returns* a value to a variable. The argument of PEEK is the memory address A, which may be an expression, variable, or an integer constant. It must not exceed the memory range of your computer.

For example:

```
X=PEEK(4096)
```

causes X to be set to the number found in memory location 4096. And:

```
100 FOR I=0 TO 1023
110 PRINT PEEK (I)
120 NEXT I
```

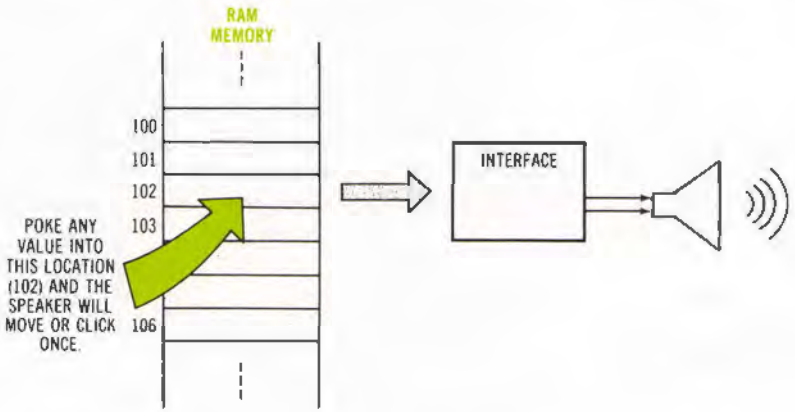
causes the contents of memory locations 0 to 1023 to be printed out on the screen. (We sometimes refer to this as a memory *dump*.) Note that PEEK(I) may appear alone in the PRINT statement just as if it were an expression.

By now you are probably wondering what is so special about direct memory access with PEEK and POKE. There are many uses, but we will cover three of the most common here. They include: controlling external devices, controlling screen information (graphics), and setting option switches in the BASIC operating system.

Controlling the External World With PEEK and POKE

One of the more important uses of PEEK and POKE is in the manipulation of external hardware devices connected to the computer. Devices that may be hooked up to the computer include speakers, joysticks, lights, oscillators, a/d and d/a converters, sensors, and so on. The way PEEK and POKE are used is based on what kind of hardware device is hooked up, and how the designer has implemented the "interface" between the device and the computer. In most cases the external device contains circuitry that decodes a unique fixed address of some memory location in RAM. We can think of the external device as just another memory location. PEEK will allow us to read any information produced by an external device. POKE will allow us to send information to an external device.

For example, consider a speaker connected to the computer memory through an interface that simply pulses the speaker whenever the memory address it is attached to is POKEd from BASIC.



The interface is designed so that a "flip flop" hooked up inside the device, moves the speaker cone whenever the memory location is POKEd into with a value, or when it is PEEKed at. The action of PEEKing or POKEing to the location containing the speaker causes the cone to move regardless of the actual data PEEKed or POKEd. This is because the flip-flop is connected to the address bus of the computer.

Because PEEK is somewhat faster in execution than POKE we can write a simple program that causes the speaker to click several times, and simulate the sound of a ball hitting a wall:

```
100 X = PEEK(102)-PEEK(102)+PEEK(102)-PEEK (102)+PEEK(102)-
        PEEK(102)+PEEK(102)-PEEK(102)+PEEK (102)-PEEK(102)
```

Here we alternately PEEK at location 102 ten times. Address 102 is the location that the speaker flip-flop decodes. The reason we alternately add and subtract the PEEKs is that we end up with X equal to zero.

We can also make tones and melodies with the speaker by adding a little more programing. Suppose we wrote this program:

```
100 FOR I=1 TO 100: POKE 102,0; NEXT I
```

This program will pulse the speaker 100 times by POKEing into location 102. The result will be a medium pitched tone of about 1000 to 5000 hertz. We can see that the program can be expanded so the user could INPUT the length of the FOR...NEXT loop to control how long the tone lasts (duration). Furthermore, we can place another delay inside the main loop to slow down the time *between* pulses. This will have the effect of lowering the tone, like this:

```
100 FOR I=1 TO 100: POKE 102,0: FOR J=1 TO 10: NEXT J:NEXT I
```

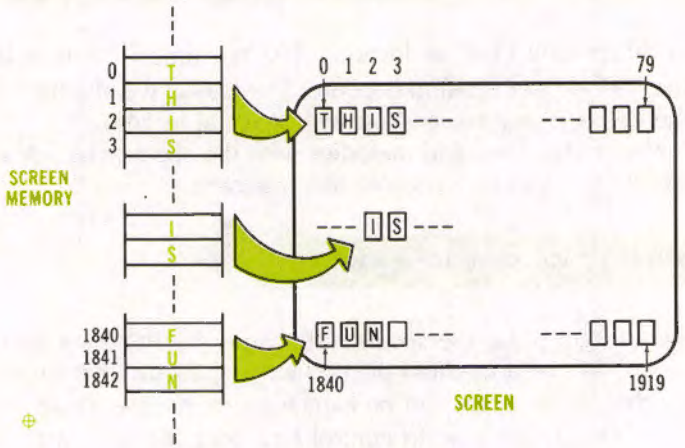
Now the tone lasts longer and has a much lower pitch than before.

Manipulating Screen Data With PEEK and POKE

One particularly useful application of PEEK and POKE involves manipulating character and graphics on the terminal screen without using PRINT statements. This technique is made possible on some computers which represent character positions on the screen in a special chunk of RAM memory called the "screen memory." In this method the computer automatically scans these memory locations, converts the contents to a specific ASCII character, and then displays the characters on the screen in a unique position. Computers that represent screen information in memory are said to be "memory mapped." The memory containing the screen data can be visualized as shown in the next illustration.

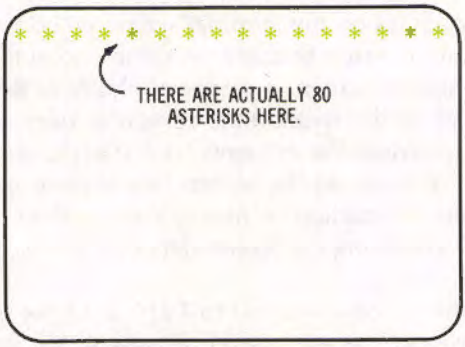
In this example memory locations 0 to 1919 hold the screen information. This is about 2K bytes of memory, which represents the upper limit that can be displayed on a standard video terminal without smearing from loss of bandwidth. The screen is usually organized as 80 columns by 24 rows for a total of 80 × 24 or 1920 characters. Although the screen is organized as rows and columns, the actual memory is organized as one long series chain of bytes. Thus every 80th memory location is the beginning of a new row on the

screen. Keep in mind that the scanning of the screen memory is done automatically and is invisible to the user.



We can easily use POKE to put any character we want in any particular location on the screen from BASIC. For example suppose we want to draw a horizontal line of asterisks (*) on the screen, perhaps for a boundary. Assuming the screen memory begins at 0, the code is:

```
100 FOR I=0 TO 79
110 POKE I,ASC("*")
120 NEXT I
```



Here the index of the FOR...NEXT statement I is used as the address of the screen row. ASC("*") converts the asterisk character to its equivalent ASCII code so it can be stored in screen memory. The FOR...NEXT loop causes

successive * characters to be stored in the first row of screen memory which automatically places the characters on the screen itself.

If we wish to place a specific character on the screen using X-Y rectangular coordinates we can use statements that convert X and Y to values that access the proper memory location corresponding to the X-Y coordinate. Note that the actual screen memory may not start at location 0 as we have shown so far. In this case a base value B must be added to our results to offset the locations so they fall into the screen memory area:

```
198 REM -C IS THE ASCII CODE FOR A CHARACTER
199 REM -B IS THE BASE ADDRESS OF THE SCREEN MEMORY
200 POKE B+(Y*80)+X,C
```

This code POKEs the character code C into memory location specified by X and Y. Y determines the row and X determines the column in the row.

Setting Controls and Options With PEEK and POKE

In many computers PEEK and POKE are used to set various operating modes for BASIC and to obtain the status of certain functions.

For example, in one particular computer, POKE 50,127 turns on the inverse video flag so subsequent characters sent to the screen appear in reverse video (black characters on a white background). POKE 50,255 sets the video back to the normal mode. POKE 33,W1 sets the width of the scrolling window according to the value of W1. POKE−16304,0 sets the computer to operate in the color graphics mode.

Notice that in this last example, the address that the POKE instruction will affect is given as a negative number. This happens because of the fact that internally most computers represent numbers, so that the most significant bit in the number is used to store the arithmetic sign of the number. If it is a positive number, then this bit is a 0. If the number is negative, then the bit is a 1. Since most computers also use 16 bits to store memory addresses, then the highest memory address that can be stored is 65,535 using all 16 bits. If however, the first bit is used to store the sign, then only 15 bits are left for the address itself. This means that the highest address that can be stored in 15 bits in 32,767. If we want to address memory locations that are above this using a signed integer, we must determine what negative number will result in the proper bits being on so as to represent the memory address. This is a somewhat difficult process, since most computers store negative numbers in a form called "twos complement." The point is that it can be done if you really need to PEEK or POKE into memory location above 32,767.

Thus, to access memory 42,000 with POKE you can write:

```
POKE 32767-42000,I
```

In using POKE to control software options, the internal software continually checks the various bytes in memory that have been set by POKE, and uses the value found there to control certain routines while the computer operates. The POKE, in essence, turns a software switch on or off, or provides a certain parameter that is used in the computer's internal processing.

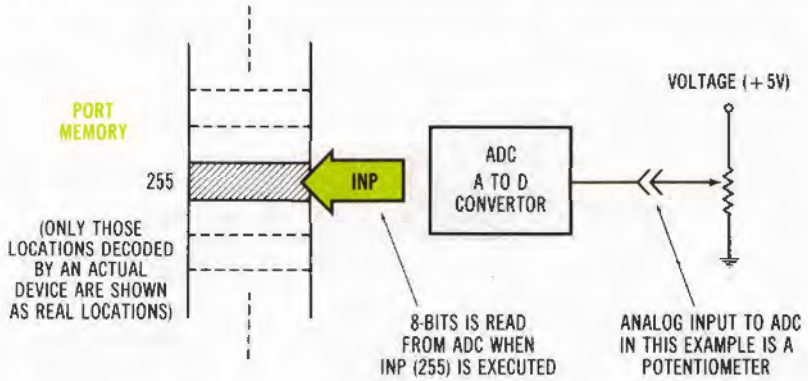
PEEK has similar uses, but is mainly used for checking the status or value of some external or internal event. For example, in a certain computer PEEK(−16384) reads the keyboard and returns the ASCII equivalent of the key pressed. This way no carriage return is needed in a program that accepts keyboard entry (good for programs for children to keep input simple). PEEK(−16336) toggles the external speaker attached to the computer, and X=PEEK (−16286) reads an external push-button switch on the game paddle and if X is greater than 7.27, it means the switch was pressed.

These uses of PEEK and POKE suggest that our programs can communicate even more actively with the intended user, and provide a more exciting and "earthy" computer interaction.

INP, OUT AND WAIT

Certain computers that are built around the Intel 8080 and Zilog Z-80 microprocessors have special input and output locations called I/O ports. The purpose of the I/O ports is to provide a method for communicating 8-bit data from the computer to and from the outside world. Other computers use memory mapped I/O which means any memory location can serve as an I/O port. The port approach is actually a special page of 256 addresses in memory. The chip is designed so that the hardware engineer can decode these special addresses and easily hook up various devices to them such as keyboards, external displays and lights, d/a and a/d converters, speakers, and so on.

In order to access these ports from a BASIC program two special keywords called INP and OUT are provided. INP is a function statement which accepts an integer byte argument (0-255) which specifies the port, and returns an 8-bit value from the port. The value of the 8 bits is determined by what kind of device is connected to the port, and how many bits are being used. As an example of INP, imagine an analog to digital converter (ADC) which is hooked up to port 255:



At the input of the ADC we have attached a potentiometer (pot) to a voltage reference and the wiper is connected to the ADC single input. Varying the pot shaft changes the input voltage to the ADC and the ADC produces a varying digital output in response to this.

If we write the statements:

```
100 PRINT INP(255)
110 GOTO 100
```

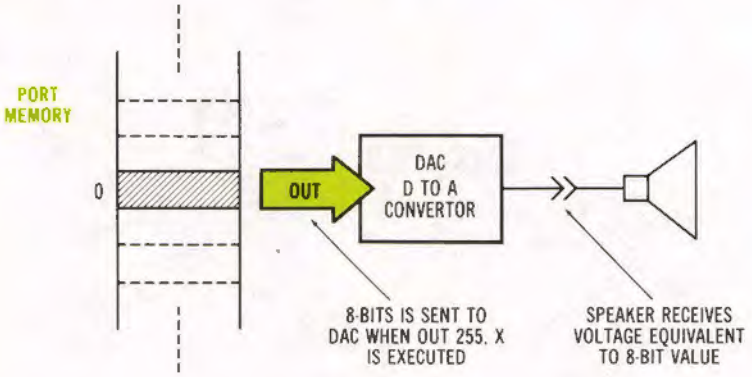
it will cause the computer to read the output of the ADC and PRINT this value on the terminal over and over. As we vary the pot shaft the number displayed on the screen should change in step. We now have a neat way to input position information from a user to the computer program. BASIC could use this result to move an object on the screen, change the rate of some process, etc.

In BASIC, OUT performs the complementary function of INP, that is, OUT is a statement that *sends* an 8-bit value in the range of 0 to 255 to a specified output port in the range 0 to 255. For example, consider a DAC (digital to analog converter) connected to port zero of the computer, with a speaker attached to its output. The DAC takes an 8-bit digital value from the port zero and converts it into an equivalent voltage. Thus, depending on how the value sent to the DAC varies, we can make the speaker vary to make sounds.

In this example we could make a tone by writing the code:

```
100 I=1
110 OUT 255,I
120 I=-1
130 OUT 255,I
140 GOTO 100
```

This will make the computer OUTput alternately a +1 and a −1 to the DAC and will result in a square wave coming from the speaker. Other uses for



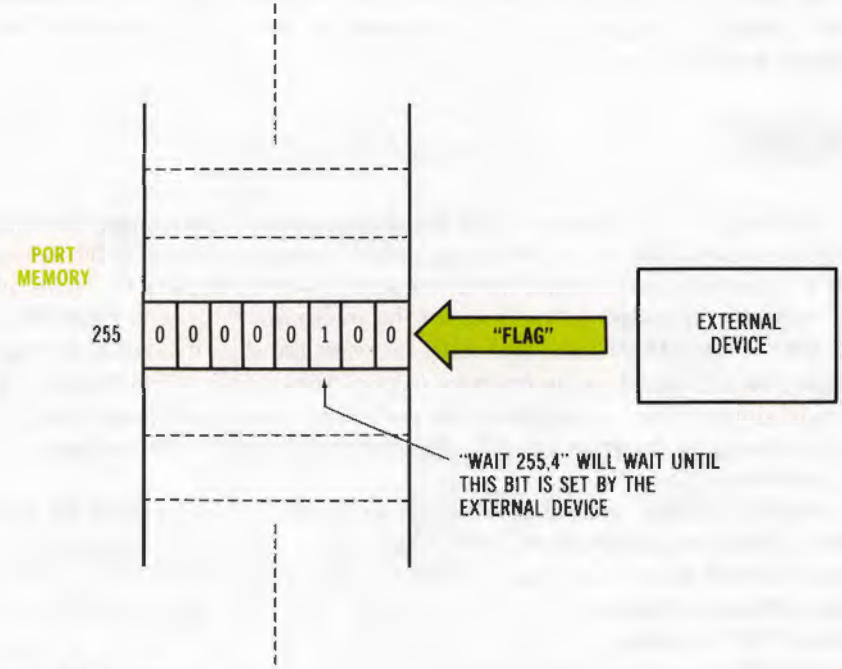
OUT can be devised in a similar manner. For example, a simple pure tone could use the SIN function like this:

```
100 FOR I=0 TO 2*3.14159 STEP. 01
110 R=SIN(I)
120 OUT 255, R
130 NEXT I
140 GOTO 100
```

Some computers use a third keyword, WAIT, in addition to INP and OUT. The purpose of WAIT, as the name implies, is to cause the program to wait until an external device has reached a certain state. This is useful, for example, if a device cannot input information as fast as the computer can read it. Usually in this case the device will set a "flag" when it is ready to have its information read using an INP statement. A flag can consist of a single bit set in a particular input port. Let's assume this flag is the third bit from the right in port 255. The WAIT instruction will then have the form

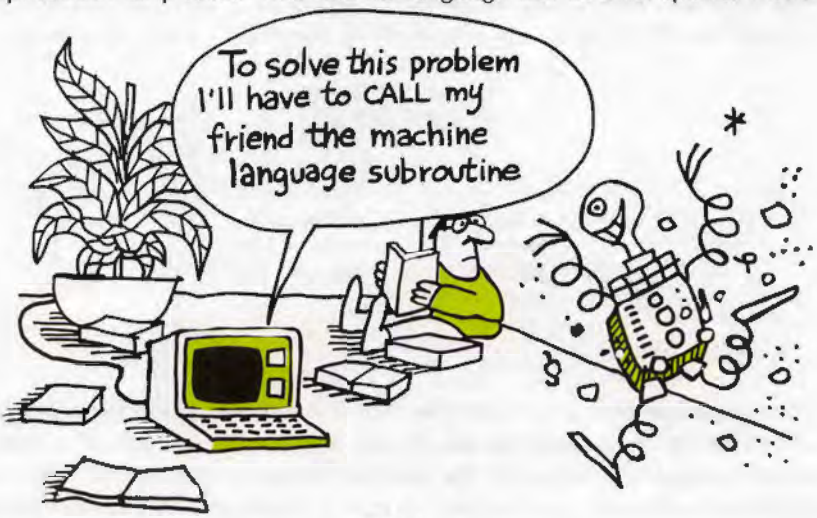
WAIT 255, 4

Here 255 is the number of the port, (just as in an OUT statement), and the 4 is a number which will be logically ORed with the value read in at port 255. In this case, since 4 has a binary value of 00000100, only the third bit from the right will be examined by the WAIT statement. If the result is 0, then the WAIT statement is repeated. It will be repeated until the bit changes to a 1, at which time the program will go on to the statement following the WAIT.



CALL (OR USR)

To understand the CALL (or USR) statement we must first realize that there is a second kind of programming language available to the microcomputer, that is different than BASIC. This second language is the one that operates the microprocessor chip itself. This second language has its own special instruc-



CALL

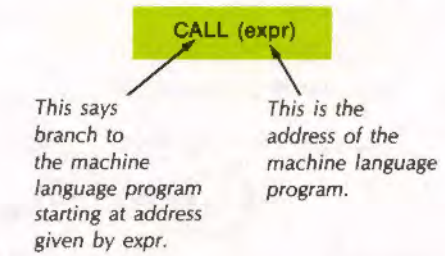
tion set, rules of usage and formatting conventions. The language is referred to as the "machine language" of the computer. It uses special codes and instructions that look like this:

```
LDA F000
```

To the computer this means "load the accumulator register inside the microprocessor with data (8-bits) that is located at memory address F000 (where F000 is a hexadecimal address which equals 61440 in decimal)." There are other machine language instructions for the microprocessor, and these allow 8-bit data to be moved back and forth between registers inside the microprocessor, data to be placed in memory or read from memory into the microprocessor, simple binary arithmetic to be performed, and so on. In essence, the machine language program controls the elementary logic of the microprocessor operations.

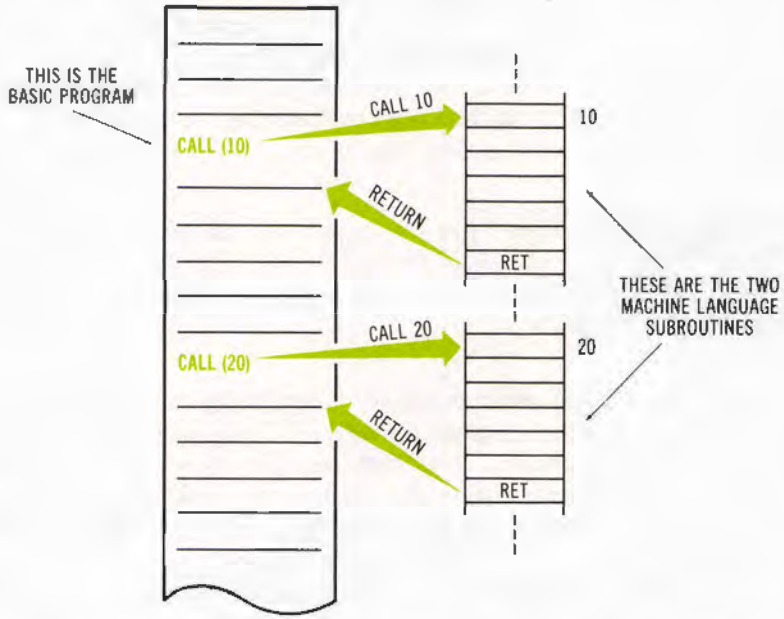
Machine language programs allow us to do things that would be very difficult, if not impossible, to do with BASIC alone. A machine language program resides in an area of memory that is not occupied by BASIC. It has a starting address somewhere in the free memory space not used by BASIC and is treated like a special form of subroutine.

The machine language subroutine has its own return statement which is usually designated by the instruction RET. What CALL does is provide a way for us to branch out of BASIC and execute a specific machine language subroutine. The machine language program does something special until it finds the RET instruction which is usually at the end of the program. RET tells the computer to transfer control back to BASIC and exit from the machine language program. BASIC begins executing again as if nothing happened. The format of the CALL statement is something like the GOSUB statement:



The expression *expr* represents the memory address of the beginning of the user's machine language program. It may also be the address of a machine language program provided in the systems firmware or monitor. This means you can take advantage of the many machine language utility subroutines that are available in the operating system of the computer. As an example of how

to visualize the CALL statement imagine you want to use a machine language subroutine that begins at memory location 10 and then later in your BASIC program you want to use one that begins at location 20. The sequence might look like that shown in the next illustration.



What we have here are two machine language subroutines located in memory, one starting at location 10 and the other starting at location 20. The BASIC program executes one statement at a time until the first CALL is found. Then the CALL 10 statement sends the computer off to execute the machine language program that starts at location 10 in RAM (or ROM in some cases). None of the variables or arrays are changed in the BASIC program, and their values are automatically saved by BASIC when it leaves the main program. The machine language program then operates and performs its specific function (which we will describe soon). When the RET is encountered the control is sent back to the next statement after the CALL in BASIC. The second CALL 20 works in the same manner.

What CALL Can Do

What can we do with the CALL statement and the subroutines it can access? There are literally hundreds of uses for machine language programs and we will outline just a few of the more popular uses here.

Because of its microprocessor controlling capabilities, machine language programs can do things extremely fast. One typical use is in producing fixed time delays from microseconds to seconds, minutes, and hours. You can, for example, write a machine language program that simply goes into a loop for a

specified period of time when it is called from BASIC. The delay could be used for controlling some external event, like how long a process works, making musical tones which have a precise frequency depending on the delay time, and so on.

A simple delay program, written in the machine language of the 8080, might look like this program. Its function is to simply loop for exactly 256 times. Since the microprocessor instructions execute at a precisely defined rate that you can look up on a data sheet, we can precisely know how long the loop takes by multiplying the instruction time by the number of cycles it operates. The code would look like this:

```
0010        MVI A,255
.     LOOP: DEC A
.           JNZ LOOP
.           RET
.
```

This machine language program starts at memory address 16 decimal (0010 hex).

This says load the accumulator register inside the microprocessor with the value 255, decrement the accumulator with DEC by 1, and then jump to the label LOOP if the accumulator is not zero yet. This causes the same series of instructions to execute until the value in the accumulator is decremented down from 255 to 0. When 0 is finally reached, we skip to the next instruction after the JNZ which, as it should be, is a RETurn.

We have looked up on our microprocessor data sheet that the instructions in the main loop (DEC and JNZ) take the total of 7.5 microseconds with a 2 MHz clock frequency. Thus the loop takes 7.5 microseconds × 256 loop cycles which equals 1930 microseconds, or 1.9 milliseconds. The MVI instruction takes 3.5 microseconds and the RET takes 5 microseconds so the total time to execute the loop is 1930 + 3.5 + 5 which equals 1938.5 microseconds. We have to add the time it takes to make the CALL itself which is about 25 to 50 microseconds. Thus the total delay time is fairly close to 2000 microseconds, or 2 milliseconds.

Now if we want to produce a 2 millisecond pause in our BASIC program we could write:

```
100 CALL (16)
```

and the program would stop for about 2 milliseconds. We could increase the delay time easily by putting the CALL (10) statement in a loop. For example consider you want a delay of one second. This would do it:

```
100 FOR I=1 TO 500: CALL (16): NEXT I
```

Now the CALL statement is executed 500 times which gives an overall delay of about 500 × 2 mS, or 1000 ms which is one second. Actually the time will be slightly larger than this because the FOR...NEXT instruction takes up time independently of the machine language program.

To see how the delay program is useful, let us suppose that we have a speaker hooked up to a memory mapped I/O port so when we address the memory location it causes the speaker cone to move once. We can use POKE(−16336), where −16336 is the address of the speaker port, to make the cone move once. We can then put this statement inside the loop we wrote and the CALL will control how long the delay is between successive POKEs at the speaker, or the overall frequency:

```
100 FOR I=1 TO 500: POKE −16336,0: CALL (16): NEXT I
```

This should begin to give you an idea of what we can do with the CALL statement. However, you can use CALL and never write a bit of machine language program. What can be done is this. We can use CALL to access a machine language program that is provided within the system monitor. The manufacturer will usually document the address of various subroutines of the utility type so that you can CALL them from your BASIC program. These utility routines consist of programs that home the cursor and clear the screen, clear the cursor from end of page or end of line, produce a line feed, scroll up or down one line of text, and so on. What actually happens when the call is made is that the address in the CALL specifies a utility machine language subroutine that performs the desired function without your doing anything. Exit is like in the normal CALL. For example, to clear the screen normally we would have to issue a succession of PRINT statements that caused enough line feeds to move any text off the screen, like this for a 24 line screen:

```
100 FOR I=1 TO 24: PRINT:NEXT I
```

Look how much simpler this is with a CALL:

```
100 CALL (−936)
```

Not only does this clear the screen, it also homes the cursor to the top left corner of the screen. The previous statements left the cursor at the bottom left corner.

Not all versions of BASIC use the CALL statement to access machine language programs. Although the operation of CALL is certainly easy to understand, it is somewhat inflexible when you wish to "pass" variables back and

forth between the machine language program and the BASIC program. Passing variables is done when you wish to do things to a value that BASIC can't do; for example, you might want to pass the time of day to a machine language program that then puts the data into a hardware clock in the computer. A BASIC statement more suited to passing data is called USR (pronounced user). USR performs differently than the CALL statement. The format of USR is:

```
y = USRn(x)
```

Here "n" is a digit in the range of 0 to 9. This corresponds to a similar digit supplied by another statement called DEF USR. (In some BASICs this is called USERLOC. We'll use DEF USR for now.) The purpose of DEF USR is to supply the starting address for a particular USR. More on DEF USR in a second. The "x" in USR is called the "argument." It can be an expression or a numeric or string variable. Its value gets "sent" or passed to the machine language program that you call. How it gets passed and where it ends up is a function of the particular BASIC you are using and the microprocessor it is implemented for. Consult your manual for this. The value "returned" by the USR function ends up in the variable to the left of the equal sign, in this case the "y." Usually it is the same "type" (string or numeric) as the argument passed by USR.

The format of DEF USR is:

```
DEF USRn = startaddr
```

where "n" is a digit in the range of 0 to 9 that corresponds to the digit "n" used in USR. This means there can be up to 10 individual USR statements in a single program. The value "startaddr" is the starting address of the machine language program USRn. Thus, DEF USR tells "where" to branch to when the USR instruction is encountered in the program. Usually startaddr is an integer in the range of 0 to 65535.

An example of USR would be:

```
100 DEF USR3=24000
    .
    .
    .
500 M=USR3(Z+2)
```

Here statement 100 says that machine language routine number 3 starts at address 24000 decimal. Then line 500 "calls" this routine, sending the value of Z+2 to the routine. The routine would, of course, have been set up previously by the program, either being POKEd in with DATA statements or BLOADed off the disk into memory. The routine then executes and does something with the value of Z+2. The resultant value is sent back to the BASIC program and put in variable M in line 500.

The proper use of USR requires a firm understanding of machine language programming, so you should use it with caution. You can, of course, use the USR function without passing variables. In this case the variable to the left of the equal sign would be a "dummy," as would the variable in the argument. Dummy means it's not used by the program but still must appear in the statement for it to work correctly.

CSAVE AND CLOAD

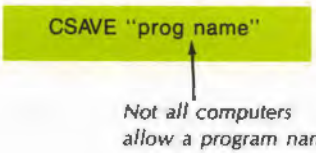
One of the problems with small computers of today (and for that matter all computers that use volatile random-access storage) is that when you turn the power off, your program stored in random-access memory is lost. This is because RAM memory requires electricity to maintain its state—power must constantly be applied to the memory circuits to "hold" up its state. What is really needed is some form of permanent storage media, one that would allow us to put a long BASIC program on it and then take it off at any later time. Historically, the first form of permanent computer storage was paper tapes. These were long delicate ribbons of paper which contained holes punched in a specific order that the computer could understand and convert back to a



SAVE

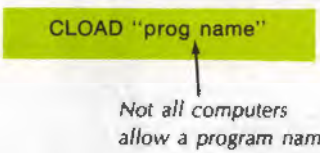
program. Today most manufacturers use magnetic media for storing computer programs. There are two popular devices that use magnetic storage—cassette tape recorders and floppy disks. The current state of the floppy disk technology is that it is a rather significant expense in the overall computer system. A typical floppy disk drive costs about $500 to $1200 depending on its size and speed, while a "garden variety" cassette tape recorder costs under $40. And cassette tape is also inexpensive and also easy to find. So naturally most small computers for the home come with cassette tape recorders either built in or the computer has connectors for hooking up an external recorder you provide.

In order to use the cassette storage functions, BASIC provides two simple commands: CSAVE and CLOAD. CSAVE works like this:



The CSAVE command causes the current program in memory to be stored, or written, on the cassette tape. In some computers the program can be assigned a special name so that it can later be read into the computer memory. A name also allows us to store several programs on a single tape and then have the computer retrieve only the one we specify.

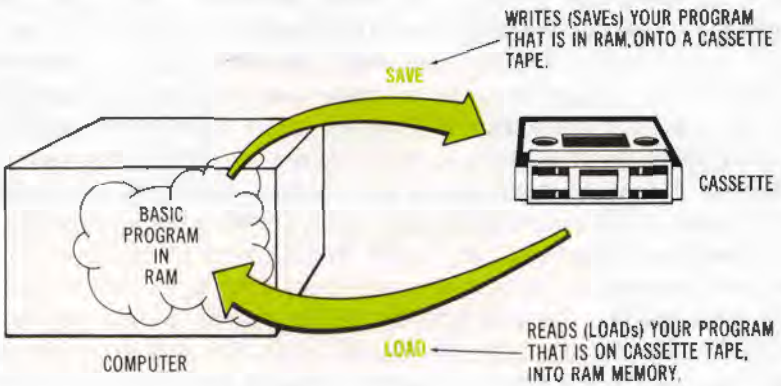
The complementary function of CSAVE is the CLOAD command which looks like this:



The CLOAD command causes the program called "prog name" to be retrieved from the cassette tape and loaded, or read, into the computer's RAM.

In both cases the user will have to start the tape recorder with the levers and switches in the right state before the command is implemented. Some computers will automatically search for a certain program you have named; others require that you locate the program on the tape and then start the recorder in the proper mode. If you have to locate the program yourself, then it means that you will have trouble storing programs that are long and require using different subprograms. In other words, it is difficult to make programs that have different parts which easily merge together if you do not have the "program name" option available on the recorder commands.

We can visualize the CSAVE and CLOAD commands like this:



The way CSAVE and CLOAD are used in the development of a BASIC program is like this. First you write your BASIC program and then enter it into the computer from the keyboard. You check that it operates properly and get all the bugs out. Now with the program you want to save in memory, you put a fresh tape in the cassette recorder, then start it in the record mode. At the computer you type the word CSAVE, followed by an optional program name you make up. Now when you press the carriage return key on the computer the circuits will detect what you want done, and begin to convert the program in memory into a series of tiny electrical impulses and send them to the cassette recorder.

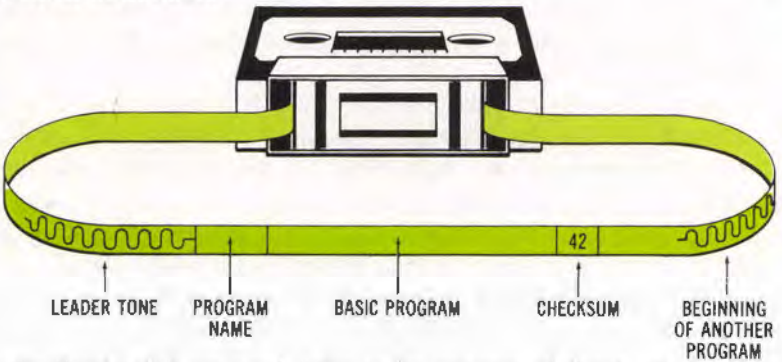
The electrical impulses are like the holes punched in a paper tape; they carry information in the way they are ordered on the tape. But instead of being large holes, the impulses create extremely tiny magnetic areas on the cassette to align in certain patterns; therefore, we can store very large programs on very small amounts of tape. In fact, for a reference, typically an 8K byte BASIC program can fit on less than 10 feet of cassette tape.

Because there are thousands and thousands of impulses to store in a typical program, it is easy for a program to miss a bit of code, or pick up noise on the tape itself, which all serves to make the copy of the program wrong. If we try to read it back into the computer it may not run if it has bad information in it. The way around this is that most manufacturers include an error-checking method in the circuits that CSAVE and CLOAD data. The error checking works by adding up in a special manner all the bits that are stored so that when the program is finished, a unique identifying number results. Then the computer takes this number and puts it on the tape at the end of the program area. Now when the program is to be read back into memory with CLOAD, the computer takes all the incoming impulses and adds them up again. It then compares the new sum with the one that was stored on the tape. If they agree then there were no errors when we read or stored the program. If the check sum is different, then an error occurred either when we loaded the program or when

it was originally saved. If the error occurred when we originally saved the program, we must save it over again. If the error was a read error, that is something that went wrong when we CLOADed the program, then we must try to CLOAD it again. Usually, you must carefully adjust the volume and tone controls on the tape recorder to make sure the tape is properly read. When you read a program back into memory from tape, it destroys the program currently in memory, so it is important that the program is saved right the first time. One way people usually get around this problem is to make two or three copies of a program in one shot. Since CSAVE is a command that can also be used as a statement, you could type CSAVE:CSAVE:CSAVE and the computer would record the program in memory three times on the tape. Another thing found on a cassette tape circuit is the ability to write a "header" that contains the program name for identification. The computer can search through the tape until it finds the correct program name and then read the program itself into memory.

Some machines place a one or two second "leader" tone on the tape before the actual program data. This leader tone is used to get the circuits in the computer synchronized with the information on the tape itself. The leader tone guarantees that the computer will lock onto the incoming data stream when it occurs.

A drawing of how a BASIC program appears on the cassette tape is shown in the next illustration.



In the floppy disk we don't have to be concerned about not getting a good copy of our program. This is because most floppies have such good circuits and low error rates. Also, the floppy will store a program very fast and load it back into RAM just as fast. Typical load time for a 8K byte BASIC program from floppy disk is under one second. The commands SAVE and LOAD are used with floppy disk instead of CSAVE and CLOAD.

BASIC COMMAND AIDS

Perhaps this section should just be called "making life easier on the computer." Given the fact that there is substantial power resting under our finger tips in the way of the internal processing ability of the computer, manufacturers of

BASIC have developed several commands for helping in the program development stage. These commands vary from computer to computer, but for the most part they are much alike. Of all of the commands, only NEW is really necessary, so let us begin with it.

NEW or SCRATCH

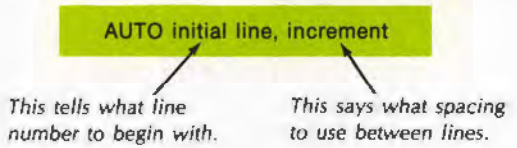
The purpose of the NEW or SCRATCH command is to completely remove (erase) the current BASIC statements in memory. Recall that you can have only one program in memory at a time (not counting any machine language programs). Therefore, there will be times when you need to erase the old program and enter a new program. NEW, when typed in at the command level, causes every statement and line in the computer to be removed or scratched. Some versions of BASIC call the NEW command SCRATCH or just SCR.

Usually you use NEW after you have made a copy of the BASIC program you are working on (remember CSAVE). Once you have a copy CSAVEd you can use NEW to enter and develop a new program.

NEW is also used when you are just trying some simple statements out, perhaps for practice or teaching.

AUTO

Line numbers may be generated automatically by the AUTO command. What AUTO does is provide automatic insertion of line numbers when you are entering program lines into the computer. AUTO saves you having to type a line number manually. It is useful when you have a long program to enter and already know what the line numbers are to be. The format of AUTO is:



As an example, suppose you want line numbering to begin at 100 and have an increment of 10:

```
AUTO 100,10
100 INPUT X,Y
110 PRINT SQR(X^2 + Y^2)
120  C
OK
```

AUTO will continue to number automatically until a Control/C is typed, which will return BASIC to the command level. If the *initial line* is omitted, it

is assumed to be 10; and if the *increment* is omitted, it is also assumed to be 10. For example:

```
AUTO
10 PRINT "THIS IS A TEST"
20^C
OK
```

What if AUTO generates a line that already exists in a program currently in memory? Usually the computer will print an asterisk, warning the user that any input will replace the existing line.

CLEAR

CLEAR is the BASIC command that sets all program variables to zero. You would use CLEAR when you are developing a program. After the program is test run, the variables assume certain values that are not what they were when the program started. Now if you want to return the program and have all the variables reset to zero, you will have to type CLEAR, before you type RUN. On some computers typing RUN also clears all variables to zero. Of course, it is good programming practice to make all variables start out at their initial values inside the program itself. You can do this before you execute any program statements that use the variables.

For example, if you see statements like this at the beginning of a program you can be almost sure that variables are set by the program:

```
10 LET A=0: LET B1=0: LET Z=0
```

The command CLEAR *expr* is used to set the string space to the value specified by the expression *expr*. The string space is the amount of memory set aside by BASIC to hold the total length of any and all string variables and constants in the program. Usually string space defaults from 50 to 100 bytes when BASIC is first used. You should not confuse setting aside string space with DIMensioning a string array. DIM simply allocates space for string arrays. CLEAR *expr* sets aside string space for undimensioned strings, strings that do not use arrays.

FRE(0)

If you type the command FRE(0) (on some computers the command MEM does the same thing) BASIC will print out how much memory space remains unused by the program currently in memory. This is useful if you want to know how much memory your computer has before you start to write your program, and will also let you figure out how large your program is.

For instance, if you have a program named "BIGPROG" on a cassette tape, the following sequence of commands will tell you how long the program is:

```
NEW
PRINT FRE(0)
14510
CLOAD"BIGPROG"
OK
PRINT FRE(0)
1105
```

Now you know that BIGPROG must be 13405 bytes long, because with no program in memory there are 14510 bytes of free space, and after you load BIGPROG there are only 1105 bytes left. The difference, 14510 less 1105, is the size of the program. Actually, a program will use some more bytes when you RUN it, for storing variables, so that just because a program fits in memory when you first load it, doesn't necessarily mean that it will fit when you start to run it. Also, if you reserve space for strings, this will be subtracted from the available memory.

CONT

CONT is a BASIC command that means CONTinue. Its purpose is to continue program execution after a Control/C has been typed or a STOP or END statement was executed. When CONT is typed execution resumes at the statement where the break occurred.

CONT is useful in debugging, especially where an "infinite loop" is suspected. An infinite loop is a series of statements from which there is no escape. Typing Control/C causes a break in execution and puts BASIC in the command level. Direct mode statements (such as PRINT) can then be used to print out intermediate values. Execution can then be restarted by typing the CONT command, or by executing a direct mode GOTO, which causes execution to resume at the specified line number. However, if you change the program or the variables and try to CONT the program you will get a CANT error.

Some BASICs call the CONT command just CON. The manual with the BASIC language will explain the proper way.

DELETE

DELETE is for eliminating specific lines from your program. We have learned already that you can eliminate any line in a program by simply typing the line number and following it with a carriage return. DELETE allows us to eliminate several successive lines at one time. The most general form of DELETE is:

```
DELETE beginning line number-ending line number
```

This form of DELETE will make the computer eliminate all lines from the specified beginning *line number* to the specified *line number*. You can also type:

```
DELETE-line number
```

This will delete all lines of the current program, from the beginning line of the program to the specified *line number*.

In some computers the DELETE command is just DEL. Whatever the case, be careful when using DELETE so that you do not accidentally delete your entire program.

EDIT

The EDIT command allows editing of a specified line without affecting other lines. The EDIT command usually has a powerful set of subcommands that allows the deleting of only specified items of text in a line, or the insertion of specified text, or the replacement of specified text with other text.

On some computers, instead of EDIT cursor control editing is provided. Instead of using special subcommands, cursor editing allows you to position the cursor with keyboard keys over text you wish to alter and directly type over the old text. You can then copy the old text by moving the cursor through the rest of the statement line. Cursor editing is easier to use, but is not as powerful as command editing.

RENUM or REN

The RENUM command allows all your program lines to be "spread out" so that a new line or lines may be inserted between existing lines. We use RENUM when a program has lines in it which are too close together, or simply do not follow any order. The format of RENUM is:

```
RENUM start,initial line, increment
```

Here *start* is the new number of the first line to be resequenced. If *start* is omitted it is assumed to be 10. *Initial* line is the line number at which resequencing will begin. Lines less than the *initial line* will not be renumbered. If the *initial line* is omitted, the whole program will be resequenced. Increment is the value of the increment between lines to be resequenced. If increment is omitted, it is assumed to be 10. Some examples of RENUM are:

RENUM Renumbers the whole program to start at line 10, with an increment of 10 between line numbers.

RENUM 100,100 Renumbers the whole program to start at line 100 with an increment of 100.

RENUM 6000,5000,1000 Renumbers the lines from 5000 up so they start at 6000 with an increment of 1000.

With RENUM, all line numbers appearing after a GOTO, GOSUB, ON . . . GOTO, ON . . . GOSUB will be properly changed to reference the new line number.

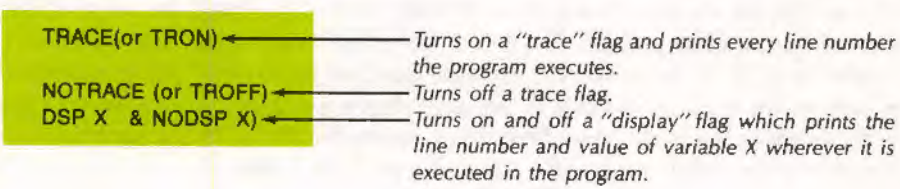
Note that RENUM cannot be used to change the order of program lines (for example, RENUM 15,3 when the program has three lines numbered 10, 20, and 30), nor to create line numbers greater than the maximum permitted in your version of BASIC (65529).

Also, if a line number appears after one of the statements above but does not exist in a program, an UNDEFINED LINE XXXX IN YYYY will usually be printed.

Debug Commands

Whenever you write a program you will undoubtedly discover that the program does not work exactly as you planned. This may be discovered when the program runs and produces an error message instead of what you expected. Most of the time the error messages are sufficiently informative to let you find the error and correct it. For example, the message *** REDIMENSIONED ARRAY is fairly self explanatory and means you have attempted to redimension an array that was previously dimensioned. So all you have to do is find the second DIM statement in the program and eliminate it (unless, of course, it was purposely redimensioned; in which case you have to figure out another way to accomplish the goal). But often we get errors that are not so easy to find and fix. Things like a program that keeps branching to the wrong location and produces a secondary error because of this. Or errors due to using several levels of nested loops and having the wrong index on the associated NEXT statements.

To solve these kinds of problems BASIC provides several special commands called the "debugging" commands. The number and type of the debugging commands varies with different computers, but there are several commands that are fairly typical and appear on many of the home computers, including:



Trace and Notrace (TRON and TROFF)

TRACE (called TRON in some BASICs) is a command that places the computer in the tracing mode. What TRACE does, in effect, is cause the computer to print every line number that is encountered as the program executes. Because the line numbers appear along with all the normal information the program displays, brackets around the line numbers tell us that they are line numbers. So what TRACE does is follow the program flow as it executes, showing you how the branches are made and to where the program is being directed. In essence, the trace is like a running road map of the line numbers the program is following as it cycles along. You can use the line numbers to check that the program is following the course you expect.

For example, suppose you have this program and find that it is hanging up somewhere and not completing its function. What we do is type TRACE which places the computer in the trace mode and then type RUN to start the program. Here is what happens:

```
TRACE
RUN
{1} {5} {10} {50} {60} {70} {100} {110} {120} {100} {110} {120} {100} {110}
{120} {100} {110} {120} {100} {110} {120} {100} {110} {120} {100} {110}
{120} {100} {110} {120} {100} {110} {120} {100} {110} {120} {100} {110}
{120} . . . . . .
```

What we can tell from this display of trace is that the lines 100, 110, 120 keep repeating over and over. We only showed a portion of the TRACE, but you would eventually see the screen full of the numbers 100, 110, and 120. This obviously points to the fact that the program is caught in an "infinite loop," repeating the same code over and over. TRACE has pointed to the lines at fault, so we can use LIST to examine the code like this:

```
LIST 100-120
100 I=I+2
110 IF I=9 THEN RETURN
120 GOTO 100
```

Take a close look at the program. The variable I is incremented by 2, tested for being equal to 9, and if it is RETURN. If it is not equal to 9, it loops back to line 100 and increments again. That seems fine, but what if the value of the index I was an even number like 0 or 2 when the loop started. The values of I would be 0, 2, 4, 6, 8, 10, 12, 14, etc. Because the index I never equalled exactly 9, the loop repeated way beyond what we wanted. All we have to do to fix things is to change the comparison inside the IF . . . THEN statement so it causes a branch if we *exceed* the value of 9, like this:

```
110 IF I = >9 THEN RETURN
```

Now we have solved a program "bug" with the aid of the TRACE mode. Since we are done and do not want the line numbers caused by TRACE anymore, we use an associated command called NOTRACE (or TROFF in some BASICs).

NOTRACE simply turns off the TRACE flag and stops the generation of line numbers to appear on the screen.

There are other similar uses for TRACE, but all fall into the area of tracing the path of the program execution as the program runs.

DSP

The DSP command, pronounced DSPlay, is used to print out the actual value of a variable and the line number where it is executed. The format of DSP is:

```
DSP var
```

where *var* is a variable which appears in the program and which is to be displayed. What variable does is cause the line number and the value of *var* to be printed out each time the computer finds it in its execution path. What this does is allow us to see how the values of some variables are changing without using a separate PRINT statement for it *everywhere* it appears in the program.

For example, in our previous "infinite loop" program section we could have typed, DSP I and run the program to cause this effect:

```
DSP I
RUN
100 I=2
100 I=4
100 I=6
100 I=8
100 I=10
100 I=12
 .
 .
 .
 .
```

Here we see how DSP works. The value of I and what it equalled each time it was incremented was printed out in a long column. Note the printing of I

would continue until we got an overflow error or stopped the computer with a Control/C.

Now it is obvious that the variable I never equalled the value of 9, and we can fix the program even faster.

We often use DSP as a statement rather than a command. For example:

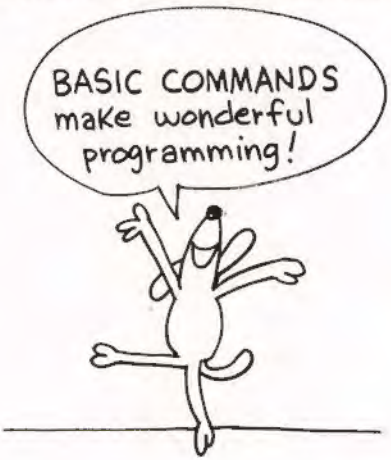
```
999 DSP X
```

when line 999 is executed the DSPLAY flag will be turned on and all subsequent lines where the variable X appears and actually changes the value of X or redefines it, will be printed out.

When we want to turn the display flag off we use NODSP *var*, where *var* is the same variable that appeared in a previous DSP command or statement. There is no limit to the number of DSP commands or statements you have in effect at the same time. Since the variable is actually printed out with the value, we can tell commands from statements. In the case of several DSP commands the variables following them will be printed in the order in which they are executed. Thus you will get a mix of variables in the display and will have to visually sort them out. Sometimes we forget where the DSP statement appears inside a program, and this makes turning the DSP flag off rather difficult. That is why it is better to keep the DSP as a command rather than a statement. Note also that RUN clears the DSP mode, so to restart a program without turning DSP *var* off, use a GOTO or CONT command in the immediate mode to start the program up again.

VIDEO PLOTTING WITH TAB

We can do some pretty neat graphing with TAB and equations. For example suppose we wanted to print out a simple equation such a Y=X. Consider that



the X axis is down the screen (0 is at top) and the Y axis is across the screen (0 at the left). Now consider what happens with this program:

```
10 FOR X=1 TO 15: Y=X
20 PRINT TAB(Y) "*"
30 NEXT X
RUN
 *
  *
   *
    *
     *
      *
       *
        *
         *
          *
           *
            *
             *
```

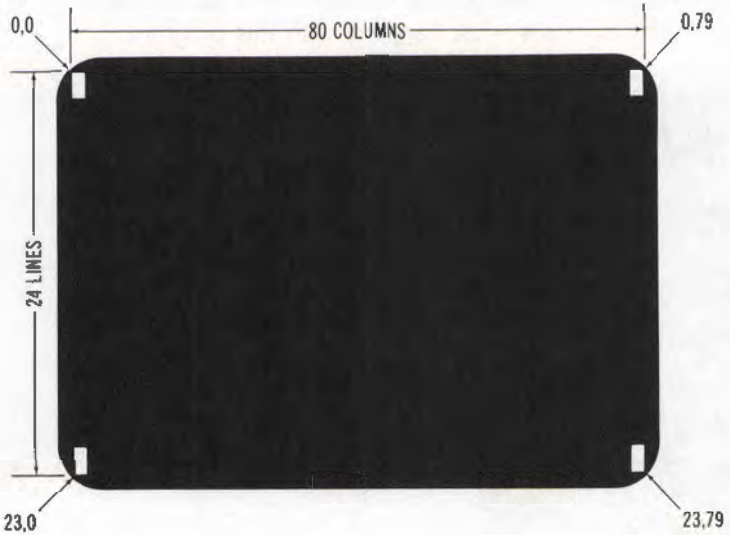
Here we have a FOR . . . NEXT loop which increments through 15 values of X. The variable Y is set to equal the value of X and then we TAB Y spaces to the right and print an asterisk character. Since the equation is Y=X we get a slanted line. Although the slant isn't 45 degrees as would be expected, we still get a straight line. The reason the slant isn't 45 degrees is because the vertical spacing on the terminal doesn't equal the horizontal spacing. Usually there are two to three times as many horizontal column positions as there are vertical positions. The typical terminal or crt screen is 80 columns by 24 lines, as shown in the illustration at the top of page 190.

Frequently we see formats of 40 by 40, 64 by 24, and so on. The point is that the spacing is not equal and there will be some distortion in your graphs.

In the example Y=X we have a "linear" equation. Linear equations produce straight lines of different angles and different offsets on the screen. But, we also can print curved-line graphs. Consider the equation Y=X*X/10. Can you tell how the RUN of this program would look?

```
10 FOR X=1 TO 10
20 Y=X*X/10
30 PRINT TAB(Y) "*"
40 NEXT X
```

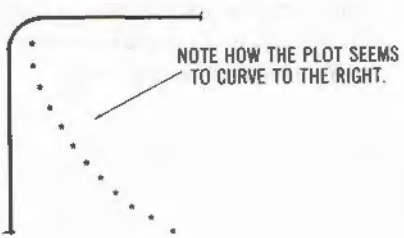
What happens is that the equation Y=X*X/10 generates decimal fractions between 0 and 10. When X is 1, X*X/10 is 0.1 and because we can only tab



an integer number of spaces, TAB(Y) is zero. A table of the values of X, X*X/10, and TAB(Y) are as follows:

X	Y (X*X/10)	TAB (Y)
1	0.1	0
2	0.4	0
3	0.9	0
4	1.6	1
5	2.5	2
6	3.6	3
7	4.9	4
8	6.4	6
9	8.1	8
10	10.0	10

A RUN of the program produces this graph:



What Is Your SIN?

We can take advantage of some of the more powerful BASIC numeric functions in our graphing. For example the trigonometric functions SIN, COS, and

ATN may be easily used to create harmonic curves for biorythms, music analysis, modulation studies, and so on. Let us consider the SIN function that we learned earlier produces the trigonometric sine of the angle specified as its argument. So we could use the formula Y=SIN(X) to calculate the value of the Y coordinate we want to TAB out to. Since X is expected to be in radians rather than degrees for the trig functions, we will make X vary from 0 to 6.28 which is one full circle in radians. We will use a FOR . . . NEXT loop with a variable STEP size to control the number of points of the curve that are displayed.

To make things even more interesting we will make the string that gets PRINTed out a message. Remember this message can be any characters on the keyboard you wish. Here is the program:

```
 0 REM - SIN CURVE
10 INC=.3
20 FOR X=0 to 6.28 STEP INC
30 Y=SIN(X)
40 Y=Y*20+20
50 PRINT TAB(Y) "MITCH"
60 NEXT X
```

Note that we changed Y in statement 40 to Y*20+20. This is because Y=SIN(X) produces decimal fractions between −1 and +1 and we cannot TAB a negative number of spaces. So we simply multiply Y by 20 so the numbers are −20 to +20 and then add 20 so the numbers are offset to 0 to 40. Since the screen width varies with different BASICs we could increase the width to 80 by writing Y=40*Y+40.

Here is a RUN of the SIN CURVE program.

```
                    MITCH
                      MITCH
                        MITCH
                          MITCH
                            MITCH
                              MITCH
                              MITCH
                            MITCH
                          MITCH
                        MITCH
                      MITCH
                    MITCH
                  MITCH
                MITCH
MITCH
MITCH
  MITCH
    MITCH
      MITCH
        MITCH
```

USING A LINEPRINTER

If you have a lineprinter connected to your computer you can use it in much the same way you use your video screen—to print the output of your programs, and to list them. The printer has the advantage of creating a permanent record, but it is much slower than the video screen.

The statements you use to output information to the printer are very similar to those used with the video screen. These statements are LLIST, which is similar to LIST, LPRINT, which is similar to PRINT, and LPRINT USING, which is similar to PRINT USING, which will be discussed in Chapter VI.

LLIST

This statement, usually used in the command mode, will cause your program to be listed on the lineprinter. If you try to execute this statement when you don't have a lineprinter, or when the printer isn't ready to receive data (perhaps because it's not turned on), you may get an error message, or your computer may "hang up," that is, go into an endless loop until you reset it. All the variations in the ways to use LIST apply also to LLIST. For example, the following statements are all valid:

```
LLIST 10          list on the printer only line 10
LLIST 50-100      list on the printer lines 50 to 100
LLIST 1000-       list on the printer all the lines following 1000
LLIST -500        list on the printer all the lines up to 500
```

With certain combinations of computers and printers a problem can arise when a program line to be listed is longer than the number of characters the printer can print on one line. Sometimes the printer will come to the end of the line and keep typing in the same place, without going on to the next line. Also, sometimes the printer will go back to the beginning of the same line and type over it, again and again. These and similar problems can be caused by "misunderstandings" between the computer and the printer. (For instance, the computer may assume that the printer will always execute a line feed every time it sends a carriage return, while the printer assumes that if the computer wants a carriage return it will send one.) Sometimes the problems can be solved by resetting switches in the printer; other times the "driver" routine in the BASIC interpreter, which tells the printer what to do, may have to be modified.

LPRINT

LPRINT is used in exactly the same way that PRINT is, and you can do all the things with LPRINT that you have already learned with PRINT. LPRINT will send your output to the printer in just the same way PRINT sends it to the screen.

For example, the following statements are all valid:

```
LPRINT 2+2
LPRINT A$, B, C$, D
LPRINT TAB(10); B$
LPRINT "ENTER YOUR NAME";
```

One of the main differences between LPRINT and PRINT is that the number of characters you can put on a single line is sometimes longer on a printer than it is on the video screen. You will have to keep this in mind if you want your output to the lineprinter to be formatted in a particular way. Another difference is that on many printers a line will not be printed until it is full (that is, if you have an 80 column printer, nothing will be printed until 80 characters have been sent), or until a linefeed has been sent to the printer. This means that if you execute a statement like

```
LPRINT"HELLO, THERE.";
```

it will not be printed right away. To make the printer print it, you will have to send enough text to fill up the line on the printer, or else execute another LPRINT statement (without a semicolon) to cause a carriage return.

REVIEW QUESTIONS

1. Write a program, using PEEK, which will print a "map" of a 512-byte section of memory: that is, for each byte of memory it will print an "X" if the byte is occupied (not zero) and a "–" if it is empty (equal to zero).

2. If you use INP to read data into an input port from a device that tells you whether a number of lights are turned on or off in your house, how many lights can you find out about with one INP instruction?

3. What is the difference between "CALL 1000" and "GOSUB 1000?" Does the 1000 mean the same thing in both cases?

4. Does the word "LOAD" in "CLOAD" mean that you are loading a program or data? Does it mean you are loading into memory or onto tape?

5. What will happen if you are running the following program:

```
10 PRINT"CAT",
20 GOTO 10
```

and you then break into the program with "control/C" or "BREAK", and then you type

```
10 PRINT"DOG",
CONT
```


ADVANCED
BASIC

CHAPTER

6

Advanced BASIC

When you buy something—whether it's an automobile, a stereo system, or a condominium—you can either make do with a basic no-frills model, or you can add extra features to make it a little more luxurious or easier to use. The same is true of programming languages.

With the BASIC statements you have learned so far in this book you can write programs to do almost anything you want. But you may have found yourself, every so often, in situations where you said, "This is a little awkward—if only there were a way to do it more easily."

This chapter will cover some of these "luxury" features: BASIC statements that will help you write simpler, clearer programs, more easily, with more elegantly presented output. Not every version of BASIC has all of these features, but if yours does, you will enjoy learning to use them, and after a time you will probably find them indispensable.

Of course, as with other luxuries, there is a price to pay for the added convenience. One possible problem is the "transportability" of programs from one machine to another. Suppose you write a program on a machine with some of the advanced BASIC statements we are about to describe, and you use these statements in your program. Later you might want to give this program to a friend. What happens if your friend's computer is a simpler model, which doesn't understand the advanced statements? The program wouldn't run. In this case, we say the program is not "transportable" from your machine to his. In most cases, of course, you are writing for a specific machine, so this is not a problem.

These luxury features might also cause a problem in your pocketbook. In most personal computers the instructions which tell the computer how to decode BASIC statements are kept in a part of the machine called ROM, for Read Only Memory. The more BASIC statements the machine must understand, the bigger the ROM has to be. And the more ROM there is, the more the computer tends to cost. Fortunately, the cost of computer memories is coming down—and that makes the "luxury" statements we are about to describe one of the best deals around.

In this chapter we'll be talking about six different categories of BASIC statements.

Under PROGRAM CONTROL we'll cover WHILE...WEND, an elegant way to do loops; ELSE, an extension of IF...THEN conditional statements; and DEF FN, which lets you define your own functions.

The INPUT/OUTPUT section will include LINE INPUT, which gives you a more flexible way to handle keyboard input; PRINT USING, an amazingly powerful way to format output; INKEY$, which takes you into the heart of the keyboard input process; and several other functions.

In ADVANCED STRING FUNCTIONS we'll cover INSTR, which lets you search for a word in a sentence; MID$, which will alter the middle of a string; STRING$, which will create a long string with a very short statement; and SPC(n) and SPACE$ which create strings of spaces.

Under CREATIVE ERROR HANDLING we'll talk about ON ERROR GOTO, RESUME, ERROR, ERR, and ERL, which all work together to make it possible for you to actually recover from your errors, and even make some errors of your own, on purpose!

In DEFINING AND CONVERTING VARIABLE TYPES you'll learn about the different ways BASIC stores numbers, and how you can control the process using DEFINT, DEFSNG, DEFDBL, DEFSTR, and CDBL, CINT, CSNG.

Finally, in MISCELLANEOUS BUT USEFUL, you'll learn how VARPTR lets you look into the inner workings of BASIC, how OCT$, HEX$, &H and &O can convert from one number system to another, and how SWAP lets you exchange two variables.

PROGRAM STRUCTURE

The longer and more complex your programs become, the more important it is to pay attention to the overall structure of the program. Have you ever written a long program and then, a month (or even a week) later, looked at it again and been mystified by parts of it? If so, you are not alone. Almost everyone who programs has found himself at one time or another saying, "Now what in the world did I mean by that?"

A well-structured program is easy for the programmer himself to understand (and, therefore, easy to change later if necessary) and is also easy for someone else to understand. (And sooner or later you will be writing programs that other people will want to use—and modify.)

What do we mean by "well structured?" One of the most important characteristics of a well-structured program is that it is divided into separate functional units. That is, the program consists of a number of distinct sections, each of which performs a specific well-defined function. Look back at the TIC-TAC-TOE program of Fig. 3-8. Here, the different sections of the program are separated by REM statements, and each section is also described by a REM statement, such as CLEAR ARRAY AND ZERO VARIABLES or CHECK FOR OPENING MOVE.

Also, some sections of the TIC-TAC-TOE program consist of subroutines, like DISPLAY BOARD (lines 7000-7020). Subroutines are a good way to divide your program into functional units, because you know that every time you "call" (execute a GOSUB to) the subroutine from the main program, it will do exactly the same thing. Once you've written a subroutine and gotten it to work, you can forget about it and go on to other parts of your program.

Another way to separate the different parts of your program is with FOR...NEXT loops (see Chapter 2), because you know that whatever instructions are between the FOR and the NEXT will always be executed the same way, the number of times specified in the FOR statement.

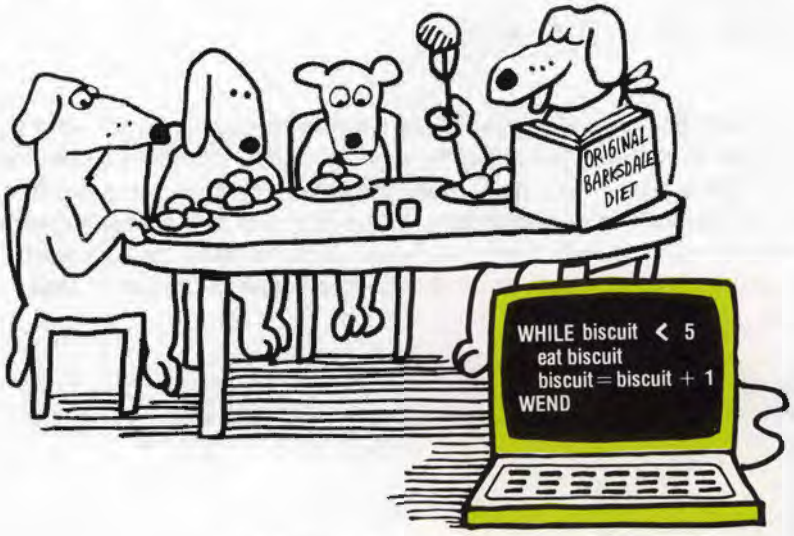
The section which follows will introduce some BASIC statements which, like GOSUB and FOR...NEXT, make it easier to write your program in separate functional units, thus creating a well-structured program that is easy to read, understand, debug, and modify.

WHILE...WEND—Easy Looping

WHILE...WEND means, "WHILE something is true, execute the statements between the WHILE and the WEND." (WEND indicates the END of the WHILE...WEND loop.)

In Chapter 2 you learned how to repeat an operation a certain number of times, by using either an IF statement and a GOTO or a FOR...NEXT loop.

As an example, let's write some routines to find the factorial of a number. (Recall that the factorial of a number is simply the number multiplied by all the numbers smaller than itself. For example, the factorial of 5 equals 5 * 4 * 3 * 2 * 1, or 120.)



WHILE-WEND

Here's the program using an IF statement and a GOTO:

```
10 INPUT"NUMBER"; N
20 F=1
30 F=F*N
40 N=N-1
50 IF N>0 THEN GOTO 30
60 PRINT"FACTORIAL IS"; F
```

Try it! The program works all right, but using the IF statement and the GOTO means that in order to figure out what statements are going to be repeated N times in the loop, we have to read down to the IF statement, and then search back up to the line number it references. This isn't hard in a short loop, but for long loops in larger programs it can be very confusing. Many programmers today try to avoid the use of GOTO statements for this reason.

Perhaps you've been on the kind of treasure-hunt where you find a note that says, "Look for the next clue in a hollow tree," and when you find the hollow tree there's a note saying, "Go to the fourth fence-post from the road." Using too many GOTOs is a little like a treasure-hunt: you're never quite sure what you're doing or where you're going next.

Here's the same program using a FOR...NEXT loop:

```
10 INPUT"NUMBER"; N
20 F=1
30 FOR J = N TO 1 STEP -1
40 F=F*J
50 NEXT J
60 PRINT"FACTORIAL IS"; F
```

That's not bad. The FOR and NEXT statements clearly set off what's going to be done in the loop. But notice how we've had to introduce a new variable J just to count how many times to do the loop. The more variables in a program, the harder it is to understand, so this is a step in the wrong direction.

Using WHILE...WEND both does away with the need for a counting variable, and makes for a clearer program. Here's how the program looks using WHILE...WEND:

```
10 INPUT"NUMBER"; N
20 F=1
30 WHILE N>0
40   F=F*N
50   N=N-1
60 WEND
70 PRINT"FACTORIAL IS"; F
```

The N>0 following the WHILE in line 30 is the third essential element of the WHILE...WEND loop, besides the WHILE and the WEND. When the computer gets to the WHILE statement in the program it remembers the expression N>0. It then executes the statements down to the WEND. When it gets to the WEND it evaluates the expression following the WHILE, N>0 in this case, to see if it is true. If it is, it goes back up to the WHILE and executes all the statements in the loop again. If the expression is not true, it goes to the next statement following WEND.

In this example, as long as N>0, the WHILE...WEND loop will keep multiplying the product F by the number N, and then reducing N by one. When N=0, the process stops.

The WHILE...WEND loop makes for a more readable program, since we can see immediately that the statements between WHILE and WEND will be executed repeatedly as long as N>0. (Indenting the lines between the WHILE and the WEND makes it even easier to follow.) There are no GOTOs and no unnecessary variables.

ELSE—An Alternative to THEN

ELSE is an optional addition to an IF...THEN statement. It means, "IF the expression following the IF is not true, THEN do the statements following the ELSE."

You learned about IF...THEN conditional clauses in Chapter 2. The idea there was, IF the premise (the expression following IF) is true, THEN the statements following the THEN will be executed. Otherwise nothing happens, and control goes to the next program line.



IF THEN ELSE

For example, suppose we want to use IF...THEN to print an appropriate greeting: "Good Morning" if the time is before noon, and "Good Afternoon" otherwise. Assume T is the hour (in 24-hour time). We could write:

```
10 IF T<12 THEN PRINT "GOOD MORNING!"
20 IF T=>12 THEN PRINT "GOOD AFTERNOON!"
```

But the expression T=>12 in line 20 seems as if it should be unnecessary—after all, all it does is to say the opposite of T<12 in line 10. And it makes the program harder to read.

We can get rid of it if we rewrite the program this way:

```
10 IF T<12 THEN PRINT"GOOD MORNING" : GOTO 30
20 PRINT"GOOD AFTERNOON"
30 (next program line)
```

But now the program is harder to understand because of the GOTO we had to add. It seems like we can't win!

However, by using ELSE we can let the program do one thing if the premise is true, and something ELSE if it is false:

```
10 T<12 THEN PRINT "GOOD MORNING!"
        ELSE PRINT "GOOD AFTERNOON!"
```

Isn't that easier to read?

An IF...THEN...ELSE statement can also be used as the result of another IF...THEN...ELSE statement. Here's an example that prints an appropriate greeting depending on your age:

```
10 INPUT "ENTER YOUR AGE IN YEARS"; AG
20 IF AG<18 THEN PRINT "HI, KID!"
            ELSE IF AG<65 THEN PRINT "HI!"
                          ELSE PRINT "HI, POPS!"
```

Notice the way the lines are indented to match up the corresponding THENs and ELSEs. This also makes the listing easier to read (although it takes up more memory).

Finding Prime Numbers

Here's a program that uses both WHILE...WEND and ELSE statements. The program will tell you whether a number you type in is a prime number, and if not, what its divisors are. A prime is a number that can only be divided by itself and one. Mathematicians consider prime numbers to be the fundamental

building blocks for our number system, and they are important in such fields as cryptography—the use of codes for transmitting information.

```
 10 PRINT"PRIME NUMBER FINDER. TYPE 0 TO TERMINATE."
 20 WHILE N<>0
 30   INPUT"NUMBER"; N
 40   I=2 : NP=0
 50   WHILE I <= SQR(N)
 60     IF N/I - INT(N/I) = 0
        THEN PRINT N "IS DIVISIBLE BY" I : NP=1
 70     I=I+1
 80   WEND
 90   IF N=0 THEN PRINT "FINISHED."
            ELSE IF NP=1 THEN PRINT N "IS NOT A PRIME."
                         ELSE PRINT N "IS A PRIME."
100 WEND
110 STOP
```

As you can see, we've tried something new here. Look at all those WHILEs and WENDs. What we've done is to nest one WHILE... WEND loop inside another, just as we nested one FOR...NEXT loop inside another in Chapter 3. The outer WHILE...WEND loop, from line 20 to line 100, has a very simple job: it waits to see if you've typed 0, which terminates the routine. If so, control goes to the statement following the loop, in this case the STOP at line 110. .

The inner WHILE...WEND loop, from line 50 to line 80, has a more complex function. Starting with I=2, it checks to see if N is exactly divisible by I. How does it do this? In line 60 it divides N by I. Then it takes the integer part of N/I, using the INT function. (Recall that INT has the effect of chopping off any digits to the right of the decimal place.) Then it subtracts these two values. If the difference is zero, then N is exactly divisible by I, and is therefore not a prime number.

For example, suppose that N is 10 and I is 2. When we substitute these values in N/I we get 10/2, which is exactly 5: there are no digits to the right of the decimal point. Now the integer part of 5 is also exactly 5. So N/I is equal to INT(N/I), and when we subtract them the result is zero. When the program figures this out in line 60 it will print "10 IS DIVISIBLE BY 2." It will also set a variable NP (for "Not Prime") to one. (A variable used in this way is sometimes called a "flag.") This is a way the program can "remember" that the number is not a prime. Then it will increase I by one and do the same calculation again.

This time I will be 3. 10 divided by 3 is 3.33333, which certainly does have digits to the right of the decimal point. When the program evaluates INT(N/I) it will get 3 exactly, which is not the same as 3.33333. The difference is, therefore, not zero, and the program will not print the "is divisible by" message, nor will it set the NP flag to one.

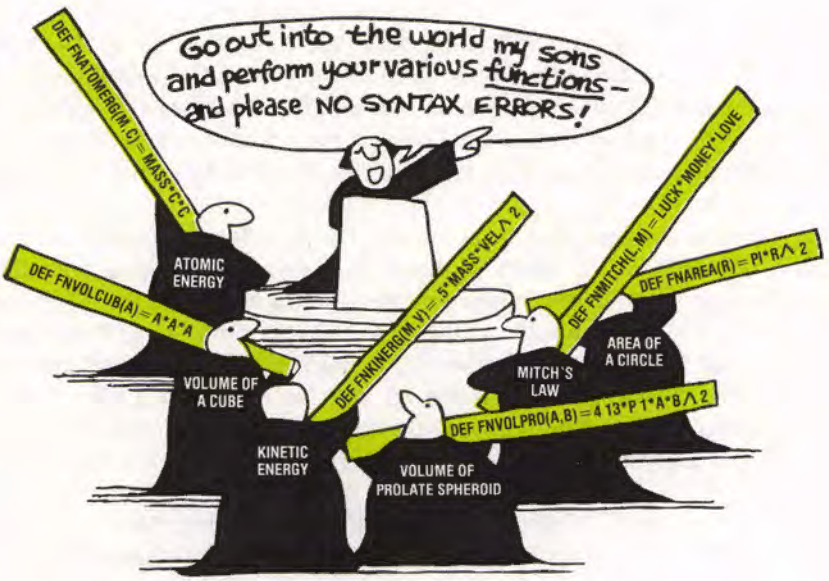
The program keeps dividing N by larger and larger values of I to see if it is exactly divisible by any of them. When it reaches a value of I which is equal to the square root of N, it knows it can stop. Why? Because all the divisors larger than the square root, when divided into the number, will yield answers smaller than the square root, and the program will have already tried all these numbers. For example, the square root of 12 is 3.4641. We divide by 2, and then by 3. But we don't need to divide by 4, since 12 divided by 4 equals 3, and we've already divided by 3.

Once the inner WHILE...WEND loop is finished, the program prints the appropriate message: if N=0 it knows that the user typed 0 to terminate the program, so it types "FINISHED." If N is not zero, then it checks to see if the NP ("not prime") flag is set. If it is—as the result of N being exactly divisible by one of the values of I—then it prints the number and "IS NOT A PRIME." If the NP flag is not set, the program prints the number and "IS A PRIME."

DEF FN—Adding to BASIC's Vocabulary

DEF FN stands for "DEFine FuNction."

An example would be: DEF FN XX(A,B) = A * B. This BASIC statement means that we are defining the function XX to be A times B. XX can be any one- or two-character name, like other BASIC variables. The variables A and B are called the "arguments" of the function. These "arguments" are used in an interesting way—we'll get back to them later.



DEF FN

Once you've defined a function (like XX), using a DEF FN statement somewhere at the beginning of your program, you can use it any time you want throughout your program, the same way you use the built-in BASIC functions like SIN, AND, or LEFT$.

How do you use the function in your program? With a statement like C=FN XX(J,K). What this statement does is to take the values of J and K, multiplies them (because the function XX has already been defined as one that multiplies its two arguments together), and assigns the result to the variable C. Let's put this all together in a little program:

```
10 DEF FN XX(A,B) = A * B
20 J=3
30 K=4
40 C=FN XX(J,K)
50 PRINT C
```

If you RUN this program you should get a 12 printed out on the screen, because 12 is the product of 3 times 4.

But now we can almost hear you saying, "Wait a minute! How can you use A and B in the definition of the function, and J and K when you want to evaluate it later?"

Good question. The answer is this: the arguments used when the function is defined do not have to be the same as those used when it is "called" (evaluated) later in the program.

The arguments used in the DEF FN statement are what are sometimes called "dummy variables." That means that they are used only in the definition, and actually never take on any values. They make it clear what variables are going to be used, and how the function that is being defined acts on them, but they are not used at all outside of the definition. In fact, if you use A and B somewhere else in your program as regular variables, and then execute a DEF FN statement that uses A and B, the values you originally assigned to A and B will not be changed.

On the other hand, the variables you use to call the function later in your program are regular BASIC variables whose values will be used to calculate the function. You can use different variables as arguments every time you call the function. You can call it once with J and K, the next time with L and M, and so on—whatever your program requires. The values of whatever variables you put in the parentheses will be used to calculate the function.

This is a very useful feature of DEF FN, because it means that, unlike a subroutine, you do not have to always use the same set of variables to perform a function.

As an example of what we mean, look back at the "Enhanced Loan Program" in Fig. 2-3 of Chapter 2. Line numbers 5000-5020 of this program contain a "round-off" routine, which is called as a subroutine with a GOSUB 5000 from lines 1314 and 1534. Here's the subroutine:

```
5000 REM -- ROUND OFF SUBROUTINE
5010 LET X = (INT(100*(X+.005)))/100
5020 RETURN
```

Let's change this round-off subroutine to a "defined function," using DEF FN. At the beginning of the program we will put the statements

```
900 REM -- ROUND-OFF FUNCTION
910 DEF FN RO(X) = (INT(100*(X+.005)))/100
```

Here the DEF FN statement defines the function RO (for Round Off), of the dummy variable X, to be equal to (INT (100*(X+.005)))/100.

We can now delete the subroutine (lines 5000-5020) since the new function definition takes its place. We can also delete lines 1312, 1316, 1532 and 1536. They were needed because the old subroutine had to round off a variable called X, while now, since we are using a defined function, we can operate on any variable name we want. Then we change lines 1314 and 1534:

```
1314 A=FN RO(A)
1534 I1=FN RO(I1)
```

What happens now when we RUN the program? First, the program encounters the DEF FN statement. It doesn't compute anything here; it simply remembers that RO is now a defined function and what the function is. Later, the program will get to line 1314, which is A=FN RO(A). Seeing the FN RO, it goes back and looks up the definition of RO in line 910—and finds it is the Round Off function. It knows it has to perform this function on whatever variable is in the parentheses following RO in line 1314—in this case A. So it uses the X in the definition on line 910 as a pattern for doing this. Every operation that is shown being done on X in line 910, the program will now actually do to A. First it will add .005 to A, because it sees that was done to X in the definition. Then it will take the integer part of the result, and so forth. The result is that A is rounded off as we wanted.

Similarly the program will round off the variable I1 when it gets to line 1534.

Notice how we have not only shortened the program by five lines, we have also made it more readable, since FN RO is more easily recognized as the "Round Off" function than is GOSUB 5000.

The above example used only one argument, X. But DEF FN can use any number of arguments. Here's an example of a program using two arguments.

This program computes the area of a triangle, using the relationship that the area of a triangle is equal to half of its base times its height.

```
10 DEF FN AT(B,H) =B* H / 2
20 INPUT"BASE"; B1
30 INPUT"HEIGHT"; H1
40 PRINT"AREA IS"; FN AT(B1,H1)
RUN
BASE? 10
HEIGHT? 5
AREA IS 25
```

Here, the function AT (area of triangle) has the arguments B and H (for base and height) and is defined as B * H /2 in line 10. When we "call" the function in line 40, we do so with the variables B1 and H1, whose values we obtained in the INPUT statements on lines 20 and 30. In a longer program we might "call" the function AT again with variables B2 and H2, or any other names we like.

Notice that we don't have to assign the value which the function AT calculates to a variable before we can PRINT it—we simply include the function in the print statement as we could with any other variable or expression.

ADVANCED INPUT/OUTPUT

So far you have used INPUT and PRINT to permit your program to communicate with the human being who is sitting at the keyboard using it. This section will describe some statements which will extend the flexibility and ease of typing data into the computer, and of reading it out on the screen.

These statements will be particularly important when you write "serious" programs—for instance, for business applications—which involve large amounts of numerical data. They will also be helpful for programs which may be used by people who do not know very much about computers or programming, as in a typical office. In this section you will learn how to easily format the output of tables and lists, how to make better-looking screen displays, and how to help ensure that the data typed in by the users of your program will not cause an inadvertent "crash" or program failure.

LINE INPUT—Typing Anything You Like

LINE INPUT means "INPUT a LINE of characters (usually up to 255 characters), which can include commas, colons, and quotes. Assign this string to a single variable."

In many situations INPUT is the only statement you need to type information into your computer from the keyboard. But sometimes it can be a little awkward.

So far in the examples in this book, the input to programs has been in the form of numbers or one-word character strings. But suppose you are writing a program that requires complete sentences for input, like "Eliza." "Eliza" is a game program that pretends to be a psychiatrist. It asks if you have any problems, and you answer it in conversational English. For example, you might type in "My family doesn't understand me."

The trouble with using the INPUT statement in this situation is that if you include a comma in the sentence you are typing in, all the words following the comma will be chopped off, because INPUT will think they belong to a different variable. (The same is true of colons.)

Here's an example of what can happen:

```
10 PRINT"TELL ME YOUR PROBLEM."
20 INPUT A$
30 PRINT A$
RUN
TELL ME YOUR PROBLEM.
?WELL, I DON'T GET ALONG WITH MY BOSS.
WELL
```

Notice how the computer has "forgotten" all of the input sentence following "WELL." You can avoid this problem if you always enclose your input in quotes, but that can get tedious after a while. Also, the question mark, which INPUT always prints before your input, looks out of place here.

Substituting LINE INPUT A$ for INPUT A$ solves these problems.

```
10 PRINT"TELL ME YOUR PROBLEM."
20 LINE INPUT A$
30 PRINT A$
RUN
TELL ME YOUR PROBLEM.
WELL, I DON'T GET ALONG WITH MY BOSS.
WELL, I DON'T GET ALONG WITH MY BOSS.
```

Now the program "remembers" the entire input sentence. With LINE INPUT you can input any characters you want, including leading blanks, commas, colons, and quotes, and they will all be assigned to the variable A$. Only pressing "Return" (or "Enter") can terminate the input line you are typing in. LINE INPUT also does not automatically output the question mark as a prompt.

If you want you can use a "prompt string" in LINE INPUT. (This is similar to the prompt string with INPUT—see Chapter 2.) Let's combine lines 10 and 20 from the above program. The result is:

```
10 LINE INPUT"TELL ME YOUR PROBLEM."; A$
```

This will work just the same as lines 10 and 20 did in the above program.

One of the most important uses for LINE INPUT is to make sure that even if you make a mistake typing in your input, you will receive an error message that you can understand, so that you will know how to corrrect your mistake.

For example, let's look back at the Metric Conversion Program in Fig. 2-1. Suppose you are new to computers and are trying to use this program. You RUN the program, and get the same output as is shown on page 80. But now, because you are unsophisticated, you type "TWO" instead of "2." What will happen?

The INPUT statement is waiting for a numeric variable, either 1,2,3, or 9. When it sees the "TWO" it will immediately print:

```
?REDO
?
```

Now, what will you make of this? You may figure out that REDO means to try again, but you probably will not know exactly what your mistake was. Let's fix up the program to give a more self-expanatory error message.

First, change line 2070 to read:

```
2070 LINE INPUT"ENTER YOUR SELECTION"; S$
```

Then add these lines:

```
2075 IF S$="ONE" OR S$="TWO" OR S$="THREE" OR S$="NINE"
     THEN PRINT"PLEASE DO NOT SPELL OUT THE NUMBERS. USE
          THE DIGITS 1,2,3, OR 9." : GOTO 2070
2077 S= VAL(S$)
```

Now if you spell out the numbers you will get an error message that is readily understandable. Similarly, LINE INPUT can be used to "trap" other kinds of input errors ("trap" means to catch an error before it can cause trouble), such as inputting two variables where only one is expected.

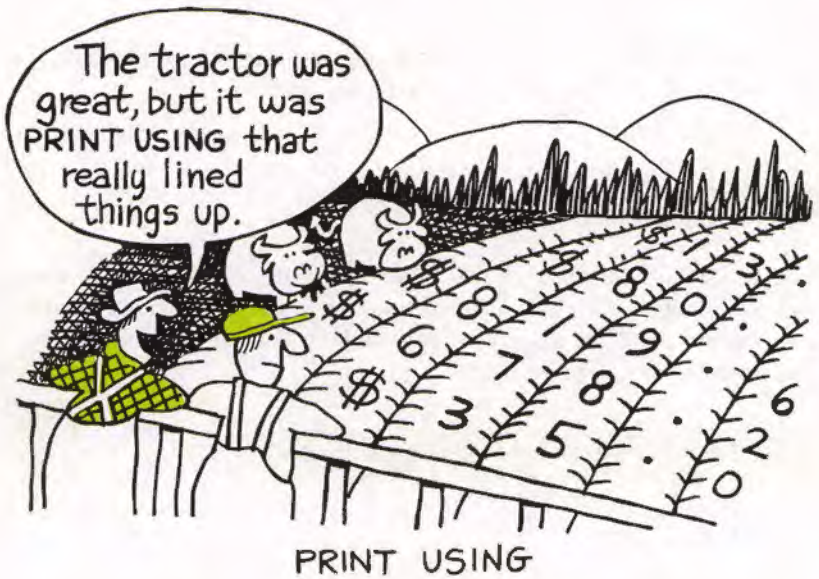
You have probably noticed something else here: since the input of a LINE INPUT statement is always assigned to a string variable (A$ in this example), if we want to turn it back into a number we have to change its type, using the ASC function as we did in line 2077.

PRINT USING—Super Output Formatter

PRINT USING means, "PRINT a number of variables, USING the format which is specied by a special variable in the statement."

PRINT USING is a very powerful and versatile statement for making your output look good, especially if you are displaying data in tabular form. Here's an example of a PRINT USING statement that will print out three variables:

```
PRINT USING X$; A,B,C
```



PRINT USING

A, B and C can be any variables, just as in a regular PRINT statement. It's the string variable preceding the semicolon (X$ in this case) that gives the statement its power. What X$ does is to specify the format in which the variables A,B,C will be printed. How does it do that? Suppose we write the following program:

```
10 INPUT A
20 X$ = "$$###.##"
30 PRINT USING X$; A
```

What are all those funny-looking characters in the X$ string? Let's examine them one by one and see what they do.

$$ means, "print a dollar sign just to the left of the left-most digit of the variable."

means, "print one of the variable's digits here." So ## means, "print two digits here."

. (a period) means, "put a decimal point in this position in the variable."

You can think of the X$ string as a pattern for how the output is going to look on the screen. When the program gets to the PRINT USING statement it fills in the value of the variable, using the positions shown in X$. In the example above, X$ is saying, "print a dollar sign, then two digits, then the decimal point, then two more digits." Try running the program.

```
RUN
9
$9.00
.1
$0.10
4.055
$4.06
```

Isn't that the slickest thing you've ever seen? The program not only puts the dollar sign in the right place, it performs a round-off function too!

Imagine the trouble it would be to do all the formatting shown above using regular PRINT statements. Here's how we might try to get the dollar sign as close to the number N as possible if we expected up to four digits:

```
10 INPUT"NUMBER"; N
20 IF N<1 THEN PRINT"          $"; : GOTO 60
30 IF N<10 THEN PRINT"         $"; : GOTO 60
40 IF N<100 THEN PRINT"        $"; : GOTO 60
50 PRINT"$";
60 PRINT N
70 GOTO 10


READY
>RUN
NUMBER? .01
                $ .01
NUMBER? .1
                $ .1
NUMBER? 1
                $ 1
NUMBER? 10
               $ 10
NUMBER? 100
              $ 100
NUMBER? 1000
             $ 1000
```

Well, at least the decimal points line up! But no matter what we do we can't get the dollar-sign right up against the left-most digit, because BASIC (except in a PRINT USING statement) always leaves room for a minus sign to the left of the number, whether the number is negative or not. So even if we went to the trouble to add a round-off routine to the above program, we still wouldn't be able to do what a PRINT USING statement does. And, of course, our program would be much more cumbersome.

Let's see how PRINT USING might be used in an actual program. Go back to the Enhanced Loan Program of Fig. 2-3. If you typed it in and ran it on your computer, you may have noticed some problems with the format of the output. First, the numbers don't line up—the decimal points aren't all in the same column. Then, although the numbers are all in dollars, there are no dollar signs; and sometimes there is only one digit to the right of the decimal point. Not only that, but on some computers there is no room for five variable fields on the screen when they are separated by commas, so that the row "wraps around" to the next line on the screen, making a real mess. And, to top it off, if the computer has to output a very small number, as it does in the last column, it changes to something called "exponential notation," which looks rather unsightly in this sort of table. (See Appendix F for a discussion of expontential notation).

Modify the program in Fig. 2-3 like this:

```
1705 X$="### $$###.## $$###.##$$###,###.## $$###,### .##"
1710 PRINT USING X$; J,I1,P,B,T
```

Also, delete the round-off subroutine at 5000 and all the references to it, since PRINT USING does its own rounding off. RUN the program. (An example of output from this modified program is shown in Fig. 6-1.)

Wow! Everything is in neat columns, the dollarsigns are where they should be, numbers are rounded off, and there are even commas to separate the hundreds column from the thousands column. (That's what the commas are for in X$.) What determines the spacing of the columns? The columns are separated by the same number of spaces that appear in X$.

There are a lot more clever things you can do with PRINT USING, such as formatting string variables. However, the characters used to specify what to do can vary from one computer to the next, so we won't describe them here. What you've learned so far should give you an idea of how useful this statement can be.

INKEY$—Finer Control of the Keyboard

INKEY$ means, "INput a single KEYstroke from the keyboard." It is not necessary to terminate this function with "return" (or "enter").

AMOUNT OF LOAN 5000
ANNUAL INTEREST RATE .09
TERM IN YEARS 4
MONTHLY PAYMENT IS $124.43

MONTH	INTEREST	PRINCIPLE	BALANCE	TOTAL-INT
1	$37.50	$86.93	$4,913.07	$37.50
2	$36.85	$87.58	$4,825.49	$74.35
3	$36.19	$88.24	$4,737.26	$110.54
4	$35.53	$88.90	$4,648.36	$146.07
5	$34.86	$89.56	$4,558.79	$180.93
6	$34.19	$90.24	$4,468.56	$215.12
7	$33.51	$90.91	$4,377.65	$248.64
8	$32.83	$91.59	$4,286.05	$281.47
9	$32.15	$92.28	$4,193.77	$313.61
10	$31.45	$92.97	$4,100.79	$345.07
11	$30.75	$93.67	$4,007.12	$375.82
12	$30.05	$94.37	$3,912.75	$405.88
13	$29.35	$95.08	$3,817.67	$435.22
14	$28.63	$95.79	$3,721.87	$463.85
15	$27.91	$96.51	$3,625.36	$491.77
16	$27.19	$97.24	$3,528.12	$518.96
17	$26.46	$97.97	$3,430.16	$545.42
18	$25.73	$98.70	$3,331.45	$571.15
19	$24.99	$99.44	$3,232.01	$596.13
20	$24.24	$100.19	$3,131.83	$620.37
21	$23.49	$100.94	$3,030.89	$643.86
22	$22.73	$101.70	$2,929.19	$666.59
23	$21.97	$102.46	$2,826.73	$688.56
24	$21.20	$103.23	$2,723.51	$709.76
25	$20.43	$104.00	$2,619.51	$730.19
26	$19.65	$104.78	$2,514.72	$749.83
27	$18.86	$105.57	$2,409.16	$768.70
28	$18.07	$106.36	$2,302.80	$786.76
29	$17.27	$107.16	$2,195.64	$804.03
30	$16.47	$107.96	$2,087.68	$820.50
31	$15.66	$108.77	$1,978.91	$836.16
32	$14.84	$109.59	$1,869.33	$851.00
33	$14.02	$110.41	$1,758.92	$865.02
34	$13.19	$111.24	$1,647.68	$878.21
35	$12.36	$112.07	$1,535.61	$890.57
36	$11.52	$112.91	$1,422.70	$902.09
37	$10.67	$113.76	$1,308.95	$912.76
38	$9.82	$114.61	$1,194.34	$922.58
39	$8.96	$115.47	$1,078.87	$931.53
40	$8.09	$116.34	$962.53	$939.62
41	$7.22	$117.21	$845.32	$946.84
42	$6.34	$118.09	$727.24	$953.18
43	$5.45	$118.97	$608.26	$958.64
44	$4.56	$119.87	$488.40	$963.20
45	$3.66	$120.76	$367.63	$966.86
46	$2.76	$121.67	$245.96	$969.62
47	$1.84	$122.58	$123.38	$971.46
48	$0.93	$123.38	$0.00	$972.39

Fig. 6-1. Sample output after modifying Loan program of Fig. 2-3.

When you are entering numbers or string data from the keyboard you usually type in the characters, and then, since the program has no other way of telling when you're done, you press "return" (or enter") to tell it to start



INKEY$

processing the number or string. This works well in the usual case where several characters must be typed for each input. But what about the situation where a one-character input is all that's necessary? If you know you're going to type only one character and the program knows you're going to type only one character, then having to press "return" after you type it is just a nuisance.

Take, for example, the Metric Conversion program in Fig. 2-4. The program begins by giving you a "menu" of four choices. It prints:

```
1 - INCHES TO CENTIMETERS
2 - POUNDS TO KILOGRAMS
3 - QUARTS TO LITERS
9 - TO END THE PROGRAM
```

Then it executes line 2070, which is

```
2070 INPUT "ENTER YOUR SELECTION"; S
```

You then respond with a number, say 1 if you want to convert inches to centimeters. After you type the number you press "return." But really, why

should you have to press "return?" Since the only expected inputs are one digit long (either 1,2,3 or 9), why not let the program go straight to the conversion routine you've selected as soon as you've typed the number? Of course, you can't do that if you use INPUT or LINE INPUT, which will wait forever for "return" to be pressed before they will go on with the rest of the program. However, using INKEY$, it is possible to eliminate the need for pressing "return."

Let's change line 2070 to use INKEY$:

```
2065 PRINT"ENTER YOUR SELECTION";
2070 S$=INKEY$ : IF S$="" THEN GOTO 2070
2075 PRINT S$ : S=VAL(S$)
```

As you can see, there are important differences between using INKEY$ and using INPUT. First, INKEY$ is not a complete BASIC statement as INPUT is; it is a BASIC variable. More exactly, it is a one-character string variable. So the first thing we do is to let the variable that we want to examine, S$ in this case, be equal to (take on the value of) INKEY$. We do this in the first part of line 2070, after we have printed out the "ENTER YOUR SELECTION" message in line 2065. (We have to print this prompt message in a separate statement because it is not possible to put a prompt string in INKEY$.)

So far so good. You probably expect that the next thing we'll do is to analyze S$ to see if it is 1, 2, 3 or 9. But not so fast! What happens if, when the program gets to line 2070, the user hasn't typed anything yet? Actually, this is a pretty good bet, since the time that elapses between line 2065, when the program prints "ENTER YOUR SELECTION" and line 2070, when it first checks for your input, is less than a millisecond. Unless you have very good reaction time, there won't be anything typed on the keyboard for INKEY$ to read. In this case there is no point in the program trying to analyze the input—there isn't any input yet. What we want to be able to do is wait until some key is pressed. That's what the second part of line 2070 does: if INKEY$ = "" THEN GOTO 2070. The two quote marks with nothing between them mean "no character." (This is sometimes called the "null string.") It is the same as saying that nothing was typed. So the statement says: "If nothing was typed, go execute line 2070 again." Since the first part of line 2070 simply reads the keyboard, the program will continue to execute this line until a character is typed on the keyboard—in other words, until INKEY$ becomes something other than the null string.

Once a character is typed, INKEY$ takes on the value of that character, the program finds that S$ is no longer equal to "", and goes on (finally!) to the next line number, 2075. In this line the first thing we do is to print the character—since INKEY$ does not print (or "echo") the character it has gotten from the keyboard, and the user would probably feel uncomfortable about typing something and not seeing it appear on the screen. Then we change the

string character S$ to a numerical value S, using the VAL function, so that the value of S can be analyzed by lines 2081 to 2089. (Refer to the section on string functions in Chapter 4 for a description of ASC.)

Menu-selection routines such as this are typical situations where the INKEY$ function is useful. Another place where INKEY$ is useful is where you have to read a large amount of text on the screen, and the program needs you to signal when you are done with each screenfull so it can print out the next one.

You could write:

```
1000 INPUT"PRESS RETURN TO CONTINUE"; X
```

but then the "return" key is the only key that will work.

Instead, let's use INKEY$:

```
1000 PRINT"PRESS ANY KEY TO CONTINUE: ";
1010 IF INKEY$="" THEN GOTO 1010
```

After printing the prompt in line 1000, the program will wait—in line 1010—until some key, no matter what, is pressed before continuing on with the rest of the program. Now, even someone who doesn't know where the "return" key is can read the next screenfull of instructions.

In the above example, since we don't care what key was pressed and, thus, don't need to store the character, we don't need to assign INKEY$ to another variable like S$—we can check directly to see if it is equal to the null string and, if not, proceed with the rest of the program.

INKEY$ is also useful in game programs, where you might want to move something (a spaceship, for example) around the screen using the arrow (or cursor control) keys. The exact instructions for doing this kind of thing usually depend on what kind of graphics capability your computer has, but for a simple example see the "Mini Space Invaders" game in the description of SPACE$ later in this chapter.

INPUT$—Reading More Keystrokes

INPUT$(N) means, "INPUT N keystrokes from the keyboard (or other device)." It is not necessary to terminate this function with "return" (or "enter").

INPUT$ is like INKEY$ (see the preceding description) in that it reads the keyboard and does not need to be terminated by "return." However, where INKEY$ will read only one character, INPUT$ will read any number of characters (up to 255). Exactly how many characters it will read is specified by the

number N in the parentheses following the INPUT$. INPUT$ will wait until this many characters have been typed before it goes on to the next statement.

Here's an example of how we might use INPUT$. Suppose we are writing a program in which we want to type in the day of the week. We then want to assign to a variable DW (Day of the Week) a number from 1 to 7 depending on the day.

Now, it's not really necessary to wait for the complete word (like "Wednesday") to be typed out. The first two letters are sufficient to determine which day we have in mind. (One letter isn't enough because Saturday and Sunday both start with "S" and Tuesday and Thursday both start with T.)

So we can write the following routine:

```
100 PRINT"PLEASE TYPE THE DAY OF THE WEEK"
110 DA$=INPUT$(2)
120 IF DA$="SU" THEN DW=1
130 IF DA$="MO" THEN DW=2
140 IF DA$="TU" THEN DW=3
150 IF DA$="WE" THEN DW=4
160 IF DA$="TH" THEN DW=5
170 IF DA$="FR" THEN DW=6
180 IF DA$="SA" THEN DW=7
190 PRINT"THANK YOU."
```

The program waits, in line 110, for us to type two letters. When we do, the program goes on to analyze the letters, without waiting for the rest of the word, and interrupts us (if we are still typing) with a "THANK YOU" message.

One difference to notice between INPUT$ and INKEY$ is that INPUT$ always waits for N characters to be typed, even if N is 1. It never returns the null string (as INKEY$ does), so we don't have to check for the null string and keep executing the statement again and again until we find a different character, as we do with INKEY$.

For example, if we want the program to pause until we press any key, we can write

```
1000 PRINT"PRESS ANY KEY TO CONTINUE: ";
1010 INPUT $(1)
```

This is simpler than the corresponding example shown in the description of INKEY$.

The greatest usefulness of INPUT$ is in reading data from other devices than the keyboard, such as a telephone modem. This is because (like INKEY$) INPUT$ will accept any ASCII input without printing it, including control codes, backspaces, and so on. In communications over a telephone modem these control codes may have different meanings than they do in normal

BASIC. For example, receiving a "return" character doesn't necessarily mean that the message is over. Using INPUT$ permits the program to analyze these characters and take appropriate action, rather than letting the regular INPUT statement do all the analyzing. However, using INPUT$ in this way is complex and varies from one machine to another. For further information you should read the manuals for your own computer.

POS—Where Is the Cursor?

POS(N) means, "the horizontal POSition of the cursor."

This function will return a number between 1 and whatever the width of your display screen is (from 1 to 64 on the TRS-80 Mod I or III, from 1 to 40 on the Apple II, etc). This number tells you how many characters along the current line the cursor is positioned. (It gives no information on the vertical position of the cursor—that is, which line the cursor is on.) The "N" in parentheses after the POS is a "dummy" argument: it doesn't matter what number you put there.

Since whatever you PRINT on the screen or INPUT from the user will be written starting at the screen location where the cursor is positioned, this function can be a useful one for making sure that input and output on the screen looks the way you want it to.

Suppose for example that your program is PRINTing a list of numbers of different sizes from an array "A," numbers like 403,629, 7.4, 0.000043 and so on. What we want to do is to PRINT as many of these numbers as we can on the same line, but when we get to the end of the line we don't want the number to be "split," with half the number on one line and half on the next line. Since the PRINT statement doesn't care if it splits a number this way, we'll use the POS function to make sure that every number is printed with all its digits on the same line.

Let's assume that none of the numbers will have more than 8 characters, including the decimal point, and that we will separate each pair of numbers with a space. Also, we'll suppose that we start off with 50 numbers in our array "A," and that the screen is 40 characters wide.

```
10 FOR J=1 to 50
20 PRINT A(J);
30 PRINT " ";
40 IF POS(0) > 31 THEN PRINT
50 NEXT J
```

For each of the 50 numbers that we are going to print, we first print the number, and then a space, in lines 20 and 30. By ending both these PRINT statements with semicolons, we keep our output on the same line. Since the 8 characters of the number, plus 1 for the space in between, makes nine, we

want to make sure that we are no closer than 9 characters to the edge of the screen when we print each number. This will ensure that there's room to print the number without splitting it at the edge of the screen (This is called "wrap around.").

We do this in line 40, which asks if POS(0) (the cursor position) is greater than 31 (since 40 minus 9 is 31). If it is, we execute a PRINT statement, which has the effect of causing a linefeed—that is, moving the cursor down to the next line. If POS(0) is not greater than 31, we simply continue to print the next number, on the same line. The loop continues on in this way until all 50 numbers have been printed.

ADVANCED STRING FUNCTIONS

When most people think of computers they probably think of machines whose purpose is to perform numerical calculations. It's true that computers got their start as "number crunchers"—devices to do numerical calculations as rapidly as possible. Early programming languages, like FORTRAN, were oriented almost exclusively toward numbers, and had only a primitive ability to deal with anything else.

Over the years, however, people began to realize that computers could be used to process other sorts of data besides numbers, like pictures, and sound, and words. Pictures (computer graphics) and music can be implemented to varying degrees on a variety of personal computers, but the BASIC statements used vary a great deal from one machine to another.

However, the manipulation of "character strings" (letters, words and sentences) is considerably more standardized among the different versions of BASIC.

String-handling capabilities are useful in business programs for handling data which is mostly text-oriented, such as mailing lists and inventory control. But they have also made possible the use of computers for a variety of other kinds of programs: educational, entertainment, and even artistic. There are, for example, the "Adventure" games, in which, using short sentences (like "GO NORTH"), you attempt to find your way out of various mazelike castles, haunted houses, and so forth. Then there is the famous "Eliza" program, which imitates a psychiatrist (see the section under LINE INPUT for a brief description of this program); there is a growing number of programs that write short stories or poetry, and there are interactive computer stories in which you can "take part" in the story, talk to the other characters, and influence the plot.

In this section we will cover some BASIC functions that make manipulating strings faster and easier and, thus, help promote the writing of these types of programs.

INSTR—Searching for Strings

INSTR means, "Look for one string IN another STRing."

INSTR is a very useful and powerful function, usually used to look for the occurrence of a specific word or name in a sentence or data file. It has this general form:

```
X=INSTR(N,S$,W$)
```

This statement will look for the string W$ (which might be a word) in the string S$ (which might be a sentence), starting at character position N in string S$. It will return the character position in the string S$ at which the match begins, or return 0 if no match is found (that is, these numeric values will be assigned to the variable X). The N is optional; if it is left out, the entire string S$ will be searched. Either S$ or W$ can be replaced with string constants—that is, actual strings enclosed in quotes, like "MITCH."

As an example, let's go back to the "psychiatrist" program referred to in the description of LINE INPUT in this chapter. This program first asks if the user has any problems he wants to discuss, then waits for a reply. Let's say that, if the user then types in a sentence that contains the word "MOTHER," the program (playing the role of a psychiatrist) will respond "TELL ME MORE ABOUT YOUR FAMILY."

Here's how, using INSTR, we can accomplish this:

```
 10 PRINT"TELL ME YOUR PROBLEM."
 20 LINE INPUT S$
 30 IF INSTR(S$,"MOTHER") THEN GOTO 100
 50 PRINT"PLEASE GO ON." : GOTO 20
100 PRINT"TELL ME MORE ABOUT YOUR FAMILY." : GOTO 20
```

Lines 10 and 20 are the same as in the example under LINE INPUT. Line 30, using INSTR, searches the input sentence S$ for the word "MOTHER." If it finds it, control branches to line 100. If this word is not found, the program, not knowing what else to say, prints "PLEASE GO ON."

While this program is perhaps not quite so clever as a real psychiatrist, it does give you an idea how INSTR can be used.

Notice that INSTR only detects the first occurrence of a word in a sentence (or a string in another string). If you want to see whether the word occurs more than once, you will have to search the sentence again, starting just past the point where the first match was found. That's where the variable N in the INSTR function comes in handy.

Here's a program that will count how many times the word "the" appears in a sentence that you type in:

```
10 LINE INPUT "TYPE SENTENCE: "; S$
20 W$="THE" : N=1 : J=0
30 P=INSTR(N,S$,W$)
40 IF P<>0 THEN J=J+1 : N=P+1 : GOTO 30
50 PRINT"THE WORD < THE > APPEARED" J "TIMES."
```

In this program the variable J counts how many times "THE" has been found in the string S$; P is the position in S$ where a match is found; and N is the starting position of the search in S$.

N starts at 1 (the character position at the beginning of the sentence) and J starts at 0, since we haven't yet counted any THEs. Then, in line 30, we examine the input sentence S$ (which was typed in line 10), starting at position N (1 in this case) to see if the word "THE" occurs in the sentence. If a match is found, P will be the position of the match in S$, and P<>0 will be true, so we increment J (J=J+1), and set N equal to P+1; that is, just past the starting position of the match. Then we go back to line 30 to search again. When no more matches are found, P=0, and control goes to line 50 where the total number of matches is printed out.

A Substitute for INSTR

If your computer doesn't have the INSTR function, you can perform the same function using a subroutine and two of the string functions you learned earlier: MID$ and LEN (see Chapter 4). The main difference in operation is that the subroutine will be slower than the built-in INSTR function.

This subroutine will search the string S$ for the occurrence of W$. It will return with the variable P equal to the position where the match was found in S$, or with P=0 if no match was found. (Note: the single quote (') has the same meaning as a REM.)

```
1000 LS=LEN(S$) : LW=LEN(W$) '-FIND LENGTHS OF WORD AND SENTENCE
1010 FOR P=1 TO LS-LW '---STEP THROUGH S$
1020 IF W$ = MID$(S$,P,LW) THEN RETURN '---LOOK FOR MATCH
1030 NEXT P '---MOVE TO NEXT POSITION IN S$
1040 P=0 : RETURN '---DIDN'T FIND IT, SO RETURN
```

After finding the lengths of S$ and W$ with the LEN function in line 1000, we start with the pointer P set to the first position in the string S$—namely 1. Then, in line 1020, we check to see if the string W$ is identical to the string MID$(S$,P,LW)—that is, the string in S$ which is LW characters long (the length of the word we are searching for) and which starts at position P. If it isn't, we increment P by one (in the NEXT P statement in line 1030) and compare W$ to the next group of LW characters. If we do all the values in the FOR...NEXT loop without finding a match, we set P=0 and return. If we do

find a match in line 1020, then we RETURN, with P still set to the position where the match occurred. The reason the FOR...NEXT loop only goes up to LS-LW is that when we get so close to the end of the sentence S$ that the word W$ won't fit anymore, then there is no point in looking further. Thus, if our word W$ is 6 characters long, there is no point in looking closer than 6 characters to the end of the sentence S$ for the word, since it won't fit. The character position at this point is the length of the sentence minus the length of the word, or LS-LW.

Let's use this subroutine in the psychiatrist program shown above, instead of the INSTR function. All we need to do is to add the subroutine and change line 30 to read:

```
30 W$="MOTHER" : GOSUB 1000 : IF P <> 0 THEN GOTO 100
```

This can be a very useful subroutine if you're dealing with a lot of strings and your version of BASIC doesn't have the INSTR function. Try it!

SPACE$—All Spaced Out

SPACE$(N) means, "Print N spaces on the screen (or printer)."

SPACE$(N) is a string variable equal to N spaces. You can use it in a PRINT statement or anywhere else you would use a string variable. If you PRINT STRING$(5), then 5 spaces will be printed on the screen. N itself can be a numerical variable or even an expression. That is, you can also use either of the following lines to print 5 spaces:

```
10 N=5 : PRINT SPACE$(N);

10 PRINT SPACE$(3+2);
```

SPACE$ is useful for formatting output in a variety of situations. For example, here's how SPACE$ can be used to center a title in the middle of the screen. Assume your screen is 80 characters wide. (If it isn't, change the 80 in line 20 to the appropriate value).

```
10 T$="BASIC PROGRAMMING PRIMER"
20 N = (80-LEN(T$))/2
30 PRINT SPACE$(N); T$
```

In line 10 we let T$ represent the title. The title is LEN(T$) characters long, which leaves 80 minus LEN(T$) spaces left over on the line. To center the title

we want half these spaces to precede the title and half to follow it (although we don't need to worry about the ones that follow); so we divide 80-LEN(T$) by 2. Then we print that many spaces (in line 30) and then the title itself, and, presto, the title is centered.

"Micro Space Invaders"

Here's a very simple game program that also uses SPACE$. Although it is not so exciting as an arcade game, it does pretty well for such a short program.

This is how the game works. When you RUN it, the numbers 1 through 5 appear at the top of the screen. These numbers represent five sectors of the galaxy. Soon an asterisk appears below one of the numbers. This asterisk represents the hostile space invader's ship. You try to shot down the ship with a phasor torpedo by pressing the appropriate number key: "2" if the ship appears in sector 2, "3" if it appears in sector 3, and so forth. When you press a number key a plus sign appears below the corresponding number. Unfortunately, you don't have much time to hit the right key, because the hostile ship is approaching at Warp Factor eleven. If you aren't fast enough, the program responds with the message "YOU TOOK TOO LONG," your own ship is partially destroyed, and you lose a point. If you are quick enough, but hit the wrong key (thus blasting a sector of the galaxy where there is no hostile ship), the program says "A MJSS," and you lose a point. Only if you press the right key in the time available do you score a hit on the hostile ship and add a point to your score. The program keeps track of your score for you.

Here are some examples of the game's output:

```
    1     2     3     4     5
                *
                +
A HIT!!
YOUR SCORE IS 1
PRESS SPACE BAR TO CONTINUE.

    1     2     3     4     5
                            *
                      +
A MISS.
YOUR SCORE IS 0
PRESS SPACE BAR TO CONTINUE.

    1     2     3     4     5
                *
YOU TOOK TOO LONG.
YOUR SCORE IS -1
PRESS SPACE BAR TO CONTINUE.
```

This is the program listing. If the program goes too fast for you, you can change the 30 in line 70 to a larger number. If it's too slow, change 30 to a smaller number.

```
 10 REM ---- INITIALIZE
 20 X=RND(5) : CLS
 30 PRINT"     1     2     3     4     5"
 40 REM ---- PRINT THE ASTERISK
 50 PRINT SPACE$ (X * 5); "*"
 60 REM ---- START THE TIMING LOOP
 70 FOR J=1 TO 30
 80 A$=INKEY$
 90 REM ---- IF NO KEY PRESSED, GO TO "NEXT"
100 IF A$="" THEN GOTO 170
110 REM ---- IF KEY PRESSED, PRINT PLUS
120 A=ASC(A$)-48 : PRINT SPACE$ (A * 5); "+"
130 REM ---- IF NUMBER PRESSED IS SAME AS X
140 IF A=X THEN PRINT "A HIT!!" : S=S+1 : GOTO 210
150 REM ---- NUMBER PRESSED IS NOT X
160 PRINT "A MISS." : S=S-1 : GOTO 210
170 NEXT J
180 REM ---- NO KEY PRESSED IN TIME
190 PRINT "YOU TOOK TOO LONG." : S=S-1
200 REM ---- PRINT SCORE AND DO AGAIN
210 PRINT "YOUR SCORE IS"; S
220 PRINT "PRESS SPACE BAR TO CONTINUE.";
230 IF INKEY$=" " THEN GOTO 20 ELSE GOTO 230
```

Here's how the program works. The variable X represents the segment of the galaxy where the hostile ship will appear. X is determined by using the random number function RND in line 20. In line 30 the numbers from 1 to 5 are printed at the top of the screen. Then the asterisk is printed under the number specified by X. How? By using the SPACE$ variable to print a number of spaces equal to X times 5, and then printing the asterisk following the spaces. (X is multiplied by 5 because the numbers 1 through 5 at the top of the screen are 5 spaces from each other.)

Now the program starts a FOR...NEXT timing loop between lines 70 and 170. The first thing the program does in the loop is to use the INKEY$ function to see if a key has been pressed on the keyboard. (Refer to the discussion of INKEY$ earlier in this section.) If not (A$=" "), then it goes to the NEXT J statement in line 170. This counts as one pass through the loop, leaving 29 passes to go (assuming the loop variable in line 70 is 30). Starting the loop again, the program again checks the keyboard—and again, and again. Each time it checks and finds nothing there it lowers the loop variable J by one.

While all this is going on you are trying to figure out, in your slow human way, what key to press. Let's say you press a key before the timing loop is over. In line 80, A$ is set to the character that you pressed. In line 120 we

convert it from a string character to a number by using the ASC function and then subtracting 48 (since, in the ASCII code, the character "0" (zero) is represented by 48, "1" by 49, and so on). Now, in line 120, we print the plus sign, spacing it over A times 5 spaces (just as we did with the asterisk in line 120).

If A equals X we print "A HIT!!" and increment the score S by one. If A is not equal to X, we print "A MISS" and decrement the score. And if we exit from the loop without any key at all being pressed, we print "YOU TOOK TOO LONG" in line 190 and decrement the score.

No matter which of the above outcomes took place, the program now prints "HIT SPACE BAR TO CONTINUE," and waits for you to digest the results of the last play and hit the space bar. When you do, it starts over, picking a new value of X.

SPC—Another Way to Space Out

SPC(N) means, "Print N spaces." SPC(N) can only be used in a PRINT (or LPRINT) statement.

SPC(N) performs much the same function as SPACE$(N): it saves you the time required to type a row of spaces and also shortens your program. The difference is that SPACE$ (which was described previously) is itself a variable which can be used in such statements as X$=SPACE$(5), while SPC(N), on the other hand, is a function which must be used in a PRINT (or LPRINT) statement. In this respect SPC(N) is like the TAB(N) function: they are both used as formatting aids inside of PRINT statements.

This sample program will print a title with three spaces between each of the words:

```
10 PRINT "WAR"; SPC(3); "AND"; SPC(3); "PEACE"
```

STRING$—Stringing Things Out

STRING$(N,X$) means "This variable is equivalent to a STRING of N characters, each of which is the character represented by X$."

Here's an example of the use of STRING$:

```
10 X$="-"
20 PRINT STRING$(10,X$)
RUN
----------
```

When you RUN this program it prints a string of 10 dashes, because the variable X$ has been set to a dash. You could achieve the same result with this statement: PRINT STRING$(10,"−"). N can be any number from 1 to 255, or it can be a numeric variable (like J) or expression (like J*N+2). X$ can be any string variable, but if it is longer than one character, only the first character will be used in STRING$.

STRING$ is very much like SPACE$ (described above), except that SPACE$ is equivalent to a string of N spaces, while STRING$ is equivalent to a string of N characters, which may be any character. (They must all be the same character, of course.)

Like SPACE$, STRING$ is useful for formatting output on the screen (or on a printer); but it is more versatile. For instance, STRING$ can be used to produce quick graphs of data or mathematical functions.

For example, here's a short program that draws a graph of the square of a number. The square of a number is the number multiplied by itself. The square of 3 is 9, the square of 10 is 100, and so on. Note that CLS means "Clear Screen" and causes the entire contents of the screen to be erased.

```
10 CLS
20 FOR X=1 TO 14 '----USE 14 LINES ON THE SCREEN
30 Y= (X^2) / 4 '----SET Y EQUAL TO SQUARE OF X
40 PRINT X; TAB(7) '----PRINT THE NUMBER X, TAB OUT TO 7
50 PRINT STRING$(Y,"*") '----PRINT Y ASTERISKS
60 NEXT X
70 GOTO 70 '----STAY HERE UNTIL BREAK
```

If you type in this program and RUN it, you'll get a picture on your screen that looks like Fig. 6-2.

```
 1
 2  *
 3  **
 4  ****
 5  *****
 6  *********
 7  ************
 8  ****************
 9  ********************
10  ************************
11  ******************************
12  ************************************
13  ******************************************
14  ************************************************
```

Fig. 6-2. Result of program to draw a graph of the square of a number.

If you're used to reading graphs you'll notice that this one is turned sideways compared with a typical graph. Here, the X-axis starts at the upper left-

hand corner and runs down the left-hand side of the screen (from 1 to 14), and the Y-axis runs from left to right horizontally across the screen. The length of each line of stars (asterisks) represents the value of Y.

But, you may ask, if the square of the number 2 is 4, why aren't there 4 stars beside the 2? And 9 stars beside the 3? The answer is that in order to fit the square of 14, which is 196, on the screen, we had to "scale down" the values of X by dividing them by four. So every line of stars has only one quarter as many stars as the number it represents (rounded off to a whole number of stars, of course, since the computer won't print part of a star).

How does the program make this graph? Most of the program is a FOR...NEXT loop, in which the loop variable X runs from 1 to 14. Each time the program goes through the loop it finds the square of X (which is the same as raising it to the power of 2, or X ^ 2). Then, as we mentioned, we divide the result by four so the stars will fit on the screen. The program then prints the numerical value of X and TABS out 7 spaces so each row of stars will start in the same column. The important part of the program comes in line 60, where STRING$(Y,"*") lets us print a line of exactly Y stars.

When the printing of the graph is done (the FOR...NEXT loop is finished) the program goes into a "pause" in line 70 by continuously executing GOTO 70. This pause keeps the program from printing anything else (like "READY" or "OK") after the graph is printed. Printing something else might cause the whole screen to scroll up, thus losing the top line of the graph.

MID$ (on the left)—Easy String Modification

MID$(A$,P,N) = R$ means, "Replace the MIDdle of string A$ with string R$, for a length of N characters, starting at position P in A$." (This definition only applies when MID$ is on the left of the equal sign. When it's on the right, see the description of MID$ in the String Functions section of Chapter 4.)

Let's illustrate this definition with an example:

```
10 A$="THE CAT ATE THE CHICKENS."
20 MID$(A$,5,3) = "DOG"
30 PRINT A$
RUN
THE DOG ATE THE CHICKENS
```

We start off with a string A$: "THE CAT ATE THE CHICKENS," and in line 20 we use MID$ to change the word CAT to DOG. So when we RUN the program, the modified version of A$ will be printed out in line 30. In line 20 the A$ (in the parentheses following the MID$) means that A$ is the string to be modified. The 5 means that the modification will start at the 5th character position in A$, since, in this example, the letter "C" in "CAT" is 5 letters from the start of the string A$. (The first character in the sentence, "T," is in posi-

tion 1.) The last item in the parentheses, the 3, means that 3 characters will be replaced. The string on the right of the equal sign, "DOG," is the new string that will be placed in A$.

Don't confuse MID$ when it is used on the left of the equal sign with MID$ when it is used on the right. As you learned in Chapter 4, MID$ used on the right of the equal sign is a way of creating a new string variable which consists of the middle of the old one. For example, if A$="THE CAT ATE THE CHICKENS," and you execute the statement B$=MID$(A$,5,3), then B$ will be equal to the string "CAT" (which starts at the 5th character of A$ and is 3 characters long), but the string A$ will not be changed at all.

When would you want to use MID$ on the left of the equal sign? Imagine this situation: You have used your computer to write a long letter to someone named JANE. This letter might be stored in your computer's memory, or it might be stored on tape or disk. In this example the way it is stored doesn't matter, since we will only examine a small part of the letter. Let's assume the sentences that make up the letter are in the computer's memory in the form of string variables. After you have written the letter, you decide that you want to also send the same letter to someone named MARY. (Perhaps there's more than one lady in your life.) Unfortunately, the name JANE appears many times in the letter, so that it will be tedious and time-consuming to change it every time to the name MARY. You decide to write a program which will do the changes automatically. (This will not only permit you to change JANE to MARY, but also to ANNE, or LISA, or whatever other lady you are thinking about at the moment.)

To show you how this program works we have added a line at the beginning of the "change the name" routine which will read a string variable L$ directly from the keyboard, and another line at the end to print out the result; but if you were really modifying a long letter as we describe, there would of course be many strings to modify. These would have to be assigned to the string L$ one at a time to be modified. For simplicity we don't show this part of the routine.

What the following program does is to let you type in a sentence in which the name JANE can appear as often as you want. Then the program looks through the sentence, which has been assigned to the string variable L$, and changes every occurrence of the name JANE to MARY.

```
10 LINE INPUT "TYPE INPUT SENTENCE"; L$
20 P=INSTR(L$,"JANE")
30 IF P<>0 THEN MID$(L$,P,4) = "MARY" : GOTO 20
40 PRINT L$ : GOTO 10
RUN
SWEETEST JANE, MOST LOVELY JANE, YOU ARE THE ONLY ONE FOR ME.
SWEETEST MARY, MOST LOVELY MARY, YOU ARE THE ONLY ONE FOR ME.
```

That's a short routine to do so much, isn't it? It can be so short because INSTR (see the description of INSTR earlier in this section) and MID$ are such powerful BASIC statements.

Let's find out how it works. After reading in the sentence L$, we use the INSTR statement to see if the sentence contains the word "JANE." If "JANE" is found, then P is set to the place where the match begins, and we go to line 20, which uses the MID$ statement to replace the word "JANE" with the word "MARY," starting at position P (where "JANE" begins) and replacing 4 letters. Then we go back to line 20. Here we execute the INSTR statement again, but this time, since we have changed the first "JANE" to "MARY," we won't find that particular "JANE" again. If, however, "JANE" occurs more than once, it will find the next occurrence. The MID$ in line 30 will then change this "JANE" to "MARY" and go back to line 20 to search again. The process will continue until all the "JANEs" in L$ have been changed to "MARYs", at which point the INSTR statement will return P=0. When P=0 the MID$ statement will not be executed, and the program will continue to line 40 to print out the modified version of L$.

Of course, you must be careful using this program: if JANE and MARY ever compare their letters, you could be in big trouble!

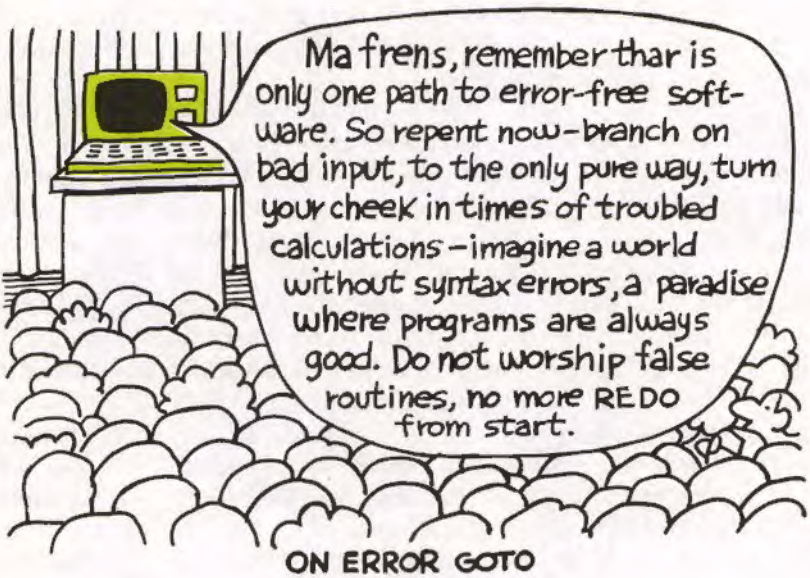
DEALING WITH ERRORS—CREATIVELY

Until now you may have assumed that if an error occurred in your program BASIC would, as inevitably as death and taxes, exit from the program, print out an error message (like "SYNTAX ERROR"), and revert to the BASIC monitor mode with a "READY" or "OK" message. This is what BASIC normally does. The process works all right while you are developing and debugging a program, since you get immediate feedback on any mistakes you've made, and can then make the needed corrections. But once the program is finished, and in the hands of a user, error messages are not so useful. The last thing you want the user to have to deal with is the sudden appearance of a phrase like "OVERFLOW ERROR IN LINE 290" on the screen, followed by the program "bombing" (ceasing to operate, and returning to the monitor mode).

Some errors are not caused by the programmer, but may be the result of bad input on the part of the user, so that even a correctly written program is no assurance that errors will not occur when the program is run. Is there some way that we can keep errors from "bombing" the program? The answer to that question is the subject of this section.

ON ERROR GOTO and RESUME—Confronting Your Errors

ON ERROR GOTO N means "ON the occurrence of any kind of ERROR, GOTO line number N and execute the routine there, without causing an error message."



RESUME M means "The error has been handled, so RESUME the regular program, at line number M."

ON ERROR GOTO and RESUME, taken together, provide a way for the programs you write to take over the job of processing errors from the BASIC monitor mode. ON ERROR GOTO provides a way to transfer control from your program to the routine you write to handle errors, and RESUME provides a way to get back to the program.

As an example, we'll look again at the program for finding factorials that we introduced in the WHILE...WEND section. Here it is.

```
10 INPUT"NUMBER"; N
20 F=1
30 F=F*N
40 N=N-1
50 IF N>0 THEN GOTO 30
60 PRINT "FACTORIAL IS"; F
70 GOTO 10
```

We'll use the version of the program with an IF statement and a GOTO, rather than the WHILE...WEND loop, so that even if your computer doesn't have the WHILE...WEND function you can still try out the example.

If you typed in this program earlier and experimented with it, you probably found that it worked fine for comparatively small values of N: say up to about 33. (The exact number will vary from one computer to another.) When you tried to input a larger number, 34 for example, the program "bombed,"

BASIC printed "OV ERROR" or "OVERFLOW IN 30" or some similar error message, and then reverted to the monitor mode with a "READY" or "OK" message. (In some versions of BASIC an error only occurs here when the variables F and N are integers. For more information on the difference between integers and other variables, see the section on "Variable Types" later in this chapter.)

Here's how the results of running the factorial program might have looked if you had typed in some large numbers:

```
NUMBER? 32
FACTORIAL IS 2.63131E+35
NUMBER? 33
FACTORIAL IS 8.68332E+36
NUMBER? 34
OVERFLOW IN 30
READY
```

What's going on here? Why do we get an error message? Well, the problem is that your computer can only handle numbers up to a certain size, and the factorial of a number like 34 is a really huge number—in fact, it is 37 digits long! When numbers get this large, BASIC says "I give up, it's too big for me, help!" That's what the "OVERFLOW" error message means: that a number was too large for BASIC to handle.

So, if you used this factorial routine in a program that someone else was going to use, then every time they attempted to type in too large a number, BASIC would print the error message and bomb the program—and the user might wish that he had used someone else's program instead of yours.

Is there a way to avoid having the program bomb when the numbers get too big? There certainly is: by using ON ERROR GOTO and RESUME. Let's modify the program like this:

```
  5 ON ERROR GOTO 100
 10 INPUT"NUMBER"; N
 20 F=1
 30 F=F*N
 40 N=N-1
 50 IF N>0 THEN GOTO 30
 60 PRINT"FACTORIAL IS"; F
 70 GOTO 10
100 REM ---- ERROR-CATCHING ROUTINE
110 PRINT"TOO LARGE A NUMBER TO COMPUTE FACTORIAL."
120 RESUME 10
```

As you see, we've inserted a new line 5, "ON ERROR GOTO 100." This line tells BASIC, "Don't do anything now, but if an error occurs in the pro-

gram, don't print the message and bomb the program; just go to line 100 and start executing the instructions there." It's important to realize that this line has no immediate effect on the program—in fact, it has no effect at all unless an error occurs.

We've also added the new lines 100 to 120. These lines deal with any error that occurs. (If no errors occur, this routine will never be executed.) This kind of routine is often called an "error-trapping" routine, since its purpose is to "trap" errors before they can do any harm. In this particular case, the only thing that the routine does—if an error occurs and control passes to the routine—is to print a message "TOO LARGE A NUMBER TO COMPUTE FACTORIAL." This tells the user of the program that he can't expect the computer to handle the factorial of the number N that he typed in: it's too large.

Then, to return to the main program, the error-trapping routine executes the RESUME 10 instruction in line 120, to return control to line 10. RESUME not only takes you back to the main program, it also has the effect of "resetting" the ON ERROR GOTO statement, so that when the next error is detected control will pass again to the error-trapping routine. If you returned to the main program without using RESUME (with a GOTO, for example), then when the next error occurred BASIC would handle it in the "normal" way, by taking over, bombing the program, and printing an error message.

RESUME can be followed—as it is here—by the line number in the main program to which you want control to return from the error-trapping routine. You can also say "RETURN 0," or simply "RETURN." These statements will cause BASIC to go back and resume execution of the program at the same line in which the error occurred. If you do this, you have to be sure the error has been corrected before you execute the RESUME. Otherwise, control will go back to the line with the error in it, and since the error is still there, BASIC will head right back to the error-trapping routine, creating an endless loop. (You could avoid this, for example, by having the error-trapping routine reset N to the largest value BASIC can handle.) You can also say "RESUME NEXT," which will cause control to resume at the line following the line in which the error occurred.

Try out the program. It works fine, doesn't it? No matter how large a number you type in, the program answers it with "TOO LARGE A NUMBER TO COMPUTE FACTORIAL," and goes right back to ask "NUMBER?" again.

There are, however, some possible pitfalls in the use of ON ERROR GOTO, things that could cause trouble if you're not aware of them.

One such pitfall is this: once an error occurs, and you're in the error-trapping routine, you shouldn't let another error happen before you go back to the main program with the RESUME statement. If you do, the error-trapping routine will not trap it—control will return to BASIC which will bomb the program and print out its own error message in the usual way.

Another possible pitfall is that once you've executed the ON ERROR GOTO statement, then any error, no matter what it is or where it is in the program, will cause control to go to the error-trapping routine. This is true

even of errors you make in direct statements in the monitor mode. For instance, if you typed LISP instead of LIST to list your program, control would go immediately to your error-trapping routine, even though your program wasn't even running at the time. The best way to avoid these problems is to execute the statement "ON ERROR GOTO 0" when you no longer expect the particular error that your error routine is intended to trap. This statement has the effect of cancelling your original ON ERROR GOTO N statement, so that errors will cause the normal BASIC error messages instead of going to your routine. We'll give an example of how to do this in the next section, on ERR and ERL.

ERR and ERL—What Was the Error, and Where Did it Happen?

ERR means "What was the number of the error?"
ERL means "On what line number did the error occur?"

ERR and ERL have similar functions, so we will describe them together here. They are both typically used in an error-trapping routine to give the routine information about the error. (For a discussion of error-trapping, see the last section, ON ERROR GOTO and RESUME.) Often, they are used simply to ensure that the error which occurred is the one you expected (the one the error-trapping routine is designed to deal with) and not a more serious unexpected error.

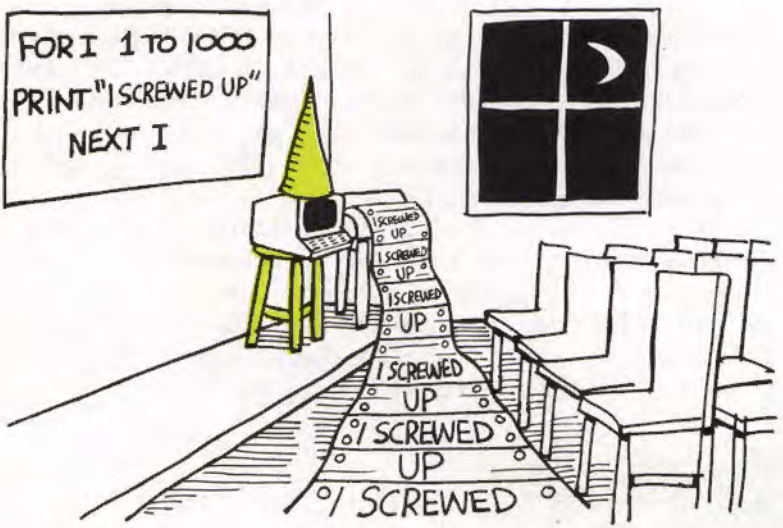
To create such an "unexpected" error, try making a typing mistake in one of the lines of the factorial program described in the last section. For example, change the word INPUT in line 20 to INPIT. Now RUN the program. What happens? You would expect BASIC to recognize the error, halt execution of the program, print "SYNTAX ERROR," and then wait in the command mode for you to tell it what to do. But no—it prints out "TOO LARGE A NUMBER." Why? Because in line 5 you told it, "ON ERROR GOTO 100," and that's what it does faithfully—no matter what the error is—until you tell it to do otherwise. And while this is what you want if the error is really caused by inputting too large a number, it isn't very useful if the error is something else, like writing INPIT instead of INPUT.

Let's change the error-trapping routine so that it is a little more discriminating—that is, so that it can respond differently to different errors. We'll have it not only check that the error is the same kind of error that we expected (that is, the same error number), we'll also have it check that the error took place on the same program line that we expected it to. To do this, we add line 105, making the program look like this:

```
 5 ON ERROR GOTO 100
10 INPUT"NUMBER"; N
20 F=1
30 F=F*N
```

```
 40 N=N-1
 50 IF N>0 THEN GOTO 30
 60 PRINT"FACTORIAL IS"; F
 70 GOTO 10
100 REM ---- ERROR-TRAPPING ROUTINE
105 IF ERR <>6 OR ERL <>30 THEN ON ERROR GOTO 0
110 PRINT"TOO LARGE A NUMBER."
120 RESUME 10
```

Now, when an error occurs, the error-trapping routine first checks to see if ERR (the error number) is equal to 6, which is the number of the OVERFLOW error. (This number may be different on different computers. You can check the BASIC manual for your computer to see what number to use.) Then, it checks to see if ERL (the line number where the error occurred) is equal to 30, which is the line number where we expect the overflow error to occur (since this is where the factorials—which may cause overflow—are generated by multiplication). If either of these conditions is not true, then the routine knows that an unanticipated error has taken place, one which the routine cannot deal with itself. At this point, in order to turn control back over to the BASIC monitor so that the proper error message can be printed out, the routine executes an "ON ERROR GOTO 0" statement. This has the effect of cancelling the "ON ERROR GOTO 100" statement in line 5, and control then goes back to the BASIC monitor, which prints the appropriate error message. Notice that control does *not* go to line number 0.



ERROR

ERROR—Stirring Up Trouble

ERROR N means, "Make BASIC think that ERROR number N has occurred."

This statement has the effect of imitating (simulating) an error, so that BASIC thinks one has taken place, when in fact no error has occurred. Now why would we want BASIC to think that an error had taken place? The usual reason is that we want to test an error-trapping routine. (See the preceding discussion of ON ERROR GOTO, RESUME, and ERR and ERL.) Let's try out this idea, using the Factorial program with the error-trapping routine from the last section. We'll make only one change in this program: we'll leave off the part of line 105 that says "OR ERL < > 30." The reason for this will be explained later.

```
 5 ON ERROR GOTO 100
10 INPUT"NUMBER"; N
20 F=1
30 F=F*N
40 N=N-1
50 IF N>0 THEN GOTO 30
60 PRINT"FACTORIAL IS"; F
70 GOTO 10
100 REM ---- ERROR-TRAPPING ROUTINE
105 IF ERROR<>6 THEN ON ERROR GOTO 0
110 PRINT"TOO LARGE A NUMBER TO COMPUTE FACTORIAL."
120 RESUME 10
```

RUN the program. When you type in a number which is too large, the program will "trap" to the error-handling routine in line 100 and print "TOO LARGE A NUMBER." Now, instead of actually causing an error by inputting too large a number, let's just "imitate" an error. Break into the program by typing "break" (or "control-C" or "control-break") so that BASIC halts execution of the program and returns to the monitor mode, printing "READY" or "OK."

Now type "ERROR 6" (or whatever number is the overflow error on your computer). Lo and behold, the error-trapping routine is executed in exactly the same way as it was when we typed in too large a number. So we have checked out the operation of the error-trapping routine without having a real error take place. Of course, in this example it's easy to generate a real error by typing too large a number, so that the ERROR statement isn't essential to checking out the error-trapping routine. But some errors are difficult to cause when you want them—those relating to malfunction of a tape or disk unit, for example—and in these cases ERROR is a very useful function.

Notice that the reason we had to delete the "OR ERL < > 30" part of line

105 is that, if we imitate the error in command mode by typing "ERROR 6" as explained above, then the error is no longer taking place in line 30. So, if the routine checks to see that it did happen in line 30 and finds that it didn't, it will come to the mistaken conclusion that an unanticipated error has occurred and will execute the ON ERROR GOTO 0 part of the statement, which will cause BASIC to print "OVERFLOW" instead of the "TOO LARGE A NUMBER" message that we want.

ERROR Used as an Extended Form of GOSUB

ERROR can also be used in another way, one the designers of BASIC probably didn't think of. In the following program we will use ERROR not to imitate an error, but almost as if it were a normal GOSUB statement. The advantage of using ERROR Instead of GOSUB is that when an ERROR statement is executed, ERR is automatically set to the error number N (specified by the number N in "ERROR N"), and ERL is set to the line number where the error occurred. It can be very useful for a subroutine to be able to figure out the line number from which it was called, and it can also be useful to be able to pass an "argument" (a number with different values) to the subroutine from the program which calls it.

Here's a program which shows how this might be done. The program asks for the user's name and age. If the name is longer than 15 characters it prints "PLEASE USE A SHORTER NAME." If the age is less than 3 years or greater than 80, it assumes the user has made a nonserious reply and prints "PLEASE BE HONEST." (If you really are more than 80 or less than 3 years old, you can change line 50 accordingly!)

```
 10 ON ERROR GOTO 80
 20 INPUT"PLEASE ENTER YOUR LAST NAME"; LN$
 30 IF LEN(LN$) > 15 THEN ERROR 38
 40 INPUT"AND YOUR AGE"; AG
 50 IF AG<3 OR AG>80 THEN ERROR 38
 60 PRINT"THANK YOU."
 70 STOP
 80 REM ---- ERROR-TRAPPING ROUTINE
 90 IF ERR <> 38 THEN ON ERROR GOTO 0      'UNEXPECTED ERROR
100 IF ERL=30 THEN PRINT"USE A SHORTER NAME." : RESUME 20
110 IF ERL=50 THEN PRINT "PLEASE BE HONEST." : RESUME 40
120 ON ERROR GOTO 0     'UNEXPECTED LINE NUMBER
```

In this program we use the ERROR statement to branch to the "error-trapping" routine at line 80 if either the name typed in is too long, or the age typed in is less than 3 or more than 80. Neither of these conditions is really an "error," of course. We are simply using the error-handling statements to permit the routine to figure out which part of the main program called the routine.

Let's go through this step by step for the case of the user typing in an inappropriate age: 120, for example.

The first line of the program makes BASIC aware that, in the event of an error, control should go to line 80. Then the program asks for the user's last name and his age. If, when he is asked his age, the user types (for example) 120, then in line 50 the program will see that AG is greater than 80 and will go on to the next part of the statement, which is ERROR 38. (Actually, any valid error number could be used here.) Executing this statement takes us immediately to the error-trapping routine on line 90. There, on line 90, we first verify that the error is the one we expect (that is, number 38). If it isn't, we assume a "real" error (that is, one we hadn't anticipated, such as a syntax error) has taken place, and execute the ON ERROR GOTO 0 statement so that control will return to BASIC to deal with the error in the normal way. However, if the error is number 38, we then go on to check what line number it occurred on using the ERL function. In this case, it occurred on line number 50, so we will execute the second part of line 110, which is

```
PRINT"PLEASE BE HONEST." : RESUME 40
```

Control will then go back to line 40, where the user will again be asked his age. If the error routine has not been called from either of the expected line numbers, 30 or 50, then the routine goes on to execute the "ON ERROR GOTO 0" statement in line 120, which will return control to BASIC to deal with a normal error.

When To Use an Error-Trapping Routine

To summarize this section, we can say that you probably won't need to use any of the error-trapping terms we have been discussing when you are writing and debugging your program. In fact, in these stages of program development, error-trapping routines might even be a disadvantage, since you want BASIC to tell you about any errors in your program.

However, to make a finished program as "user-proof" as possible, it can often be an advantage to include an error-trapping routine to deal with all the errors which you can imagine being made by users of the program. No matter what the user of your program does, it is almost always better to have the program keep running and deal with the error itself, rather than bombing and turning control back over to BASIC. This way you can print the kind of error message you want to, rather than letting a BASIC error message reveal to the user that you have failed to foresee something the program or the user might do.

And if your error-trapping routine can't figure out what sort of error the user made, it can always print out some sort of outrageous general-purpose error message, like "WE REGRET TO INFORM YOU THAT YOU HAVE VIOLATED SECTION 6SJ7 OF THE COMPUTER USER'S GUIDE. CONSIDERING THE

DEGREE OF YOUR INCOMPETENCE WE HAVE NO CHOICE BUT TO TERMINATE THIS PROGRAM. GOODBYE!"

Of course there is no "SECTION 6SJ7" (and no "COMPUTER USER'S GUIDE" either, for that matter). But perhaps the user of your program will be so demoralized by this message that he will fail to notice that you have not anticipated whatever error it was that caused the message! (Actually we're just kidding about this. In fact, it might not be such a good idea for your program to completely take over the error-handling process, since then in the event of a malfunction in the user's computer he would have no way to find out about it—all he'd get would be your error message!)

VARIABLE TYPES—INTEGERS, SINGLE AND DOUBLE PRECISION, STRINGS

In Chapter 1 you learned the difference between string variables (like A$) and numeric variables (like A). You learned that the names of string variables are followed by a dollar sign, and that the value of a string variable is always a string of alphanumeric characters (such as "CAT" or "12 O'CLOCK"); and you learned that a numeric variable is not followed by a dollar sign and that it's value is a numeric quantity like 2 or 10 or 6.123. In the program examples we have used so far you have not needed to know more than these two kinds of variables, numeric and string. However, the BASIC interpreter in your computer has been making at least one other distinction: it has been dividing numeric variables into two types, integers and single-precision variables. Let's explain the difference between these two kinds of variables.

Briefly, *an integer is a number with no digits to the right of the decimal point.* That is, it is a whole number, with no fractional part. Thus, 10 and 2 and −9 and 4096 are integers, while 4.7 and 2/3 and 0.002 are not. And so, not surprisingly, an integer variable is one whose value is an integer. On most personal computers there is another restriction: the value of an integer variable must not be any larger than 32767 nor any smaller than −32768.

What are single-precision variables? Here the answer depends to some extent on which computer you are using. On a typical personal computer a *single-precision variable is one whose value consists of 6 or 7 digits (with the decimal point anywhere) and whose range (size) is between about 1E+38 and 1E−38.* (These two numbers are written in exponential notation. If writing numbers this way is new to you, turn to Appendix F, "Exponential Notation.")

It's easy to see how many digits your version of BASIC is working with. Just type this line, in command mode:

```
PRINT 1/3
```

The computer will respond with a string of 3s:

```
.333333
```

Since 1 divided by 3 produces an infinite string of 3s following the decimal point, we can see at once just how many digits BASIC is working with just by counting the number of 3s in this example. In this case, we can conclude that our computer's single-precision routines are using 6 digits (if seven 3s had been printed out, our computer would be using seven digits, and so on).

Let's write a simple program to demonstrate the definition of single-precision variables.

```
10 J=2.3456789
20 PRINT J,
30 J = J * 10
40 GOTO 20
```

We start with a number J which is 8 digits long. We print it, multiply it by 10, print the result, multiply the result by 10, and so on and so on. If you type in this program and RUN it, your computer should print out something like that seen in Fig. 6-3.

```
READY
 RUN
 2.34568          23.4568          234.568          2345.68
 23456.8          234568           2.34568E+06      2.34568E+07
 2.34568E+08      2.34568E+09      2.34568E+10      2.34568E+11
 2.34568E+12      2.34568E+13      2.34568E+14      2.34568E+15
 2.34568E+16      2.34568E+17      2.34568E+18      2.34568E+19
 2.34568E+20      2.34568E+21      2.34568E+22      2.34568E+23
 2.34568E+24      2.34568E+25      2.34568E+26      2.34568E+27
 2.34568E+28      2.34568E+29      2.34568E+30      2.34568E+31
 2.34568E+32      2.34568E+33      2.34568E+34      2.34568E+35
 2.34568E+36      2.34568E+37
OVERFLOW IN 30
READY
> -
```

Fig. 6-3. Large numbers in exponential notation.

The first thing to notice about Fig. 6-3 is the first number the computer printed out: 2.34568, instead of the 2.3456789 that we entered. Our variable J has been rounded off so it is only 6 digits long, instead of the 8 we typed in. What happened to the other 2 digits? Well, unless someone tells it otherwise, your computer always assumes that the variables and the numbers you use in

your programs are single precision. And since a single-precision number can only have 6 digits (at least in the computer used for this example), two are chopped off. Actually, there's no use in even typing in more than 6 digits—anything more than that will be lost.

The second thing to notice about this output is that once the numbers reach a certain size, they are printed out in exponential notation. Actually, BASIC stores single-precision variables in exponential form no matter what size their values are. It prints out the smaller numbers in the "normal" way (as 4123 instead of 4.123E+3, for example) merely as a convenience to the human using the computer.

In exponential notation, part of the variable is used for storing the mantissa (the six digits written before the E) and part is used to store the exponent (the two-digit number following the E). If the mantissa gets too big, some of the digits are just chopped off as we saw, but if the exponent gets too big, the computer can't store the number at all and has to print an error message. That's what's happening in the printout: when the program tries to multiply 2.34568E+37 by 10, the error message "OVERFLOW IN 30" results and the program stops. Why? Because the exponent of a single-precision value cannot be any larger than about 37. (Actually, it's between 37 and 38. This depends on your computer. Remember, however, that when a number's exponent is 38, then the number is 38 digits long! Since such large numbers are not often encountered, the fact that the computer can't deal with them is not such a serious limitation.) Here's what 2.34568E+38 looks like written out:

```
234,568,000,000,000,000,000,000,000,000,000,000,000
```

Here's another program. This one generates numbers that get smaller and smaller, since we divide by 10 each time we print out the number (instead of multiplying, as in the example above). We have had to add a line to STOP the program (line 30) when BASIC starts to think that J is zero. This is because BASIC, instead of causing an error message when the exponent goes from −38 to −39, simply decides that a number that small is the same as zero and would continue to print out zeros from then until doomsday if we didn't stop the program.

```
10 J=2.75
20 PRINT J,
30 IF J=0 THEN STOP
40 J=J/10
50 GOTO 20
```

Fig. 6-4 shows the output from the program.

```
READY
>RUN
 2.75          .275          .0275         2.75E-03
 2.75E-04      2.75E-05      2.75E-06      2.75E-07
 2.75E-08      2.75E-09      2.75E-10      2.75E-11
 2.75E-12      2.75E-13      2.75E-14      2.75E-15
 2.75E-16      2.75E-17      2.75E-18      2.75E-19
 2.75E-20      2.75E-21      2.75E-22      2.75E-23
 2.75E-24      2.75E-25      2.75E-26      2.75E-27
 2.75E-28      2.75E-29      2.75E-30      2.75E-31
 2.75E-32      2.75E-33      2.75E-34      2.75E-35
 2.75E-36      2.75E-37      2.75E-38      0
BREAK IN 30
READY

> -
```

Fig. 6-4. Small numbers in exponential notation.

Here's what 2.75E-38 looks like written out:

0.000,000,000,000,000,000,000,000,000,000,000,000,0275

Let's summarize the differences between an integer variable and a single-precision variable. An integer variable must lie in the range between +32767 and −32768 and must be a whole number, with no fractional part (that is, no digits to the right of the decimal point). A single precision variable, on the other hand, can have any value between about 1E+37 and 1E−38, but can have only six or seven digits.

When Should You Use Integers, and When Single Precision?

As you have probably guessed, integer and single-precision variables are useful for different kinds of things. Integers are good for counting small numbers of discrete items such as how many oranges in a sack or how many months in a year—things that have no fractional part. Single-precision numbers are good for measuring continuous quantities that are not necessarily whole numbers, like how fast your car is traveling or how long a table is.

Why not always use single precision? After all, any number that can be expressed as an integer can also be expressed as single precision, (The reverse is not, of course, true.) Integers have two advantages—they're small, and they're fast.

Integers are small in the sense that they take up less room in your computer's memory than single-precision variables do. An integer value is stored in two bytes, while a single-precision value requires 4 or 5 bytes (or more), depending on your particular computer. So if you have a large program and not much memory space in your computer, it pays to use integer variables

wherever possible. (For a further description of how computers store variables read Appendix C, "When Programs Get Too Big.")

The second reason to prefer integers over single-precision variables is speed. Say you have a program with a FOR. . .NEXT loop in it, like this:

```
10 FOR J=1 TO 2000
  .
  .
(other program lines here)
  .
  .
100 NEXT J
```

If you don't tell it otherwise, BASIC assumes that all variables are single precision, including the J in the FOR. . .NEXT loop. But, when BASIC is actually executing a FOR. . .NEXT loop, it needs J to be an integer so it can use it for counting. So every time the FOR statement is executed (2000 times in the example) BASIC changes J from a single-precision variable to an integer variable, increments the loop, and then changes it back so it is available to other statements inside the loop as a single-precision variable. Naturally all this changing back and forth takes time. So, if you want your FOR. . .NEXT loops to run as fast as possible, use integer variables.

BASIC also performs arithmetic faster on integers than it does on single-precision variables. Actually, your BASIC interpreter has a complete set of routines for doing arithmetic with integers (add, subtract, multiply and divide) and another set of routines for doing arithmetic on single-precision variables. (There is a third set, too, for double precision, which we will discuss later.) Since integer variables are only two bytes long, instead of four or five, the routines to do arithmetic on them are much faster than the corresponding routines for single precision. So, if you have a lot of arithmetic to do, it pays to speed it up by using integers, provided of course that the numbers you are operating on lie between 32767 and −32768.

But how do we get a variable to be an integer, if BASIC assumes all numeric variables are single precision? We'll find that out in the next section.

Type Declaration Characters

Type declaration characters are special symbols which are written immediately after a variable name to specify which of four types the variable is. These four types, and the character that specifies each one, are:

- *Integer, indicated by a percent sign (%).*
  Examples: A%, Z3%.

- *Single Precision, indicated by an exclamation point* (!).
  Examples: B!, Q4!.

- *Double Precision, indicated by a number sign* (#).
  Examples: D#, P9#.

- *String, indicated by a dollar sign* ($).
  Examples: F$, S7$.

In the last section we talked about the advantages of making variables be integers in some cases, and single-precision variables in others. But how do we force a variable to be one of these types? One way is with "Type Declaration Characters." This long name simply means that we add one of the special characters listed above to the variable name to force the variable to be a particular type. You already learned to use the type declaration character for strings in Chapter 1: it's a dollar sign. So you know that while A1 is a numeric variable, A1$ is a string variable.

Integer and single-precision variables also have type declaration characters. (Let's abbreviate them to T.D.C.s from now on.) There is also a fourth T.D.C., for double-precision variables, which we will cover in the next section. The T.D.C. for integers is a percent sign (%), and the T.D.C. for single-precision variables is an exclamation point (!). If you want to specify the type a numeric variable is going to be (integer or single-precision), all you have to do is add the appropriate T.D.C. to the name every time you write it. Thus, A% and B2% are integer variables, and A! and B2! are single-precision variables. The T.D.C. actually becomes part of the name and can be used to distinguish two variables which would otherwise have the same name. Thus, H2% and H2! are completely different variables and can both be used in the same program. BASIC will keep them separate.

As an example of how to use T.D.C.s, and also as a demonstration of the advantages of using integers in certain situations, let's look at the following loop:

```
10 FOR J=1 TO 2000
20 NEXT J
```

Because BASIC assumes that every variable is single precision unless told specifically otherwise, it will treat the loop variable J as single precision. But because it needs J to be an integer to use it as a loop variable, it will convert it to an integer each time it needs to increment it in the loop. This takes time, and makes the program run slower. To see just how much slower, do this: run the program, and time it. If you have a stopwatch you can time it very accurately, but the second hand on your watch will do. Write down how long it took. (On our computer it took 10 seconds.)

Now, change the program like this:

```
10 FOR J%=1 TO 2000
20 NEXT J%
```

We've changed J to an integer variable by adding the percent sign. Now try running the program and timing it. It should be substantially faster than the example where J is a single-precision variable. (On our computer it took only 7 seconds.)

Since variables are single precision unless otherwise specified, we don't often have to use the exclamation point (!) T.D.C. to force a variable to be single precision. Sometimes it is useful though, as we'll see in the section coming up, on DEFINT, DEFSNG, etc.

Double-Precision Variables—When Six Digits Are Not Enough

Double Precision is a type of BASIC variable which can represent numbers with more significant digits than can integer or single-precision variables.

What do you do if you need to write a program which deals with numbers that have more significant digits than the six or seven digits that single-precision variables provide? For example, suppose you're in real estate and you've just earned a five percent commission on a house you sold for $1,234,567.75. (The house is in Beverly Hills.) You need to know to the penny just what your commission is. Write the following program:

```
10 P=.05
20 SP=1234567.75
30 C=P * SP
40 PRINT C
```

Here, P is the percentage of your commission, SP is the selling price, and C is the commission you are trying to find. When you RUN the program you will get something like $61728.4. (This may be slightly different on different computers.) Well, this is a lot of money, but it is the wrong answer. In fact, the program doesn't even print out the last digit (the penny's digit).

How can we get a completely accurate answer? BASIC has another variable type we haven't used yet, called "double precision." As the name implies, double precision gives us more digits in our answer than does single precision (actually more than twice as many).

To see the difference between single and double precision, write the following little program:

```
10 A=1
20 B=3
30 PRINT A/B
```

If you run this program you'll get an answer like .333333. As you can see, this answer is correct to six digits. (This is similar to the example in the description of single precision in the last section.)

Now let's change this program to double precision. The type declaration character (T.D.C.) for double precision is the number sign (#), so we'll rewrite the program this way:

```
10 A#=1
20 B#=3
30 PRINT A#/B#
```

RUN that, and what do you get?

```
.3333333333333333
```

Sixteen digits! That's a lot of precision. We could handle the national debt, to the nearest penny, with that many digits.

Now we're ready to figure the commission in our real estate problem. We'll modify the program this way:

```
10 P#=.05
20 SP#=1234567.75
30 C#=P# * SP#
40 PRINT C#
RUN
61728.38841982465
```

We can easily round off this number to $61728.39. Now our answer is correct to the nearest penny—in fact, we had a lot more precision than we needed for this particular problem.

Double precision is most used in financial programs involving large amounts of money, and in scientific and engineering problems. Even if you don't need sixteen digits of precision too often, it's nice to know they're there when you need them.

Naturally, there is a price to pay for using double precision. Just as single-precision variables are both slower and take up more room in the computer's memory than do integer variables, so double precision is slower and takes up

more space than single precision. It requires eight bytes to store the value of a double-precision variable, twice that of a single-precision variable, which is itself twice as large as an integer. So, if you want to save memory space and computing time, don't use double precision unless it's absolutely necessary.

Fig. 6-5 presents a summary of the four variable types.

Variable Type	Type Declaration Character	Example of Variable	Example of Values	Range of Values	Bytes of Memory Used by Variable
STRING	$	A$ B2$ ZZ$	"CAT" "CHICKEN" "12 P.M." "4.234"	strings from 0 to 255 chars	1 byte per character, plus 6 bytes for name.
INTEGER	%	A% B2% ZZ%	12 30000 −207 0	from −32768 to 32767	2 bytes for value, plus 3 bytes for name
SINGLE PRECISION	!	A! B2! ZZ!	12.45 999999.9 3.2E+19 −17.4	mantissa: 6 digits. exponent: −38 to +38	4 bytes for value, plus 3 bytes for name
DOUBLE PRECISION	#	A# B2# ZZ#	12.45 7.7777777777777 1.24932434342E-9 −0.234567892332	mantissa: 16 digits. exponent: −38 to +38	8 bytes for value, plus 3 bytes for name

Fig. 6-5. Summary of BASIC variable types.

DEFINT, DEFSNG, DEFDBL and DEFTR—Defining Variable Types

DEFINT L means, "Define all variables beginning with the letter L to be integer variables."

DEFSNG L means, "Define all variables beginning with the letter L to be single-precision variables."

DEFDBL L means, "Define all variables beginning with the letter L to be double-precision variables."

DEFSTR L means, "Define all variables beginning with the letter L to be string variables."

The type declaration characters discussed in the last section accomplish their purpose of specifying which type a variable will be, but they also add an extra character to the variable name. So, if you are typing in a long program, you may find it inconvenient and time-consuming to be constantly inserting dollar signs, percent signs, and so on into your listing. In many programs, for example, almost all the variables are defined as integers, which means a lot of percent signs to type.

Fortunately, BASIC comes to our rescue with a clever labor-saving device: the type declaration definitions "DEF---" (here DEF--- stands for DEFINT, DEFSNG, DEFDBL or DEFSTR). These statements let you define—in one

statement—a whole group of variables to be of a particular type (integer, single precision, etc.). Once defined in this way you no longer have to use the type declaration characters (!,%, #, and #) after each instance of the variable name.

For instance, DEFINT N means that all variables whose names start with the letter N are now defined to be integer variables. So every time you write such a variable name in your program, like N, or N2, or NZ, BASIC will assume it is an integer. Similarly, DEFDBL Z defines all variables beginning with the letter Z to be double-precision variables, and DEFSTR Q defines all variables beginning with the letter Q to be strings. DEFSNG S defines all variables beginning with the letter S to be single precision, but since BASIC assumes a variable is single precision unless told otherwise, it isn't usually necessary to use a DEFSNG statement. (We say the "default" variable type is single precision.)

You can also use a list of letters or a range of letters in a DEF statement. For example, DEFDBL A,G,P defines all variables starting with the letters A or G or P to be double precision, while DEFINT I-N defines all variables whose names start with I, J, K, L, M or N to be integers. (In the FORTRAN programming language the variables I, J, K, L, M and N all default to integer variables. Some BASIC programmers follow this tradition by starting their programs with DEFINT I-N).

Another common statement near the beginning of programs that use mostly integer variables is DEFINT A-Z. This defines all variables in the program to be integer, no matter what letter they begin with. Using this statement ensures that all FOR...NEXT loops will use integer variables, and that all arithmetic will be done with integer variables. Since using integers cuts down on memory space and increases speed, this statement buys you a lot of benefit for very little effort.

But what happens if you want to use, say, a string variable in a program after you have already used the DEFINT A-Z statement? Easy: just use a type declaration character after the string variable name.Type declaration characters always take precedence over DEF--- statements; that is, no matter how you have defined a variable with a DEF--- statement, as soon as you use "!" or "#" or "$" or "%" after the name, BASIC will override the DEF--- statement and assign the variable to whatever type the character indicates (! for single precision, % for integer, and so on).

In the following program, for example, the variable A will be an integer variable, B will be a single-precision variable, P will be double precision, and Q will be a string.

```
10 DEFINT A-L : DEFDBL M-Z
20 A = 1/3
30 B! = 1/3
```

```
40 P = 1/3
50 Q$ = "1/3"
60 PRINT A, B!, P, Q$
```

When you RUN this program you'll get

```
0             .333333          .3333333432674408          1/3
```

A is zero because it is defined as an integer type in the DEFINT A-L statement, and 1/3 rounded to the nearest integer is zero. B! is single precision, because even though variables beginning with "B" are defined to be integers by the DEFINT A-L statement, the type declaration character "!" takes precedence over the DEFINT statement; and the single-precision representation of 1/3 has six digits. P is double precision because it is defined that way in the DEFDBL M-Z statement. Thus, P has 16 digits, but only six of them are accurate because BASIC used the single-precision division routine to divide 1 by 3, and then assigned the result to a double-precision number. (For more on arithmetic with different variable types see the next section on CINT, CSNG and CDBL.) Q$ is a string, because even though all variables beginning with the letter Q are defined to be double precision by the DEFDBL M-Z statement, the type declaration character "$" takes precedence over the DEFDBL statement. Q$ is not an arithmetic expression at all, but simply the string "1/3."

Array variables can also be any of the four variable types described above. (See the description of arrays and array variables in Chapter 3.) Their type can be defined with the type declaration characters !, %, #, and $, or it can be defined with a DEF--- statement. The program which follows provides an example of array variables whose type is defined with DEF--- statements.

Word Scrambling Program

In this program, to avoid having to type in the type declaration characters each time we use a variable name, we will use the statements DEFSTR A-G and DEFINT H-Z. This specifies that all variables whose names start with letters from A to G will be strings, and all variables whose names start with letters from H to Z will be integers.

The purpose of the program is to print out all the possible arrangements of letters in a word you type in. For instance, if you type in "CAT" the program will print out "CAT ACT TCA CTA ATC TAC." You can type in a word of any length, but look out! The number of possible arrangements of letters in a word is equal to the factorial of the number of letters. (For a discussion of factorial see the WHILE...WEND section in this chapter.) So, large words will have a great many arrangements: A five-letter word will have 120, and a six-letter word will have 720.

In Fig. 6-6, for example, are all 120 ways the letters can be arranged in the five-letter word 'STONE."

STONE	SNOET	TSOEN	OTSNE	OESNT	NESTO
STNOE	ENSOT	SENTO	OSNTE	TONES	NETSO
NTOSE	NOTES	SNTEO	SETON	TOENS	TESNO
OESTN	NTEOS	NOEST	SEOTN	EONTS	SONET
OETSN	OSTNE	NOSET	SOTEN	SNTOE	SOENT
SNETO	ONSTE	NSEOT	NTESO	SNOTE	NTOES
STNEO	ONTSE	TEONS	NTSEO	ETSON	NETOS
STENO	SOETN	TENOS	OENST	OTESN	EEOTS
NSOET	STOEN	TOSNE	SEONT	OTSEN	NTSOE
NESOT	ESNTO	SONTE	SENOT	TENSO	OTNSE
ONTES	NSTEO	ESTON	EOTNS	SETNO	TSEON
TNEOS	ONEST	ESOTN	NOETS	ENOST	TEOSN
TSONE	ONSET	OSTEN	NSTOE	EOSNT	ENSTO
TSNOE	SNEOT	TNESO	NSOTE	OTNES	ENTSO
TNOSE	ETONS	TNESEO	TESON	OTENS	ETSNO
EOSTN	ETNOS	EONST	TOESN	OENTS	OSNET
EOTSN	SOTNE	ESONT	TOSEN	TNSOE	OSENT
NSETO	NOSTE	ESNOT	ETNSO	TONSE	TNOES
TSNEO	NOTSE	EOTNS	ESTNO	STEON	ENTOS
TSEND	OSETN	ONETS	NEOST	ETOSN	ENOTS

Fig. 6-6. Arrangements of the word "stone."

Notice the words STENO, ONSET, TONES, and NOTES; and some words that might make good company names, like ETSON, TENSO, and OSNET.

This program might be useful for thinking up product or company names, given a certain group of letters (it is said that "Exxon" was invented this way), or for helping with crossword puzzles, or games such as anagrams and scrabble.

The program makes use of an unusual feature—a "recursive subroutine." A recursive subroutine is one which calls itself—that is, there is a GOSUB to the subroutine inside the subroutine itself! In this case for example, the subroutine starts at line 200 and goes on to line 360, but inside the subroutine, at line 240, there is a GOSUB 200. We'll explain how this works below. Recursion is a powerful technique in some circumstances for making programs simpler, shorter, and more efficient. The idea of recursion isn't hard once you get used to it, but figuring it out the first time may require you to put on your thinking cap. Fig. 6-7 shows the program.

How the Program Works—We'll talk about the "normal" parts of the program first, and save the recursive subroutine until last.

After defining the variable types as we discussed above, and clearing the screen, the program INPUTs, from the user, the word to be rearranged, and then figures out how long it is with the LEN statement in line 50. Using this

```
 10 ' ---- WORD SCRAMBLING PROGRAM
 20 CLS : DEFSTR A-G : DEFINT H-Z
 30 '
 40 ' ---- INPUT SECTION
 50 INPUT"WORD"; B : L=LEN(B)
 60 DIM A(L), Z(L)
 70 '
 80 ' ---- PUT LETTERS IN ARRAY A
 90 FOR N=1 TO L
100 A(N)=MID$(B,N,1)
110 NEXT N
120 '
130 ' ---- MAIN PROGRAM
140 J=L
150 GOSUB 200
160 STOP
170 '
180 ' ---- RECURSIVE SUBROUTINE
190 '
200 IF J=1 THEN GOSUB 390 : RETURN ' ---- PRINT & RETURN
210 Z(J)=1                              'INITIALIZE LOOP
220 ' -------------START OF LOOP--------------
230   J=J-1                             'REDUCE LENGTH OF SUBSTRING
240   GOSUB 200                         'GOSUB TO ITSELF
250   J=J+1                             'INCREASE LENGTH OF SUBSTRING
260   GOSUB 330                         'ROTATE J CHARACTERS
270   Z(J)=Z(J)+1                       'INCREMENT LOOP
280   IF Z(J) < J+1 THEN GOTO 230       'LOOP DONE?
290 ' --------------END OF LOOP---------------
300 RETURN
310 '
320 ' ---- SUBROUTINE TO ROTATE J CHARS IN ARRAY A
330 N=J : AT=A(N)
340 A(N)=A(N-1)
350 N=N-1
360 IF N=1 THEN A(N)=AT : RETURN
          ELSE GOTO 340
370 '
380 ' ---- SUBROUTINE TO PRINT WORD FROM CHARS IN ARRAY A
390 FOR Y=1 TO L
400 PRINT A(Y) ;
410 NEXT Y
420 PRINT"    ";
430 RETURN
440 '
450 END
```

Fig. 6-7. Word scrambling program.

length L it then sets up two, one-dimensional arrays that are L units long. Notice that because of the variable definition statements in line 20, the array "A" is a string array, and the array "L" is an integer array. In the next section of the program (lines 80 to 110), the program assigns the letters in the original

word to the array "A"—that is, it puts each letter in a different cell of the array, using the FOR...NEXT loop in lines 90 to 110.

There are two "normal" (nonrecursive) subroutines in the program. The one from lines 390 to line 410 simply prints out the letters in array "A." This subroutine is used after every time the letters are rearranged by the rest of the program, so that the new arrangement is printed out on the screen.

The purpose of the other nonrecursive subroutine is to rotate the letters in the array. What does "rotate" mean? Let's demonstrate this by taking the word "CAT" and rotating its letters one character to the right. We get "TCA." The C moved one character to the right, the A moved one character to the right, but the T didn't have any place to go, so it "wrapped around" and appeared on the left of the word. Fig. 6-8 shows this process.

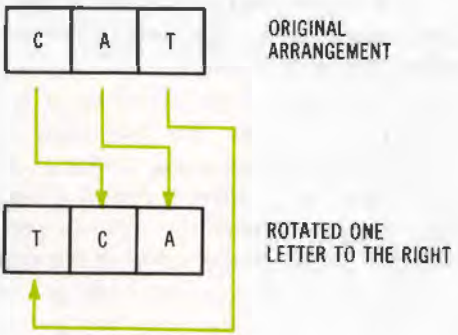


Fig. 6-8. Rotating a word.

This subroutine will rotate either all of the letters in the array, or only some of them, depending on the value of the integer variable J. That is, if the word typed into the program was five letters long, this subroutine could rotate either all five letters, or the first two letters, or the first three, or the first four.

How does this rotation subroutine work? Say, for example, that we're going to rotate three letters (either a three-letter word or the first three letters of a longer word). In this case J equals 3. When we start the subroutine we let N equal 3 as well. In line 330 we take the letter in the last array cell of the three cells we're going to rotate—this is A(3)—and temporarily store its value in variable AT. Then, in line 340, we store A(N-1) in A(N). Since at this point N=3 and N−1=2, this is equivalent to putting the letter that was in cell A(2) into cell A(3). Then, in line 350 we reduce the value of N by one, making it 2. Since in line 360 we find that N is not equal to 1, we go back to line 340 and execute it again, only this time N is 2 so we are transferring the contents of A(2) into A(1). We reduce N by one, and this time when we get to line 360, N *is* 1, so we set the first cell of the array, A(1), equal to the value was stored temporarily in AT. In this way, the contents of every cell have been moved one place to the right, and the rightmost cell has been moved around to the leftmost cell. Different numbers of cells can be rotated the same way.

The recursive subroutine, which is the heart of the program, uses these "normal" subroutine in the following way:

Suppose you input a one-letter word. After storing the word in array A, the main program will call the recursive subroutine from line 150, having set J equal to the length of the word (1 in this case). When the recursive subroutine executes the IF statement in line 200, J is equal to 1, so it will call the print subroutine with a GOSUB 390. Then the recursive subroutine returns to the main program, which STOPs in line 160. In this case, the recursive subroutine never had a chance to call itself. And, since the word had only one letter, there was only one way to arrange it, so we're done.

Now suppose we type in a two-letter word. This time when we get to line 200, J is equal to 2, so we pass by the rest of line 200 and enter a loop. The purpose of the loop is to cause all the statements in the loop (between lines 220 and 290) to be executed J times. The loop uses the counting variable Z(J) to do this. We must use an array variable here to do the counting, because later, when we have a three-letter word, we're going to start another loop with a different J while the first loop is still counting. In fact, for larger words, we'll have many loops, all counting at the same time, and all using the same part of the program, the lines from 220 to 290. The only way we'll keep straight what is happening is that each loop will have a different value of J and, therefore, a different counting variable Z(J). Fig. 6-9 shows the relationship of (1) the size of the word or part of a word we're working on, (2) the value of J, and (3) the value of the counting variable Z(J).

And now—if you're still with us—comes the interesting part. In line 230 we say J=J−1, so that now we are only dealing with the first letter of our two-letter word. Then (watch this!) we call the very subroutine that we're still in! But this time, when we enter the subroutine at line 200, J is equal to 1, so we do just what we did when we were talking about a one-letter word: print the word and then return. But this time when we return we are still in the subroutine: we come back to line 250. Here we increase J by one so it is 2 again, and then call the rotate subroutine to rotate the two letters. Then we check the counting variable to see if we have done all the above things twice, and since we haven't, we go back to line 230 to do them once more. When we have done them twice we will exit from the loop and go to line 300, which will RETURN us to the main program, where we'll STOP in line 160.

So, twice the program will first print and then rotate the two letters. The result is that the two possible combinations of the two letters are printed out.

Now, suppose you input a three-letter word. The program will call the recursive subroutine, which will do the following, three times: first it will set J equal to 2, then it will call itself, and do with the first two letters of the word just what we described for two-letter words: that is, it will print the word, rotate the two letters, print the word again, and rotate the two letters again. Then it will return from the J=2 version of the routine, increase J to 3 again, and rotate all three letters one place to the right. The subroutine will, as we said, do the above things three times. Then it will return to the main program. So if there were two combinations of letters for two-letter words, now for three-letter words there will be three times as many, (since we have just done,

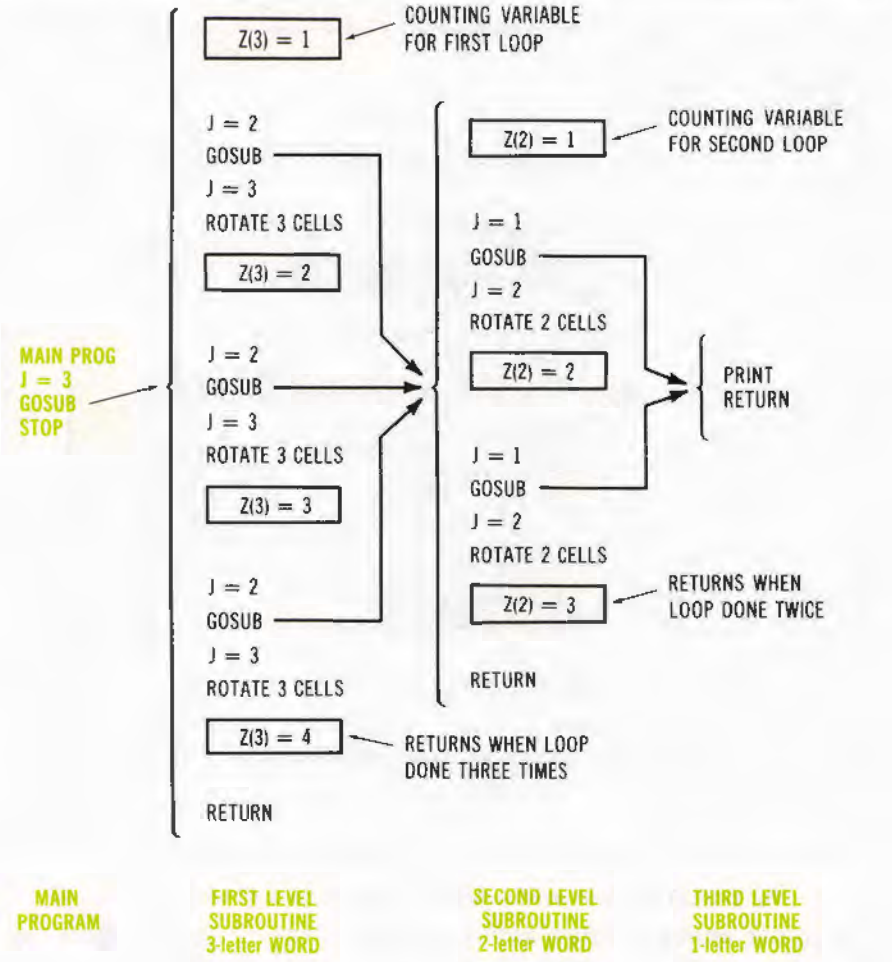


Fig. 6-9. Loops and counting variables.

three times, the same thing we did for two-letter words) which makes six possible combinations; and thus all six combinations of the word will be printed out. For a diagram of this process for three-letter words, see Fig. 6-10.

For four-letter words the subroutine will do the following, four times: first, it will do just what it does with three-letter words—print out the word with the first three letters arranged in all possible ways. Then second, it will rotate all four letters. When it has executed these two steps four times, it is done. Since three letters can be arranged in six possible ways, and we have done each of these six arrangements four times (rotating all four letters four times), we've covered all 24 possible arrangements!

And so on, for five letters, or six, or however many you have time to watch being printed out.

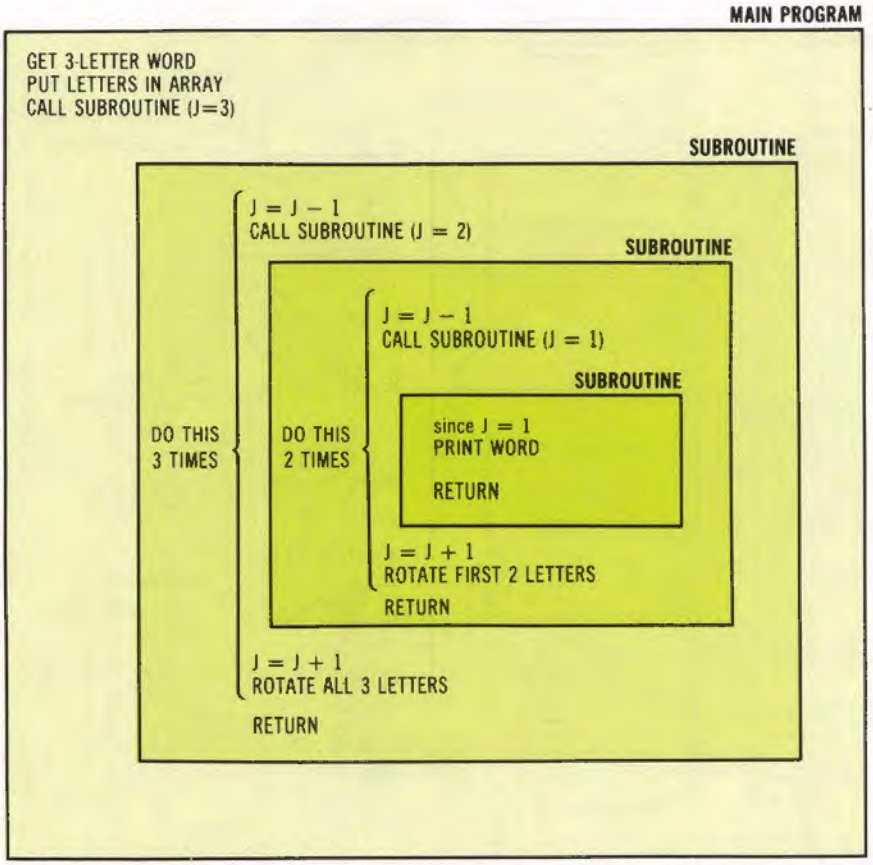


Fig. 6-10. Diagram of word scrambling program for a three-letter word.

Recursive procedures such as this are important in computer programming, and also in many other fields, such as music, art, and mathematics.

CINT, CSNG, and CDBL—Converting Variable Types

CINT(A) means "Convert variable A to an integer variable."
CSNG(A) means "Convert variable A to a single-precision variable."
CDBL(A) means "Convert variable A to a double-precision variable."

These three functions are used to change any of the three numeric variable types (integer, single precision, or double precision) to either of the other two types.

Before we describe how to use these functions, you should be aware that BASIC does many conversions of variables from one type to another which the programmer may not even be aware of. For instance, suppose there is a single-precision variable A which has a value of 3.33333. Now, suppose you execute the statement TAB(A). You can't TAB over to a fractional position, so BASIC automatically converts A to an integer with a value of 3. (A's original

type and value are not altered, of course.) Similarly, if you use a single-precision variable J as the counting variable in a FOR...NEXT loop, BASIC will convert it to an integer before using it as a counter.

BASIC also does automatic conversions when it operates on arithmetic expressions which include different variable types. To see what we mean, type in this program:

```
10 A#=6.123456789
20 B%=8
30 PRINT A# * B%
40 C = A# * B% : PRINT C
```

When you RUN this you'll get:

```
48.987654312
48.9877
```

Since both results are obtained by multiplying the same two numbers, 6.123456789 and 8, together, why aren't they the same? Well, A# is defined as a double-precision variable by the # sign, and B% is defined as an integer by the % sign. When they are multipled together BASIC carries out the multiplication operation using an arithmetic routine which is *as precise as the most precise variable to be operated on.* Single precision is more precise than integers, double precision is more precise than single precision. (For more discussion of the precision of different variable types, see the section "Variable Types" in this chapter.) In this case, since one of the variables is double precision, the multiplication will be carried out using a double-precision multiply routine. So BASIC first converts B% to double precision, then carries out the multiplication. If we simply print the result, as in line 30, we'll get a double-precision answer, with up to 16 digits of precision—in this case, 48.987654312. (There aren't sixteen digits shown here because BASIC chops off trailing or nonsignificant zeros.)

However, BASIC will also automatically convert a number to match the type of the variable it is being assigned to, giving it more or less precision as necessary. So in line 20, when we say "C = A# * B%," we are assigning a double-precision number (the result of the multiplication) to a single-precision variable C. (BASIC assumes that a variable is single precision unless told otherwise.) BASIC will therefore change this number to single precision before assigning it to C, so that when we print C, as in line 40, we'll get 48.9877, a single-precision number.

The terms "precision" and "accuracy" are sometimes confused. Roughly speaking, precision means how carefully you measure something, and accuracy means how close you come to the right answer. A quantity may be known precisely but not accurately, or accurately but not precisely.

For instance, suppose you use a ruler to measure, say, a pencil. You put the eraser end of the pencil at the 1-inch mark on the ruler, and read the length where the point is at 6 and 23/32 inches on the ruler. You then write down 6 and 23/32 as your answer. Unfortunately, you forgot to subtract the 1 that you started with: the pencil is actually 5 and 23/32 inches long. So, your answer has a precision of 1/32 of an inch, but it is only accurate to 1 inch.

On the other hand, maybe you toss the ruler down casually alongside the pencil for a quick estimate and see that the pencil is about five and one-half inches long. This answer only has a precision of ½ or maybe ¼ inch, but it is more accurate than the answer in the previous example.

Keep this distinction in mind when you are thinking about using variables of different precision in your programs. Just because you use, say, a double-precision number, doesn't necessarily mean that your answers will be accurate to 16 digits. For your answers to be accurate the input to your program must be equally accurate. (Your program must be correct, too, of course.) Doing very precise calculations with numbers that weren't that accurate to start with is one of the mistakes that gave rise to that well-known expression in the computer world: "Garbage In, Garbage Out."

BASIC is usually very clever at changing the precision of numbers in such a way that we don't even need to think about it. Sometimes, however, we will find it useful to force BASIC to change the precision of numbers when it wouldn't otherwise do so. That's what the functions CINT, CSNG, and CDBL are for.

For example, suppose we are writing a program that uses integer variables A% and B%. At some point in the program we want to multiply these two variables together and print the answer, but we want this answer to be correct to the last decimal place. This is no problem if A% and B% have small values, so that their product (the result of multiplying them together) is still in the range of integer variables (from 32767 to −32768). But what happens if A% and B% are so large that their product is no longer in this range? Try out this program:

```
10 A%=32111
20 B%=23999
30 PRINT A% * B%
40 C# = A% * B% : PRINT C#
50 PRINT CDBL(A%) * CDBL (B%)
```

You should get the following results:

```
7.70632E+08
770631872
770631889
```

We set A% and B% to 32111 and 23999, which are in the range for integers, but as you can probably see, when we multiply them together the result will be out of that range (greater than 32767). Can we still find a way to multiply them and print out an answer correct to the last decimal place? Our program attempts to do this in three different ways.

First, in line 30, we try multiplying them and printing out the answer directly. What happens? Since the answer is too big to be an integer, BASIC (without our permission!) converts it to single precision, and since the number is too large to print in the normal fashion in single precision, BASIC prints it in exponential notation. (For an explanation of exponential notation, see Appendix F.) The result, 7.70632E+08, has only six digits of precision, and since we can guess that when we multiply two 5 digit numbers together we will need 9 or 10 digits of precision, we know that some of the digits have been chopped off; so, this approach is not adequate.

All right, let's try converting the answer to double precision and printing that out, as in line 40. Here C# is defined as double precision by the # sign, and, sure enough, when we print it out we get nine digits, which is, in fact, the number of digits in the answer. The only trouble is, the last two digits are wrong! Why is this? Well, BASIC multiplied the two numbers together using the precision of the most precise number. Since both A% and B% are integers, BASIC starts by using the integer multiply routine. But then it finds that the result is too big to be an integer! What does BASIC do now? It goes back, changes both variables to single precision, and multiplies them again, using the single precision multiply routine. Then, since C# is double precision, it converts the single precision answer to double precision and assigns it to C#. But, since this answer started off as single precision, with only six significant digits, converting it to double precision doesn't really make it any more precise. The last two digits were lost in the multiply routine and can't be reclaimed. True, nine digits are printed out, but the last two have no meaning. So, we still haven't gotten the correct answer. What now?

CDBL to the rescue! (It's pronounced "C-Double.") In line 50 we convert both A% and B% to double precision using CDBL. Then we multiply them together. This time, BASIC uses the double precision multiply routine, since both numbers are double precision, and—it works! The double-precision routines can handle up to 16 digits, so they can handle 9 with room to spare. Our answer is finally correct to the last decimal place.

CINT (pronounced "C-Int") converts either single- or double-precision variables to integer variables. The value of the number to be converted must be in the range 32767 to −32768 or an OVERFLOW error will occur. CINT is similar to the INT function you learned about in Chapter 2. The difference is that CINT turns a single- or double-precision variable into an integer variable by rounding off the fractional part, while INT removes the fractional part of any number without changing its variable type.

CINT is useful when you have a single- (or double-) precision number which you want to also use as an integer (so you can do some operations on

it with fast integer arithmetic, for example). To create the integer version of, say, a double-precision variable, you can write V%=CINT(V#). Now you have two variables of different types, one double precision and one integer, and appropriate operations can be carried out on either one of them. Similarly, CSNG can create single-precision variables out of either integer or double-precision variables.

MISCELLANEOUS, BUT USEFUL

This section describes a group of advanced BASIC functions, which, although they don't fall neatly into any of the categories discussed earlier, can be dispensable in certain situations.

VARPTR—BASIC Looks Into Memory

VARPTR means, "Return the PoinTeR to a particular VARiable in the computer's memory."

For example, A=VARPTR(C#) will assign to A a number which is the first of a series of what are called memory "addresses" which contain the numerical value of the double-precision variable C#. (VARPTR is pronounced "var-pointer.") Similarly, B=VARPTR(D$) will assign to B a number which is the first of a series of addresses containing data about D$.

Before we can describe how VARPTR is used, we need to explain a little about how BASIC stores variables in your computer's memory.

As you know, the computer's memory is divided into units called "bytes." Each byte is 8 bits long, so a byte can have any of 256 possible values, from 0 to 255. (This is explained in more detail in Appendix A, "Numbering Systems.") Each byte has a separate "address"—a number (usually between 0 and 65535, although in 16-bit computers address can go up to 262,143)—so that BASIC can tell one byte of memory from another. The value of each BASIC variable is stored in memory, and, depending on which type the variable is (integer, single- or double-precision, or string), it takes up a different number of bytes. Fig. 6-11 is a representation of each of the four variable types stored in memory. We'll cover this figure in more detail below.

When writing ordinary BASIC programs you don't usually need to know where your variables are stored in memory or how many bytes each one requires. BASIC takes care of these "housekeeping" chores without any help from the programmer. However, if you are writing an assembly-language or machine-language program to work with your BASIC program, then you may need to know more about variable storage. (For a discussion of assembly-language programming, see the section on CALL and USR in Chapter 5.)

Suppose, for example, that you want an assembly-language program to take a particular string variable, already generated in your BASIC program, check it for lowercase letters, and convert any that it finds to uppercase. (Or you might want to count the number of words in a string or remove carriage returns.

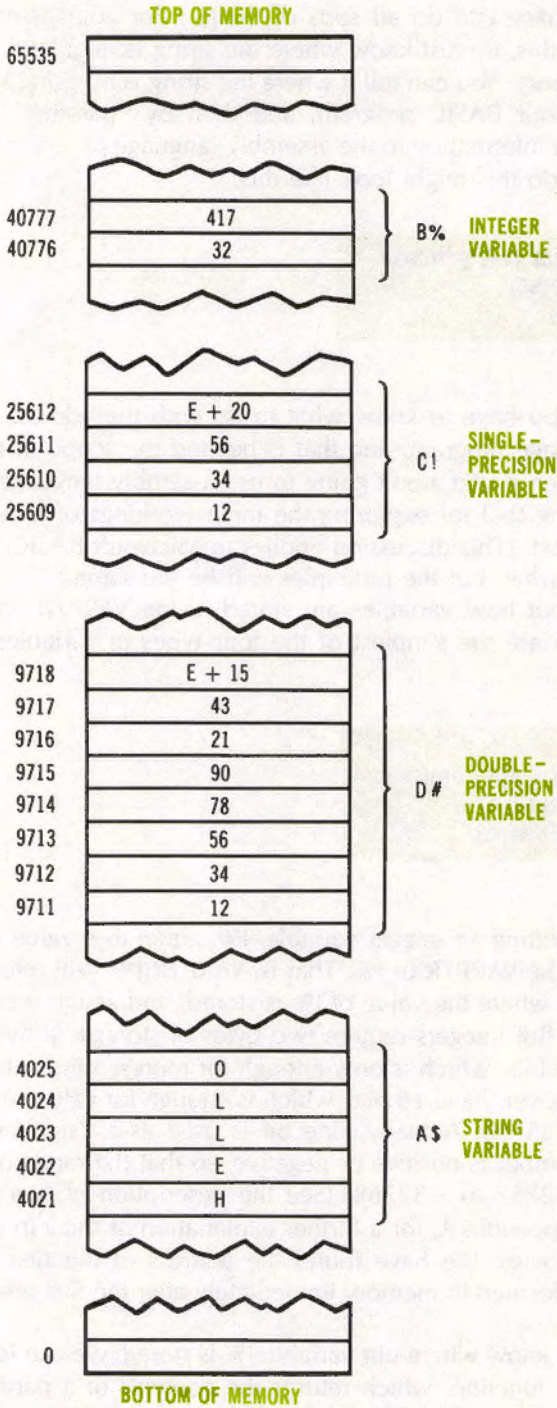


Fig. 6-11. Storage of variables in memory.

Assembly language can do all sorts of things.) For your assembly-language program to do this, it must know where the string is: that is, it must know its address in memory. You can tell it where the string is by using VARPTR to find the string in your BASIC program, and then by "passing" (that is, communicating) the information to the assembly-language program with USR. The instructions to do this might look like this:

```
10 S$="THIS IS THE STRING."
20 A=VARPTR(S$)
30 D=USR1(A)
```

Of course, you have to know what to do with the address A inside your assembly-language program, and that is beyond the scope of this book.

However, even if you aren't going to use assembly language, VARPTR can be an interesting tool for exploring the inner workings of how BASIC stores numbers and text. (This discussion applies to Microsoft BASIC. Other BASICs will vary somewhat, but the principles will be the same.)

Let's figure out how variables are stored, using VARPTR. We'll start with integers, which are the simplest of the four types of variables. Type in this program:

```
10 J% = 3
20 V = VARPTR(J%) : PRINT V
30 PRINT PEEK(V)
40 PRINT PEEK(V+1)
```

We start by setting an integer variable, J%, equal to a value of 3. Then we set V equal to the VARPTR of J%. That is, VARPTR(J%) will return the address of the first byte where the value of J% is stored, and assign it to V, which we then print out. But integers require two bytes of storage. Why? Because one byte contains 8 bits, which is only enough for representing values up to 255. Two bytes, however, have 16 bits, which is enough for values up to 256 times 256, which is 65536. Actually, one bit is used as a "sign bit" to indicate whether the number is positive or negative, so that the range of values is cut in half: from 32767 to −32768. (See the description of two's complement arithmetic in Appendix A, for a further explanation of this.) In any case J% is stored in two bytes. We have found the address of the first byte, and the second byte is located in memory immediately after the first one, in address V plus 1.

Now that we know where our variable J% is stored, we can look at its value using the PEEK function, which returns the contents of a particular memory location. (For a discussion of PEEK see Chapter 5.) In line 30 of our example

program we use the PEEK function to look at the first byte of J%, and in line 40 we look at the second byte.

Let's RUN the program and see what happens. In line 20 we will find the address where J% is stored. This address will vary a great deal, depending not only on what kind of computer you are using and how much memory you have, but also on whether you already have other programs in memory. Let's say this address turns out to be 26000. (All numbers in this discussion are decimal numbers.) We assign this number to V. Next we PEEK into V, which is an address with a value of 26000, and then we PEEK into the next address, V+1, which is 26001. We print out the values stored in these two addresses:

```
RUN
26000
3
0
```

The 26000 is the address of the first byte where J% is stored, the 3 is the contents of the first byte, and the 0 is the contents of the second byte. Since the value we set J% to was 3, we can deduce that the address 26000 contains the lower part of J% (the 3), and 26001 contains the upper part, which is 0. We call the lower part of the number the "least significant byte," or LSB, and the upper part we call the "most significant byte," or MSB. The LSB holds the lower 8 bits of the number, and the MSB holds the upper 8 bits.

To get a feel for this, try changing the value of J% and running the program again. For values of J% up to 255 you will find that this value appears, unaltered, in the LSB, while the MSB remains zero. But when you set J% to 256 and RUN the program, something different happens. Now the LSB is 0, and the MSB is 1. Why? Because the LSB was "full" when it contained a value of 255, so that adding one more caused it to become zero, with 1 to "carry" into the MSB. This is just like what happens when you add a 1 to a 9: the 9 (the one's column) is already full, so adding a 1 causes it to become 0, and a 1 to appear in the next column over. Fig. 6-12 shows how this works. What happens if J% is 257? The LSB will be 1, and the MSB will remain 1. If J% is 258, then LSB=2 and MSB=1. As we count upward the MSB will remain at 1 until J% is 512 (which is 2 times 256), at which point it will become 2. When J% is 32767, both the LSB and the MSB will be full and trying to add one more will cause an OVERFLOW error.

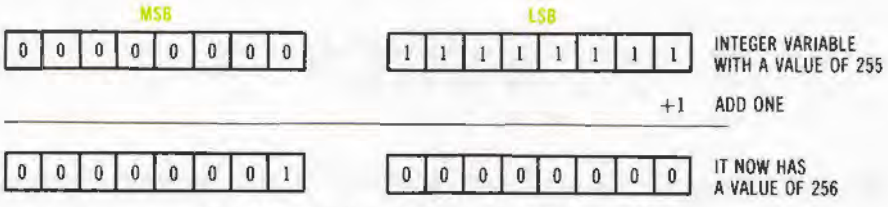


Fig. 6-12. Carrying a bit from the LSB to the MSB.

We summarize this in Fig. 6-13.

Value of J%	MSB	LSB
0	0	0
1	0	1
2	0	2
254	0	254
255	0	255
256	1	0
257	1	1
258	1	2
510	1	254
511	1	255
512	2	0
513	2	1
514	2	2
32765	127	253
32766	127	254
32767	127	255
32768	(cannot be represented)	

Fig. 6-13. Integer representation of numbers.

Every time we add 256 more units to the LSB, we increase the value of the MSB by one. This gives us a rule (sometimes called an "algorithm") for figuring out the value J% if we know the values of the LSB and the MSB:

```
J% = LSB + (MSB * 256)
```

So, if you PEEK into memory and find the LSB and the MSB of a particular integer, you can find the decimal value of the integer by multiplying the MSB by 256 and then adding the LSB. Similarly, if you know the decimal value of an integer, you can find the values of the LSB and the MSB:

```
MSB = INT (J%/256)
LSB = J% -(MSB * 256)
```

The other variable types are more complex and require more bytes of memory storage than integers do.

Single-precision variables require four bytes (five on some computers), arranged this way:

```
Y = LSB of mantissa
V+1 = Next most significant byte (NMSB) of mantissa
V+2 = MSB of mantissa, with one sign bit
V+3 = exponent, with one sign bit
```

(For a discussion of "mantissa" and "exponent," refer to Appendix F, Exponential Notation.)

Double-precision variables require 8 bytes of memory, arranged like this:

V = LSB of mantissa
V+1 = NMSB of mantissa
V+2 = NMSB of mantissa
V+3 = NMSB of mantissa
V+4 = NMSB of mantissa
V+5 = NMSB of mantissa
V+6 = MSB of mantissa
V+7 = exponent

(Remember, the actual storage arrangements of single- and double-precision variables vary somewhat from one version of BASIC to another, so for more precise information you should consult the manuals for your own computer.)

String variables are stored somewhat differently than numeric variables. Instead of pointing to the string itself, VARPTR returns the first of three addresses, the first of which is the length of the string, and the last two of which are in turn pointers to the actual location of the string in memory.

V = length of string
V+1 = LSB of address where string starts
V+2 = MSB of address where string starts

To show how VARPTR is used to find a string in memory, let's look at the following program, which will find a string and print it out by PEEKing at the values of the characters it finds in memory.

```
10 S$ = "THIS IS THE STRING"
20 V = VARPTR(S$)
30 L = PEEK(V)
40 A = PEEK(V+1) + ( 256 * PEEK(V+2) )
50 '
60 FOR J=1 TO L
70 PRINT CHR$ ( PEEK(A+J-1) );
80 NEXT J
```

This program first finds the variable pointer V for the string S$. Then it finds L, the length of the string, which is stored in the first address pointed to by V. The second and third addresses pointed to by V contain the address where the string starts. We reconstruct this address by multiplying its MSB by 256 and adding the LSB. The result is A. To print out the string, we set up a loop to count through the number of characters; L, in the string. The first character is at address A, the second at A+1, and the last at A+L−1. We PEEK at each

location, then change the numeric value we find there to a string value with the CHR$ function, and print it out.

Fig. 6-14 shows how the pointers are arranged for a string variable.

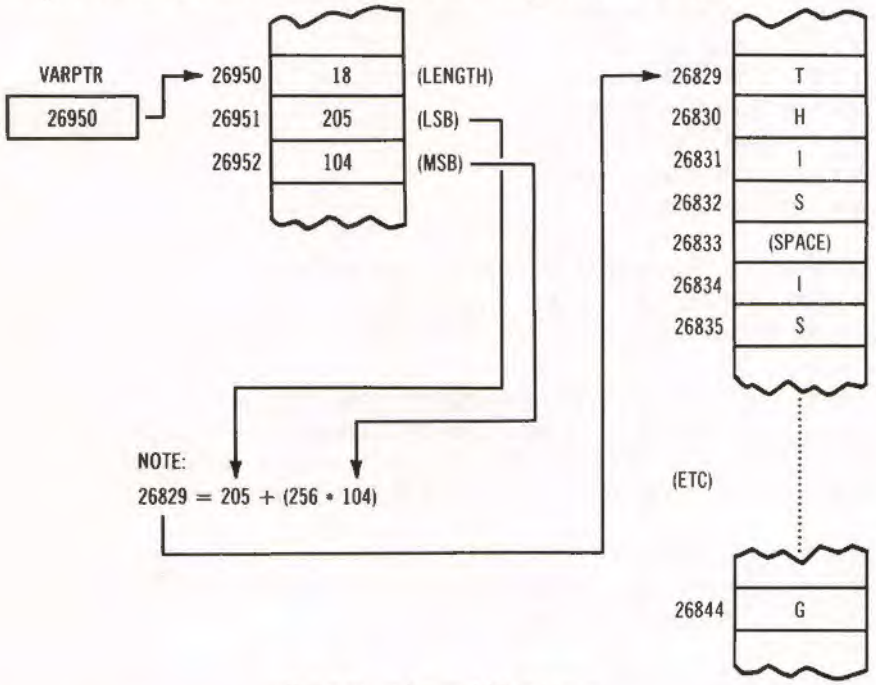


Fig. 6-14. String variable storage.

VARPTR returns a value, 26950, which is a pointer to the three locations that hold: first, the length of the string, which is 18 (in 26950); second, the LSB of the address, which is 205 (in location 26951); and last, the MSB of the address, which is 104 (in location 26952). We can calculate the actual address where the string starts by multiplying the MSB of the address by 256 and adding the LSB, which gives us 26829. Remember that all these addresses will be different, depending on what computer you're using, how much memory it has, and what programs are already in memory.

Also, keep the following in mind: when using VARPTR, you should place the VARPTR statement as close as possible in your program to the place where the value it returns will be used. For instance, it should go immediately before the USR statement, if you are going to use the value it returns in an assembly-language routine. The reason for this is that BASIC is constantly changing the storage locations of the string variables in memory. It does this when you change the length of a string variable in your program, and it also occasionally does it all by itself to compact existing strings into the smallest possible space. So, if you use VARPTR to get an address in one part of your program, and then do some string variable operations before you make use of the address, it might well have changed before you use it!

HEX$, OCT$, &H and &O—Hexadecimal and Octal Conversions

HEX$(N) means, "Convert the decimal number N to HEXadecimal notation."

OCT$(N) means, "Convert the decimal number N to OCTal notation."

&Hnnnn means, "Convert the hexadecimal number nnnn to decimal notation."

&Onnnn means, "Convert the octal number nnnn to decimal notation."

(In the above definitions "N" represents either a decimal number or a variable which has the value of that number. "nnnn" represents a hexadecimal or octal number, but cannot be a variable.)

These four functions are used to convert numbers back and forth between hexadecimal (or octal) notation and decimal notation. They are very useful functions, especially if you will be using any assembly-language routines with your BASIC program, but, unfortunately, not all versions of BASIC have all four of them. For a discussion of hex ("hex" is a shortened form of "hexadecimal") and octal numbers, read Appendix A, "Numbering Systems."

HEX$ (or OCT$) takes the value of a *numeric* variable or constant and turns it into a *string* variable which is the hex (or octal) equivalent of that value. For example, if you execute the statement PRINT HEX$(255), your computer will print FF, which is the hex equivalent of the decimal number 255.

Here's a program that will print the first 16 decimal numbers and their hex equivalents:

```
10 FOR D=0 TO 15
20 H$=HEX$(D)
30 PRINT D; H$
40 NEXT D


RUN
 0  0
 1  1
 2  2
 3  3
 4  4
 5  5
 6  7
 8  8
 9  9
10  A
11  B
12  C
13  D
14  E
15  F
```

HEX$ and OCT$ are useful if you want to see what the hex or octal representation of a number is. However, you can't use the converted form of the number in numerical calculations, since it is a string variable. In other words, an expression like " PRINT HEX$(127) + 2" is not legal.

&H and &O convert the hex or octal representations of numbers into decimal form. (This is the opposite of HEX$ and OCT$.) A limitation of &H and &O is that they work only with constants, not variables. In other words, you can write $H7FF0, where 7FF0 is a hexadecimal number, but you can't write &HX, where X is a variable whose value is a hexadecimal number. This restricts the use of &H and &O to situations where you know when you write your program what values you will want to convert. You can't, for example, use &H and &O to convert values you type into a program with an INPUT statement.

&H is often used for converting addresses, since addresses are frequently expressed in hex form. For example, if you want to PEEK into the hex memory address 7AB0, you don't need to convert 7AB0 to its decimal equivalent (which is 31488). You can simply write:

```
PRINT PEEK (&H7AB0)
```

&H is also useful if you want to POKE a machine-language routine into memory. You don't need to convert the hex values the routine was originally written in into decimal values, if you use &H. For example, suppose that you want to POKE a five-byte routine into memory starting at location 7ABO. The five bytes are 7A, 2B, 3C, 00, and FF. This program will do it:

```
10 FOR J=0 TO 4
20 READ B
30 POKE B, &H7AB0 + J
40 NEXT J
50 DATA &H7A,&H2B,&H3C,&H00,&HFF
```

In this program we READ the first value, 7A (hex), from the list of DATA items, then POKE it into address 7AB0 (hex). The next time through the loop we read the next item, 2B, and POKE it into address 7AB0 plus 1, which is 7AB1. The process continues until all five values have been POKEd in.

OPTION BASE—Saving Array Space

OPTION BASE means, "Use the OPTIONal BASE for arrays, instead of 0."

When you create an array you use the DIM statement, which establishes how many elements array will have. (Refer to Chapter 3 for an explanation of arrays.) For example, DIM A(4) sets up an array with five elements: (A0), A(1),

A(2), A(3), and A(4). The zero element, A(0) in this case, is always the first element of an array. Many times when you set up an array this is just what you want; it is convenient to have the array values start at the zero element. However, there are situations when it is more convenient for the array to start at the first element, A(1). For instance, in the TIC-TAC-TOE program described in Chapter Three, a 3 by 3 array "A" is used to represent the TIC-TAC-TOE board. So it will be easier to visualize, the subscripts discussed in the program run from 1 to 3 (instead of 0 to 2). However, since a DIM A(3,3) statement is used to set up the array, it is actually a 4 by 4 array, because each of the two dimensions has subscripts running from 0 to 3, making 4 elements in each dimension, for a total of 16. These unused elements represent a waste of memory space.

This is not a serious problem in one-dimensional arrays, which have only one zero element. But in multidimensional arrays there can be many more wasted elements since the number of zero elements increases rapidly with the dimension of the array and the number of elements in it. In the TIC-TAC-TOE program referred to above, only 9 elements (3 by 3) are used out of 16 (4 by 4), so that 7 are wasted. In a 20 by 20 by 20 three-dimensional array, which has 8000 elements, there are 1141 zero elements! If they are not used, this represents a considerable loss of memory space.

The purpose of OPTION BASE is to avoid wasting these elements. In the example of the TIC-TAC-TOE program, we could add the following statement to the program:

```
9 OPTION BASE 1
```

This statement tells the BASIC interpreter to make the first element in all arrays 1 instead of zero. When we then execute the DIM A(3,3) in line 10, an array will be set up with two dimensions, each with subscripts running from 1 to 3 (instead of from 0 to 3 as before), so that there will be 9 elements instead of 16.

In some circumstances you might even want to start your array elements with a number greater than one. For example:

```
10 OPTION BASE 10
20 DIM A(12)
```

will set up an array with three elements: A(10), A(11), and A(12).

SWAP—Exchanging Two Variables

SWAP means "SWAP (exchange) the values of two variables."

Here's how we would exchange the values of two variables, A and B, if we did not have the SWAP statement on our computer:

```
10 T=A
20 A=B
30 B=T
```

First, in line 10, we assign the value of the variable A to another variable, T, which acts as temporary storage. Then we assign the value of B to the variable A in line 20; and finally, in line 30, we take the old value of A, which we had temporarily assigned to T, and assign it to B.

However, using the SWAP statement, all we have to say is:

```
10 SWAP A,B
```

Here one statement does everything three statements did above, and we don't have to use an additional variable for temporary storage.

IN CONCLUSION

Congratulations! If you have read this far, and if you have understood most of what you read, and experimented with the various BASIC functions on your own computer as you encountered them in this book, then you are well on your way to becoming an expert programmer.

All you need to do now is remember the old adage, "Practice makes perfect." Programming is a unique field in that the more you program, the better you get at it, even if you do not have a teacher to watch over your progress. This is because your computer is one of the best teachers you could ask for. If you make a mistake, it will let you know. If you are a slow learner, it will be patient with you. And when you've finally gotten everything right, it will reward you with a program that does just about anything you want it to.

There is another adage, "The pen is mightier than the sword." Adapting this to modern times, we might say that information is more powerful than energy or physical force. The evidence for this is everywhere; computers, the most sophisticated processors of information, are becoming more and more important in almost every part of our lives. And now, with your newly gained knowledge of programming, you are in an excellent position to join the computer revolution, and wield a tool that is even mightier in our day than the pen has been since ancient times.

REVIEW QUESTIONS

1. What are three ways to make programs do loops?

2. In this example A and B are variables that can be either true (equal to 1) or false (equal to zero). You can type in either a zero or a one for A and B. Will this program print out accurate statements about the truth of A and B?

```
10 INPUT A, B
20 IF A THEN IF B THEN PRINT"BOTH TRUE" ELSE
PRINT"A TRUE, B FALSE" ELSE PRINT"A FALSE"
```

3. What are two advantages of using DEF FN instead of a subroutine?

4. What are two differences between LINE INPUT and INPUT?

5. Is there any keyboard character you can NOT use in the string you type into a LINE INPUT statement?

6. In the following program, if you hold the"A" key down, will a series of "A"s be printed?

```
10 IF INKEY$ = "A" THEN PRINT "A" : GOTO 10
20 STOP
```

7. What value will P have after you run this program?

```
10 S$="CATCATCAT" : W$="CAT"
20 P=INSTR(4,S$,W$)
```

8. What happens if you execute this statement?

```
10 PRINT STRING$(500, "*")
```

9. Will this program print the message "CATS DEVOUR MICE"?

```
10 A$="CATS EAT MICE"
20 MID$(A$,6,6)="DEVOUR"
30 PRINT A$
```

10. What's the difference between RESUME and RESTORE?

11. What's the difference between ON ERROR GOTO 0 and RESUME?

12. What are two advantages of using integers instead of single-precision variables? What are the disadvantages?

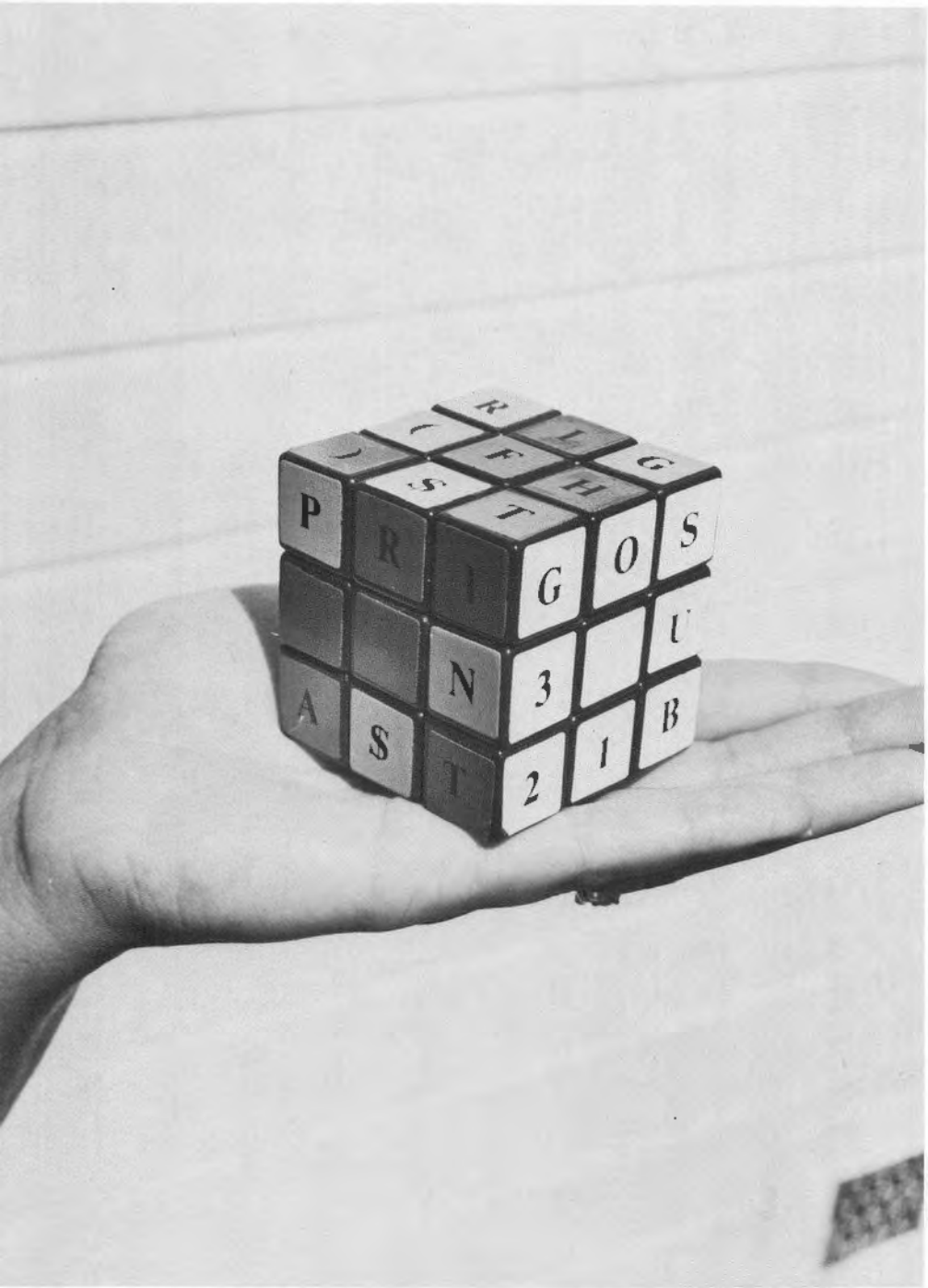
13. How many bytes of memory are used by integers? Single-precision variables? Double-precision variables? Strings?

```
10 PRINT HEX$(255), OCT$(63), &HFF, &077
```

14. Tell what will be printed out if you run the following program.

```
10 PRINT HEX$(255), OCT$(63),&HFF,&077
```

15. Exercise, for extra credit: Write a program which uses a recursive subroutine to find the factorial of a number.


P R I G O S
A S T 2 1 B
N 3 U

CHAPTER

7

Rubik's Cube Program

This chapter describes a program which will "model" or simulate the Rubik's cube on your computer. If you are not familiar with Rubik's cube, well—where have you been? Rubik's cube is by far the most famous puzzle of the century, and is an object of study and fascination to mathematicians, computer scientists, small children, and just about everyone else who has an inquiring turn of mind.

What is the value of a program about the Rubik's cube to someone who is learning BASIC? What we hope to do in this chapter is to give you an idea how to go about using your computer to solve a fairly complicated problem, starting with a rough plan of what you want to accomplish, and ending up with a working program. Along the way we will explore such topics as "top down" and "bottom up" program design, flowcharting, and the use of different data representations to "model" real-world situations. Our aim is that when you finish this chapter you will feel confident about attacking programming jobs that before seemed too large or too complicated. You will also end up with a program which is a lot of fun, can be used for serious study of Rubik's cube, and will amaze your friends!

Where did the Rubik's cube come from? Erno Rubik is a teacher of architecture and design in the Hungarian city of Budapest. The idea for the cube came to him when he was trying to find a way to help his students visualize three-dimensional shapes. Rubik's cube is a 3 by 3 by 3 matrix, any face of which (front, back, left, right, top and bottom) is free to rotate with respect to the remaining five faces. Each face starts off being a single color, but after a few twists of the various faces the colors become hopelessly scrambled, and the puzzle then consists of getting the cube back to its original configuration, a task which may take weeks (if you can do it all). The difficulty of the puzzle can be appreciated when you realize that there are 43,252,003,274,489,856,000 possible ways to arrange the cube, and only one of them is correct.

All this is practically impossible to describe to someone who has not seen a cube, but it is self-evident once you have one in your hands. If you find this

discussion confusing, you should hurry out and buy a cube before reading further.

What do we want our program to do? Essentially, we would like to put a picture of the cube on the computer screen, and then, by typing in commands, be able to rotate whatever face we select. It would also be nice if the computer could remember the series of rotations ("moves") that a lead to a particular configuration, and be able to retrace a number of moves if we find we have made a mistake. And, of course, we want to be able to restore the cube to its original unscrambled state whenever we want. With these capabilities we will be able to explore simple configurations without always having to deal with a hopelessly scrambled cube.

Our program will not "solve" the cube—that is, restore a scrambled cube to its original state without human intervention. While this task is entirely practical, and has in fact been accomplished by many programs, it involves a complex set of algorithms which are beyond the scope of this book.

NAMING THE PARTS OF THE CUBE

In order to talk about the cube we need to agree on what to call its various parts. Look at Fig. 7-1. The cube is composed of smaller cubes. Let's call these smaller cubes "cubies."

When you hold a Rubik's cube in your hand, different amounts of each of these cubies are visible, depending on where they're situated on the cube. There are 8 corners to the cube, with a cubie occupying each corner. You can see 3 faces of each of these "corner" cubies. Let's call the faces of the cubies "facelets," to distinguish them from the faces of the cube itself, which we'll continue to call "faces." (These names are taken from Douglas Hofstadter's excellent article on Rubik's Cube in the March 1981 issue of Scientific American.)

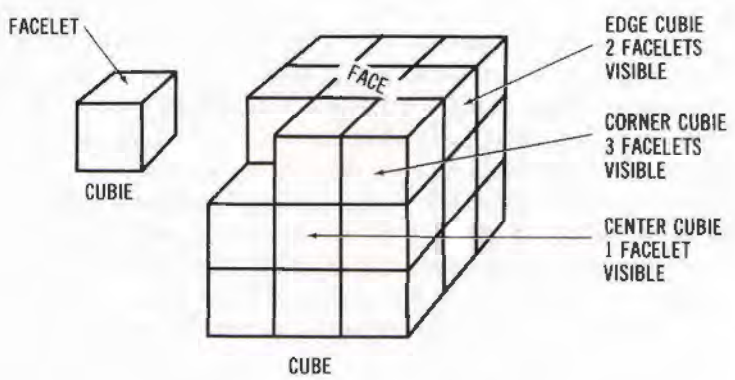


Fig. 7-1. The parts of the cube.

There are also 6 cubies which we can call "center" cubies—they occupy the center of each of the 6 faces of the cube, so they each have just one facelet showing. And finally there are 12 "edge" cubies, each with 2 facelets visible. This is a total of 54 visible facelets (since 8 * 3 plus 6 * 1 plus 12 * 2 is 54). This makes sense, since there are 6 faces on the cube, each with nine facelets, which is also 54 (3 * 3 * 6).

Next we need to agree on what to call the six faces of the cube, so we can distinguish one face from another. On a real cube, of course, the faces are different colors, but there is no standardized coloring method, and in a computer program, where we will never rotate the entire cube at once (the way you can turn a real cube around in your hands), it is better to use names that relate to the positions of the faces. The names we will use are front, back, left, right, up, and down; which we will usually refer to by their intials F, B, L, R, U, and D. We could have used "top" and "bottom" instead of "up" and "down," but then we would have had two words starting with "B" ("back" and "bottom"). We could also have compromised with "top" and "down", but that might cause confusion if someone saw a "top" and then wondered where the corresponding "bottom" was.

Let's also review what happens when you rotate one of the faces of the cube. There are six faces, each of which can be rotated in three different ways. You can turn a face a quarter-turn (90 degrees) clockwise, you can turn it a quarter-turn (90 degrees) counter-clockwise, or you can turn it two quarter-turns clockwise (which is half a turn, or 180 degrees, and the same as turning it two quarter-turns counter-clockwise). Let's imagine rotating a face, say the front, one quarter-turn clockwise, as shown in the diagram of Fig. 7-2.

When you do this, all the facelets on the F face move around to new positions, still on the F face. Also, the three facelets on the L face which are adjacent to the F face, move up to new positions on the U face (still adjacent to the F face). Similarly, three facelets on the D face move to the L face, three facelets on the R face move to the D face, and three facelets on the U face move to the R face.

Try this with a real cube. Later, when we start writing the program, it will be important to have a clear understanding of what is involved in the rotation process.

INITIAL PLANNING DECISIONS

In order to plan how to write our program we need to make some decisions about what we want it to do, and how we want it to do it.

The Picture on the Screen

The most obvious question is: how will we display a three-dimensional cube on a two-dimensional computer screen? If you think about it, you can probably imagine several ways to go about this.

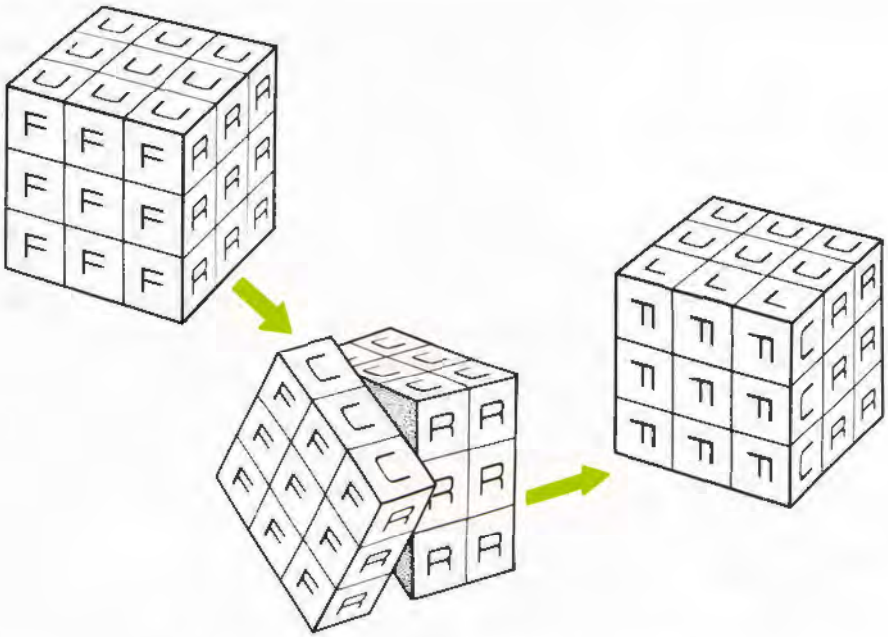


Fig. 7-2. Front face rotated one-quarter turn clockwise.

We could use two side by side pictures, one showing, for example, the front, upper, and right sides, while the other shows the lower, back, and left sides. Such a display might look like that shown in Fig. 7-3.

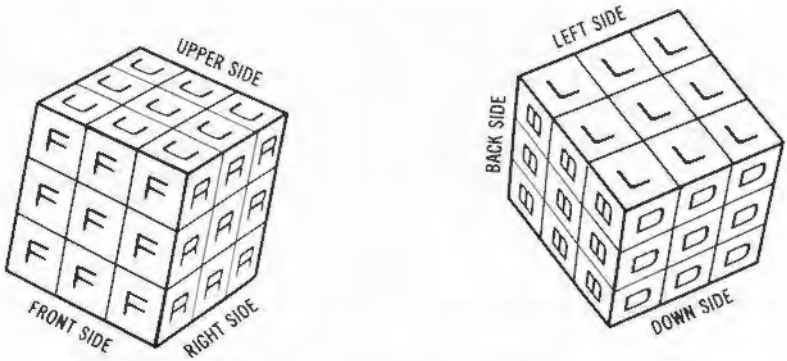


Fig. 7-3. Two side-by-side cubes.

We could "unfold" the cube like a cardboard box and show the unfolded and flattened faces joined by their corners. That might look like that shown in Fig. 7-4.

We could use an "exploded" view, in which each face retains its same position, but is moved far enough from the other faces so that we have an unobstructed view of it. The three diagrams of Fig. 7-5 show how the faces are moved apart to achieve this representation.

None of these three approaches is perfect. In the first approach, showing two pictures, it is hard to visualize how the two pictures join together, and hard to imagine what will happen when we rotate a face of the cube. In the second approach it is hard to visualize how the separated edges join together, especially those farthest from each other, like the left side and the back. The exploded view overcomes these difficulties, although it does have an abstract quality that takes a little getting used to.

The exploded view seems promising, but before we decide to use it in our program, lets see how it looks on the computer screen. To do this in a simple way we'll simply write the following short program.

```
100 CLS
110 PRINT"                U / U / U"
120 PRINT"     L        U / U / U"
130 PRINT"  L /       U / U / U    B   B   B"
140 PRINT"L / L                    ----------"
150 PRINT"/ L /                    B   B   B"
160 PRINT"L / L                    ----------      R"
170 PRINT"/ L /                    B   B   B     R /"
180 PRINT"L /     F   F   F                    R / R"
190 PRINT"/       ----------                   / R /"
200 PRINT"        F   F   F                    R / R"
210 PRINT"        ----------                   / R /"
220 PRINT"        F   F   F       D / D / D    R /"
230 PRINT"                      D / D / D      /"
240 PRINT"                    D / D / D"
250 PRINT STRING$(40, "-")
260 PRINT"COMMAND: ";
270 GOTO 270
```

In this program the CLS in line 100 clears the screen, and the PRINT statements simply reproduce all the characters and spaces that appear between the quote marks. The STRING$ statement in line 250 prints a line of 40 dashes (see the section on STRING$ in Chapter 6). Line 260 prints the word "COMMAND" on the screen as a prompt to the user to type in his input. (The user can't really input anything in this trial program, of course, but in the real program he'll type instructions to the program following "COMMAND.") Finally, line 270, by constantly going to itself, "freezes" the display so that BASIC will not print "READY" or "OK."

Depending on the size of the letters and the line spacing on your particular computer screen, you will get a display something like Fig. 7-6 when you run the program.

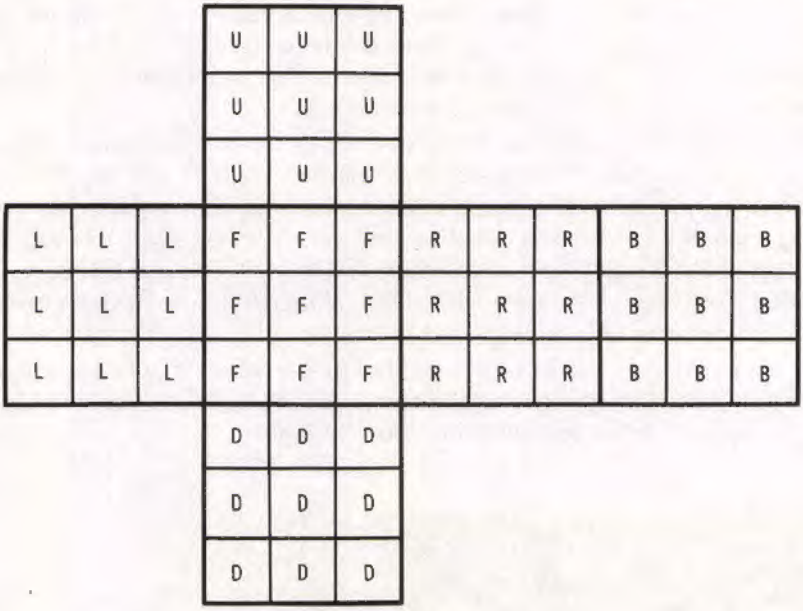


Fig. 7-4. Unfolded cube.

Compare this display with the exploded view in Fig. 7-5. It may seem confusing at first, but after you look at it for a time it begins to make sense. (Remember that you are looking at the *outside* of the front, right, and top faces, but at the *inside* of the back, left, and down faces: that is, you are seeing them as they would be seen from inside the cube.) We'll use this display in our program.

We've chosen the size of the display to be 40 characters wide and 16 lines high, since some computers have only a 40 column width and others can only display 16 lines. This way, the display should be adaptable to most computers.

Representing the Cube With an Array

The second major decision we must make is how to go about representing the cube in the computer program. That is, what sort of "data structure" in the program will we use to hold the information about the configuration of the cube? The cube is a 3 by 3 by 3 stack of cubies, so that the first idea you might have is to use a 3 by 3 by 3 BASIC array, with each array cell representing one cubie. (This would be something like stacking up three of the 3 by 3 arrays you used for the TIC-TAC-TOE board in Chapter 3.)

However, we've already seen that the corner cubies have three facelets and the edge cubies have two. And these cubies have different orientations, depending on how they're rotated. For example, suppose we rotate the cubie at

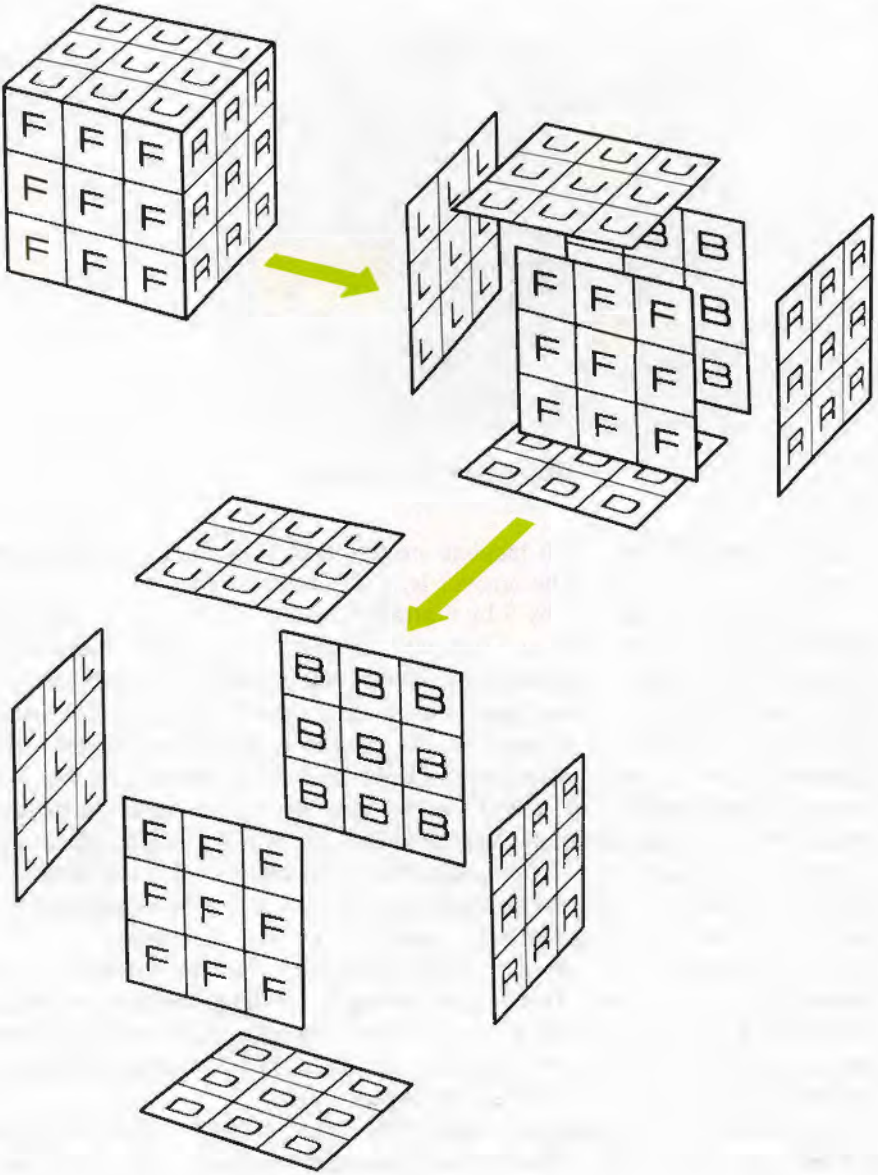


Fig. 7-5. Exploded view of cube.

"up right front" down to the position at "down left front," as shown in Fig. 7-7.

It is still the same cubie, but now two of its three facelets are facing in different directions. Clearly, we need to keep track not just of where each

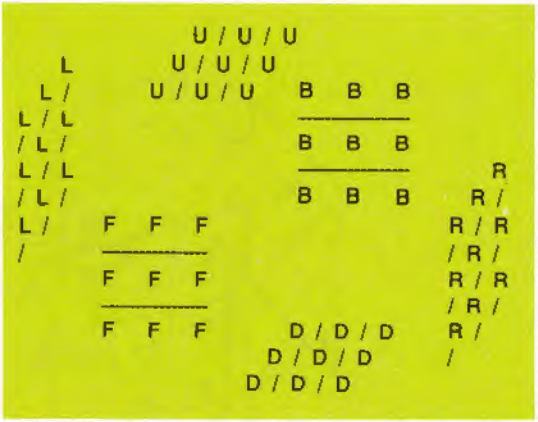


Fig. 7-6. The screen display.

cubie is, but which way all its facelets are oriented. How could we do this in a simple 3 * 3 * 3 array? The answer is, it wouldn't be easy.

One solution is to use a 5 by 5 by 5 array instead of a 3 by 3 by 3. Why do that? Well, with a larger array we can use a whole bunch of cells in the array to represent one cubie: a corner cubie can be represented by 8 cells, and an edge cubie by four. This way, when we rotate a face of the cube, we rotate two "slices" of the array instead of one, and now there are enough cells representing each cubie so that we can keep track of its orientation. Fig. 7-8 shows how this would look. Don't worry if you don't understand all the details of the array representation. As it turns out, we're not going to use arrays to represent the cube. We are showing them here because it is useful to see how the planning process can include the exploration of various alternatives. Our reasons for discarding the array approach will be clear soon.

Imagine rotating the "up right front" cubie one half turn down to the "down left front" position. This is represented by rotating the front two slices of the 5 by 5 by 5 array in Fig. 7-5. Note how, when the eight array cells that represent the "up right front" cubie move down to the "down left front" position, the orientation of the facelets is preserved.

To get an idea what might be involved in using a 5 * 5 * 5 array to represent the cube, let's write a short program to rotate one face of the cube one-quarter turn clockwise. For simplicity we'll rotate only the inner 3 by 3 array, representing the facelets on the particular face we are rotating. We won't worry, for the moment, about the facelets on the adjacent faces. Fig. 7-9 shows the array and the program steps needed to rotate it.

Since the array is going to hold letters representing faces of the cube (F, B, U, D, L, R), we use a string array rather than a numeric array. (We could also have used a numeric array and stored ASCII representations of the letters.) BASIC operates on one array element at a time, but since we will be moving

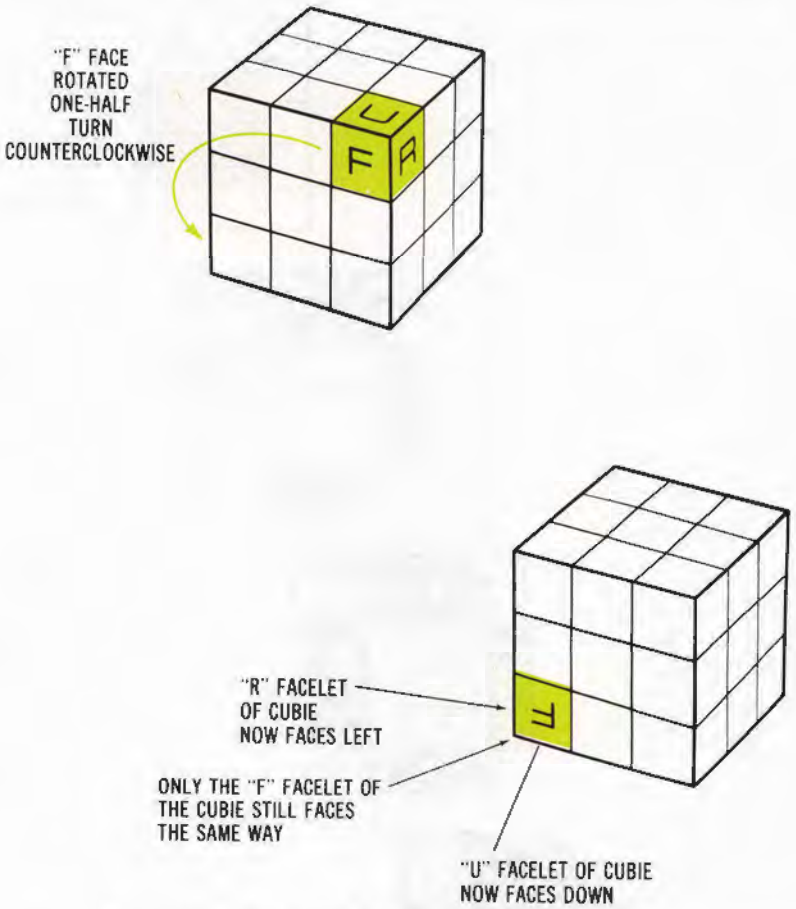


Fig. 7.7. "F" face turned one-halfturn counterclockwise.

groups of three-array elements at a time (representing a row of facelets), we will need to use FOR...NEXT loops which count up to three to move each row of array cells to its new position.

We start by putting the top row (elements 1, 2, and 3) into a temporary variable T$. This keeps them from being "written over" and lost in step (b) when we move the left-hand row (elements 7, 4, and 1) into the top row. We continue to work around the array, moving the bottom row elements into the left column, the right column elements into the bottom row, and finally the top row elements, which were stored in T$, into the left column.

Of course, when we really rotate a face of the cube, we need to change not only the face itself, but the cells adjacent to it on four other faces. So, as you can see, it will take quite a few program steps to perform a rotation. And there are six faces, each of which will require a separate rotation routine. Might there be an easier way to represent the cube in our program?

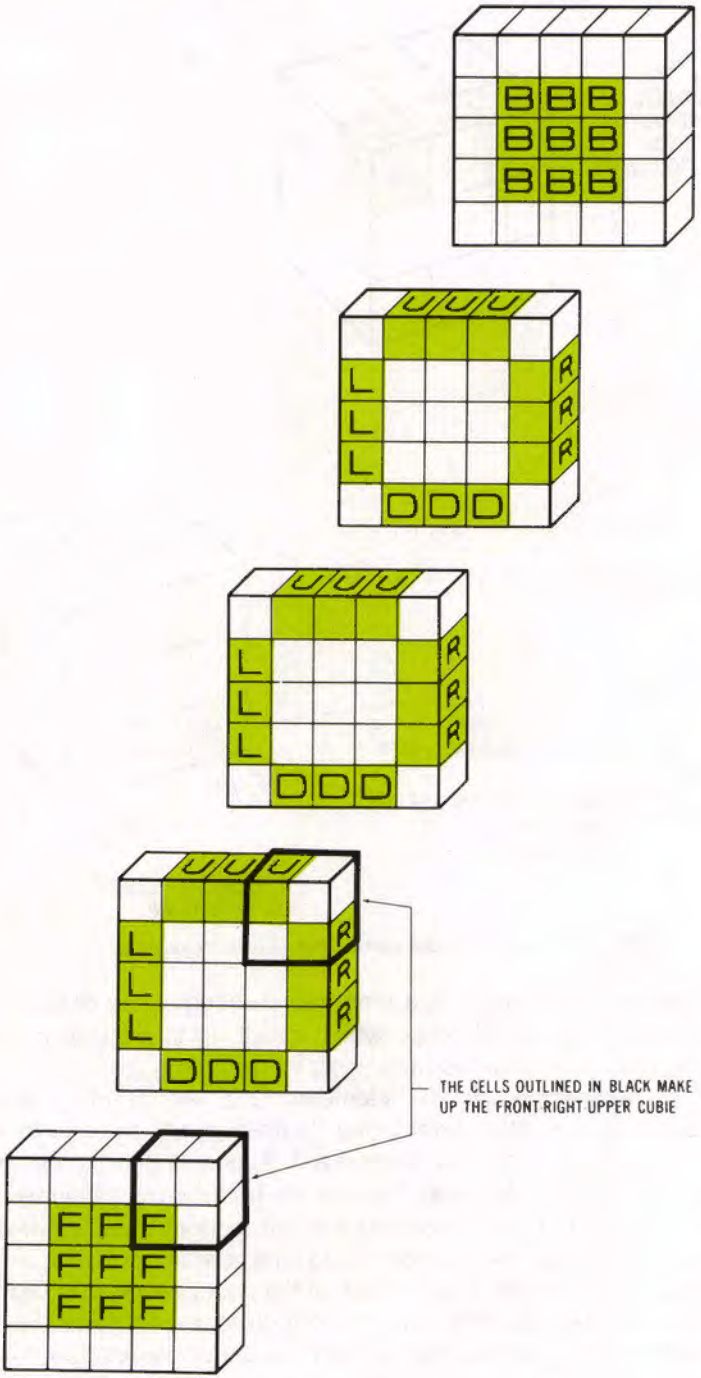
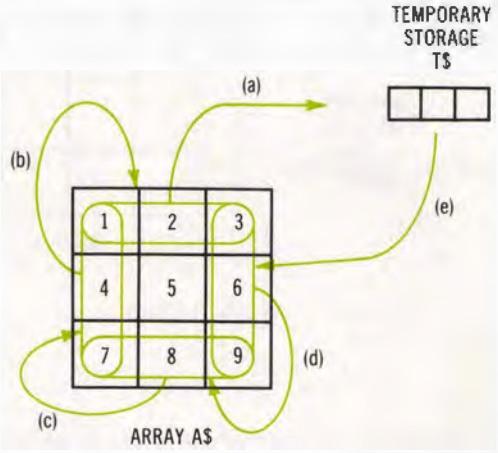


Fig. 7-8. Using a 5 by 5 by 5 array to represent the cube in the computer program.




```
(a) 200 FOR J = 1 TO 3
    210 T$ (J) = A$(J,1)
    220 NEXT J
(b) 230 FOR J = 1 TO 3
    240 A$ (J,1) = A$(1,4-J)
    250 NEXT J
(c) 260 FOR J = 1 TO 3
    270 A$ (1,J) = A$ (J,3)
    280 NEXT J
(d) 290 FOR J = 1 TO 3
    300 A$ (J,3) = A$ (3,4-J)
    310 NEXT J
(e) 320 FOR J = 1 TO 3
    330 A$ (3,J) = T$ (J)
    340 NEXT J
```

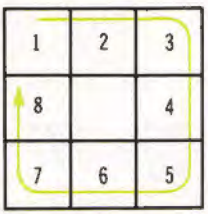
Fig. 7-9. Rotating a 3 by 3 portion of the array.

Representing the Cube With Strings

Let's try another approach, just to see how it works out. Think of each face of the cube as being represented, not by array elements, but by a single string variable. The question then is, which facelets will correspond to which characters in the string? If we make a clever decision about how to do this, it might make the rest of the program easier. Since we'll be rotating the faces of the cube, it makes sense to arrange the string so that rotations are as easy as possible. In this case, we'll try winding the string around the outside cells of the face, like a snake chasing its tail. That looks like it might be easy to rotate.

Something to note here is that the middle cell of each face never moves, no matter how you twist its face or any of the other faces; so, we don't need to worry about rotating the six middle cells of the cube or even about representing them in the computer program. For this reason our string only needs to be eight characters long. Fig. 7-10 shows how we use a string to represent a face of the array and also shows a program that will rotate the face one-quarter turn clockwise. The colored lines represent the "snake chasing his tail." Note that the first character in the string is always in the same place (the top left-hand corner) even though the characters which make up the string (numbers in this case) move around.

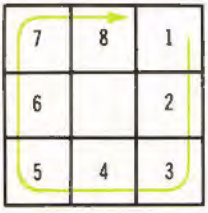
Compare this program with the one in Fig. 7-9 which rotated the face using arrays. The program using strings is much shorter. Using LEFT$ and RIGHT$ statements we simply took the last two characters of the string and put them at the beginning of the string, then took the first six characters and put them at the end. This is equivalent to rotating the face of the cube, since, as far as the program is concerned, the string and the face of the cube are the same thing. The BASIC string functions (RIGHT$, MID$, etc.) are very powerful in certain situations—in this case, their ability to operate on groups of characters at one



F$ = "12345678"

(A) Before rotation.

ROTATING
1/4 TURN
CLOCKWISE
GIVES



F$ = "78123456"

(B) After rotation.

```
10 F$="12345678"
20 PRINT F$
30 F$=RIGHT$(F$,2)+LEFT$(F$,6)
40 PRINT F$

>RUN
12345678
78123456
READY
> -
```

This statement is all that is needed to perform the rotation.

Fig. 7-10. Using a string to represent a face of the cube.

time, instead of a single character at a time as we must do with arrays, makes our rotation routine much shorter and easier to understand.

Based on this comparison, it seems reasonable to conclude that using strings to represent the cube will probably be easier than using an array, so that is what we will do.

"Bottom Up" Program Design

The discussions above, on how to represent the cube on the screen and on whether to use arrays or strings to represent the cube in our program, are an example of what is called "bottom up" program design. By bottom up we mean that we start with the details of the program, such as actual lines of code, rather than with generalities. Writing brief sections of code, as we have done above, lets us explore how well certain approaches are likely to work out. However, there is a danger that if we concentrate too hard on such details we will lose sight of the overall design of the program. And writing a solid, error-free program in a minimum amount of time is actually more dependent on the overall program design than it is on the details of coding. Usually in planning a program you need to think both about how to actually write the code (Bottom Up planning) and, at the same time, about the overall program design (which is called Top Down planning). But, the emphasis should be on overall program design. We'll discuss that next.

Flowcharting and "Top Down" Program Design

"Top Down" simply means that we start planning the program with the most general concepts we can think of and work our way down to the specific details. These general concepts that we start with should be expressed in plain English and should not be dependent on the actual details of writing the program. Let's see how to go about this.

We'll assume that we want the program to display a picture of the cube, and that we also want the program to get input from the keyboard about what face of the cube the user wants to rotate. Which of these shall we do first? It seems like it would be nice to start with a picture of the cube in its initial, unmessed-up state. So now we have two things we want to do, and we can put them in order as shown in Fig. 7-11.

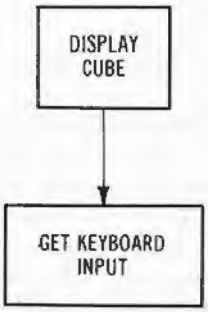


Fig. 7-11. A simple flowchart.

What we've drawn here is a simple flowchart. You already learned how flowcharts represent programs in Chapter 3; here we will show how they can be used as an aid in program design.

Once the program finds out which face the user wants to rotate, it should actually carry out the rotation. Let's assume that the user will specify which face is to be rotated by typing its initial: F, B, U, D, L, or R. Adding this to our flowchart, it now appears as shown in Fig. 7-12.

As you recall from Chapter 3, we put decisions in diamond-shaped boxes to distinguish them from other program actions. The "yes" and "no" outside the decision diamond show which way program control will go depending on the result of the decision in the box.

We'll also assume that we want the program to be able to: (1) print out the sequence of moves we've made so far; (2) go back a move if we want; and (3) restore the cube to its original state. Specifically, if the user types "P" we'll print out all the moves; if she types "S" (for "subtract") we'll go back one move (that is, "undo" the last move on the list and remove it from the move list); and if she types an exclamation point (!), we'll reinitialize the cube to its original state. (We chose the exclamation point because it is unlikely to be typed by mistake. It might be depressing to have your cube reinitialized just when you were in the middle of a complex series of moves!) Let's add these functions to the flowchart as shown in Fig. 7-13.

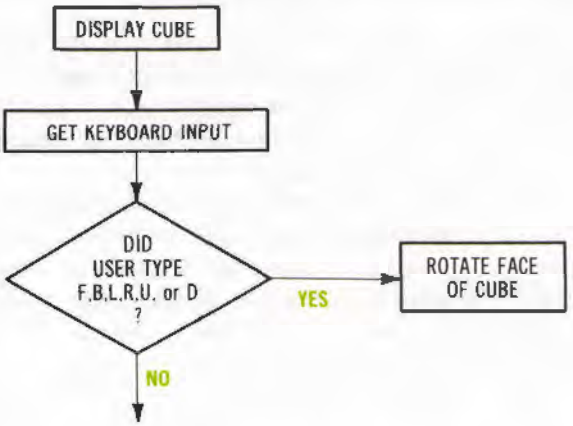


Fig. 7-12. Additions to simple flowchart of Fig. 7-11.

Once we've performed a particular function, we probably want to go back and display the cube again, then wait for further input from the keyboard. Also, before we display the cube for the first time, there will probably be some variables to set to certain initial values, and so on, as there usually are at the beginning of a program. And finally, if the user types something the program doesn't understand, it should print an error message. We have included these functions in Fig. 7-14.

Fig. 7-14 shows, in a general way, what we want the program to do. It is the "master" (most important and most general) flowchart for the program, and we will refer to it many times as we write the program. A diagram like this is extremely helpful in the creation of a program. You should always draw one before you begin writing any but the simplest programs.

Where it seems practical, it is also good to break up the main sections of the program—which are represented by boxes in the master flowchart—into additional flowcharts. In this case, most of the boxes represent functions that are already so simple they can't be broken down much more. However, we already know that there are 6 faces of the cube, and that each face will need at least one separate routine to rotate it, so we can show this in a box which is the "inside view" of the "Rotate Faces" box in the master flowchart. This is shown in Fig. 7-15.

In fact, we can even go a little farther in this direction. When you rotate a face of the cube a quarter turn you have two choices of which way to turn it—clockwise or counterclockwise. Turning it counterclockwise a quarter turn is the same as turning it three quarter turns clockwise, but doing something three times is tedious, so we'll allow the user to make two different kinds of moves—clockwise, which we'll represent by a plus sign (+), and counterclockwise, which we'll represent by a minus sign (−). (We could introduce another move, two quarter turns clockwise—which is the same as two quar-

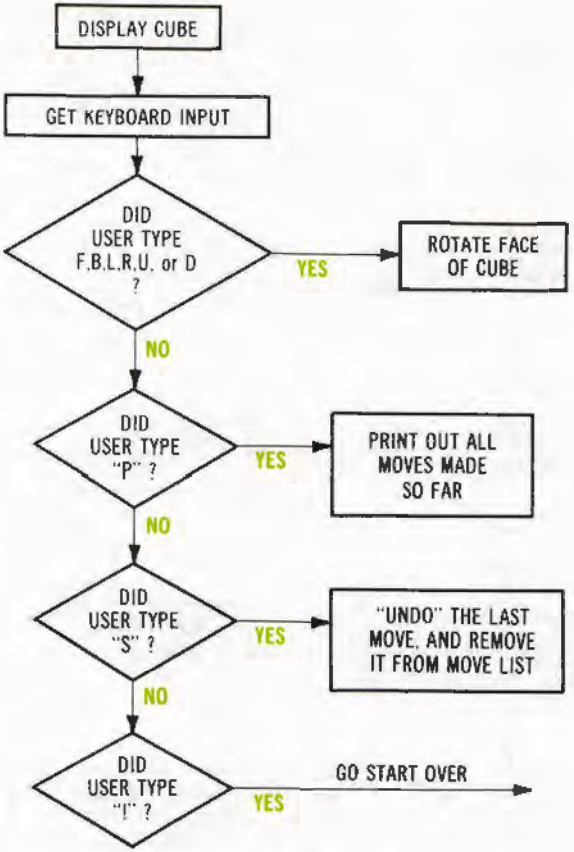


Fig. 7-13. More additions to flowchart.

ter turns counterclockwise—but the gain here isn't so great, and it complicates the program, so we'll leave it out.)

In order to rotate a face of the cube, the user will type the letter representing the face, followed by a plus or minus sign. So F+ means to rotate the front face one quarter turn clockwise, L− means to rotate the left face one quarter turn counterclockwise, and so forth.

Now we can finish the flowchart for the "rotate faces" part of our diagram. Remembering that a quarter turn clockwise is the same as three quarter turns counterclockwise, we'll have the program simply do three clockwise rotations whenever the user asks for one counterclockwise rotation. This way we can cut the number of rotation routines we need in half. The result is shown in Fig. 7-16.

Notice how we have used a variable SF to "remember" whether the user typed a plus or a minus sign. This makes for a simpler program than including a separate input routine for the plus or minus sign in each of the six separate routines for the six faces.

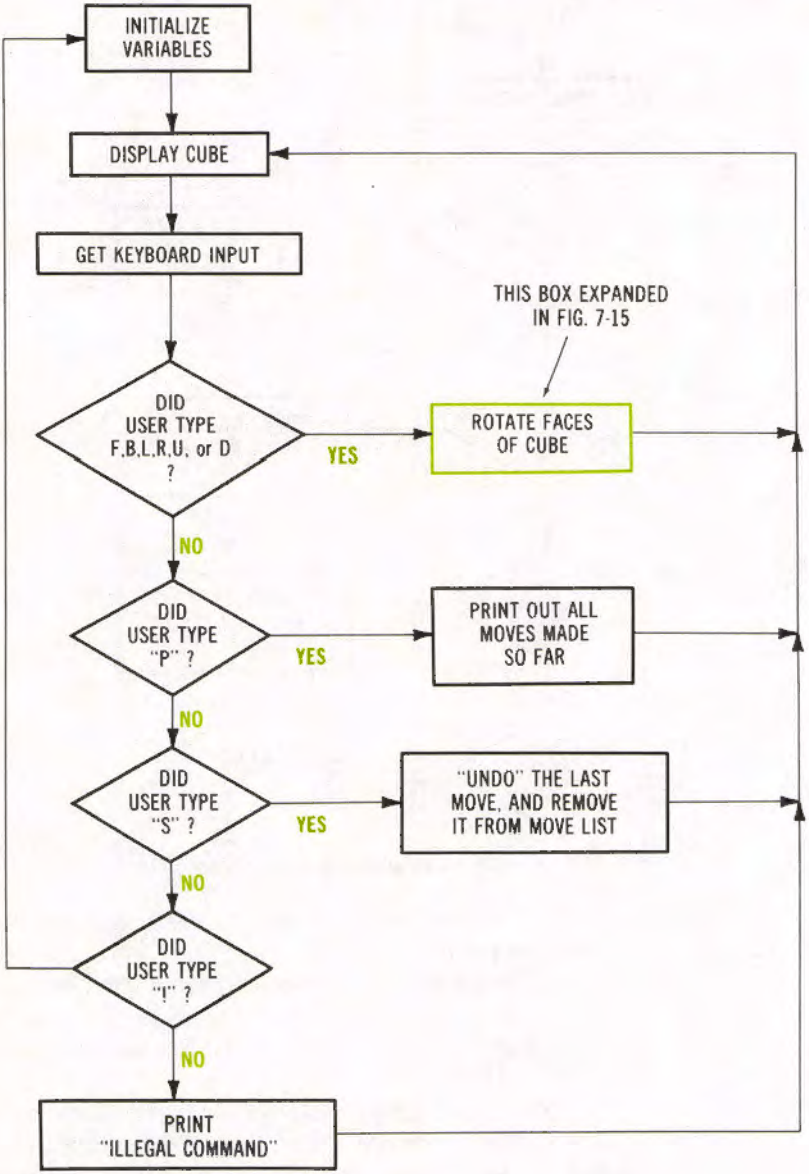


Fig. 7-14. The master flowchart.

Let's "walk through" the flowchart with a specific example. Suppose the user types "F+." Looking back to the master flowchart in Fig. 7-14, we see that after initializing variables and displaying the cube, the program will wait for the user to type a letter. If this letter is F, B, L, R, U or D the program will

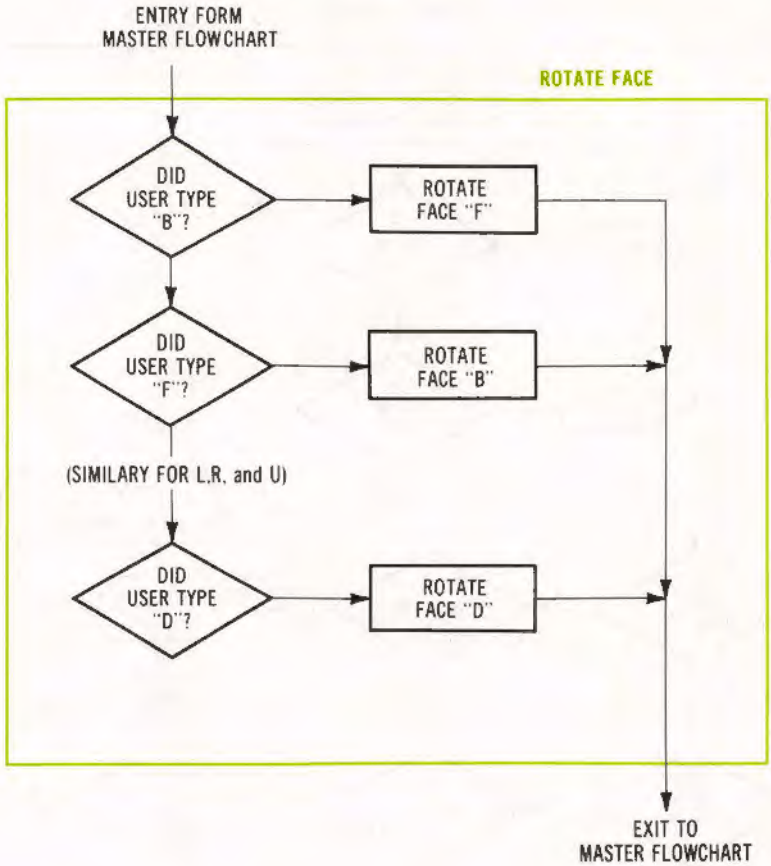


Fig. 7-15. "Inside view" of the "rotate faces" box in master flowchart of Fig. 7-14.

then go to the box labelled "rotate faces of cube." In this case the letter is F, so it goes to this box (which represents, of course, a section of the program). This box is shown in detail in the flowchart of Fig. 7-16, so we continue with the program at the top of this chart. Here we see that the program will wait for a second character to be typed on the keyboard. Since the character is a plus sign, the program will set varible SF to 1, and then go on to figure out exactly which letter was typed. Since it was F, it will take the "yes" exit from the "F" diamond, and then encounter the "is SF=1" diamond, from which it will also take the "yes" exit. This leads to the box labelled "rotate face F once," which is just what we wanted to have happen. The program then exits from the "rotate faces" section of the flowchart, and returns to the master flowchart, where it goes back to display the cube and wait for further keyboard input. Try "walking through" the flowchart using a different input, like D–.

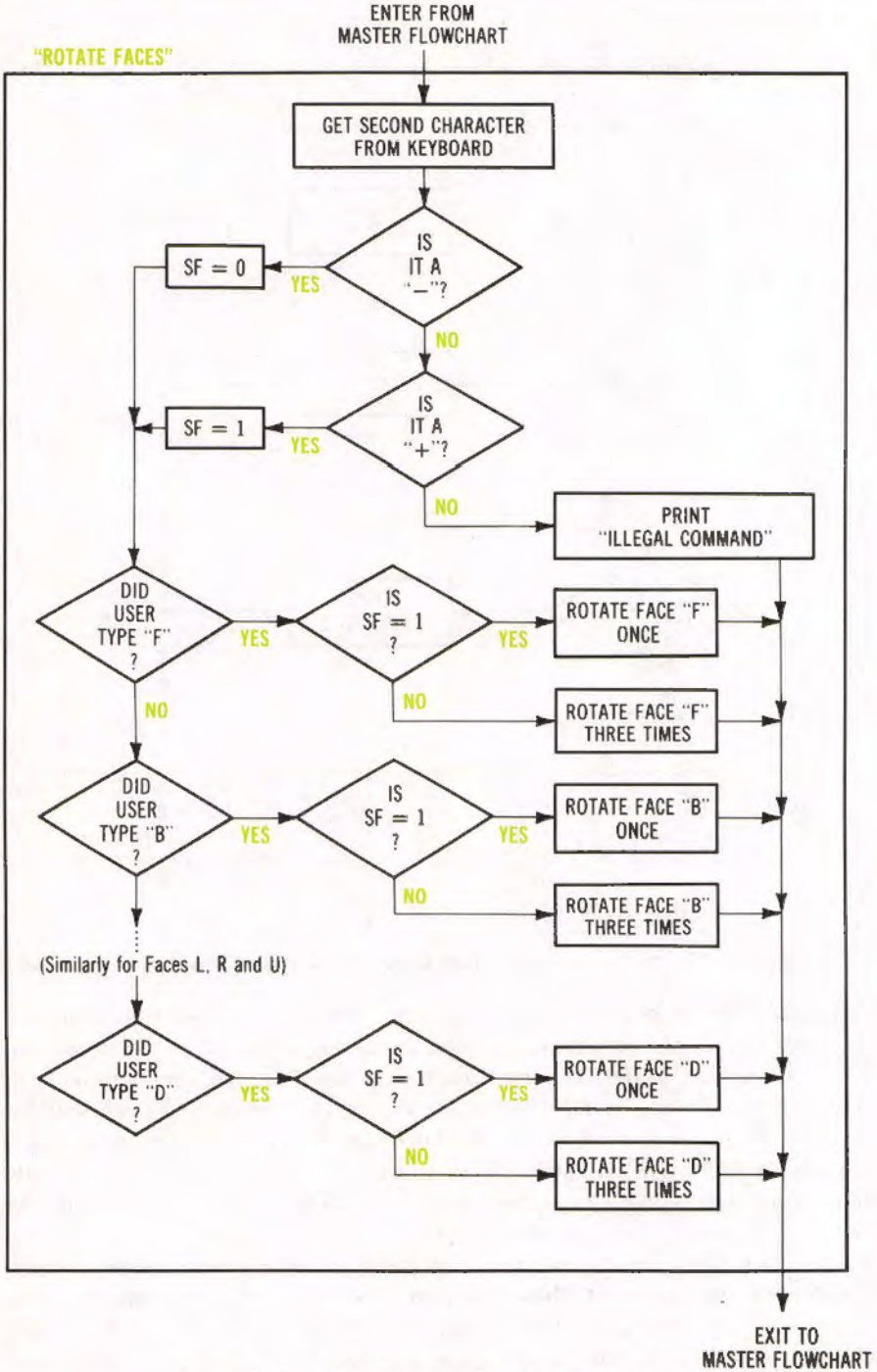


Fig. 7-16. Details of the "rotate faces" program section.

WRITING THE PROGRAM

We are now ready for the next step in the programming process, which is (at last!) actually writing sections of code to correspond to the boxes in our flowcharts. In general, we will try to use subroutines to correspond to each box in our flowchart. This will make it easier to keep clear in our minds what is happening.

The first thing we'll do is to write the main part of the program. The program lines in the "Main Program" will correspond closely to the boxes in the master flowchart. The function of the main program is simply to link together the other parts of the program, such as the "display" section and the "rotate faces" section. The simpler we keep the main part of the program, the easier it will be to see how all the other parts are related to each other.

The Main Program

Fig. 7-17 shows the main program. What the main program mostly does is to call subroutines. First (line 160), it calls a subroutine to display the cube. Then, it gets a character from the keyboard, using the INKEY$ function, in lines 170 and 180. (For a description of how INKEY$ is used, see Chapter 6.) If the key is an exclamation point, the program (in line 190) will simply go back to the beginning at line 30, reinitialize itself, and start again. (There is a complete listing of the program in Fig. 7-22 but don't look at it yet. It might seem a little too forbidding if you're not prepared for it. Remember the tortoise and the hare, and take it slow and easy. You'll get there!)

```
140 ' ---------MAIN PROGRAM---------                    
150 '                                                    
160 GOSUB 640 '                                          DISPLAY
170 A$=INKEY$ : IF A$="" THEN GOTO 170 '                 GET INPUT
180 PRINT A$;                                            
190 IF A$="!" THEN GOTO 30 '                             RESTART
200 IF A$="F" OR A$="B" OR A$="L" OR A$="R" OR           
A$="U" OR A$="D" THEN GOSUB 270 : GOTO 160 '             ROTATE FACE
210 IF A$="S" THEN GOSUB 380 : GOTO 160 '                GO BACK ONE
220 IF A$="P" THEN GOSUB 490 : GOTO 160 '                PRINT MOVES
230 GOSUB 1450 : GOTO 160 '                              ILLEGAL INPUT
```

Fig. 7-17. The main program.

At this point you might have a question: if we haven't written the subroutines yet, how do we know what line numbers to give them? How do we know, for instance, to write "GOSUB 270" when we want to call the "rotate face" subroutine in line 200? The answer to this, of course, is that after writing the program we used a RENUM function to renumber all the program lines so that they would be a uniform 10 apart. (RENUM is a function which is available in some versions of BASIC or on some operating systems, which permits you to renumber some or all of the lines in your program. It can be very useful for "cleaning up" the line number sequence in a program after it

has been developed, as we have done here, and it is also helpful when you find you absolutely have to put 10 new lines of code between, say, line 600 and line 602.)

When we first sat down to write the main program we used some conservative guesses about how big the subroutines were going to be and assigned them numbers accordingly. Specifically, we guessed that no subroutine would be longer than 100 lines, so we started the main program at 10, the first subroutine at 1000, the second subroutine at 2000, the third at 3000, and so on. This way we could start writing the more general parts of the program first, and still know where the subroutines (which are less general and had not yet been written) would be located. After the program was written and debugged, we renumbered it.

But let's get back to our discussion of the main program.

If the user did not type an exclamation point, the program will branch to different subroutines, depending on what character was typed by the user. In line 200, if the user types any of the letters that represent faces of the cube (F,B,L,R,U,D) control goes to a subroutine to rotate the face.

If the user types an "S" (line 210) a subroutine will be executed which has the effect of "undoing" the last move the user made. (We'll explain how this works in the next section.) If the user types a "P" (line 220) the program will branch to a subroutine (at line 490) which will print out all the moves that have been made so far. And if the computer doesn't recognize the character the user typed, it branches to a subroutine at 1450 which prints "ILLEGAL COMMAND." After executing any of these subroutines, the program always returns to the beginning of the main program at line 160.

That's all there is to the main program! Notice how each part of it corresponds to a box in the master flowchart. The rest of the overall program consists of subroutines that are called from the main program, and some initial definitions and statements at the beginning of the program.

The Move List

We've mentioned the subroutines, activated by typing "S" and "P," to go back a move, and to print out all the moves made so far. How do these routines work? They both operate on a string array, initialized with a DIM SK$(200) statement, which we'll call the "move list." Whenever we perform a rotation of a cube face (a "move"), the program stores that move on the move list. If, for example, the user types "D+," the program will not only rotate the "down" face of the cube a quarter turn clockwise, it will also add the string "D+" to the string array SK$. A pointer, which is a numeric variable L, points to the next available cell of SK$, so that the program knows where to put the move. L is incremented after each move.

If the user types "P," the program will branch to a subroutine to print out all the moves on the move list from 1 to L.

If the user types "S," the program will branch to a subroutine which will do the inverse of the last move on the list and then "remove" that move from the

list by decrementing the pointer L. For example, if the last move the user typed in was R+, and the user now types "S," the subroutine at 380 will take the last move on the move list (R+), change the sign of the move from R+ to R−, perform an R− on the cube, and finally remove the R+ from the move list by executing an L=L−1. Turning any face of the cube a quarter turn clockwise and then a quarter turn counterclockwise, leaves it the way it was before the first move was performed. Executing the move and then executing it again with the sign changed has the same effect.

We'll discuss the operation of the move list in more detail as we encounter the various sections of the program.

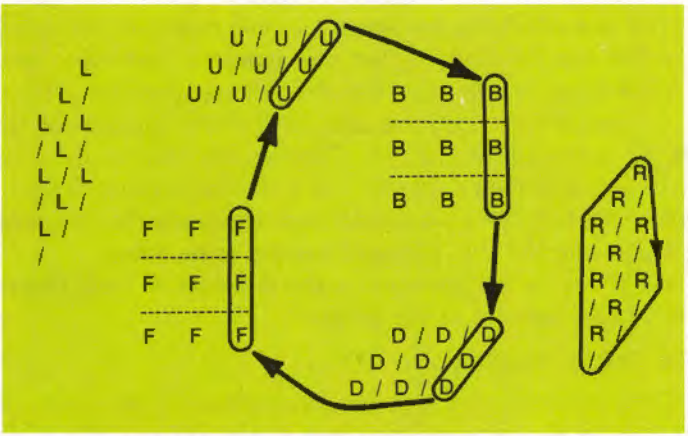
The Display Routine

Unfortunately, the time has now come to bite the bullet. The display routine (as you can see if you look ahead to Fig. 7-20), is by far the messiest-looking part of the whole program. A casual glance at it will probably convince you that you'll never be able to understand it. Fortunately, it is not really that difficult, if you take it one step at a time. We need to get the display routine working before we start work on the rotation routines, so that we can use it to see if the rotation routines are doing what they're supposed to.

First, let's look at the actual screen display in Fig. 7-18. In Fig. 7-18A, we show the cube in its initial state, and in Fig. 7-18B, we show it after the move R+. Notice how all the edges adjacent to the R face have moved one face clockwise from their original positions. (Compare these views with Fig. 7-5 to see how the cube is "exploded" to arrive at this picture.)

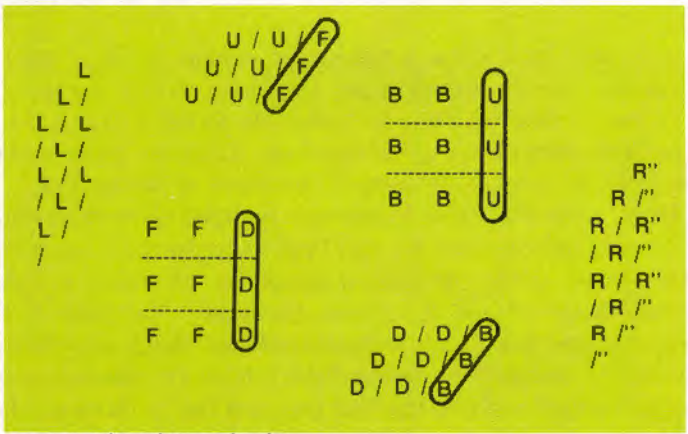
Now, while this exploded view may make sense to us humans (because in our visual imaginations we can "put the cube back together" when we look at this view), to the computer it seems like a hopeless jumble of symbols and spaces strewn all over the screen. Computers like things to be in orderly lists or arrays, and this view is certainly not that. That's why the display program is so messy looking. Because the display, from the computer's point of view, has no regularities, we find that our program has to plot each facelet separately, without the help of the FOR...NEXT loops or subroutines which we might have used if we had chosen a display with a more regular pattern. (Even so, the advantages of the "exploded" display more than make up for the messiness of the display routine.)

In order to write the display routine we need to agree on how to number the facelets on each face, so both we and the computer will know what to call each facelet. Look back at Fig. 7-10. Here we numbered the cells of a 3 by 3 face clockwise, from 1 to 8, starting at an arbitrary corner. (Remember that we don't need to number the center facelet, since the center facelets never change their positions.) Following this procedure, we will now number all six faces of the cube, starting at arbitrarily chosen corners. The result is shown in Fig. 7-19. This is an important diagram, since it shows us how to refer to every single facelet on the cube (number 3 on face B, for example).



Rotating the "R" face

(A) Cube in initial state.



Rotating the "R" face moves the adjacent facelets on the F, U, B, and D faces to new positions.

(B) Cube after move R+.

Fig. 7-18. Display of cube on computer screen.

Before we can display the cube we need to set all the facelets to their initial values—that is, we want each face to be all the same "color" (that is, have the same letter, like F or R). As we discussed earlier, we will not actually use colors, but letters which stand for the name of the face where a particular facelet started out. Initially then, all the facelets on the front or "F" face will be assigned a letter "F," all the facelets on the right face will be assigned a letter "R," and so forth. Later, when faces of the cube have been rotated, the R facelets may not all be on the R face, the D facelets may not all be on the D

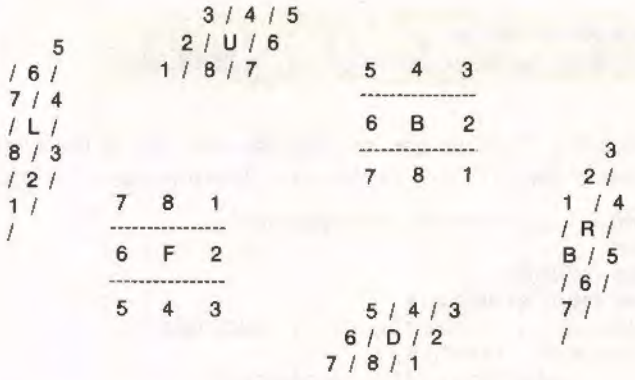


Fig. 7-19. Numbering of facelets on the cube.

face, and so on, just as the original colors move to new faces on a real cube when it is rotated.

The program sets the initial values of the facelets in program lines 70 to 120, which look like this:

```
 50 ' ------INITIALIZE CUBE FACE STRINGS------
 60 '
 70 F$=STRING$(8,"F")
 80 B$=STRING$(8,"B")
 90 L$=STRING$(8,"L")
100 R$=STRING$(8,"R")
110 U$=STRING$(8,"U")
120 D$=STRING$(8,"D")
130 '
```

Now we're ready to attack the actual code of the display routine. In the first line of the routine, line 640, we position the cursor at the upper left-hand corner of the screen, without clearing the screen, using a PRINT@ 0, " "; statement. (This statement works on a TRS-80. Many computers use other statements to position the cursor without clearing the screen, such as "LOCATE" or "HPOS" and "VPOS." If you adapt this program for your computer, you should use whatever statement works with your version of BASIC.) We could simply clear the screen at this point, but repositioning the cursor without clearing the screen makes for a display that changes more smoothly from one move to the next.

Refer to the display in Fig. 7-18 and the numbering system in Fig. 7-19. The display is printed in lines on the screen, using PRINT statements, just as if it were text. Each program line contains one PRINT statement and corresponds to a line on the screen display. The first PRINT statement, in line 650, is the first line to be printed on the screen. Here it is:

```
650 PRINT STRING$ (14, " ")
MID$(U$,3,1) " / " MID$(U$,4,1) " / " MID$(U$,5,1)
```

From Fig. 7-18 we can see that the first line of the display contains three facelets of the "U" face of the cube. These facelets are in the row of the "U"

```
620 '-----------DISPLAY ROUTINE-----------
630 '
640 PRINT@ 0, " ";
650 PRINT STRING$(14, " ")
MID$(U$,3,1) " / " MID$(U$,4,1) " / " MID$(U$,5,1)
660 PRINT " " MID$(L$,5,1)
"      " MID$(U$,2,1) " / U / " MID$(U$,6,1)
670 PRINT " " MID$(L$,6,1) " / "
MID$(U$,1,1) " / " MID$(U$,8,1) " / " MID$(U$,7,1) "    "
MID$(B$,5,1) "    " MID$(B$,4,1) "  " MID$(B$,3,1)
680 PRINT MID$(L$,7,1) " / " MID$(L$,4,1)
STRING$(18, " ") STRING $(9,"-")
690 PRINT "/ L /" STRING$(18," ")
MID$(B$,6,1) "  B  " MID $(B$,2,1)
700 PRINT MID$(L$,8,1) " / " MID$(L$,3,1)
STRING$(18, " ") STRING$(9,"-") STRING$(7," ")
MID$(R$,3,1)
710 PRINT "/ " MID$(L$,2,1) " /" STRING$(18," ")
MID$(B$,7,1) "   " MID$(B$,8,1) "   " MID$(B$,1,1) "   "
MID$(R$,2,1) " /"
720 PRINT MID$(L$,1,1) " / "
MID$(F$,7,1) "    " MID$(F$,8,1) "    " MID$(F$,1,1)
STRING$(20," ")
MID$(R$,1,1) " / " MID$(R$,4,1)
730 PRINT"/      " STRING$(9,"-") STRING$(20," ") "/ R /"
740 PRINT STRING$(6," ")
MID$(F$,6,1) " F " MID$(F$,2,1)
STRING$(20," ")
MID$(R$,8,1) " / " MID$(R$,5,1)
750 PRINT STRING$(6, " ") STRING$(9,"-") STRING$(20," ")
"/ " MID$(R$,6,1) "/ "
760 PRINT STRING$(6," ")
MID$(F$,5,1) "    " MID$(F$,4,1) " " MID$(F$,3,1)
STRING$(6," ")
MID$(D$,5,1) " / " MID$(D$,4,1) " / " MID$(D$,3,1 "    "
MID$(R$,7,1) " /"
770 PRINT STRING$(18, " ")
MID$(D$,6,1) " / D / " MID$(D$,2,1)
STRING$(8, " ") "/"
780 PRINT STRING$(15," ")
MID(D$,7,1) " / " MID$(D$,8,1) " / " MID$(D$,1,1)
790 PRINT STRING$(40,"-")
800 PRINT "COMMAND: ";
810 RETURN
```

Fig. 7-20. Display routine listing.

face which is closest to the "B" face, and from Fig. 7-19 we can see that these facelets are numbered 3, 4, and 5. So, in program line 650 we first print a string of 14 spaces, to move the cursor over to where the "U" face is on display. Then, using the MID$ function, we print the contents of the third character of the string U$, (the string U$ represents the "U" face) and then we print a space, a slash, and another space to separate this facelet from the next one. Similarly, we print the contents of the fourth and fifth characters of the string U$; with slashes between them. (For a review of how MID$ works, see Chapter 4.)

That's really all there is to the display routine. Each line in the display is transformed into a separate PRINT statement, in the same way we described for line 650. The complete listing for the display routine is shown in Fig. 7-20. Many lines are longer than line 650, because on the display they cover parts of two or even three faces. Line 670, for example, prints facelet 6 from face "L," facelets 1, 8, and 7 from face "U," and facelets 5, 4, and 3 from face "B."

This routine is an example of what might be called the "brute force" method of programming. It is not particularly elegant or concise, but it gets the job done, slowly munching its way through the display one line at a time.

Getting Ready to Rotate

If the main program discovers that the user has typed one of the letters that corresponds to a face of the cube, it immediately branches to the "Get Direction to Rotate Face" subroutine at line 270. This subroutine is actually the "outermost" or highest of three levels of subroutines that we will use in rotating the faces. Here it is:

```
250 '-------GET DIRECTION TO ROTATE FACE-------
260 '
270 A2$=INKEY$ : IF A2$="" THEN GOTO 270 '      GET SIGN
280 PRINT A2$;
290 IF A$="-" THEN SF=0 : GOTO 320 '            SET SIGN FLAG
300 IF A2$="+" THEN SF=1 : GOTO 320 '
310 GOSUB 1450 : RETURN '                       ILLEGAL INPUT
320 L=L+1 '                                     INCREMENT POINTER
330 GOSUB 550 '                                 GO ROTATE
350 '
```

The purpose of this subroutine is to figure out which way the user wants to rotate the cube face. It does this in an INKEY$ statement and assigns the character typed in to the variable A2$. This happens in lines 270 and 280. If you want a clockwise rotation you type "+." When the routine finds a "+" it sets the SF variable (for "sign flag") to one, in line 290. If you want a counterclockwise rotation, you type "−," and the routine sets SF to zero, in line 300. If you don't type either of these two characters, the routine branches to

the "ILLEGAL COMMAND" routine and returns. If "+" or "−" was typed, the routine increments the pointer L, in line 440, which shows how many "moves" there have been so far, and then calls the rotation subroutine, which is located at line 550.

Look back at Fig. 7-16. It is an expanded view of the "Rotate Faces" box from the master flowchart. As you recall, the idea here was that one counterclockwise rotation is the same as three clockwise rotations. We can translate this diagram into a subroutine which consists of the series of program statements from line 550 to line 600. Each of the lines in this routine deals with a separate face; otherwise, the lines operate in exactly the same way. For example, here is line 550:

```
550 IF A$="F" THEN
IF SF=1 THEN GOSUB 850 : SK$(L)="F+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 850 : NEXT J :
        SK$(L)="F−" : RETURN
```

This line first uses an IF statement to find out if the letter typed in was an "F." If it was not, control goes to the next line. If it was an "F," the control goes to another IF statement on the same line (see the discussion of nested IF statements in the ELSE section of Chapter 6). This second IF statement determines whether SF is 0 or 1, that is, whether you requested clockwise or counterclockwise rotation. If SF is 1, you want clockwise rotation and we execute the THEN part of the statement, which has two parts. The first part goes to the rotation subroutine at 850, which carries out the actual rotation of the "F" face. The second part puts the string "F+" into the array SK$, which holds the list of moves made so far.

If you requested counterclockwise rotation, the ELSE part of the inner IF statement is executed. This also consists of two parts. The first part is for the FOR. . . NEXT loop which counts from 1 to 3. Inside the loop is the GOSUB to the Rotate Face "F" routine at line 850. So, this routine will be called three times, thus rotating the face clockwise three times, which is the same as rotating it counterclockwise once. The second part of the ELSE statement then puts the string "F−" into the array SK$.

After all this we then RETURN to the "Get Direction" routine, and from there RETURN immediately to the main program. The statements from line numbers 560 to 600 perform the same function for the other five faces of the cube. This part of the program is shown in Fig. 7-21.

Notice that the routine from line 550 to line 600 is the second level of subroutine in the "Rotate Faces" part of the master flowchart. The first level, as we mentioned above, is the routine in which we get the "+" or "−" sign, from line 270 to 340. The first level subroutine calls the second level, and the second level calls the third level, which is composed of the six routines, from

```
530 '-------FIND FACE TO BE ROTATED AND CALL ROUTINES-------
540 '
550 IF A$="F" THEN
IF SF=1 THEN GOSUB 850 : (SK$(L)="F+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 850 : NEXT J :
        SK$(L)="F-" : RETURN
560 IF A$="B" THEN
IF SF=1 THEN GOSUB 950 : SK$(L)="B+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 950 : NEXT J :
        SK$(L)="B-" : RETURN
570 IF A$="L" THEN
IF SF=1 THEN GOSUB 1050 : SK$(L)="L+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1050 : NEXT J :
        SK$(L)="L-" : RETURN
580 IF A$="R" THEN
IF SF=1 THEN GOSUB 1150 : SK$(L)="R+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1150 : NEXT J :
        SK$(L)="R-" : RETURN
590 IF A$="U" THEN
IF SF=1 THEN GOSUB 1250 : SK$(L)="U+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1250 : NEXT J :
        SK$(L)="U-" : RETURN
600 IF A$="D" THEN
IF SF=1 THEN GOSUB 1350 : SK$(L)="D+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1350 : NEXT J :
        SK$(L)="D-" : RETURN
610 '
```

Fig. 7-21. Program listing of six rotation subroutines.

line 950 to 1410, which actually carry out the rotation for each of the six faces. We'll describe this third or "lowest" subroutine level next.

The Rotation Routines

Now, at last, we are finally getting to the rotation of the cube faces. Starting from the "top down," we have reached the lowest level: that is, the most specific, least general part of the program design. Here we will write the code to actually carry out the rotation operations. We are, in fact, at the same place we were when we investigated the details of actual coding in the "bottom up" approach.

Let's review what is necessary to carry out a rotation. As an example, imagine that we are going to rotate the "R" face of the cube one quarter turn clockwise. Refer here to Fig. 7-19. This diagram, if you study it carefully, should make it clear what we need to do to rotate each face of the cube. When we rotate the "R" face, not only will the facelets on face "R" each move two positions clockwise, but the following four other transfers will take place on the adjacent faces:

(1) "F" facelets 1, 2, 3 will go to the "U" face, positions 5, 6, 7.
(2) "D" facelets 1, 2, 3 will go to the "F" face, positions 1, 2, 3.

(3) "B" facelets 1, 2, 3 will go to the "D" face, positions 1, 2, 3.
(4) "U" facelets 5, 6, 7 will go to the "B" face, positions 1, 2, 3.

Now, how can we get the program to do all this? We've already written a little routine to rotate the 8 facelets on the "R" face itself, as shown in Fig. 7-10. (This routine has no effect on the adjacent faces.) Since we're going to be using MID$ statements a lot, we'll translate the program line in Fig. 7-10 which actually carries out the rotation into a form using MID$ statements instead of LEFT$ and RIGHT$:

```
R$ = MID$(R$,7,2) + MID$(R$,1,6)
```

What about the transfers of the facelets on the adjacent faces, in steps (1) to (4) above? Let's write a statement that will do the transfer of the three facelets shown in step (1). That is, move the contents of position 1 on the "F" face to position 5 on the "U" face, move the contents of position 2 on the "F" face to position 6 on the "U" face, and move the contents of position 3 on the "F" face to position 7 on the "U" face. Here's the statement:

```
MID$(U$,5,3) = MID$(F$,1,3)
```

Step 1.

Isn't that easy? The use of string-handling functions lets us move three facelets in a single statement. (If the use of MID$ on the left side of the equal

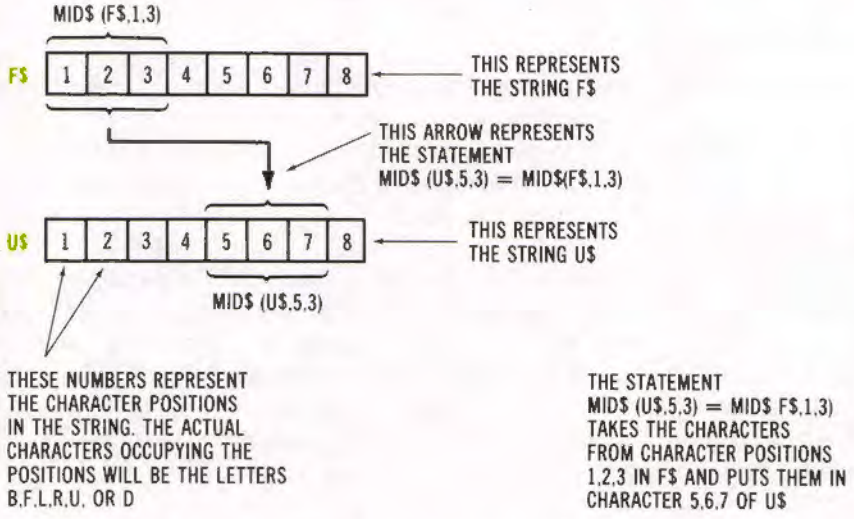


Fig. 7-22. Transferring three characters from string F$ to string U$.

sign is not familiar to you, read the appropriate section in Chapter 6.) What this statement does is to transfer three characters from the string F$, starting at position 1, into three character spaces in string U$, starting at position 5.

The diagram in Fig. 7-22 shows how three characters (representing three facelets) are transferred from string F$ to string U$.

If the version of BASIC on your computer does not include the use of MID$ on the left side of the equal sign, you can translate these statements into ones that use MID$ only on the right. These equivalent statements are more cumbersome, but they will work just as well. For example, suppose you have the statement (from line 1290) using MID$ on the left:

```
MID$(B$,3,3)=MID$(L$,5,3)
```

We can rewrite this as:

```
B$=MID$(B$,1,2)+MID$(L$,5,3)+MID$(B$,6,3)
```

The diagram in Fig. 7-23 shows how this works.

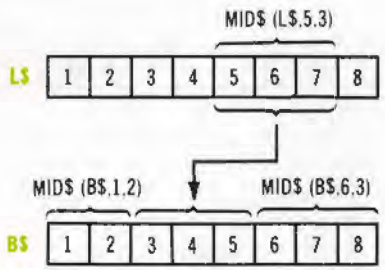


Fig. 7-23. Converting MID$ statements.

Now (except for one small problem) we can write similar statements to do the other transfers in steps (2), (3) and (4):

```
MID$(F$,1,3)=MID$(D$,1,3)
MID$(D$,1,3)=MID$(B$,1,3)
MID$(B$,1,3)=MID$(U$,5,3)
```

Step 2.
Step 3.
Step 4.

The small problem is this: if, in step (1), we go right ahead and write the contents of the F facelets 1, 2, and 3 into the U positions 5, 6, and 7, we will be destroying the contents of these variables; that is, we will no longer know whether the U positions 5, 6, and 7 contained R's, B's, L's, or what, since we will have "written over" them with the contents of the F positions. So, in step (4), when we are ready to transfer the contents of these U positions into B positions 1, 2, and 3, we wouldn't really be transferring the contents of the U positions; we'd be transferring the contents of the F positions. This would be a major foulup.

To avoid this problem the first thing we do is to save the contents of these three F positions in a "temporary" string variable called T$, instead of writing them to the U positions. Then, when the other steps have been finished and the contents of the U positions have been safely transferred to the B positions, we put the contents of T$ back into the U positions. A diagram of the process is shown in Fig. 7-24.

It's like four people (call them F, U, B and D) sitting in four chairs arranged in a circle. If they all decide to move one place to the right, someone (say F)

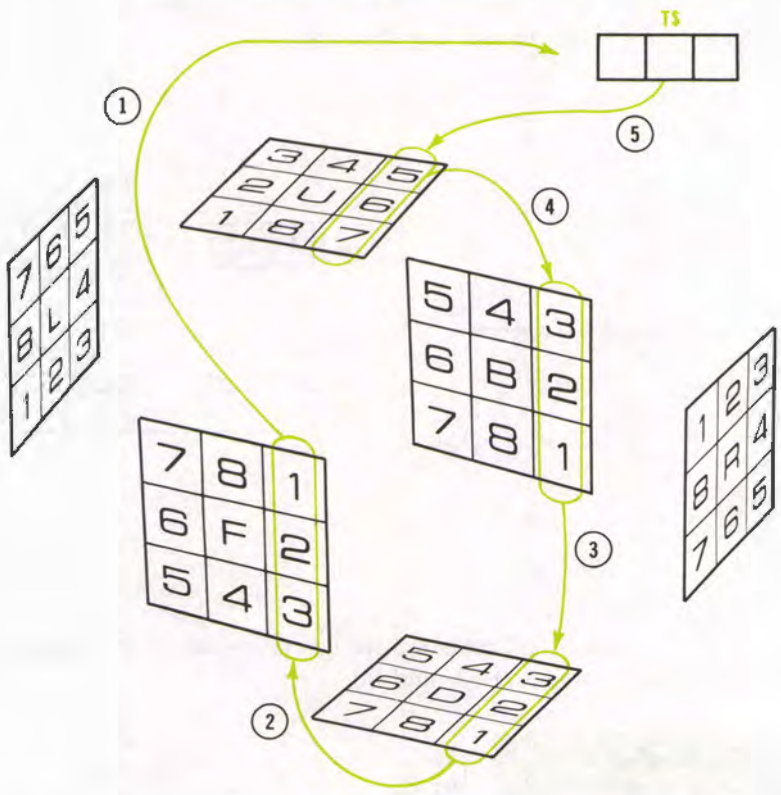


Fig. 7-24. Saving contents of "F" positions in a temporary string variable, T$.

will have to stand up. Then the person on F's right (who is D) moves into F's chair, the person on D's right (who is B) moves into D's chair, the person on B's right (who is U) moves into B's chair, and finally F, who has been in temporary storage (standing up) can sit down again in the vacant chair, which is U's.

The program steps to do this look like this:

```
1130 '---------ROTATE FACE "R"---------
1140 '
1150 R$=MID$(R$,7,2)+MID$(R$,1,6)
1160 T$=MID$(F$,1,3)
1170 MID$(F$,1,3)=MID$(D$,1,3)
1180 MID$(D$,1,3)=MID$(B$,1,3)
1190 MID$(B$,1,3)=MID$(U$,5,3)
1200 MID$(U$,5,3)=T$
1210 RETURN
1220 '
```

Rotate face "R" itself.

Rotate adjacent faces.

That's all we have to do to rotate the R face, and it is almost all we have to do for the other five faces. The only additional problem arises when we have to do a transfer which involves the three character positions 7, 8, and 1 (as in the upper row on the front face, for example). Since two of these characters are on the end of the string and one is on the beginning, the transfer will require two different equalities using MID$ statements, one to transfer positions 7 and 8, and the other to transfer position 1.

For instance, to transfer D positions 7, 8, and 1 to L positions 7, 8, and 1, we would have to write:

```
MID$(L$,7,2)=MID$(D$7,2) : MID$(L$,1,1)=MID$(D$,1,1)
```

What happens in line 870 in the section to rotate face F is diagrammed in Fig. 7-25.

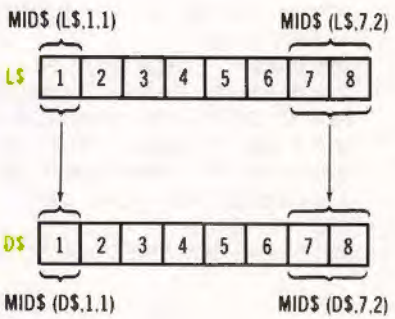


Fig. 7-25. Transferring "D" positions to "L" positions.

Similarly, line 1270, in the section to rotate face U, puts the contents of the R positions 1, 2, and 3, into F positions 7, 8, and 1. This is shown in Fig. 7-26.

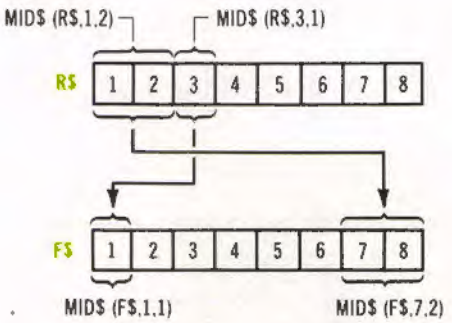


Fig. 7-26. Transferring "R" positions to "F" positions.

```
MID$(F$,7,2)=MID$(R$,1,2) : MID$(F$,1,1)=MID$(R$,3,1)
```

The rotation of all six faces is carried out in a similar way by the routines from line 830 to 1410. (Refer to the listing of the entire program in Fig. 7-27.)

```
 10 '--------INITIALIZE--------
 20 '
 30 CLS : CLEAR 1000 : DIM SK$(200) : L=0
 40 '
 50 ' -------INITIALIZE CUBE FACE STRINGS-------
 60 '
 70 F$=STRING$(8,"F")
 80 B$=STRING$(8,"B")
 90 L$=STRING$(8,"L")
100 R$=STRING$(8,"R")
110 U$=STRING$(8,"U")
120 D$=STRING$(8,"D")
130 '
140 '---------MAIN PROGRAM---------
150 '
160 GOSUB 640 '                                        DISPLAY
170 A$=INKEY$ : IF A$="" THEN GO TO 170 '               GET INPUT
180 PRINT A$;
190 IF A$="!" THEN GOTO 30 '                            RESTART
200 IF A$="F" OR A$="B" OR A$="L" OR A$="R" OR
A$="U" OR A$="D" THEN GOSUB 270 : GOTO 160 '            ROTATE FACE
210 IF A$="S" THEN GOSUB 380 : GOTO 160 '               GO BACK ONE
220 IF A$="P" THEN GOSUB 490 : GOTO 160 '               PRINT MOVES
230 GOSUB 1450 : GOTO 160 '                             ILLEGAL INPUT
240 '
250 '---------GET DIRECTION TO ROTATE FACE---------
```

Fig. 7-27. Complete Rubik's Cube program listing.

```
260 '
270 A2$=INKEY$ : IF A2$="" " THEN GOTO 270 '                GET SIGN
280 PRINT A2$;
290 IF A2$="-" THEN SF=0 : GOTO 320 '                       SET SIGN FLAG
300 IF A2$="+" THEN SF=1 : GOTO 320
310 GOSUB 1450 : RETURN '                                   ILLEGAL INPUT
320 L=L+1 '                                                 INCREMENT POINTER
330 GOSUB 550 '                                             GO ROTATE
340 RETURN
350 '
360 ' ---------DO INVERSE OF LAST MOVE ON LIST----------
370 '
380 A$=SK$(L) '                                             GET MOVE FROM LIST
390 IFL<1 THEN GOSUB 1450 : RETURN '                        LIST EMPTY
400 IF RIGHT$(A$,1)="-" THEN SF=1                           SET FLAGS TO INVERSE
410 IF RIGHT$(A$,1)="+" THEN SF=0
420 A$=LEFT$(A$,1) '                                        GET FACE
430 GOSUB 550 '                                             GO ROTATE
440 L=L-1 '                                                 DECREMENT POINTER
450 RETURN
460 '
470 '---------PRINT MOVE LIST---------
480 '
490 CLS : IF L>0
THEN FOR K=1 TO L : PRINT SK$(K)"   "; : NEXT K
500 LINE INPUT"
PRESS -ENTER- TO CONTINUE"; Z$ : CLS
510 RETURN
520 '
530 '---------FIND FACE TO BE ROTATED AND CALL ROUTINES---------
540 '
550 IF A$="F" THEN
IF SF=1 THEN GOSUB 850 : SK$(L)="F+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 850 : NEXT J :
        SK$(L)="F-" : RETURN
560 IF A$="B" THEN
IF SF=1 THEN GOSUB 950 : SK$(L)='B+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 950 : NEXT J :
        SK$(L)="B=" : RETURN
570 IF A$="L" THEN
IF SF=1 THEN GOSUB 1050 : SK$(L)="L+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1050 : NEXT J :
        SK$(L)="L-" : RETURN
580 IF A$="R" THEN
IF SF=1 THEN GOSUB 1150 : SK$(L)="R+" : RETURN
        ELSE FOR J=1 to 3 : GOSUB 1150 : NEXT J :
        SK$(L)="R-" : RETURN
590 IF A$="U" THEN
IF SF=1 THEN GOSUB 1250 : SK$(L)="U+" : RETURN
        ELSE FOR J=1 TO 3 : GOSUB 1250 : NEXT J :
        SK$(L)="U-" : RETURN
```

Fig. 7-27 —cont. Complete Rubik's Cube program listing.

```
600 IF A$="D" THEN
IF SF=1 THEN GOSUB 1350 : SK$(L)="D+" : RETURN
        ELSE FOR J-1 TO 3 : GOSUB 1350 : NEXT J :
        SK$(L)="D-", : RETURN
610 '
620 '---------DISPLAY ROUTINE---------
630 '
640 PRINT@ 0," ";
650 PRINT STRING$(14," ")
MID$(U$,3,1) " / " MID$(U$,4,1) " / " MID$(U$,5,1)
660 PRINT "      " MID$(L$,5,1)
"     " MID$(U$,2,1) " / U / " MID$(U$,6,1)
670 PRINT "   " MID$(L$,6,1) " /   "
MID$(U$,1,1) " / " MID$(U$,8,1) " / " MID$(U$,7,1) "   "
MID$ (B$,5,1) "   " MID$(B$,4,1) "   " MID$(B$,3,1)
680 PRINT MID$(L$,7,1) " / " MID$(L$,4,1)
STRING$(18, " ") STRING$ (9,"-")
690 PRINT "/ L /" STRING $(18," ")
MID$(B$,6,1) "   B   " MID$(B$,2,1)
700 PRINT MID$(L$,8,1) " / " MID$(L$,3,1)
STRING$(18," ") STRING$(9,"-") STRING$(7," ")
MID$(R$,3,1)
710 PRINT "/ " MID$(L$,2,1) " /" STRING$(18," ")
MID$(B$,7,1) " " MID$(B$,8,1) "   " MID$(B$,1,1) "   "
MID$(R$,2,1) " /"
720 PRINT MID$(L$,1,1) " /   "
MID$(F$,7,1) " " MID$(F$,8,1) "   " MID$(F$,1,1)
STRING$(20," ")
MID$(R$,1,1) " / " MID$(R$,4,1)
730 PRINT"/     " STRING$(9,"-") STRING$(20," ") "/ R /"
740 PRINT STRING$(6," ")
MID$(F$,6,1) "   F   " MID$(F$,2,1)
STRING$(20," ")
MID$(R$,8,1) " / " MID$(R$,5,1)
750 PRINT STRING$(6," ") STRING$(9,"-") STRING$(20," ")
"/ " MID$(R$,6,1) " /"
760 PRINT STRING$(6," ")
MID$(F$,5,1) " " MID$(F$,4,1) "   " MID$(F$,3,1)
STRING$(6," ")
MID$(D$,5,1) " / " MID$(D$,4,1) " / " MID$(D$,3,1)
MID$(R$,7,1) " /"
770 PRINT STRING$(18, " ")
MID$(D$,6,1) " / D / " MID$(D$,2,1)
STRING$(8," ") "/"
780 PRINT STRING$(15," ")
MID$(D$,7,1) " / " MID$(D$,8,1) " / " MID$(D$,1,1)
790 PRINT STRING$(40,"-")
800 PRINT "COMMAND: ";
810 RETURN
820 '
830 '---------ROTATE FACE "F"---------
840 '
```

Fig. 7-27 —cont. Complete Rubik's Cube program listing.

```
 850 F$=MID$(F$,7,2)+MID$(F$,1,6)
 860 T$=MID$(L$,7,2)+MID$(L$,1,1)
 870 MID$(L$,7,2)=MID$(D$,7,2) : MID$(L$,1,1)=MID$(D$,1,1)
 880 MID$(D$,7,2)=MID$(R$,7,2) : MID$(D$,1,1)=MID$(R$,1,1)
 890 MID$(R$,7,2)=MID$(U$,7,2) : MID$(R$,1,1)=MID(U$,1,1)
 900 MID$(U$,7,2)=MID$(T$,1,2) : MID$(U$,1,1)=MID$(T$,3,1)
 910 RETURN
 920 '
 930 '---------ROTATE FACE "B"---------
 940 '
 950 B$=MID$(B$,7,2)+MID$(B$,1,6)
 960 T$=MID$(L$,3,3)
 970 MID$(L$,3,3)=MID$(U$,3,3)
 980 MID$(U$,3,3)=MID$(R$,3,3)
 990 MID$(R$,3,3)=MID$(D$,3,3)
1000 MID$(D$,3,3)=T$
1010 RETURN
1020 '
1030 '---------ROTATE FACE "L"---------
1040 '
1050 L$=MID$(L$,7,2)+MID$(L$,1,6)
1060 T$=MID$(F$,5,3)
1070 MID$(F$,5,3)=MID$(U$,1,3)
1080 MID$(U$,1,3)=MID$(B$,5,3)
1090 MID$(B$,5,3)=MID$(D$,5,3)
1100 MID$(D$,5,3)=T$
1110 RETURN
1120 '
1130 '---------ROTATE FACE "R"---------
1140 '
1150 R$=MID$(R$,7,2)+MID$(R$,1,6)
1160 T$=MID$(F$,1,3)
1170 MID$(F$,1,3)=MID$(D$,1,3)
1180 MID$(D$,1,3)=MID$(B$,1,3)
1190 MID$(B$,1,3)=MID$(U$,5,3)
1200 MID$(U$,5,3)=T$
1210 RETURN
1220 '
1230 '---------RATE FACE "U"---------
1240 '
1250 U$=MID$(U$,7,2)+MID$(U$,1,6)
1260 T$=MID$(F$,7,2)+MID$(F$,1,1)
1270 MID$(F$,7,2)=MID$(R$,1,2) : MID$(F$,1,1)=MID$(R$,3,1)
1280 MID$(R$,1,3)=MID$(B$,3,3)
1290 MID$(B$,3,3)=MID$(L$,5,3)
1300 MID$(L$,5,3)=T$
1310 RETURN
1320 '
1330 '---------ROTATE FACE "D"---------
1340 '
1350 D$=MID$(D$,7,2)+MID$(D$,1,6)
1360 T$=MID$(F$,3,3)
```

Fig. 7-27 —cont. Complete Rubik's Cube program listing.

```
1370 MID$(F$,3,3)=MID$(L$,1,3)
1380 MID$(L$,1,2)=MID$(B$,7,2) : MID$(L$,3,1)=MID$(B$,1,1)
1390 MID$(B$,7,2)=MID$(R$,5,2) : MID$(B$,1,1)=MID$(R$,7,1)
1400 MID$(R$,5,3)=T$
1410 RETURN
1420 '
1430 '---------COMMAND NOT UNDERSTOOD---------
1440 '
1450 PRINT" -- ILLEGAL COMMAND.";
1460 FOR J=1 TO 500 : NEXT J
1470 RETURN
```

Fig. 7-27 —cont. Complete Rubik's Cube program listing.

And that, believe it or not, is really all there is to the whole program! It wasn't so hard, was it? At least it wasn't so complicated as it sounded when we first presented the problem. Understanding—and writing—long programs is largely a matter of breaking them down into separate pieces, each of which is small enough to be easily understood. The hardest, most interesting, and most creative part of programming is not usually writing the actual program lines; it is planning the structure of the program in such a way that each different program function is cleanly separated from the others, yet works together with them to produce the desired result.

The complete program listing, showing how all the parts fit together, is shown in Fig. 7-27. Type it into your computer, and try it out! If you get frustrated trying to solve the cube, you might buy one of the books that describes a systematic approach to finding a solution. A good one is *The Simple Solution to Rubik's Cube,* by James G. Nourse. Using our program with the book can be a lot easier than trying to use a real cube. You have a list of the moves you've made so far, you can go back any number of moves, and, if you're hopelessly lost, you can unscramble the cube in a millisecond with a single push of the "!" key! Happy cubing.



APPENDIX

A

Numbering Systems

"What's one and one and one and one and one and one and one and one and one and one?"
"I don't know," said Alice. "I lost count."
"She can't do addition," said the Red Queen.
(Lewis Carroll, *Through the Looking Glass)*

No one knows when the first number was recorded, but most likely it dates back to Biblical times. Among the oldest system of numbers was that of the Chinese, which was first based on a system of laying sticks in patterns and later was based on symbols drawn with pen and ink (Fig. A-1).



(A) Chinese "stick" number system.



(B) Chinese "pen-and-ink number systems.

Fig. A-1. First number systems.

Calculating in these number systems was exceedingly difficult. This was because each time the basic numerals were exceeded, a new numeral had to

be invented. In Roman numerals, when you needed to count above 100, you used a *C*, and above 1000 an *M*. The real problem came when these numbers had to be multiplied. The actual process of counting took place on counting boards, such as the Chinese abacus, where answers were converted back to the notation system.

Our current decimal system is much more streamlined than those of the ancient civilizations. We only have to learn the 10 basic symbols and the positional notation system in order to count to any number. For example, what is the meaning of the number 256? In positional notation, the value of each digit is determined by its position. The four in 4000 has a different value than the 4 in 400. Thus, in 256 we have three digits, and each must be "interpreted" in light of where it is in order and relation to the other digits. We learn that the rightmost digit is interpreted as the number of "ones," the next to the left as the number of "tens," and the next digit as "hundreds." The general formula for representing numbers in the decimal system using positional notation is:

$$a_1 10^{n-1} + a_2 10^{n-2} + \ldots + a_n$$

which is expressed as $a_1 a_2 a_3 \ldots a_n$, where n is the number of digits to the left of the decimal point. Therefore,

$$\begin{aligned} 256 &= (2 \times 10^2) + (5 \times 10^1) + (6 \times 10^0) \\ &= 2 \text{ hundreds} + 5 \text{ tens} + 6 \text{ ones} \end{aligned}$$

In the decimal system we use 10 as the basic multiplier. We call 10 the *base* or *radix*. Most of recorded history shows mankind counting in the decimal system (base 10). However, it is not difficult to imagine a race of one-armed people who used the quinary system (base 5). We see examples of the duodecimal system in clocks, rulers, the dozen, and so on.

THE BINARY SYSTEM

Although the seventeenth-century German mathematician Leibnitz was given most of the credit for invention of the binary number system with a base of 2, it was probably the ancient Chinese who realized the simple and natural way of representing numbers as powers of 2.

Early computers used relays and switches as their basic elements. The operation of a switch or a relay is itself binary in nature. A switch can either be on (1) or off (0). Modern computers use transistors like those found in televisions and radios. These components can be arranged to be in one or two "states": on or off. As a matter of fact, the more distinctly different the two states, the more reliable the computer's operation.

The idea is to make the devices work in such a manner that even slight changes in their characteristics will not affect the operation. The best way of doing this is to use a *bistable device,* which has two states.

If a bistable device is in stable state X, an energy pulse will drive it to state Y; and if the bistable component is in stable state Y, an energy pulse will drive it to state X. It is easy for a bistable component to represent the number 0 or 1:

stable state X = 1
stable state Y = 0

Counting

The same type of positional notation used in the decimal system is used in the binary. Since there are only two possible states for a numeral, either we count the position value or we don't count it. The general rule is: The binary number $a_1a_2a_3 \ldots a_n$ is expressed in decimal as:

$$a_1 2^{n-1} + a_2 2^{n-3} + \ldots + a_n$$

Therefore, the binary number 11010 is converted to decimal as follows:

$$\begin{aligned} N &= a_1 2^{5-1} + a_2 2^{4-1} + a_3 2^{3-1} + a_4 2^{2-1} + a_5 2^{1-1} \\ &= a_4 16 + a_2 8 + a_3 4 + a_4 2 + a_5 1 \end{aligned}$$

Substituting the values for a_1, a_2, a_3, a_4, and a_5:

$$\begin{aligned} 11010 &= (1 \times 16) + (1 \times 8) + (0 \times 4) + (1 \times 2) + (0 \times 1) \\ &= 16 + 8 + 0 + 2 + 0 \\ &= 26 \text{ (decimal system)} \end{aligned}$$

Table A-1 lists the first 20 binary numbers.

Table A-1. The First 20 Binary Numbers

Decimal	Binary	Decimal	Binary
1	1	11	1011
2	10	12	1100
3	11	13	1101
4	100	14	1110
5	101	15	1111
6	110	16	10000
7	111	17	10001
8	1000	18	10010
9	1001	19	10011
10	1010	20	10100

A simpler way to convert binary numbers to decimal is to use a weighting table (Fig. A-2). This is simply a reduction of the expansion formula just presented. Write down the value of the positions in the binary number over the binary digits, arrange them as an addition, and add them.

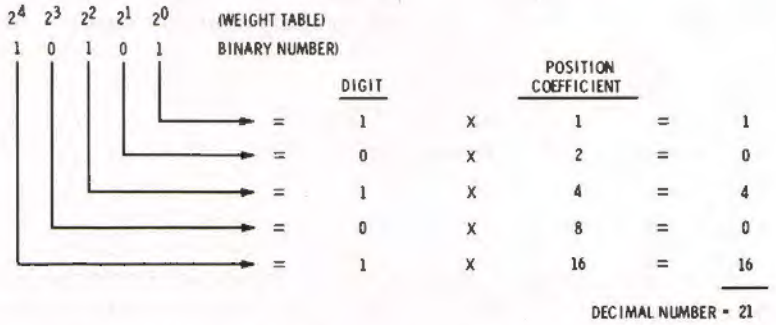


Fig. A-2. Binary-to-decimal conversion using the weighting method.

Frequently we will want to convert in the opposite direction, from decimal to binary. For this method we repeatedly divide the decimal number by 2, and the remainder after each division is used to indicate the coefficients of the binary number to be formed. Fig. A-3 shows the conversion of 47_{10} to binary. Note that decimal 47 is written 47_{10} and that binary numbers are given the subscript 2 if there is danger of confusing the number systems.

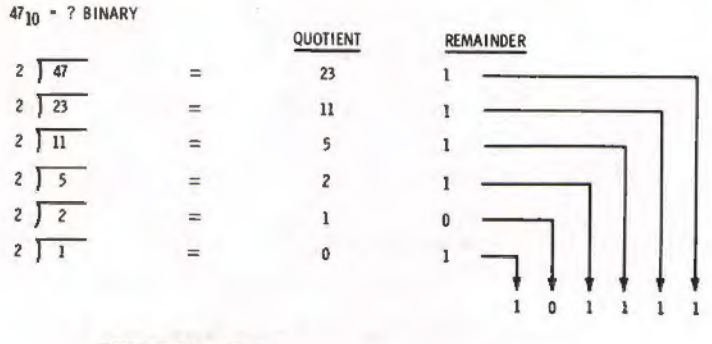


Fig. A-3. Decimal-to-binary conversion using the division method.

Fractional numbers are treated in the same manner as in the decimal system. In the decimal system:

$$0.128 = (1 \times 10^{-1}) + (2 \times 10^{-2}) + (8 \times 10^{-3})$$

In the binary system:

$$0.101 = (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$$

Binary Addition and Subtraction

Addition in binary is as easy as addition in decimal, and follows the same rules. In adding decimal 1 + 8, we get a sum of 9. This is the highest-value digit. Adding 1 to 9 requires that we change the digit back to 0 *and carry 1*. Similarly, adding binary 0 + 1, we reach the highest-value binary digit, 1. Adding 1 to 1 requires that we change the 1 back to a 0 and carry 1, i.e., 1 + 1 = 10. Thus, for example, add binary 101 to 111:

$$\begin{array}{r} 101_2 = 5_{10} \\ +\ 111_2 = 7_{10} \\ \hline 1100_2 = 12_{10} \end{array}$$

The four rules of binary addition are:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0, \text{ carry } 1 \end{aligned}$$

Here are some examples:

101	5	11.01	3¼
+ 110	6	101.11	5¾
1011	11	1001.00	9

Subtraction is just inverted addition. It is necessary to establish a convention for subtracting a large digit from a small digit. This condition occurs in binary math when we subtract a 1 from 0. The remainder is 1, and we borrow 1 from the column to the left. Just as in decimal subtraction, if the digit on the left is a 1, we make it a zero, and if it's a zero, we make it a 1. The rules for binary subtraction are:

$$\begin{aligned} 0 - 0 &= 0 \\ 1 - 0 &= 1 \\ 1 - 1 &= 0 \\ 0 - 1 &= 1, \text{ borrow } 1 \end{aligned}$$

Here are two examples:

10000	16	110.01	6¼
− 11	− 3	− 100.1	− 4½
1101	13	1.11	1¾

Binary Multiplication and Division

There are only four basic multiplications to remember in the binary system, instead of the usual 100 we memorize in the decimal system. The binary multiplication table is:

$$0 \times 0 = 0$$
$$1 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 1 = 1$$

The following examples illustrate how easy binary multiplication is compared with decimal. The rule to remember is: "copy the multiplicand if the multiplier is a 1, and copy all 0's if the multiplier is a 0. Then add down, as in decimal multiplication."

Binary	*Decimal*	*Binary*	*Decimal*
1100	12	1.01	1.25
× 1010	× 10	× 10.1	× 2.5
0000	120	101	625
1100		1010	250
0000		11.001	3.125
1100			
1111000			

Binary division is also very simple. Division by zero is forbidden (meaningless), just as in decimal division. The binary division table is:

$$\frac{0}{1} = 0$$

$$\frac{1}{1} = 1$$

Examples of binary division are:

Binary	*Decimal*
101	5
101)11001	5)25
101	
101	
101	

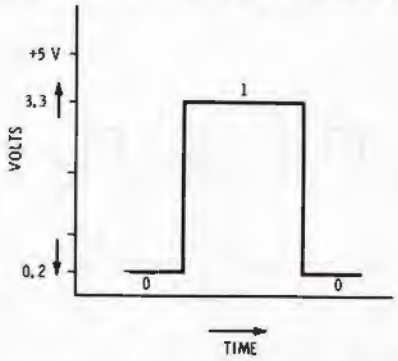
Because of the difficult binary additions and subtractions that result when the numbers are large, octal or hexadecimal notation is often used.

Representing Binary Numbers

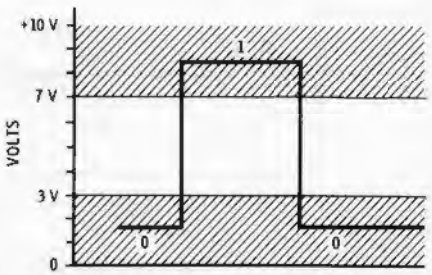
Information in digital computers of today is processed by the switching and storing of electrical signals. Computers operating in the binary number system need represent only one of two values (1 and 0) at a time. A single wire can be utilized for this purpose. A method for representing a binary digit on a signal line is shown in Fig. A-4A. In this method a small positive voltage is used to represent a 0, and a larger positive dc voltage is used to represent a 1.

Much importance is placed on the actual voltage values used to represent the binary digit. Usually, the circuitry used to transmit and receive these signals determines the range of voltages. The most ideal circuit is one in which the two logic levels are far apart (Fig. A-4B).

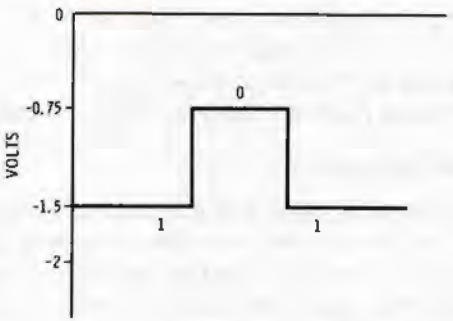
Note that the "1" signal is positive with respect to the "0" signal. This convention could also have been reversed, i.e., the most negative signal called a "1" and the more positive signal a "0". (See Fig. A-4C.) Usually, one convention is chosen by the designer and then used throughout the computer.



(A) Binary digit representation (TTL).



(B) Binary digit representation (CMOS)

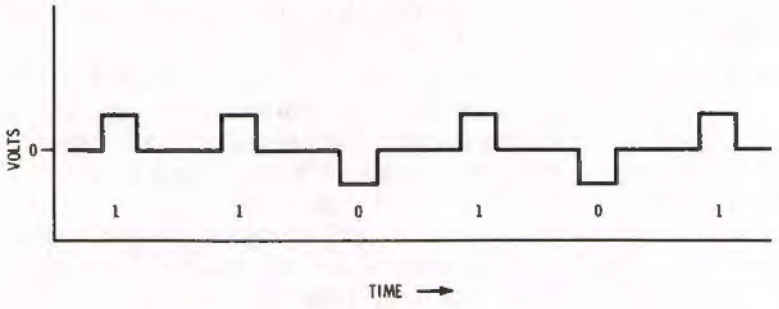


(C) Binary digit representation—negative logic (ECL).

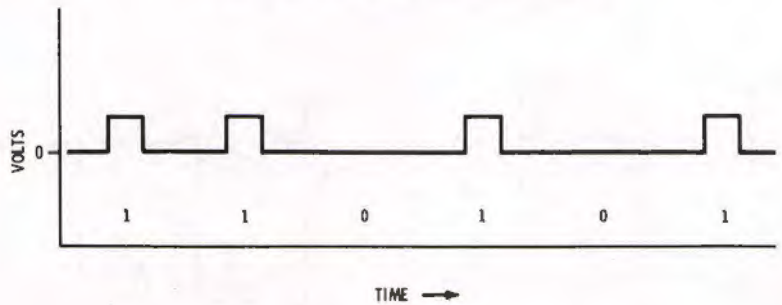
Fig. A-4. Representing binary numbers.

Pulse Representation of Binary Numbers

Binary digits are often transmitted and received as a burst of pulses. Fig. A-5A shows a system in which a positive pulse represents a 1 and a negative pulse a 0. The signal line remains at some in-between value when no pulse is being sent. This technique is used frequently in magnetic recording, and is called *return-to-zero* (RZ) encoding.



(A) RZ method of representing binary digits.



(B) NRZ method of representing binary digits.

Fig. A-5. Pulse representation of binary numbers.

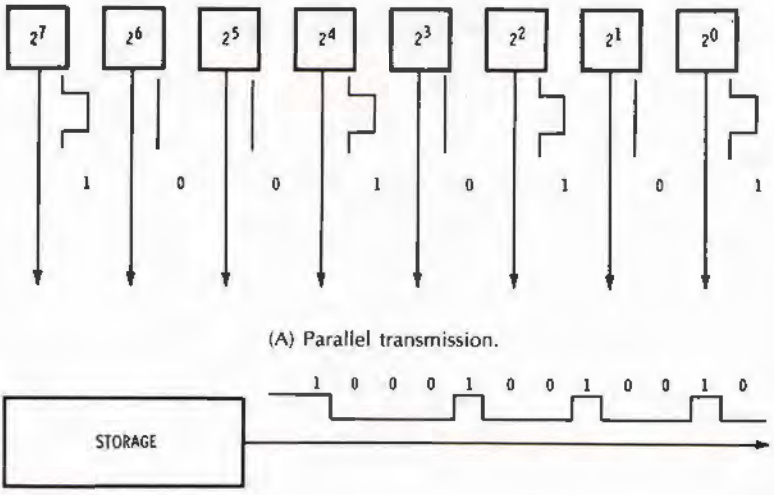
A more popular technique is shown in Fig. A-6B. A 1 is represented by a pulse, and a 0 as no pulse. The receive circuitry must keep in synchronization with the incoming signal in order to know when a binary digit is occurring. This technique is called *non-return-to-zero* (NRZ) encoding.

Serial and Parallel Transmission

So far, methods of representing and transmitting a single binary digit have been illustrated. We will find that it is often necessary to transmit complete binary numbers, which is accomplished by transmitting each binary digit over its own wire. Thus, an *n*-digit binary number would require *n* wires or signal lines. This is called *parallel transmission*. Fig. A-6A illustrates an 8-bit binary number (10010101) being transmitted over eight parallel lines. In such a system each line is assigned a different weight, based on the positional notation

of the binary number system. The leftmost binary digit is assigned the weight of 2^{n-1}, where n is the number of binary digits (8 in this case).

The other method of transmitting binary data is called *serial transmission*. In this method the signals representing the binary digits are transmitted one at a time in sequence, usually starting with the rightmost digit (Fig. A-6B). This method requires some synchronization in order to distinguish several 0's or 1's that follow each other in a sequence.



(A) Parallel transmission.

(B) Serial transmission.

Fig. A-6. Parallel and serial transmission.

Negative Numbers

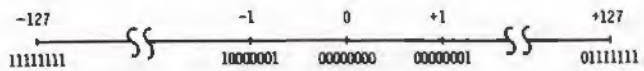
The normal way to express a negative number is to place a minus sign in front of the number. When a negative number is subtracted from a positive number, we *change the sign and add*. For example, $256 - (-128) = 256 + 128 = 384$.

Digital computers use binary storage devices to store and represent binary digits. Seven such devices can represent the binary numbers from 0000000 to 1111111 (0 to 127_{10}). However, if we wish to increase the range to include the negative numbers from 0000000 to −1111111, we need another binary digit, or bit. This bit is called the *sign bit* and is placed in front of the most significant digit of the binary number.

The convention for the sign bit is: If the sign bit is 0, the number is positive; and if the sign bit is a 1, the number is negative. The remaining digits form the absolute value of the number. This numerical storage mode is called *signed binary*. Fig. A-7A shows signed binary numbers from +127 to −127, and the signed binary number line is shown in Fig. A-7B.

INTEGER	SIGNED BINARY CODE							
	s	b_7	b_6	b_5	b_4	b_3	b_2	b_1
+127	0	1	1	1	1	1	1	1
+126	0	1	1	1	1	1	1	0
↓				↓				
↓				↓				
↓				↓				
+3	0	0	0	0	0	0	1	1
+2	0	0	0	0	0	0	1	0
+1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
-1	1	0	0	0	0	0	0	1
-2	1	0	0	0	0	0	1	0
-3	1	0	0	0	0	0	1	1
↓				↓				
↓				↓				
↓				↓				
-126	1	1	1	1	1	1	1	0
-127	1	1	1	1	1	1	1	1

(A) Seven-bit magnitude table.



(B) Signed binary number line (seventh-bit magnitude).

Fig. A-7. Signed binary code.

Signed binary, although frequently used, has a few minor flaws that make it less flexible than other codes for negative numbers. Any arithmetic operation requires checking the sign bit and then either adding or subtracting the numerical values, based on the signs.

The Use of Complements

The use of complemented binary numbers makes it possible to add or subtract binary numbers using only circuitry for addition. To see how negative numbers are used in the computer, consider a mechanical register, such as a car mileage indicator, being rotated backwards. A five-digit register approaching and passing through zero would read as follows:

00005
00004
00003
00002
00001
00000
99999
99998
99997
etc.

It should be clear that the number 99998 corresponds to −2. Furthermore, if we add

$$\begin{array}{r} 00005 \\ +\ 99998 \\ \hline 1\ 00003 \end{array}$$

and ignore the carry to the left, we have effectively formed the operation of subtraction: 5 − 2 = 3.

The number 99998 is called the *ten's complement* of 2. The ten's complement of any decimal number may be formed by subtracting each digit of the number from 9, and then adding 1 to the least significant digit of the number formed. For example:

normal subtraction	*ten's complement subtraction*	
89	89	89
− 23	− 23	+ 77
66		→1 66
		DROP CARRY

Two's Complement

The two's complement is the binary equivalent of the ten's complement in the decimal system. It is defined as that number which, when added to the original number, will result in a sum of zero, ignoring the carry. The following example points this out:

1101	number
0011	two's complement
→1 0000	sum
IGNORE CARRY	

The easiest method of finding the two's complement of a binary number is to first find the one's complement, which is formed by setting each bit to the opposite value:

11011101	number
00100010	one's complement

The two's complement of the number is then obtained by adding 1 to the least significant digit of the one's complement:

11011101	number
00100010	one's complement
+1	add one
00100011	two's complement

The complete signed two's complement code is obtained for negative numbers by using a 1 for the sign bit, and two's complement for the magnitude of the number. Fig. A-8A shows the signed two's complement code, and its number line is shown in Fig. A-8B.

In contrast to the signed binary code, in the signed two's complement code, numbers can be added without regard to their signs and the result will always be correct. The following examples should make this clear:

0000101	5	1111011	−5	1111011	−5
+1111110	+ (−2)	+ 0000010	+ (+2)	+1111110	+ (−2)
1 0000011	3	11111101	(−3)	1 1111001	(−7)
IGNORE				IGNORE	

Notice that it is impossible to add +64 to +64 in a 7-bit code and +128 to +128 in an 8-bit code. Also note that in comparing the two systems, signed binary and two's complement, the largest negative two's complement number that can be represented in 8 bits is −128, while in signed binary it's −127. Changing a negtive integer from signed binary to two's complement requires simply complementing all bits except the sign bit, and adding 1.

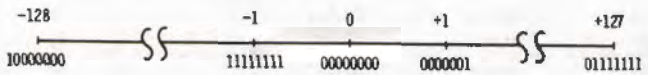
Binary-Coded Number Representation

Since computers operate in the binary number system, while people use the decimal system, it was only natural that some intermediate system be developed. Computers, and some calculators and "intelligent" instruments, use a *binary-coded decimal* system. In such systems, a group of binary bits is used to represent each of the 10 decimal digits.

The binary-coded decimal (bcd) system is called a "weighted binary code" with the weights 8, 4, 2, and 1, as shown in Table A-2. Notice that 4 binary bits are required for each decimal digit, and that each digit is assigned a weight: the leftmost bit has a weight of 8; the rightmost bit a weight of 1.

INTEGER	CODE SIGNED 2's COMPLEMENT							
	s	b_7	b_6	b_5	b_4	b_3	b_2	b_1
+127	0	1	1	1	1	1	1	1
+126	0	1	1	1	1	1	1	0
⋮				⋮				
+3	0	0	0	0	0	0	1	1
+2	0	0	0	0	0	0	1	0
+1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
−1	1	1	1	1	1	1	1	1
−2	1	1	1	1	1	1	1	0
−3	1	1	1	1	1	1	0	1
⋮				⋮				
−126	1	0	0	0	0	0	1	0
−127	1	0	0	0	0	0	0	1
−128	1	0	0	0	0	0	0	0

(A) Seven-bit magnitude table.



(B) Two's complement number line.

Fig. A-8. Signed two's complement code.

There's a slight problem with using 4 bits to represent 10 decimal values. Since $2^4 = 16$, the 4 bits could actually represent 16 values. However, the next choice down, 3 bits, allows only 2^3, or 8, possible digits, which is insufficient. To represent the decimal number 127 in bcd, 12 binary bits are required instead of seven if we use pure binary:

1	2	7
0001	0010	0111

The bcd system has another property that makes it less flexible for binary computation in the computer. The difficulty lies in forming complements of its numbers. As was pointed out, it is common practice to perform subtraction by

complementing the subtrahend and adding 1. When the bcd 8-4-2-1 system is used, the complement formed by inverting all the bits may produce an illegal bcd digit. For example, complementing the bcd number 0010 (2_{10}) gives 1101 (13_{10}), which is not a bcd code.

To solve this problem, several other codes have been developed. For example, the *excess-three code* is formed by adding 3 to the decimal number and then forming the bcd code. For example:

$$\begin{array}{rl} 4 & \text{number} \\ +\underline{3} & \text{add for excess-three} \\ 7 & \\ 7 = 0111 & \text{convert 7 to bcd} \end{array}$$

Table A-2 also shows the excess-three codes for the 10 decimal digits. Now the complement of the excess-three code doesn't form any illegal bcd digits, i.e., 10_{10} or above.

The excess-three code is not a weighted code, since the sum of the bits does not equal the number being represented. On the other hand, the bcd 8-4-2-1 code is weighted but forms illegal complements.

A weighted code that does form legal complements is the *2-4-2-1 code* in Table A-2.

Table A-2. Binary-Coded Number Representation

			2-4-2-1 Coded Binary			
	Binary-	Excess-3	Weight of Bit			
Decimal Digit	Coded Decimal	Coded Binary	2	4	2	1
0	0000	0011	0	0	0	0
1	0001	0100	0	0	0	1
2	0010	0101	0	0	1	0
3	0011	0110	0	0	1	1
4	0100	0111	0	1	0	0
5	0101	1000	1	0	1	1
6	0110	1001	1	1	0	0
7	0111	1010	1	1	0	1
8	1000	1011	1	1	1	0
9	1001	1100	1	1	1	1

OCTAL NUMBER SYSTEM

It is probably quite evident by now that the binary number system, although nice for computers, is a little cumbersome for human usage. For example, communicating binary 11011010 over a telephone would be "one-one-zero-one-one-zero-one-zero," which is quite a mouthful. Also, it is easy to make errors when adding and subtracting large binary numbers. The octal

(base 8) number system alleviates most of these problems and is frequently used in the microcomputer literature.

The octal system uses the digits 0 through 7 in forming numbers. Table A-3 shows octal numbers and their decimal equivalents.

Table A-3. First 13 Octal Digits

Decimal	Octal	Binary	Decimal	Octal	Binary
0	0	0	7	7	111
1	1	1	8	10	1000
2	2	10	9	11	1001
3	3	11	10	12	1010
4	4	100	11	13	1011
5	5	101	12	14	1100
6	6	110	13	15	1101

Octal numbers are converted to decimal numbers by using the same expansion formula as that used in binary-to-decimal conversion, except that 8 is used for the base instead of 2.

$$\begin{aligned}(\text{octal})\ 167 &= (1 \times 8^2) + (6 \times 8^1) + (7 \times 8^0)\\ &= (1 \times 64) + (6 \times 8) + (7 \times 1)\\ &= 64 + 48 + 7\\ &= 119\ (\text{decimal})\end{aligned}$$

A *weighting table* (Fig. A-9) is a quick way to convert octal values to decimal.

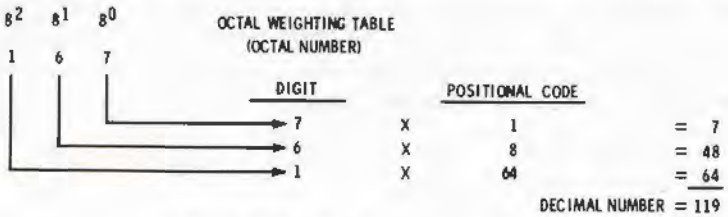
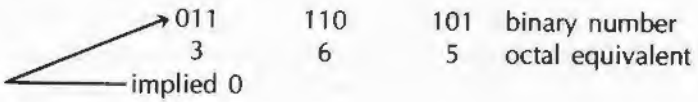


Fig. A-9. Octal-to-decimal conversion.

The primary use of octal is as a convenient way of recording values stored in binary registers. This is accomplished by using a grouping method to convert the binary value to its octal equivalent. The binary number is grouped by threes, starting with the bit corresponding to $2^0 = 1$ and grouping to the left of it. Then each binary group is converted to its octal equivalent. For example, convert 11110101 to octal:



The largest 8-bit octal number is 377_8, and the largest 7-bit octal number is 177_8. Negative octal numbers in 8-bit signed two's complement cover 377_8 (-1_{10}) to 200_8 (-128_{10}).

Conversion from decimal to octal is performed by repeated division by 8 and using the remainder as a digit in the octal number being formed. Fig. A-10 illustrates this method.

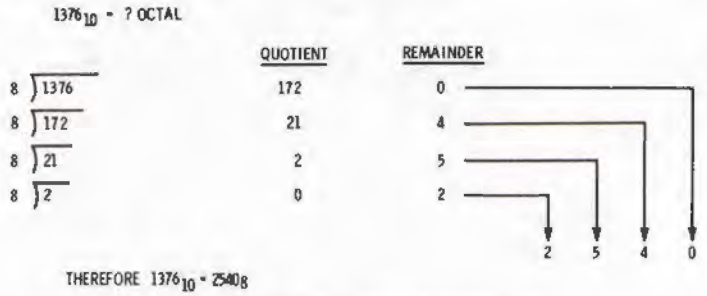


Fig. A-10. Decimal-to-octal conversion.

Addition in Octal

Octal addition is easy if we remember the following rules (which we will find also apply to hexadecimal):

1. If the sum of any column is equal to or greater than the base of the system being used, the base must be subtracted from the sum to obtain the final result of the column.
2. If the sum of any column is equal to or greater than the base, there will be a carry, equal to the number of times the base was subtracted.
3. If the result of any column is less than the base, the base in not subtracted and no carry will be generated.

Examples:

octal	*decimal*
5	= 5
+3	= 3
8	
−8	
10	= 8

octal		*decimal*
35	=	29
+63	=	+51
1 10 8		
−8−8		
1 2 0	=	80

Octal Subtraction

Octal subtraction can be performed directly or in the complemented mode by using addition. In direct subtraction, whenever a borrow is needed, an 8 is borrowed and added to the number. For example:

$2022_8 - 1234_8 = ?$

$$\begin{array}{r} 2022_8 \\ \underline{1234_8} \\ 566_8 \end{array}$$

Octal subtraction may also be performed by finding the eight's complement and adding. The eight's complement is found by adding 1 to the seven's complement. The seven's complement of the number may be found by subtracting each digit from 7. For example:

$377_8 - 261_8 = ?$

a)

$$\begin{array}{rl} 777 & \\ \underline{-261} & \text{(second number)} \\ 516 & \text{7's complement} \\ \underline{+1} & \\ 517 & \text{8's complement} \end{array}$$

b)

$$\begin{array}{rl} 377 & \text{(first number)} \\ \underline{+517} & \text{8's complement of 261} \\ 9\ 9\ 14 & \\ \underline{-8-8-8} & \\ 1\ 1\ 6 = 116_8 & \end{array}$$

Octal Multiplication

Octal multiplication is performed by using an octal multiplication table (see Table A-4) in the same manner as a decimal table would be used. All additions are done by using the rules for octal addition. For example:

$17_8 \times 6_8 = ?$

octal		*decimal*
17	=	15
$\times 6$	=	$\times 6$
1 11 2		90
$-0-8-0$		
$1\ 3\ 2 = 132_8$		

$177_8 \times 27_8 = ?$

octal		*decimal*
177	=	127
$\times 27$	=	$\times 23$
1371		381
376		254
5 11 13 1		2921
$-0-8-8-0$		
$5\ 3\ 5\ 1 = 5351_8$		

Numbers are multiplied by looking up the result in the table. The result of any product larger than 7 (the radix or base) is carried and then octally added to the next product. The results are then summed up by using octal addition.

Table A-4. Octal Multiplication Table

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Octal Division

Octal division uses the same principles as decimal division. All multiplication and subtraction involved, however, must be done in octal. Refer to the octal multiplication table. Some examples:

$$144_8 \div 2_8 = ?$$

$$\frac{144_8}{2_8} = \frac{100_{10}}{2_{10}} = 50_{10} = 62_8$$

$62_8 \div 2_8 = ?$

$$\begin{array}{r} 31 = 31_8 = 25_{10} \\ 2\overline{)62} \\ \underline{6} \\ 02 \\ \underline{2} \\ 0 \end{array}$$

$1714_8 \div 22_8 = ?$

$$\begin{array}{r} 66 = 66_8 = 54_{10} \\ 22\overline{)1714} \\ \underline{154} \\ 154 \\ \underline{154} \end{array}$$

THE HEXADECIMAL SYSTEM

Hexadecimal is another important and often-used computer number system. "Hex" uses the radix 16 and, therefore, has 16 digits. The first 10 digits are represented by the decimal digits 0 through 9, and the remaining six are indicated by the letters A, B, C, D, E, and F. There is nothing special about these letters, and any other letters could have been used. Table A-5 shows the first 16 hexadecimal digits.

Table A-5. First 16 Hexadecimal Digits

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Binary numbers are easily converted to hex by grouping the bits in groups of four, starting on the right, converting the results to decimal, and then converting to hex. For example:

1000	1010	1101	binary
8	10	13	decimal
8	A	D	hex=$8AD_{16}$

As you can probably tell, hex is preferred over octal whenever the binary number to be represented is 16 bits or more. This is because the hex code is more compact than the octal equivalent.

Conversion from hexadecimal to decimal is straightforward but time-consuming. The expansion formula, or a weighting table with an intermediary hex-to-decimal conversion, is used as shown in Fig. A-11.

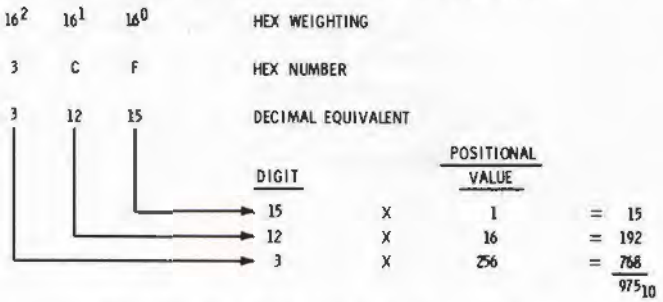


Fig. A-11. Hexadecimal-to-decimal conversion.

Conversion from decimal to hex is performed by repeatedly dividing by 16 and converting the remainder to a hex digit. The quotient becomes the next number to divide. This is shown in Fig. A-12.

$975_{10} = ?_{16}$

	QUOTIENT	REMAINDER IN DECIMAL	REMAINDER IN HEX
975/16	60	15	F
60/16	3	12	C
3/16	0	3	3

3 C F

Fig. A-12. Decimal-to-hexadecimal conversion.

Hexadecimal Addition

Addition in hex is similar to the addition procedure for octal, except the hex digits are first converted to decimal. For example:

$$
\begin{array}{rr}
 & 3CF + 2AD = ? \\
+\ 2AD & = +2\ \ 10\ \ 13 \\
\underline{3CF} & \underline{3\ \ 12\ \ 15} \\
 & 6\ \ 23\ \ 28 \\
 & \underline{-0-16-16} \\
 & 6\ \ \ 7\ \ 12 = 67C
\end{array}
$$

Subtraction in Hexadecimal

Subtraction in hex may be accomplished by either the direct or the complement method. In the direct method, the hex digits are converted to decimal. If a borrow is required, 16 is added to the desired number and the digit borrowed from is decreased by 1. In the complement method, the sixteen's complement of the subtrahend is determined and the two numbers are added. The sixteen's complement is found by adding 1 to the fifteen's complement. The fifteen's complement is found by subtracting each of the hex digits from F. For example:

```
                2BD - 1CE = ?
FFF =          15  15  15
- 1CE        -  1  12  14      second number
               14   3   1      15's complement
                       + 1
               14   3   2      16's complement
2BD=        +  2  11  13      first number
          1    16  14  15
              -16  -0  -0
                               = EF (answer)
```

ignore carry

Hexadecimal Multiplication

Direct hex multiplication is rather tedious and time-consuming. This is because there are 256 entries in a hex multiplication table. The best method is to convert to decimal by using the expansion polynomial and then convert back from decimal to hex after computation.

APPENDIX

B

Numerical Conversion

DECIMAL-HEXADECIMAL-OCTAL-BINARY-ASCII NUMERICAL CONVERSION

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
0	00	00	0 000 0000	NUL	CTRL /1
1	01	01	1 000 0001	SOH	CTRL /A
2	02	02	1 000 0010	STX	CTRL /B
3	03	03	0 000 0011	ETX	CTRL /C
4	04	04	1 000 0100	EOT	CTRL /D
5	05	05	0 000 0101	ENQ	CTRL /E
6	06	06	0 000 0110	ACK	CTRL /F
7	07	07	1 000 0111	BEL	CTRL /G
8	08	10	1 000 1000	BS	CTRL /H, BACKSPACE
9	09	11	0 000 1001	HT	CTRL /I, TAB
10	0A	12	0 000 1010	LF	CTRL /J, LINE FEED
11	0B	13	1 000 1011	VT	CTRL /K
12	0C	14	0 000 1100	FF	CTRL /L
13	0D	15	1 000 1101	CR	CTRL /M, RETURN
14	0E	16	1 000 1110	SO	CTRL /N
15	0F	17	0 000 1111	SI	CTRL /O
16	10	20	1 001 0000	DLE	CTRL /P
17	11	21	0 001 0001	DC1	CTRL /Q
18	12	22	0 001 0010	DC2	CTRL /R
19	13	23	1 001 0011	DC3	CTRL /S
20	14	24	0 001 0100	DC4	CTRL /T
21	15	25	1 001 0101	NAK	CTRL /U
22	16	26	1 001 0110	SYN	CTRL /V
23	17	27	0 001 0111	ETB	CTRL /W
24	18	30	0 001 1000	CAN	CTRL /X
25	19	31	1 001 1001	EM	CTRL /Y
26	1A	32	1 001 1010	SUB	CTRL /Z
27	1B	33	0 001 1011	ESC	ESC, ESCAPE
28	1C	34	1 001 1100	FS	CTRL <
29	1D	35	0 001 1101	GS	CTRL /
30	1E	36	0 001 1110	RS	CTRL /=
31	1F	37	1 001 1111	US	CTRL /-
32	20	40	1 010 0000	SP	SPACEBAR
33	21	41	0 010 0001	!	!
34	22	42	0 010 0010	"	"
35	23	43	1 010 0011	#	#
36	24	44	0 010 0100	$	$

DECIMAL-HEXADECIMAL-OCTAL-BINARY-ASCII NUMERICAL CONVERSION

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
37	25	45	1 010 0101	½	½
38	26	46	1 010 0110	&	&
39	27	47	0 010 0111	'	'
40	28	50	0 010 1000	(	(
41	29	51	1 010 1001	)	)
42	2A	52	1 010 1010	*	*
43	2B	53	0 010 1011	+	+
44	2C	54	1 010 1100	,	,
45	2D	55	0 010 1101	-	-
46	2E	56	0 010 1110	.	.
47	2F	57	1 010 1111	/	/
48	30	60	0 011 0000	0	0
49	31	61	1 011 0001	1	1
50	32	62	1 011 0010	2	2
51	33	63	0 011 0011	3	3
52	34	64	1 011 0100	4	4
53	35	65	0 011 0101	5	5
54	36	66	0 011 0110	6	6
55	37	67	1 011 0111	7	7
56	38	70	1 011 1000	8	8
57	39	71	0 011 1001	9	9
58	3A	72	0 011 1010	:	:
59	3B	73	1 011 1011	;	;
60	3C	74	0 011 1100	<	<
61	3D	75	1 011 1101	=	=
62	3E	76	1 011 1110	>	>
63	3F	77	0 011 1111	?	?
64	40	100	1 100 0000	@	@
65	41	101	0 100 0001	A	A
66	42	102	0 100 0010	B	B
67	43	103	1 100 0011	C	C
68	44	104	0 100 0100	D	D
69	45	105	1 100 0101	E	E
70	46	106	1 100 0110	F	F
71	47	107	0 100 0111	G	G
72	48	110	0 100 1000	H	H
73	49	111	1 100 1001	I	I
74	4A	112	1 100 1010	J	J
75	4B	113	0 100 1011	K	K
76	4C	114	1 100 1100	L	L
77	4D	115	0 100 1101	M	M
78	4E	116	0 100 1110	N	N
79	4F	117	1 100 1111	O	O
80	50	120	0 101 0000	P	P
81	51	121	1 101 0001	Q	Q
82	52	122	1 101 0010	R	R
83	53	123	0 101 0011	S	S
84	54	124	1 101 0100	T	T
85	55	125	0 101 0101	U	U
86	56	126	0 101 0110	V	V
87	57	127	1 101 0111	W	W
88	58	130	1 101 1000	X	X
89	59	131	0 101 1001	Y	Y
90	5A	132	0 101 1010	Z	Z
91	5B	133	1 101 1011	[	[

DECIMAL-HEXADECIMAL-OCTAL-BINARY-ASCII NUMERICAL CONVERSION

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P	X_2		ASCII	Key*
92	5C	134	0	101	1100	/	/
93	5D	135	1	101	1101	]	]
94	5E	136	1	101	1110	^	^
95	5F	137	0	101	1111	_	_
96	60	140	0	110	0000	`	`
97	61	141	1	110	0001	a	a
98	62	142	1	110	0010	b	b
99	63	143	0	110	0011	c	c
100	64	144	1	110	0100	d	d
101	65	145	0	110	0101	e	e
102	66	146	0	110	0110	f	f
103	67	147	1	110	0111	g	g
104	68	150	1	110	1000	h	h
105	69	151	0	110	1001	i	i
106	6A	152	0	110	1010	j	j
107	6B	153	1	110	1011	k	k
108	6C	154	0	110	1100	l	l
109	6D	155	1	110	1101	m	m
110	6E	156	1	110	1110	n	n
111	6F	157	0	110	1111	o	o
112	70	160	1	111	0000	p	p
113	71	161	0	111	0001	q	q
114	72	162	0	111	0010	r	r
115	73	163	1	111	0011	s	s
116	74	164	0	111	0100	t	t
117	75	165	1	111	0101	u	u
118	76	166	1	111	0110	v	v
119	77	167	0	111	0111	w	w
120	78	170	0	111	1000	x	x
121	79	171	1	111	1001	y	y
122	7A	172	1	111	1010	z	z
123	7B	173	0	111	1011	R	R
124	7C	174	1	111	1100	/	/
25	7D	175	0	111	1101	T	T
126	7E	176	0	111	1110	~	~
127	7F	177	1	111	1111	DEL	DEL, RUBOUT

P = Parity bit; "1" for odd number of 1's, "0" for even number of 1's.

*Those key sequences consisting of "CTRL / " are typed in by simultaneously pressing the CTRL key and the key indicated. These sequences are based on those defined for the Diablo 1640 keyboard. These key sequences may be defined differently on other keyboards.

ASCII DEFINITIONS

ACK (Acknowledgment)—Used as a general "yes" answer to various queries, but it also sometimes indicates "I received your last transmission and I'm ready for your next."

ASCII—American Standard Code for Information Interchange.

BEL (Bell)—Activates a bell, beeper, or other audible alarm.

BS (Backspace)—Moves the carriage, print head, or cursor back one space or position.

CAN (Cancel)—Indicates that the material in the previous transmission is to be disregarded. The amount of material is decided by the user.

CR (Carriage Return, or Return)—Moves carriage, print head, or cursor back to beginning of line. On most terminals, the RETURN key causes both a CR and a FF (Form feed).

DC1-DC4 (Device Controls)—These are used to control the user's terminal or similar devices. There are no standard functions assigned, except that DC4 frequently means "stop." The

CCITT suggests a number of possible assignments; in general, they prefer using the first two controls for "on," and the last two for "off," and DC2 and DC4 to refer to the more important device. In some systems, these codes are labeled XON, TAPE, XOFF, and $\overline{\text{TAPE}}$, respectively. X means "transmitter," and TAPE and $\overline{\text{TAPE}}$ means "tape on" and tape off." These labels are found on the keytops of some terminals.

DEL (Delete)—Used to delete a character. It is called RUBOUT on some terminals. It is not strictly a control character since it is not grouped with the other ASCII control characters. The DEL function has binary all-ones bit pattern ($1111\ 1111_2$), and the reason for this is historic: the only way to erase a bit pattern punched into paper tape was to punch out all the holes so that the resulting pattern was equivalent to a null. ASCII still considers DEL equivalent to a null, although many operating systems use it to erase the preceding character.

DLE (Data Link Escape)—This control function uses a special type of escape sequence specifically used for controlling the data line and transmission facilities.

EM (End of Medium)—This is used to indicate the end of paper tape (or other storage medium) or that this is the end of the material on the medium.

ENQ (Enquiry)—This is usually used for requesting identification or status information. In some systems, this code is called WRU—"Who are you?"

EOT (End of Transmission)—Code used to mark end of transmission after one or more messages.

ESC (Escape)—This code marks the beginning of an escape sequence. An escape sequence consists of a series of codes which, as a group, have a special meaning, usually a control function. On some terminals, ESC is called ALT MODE.

ETB (End of Transmission Block)—This code is used when it is desired to break up a long message into blocks. ETB is used to mark block boundaries. The blocks usually have nothing to do with the format of the message being transmitted.

ETX (End of Text)—This code is used to mark the end of a text. See SOH. This code was originally called EOM, "End Of Message," and may be labeled as such on some terminals.

FF (Form Feed)—Advance to the top of next page.

FS, GS, RS, US (File, Group, Record and Unit Separator)—These are a set of codes that are used as "information separators" for delimiting portions of information. There is no standard usage, except that FS is expected to refer to the largest division, and US to the smallest.

HT (Horizontal Tab)—Used to tab carriage, print wheel, or cursor to the next predetermined stop on the same line. It is usually up to the user to decide where the horizontal tab stops are to be positioned.

LF (Line Feed)—This code moves the carriage, print head, or cursor down one line. Most systems combine CR (carriage return) with LF, and new line is then called new line (NL).

NAK (Negative Acknowledgment)—Used to indicate "no" in answer to various queries. It is sometimes defined as "I received your last transmission, but it had errors and I'm awaiting a retransmission."

NUL (Null)—This code is used mainly as a space filler. See also SYN.

SI (Shift In)—Used after a SO code to indicate that codes revert to normal ASCII meaning.

SO (Shift Out)—Indicates that bit patterns to follow will have meanings outside the standard ASCII set and will continue to do so until SI is entered.

SOH (Start Of Heading)—This code is used to mark the beginning of a heading, when headings are used in messages along with text. Headings usually state the name and location of an addressee. This code was originally called SOM, "Start Of Message."

STX (Start of Text)—Used as a marker for beginning of text and end of heading (if used). Was originally called EOA, "End Of Address."

SUB (Substitute)—Indicates a character that is to be used to take the place of a character known to be wrong.

SYN (Synchronous Idle)—Some high-speed data communications systems use synchronized clocks at the transmitter and receiver ends. During idle periods, when there are no bit patterns to enable the receiver's clock to track the transmitter's, the receiver may drift out of sync. Every transmission following an idle period is therefore replaced by three or four SYN characters. The SYN code has a bit pattern that enables the receiver not only to lock onto the transmitter's clock, but also to determine the beginning and end points of each character. SYN characters may also be used to fill short idle periods to maintain synchronization, hence the name.

VT (Vertical Tab)—Used to tab the carriage, print head, or cursor to the next predetermined stop (usually a line).

APPENDIX

C

When Programs Get Too Big— Space-Slashing Hints

Often when writing a big program you may encounter an error statement in BASIC such as "OUT OF MEMORY." This indicates that your program is too large for the amount of memory space provided by the memory chips in your computer. How large is too large? Well, if your computer has 16K of memory (that is 16K bytes) and the BASIC interpreter itself consumes 12K bytes, then your application program cannot take up more than 16K−12K, or 4 K bytes. To determine the number of bytes consumed by your application program you can use the FRE (free) or MEM (memory) function like this:

```
PRINT FRE(0)
1024
OK
```

The FRE function returns the number of bytes "unused" by the current program. Thus, for our 16K machine with a 12K interpreter we have 1024 bytes of our 4096 bytes unused are still available, and our program is then 4096−1024, or 3072 bytes long.

This is all nice, but there is a Murphy's law that states:

FOR EVERY "K" BYTES OF AVAILABLE RAM MEMORY IN A COMPUTER, A PROGRAM WILL EXCEED THIS AMOUNT BY 256 BYTES.

Thus, you will eventually be faced with the prospect of either buying more memory chips for your computer or figuring out how to shorten your program.

Here are 8 "space-slashing" techniques you can apply to help keep the program smaller and save valuable memory space.

1. *Use multiple statements per line.* There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (remember, minimum line number is 0 or 1 and maximum is usually 65529), it takes the same number of bytes. Putting as many statements as possible on a line will cut down the number of bytes used by your program. A single line can usually contain up to 254 characters.
2. *Delete all rem statements.* This is only a good idea if you *don't* want anyone to understand how your program works. Each REM statement uses at least one byte of memory plus the number of bytes in the text of the REM statement. For example, the statement

```
100 REM PROGRAM WRITTEN BY RAMON PEABODY
```

uses up 5+1+33 or 39 bytes.
In the statement

```
200 I=I+1: REM INCREMENT INDEX
```

the REM uses 18 bytes of memory including the colon before REM. The "'" symbol used here would consume one less byte because no colon is required.

3. *Use integers instead of single-precision reals.* Integer variables are stored in 5 bytes; single precisions are stored in 7 bytes. Thus, use integer variables in FOR...NEXT loops and whenever large accurate numbers are not needed. Particularly, use integer arrays instead of real-single precision arrays to save space consumed by arrays.

   You can force a variable to be an integer variable using a DEFINT statement at the beginning of your program, or the % sign after the variable name as in A% or A%(I,J). DEFINT I-N would make all variables beginning with the letters, I, J, K, L, M, N integer variables.
4. *Use variables instead of constants.* Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
10 PI=3.14159
```

in the program, and use PI instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement, as explained in Appendix G.

5. *A program does not have to end with an END statement.* This will save seven bytes.
6. *Reuse the same variables.* Do not do this if you wish absolute clarity in your programs. If you have a variable "S" that is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use "S" again. Or if you have many separate FOR...NEXT loops, the same index may be used over in each loop. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.
7. *Use GOSUBs.* These can be used to execute sections of program statements that perform identical actions.
8. *Remember to use the zero elements of matrices.* For example X(0), BETA (0), etc. If you have a ten-element array required by a program use DIM ARRAY(9) and call the first element ARRAY(0) and the last element ARRAY(9).

STORAGE ALLOCATION IN MICROSOFT BASIC

In order to fully understand how Microsoft BASIC stores variables in memory, here is a breakdown of the bytes consumed by various BASIC number and string variables.

There are two kinds of numeric variables in BASIC—simple variables such as X, Y, ALPHA and Z9, and array variables such as X(5), ALPHA(199) and BETA(X). The actual numbers in *both* simple and array variables may be one of three types—integer, single precision, or double precision.

Table C-1. Simple Variables

SIMPLE VARIABLES	NUMBER OF BYTES REQUIRED
Integer	5 bytes (3 for name, 2 for value)
Single precision	7 bytes (3 for name, 4 for value)
Double precision	11 bytes (3 for name, 8 for value)
String	6 bytes (3 for name, 1 for length, & 2 for pointer)

Table C-2. Array Variables

ARRAY VARIABLES	NUMBER OF BYTES REQUIRED
Integer (# of elements)	*2 + 6 + (# of dimensions) *2 bytes
Single precision "	*4 + 6 + " *2 "
Double precision "	*8 + 6 + " *2 "
String "	*3 + 6 + " *2 "

Thus, the simple integer variable takes up only 5 bytes per variable, while double-precision variables take up 11 bytes each. Use integers wherever possible to save memory space.

In arrays, very many memory bytes can be used up. From Table C-2 we see that in the case of a single-precision array we consume 12 bytes for each element as follows—two bytes for the variable name, two bytes for the size of the matrix, two bytes for the number of dimensions, and two bytes for each dimension along with four bytes for each matrix element. Thus the array A(9) takes up 10*12=120 bytes.

Integer array (XY%(a,b,c. . .)) variables use only 2 bytes for each matrix element, so only 10 bytes are required for each matrix variable. The integer array A%(9) takes up 10*10=100 bytes.

String variables use one byte of string space for each character in the string, in addition to the numbers computed from Tables C-1 or C-2. This is true whether the string variable is a simple string variable like A$, or an element of string matrix, such as COST$(2,3).

When a new function is defined by a DEF statement, 6 bytes are used for the definition. Reserved words such as FOR, GOTO, or NOT, and the names or the intrinsic functions such as COS, INT, and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space is dynamically allocated on the stack (the stack is a special area of memory that is used by the BASIC interpreter to keep track of GOSUB and GOTO branches, as well as nested FOR...NEXTs) as follows:

* Each active FOR...NEXT loop uses 17 bytes.
* Each active GOSUB (one that has not returned yet) uses 5 bytes.
* Each parenthesis encountered in an expression uses 6 bytes per set, and each temporary result calculated in an expression uses 12 bytes.

APPENDIX

D

When Programs Get Too Slow— Speed Hints

Whenever you create programs that contain long complex mathematical equations and functions, or programs with long FOR...NEXT loops (such as those found in sorting arrays), you are likely to run into a speed problem. Although the computer runs at microsecond rates, the BASIC interpreter takes many steps to "understand" the statement it is to execute, and many more steps to set things up so the statement can be executed.

The following list of hints can be used to improve the execution time of your programs. Notice that some of these hints are the same as those used to decrease the space used by your program (see Appendix C). This means that in many cases you can increase the efficiency of both the speed and the size of your programs at the same time.

1. *Use variables instead of constants.* (This is the most important speed hint of all.) It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix integer variable. This is especially important within FOR...NEXT loops of other code that is executed repeatedly.
2. *For variables used often, define them at the beginning of a program.* Variables which are encountered first during the execution of a BASIC program are allocated at the start of the "variable table" (a special table BASIC sets up to tell the line number of a variable's first definition). This means that a statement such as 10 X=0: Y=4; Z=8, will place X first, Y second, and Z third in the variable symbol table (assuming line 10 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable X, it will search only one entry in the symbol table to find X, two entries to find Y, and three entries to find Z.
3. *Use next statements without the index variable when possible.* NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT statement is the same as the variable in the most recent FOR statement.
4. *Frequently referenced lines should be placed at the beginning of the program.* During program execution when BASIC encounters a new line reference such as GOTO 100 it scans the entire user program starting at the lowest line until it finds the referenced line number. Therefore, frequently referenced lines should be placed as early in the program as possible.
5. *In FOR...NEXT loops use integer variables instead of single-precision variables.* It takes much longer to increment a single-precision variable in a loop than an integer variable.

APPENDIX

E

BASIC LANGUAGE REFERENCE

The following list contains all the BASIC keywords, statements, commands, and functions listed—for the most part—in alphabetical order. (For the sake of logical clarity some keywords have been grouped in logical rather than alphabetical order.) Each keyword is explained with typical programming notation showing the exact syntax of the keyword, a description of how the keyword works, and an example of how it appears in a program statement.

In order to differentiate the commands, statements, functions, etc., we have used the labels below to indicate the different types of keywords.

Commands C
Statements ..... S
Numeric Functions ..... NF
String Functions ..... SF
Logical Functions ..... LF
Special Variables ..... SV
Format Control ..... FC

In the descriptions certain words are abbreviated for clarity. Their meanings are as follows:

Abbreviation	Full Word	Meaning
expr	expression	A constant, variable, or group of constants or variables connected optionally by mathematical operators (+,− *, /).
var	variable	A BASIC variable.
num	number	Usually an integer constant,
incr	increment	A number, usually an integer.
str$	string	A BASIC string variable or constant.
list	list	A list of constants and/or variables.
line	line	A BASIC line number.

"A"

ABS (expr)

Returns the absolute value of expr. (NF)

```
300 IF ABS(X)> 5 THEN 400
```

AND (expr1 and expr2)

Returns the logical value TRUE if both expr1 and expr2 are TRUE; otherwise returns FALSE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then AND returns a bit-by-bit AND op-

eration of the bytes representing expr1 and expr2. (LF)

100 IF A>B AND C<D THEN 500

ASC(str$)

Returns the decimal ASCII value of the first character of str$. (NF)

300 IF ASC(A$)>25 THEN 500

ATN(expr)

Returns the arctangent of expr. Expr is assumed to be expressed in radians in the range −PI/2 to PI/2. (NF)

200 PRINT ATN(X)

AUTO num,incr

Sets the automatic line numbering mode to start at line number num and to increment line numbers by incr. To exit from AUTO, type Control/C. (C)

AUTO 100,10

"C"

CALL expr

Causes BASIC program to branch to machine-language subroutine at memory location specified by the value of expr. When subroutine is terminated by a RETURN statement, execution continues at the next line after the CALL. (S)

200 IF A<500 THEN CALL PR+6

CDBL var

Converts variable var to double-precision variable. (S)

20 CDBL A

CHR$(expr)

Returns a single character corresponding to the ASCII code specified by value of expr. (ASCII codes are listed in Appendix B.) (SF)

300 PRINT CHR$(X+3)

CINT var

Converts variable var to integer variable. (S)

20 CINT A

CLEAR expr

Sets all program variables to zero. Optional expr sets data and string space to value of expr. (C)

CLEAR 200

CLOAD str$

Reads a program into memory from cassette tape with name str$. (C)

CLOAD "TEST"

CSAVE str$

Saves the current program in memory on cassette tape with name str$. (C)

CSAVE "TEST"

CONT

CONTinues program execution after a halt caused by a STOP or END statement, or after Control/C has been typed. (C)

CONT

COS(expr)

Returns the cosine of expr, where expr is assumed to be expressed in radians. (NF)

300 PRINT COS(X)

CSNG var

Converts variable var to single-precision variable, (S)

20 CSNG A

"D"

DATA<list>

Supplies values to be assigned to variables by READ statements. List elements are separated by commas. (S)

30 DATA 10,20,30,"FIRST","SECOND"

READ<list of variables>

Assigns values listed in DATA statements to numeric or string variables. Values are assigned in sequence beginning with the first value in the first DATA statement. (S)

40 READ X,Y,Z,A$,B$

RESTORE

Allows data in DATA statements to be reREAD. Next READ statement after a RESTORE begins assigning values from the first DATA statement again. (S)

200 RESTORE

DEF FNV<list of variables>=expr

Defines a user defined function with name FNV, where V is a variable name. Variables specified in the list are dummy variables. (S)

200 DEF FNAVE(A,B,C)=(A+B+C)/3

DEFINT letter1-letter2

All variables beginning with letter1 through letter2 are defined as integer type variables. (S)

50 DEFINT A-N

DEFSNG letter1-letter2

All variables beginning with letter1 through letter2 are defined as single-precision type variables. (S)

60 DEFSNG P-S

DEFDBL letter1-letter2

All variables beginning with letter1 through letter2 are defined to be double-precision type variables. (S)

70 DEFDBL T-V

DEFSTR letter1-letter2

All variables beginning with letter1 through letter2 are defined to be string type variables. (S)

80 DEFSTR W-Z

DELETE num1-num2

DELETEs lines numbered num1 through num2 from the program currently in memory; DELETEs only num1 if num2 not specified. (C)

DELETE 10-50

DIM var1(num1,num2,..),var-2(num1,num2,..),...,str$(num),..

Allocates space for array variables and DIMensions these arrays. The values of the nums set the number of elements in a given dimension (up to a maximum of 255). (S)

20 DIM A(5,10,3),B(3,3),Z$(10)

"E"

EDIT num

Allows EDITing of line number num without affecting other lines in the program. (C)

EDIT 100

END

Stops program execution. Sends carriage return and BASIC's prompt character to screen. (S)

500 END

ERL

A variable which is automatically set to the value of the line number at which a fatal error has been flagged by the BASIC interpreter. (SV)

IF ERL<200 THEN RESUME 500

ERR

A variable which is automatically set to the value of any fatal error flagged by the BASIC interpreter. (SV)

IF ERR = 10 THEN RESUME 500

ERROR num

Causes the BASIC interpreter to act as if error number num has occurred. (C)

ERROR 20

EXP(expr)

Returns the value of e (the base of natural logarithms) raised to the expr-th power. Expr must lie in the range between 0 and 87.3365. (NF)

300 PRINT EXP(COS(X))

"F"

FIX (expr)

Returns the integer portion of expr. (NF)

100 FIX(X/23)

FOR var = expr1 TO expr2 STEP expr3

Begins FOR...NEXT loop. Initializes var to expr1, then increments it by amount expr3 each time all statements through the corresponding NEXT are executed. Continues looping until var is equal to or greater than expr2 at which time execution is continued with the statement following the corresponding NEXT. (S)

FOR M = 20 TO 50 STEP 2

NEXT var1,var2,. . .

Ends FOR...NEXT loop. Var1 is the variable of most recent loop, var2, next most recent, etc. NEXT without a variable terminates most recent loop. (S)

400 NEXT M

FRE(0)

Returns the number of unused bytes in memory. 0 is a dummy argument. (C)

FRE(0)

"G"

GOSUB line

Branches to BASIC subroutine starting at line number specified by line. (S)

250 GOSUB 5000

RETURN

RETURN ends a subroutine and causes execution to resume at the statement after the line which called the subroutine. (S)

350 RETURN

GOTO line

Causes program execution to continue on line specified by the value of line. (S)

200 GOTO 1000

"H"

HEX$(var)

Returns a digit string whose numerical value is equal to the hexadecimal equivalent of var. (SF)

200 A$ = HEX$(X)

"I"

IF expr THEN statement

IF expr is TRUE then execute statement (if statement itself is a number then GOTO line number); otherwise continue execution on to next line. (S)

300 IF A> B THEN PRINT B
400 IF X>10 THEN 250

IF expr THEN statement1 ELSE statement2

Like standard IF . . . THEN except that when expr is FALSE statement2 is executed. (S)

250 IF X>20 THEN 500 ELSE PRINT "ERROR"

INP(expr)

Reads a byte from port number specified by expr (which must lie in the range 0-255). (S)

300 Z = INP(T+3)

INPUT (list of variables)

Enters data into memory from keyboard device and assigns it in order to the variables on the list. Multiple numeric inputs are separated by commas; string inputs must be separated by carriage returns. (S)

300 INPUT A,B,C,D$,E$

INKEY$

A one character string variable which allows entry of a character from the keyboard without the necessity of following with a carriage return. (SV)

400 A$ = INKEY$

INPUT$(num)

A multiple character string variable which allows entry of num characters from the keyboard without the necessity of following with a carriage return. (SV)

400 A$ = INPUT$(5)

INSTR(num,str1$,str2$)

Searches for the string str2$ embedded in str1$ and returns the numerical position of the first match, or 0 if no match is found. Optional offset num sets the position for starting the search in str1$. (NF)

300 IF INSTR(X,A$,B$)>0 THEN PRINT "GOT IT"

INT(expr)

Returns the greatest integer less than or equal to expr. (NF)

200 Z = INT(100)

"L"

LEFT$(str$,expr)

Returns the leftmost expr characters of str$. (SF)

250 C$ = LEFT$(W$,8)+"XYZ"

LEN(str$)

Returns the length of str$. (NF)

300 IF LEN(N$)>20 THEN PRINT "TOO LONG"

LET var=expr

Assigns var the value of expr. (S)

150 LET X = Y+COS(Z)

LINE INPUT "prompt string";str$

PRINTs out the (optional) prompt string and assigns all input from the keyboard to str$ until a carriage return is entered. (S)

100 LINE INPUT "ENTER YOUR NAME,AGE";D$

LIST num1-num2

LISTs the current program in memory from line number num1 through line number num2. LIST alone lists the entire program. (C)

LIST 25-250

LLIST num1-num2

Same as LIST only output directed to lineprinter. (C)

LLIST 25-250

LPRINT

See entry under PRINT.

"M"

MID$(str$,expr1,expr2) (on the right of = 's)

Returns a string expr2 characters long from within str$, beginning with the expr1-th character of str$. (SF)

150 S$ = MID$("ABCDEFGHIJKLMN", X + 3,5)

MID$(str1$,expr1,expr2) = str2$ (on the left of = 's)

Beginning with the expr1-th character of str1$, replace the next expr2 characters by str2$. (SF)

150 MID$(A$,X,Y+2) = B$

MOD (expr1 MOD expr2)

Returns the remainder of expr1 divided by expr2. (NF)

200 IF (X MOD 2) = 0 THEN PRINT "EVEN"

"N"

NEW

Deletes the current program in memory and CLEARs all variables. Used before entering a new program. (C)

NEW

NEXT

See entry under FOR.

NOT expr

Returns opposite truth value of expr. If expr is numerical performs bit-by-bit complement on the byte representing expr. (LF)

"O"

ON expr GOSUB num1,num2,num3,. . .

Branches to BASIC subroutine specified by the expr -th number of the list num1,num2,num3 . . . (S)

300 ON ABS(X) GOSUB 101,102,103,104,105

ON expr GOTO num1,num2,num3,. . .

Same as ON. . . GOSUB except that branch is absolute. (S)

300 ON ABS(X) GOTO 101,102,103,104,105

ON ERROR GOTO line

On the occurrence of any error, transfers command to the statement number specified by line. (S)

250 ON ERROR GOTO 69

OR (expr1 OR expr2)

Returns TRUE if either expr1 or expr2 is TRUE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then OR returns the results of a bit-by-bit OR operation performed on the bytes representing the values of expr1 and expr2. (LF)

200 IF A<5 OR B>10 THEN PRINT "GOOD WORK"

OUT expr1,expr2

Sends byte specified by expr2 to port specified by expr1. (S)

150 OUT P,A+5

"P"

PEEK(expr)

Returns the value of memory location with address specified by expr. Expr must lie in the range 0-65535. (NF)

200 IF PEEK(I)>100 THEN GOSUB 500

POKE expr1,expr2

Stores the byte specified by expr2 —which must lie in the range 0-255 —in memory location specified by the value of expr1 —which must lie in the range 0-65535. (S)

150 POKE 4*M+1,240

POP

Increments nested GOSUB return stack address by 1. (S)

400 POP

POS(N)

Returns the current column position of the screen cursor; leftmost position equals 0. N is a dummy argument. (NF)

100 CU = POS(Z)

PRINT var,..,expr,..,str$,..

Outputs data specified by list starting at the present cursor position. (S)

500 PRINT X,Y,Z,A+3,B$

LPRINT . . .

Same as PRINT except that output is directed to the lineprinter. (S)

500 LPRINT X,Y,Z,A+3,B$

PRINT@expr

Begin PRINTing at column number specified by expr. (S)

200 PRINT@ X + 3

PRINT USING str$,expr

PRINTs the value of expr in format specified by str$.(S)

250 PRINT USING "$$###.##", SA

"R"

RANDOMIZE

See entry under RND.

READ. . .

See entry under DATA.

RESTORE

See entry under DATA.

RENUM num1,num2,incr

Renumbers all line numbers in current program beginning with line number num2. The new se-

quence begins with num1 and lines are incremented by incr. If num2 is omitted entire program is renumbered; if incr is omitted, then increment is 10. (C)

RENUM 400,300,50

RESUME num

RESUMEs program execution at line number num after a halt generated by error trapping routine. If num is omitted, RESUMEs at next statement after error. (C)

RESUME 500

RETURN

See entry under GOSUB.

RIGHT$(str$,expr)

Returns the rightmost expr characters of str$. (SF)

200 PRINT RIGHT$(A$,5)

RND(expr)

Returns a (pseudo) random number between 0 and 1. If expr < 0, RND returns a new sequence of numbers; if expr = 0, RND returns the last number over again; if expr > 0, RND returns the next number of the current sequence. (NF)

500 IF RND(1)>.5 THEN 700

RANDOMIZE

Resets the random number generator to produce a new sequence of (pseudo) random numbers. (S)

400 RANDOMIZE

RUN num

Begins execution of program currently in memory at line number num. If num is omitted begins at lowest line number. (C)

RUN 250

"S"

SGN(expr)

Returns the sign of the value of expr: −1 if expr < 0 ; 0 if expr = 0; and 1 if expr > 0. (NF)

200 X = INT(X + .5*SGN(X))

SIN(expr)

Returns the sine of expr. Expr is assumed to be expressed in radians. (NF)

200 IF SIN(X) > .3 THEN 250

SPACE$(expr)

Returns a string of blanks expr spaces long. (SF)

150 X$ = A$ + SPACE$(I + 5)

SPC(num)

Used only to format output in PRINT statement. PRINTs a string of blanks num spaces long. (FC)

200 PRINT SPC(10);A$

SQR(expr)

Returns the square root of expr. Expr must be > 0 or =0. (NF)

250 Z = SQR(X*Y)

STOP

STOPs program execution; prints BREAK IN LINE nnn (nnn is line number at which program STOPped) and returns to command level. (S)

500 STOP

STR$(expr)

Returns the string representation of the value of expr. (SF)

300 PR$ = "$" + STR$(PR)

STRING$(expr,str$)

Returns a string expr characters long made up from the first character of str$. (SF)

200 PRINT STRING$(A,B$)

SWAP var1,var2

Exchanges the values of var1 and var2. Both must be of the same variable "type" or an error will result. (S)

300 SWAP A$,B$

SYSTEM

Causes an exit from BASIC system and transfers to operating system. (C)

```
SYSTEM
```

"T"

TAB(expr)

Used to format output in PRINT statement. Causes PRINTing to occur at column specified by value of expr. Leftmost position is 0. (FC)

```
250 PRINT TAB(I);A$;TAB(I+10);B$
```

TAN(expr)

Returns the tangent of expr, where expr is assumed to be expressed in radians. (NF)

```
300 IF TAN(X) > 1 THEN 500
```

TRON

Turns on the trace flag which causes BASIC to print each line number as that line is executed. (C)

```
TRON
```

TROFF

Turns off the trace flag. (C)

```
TROFF
```

"U"

USR(expr)

Calls a machine language subroutine at address specified by USRLOC and passes it the value of expr. (S)

```
200 USR(X*100)
```

USRLOC

A special location in memory which contains the starting address of a user's machine-language subroutine which will be called by USR. (S)

"V"

VAL(str$)

Returns the numerical value of the digit string str$. If str$ is not a digit, string returns 0. (NF)

```
250 IF VAL(X$) < 25 THEN 500
```

VARPTR(var)

Returns the memory address of the variable var. (NF)

```
AD=VARPTR(Y)
```

VTAB(expr)

Causes vertical TAB expr lines downward. (S)

```
200 VTAB(I+5)
```

"W"

WAIT port,num

Bit pattern in port specified by value of port is ANDed with value specified by num and execution is halted until logical TRUE results. (S)

```
300 WAIT 255,4
```

WHILE expr

Begins WHILE...WEND loop. If expr is TRUE execute all statements until the following WEND statement and continue looping until expr becomes FALSE; if expr is FALSE jump to statement following WEND. (S)

```
200 WHILE A < 200
```

WEND

Terminates a WHILE...WEND loop. (S)

```
500 WEND
```

WIDTH num

Sets the width of the PRINT line to num columns. (C)

```
WIDTH 45
```

"X"

XOR (expr1 XOR expr2)

Exclusive OR. Returns the value TRUE if expr1 and expr2 have different logical values, otherwise returns FALSE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then XOR returns the results of a bit-by-bit Exclusive OR operation on the bytes representing the values of expr1 and expr2. (LF)

```
IF A<5 XOR B<5 THEN 450
```

"MISCELLANEOUS"

&H str$

Returns the decimal equivalent of the numerical value of str$, where str$ is assumed to be expressed in hexadecimal. (NF)

```
200 PRINT PEEK (&H7AB0)
```

&O str$

Returns the decimal equivalent of the numerical value of str$, where str$ is assumed to be expressed in octal. (NF)

```
300 PRINT &0177
```

VARIABLE TYPE DECLARATION CHARACTERS

Character	Example	Typical Values	Definition
$	A$,Z1$	"JOHN" "1+1+?"	String variable type may contain up to 255 characters.
%	%A1,B%	−14,345,6009	Integer variable type—whole numbers with absolute value less than 32767.
!	A!, A1!	1,−.123,25.3067	Single-precision variable type containing six significant digits.
#	A#,Z1#	−300.12345678 3.141592653589 1.0000010001234	Double-precision variable type containing 14 to 16 significant digits.
D	A# 1.2345678901 D 12	1.2345678901 X 10^{12}	Double-precision variable type with scientific notation.

APPENDIX

F

Exponential (Scientific) Notation

Exponential notation, which is also sometimes called scientific notation, is a way of representing numbers that are too large or too small to write conveniently in the ordinary way. When your computer prints out numbers with an E in the middle of them, like 4.78245E+10, it is using exponential notation. In this appendix we will describe what exponential notation is, why it is useful, and how to convert between exponential and "normal" notation.

Computers are capable of storing and performing arithmetic on very large (and very small) numbers. Just how large a number can a computer store? Would you believe one billion billion billion billion? That certainly sounds large, but it's no problem for your computer. What would a number this big actually look like written down? Let's figure it out. Think for a minute about the number "one thousand." It is written as a 1 followed by 3 zeros: 1000. A million, which is a thousand times a thousand, or one thousand thousand, is a one with 6 zeros: 1,000,000. And a billion, which is a thousand times a million, or one thousand million, is a one with nine zeros after it: 1,000,000,000.

Perhaps you've noticed something here: When we multiply large numbers like one thousand and one million together, we can easily find the answer just by adding up how many zeros there are in the numbers. So, one thousand, with 3 zeros, times one thousand, with 3 more zeros, equals one million, with 6 zeros (since 3 plus 3 is 6); and one thousand, with 3 zeros, times one million, with 6 zeros, is one billion, which has 9 zeros (since 3 plus 6 is 9).

Using this idea, let's figure out how many zeros there are in one billion billion billion billion. Since we've already decided that there are 9 zeros in one billion, we can add 9 zeros to 9 more zeros and conclude that there are 18 zeros in one billion times one billion, which is a billion billion. There must be 18 plus 18 or 36 zeros in a billion billion times a billion billion, which is a billion billion billion billion. So there's our answer: a computer can hold the number one billion billion billion billion, which is a one followed by 36 zeros. Printed out, it looks like this:

1,000,000,000,000,000,000,000,000,000,000,000,000

To give yourself an idea just how big numbers like this really are, consider this: the universe is thought to be about 14 billion years old. That's 4.2 hundred million billion seconds. The speed of light is about 300,000 kilometers per second, which is 300 billion millimeters per second. (A millimeter is about the thickness of the lead in a pencil.) Multiplying the age of the universe in seconds by the speed of light in millimeters per second, we arrive at the conclusion that if a ray of light had started out at the beginning of the universe, it would have covered a distance, up to the present, of 126 billion billion billion millimeters. (Which, written out, is:

126,000,000,000,000,000,000,000,000,000.0 mm.).

That's still a good bit smaller than the size of the numbers your computer can store.

"That's all very well," you may be saying, "but how could I type in a number with all those zeros, or read it out again? I would probably make a mistake and type in the wrong number of zeros, or not count correctly how many there are on the screen."

That's a good point. To simplify the reading and writing of large numbers, mathematicians invented a whole new way of writing numbers, called exponential notation. The idea behind exponential notation is this: instead of writing down all the zeros in a big number, we just write down how many zeros there are. How many zeros in a thousand? Three. In exponential notation you write the number of zeros after the letter "E," (which, of course, stands for "Exponential"). So, for one thousand we write 1E+3. (The 1 before the E means one (1) thousand, and we'll explain the plus sign later.) How do we write one million? It has 6 zeros, so we write 1E+6. How do we write one billion billion billion billion? Easy: we know it has 36 zeros, so we write 1E+36. That's a lot more practical than writing out all those zeros, isn't it? This way of writing numbers—exponential notation—is how your computer stores single- and double-precision numbers, and how it prints out numbers that are too large to print normally.

The number which follows the E in exponential notation, and represents the number of zeros in the number, is called the "exponent" of the number.

If you want to see your computer print out a number in exponential notation, type the following (in command mode):

```
PRINT 10000 * 10000
```

This is ten thousand (which has 4 zeros) times ten thousand (another 4 zeros) so your computer should respond with:

```
1E+8
```

which represents one hundred million, a number with 8 zeros. You can experiment with this, typing in different values and seeing how BASIC responds.

But, you say, suppose I want to talk about a more complicated number? Do I always have to talk about one (1) of something, like one thousand or one million? Could we represent 2.75 million in exponential notation, for instance? No problem. To represent 2.75 million we just write 2.75E+6. If we wanted to represent one hundred and twenty-five thousand (125,000), we would write 1.25E+5. Six hundred (600) would be 6E+2, and so on. This number, which goes to the left of the E in exponential notation, is called the "mantissa." It is written with its first digit to the left of the decimal place, and any other digits to the right.

As you can see, a number written in exponential notation has three parts: first, the mantissa, which is the original number, but written with the decimal place to the right of the leftmost digit; second, the exponent, which is the number of zeros in the number; and third, a letter "E" to separate the mantissa and the exponent.

Seems pretty easy, but actually there's something we have to be careful of. Until now we have referred to the exponent (the number on the right of the E) as representing how many zeros there are in the number. Actually, this was a slight simplification. What the exponent really represents is how many digits (which may or may not be zeros) there are to the right of the first (leftmost) digit in the number. For instance, as we said above, 2.75 million (or 2,750,000), is 2.75E+6 in exponential notation. We write 6 after the E, not because there are 6 zeros in the number (there are 4) but because there are 6 digits following the 2, which is the left-most or "most significant" digit.

Another way to think of the exponent is this: it tells you how many times you must multiply the mantissa (the number on the left of the E) by ten. Why is this true? Because multiplying a number by ten is the same as adding a zero on the right of the number. Or, if the number is a decimal fraction—with zeros to the right of its decimal point—we can say that *multiplying by ten is the same as moving the decimal point to the right one place.*

Using this idea, let's figure out what 4.234E+4 is in normal numbers. 4.234 times 10 is 42.34. Multiply by ten again and you get 423.4, and a third time is 4234. Multiply by ten a fourth time and you get 42340, which is the number as we would normally write it, in nonexponential notation. We've multiplied 4.234 by ten 4 times, which moved the decimal point 4 places to the right. Ten times ten times ten times ten is ten thousand (10 * 10 * 10 * 10 = 10000) or 1E+4. So 42340 is 4.234 times ten thousand, or 4.234E+4.

Let's convert 9,321,000 to exponential notation. That's 9 million and a bit left over. How do we write one million in exponential notation? One million is a thousand times one thousand, and since one thousand is 10 multiplied by

itself 3 times, a million is 10 multiplied by itself 6 times. For any number of millions, whether 1 million, 2 million, or 9 million, we take the number of millions and multiply it by ten, 6 times. So 9,321,000 = 9.321 * 10 * 10 * 10 * 10 * 10 * 10. And, of course, the easy way to write this is 9.321E+6.

Let's compress these examples into some simple rules for converting to and from exponential notation.

RULE 1: *To convert the "normal" representation of a number larger than 1 to exponential notation, (a) move the decimal point to the left until there is only one digit to the left of the decimal point; (b) chop off any trailing zeros on the right of the number; (c) write an E after the number and follow it with a plus sign; and (d) after the E and the plus sign write down how many times you had to move the decimal point.*

For example.

3 6 0 0 0 start with this

3.6 0 0 0, move the decimal point until only 1 digit is still on the left
4 3 2 1

3.6E+4 erase any trailing zeros and write E and the exponent

RULE 2: *To convert from exponential notation to normal representation, (a) move the decimal point to the right the number of times indicated by the exponent, filling in any zeros that are necessary, (b) erase the E and the plus sign and the exponent.*

For example,

8.7654E+3 start with this

8.7 6 5 4 E+3 move the decimal point the number of times indicated by the exponent
1 2 3

8765.4 erase the E and the exponent

Small numbers, as well as large ones, can be represented by using exponential notation. How small? Would you believe one billion billion billion billionth? That's a decimal point followed by 35 zeros and a 1, like this

0.00000000000000000000000000000000001

In exponential notation we would write: 1E−36. (This is where you find out about the plus sign that we promised earlier we would explain.) As you can see, with large numbers the sign of the exponent is positive, and with small numbers it is negative. But 1E−36 is a very very small number. For convenience, let's think about something larger, like one one-hundredth, or 0.01. In exponential notation this is 1E−2. How do we know that? Well, for one thing, we'd have to move the decimal place 2 places to the right in order for the 1 to be to the left of the decimal point. Also, where a positive exponent means to multiply the number by ten the number of times indicated by the exponent, a negative exponent means to divide it by ten that many times. If we divide 1 by 10 once we get 0.1, and if we divide it again we get 0.01. That's two divisions by ten, so the exponent must be a negative 2. Other examples of negative exponents are:

0.00275 is 2.75E−3
0.99999 is 9.9999E−1
0.000512 is 5.12E−4
0.00000002 is 2E−8

We can now write down two more rules, this time for handling exponential notation with small numbers:

RULE 3: *To convert a number which is smaller than one to exponential notation, (a) move the decimal point to the right until there is one digit to the left of the decimal point, (b) write an E and a minus sign after the number, and (c) after the minus sign, write how many times you had to move the decimal point.*

For example:

.000275 start with this

0 0 0 2 7 5 move the decimal point until there is only one digit to its left
1 2 3 4

2.75E−4 add the E and the negative exponent.

RULE 4: *To convert a number which is smaller than one, from exponential notation to normal representation, (a) move the decimal point to the right the number of places indicated by the exponent, adding zeros if necessary, (b) erase the exponent.*

9.99E−3 start with this

0 0 9. 9 9 E − 3 move the decimal point
3 2 1

.00999 erase the E and the exponent.

Now we need to talk about something that can be quite confusing: negative numbers in exponential notation. The thing to keep in mind here is: *Don't confuse negative exponents with negative numbers!* The sign of the exponent and the sign of the number (which is the same thing as the sign of the mantissa of the number) are two entirely different things. So far we have talked only about positive numbers—that is, numbers greater than 0. These positive numbers can be either large numbers with an exponent greater than 0, or small numbers, with an exponent smaller than zero. (Actually, to have a positive exponent a number doesn't have to be all that large: if it is greater than one it has a positive exponent, it it is smaller than one it has a negative exponent.)

Similarly, a negative number—that is, one less than zero—can either be "large" and have an exponent greater than zero; or it can be small, and have an exponent less than zero.

Table F-1 summarizes these combinations.

Table F-1. Ranges of Numbers in Exponential Notation

Example of Number	Sign of Number	Sign of Exponent	Number in Exponential Notation	Range of Numbers
478	+	+	4.78E+2	very large positive to +1
.0478	+	−	4.78E−2	+1 to 0
−.0478	−	−	−4.78E−2	0 to −1
−478	−	+	−4.78E+	−1 to very large negative

All we're really saying here is that a very large number (which, because it is large, has a positive exponent) can be either positive or negative; and that a very small number (which, because it's small, has a negative exponent) can also be either positive or negative.

Your computer is one of the best teachers of exponential notation. Type in some large numbers, do some arithmetic with them in the command mode, experiment a little, and pretty soon you'll find exponential notation almost as easy as the regular "old fashioned" way of writing numbers.

APPENDIX

G

ANSWERS TO REVIEW QUESTIONS

CHAPTER 1

1. Only (a). Negative and fractional line numbers are illegal, as are numbers greater than 65535.

2. They're all correct, at least on most computers. Refer to the LLIST instruction in the section on lineprinters for more information.

3. Neither of them; (a) will put too many spaces between the words, and (b) won't put any.

4. (a) evaluates to 3, while (b) (c) and (d) evaluate to 2. Since BASIC does division and multiplication before it does addition and subtraction, you need to put parentheses around the "2+2" in (a) if you want the addition to take place before the sum is divided by 2.

5. (a), (b), (d). FL, F2 and FLAG are legal in most versions of BASIC (although only the first two characters of FLAG will be recognized by the program). Variable names must start with a letter, so 2F isn't legal, and they cannot contain anything but letters and numbers, so F* is illegal.

6. It's not valid. You can't use quotes inside a string, since they are already used to delimit the string.

7. Before the program is run, no value is assigned to Q (unless your version of BASIC sets it to zero automatically). After the program is run, Q will be assigned a value of 3, meaning that the Q storage box holds a value of 3; but it is never "equal" to 3 in a mathematical sense. (Note that it is perfectly acceptable to remove the LET from this sort of statement. The LET is optional.)

8. No. Since the "=" in a "LET" statement means to assign a value (to

the right of the "=") to a variable (on the left of the "="). It doesn't make sense to try to assign a variable name to a constant.

9. (a). Line 10 is an error because "IF" is a reserved word and therefore cannot be used as a variable name. Line 20 is all right. Line 30 is all right too, because IF$ is not the same name as IF. Line 40 is wrong because it tries to assign a numerical value to a string variable.

10. 10 INPUT A,B,C,D,E
    20 PRINT E;" ";D;" ";C;" ";B;" ";A

(The spaces between the letters are necessary to separate the words.)

CHAPTER 2

1. It will cause an error message, since you can't count downward in a FOR...NEXT loop without specifying "STEP −1" or some other negative number.

2. It will go through the loop 20 times, printing 0, 0.5, 1.0, 1.5, and so on. Fractional STEPs are perfectly all right.

3. If you type 0 it won't GOTO any line number, but will simply fall through to line 30 and print "NEVER SAY DIE." If you type 1 it will go into an endless loop since 20 is the line the ON GOTO statement is on. If you type 2 it will go to line 40 (the second line in the list) and print "SAY DIE."

4. Yes. The equal sign in a conditional (IF) statement is the same as a mathematical equal sign. It is not the same as the equal sign in an assignment (LET) statement. Therefore, it is correct to put the variable name on the right and the constant on the left.

5. It will print the word "CAT." Each subroutine will call the next one, until the one at 300 prints "C," then returns to the one at 200, which prints "A" and returns to the main program at 110, which prints "T."

6. No. The first expression evaluates to 2, because INT takes the next smallest integer for negative numbers, while the second expression evaluates to 1.

7. Try it!

8. If you run a program with this statement in it, it will probably spend a long time executing the statement. This is because the statement will

continue to loop back on itself until the random number it generates is exactly .5000000, and there are millions of other possibilities.

9. An interesting question. The program will run for a long time without appearing to do anything, and then you will get an "out of memory" error message. Why? Because every time you execute a GOSUB, BASIC has to store the return address someplace. Then when you execute a RETURN, it gets rid of the return address. In this program, because of the GOTO 10 in line 120, the RETURN is never executed, and BASIC has to keep storing more and more return addresses, one for each time it executes the GOSUB. Since it never executes a RETURN, memory will eventually fill up with return addresses, clogging the memory.

10. 255 times. Since string variables can't be any longer than that, you will then get a "string too long" error message when you try to add the 256th character to S$.

CHAPTER 3

1. BASIC needs to set aside memory for an array before it starts to assign values to the elements of the array. This makes it easier for BASIC to find a particular array element. DIM is used for this purpose.

2. No. It is perfectly possible, and often useful, to figure out how big your array will be after the program is running.

3. Counting the zero elements, there are five elements in each dimension, for a total of 5 times 5 times 5, or 125.

4. You will probably get an "out of memory" error, since 21 times 21 times 21 times 21 is 194,481 which is larger than most personal computer memories.

5. You will get an error message, since negative subscripts are not legal in BASIC.

6. The answer depends on which version of BASIC you are using. In Microsoft BASIC you will have the same number of strings as there are elements in the array (six in this case), and each string can be as long as you like (up to 255 characters). In some other BASICs you will get an error message, because in string arrays only one dimension is allowed, which is the length of the string.

7.
```
10 N=1
20 FOR I=1 TO 4
30 FOR J=1 TO 3
40 FOR K=1 TO 2
50 X(I,J,K) = N
60 N=N+1
70 NEXT K
80 NEXT J
90 NEXT I
```

8.
```
110 FOR I=1 TO 4
120 FOR J=1 TO 3
130 FOR K=1 TO 2
140 PRINT X(I,J,K)
150 NEXT K
160 NEXT J
170 NEXT I
```

CHAPTER 4

1. This program will work fine, printing out "1 2 3" just as you want it to. The lesson here is that you can put a DATA statement anywhere in your program, either before or after the READ statement. In fact, you don't even need to execute the DATA statement for it to work.

2. It's valid, but not useful. What it really says is: "If either 0 or 1 is equal to 1 (that is, if either of them is "true"), then print "TRUE." Since 1 is always 1, "TRUE" will always be printed.

3. BASIC has a certain order in which it performs operations. In arithmetic, for example, it always does division and multiplication before it does addition and subtraction. In logic it does AND before it does OR. In this program B and C are both 0 (false), so "B AND C" is also false. However, A is 1 (true), so "A OR B AND C" is true, and "TRUE" will be printed out. This will be clearer if you think of the expression as "A OR (B AND C)." If BASIC did the OR first, then the expression would be "(A OR B) AND C," and the statement would not print "TRUE."

4. Since EXP and LOG are inverse functions to each other, "100" will be printed out.

5. On executing the first statement, the computer will print "70." On executing the second statement, the computer will print "F." This is because CHR$ and ASC are inverse functions to each other.

6. CHR$(N) returns a single character which has an ASCII value of N, while STR$(N) returns a string of characters which are the digits of the number N.

7. ASC(A$) returns the ASCII value of the character A$, which is a number between 0 and 255, while VAL(A$) returns a number of any size which is the numeric value of the character digits in the string A$.

8. 20 PRINT LEFT$(A$,4) + "NEVER" + RIGHT$(A$,9)

9. Draw a picture of the ladder leaning against the house. The ladder is the hypotenuse of a triangle. The angle A is the angle between the ladder and the ground. The "side opposite" is the house, 12 feet high, and the "side adjacent" is the ground, 6 feet long. Here's the program:

```
10 A=ATN(12/6)
20 L=12/SIN(A)
30 PRINT"LADDER IS "; L; " FEET LONG."
```

10.
```
10 L=SQR(12^2 + 6^2)
20 PRINT"LADDER IS "; L; " FEET LONG."
```

CHAPTER 5

1.
```
10 FOR K=0 TO 511
20 IF PEEK(K)=0 THEN PRINT"-"; : GOTO 40
30 PRINT"X";
40 NEXT K
```

   A slightly fancier version of this program can be a useful one for investigating which sections of memory are being used by BASIC or your program. The user's manual for your computer should have a "memory map" which will be helpful for this.

2. Eight, since there are 8 bits in each port, and each bit can be either 0 or 1, representing "on" or "off."

3. "CALL 1000" is used in assembly-language programs and is a subroutine jump to address 1000 in memory. GOSUB 1000 is used in BASIC and is a subroutine jump to line number 1000, which is not at all the same as address 1000.

4. "CLOAD" and "CSAVE" only work with BASIC programs. On some computers data may be transferred to and from tape using the instruc-

tions PRINT# and INPUT#. "LOAD" always means to load something into memory. "SAVE" means to save it on tape or disk.

5. On most computers you will get a "CAN'T CONTINUE" error message. You can't alter a program (in this case you are changing line 10) and then continue running it with a CONT command, because, for one thing, BASIC resets all the variables to zero when you alter any of the program lines.

CHAPTER 6

1. You can do loops with an IF...THEN statement and a GOTO; or you can use a FOR...NEXT statement; or, if your version of BASIC has it, you can use WHILE...WEND.

2. Yes it will. The way to look at the IF...THEN...ELSE statements is that the entire inner IF...THEN...ELSE statement is nested inside the outer one. Rewriting it this way makes it clearer:

```
20 IF A THEN IF B THEN PRINT "BOTH TRUE"
                       ELSE PRINT"A TRUE, B FALSE"
           ELSE PRINT"A FALSE"
```

3. First, you can use any variables you want to as arguments; and second, the function has a name instead of a line number so it's easier to recognize. The disadvantage of a function is that it is only one line long, whereas a subroutine can be any number of lines.

4. LINE INPUT lets you input commas, quotes, and line feeds as part of the input string, and it doesn't automatically output the "?" as a prompt.

5. You can't use "return" (or "enter"). Also, you can't use the "break" or control-C key.

6. No. INKEY$ will only read one keystroke at a time. You have to release the key and press it again to input another character.

7. P will be 6. INSTR is looking for the word "CAT" in the sentence "CATCATCAT" but it doesn't start looking until the fourth character, so it misses the first "CAT" and finds the second one.

8. You'll get an error message, since a string can only be 255 characters long, and this statement tries to create one 500 characters long.

9. No, it will print "CAT'S DEVOURCE." This is because you are replacing 6 letters, which is how long DEVOUR is. Unfortunately, you can't use MID$ to change a string if by so doing you would change its length.

10. RESUME takes you back to the main program following ON ERROR GOTO, and RESTORE puts the internal pointer back to the beginning of a DATA statement.

11. RESUME takes you back to the main program after an error handling routine, and resets the ON ERROR GOTO statement so it will work the same way the next time an error is encountered. ON ERROR GOTO 0 turns off all other ON ERROR GOTO statements so that control will not go to your error handling routine when an error occurs.

12. Integers take up less space in memory, and they are faster in arithmetic operations. The disadvantage is that they cannot represent numbers greater than 32767.

13. Two, four, eight, and one per character; respectively.

14. FF 77 255 63

15.
```
10 F=1
20 INPUT"NUMBER"; N
30 GOSUB 100
40 PRINT F
50 STOP
100 IF N=1 THEN RETURN
110 F=F*N
120 N=N-1
130 GOSUB 100
140 RETURN
```

Index

A

ABS, absolute value function, 81
Advanced
  input/output, 207-219
  string functions, 219-229
AND, OR, and NOT, 143-146
Arithmetic operators, 29
ARRAYNAME, 106
Arrays and subscripted variables, 105-108
ASC and CHR$, 152-154
ASCII definitions 333-334
AUTO, 181-182

B

BASIC
  applications, 15-21
    business, 15-16
    educational, 18-19
    entertainment, 16-17
    scientific, 19-21
  command aids, 180-188
    AUTO, 181-182
    CLEAR, 182
    CONT, 183
    DELETE, 183-184
    EDIT, 184
    FRE(0), 182-183
    NEW or SCRATCH, 181
    RENUM or REN, 184-185
  dialects, 13-15
  no-no's, 50-51
Binary system, 310-322
  addition and substraction, 313
  binary-coded number representation, 320-322
Binary system—cont
  counting, 311-312
  multiplication and division, 314
  negative numbers, 317-318
  pulse representation of numbers, 316
  representing binary numbers, 315
  serial and parallel transmission, 316-317
  two's complement, 319-32
  use of complements, 318-319
Business applications, 15-16

C

CALL(or USR), 171-177
  what it can do, 173-176
CINT, CSNG, and CDBL, 254-258
CLEAR, 182
Combining functions, 85-87
Commas, printing with, 30-33
Concatenation, 43-44
CONT, 183
Controlling external world with PEEK and POKE, 164-165
Conversion, numerical, 331-333
CSAVE and CLOAD, 177-180
Customer billing, 16

D

DATA and READ statements, 135-139
DATA statement, format, 136
Dealing with errors, 229-238
Debug commands, 185-188
Decision making statements, 63-72
DEF FN, 204-207

DEFINT, DEFSNG, DEFDBL, DEFTR, 246-248
DELETE, 183-184
Dialects, BASIC, 13-15
Dimension with DIM, 100-110
Direct Mode programming, 35
Documentary programs, 56-58
Double-precision variables, 244-246
DSP, 187-188

E

EDIT, 184
Educational applications, 18-19
ELSE, 201-202
Enhanced Metric Conversion program, 139-142
Enhancing the Loan program, 72-76
Entertainment applications, 16-17
ERR and ERL, 233-234
ERROR, 235-236
  used as an extended form of GOSUB, 236-237
Error
  -checking subroutine, 97-101
  -trapping routine, when to use, 237-238
Expanding a program, 55-56
EXP and LOG, 149-150
Exponential notation, 351-356
Exponentation, 37

F

Figuring in BASIC, 36
Finding prime numbers, 202-204
FIX function, 83
Format for DATA statement, 136
FOR. . . NEXT, programmed loop, 63-65
FRE(0), 182-183
Function(s)
  ABS, 81
  ASC and CHR$, 152-154
  combining, 85-87
  INT, 81-83
  LEFT$, MID$, and RIGHT$, 154-157
  MOD, 151-152
  numeric, 142-152
  SGN, 151
  SQR, 150-151
  statements, simple, 81-87
  string, 152-158
  STR$ and VAL, 157-158

G

Game BASIC: TIC-TAC-TOE, 110-130
  check for a tie, 118-119
  check for a winner, 119-120
  computer move algorithm, 122-124
  detailed flowchart, 116-117
  displaying the board, 114-115
  finishing the program, 124-125
  input players move, 121-122
  overall flow of the game, 113-114
  remember the rules, 111-112
  representing the board, 112-113
  representing the markers, 113
General ledger, 15-16
Getting started, 21-22
GOTO, unconditional branch, 68-70

H

Hexadecimal number system, 326-329
  addition, 328
  multiplication, 329
  subtraction, 328
HEX$, OCT$, &H and &O, 265-266

I

IF. . . THEN, conditional branches, 66-68
INKEY$, 212-216
INP, OUT and WAIT, 168-170
Input, 44-48
  line, 207-209
  /output, advanced, 207-219
INPUT$, 216-218
INSTR, 219-221
INT, integerize function, 81-83
Inventory control, 16

L

LEFT$, MID$ and RIGHT$, 154-157
Line
  input, 207-209
  numbers, 23-25
  printer, using, 192-193
LIST, 25-26
Little variable boxes, 37-41
LLIST, 192
Loan programs, 52-55
  enhancing, 72-76
Logical functions, 341-349
  AND(expr1 and expr2), 341
  NOT expr, 345

Logical functions—cont
OR(expr 1 OR expr 2), 345
XOR(expr1 XOR expr2), 349
LPRINT, 192-193

M

Mailing label preparation, 16
Manipulating screen data with PEEK and POKE, 165-167
Metric conversion program, 76-80
enchanced, 139-142
Micro space invaders, 223-225
MID$ (on the left), 227-229
Miscellaneous BASIC functions, 258-268
MOD function, 151-152

N

Names, variable, 41-42
NEW or SCRATCH, 181
Numbering systems
binary, 310-322
hexadecimal, 326-329
octal, 322-326
Numbers, 29
Numeric
functions, 142-152, 341-349
ABS(expr), 341
ASC(str$), 342
ATN(expr), 342
COS(expr), 342
EXP(expr), 344
FIX(expr), 344
INSTR(num, str1$, str2$), 345
INT(expr), 345
LEN(str$), 345
MOD(expr1 MOD expr2), 345
PEEK(expr), 346
POS(N), 346
RUD(expr), 347
SGN(expr), 347
SIN(expr), 347
SQR(expr), 347
TAN(expr), 348
VAL(str$), 348
VARPTR(var), 348
&H str$, 349
&O str$, 349
"round-off" subroutine, 90-95
Numerical conversion, 331-333

O

Octal number system, 322-326
addition in octal, 324
division in octal, 326
multiplication in octal, 325
subtraction in octal, 324-325
ON ERROR GOTO and RESUME, 229-233
ON. . . GOSUB, indexed subroutine branching, 89-90
ON. . . GOTO, Indexal branching, 71
OPTION BASE, 266-267
Order processing, 16

P

PEEK and POKE, 161-168
controlling external world with, 164-165
manipulating screen data with, 165-167
setting controls and options with, 167-168
POS, 218-219
PRINT USING, 209-212
Print with
commas, 30-33
semicolons, 33-34
Program
documenting, 56-58
expanding, 55-56
metric conversion, 76-80
structure, 198-207
Programming in direct mode, 35

R

Randomize, 85
RENUM, 184-185
Reserved words, 51
RESTORE statement, 137
RND, randomize function, 83-85
Rubik's cube 273-308
initial planning decisions, 275-290
bottom up program design, 285
flowcharting and top down program design, 285
picture on the screen, 275-279
representing the cube with an array, 280-281
representing the cube with strings, 283-284
naming parts of the cube, 274-275
writing the program
display routine, 293-297
getting ready to rotate, 297-299
main program, 291-292

Rubik's cube—cont
writing the program
move list, 292-293
rotation routines, 299-304
RLIN line number, 26-27

S

Scientific applications, 19-21
Semicolons, printing with, 33-34
Setting controls and options with PEEK and POKE, 167-168
SGN function, 151
Simple function statements, 81-87
SIN, 190-191
SIN, COS, and ATN, 146-149
SPACE$, 222-223
SPC, 225
SQR function, 150-151
Statement(s)
DATA and READ, 135-139
decision making, 63-72
RESTORE, 137
STOP and END, 27-28
String
functions, 152-158, 342-347
CHR$(expr), 342
HEX$(var), 344
LEFT$(str$, expr), 345
MID$(str$, expr1, expr2), 345
MID$(str1$, expr1, expr2) = str2$, 345
RIGHT$(str$, expr), 347
STR$(expr), 347
STRING$(expr, str$), 347
variables, 42-44
STRING$, 225-227
STR$ and VAL, 157-158
Subroutine
concept, 87-90
error-checking, 97-101
numeric round-off, 90-95
Substitute for INSTR, 221-223
SWAP, 267-268

T

TAB, 48-50
video plotting with, 188-191
TRACE and NOTRACE, 186-187
Type declaration characters, 242-244
double precision, 243
integer, 242
single precision, 243
string, 243

U

Using a lineprinter, 192-193

V

Variable(s)
names, 41-42
string, 42-44
types, 238-258
VARPTR, 258-265
Video plotting with TAB, 188-191

W

What CALL can do, 173-176
When to use an error-trapping routine, 237-238
When to use integers or single precision, 241-242
WHILE. . . WEND, 199-201
Word scrambling program, 248

Y

Your first program, 22-23

SAMS 22014

The Waite Group's BASIC Programming Primer

Second Edition

BASIC is the most popular language among computer users today. It is designed for students and professionals alike. This primer, therefore, should be an invaluable aid to anyone who wants to learn BASIC. Some of the features of this book are:

- Covers 16-bit BASIC statements, key words, commands
- Self-tests and answers
- Nonnumeric program examples: Micro Space Invaders; Word Scrambler; Rubik's Cube
- 50 new key words
- Covers the ABCs and XYZs of BASIC
- Compatible with IBM BASIC, but applicable to any computer that runs Microsoft BASIC (TRS-80®, Apple®, Commodore PET®, etc.)
- Starts from "square zero" and proceeds step-by-step
- Directed at beginners
- Used in classrooms across the USA

The Waite Group is a developer of computer, science, and technology books. Acknowledged as a leader in the field, The Waite Group creates book ideas, finds authors, and provides development support throughout the book cycle, including editing, reviewing, testing, and production control for each title. The Waite Group has produced over 70 titles, including such best-sellers as *New C Primer Plus, MS-DOS® Developer's Guide, Tricks of the UNIX® Masters,* and *Assembly Language Primer for the IBM® PC.* Mitchell Waite, president of The Waite Group, wrote his first computer book in 1976. Today The Waite Group produces 15 to 20 new computer books each year. Authors can contact The Waite Group at 100 Shoreline Highway, Suite 285, Mill Valley, California, 94941.

ISBN 0-672-22014-8

90000

9 780672 220142

$19.95 US / $25.95 CAN

BASIC LANGUAGE REFERENCE CARD

The following list contains all the BASIC keywords, statements, commands, and functions listed—for the most part—in alphabetical order (for the sake of logical clarity some keywords have been grouped in logical rather than alphabetical order). Each keyword is explained with typical programming notation showing the exact syntax of the keyword, a description of how the keyword works and an example of how it appears in a program statement.

This removable reference card allows you to easily look up a specific BASIC element and see how it is used correctly. In order to differentiate the commands, statements, functions, etc., we have used the labels below to indicate the different types of keywords.

Commands C
Statements .....S
Numeric Functions .....NF
String Functions .....SF
Logical Functions .....LF
Special Variables .....SV
Format Control .....FC

In the descriptions certain words are abbreviated for clarity. Their meanings are as follows:

Abbreviation	Full Word	Meaning
expr	expression	A constant, variable, or group of constants or variables connected optionally by mathematical operators (+, −, *, /).
var	variable	A BASIC variable.
num	number	Usually an integer constant,
incr	increment	A number, usually an integer.
str$	string	A BASIC string variable or constant.
list	list	A list of constants and/or variables.
line	line	A BASIC line number.

"A"

ABS (expr)

Returns the absolute value of expr. (NF)

300 IF ABS(X)> 5 THEN 400

AND (expr1 and expr2)

Returns the logical value TRUE if both expr1 and expr2 are TRUE; otherwise returns FALSE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then AND returns a bit-by-bit AND operation of the bytes representing expr1 and expr2. (LF)

100 IF A>B AND C<D THEN 500

ASC(str$)

Returns the decimal ASCII value of the first character of str$. (NF)

300 IF ASC(A$)>25 THEN 500

ATN(expr)

Returns the arctangent of expr. Expr is assumed to be expressed in radians in the range −PI/2 to PI/2. (NF)

200 PRINT ATN(X)

AUTO num,incr

Sets the automatic line numbering mode to start at line number num and to increment line numbers by incr. To exit from AUTO, type Control/C. (C)

AUTO 100,10

"C"

CALL expr

Causes BASIC program to branch to machine-language subroutine at memory location specified by the value of expr. When subroutine is terminated by a RETURN statement, execution continues at the next line after the CALL. (S)

200 IF A<500 THEN CALL PR+6

CDBL var

Converts variable var to double-precision variable. (S)

20 CDBL A

CHR$(expr)

Returns a single character corresponding to the ASCII code specified by value of expr. (ASCII codes are listed in Appendix B.) (SF)

300 PRINT CHR$(X+3)

CINT var

Converts variable var to integer variable. (S)

20 CINT A

CLEAR expr

Sets all program variables to zero. Optional expr sets data and string space to value of expr. (C)

CLEAR 200

CLOAD str$

Reads a program into memory from cassette tape with name str$. (C)

CLOAD "TEST"

CSAVE str$

Saves the current program in memory on cassette tape with name str$. (C)

CSAVE "TEST"

CONT

CONTinues program execution after a halt caused by a STOP or END statement, or after Control/C has been typed. (C)

CONT

COS(expr)

Returns the cosine of expr, where expr is assumed to be expressed in radians. (NF)

300 PRINT COS(X)

CSNG var

Converts variable var to single-precision variable, (S)

20 CSNG A

"D"

DATA<list>

Supplies values to be assigned to variables by READ statements. List elements are separated by commas. (S)

30 DATA 10,20,30,"FIRST","SECOND"

READ<list of variables>

Assigns values listed in DATA statements to numeric or string variables. Values are assigned in sequence beginning with the first value in the first DATA statement. (S)

40 READ X,Y,Z,A$,B$

RESTORE

Allows data in DATA statements to be reREAD. Next READ statement after a RESTORE begins assigning values from the first DATA statement again. (S)

200 RESTORE

DEF FNV<list of variables>=expr

Defines a user defined function with name FNV, where V is a variable name. Variables specified in the list are dummy variables. (S)

200 DEF FNAVE(A,B,C)=(A+B+C)/3

DEFINT letter1-letter2

All variables beginning with letter1 through letter2 are defined as integer type variables. (S)

50 DEFINT A-N

DEFSNG letter1-letter2

All variables beginning with letter1 through letter2 are defined as single-precision type variables. (S)

60 DEFSNG P-S

DEFDBL letter1-letter2

All variables beginning with letter1 through letter2 are defined to be double-precision type variables. (S)

70 DEFDBL T-V

DEFSTR letter1-letter2

All variables beginning with letter1 through letter2 are defined to be string type variables. (S)

80 DEFSTR W-Z

DELETE num1-num2

DELETEs lines numbered num1 through num2 from the program currently in memory; DELETEs only num1 if num2 not specified. (C)

DELETE 10-50

DIM var1(num1,num2,..),var2(num1,num2,..),...,str$(num),..

Allocates space for array variables and DIMensions these arrays. The values of the nums set the number of elements in a given dimension (up to a maximum of 255). (S)

20 DIM A(5,10,3),B(3,3),Z$(10)

"E"

EDIT num

Allows EDITing of line number num without affecting other lines in the program. (C)

EDIT 100

END

Stops program execution. Sends carriage return and BASIC's prompt character to screen. (S)

500 END

ERL

A variable which is automatically set to the value of the line number at which a fatal error has been flagged by the BASIC interpreter. (SV)

IF ERL<200 THEN RESUME 500

ERR

A variable which is automatically set to the value of any fatal error flagged by the BASIC interpreter. (SV)

IF ERR = 10 THEN RESUME 500

ERROR num

Causes the BASIC interpreter to act as if error number num has occurred. (C)

ERROR 20

EXP(expr)

Returns the value of e (the bose of natural logarithms) raised to the expr-th power. Expr must lie in the range between 0 and 87.3365. (NF)

300 PRINT EXP(COS(X))

"F"

FIX (expr)

Returns the integer portion of expr. (NF)

100 FIX(X/23)

FOR var = expr1 TO expr2 STEP expr3

Begins FOR...NEXT loop. Initializes var to expr1, then increments it by amount expr3 each time all statements through the corresponding NEXT are executed. Continues looping until var is equal to or greater than expr2 at which time execution is continued with the statement following the corresponding NEXT. (S)

FOR M = 20 TO 50 STEP 2

NEXT var1,var2,. . .

Ends FOR...NEXT loop. Var1 is the voriable of most recent loop, var2, next most recent, etc. NEXT without a variable terminates most recent loop. (S)

400 NEXT M

FRE(0)

Returns the number of unused bytes in memory. 0 is a dummy argument. (C)

FRE(0)

"G"

GOSUB line

Branches to BASIC subroutine starting at line number specified by line. (S)

250 GOSUB 5000

RETURN

RETURN ends a subroutine and causes execution to resume at the statement after the line which called the subroutine. (S)

350 RETURN

GOTO line

Causes program execution to continue on line specified by the value of line. (S)

200 GOTO 1000

"H"

HEX$(var)

Returns a digit string whose numerical value is equal to the hexadecimal equivalent of var. (SF)

200 A$ = HEX$(X)

"I"

IF expr THEN statement

IF expr is TRUE then execute statement (if statement itself is a number then GOTO line number); otherwise continue execution on to next line. (S)

300 IF A> B THEN PRINT B
400 IF X>10 THEN 250

IF expr THEN statement1 ELSE statement2

Like standard IF . . . THEN except that when expr is FALSE statement2 is executed. (S)

250 IF X>20 THEN 500 ELSE PRINT "ERROR"

INP(expr)

Reads a byte from port number specified by expr (which must lie in the range 0-255). (S)

300 Z = INP(T+3)

INPUT (list of variables)

Enters data into memory from keyboard device and assigns it in order to the variables on the list. Multiple numeric inputs are separated by commas; string inputs must be separated by carriage returns. (S)

300 INPUT A,B,C,D$,E$

INKEY$

A one character string variable which allows entry of a character from the keyboard without the necessity of following with a carriage return. (SV)

400 A$ = INKEY$

INPUT$(num)

A multiple character string variable which allows entry of num characters from the keyboard without the necessity of following with a carriage return. (SV)

400 A$ = INPUT$(5)

INSTR(num,str1$,str2$)

Searches for the string str2$ embedded in str1$ and returns the numerical position of the first match, or 0 if no match is found. Optional offset num sets the position for starting the search in str1$. (NF)

300 IF INSTR(X,A$,B$)>0 THEN PRINT "GOT IT"

INT(expr)

Returns the greatest integer less than or equal to expr. (NF)

200 Z = INT(100)

"L"

LEFT$(str$,expr)

Returns the leftmost expr characters of str$. (SF)

250 C$ = LEFT$(W$,8)+"XYZ"

LEN(str$)

Returns the length of str$. (NF)

300 IF LEN(N$)>20 THEN PRINT "TOO LONG"

LET var=expr

Assigns var the value of expr. (S)

150 LET X = Y+COS(Z)

LINE INPUT "prompt string";str$

PRINTs out the (optional) prompt string and assigns all input from the keyboard to str$ until a carriage return is entered. (S)

100 LINE INPUT "ENTER YOUR NAME,AGE";D$

LIST num1-num2

LISTs the current program in memory from line number num1 through line number num2. LIST alone lists the entire program. (C)

LIST 25-250

LLIST num1-num2

Same as LIST only output directed to lineprinter. (C)

LLIST 25-250

LPRINT

See entry under PRINT.

"M"

MID$(str$,expr1,expr2) (on the right of = 's)

Returns o string expr2 characters long from within str$, beginning with the expr1-th character of str$. (SF)

150 S$ = MID$("ABCDEFGHIJKLMN", X + 3,5)

MID$(str1$,expr1,expr2) = str2$ (on the left of = 's)

Beginning with the expr1-th character of str1$, replace the next expr2 characters by str2$. (SF)

150 MID$(A$,X,Y+2) = B$

MOD (expr1 MOD expr2)

Returns the remainder of expr1 divided by expr2. (NF)

200 IF (X MOD 2) = 0 THEN PRINT "EVEN"

"N"

NEW

Deletes the current program in memory and CLEARs all variables. Used before entering a new program. (C)

NEW

NEXT

See entry under FOR.

NOT expr

Returns opposite truth value of expr. If expr is numerical performs bit-by-bit complement on the byte representing expr. (LF)

"O"

ON expr GOSUB num1,num2,num3,. . .

Branches to BASIC subroutine specified by the expr -th number af the list num1,num2,num3 . . . (S)

300 ON ABS(X) GOSUB 101,102,103,104,105

ON expr GOTO num1,num2,num3,. . .

Same as ON. . . GOSUB except that branch is absolute. (S)

300 ON ABS(X) GOTO 101,102,103,104,105

ON ERROR GOTO line

On the occurrence of any error, transfers command to the statement number specified by line. (S)

250 ON ERROR GOTO 69

OR (expr1 OR expr2)

Returns TRUE if either expr1 or expr2 is TRUE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then OR returns the results of a bit-by-bit OR operation performed an the bytes representing the values of expr1 and expr2. (LF)

200 IF A<5 OR B>10 THEN PRINT "GOOD WORK"

OUT expr1,expr2

Sends byte specified by expr2 to port specified by expr1. (S)

150 OUT P,A+5

"P"

PEEK(expr)

Returns the value of memory location with address specified by expr. Expr must lie in the range 0-65535. (NF)

200 IF PEEK(I)>100 THEN GOSUB 500

POKE expr1,expr2

Stores the byte specified by expr2 —which must lie in the range 0-255 —in memory location specified by the value of expr1 —which must lie in the range 0-65535. (S)

150 POKE 4*M+1,240

POP

Increments nested GOSUB return stack address by 1. (S)

400 POP

POS(N)

Returns the current column position of the screen cursor; leftmost position equals 0. N is a dummy orgument. (NF)

100 CU = POS(Z)

PRINT var,..,expr,..,str$,..

Outputs data specified by list starting at the present cursor position. (S)

500 PRINT X,Y,Z,A+3,B$

LPRINT . . .

Same as PRINT except that output is directed to the lineprinter. (S)

500 LPRINT X,Y,Z,A+3,B$

PRINT@expr

Begin PRINTing at column number specified by expr. (S)

200 PRINT@ X + 3

PRINT USING str$,expr

PRINTs the value of expr in format specified by str$.(S)

250 PRINT USING "$$###.##", SA

"R"

RANDOMIZE

See entry under RND.

READ. . .

See entry under DATA.

RESTORE

See entry under DATA.

RENUM num1,num2,incr

Renumbers all line numbers in current program beginning with line number num2. The new sequence begins with num1 and lines are incremented by incr. If num2 is omitted entire program is renumbered; if incr is omitted, then increment is 10. (C)

RENUM 400,300,50

RESUME num

RESUMEs program execution at line number num after a halt generated by error trapping routine. If num is omitted, RESUMEs at next statement after error. (C)

RESUME 500

RETURN

See entry under GOSUB.

RIGHT$(str$,expr)

Returns the rightmost expr characters of str$. (SF)

200 PRINT RIGHT$(A$,5)

RND(expr)

Returns a (pseudo) random number between 0 and 1. If expr < 0, RND returns a new sequence of numbers; if expr = 0, RND returns the last number over again; if expr > 0, RND returns the next number of the current sequence. (NF)

500 IF RND(1)>.5 THEN 700

RANDOMIZE

Resets the random number generator to produce a new sequence of (pseudo) random numbers. (S)

400 RANDOMIZE

RUN num

Begins execution of program currently in memory at line number num. If num is omitted begins at lowest line number, (C)

RUN 250

"S"

SGN(expr)

Returns the sign of the value of expr: −1 if expr < 0 ; 0 if expr = 0; and 1 if expr > 0. (NF)

200 X = INT(X + .5*SGN(X))

SIN(expr)

Returns the sine of expr. Expr is assumed to be expressed in radians. (NF)

200 IF SIN(X) > .3 THEN 250

SPACE$(expr)

Returns a string of blanks expr spaces long. (SF)

150 X$ = A$ + SPACE$(I + 5)

SPC(num)

Used only to format output in PRINT statement. PRINTs a string of blanks num spaces long. (FC)

200 PRINT SPC(10);A$

SQR(expr)

Returns the square root of expr. Expr must be > 0 or =0. (NF)

250 Z = SQR(X*Y)

STOP

STOPs program execution; prints BREAK IN LINE nnn (nnn is line number at which program STOPped) and returns to command level. (S)

500 STOP

STR$(expr)

Returns the string representation of the value of expr. (SF)

300 PR$ = "$" + STR$(PR)

STRING$(expr,str$)

Returns a string expr characters long made up from the first character of str$. (SF)

200 PRINT STRING$(A,B$)

SWAP var1,var2

Exchanges the values of var1 and var2. Both must be of the same variable "type" or an error will result. (S)

300 SWAP A$,B$

SYSTEM

Causes an exit from BASIC system and transfers to operating system. (C)

SYSTEM

"T"

TAB(expr)

Used to format output in PRINT statement. Causes PRINTing to occur at column specified by value of expr. Leftmost position is 0. (FC)

250 PRINT TAB(I);A$;TAB(I+10);B$

TAN(expr)

Returns the tangent of expr, where expr is assumed to be expressed in radians. (NF)

300 IF TAN(X) > 1 THEN 500

TRON

Turns on the trace flag which causes BASIC to print each line number as that line is executed. (C)

TRON

TROFF

Turns off the trace flag. (C)

TROFF

"U"

USR(expr)

Calls a machine language subroutine at address specified by USRLOC and passes it the value of expr. (S)

200 USR(X*100)

USRLOC

A special location in memory which contains the starting address of a user's machine-language subroutine which will be called by USR. (S)

"V"

VAL(str$)

Returns the numerical value of the digit string str$. If str$ is not a digit, string returns 0. (NF)

250 IF VAL(X$) < 25 THEN 500

VARPTR(var)

Returns the memory address of the variable var. (NF)

AD=VARPTR(Y)

VTAB(expr)

Causes vertical TAB expr lines downward. (S)

200 VTAB(I+5)

"W"

WAIT port,num

Bit pattern in port specified by value of port is ANDed with value specified by num and execution is halted until logical TRUE results. (S)

300 WAIT 255,4

WHILE expr

Begins WHILE...WEND loop. If expr is TRUE execute all statements until the following WEND statement and continue looping until expr becomes FALSE; if expr is FALSE jump to statement following WEND. (S)

200 WHILE A < 200

WEND

Terminates a WHILE...WEND loop. (S)

500 WEND

WIDTH num

Sets the width of the PRINT line to num columns. (C)

WIDTH 45

"X"

XOR (expr1 XOR expr2)

Exclusive OR. Returns the value TRUE if expr1 and expr2 have different logical values, otherwise returns FALSE. If expr1 and expr2 are pure numerical (nonrelational) expressions, then XOR returns the results of a bit-by-bit Exclusive OR operation an the bytes representing the values of expr1 and expr2. (LF)

IF A<5 XOR B<5 THEN 450

"MISCELLANEOUS"

&H str$

Returns the decimal equivalent of the numerical value of str$, where str$ is assumed to be expressed in hexadecimal. (NF)

200 PRINT PEEK (&H7AB0)

&O str$

Returns the decimal equivalent of the numerical value of str$, where str$ is assumed to be expressed in octal. (NF)

300 PRINT &0177

VARIABLE TYPE DECLARATION CHARACTERS

Character	Example	Typical Values	Definition
$	A$,Z1$	"JOHN" "1+1+?"	String variable type may contain up to 255 characters.
%	%A1,B%	−14,345,6009	Integer variable type—whole numbers with absolute value less than 32767.
!	A!, A11	1,−.123,25.3067	Single-precision variable type containing six significant digits.
#	A#,Z1#	−300.12345678 3.141592653589 1.0000010001234	Double-precision variable type containing 14 to 16 significant digits.
D	A# 1.2345678901 D 12	1.2345678901 X 10^{12}	Double-precision variable type with scientific notation.