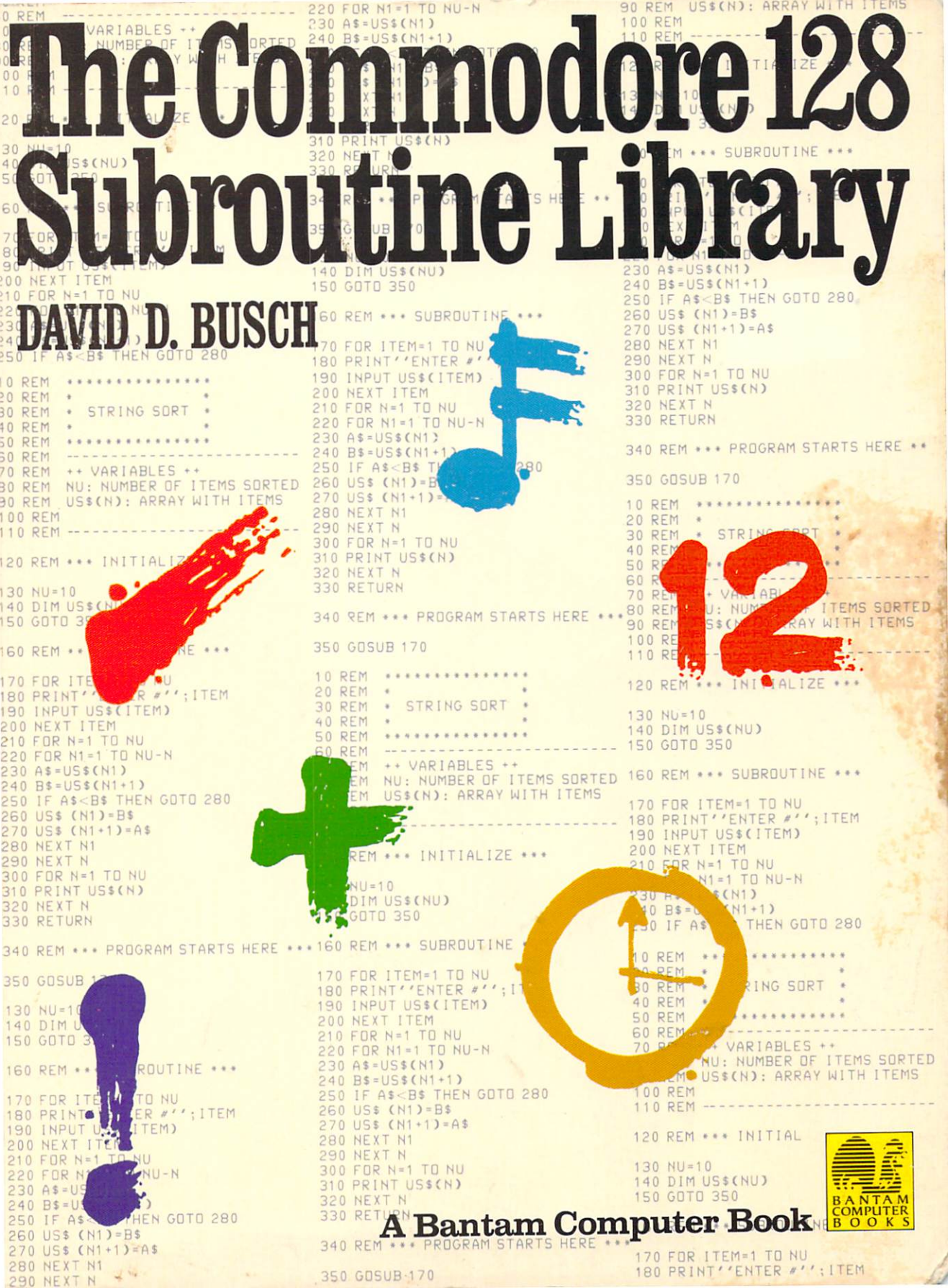
The Commodore 128 Subroutine Library

DAVID D. BUSCH



A Bantam Computer Book

THE COMMODORE 128 SUBROUTINE LIBRARY

Bantam Computer Books
Ask your bookseller for the books you have missed

THE AMIGADOS MANUAL
by Commodore-Amiga, Inc.
THE APPLE //c BOOK
by Bill O'Brien
THE ART OF DESKTOP PUBLISHING
by Tony Bove, Cheryl Rhodes, and Wes Thomas
ARTIFICIAL INTELLIGENCE ENTERS THE MARKETPLACE
by Larry R. Harris and Dwight B. Davis
THE BIG TIP BOOK FOR THE APPLE II SERIES
by Bert Kersey and Bill Sanders
THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch
COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
by Commodore Business Machines, Inc.
THE COMMODORE 128 SUBROUTINE LIBRARY
by David D. Busch
THE COMPUTER AND THE BRAIN
by Scott Ladd / The Red Feather Press
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR COMMODORE 64
by Tim Hartnell
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR IBM PC
by Tim Hartnell
EXPLORING THE UNIX ENVIRONMENT
by The Waite Group / Irene Pasternack
FRAMEWORK FROM THE GROUND UP
by The Waite Group / Cynthia Spoor and Robert Warren
HOW TO GET THE MOST OUT OF COMPUSERVE, 2d EDITION
by Charles Bowen and David Peyton
HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton
THE IDEA BOOK FOR YOUR APPLE II
How to Put Your Apple II to Work at Home
by Danny Goodman
THE MACINTOSH
by Bill O'Brien
MACINTOSH C PRIMER PLUS
by The Waite Group / Stephen W. Prata
THE NEW *jr*: A GUIDE TO IBM'S PC*jr*
by Winn L. Rosch
ORCHESTRATING SYMPHONY
by The Waite Group / Dan Shafer with Mary Johnson
PC-DOS / MS-DOS
User's Guide to the Most Popular Operating System for Personal Computers
by Alan M. Boyd
POWER PAINTING: COMPUTER GRAPHICS ON THE MACINTOSH
by Verne Bauman and Ronald Kidd / illustrated by Gasper Vaccaro
SMARTER TELECOMMUNICATIONS
Hands-On Guide to On-Line Computer Services
by Charles Bowen and Stewart Schneider
SWING WITH JAZZ: LOTUS JAZZ ON THE MACINTOSH
by Datatech Publications Corp. / Michael McCarty
UNDERSTANDING EXPERT SYSTEMS
by The Waite Group / Mike Van Horn
USER'S GUIDE TO THE AT&T PC 6300 PERSONAL COMPUTER
by David B. Peatroy, Ricardo A. Anzaldua, H. A. Wohlwend, and Datatech Publications Corp.

THE COMMODORE 128 SUBROUTINE LIBRARY

David D. Busch

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

As always, for Cathy.

THE COMMODORE 128 SUBROUTINE LIBRARY
A Bantam Book / August 1986

Cover design by J. Caroff Associates, Inc.

Book design by Nicola Mazzella

ISBN 0-553-34308-4

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

B 0 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

Someone once said, "Time is a great teacher, but, unfortunately, it kills all its pupils."

Although it is possible to learn on our own, letting others invent the wheel and then pass their experience on can be a better use of our most precious commodity.

This collection of subroutines for the popular Commodore 128 personal computer has two purposes. You'll find nearly 100 useful, ready-to-transplant subroutines and programming tips that will enable your own programs to solve tough business problems, resound with music, or sizzle with joystick action. Although this collection is not intended as a "how-to" programming guide, we

think you'll pick up a wealth of useful guidelines and tricks along the way.

The Commodore 128 Subroutine Library is intended as a companion to the *Commodore 128 System Guide* provided with your computer, and more advanced texts like *Commodore 128 Programmer's Reference Guide.* Many of the concepts you learn in those manuals are applied here in simply constructed subroutines, with only one or two statements per line for easy comprehension. Grouped by function, carefully annotated, and arranged to be readily dropped into your own BASIC software, these subroutines should save you time while sharpening your programming skills.

Time may be an excellent teacher, but the course work is more easily absorbed when you have study aids like this one.

INTRODUCTION

BASIC subroutine books have been around as long as personal computers have. However, this one is unlike any collection of subroutines that you might have seen before. You'll find eight dozen program modules you can really *use*, formatted for maximum clarity, explained completely, and rigorously tested. Most important, these subroutines were designed for optimal flexibility, so that you can easily adapt them to your own programs. These routines supply the most-needed business formulas, common software tricks for the Commodore 128, and graphics and sound routines that can serve as a springboard to more advanced programming.

You'll learn how to figure how much money can be drawn out of an annuity each month, how to program your own custom character sets, how to really *use* the Commodore 128's function keys.

What you won't find are chapters top-heavy with exotic math functions and rarely used statistical programs. Those were fine back in the days when microcomputers were used primarily by scientists, computer nuts, and other high-tech types who doted on newer and better ways of doing Fast Fourier transforms.

However, the Commodore 128, while it is a powerful, capable microcomputer, is being sold to a broad range of users, including many nontechnical owners. Some want to use the broad range of software available for the Commodore 128 and Commodore 64 to do word processing or database management or to mull over electronic spreadsheets. Others want only to play games, since the Commodore 128, with its three synthesizer-quality voices, sprites, and bit-map graphics, is a games machine *nonpareil.* Many more owners are interested in learning programming but may have a skimpy technical background. Then there are those of you who really do understand computers but would like to avoid reinventing the wheel.

The Commodore 128 Subroutine Library is meant for all of you. There are some general, useful routines included here, but the book also bristles with modules designed specifically to perform some sorely needed task for the Commodore 128 alone.

While BASIC 7.0 has added dozens of new commands to those available with the BASIC 2.0 offered with the Commodore 64, some important statements and functions common to other BASICs have been left out. This book shows you how to add to your computer the STRING$, MID$=, LINE INPUT, and SWAP (we call it EXCHANGE) commands available on the IBM PC. Better yet, you can add some *new* functions, like REPLACE and INSERT STRING, not available on any other machine!

Are you confused by even the most lucid explanations of using the joysticks to manipulate objects on the screen? Just transplant one of *ten* joystick routines included in this book.

If you find that even the Commodore 128's several hundred alphanumeric, graphics, and special characters aren't enough for you and you'd like to design your own special characters, take heart. You don't need to comprehend all the gory details. A module is included which you can use to redefine up to five characters with no problems. Using the Commodore 128's real-time clock to measure elapsed time or to control outside events is also provided for. Generate musical notes within your own programs—or add sound effects. Ready-made subroutines are provided for your use.

Intermediate and advanced programmers will find tips on routines that spice up their own arcade-quality games, while those interested in programming for business will revel in the user-friendly input routines, menus, and sort routines.

More advanced programmers can use several routines as utilities to make their work easier while doing sophisticated "soft" POKing of individual bits within a multipurpose Commodore 128 register.

We've gone light on the "basic" subroutines, although plenty of the more important conversion and financial routines are provided. The emphasis here is on modules you can't find anywhere else, and those tailored specifically for the Commodore 128, that will help you improve your programming immediately.

SOME GROUND RULES

• First, this is a BASIC subroutine guide. You don't have to know the first thing about machine or assembly language to use any of the modules in this book. We won't ask you to call up the MONITOR, and you needn't type in interminable DATA lines that will be POKEd into memory as machine-language code. Some functions, particularly sorting, are fastest when handled by machine language calls. However, if you choose to stick with BASIC, you'll find three sorting routines in this book that are perfectly usable for small lists.

• This book is intended for owners of the Commodore 128 only. In computer terms, the functionality (power) of the Commodore 128 is a *superset* (enhancement) of that of the Commodore 64. In other words, just about everything the Commodore 64 will do, the C128 will do as well. Most C64 software will run as is. However, the Commodore 128 has many new features (such as built-in 80-column screen output) and a raft of more powerful BASIC commands. So, while the C128 can run C64 programs, many intended for the Commodore 128 *cannot* be used on the Commodore 64.

Many of the subroutines in this book will work on the Commodore 64, but many more will not. No special effort was made to write the code to retain compatibility with the earlier computer. If BEGIN/BEND was a more efficient use of the IF/THEN conditional statement, we didn't avoid it. Sometimes a few changes will convert a subroutine for proper operation on the Commodore 64, but readers who want to try are on their own. In a couple of specific cases, routines were written to be compatible with the earlier computer, and they are so noted in the text.

• "Sample runs" are used only in selected cases. For many of the subroutines, such as the sound and graphics routines, presenting an image of what the screen looks like is difficult and not very useful. In other cases, sample output is downright dumb, because any intelligent user will know at a glance whether or not a sorted list is sorted properly, or whether or not an array is properly loaded.

Sample *results* are useful for those who want to double-check to see if they have typed in a subroutine properly, and they are provided in those cases, particularly in the business routines. If you use the variable values we provide in this book, you should achieve the same results we did.

• Again, we emphasize that this book is not intended as a how-to-program tutorial. Many of the techniques are explained in detail, but you should already have some experience in programming to get the most from them. Much of the time you will be able to substitute your own variable values for those provided

and to use the subroutines without fully understanding the nuts and bolts of what is going on. However, some understanding of programming is a prerequisite for this book.

Certain capabilities of the Commodore 128 are beyond the scope of this book, even though they are readily accessible from BASIC. Advanced music and sound techniques could easily encompass a book of their own. This book provides an introduction; learning to shape your own ENVELOPE can be picked up from *Commodore 128 Programmer's Reference Guide*, also available from Bantam Books.

Sprite and multicolor bit-map graphics also are worth more pages than are allotted here. Fortunately, the Commodore 128 has built-in sprite definition and manipulation commands that make these tools easier to use than ever. This book's graphics chapter has a routine, Shape Mover, that demonstrates how objects (not just sprites) can be drawn on the screen and stored in a string variable. If this introduction to sprite concepts piques your interest, *Commodore 128 Programmer's Reference Guide* has a lot more excellent information.

1 SUBROUTINE MAGIC

While many of the subroutines in this book are ready-to-run programs in their own right, they will be most useful to you when you transplant them into your own programs. We've made doing that as simple as possible.

The subroutines have been divided into sections, set off with REMark statements. The basic routine itself is clearly labeled and has a unique set of line numbers not duplicated by any other subroutine in this book. The line numbers are all high, beginning with 10000 and proceeding to nearly 30000. Therefore, you can use as many of these subroutines as you wish in a program without having to renumber them.

As a matter of fact, you may find it most convenient *not* to renumber the subroutines at all during program development. With their original line numbers intact, you can refer to a list you have compiled of routines being used (or check this book) to see whether you should type GOSUB 20100, or GOSUB 21000 to access a given routine.

You might need to renumber your program during development for some reason. Perhaps you ran out of line numbers in the main body of the program and need more room. You can still renumber without "losing" the location of your routines. Type a few lines at the beginning of the program like these:

```
1 GOTO 10
2 GOSUB 27100:REM TIMER SUBROUTINE
3 GOSUB 28200:REM CURSOR MOVER
4 GOSUB 20200:REM RANDOM RANGE
5 GOSUB 17000:REM JOYSTICK ROUTINE
10 *** START OF PROGRAM ***
```

When you are ready to renumber, you type:

```
RENUMBER 10,10,10
```

Your program is numbered in increments of 10, beginning with Line 10. Anytime you want to find your subroutines, just type:

```
LIST 1-10
```

You'll see something like this:

```
1 GOTO 10
2 GOSUB 1010:REM TIMER SUBROUTINE
3 GOSUB 850:REM CURSOR MOVER
4 GOSUB 700:REM RANDOM RANGE
5 GOSUB 620:REM JOYSTICK ROUTINE
10 *** START OF PROGRAM ***
```

The program lines 2–5 are never called, since Line 1 jumps control over them. However, the Commodore 128 will obediently renumber them to account for the new line numbers of your

subroutines. If you want to get really fancy, try programming a function key to display those lines whenever you want. In command mode type:

```
KEY 8,CHR$(147)+"LIST 1-4"+CHR$(13)
```

and press RETURN. Thereafter, pressing the F8 function key will show you the current addresses. This particular line will clear the screen first (with CHR$(147)), so you don't even have to move the cursor down to an empty line to use it.

PARTS OF THE SUBROUTINE

In addition to the subroutine itself, each module will have a REMark section that lists the major variables that you supply to the subroutine and the major variables that are returned from the subroutine. Modules that don't have variables of use to the programmer are instead described in terms of action needed and result.

SUPPLIED BY USER lists variables that *you* must define before calling the subroutine. You may need to store in the variable the amount of loan, or the interest rate. You may do this anywhere in your program prior to calling the subroutine.

RESULT lists the variables returned from the subroutine. These will generally be calculated and may then be used in your own program. In some cases, there are other variables used within the subroutine temporarily that are of no particular use to the programmer. These are *not* described in the text.

IMPORTANT NOTE: Variable names were chosen, where possible, to provide some clue as to the use of the variable. In addition, we tried to choose names different from those used in similar routines where the functions were different. However, it was impossible to cross-reference every variable name used in every subroutine to make sure there was no duplication.

In constructing programs, you should check to make sure that the same variable name is not being used by two different

subroutines in a way that will cause incorrect results. Keep in mind that only the first two characters of a Commodore 128 variable name are significant. Variables PAYMENT, PAID, PATIO, and PAYCHECK are all the *same* variable name to BASIC 7.0. Further, you should keep a list of the variables you use in your own code, to make sure you are not duplicating names. This is probably one of the most common errors programmers make. Cross-reference utilities that list all variables and their line numbers are handy tools to have.

In the "Initialization" section of the module, certain conditions that must be set up once are established. For example, DATA used by the subroutine may be read into an array. Initialization will also include defining variables supplied *to* the subroutine. You must take care of inserting these somewhere in your program so the subroutine will operate properly. What you must do is described in the text accompanying each routine.

In some cases there are several related routines. For example, there are ten joystick routines. Some of the concepts are explained only once. You may be directed to look at previous subroutines for longer explanations. In this way you can access the routines in any order, without reading the entire book.

Previous Commodore books you may have seen have had program listings that were often somewhat difficult to read. The reason for this is that Commodore uses a variety of special symbols to indicate various screen color and cursor movement options. A heart symbol is used for "clear screen" (we C128 users now have the SCNCLR option, instead), the reversed "Q" represents "Cursor Down," and so forth. Listings with these symbols must be output with a dot-matrix printer, which is often *near* letter quality but not close enough to stand up under the printing reproduction process.

All the odd special Commodore 128 characters have been left out of this book. Instead, we use their CHR$ equivalents, often by defining a variable with that value. PRINT DN$ (where DN$= CHR$(17)) is preferable to hard-to-read reversed Q's, or strange word-equivalents such as {3 CRSR DWNS} used in some of the magazines. We like the variables method so much that we de-

signed a special cursor movement subroutine (found in Chapter 8) that makes moving around the screen as simple as a single line:

```
100 PRINT HM$;LEFT$(R$,COL);LEFT$(D$,ROW)
```

Screen-clearing, cursor movement, and colors have been added only where necessary in these subroutines, to keep the length down and to allow the user to tailor modules to suit his or her own taste. If you want prompts to feature reversed printing or attention-grabbing color, feel free to make any changes you wish. You might also want to follow our example of using variables (RED$, etc.) rather than odd symbols to make your program listings more readable. This is a subroutine cookbook; the finishing touches of the meal are up to you.

THE ACCOMPANYING TEXT

Each subroutine is preceded by descriptive text that provides more information on the module and how to use it. At the top of the description is the subroutine *name*. Next is a short WHAT IT DOES summary of the routine's function. LEVEL tells you the approximate programming prowess needed to understand and, presumably, use the routine effectively.

Most of the subroutines fall into the *intermediate* category. It is assumed that users have a basic understanding of the most-often-used BASIC 7.0 statements and functions. Some of the subroutines have been deemed *novice* level. Readers who have been programming awhile will probably find such routines as MPG calculators, Menu formatters, and Exchange functions a bit elementary. However, beginning programmers find these simple routines as valuable as the more advanced folks hate them. Therefore, you BASIC veterans should be patient and remember there was once a time when you found things as simple as a FOR-NEXT loop a bit puzzling.

Because most of this book is written for the intermediate programmer, there are only a few routines that are labeled for

advanced programmers. This is something of a catch-22. Those who are really advanced have little need for prepackaged routines that PEEK bits and so forth. Those who need those functions can probably write their own, or can access the Commodore 128 Kernal routines from BASIC as quickly as they can look up something in this book.

Nevertheless some more advanced modules are provided, and these can serve as a bridge for the intermediate programmer who is on the threshold of true computer expertise and needs some additional help. When you reach this advanced stage, you'll want to move on to more sophisticated sprite and sound techniques such as those found in *Commodore 128 Programmer's Reference Guide.*

After LEVEL you'll find a HOW TO USE IT discussion of the subroutine. This section may be very brief or more lengthy, depending on the complexity of the concepts being introduced. While this book doesn't have the space for truly comprehensive coverage of all the BASIC techniques used, you'll find this section a good introduction in many cases.

LINE-BY-LINE DESCRIPTION is a brief description of the function of major subroutine lines. Not every line is discussed (the function of 300 END being obvious to every reader, we hope), but the key points are covered.

SUGGESTED ENHANCEMENTS points out improvements or modifications the reader can add when including the routine in his or her own program. Some of the subroutines, particularly the graphics and joystick modules, have rather lavish enhancements built in, at least relative to subroutines. Others are more bare-bones in scope. Users can expand these with their own changes, incorporating other subroutines in the book, or doing additional work.

Readers of these subroutine books enjoy making such modifications, especially when suggested enhancements are pointed out. You may view these recommendations as programming challenges that will help you sharpen your BASIC skills. For example, the Number of Days subroutine, as written, will calculate the number of days within a single year. The accompanying text

provides step-by-step hints on changing the routine to work when the dates span any number of years.

The suggested changes in the early part of the book are relatively similar, since the financial routines perform variations on only a few different functions. However, the modification ideas get livelier as the book progresses.

Note that not every subroutine takes care of every possible eventuality through error traps. We've tried to take care of the most obvious user errors. However, checking for every nonsensical input (such as the user's supplying negative values for interest rates, principal amounts, or loan payments in the financial subroutines) would have two negative impacts. First, the subroutines would be so long and unwieldy that the reader would hesitate to type them in. Also, the extraneous error traps would make the routines more difficult to understand.

Therefore, bear in mind that pounding randomly on the keyboard and attempting to "trick" the subroutine *may* work. You can use the "bombproof" data entry routines in the book, or devise your own to filter out foolish entries.

RESULT is a short summary outlining what happens after the routine has run.

SAMPLE VALUES, the results obtained when you use the variables as shown in the subroutine, are supplied for users who wish to make sure they have typed in the subroutine properly. While not sure-fire, comparing results will show if you are on the right track.

About all that repetition . . .

Some readers may feel that the accompanying text for the modules sometimes gets repetitious. The intent of this book is to allow the various subroutines to stand alone as much as possible, so basic information relevant to successive similar subroutines is sometimes repeated. The reader does not have to flip back and forth between subroutines to learn how a particular one operates. On the other hand, more in-depth explanations of concepts are usually *not* repeated, and the reader is directed to a previous routine for a more thorough discussion. We hope a workable

compromise has been reached between avoiding repetition and forcing excessive page flipping.

Along the same lines, for consistency's sake, each subroutine is presented in more or less the same format. Some subroutines and their descriptions lend themselves to the format more than others. Therefore, you may find some SUGGESTED ENHANCEMENTS that are brief, and SAMPLE VALUES listed as not applicable. In a few cases the modules themselves are standalone programs rather than subroutines and are not arranged in the traditional GOSUB/RETURN fashion.

ACCESSING THE SUBROUTINE LIBRARY

The only remaining point to cover is how to access your subroutine library. One of the best things about subroutines is that they can be reused many times within an existing program, and put to work in many different pieces of software as well. Once you have typed in, say, a joystick routine from this book, you will not need to retype it every time you write a new program requiring joystick handling. Because the subroutines in this book have been designed as standalone modules, with both the input and output clearly defined, they can be recycled quite easily. You will want to store your subroutine "library" on disk or tape and use them in your programs as needed.

Incorporating existing code into a program is called *merging* and can be accomplished in many different ways. The best way, which we highly recommend, is to keep all these subroutines as a single program file on your disk. Since they all have unique line numbers, there will be no conflict.

Note that you don't have to type them *all* in. Type only those you need. Reload your subroutine file each time you add a new one. That way you will maintain a growing subroutine library in one place.

When you start to write a program, load your library first and then write all the program lines using line numbers lower than 10000. Look up a subroutine's starting line number and

other requirements in this book, and access them as you wish. When a program is completely finished, you may delete those line numbers containing subroutines you don't want. Generally, this will not be a chore, as large groups of routines not used can be deleted with a few commands.

You could also keep subroutines as separate files or groups and merge them using one of the commercially available merging routines. Another technique, useful for short routines, is to load the module and LIST it to the screen, then load the main program. You can move the cursor up to the listing and hit RETURN while the cursor is on each line. The subroutine, while gone from program memory, is still in screen memory and can be merged back into your BASIC program in this way. This technique works only for subroutines that can be listed with a single screenful, but that requirement happens to encompass most of the routines in this book.

Good luck. You should find this book a shortcut to programming proficiency. To paraphrase a common saying, if you use a subroutine correctly three times, it will be a permanent part of your vocabulary. Given a bit of practice, you can soon have all your friends drooling over your programs and asking you for your favorite subroutine recipes.

2 BUSINESS AND FINANCIAL

Business and personal financial problem solving has become the most prevalent application for personal computers among adults. As recently as five or six years ago this was not the case. The earliest personal computers were owned by hobbyists, frequently those who migrated from ham radio or electronics and who were pleased to get their computers to do *anything*. Even then, a few hardy souls attempted to channel their new-found computer power into business applications, such as primitive word processing.

The introduction of VisiCalc, the first of a new generation of microcomputer-oriented management decision-support tools, nudged the business community toward the acceptance of these small,

desktop computing machines. In 1981 the advent of the IBM PC helped legitimize personal computers among major corporations.

Today very low-cost machines like the Commodore 128 bring this tool within the reach of any business or individual. For less than $900 for a computer/1571 disk drive/1792 monitor package, users have available an excellent, business-quality keyboard, highly legible 80-column viewing, and floppy disk storage equal to a single-drive IBM PC costing twice as much. As more business software becomes available for the Commodore 128, we'll see these computers put to wider use by budget-conscious individuals.

Some users will also be writing their own programs and routines in BASIC 7.0. Business programs have some things in common with games and utilities in BASIC; they also have their own special requirements. A business application will rarely deal with RND but will often have to handle dollars-and-cents. Money matters—figuring loan amounts, monthly payments, interest—and formatting of the output are all important considerations. Business applications also involve keeping track of the date or time in order to pinpoint when a transaction took place.

The business subroutines in this book are not limited to those in this chapter. When writing your own programs, you might want to take advantage of special user input routines found in Chapter 3, or sorts, like those in Chapter 4. The Function Key routines in Chapter 8 can make your programs friendlier, and the Cursor Mover will help you design your screen displays.

This chapter concentrates on financial algorithms useful for figuring mortgage or automobile payments, the number of payments, or the interest accrued. You'll find a routine that tells you how much money can be withdrawn to deplete an annuity fund in a given span of years. Depreciation, using simple straight-line calculation, is also covered.

Typical formulas that might be needed, such as converting Fahrenheit to Celsius or calculating an automobile's MPG, are included. One subroutine shows a sample menu that you can adapt to your own programs by substituting options of your own devising. State abbreviations, and routines to calculate the num-

ber of days between dates and to convert dates from one format to another, round out this chapter.

Of course, data files are a key to storing and retrieving business information. The last two routines in this chapter are disk write/disk read subroutines that you can adapt to your own programs to save and access your files.

IMPORTANT NOTE:

The caret symbol (^) in the program listings represents the *up arrow* key on the Commodore 128, which is located between the asterisk (*) and the RESTORE key. Enter the symbol by typing the up arrow.

LOAN AMOUNT

WHAT IT DOES: Calculates size of loan, given monthly payment, interest rate, and length of loan.

LEVEL: Intermediate

```
100 REM ***************
110 REM *               *
120 REM * LOAN AMOUNT   *
130 REM *               *
140 REM ***************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM    SUPPLIED BY USER --
180 REM      RATE:      INTEREST RATE
190 REM      NUMBER:    MONTHS OF LOAN
200 REM      PAYMENT:   MONTHLY PAYMENT
210 REM    RESULT --
220 REM      LOAN:      LOAN AMOUNT
230 REM
240 REM --------------------------------

250 REM *** INITIALIZE ***

260 RATE=10
270 PAYMENT=10
280 NUMBER=36
```

```
290 GOSUB 10010
300 PRINT "LOAN AMOUNT : ";LOAN
310 END

10000 REM *** SUBROUTINE ***

10010 RATE=RATE/1200
10020 LOAN=PAYMENT*(1-(1+RATE)^-NUMBER)/RATE
10030 LOAN=INT(LOAN*100)/100
10040 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the maximum amount of money that can be borrowed, given a fixed interest rate, the desired monthly payment, and the months the loan will run.

You might use this subroutine to calculate how expensive an automobile you can buy given, say, a 36-month repayment period, a 15 percent interest rate, and the top monthly payment you can afford, say, $200. In this case, the subroutine would deliver the answer: $5769. Since very few cars can be purchased for that little, you might want to play with the figures a bit. What if a 48-month loan is taken out instead? In that case a more reasonable $7186 can be borrowed.

Having these figures available allows you to make some intelligent decisions. For example, extending the loan by 12 months provides $1417 more principal to borrow, but at the cost of $2400 in additional payments ($200 × 12). Is the purchase worth an additional $1000 in interest? Or can you finance the auto by finding the extra $1400 from some other source, such as by trading in a third car that you had planned on keeping an extra year? Or should you shop a bit more extensively for a better interest rate? If your credit union offers a bargain-basement 12 percent interest rate, you can borrow $7594 at the same interest rate, more than $400 more without increasing the monthly payment.

Or, if you already have the car picked out, this routine will tell you how much down payment you will have to come up with to make up the difference between the loan amount and the price of the car.

LINE-BY-LINE DESCRIPTION

Lines 260–280: Define the interest RATE, monthly PAYMENT you can afford, and the NUMBER of payments to be made. Your program can substitute INPUT lines to receive these figures from the user.

Lines 290–310: Access subroutine and display the result.

Line 10010: Change yearly interest rate in whole percent to decimal figure per month, e.g., 12 percent equals 12/1200 or .01 per month.

Line 10020: Calculate loan AMOUNT.

Line 10030: Round off AMOUNT. With the Commodore 128 you may also use PRINT USING for formatting numbers, such as dollars-and-cents. However, this rounding method is used to show you how the rounding can be done. Note that in this case the odd cents are merely cut off (truncated) at the decimal point. In the next subroutine PAYMENT AMOUNT, the method for correctly rounding up or down is shown. Either of these techniques can be used with the Commodore 64, which did not have PRINT USING available in its BASIC.

YOU SUPPLY

You must define these variables: PAYMENT (the monthly payment desired), RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent), and NUMBER (number of months loan will run). The subroutine will return LOAN, or the maximum loan amount, given those parameters.

SUGGESTED ENHANCEMENTS: Add your own bombproof input routines, based on those in Chapter 3.

RESULT

Loan amount calculated.

SAMPLE VALUE: $309.91

PAYMENT AMOUNT

WHAT IT DOES: Calculates monthly payment, given interest rate, number of payments, and loan amount.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * PAYMENT AMOUNT *
130 REM *                  *
140 REM ******************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    RATE:     INTEREST RATE
190 REM    LOAN:     AMOUNT OF LOAN
200 REM    NUMBER:   MONTHS OF LOAN
210 REM  RESULT --
220 REM   PAYMENT:   MONTHLY PAYMENT
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 LOAN=100
270 RATE=10
280 NUMBER=36
290 GOSUB 10110
300 PRINT "AMOUNT OF PAYMENT : ";PAYMENT
310 END

10100 REM *** SUBROUTINE ***

10110 RATE=RATE/100
10120 PAYMENT=LOAN*(RATE/12)/(1-(1+(RATE/12))^-NUMBER)
10130 PAYMENT=INT((PAYMENT+.005)*100)/100
10140 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the monthly payment, given a fixed interest rate, the loan amount, and the months the loan will run.

You might use this subroutine to calculate your monthly auto payment given, say, a 36-month repayment period, a 15 percent interest rate, and an amount to be financed of, say,

$8000. It will produce the answer, $277. By shopping around for different interest rates, or by varying the number of payments, you can calculate the effect on your monthly payment until a satisfactory amount has been worked out.

The subroutine would also be valuable for those considering consolidating a number of debts. Add up the current payoffs of the loans you wish to combine and then use this subroutine to calculate how much your new monthly payment will be.

LINE-BY-LINE DESCRIPTION

Lines 260–280: Define the amount of the LOAN, the interest RATE in whole percent per year, and the NUMBER of monthly payments. Your program can substitute INPUT lines to have this information entered by the user.

Lines 290–310: Access subroutine and display result.

Line 10110: Change RATE to percentage.

Line 10120: Change months to YEARS.

Line 10130: Calculate PAYMENT.

Line 10140: Round off PAYMENT to two decimal places. Note how this routine differs from that in LOAN amount. The decimal fraction .005 is first added to the PAYMENT before the integer portion is taken, rounding *up* numbers .006 or higher. That is, .00612 would become .01012 and would be rounded off to .01. The figure .00412 would be increased to .00912, and would be rounded off to .00.

YOU SUPPLY

You must define these variables: LOAN (the original amount to be financed), RATE (interest rate in percent, i.e.,10.5 equals 10.5 percent), and NUMBER (number of months loan will run). The subroutine will return PAYMENT, which is the monthly payment, against principal and interest.

SUGGESTED ENHANCEMENTS: Add your own bombproof input routines, based on those in Chapter 3.

RESULT

Loan payment calculated.

SAMPLE VALUE: $3.23

NUMBER OF PAYMENTS

WHAT IT DOES: Calculates number of payments given interest rate, monthly payment, and loan amount.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                    *
120 REM * NUMBER PAYMENTS *
130 REM *                    *
140 REM ********************
150 REM -----------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    RATE:     INTEREST RATE
190 REM    LOAN:     AMOUNT OF LOAN
200 REM    PAYMENT: MONTHLY PAYMENT
210 REM  RESULT --
220 REM   NUMBER:    TOTAL NUMBER OF PAYMENTS
230 REM   EXTRA:     FINAL PAYMENT
240 REM
250 REM -----------------------------------

260 REM *** INITIALIZE ***

270 LOAN=1500
280 RATE=12
290 PAYMENT=100
300 GOSUB 10210
310 PRINT NUMBER-1;" WHOLE PAYMENTS OF ";:PRINT USING
CASH$;PAYMENT
320 PRINT "AND  ONE PAYMENT OF ":PRINT USING CASH$;EXTRA
330 END

10200 REM *** SUBROUTINE ***

10210 CASH$="$####.##"
10220 RATE=RATE/1200
10230 NUMBER=LOG(PAYMENT/(PAYMENT-LOAN*RATE))/LOG(1+RATE)
10240 EXTRA=PAYMENT*(NUMBER-INT(NUMBER))
10250 NUMBER=INT(NUMBER)+1
10260 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the number of payments, given a fixed interest rate, the loan amount, and the monthly payment required.

You might use this subroutine to calculate how long your auto loan will run, given an interest rate of, say, 15 percent, a loan amount of $8000, and a monthly payment of $250. Since most automobile loans are for fixed periods of 18, 24, 36, or 48 months, the figures will be approximate. That is, an answer of 41 months will be produced using the 15 percent/$250/$8000 example. So, you will know that you can borrow somewhat more than $8000 for 48 months, or somewhat less for 36 months.

More commonly, you will use this subroutine to figure out how long it will take to pay off a debt, such as a credit card account, with an open-ended number of payments. If your charge card balance is $3000 and you plan on making $150 monthly payments until it is paid off, given an 18 percent monthly interest rate, the program will inform you that it will take 24 months to dispose of the balance.

LINE-BY-LINE DESCRIPTION

Lines 270–290: Define the amount of LOAN, the interest RATE in whole percent, and the monthly PAYMENT desired. Your subroutine can substitute INPUT statements to have this information supplied by the user.

Lines 300–330: Access subroutine and print result in whole and partial payments.

Line 10210: Set up CASH$ as a PRINT USING format. This is used as a faster way of formatting dollars-and-cents output. You should set up your PRINT USING format to take into account the largest amount of money you expect to handle in your routine. PRINT USING is discussed more completely on page 78 of your System Guide.

Line 10220: Change RATE to monthly decimal value, that is, 12 percent per year equals 12/1200 or .01 per month.

Line 10230: Calculate number of payments.
Line 10240: Calculate final, partial payment.
Line 10250: Figure number of whole payments.

YOU SUPPLY

You must define these variables: LOAN (the original amount to be financed), RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent), and PAYMENT (the amount of the monthly payment). The subroutine will return NUMBER, which is the number of monthly payments that will be required.

SUGGESTED ENHANCEMENTS: Your input routines.

RESULT

Number of loan payments calculated.

SAMPLE VALUE: 16 whole payments of $100.00 and one payment of $33.30

REMAINING BALANCE

WHAT IT DOES: Calculates the balance remaining on a loan after a specified number of payments.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                     *
120 REM * REMAINING BALANCE *
130 REM *                     *
140 REM **********************
150 REM -----------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    RATE:      INTEREST RATE
190 REM    LOAN:      AMOUNT OF LOAN
200 REM    NUMBER:    LAST PAYMENT MADE
210 REM    PERIODS:   PAYMENTS PER YEAR
```

```
220 REM     PAYMENT:  MONTHLY PAYMENT
230 REM   RESULT --
240 REM     BALANCE:  CURRENT BALANCE OWED
250 REM
260 REM ------------------------------------

270 REM *** INITIALIZE ***

280 LOAN=5000
290 RATE=12.5
300 NUMBER=46
310 PERIODS=12
320 BALANCE=LOAN
330 PAYMENT=112.49
340 GOSUB 10310
350 PRINT "BALANCE ON LOAN AFTER ";NUMBER;" PAYMENTS:"
360 PRINT USING CASH$;BALANCE
370 END

10300 REM *** SUBROUTINE ***

10310 RATE=RATE/100
10320 CASH$="$####.##"
10330 FOR N=1 TO NUMBER
10340 S1=INT((BALANCE*RATE/PERIODS)*100+.5)/100
10350 S2=PAYMENT-S1
10360 BALANCE=BALANCE-S2
10370 NEXT N
10380 RETURN
```

HOW TO USE SUBROUTINE

Many loans are paid off before the end of the calculated amortization period. That is, you may trade in your car after three years even though the automobile loan was for 48 months. As a nation, we change residences approximately once each seven years per family; mortgage loans commonly extend from 15 to 30 years. When a loan is paid off early, we don't simply multiply the number of payments remaining—such a figure would include unearned interest that is not owed. Instead, we need to figure the remaining balance. When some interest is paid in advance, financial institutions use a special rule to determine how much is to be refunded. This subroutine assumes you pay interest and a portion of the principal as it is due.

LINE-BY-LINE DESCRIPTION

Lines 280–330: Define LOAN value, interest RATE in whole percent (i.e., 12.5 equals 12.5 percent per annum), PERIODS as the number of payments made each year, and monthly PAYMENT. NUMBER is the payment number of the last payment to be made. If you do not know the PAYMENT amount but do know all the other factors (plus the original term of the loan), you can figure PAYMENT, using the PAYMENT AMOUNT subroutine, and plug the value in here.

Lines 340–370: Access the subroutine and print results.

Line 10310: Change interest RATE to decimal fraction.

Line 10320: Set up CASH$ as PRINT USING format.

Line 10330: Start FOR-NEXT loop from 1 to number of payments already made.

Line 10340: Calculate interest due each individual payment period.

Line 10350: Subtract interest due from payment to determine amount applied to principal.

Line 10360: Reduce balance by amount applied to principal, resulting in new remaining balance.

Line 10370: Repeat until last payment reached.

YOU SUPPLY

You must define variables LOAN, RATE, NUMBER, and PERIODS.

SUGGESTED ENHANCEMENTS: Input routines.

RESULT

BALANCE, the amount owed after the payments specified, is calculated.

SAMPLE VALUE: $1458.36

YEARS TO REACH DESIRED VALUE

WHAT IT DOES: Calculates number of years required to reach desired amount, given interest rate and original amount.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                  *
120 REM * YEARS TO REACH   *
130 REM * A DESIRED VALUE  *
140 REM *                  *
150 REM ********************
160 REM ------------------------------------------------
170 REM       ++ VARIABLES ++
180 REM   SUPPLIED BY USER --
190 REM     RATE:      INTEREST RATE
200 REM     AMOUNT:    AMOUNT TO BE COMPOUNDED
210 REM     FUTURE:    FUTURE VALUE DESIRED
220 REM     PERIODS:   NUMBER OF COMPOUNDING PERIODS
230 REM   RESULT --
240 REM     YEARS:     WHOLE YEARS NEEDED TO REACH VALUE
250 REM     MNTHS:     ADDITIONAL MONTHS NEEDED
260 REM
270 REM ------------------------------------------------

280 REM *** INITIALIZE ***

290 AMOUNT=1000
300 RATE=10
310 PERIODS=365
320 FUTURE=2000
330 GOSUB 10410
340 PRINT USING CASH$;AMOUNT;:PRINT " WILL COMPOUND TO"
350 PRINT USING CASH$;FUTURE;:PRINT " IN ";YEARS;" YEARS
AND";MNTHS;" MONTHS"
360 PRINT "AT ";RATE*100;" PERCENT COMPOUNDED ";PERIODS;" TIMES"
370 PRINT "A YEAR."
380 END

10400 REM *** SUBROUTINE ***

10410 CASH$="$###,###.##"
10420 RATE=RATE/100
10430 YEARS=LOG(FUTURE/AMOUNT)/((LOG(1+RATE/PERIODS))*PERIODS)
10440 MNTHS=INT((YEARS-INT(YEARS))*12)
10450 YEARS=INT(YEARS)
10460 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the number of years required to reach a desired money value, given a fixed interest rate and the original investment value. The routine assumes that no additional amounts are added to the principal. That is, an original amount is deposited in a bank and left there to accumulate for a number of years. An inheritance might be placed in the bank and allowed to build until retirement, college, or some other need for the money arises.

LINE-BY-LINE DESCRIPTION

Lines 290–320: Define FUTURE, desired future value, the interest RATE in whole percent per year, and the PERIODS, the number of compounding periods per year. Your subroutine can substitute INPUT statements to allow the user to enter these figures.

Lines 330–380: Access the subroutine and print the results.

Line 10410: Set up CASH$ as PRINT USING format.

Line 10420: Change RATE to decimal figure.

Line 10430: Calculate number of years needed to produce the goal value.

Lines 10440–10450: Figure number of whole months and years.

YOU SUPPLY

You must define these variables: FUTURE (desired future value), RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent), and PERIODS (number of compounding periods per year). The subroutine will return YEARS and MNTHS, or the number of years and months that will be required to reach the desired value.

SUGGESTED ENHANCEMENTS: Write a program that will test various values for AMOUNT and/or FUTURE value and will

provide a printout comparing how increasing the initial amount can reduce the time needed to achieve the desired value, or how changing the required FUTURE amount alters the figure required as the initial AMOUNT.

RESULT

YEARS and MNTHS are calculated.

SAMPLE VALUE: 6 years, 11 months

FUTURE VALUE

WHAT IT DOES: Calculates compounded amount of investment, given original value, interest rate, and time period

LEVEL: Intermediate

```
100 REM ********************
110 REM *                    *
120 REM * FUTURE VALUE OF    *
130 REM * A SINGLE DEPOSIT   *
140 REM *                    *
150 REM ********************
160 REM ------------------------------------------
170 REM       ++ VARIABLES ++
180 REM   SUPPLIED BY USER --
190 REM     RATE:      INTEREST RATE
200 REM     AMOUNT:    AMOUNT TO BE COMPOUNDED
210 REM     PERIODS:   NUMBER OF COMPOUNDING PERIODS
220 REM     YEARS:     YEARS COMPOUNDED
230 REM   RESULT --
240 REM    FUTURE:     FUTURE VALUE OF THE DEPOSIT
250 REM
260 REM ------------------------------------------

270 REM *** INITIALIZE ***

280 AMOUNT=1000
290 RATE=10
300 PERIODS=365
310 YEARS=10
320 MNTHS=6
330 GOSUB 10510
340 PRINT USING CASH$;AMOUNT;:PRINT " WILL COMPOUND TO"
```

```
350 PRINT USING CASH$;FUTURE;:PRINT " IN ";YEARS;" YEARS
AND";MNTHS;" MONTHS"
360 PRINT "@ ";RATE*1200;" PERCENT COMPOUNDED ";PERIODS;" TIMES"
370 PRINT "A YEAR."
380 END

10500 REM *** SUBROUTINE ***

10510 CASH$="$###,###.##"
10520 RATE=RATE/1200
10530 TT=YEARS*12+MNTHS
10540 FUTURE=AMOUNT*(1+RATE/PERIODS)^(PERIODS*TT)
10550 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the compounded future value of an investment, given the interest rate, present value, and original amount.

You might use this subroutine to calculate how much your savings account will be worth if allowed to compound for a given period of time.

LINE-BY-LINE DESCRIPTION

Lines 280–320: Define original principal AMOUNT, the interest RATE in whole percent, and the number of YEARS to be compounded.

Lines 330–380: Access the subroutine and print results.

Line 10510: Set up CASH$ as PRINT USING format.

Line 10520: Change RATE to decimal value.

Line 10530: Figure total number of months

Line 10540: Calculate future value.

YOU SUPPLY

You must define these variables: AMOUNT (the original amount), RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent), and YEARS (number of years to be compounded). The subroutine will return FUTURE, or value of the compounded investment.

SUGGESTED ENHANCEMENTS: Write a program that will print out a chart showing how single deposits of differing sizes will grow at various interest rates, or for different periods of time.

RESULT

Compound interest calculated.

SAMPLE VALUE: $2857.70

REGULAR DEPOSITS

WHAT IT DOES: Calculates the value of an account to which regular deposits are made of specified amounts at a given interest rate.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                    *
120 REM * REGULAR DEPOSITS *
130 REM *                    *
140 REM ********************
150 REM ---------------------------------------
160 REM          ++ VARIABLES ++
170 REM
180 REM  SUPPLIED BY USER --
190 REM    PAYMENT:     REGULAR DEPOSIT AMOUNT
200 REM    RATE:        INTEREST RATE
210 REM    PERIODS:     NUMBER OF DEPOSITS MADE
220 REM                 EACH YEAR
230 REM    YEARS:       NUMBER OF WHOLE YEARS
240 REM    MNTHS:       NUMBER OF WHOLE MONTHS
250 REM  RESULT --
260 REM    AMOUNT:      AMOUNT PRODUCED
270 REM
280 REM ---------------------------------------

290 REM *** INITIALIZE ***

300 PAYMENT=10
310 RATE=10
320 NUMBER=12
330 YEARS=3
340 MNTHS=6
```

```
350 GOSUB 10610
360 PRINT "DEPOSITING ";:PRINT USING CASH$;PAYMENT;
370 PRINT NUMBER;" TIMES PER YEAR"
380 PRINT "FOR ";YEARS;" YEARS WILL YIELD ";
390 PRINT USING CASH$;AMOUNT
400 END

10600 REM *** SUBROUTINE ***

10610 CASH$="$####.##"
10620 YEARS=(MNTHS/12)+YEARS
10630 RATE=(RATE/NUMBER)/100
10640 AMOUNT=PAYMENT*((1+RATE)^(NUMBER*YEARS)-1)/RATE
10650 RETURN
```

HOW TO USE SUBROUTINE

Christmas Clubs may well be a thing of the past in many parts of the country, but some of us still manage to make regular bank deposits of a fixed amount. The goal may be to build up an annuity for college or retirement (specifically an IRA, perhaps). While few may have the discipline to keep up the regular payments unless we are forced to do so by a fixed insurance annuity plan, it is useful to calculate the results of such a program. You furnish the amount deposited, interest rate, number of deposits per year, and time period. This subroutine does the rest.

LINE-BY-LINE DESCRIPTION

Lines 300–340: Define PAYMENT, RATE, NUMBER, YEARS, and MNTHS.

Lines 350–400: Access the subroutine and display results.

Line 10610: Set up CASH$ as PRINT USING format.

Line 10620: Combine YEARS and MNTHS to figure total number of YEARS of the fund.

Line 10630: Change RATE to decimal value.

Line 10640: Calculate the amount produced.

YOU SUPPLY

You must define variables PAYMENT, RATE, PERIODS, YEARS, and MNTHS.

SUGGESTED ENHANCEMENTS: Write a program to show results of increasing the amount of deposits, or of varying the interest rate.

RESULT

AMOUNT produced from the deposits is calculated.

SAMPLE VALUE: $500.41

DEPOSIT AMOUNT

WHAT IT DOES: Calculates the amount that must be deposited each period to produce a desired amount.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * DEPOSIT AMOUNT *
130 REM *                  *
140 REM ******************
150 REM -------------------------------------
160 REM          ++ VARIABLES ++
170 REM
180 REM  SUPPLIED BY USER --
190 REM    AMOUNT :      AMOUNT DESIRED
200 REM    RATE:         INTEREST RATE
210 REM    PERIODS:      NUMBER OF DEPOSITS MADE
220 REM                  EACH YEAR
230 REM    YEARS:        NUMBER OF WHOLE YEARS
240 REM    MNTHS:        NUMBER OF WHOLE MONTHS
250 REM  RESULT --
260 REM    PAYMENT:      AMOUNT TO BE DEPOSITED
270 REM
280 REM -------------------------------------

290 REM *** INITIALIZE ***

300 AMOUNT=400
310 RATE=10
320 NUMBER=12
330 YEARS=3
340 MNTHS=6
350 GOSUB 10710
360 PRINT "DEPOSITING ";:PRINT USING CASH$;PAYMENT;
370 PRINT NUMBER;" TIMES PER YEAR"
380 PRINT "FOR" ;YEARS;" YEARS WILL YIELD ";
```

```
390 PRINT USING CASH$;AMOUNT
400 END

10700 REM *** SUBROUTINE ***

10710 CASH$="$####.##"
10720 YEARS=(MNTHS/12)+YEARS
10730 RATE=(RATE/NUMBER)/100
10740 PAYMENT=AMOUNT*RATE/((RATE+1)^(NUMBER*YEARS)-1)
10750 RETURN
```

HOW TO USE SUBROUTINE

This one is a variation on the last. Here you supply the amount desired at the end of the number of years and months specified. The subroutine will tell you how much the regular monthly deposits must be in order to reach the goal.

LINE-BY-LINE DESCRIPTION

Lines 300–340: Define AMOUNT, RATE, NUMBER, YEARS, and MNTHS.

Lines 350–400: Access the subroutine and print the results.

Line 10710: Define CASH$ as PRINT USING format.

Line 10720: Combine YEARS and MNTHS to produce total YEARS.

Line 10730: Change RATE to decimal fraction.

Line 10740: Calculate the regular payment.

YOU SUPPLY

Your program should define AMOUNT, RATE, NUMBER, YEARS, and MNTHS before calling this subroutine.

SUGGESTED ENHANCEMENTS: Write a program to show the result of increasing or decreasing the amount desired.

RESULT

Regular payment needed to reach a given amount is calculated.

SAMPLE VALUE: $7.99

ANNUITY WITHDRAW

WHAT IT DOES: Calculates amount of money that can be withdrawn each period from a fund deposited at a given interest rate.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                  *
120 REM * ANNUITY WITHDRAW *
130 REM *                  *
140 REM ********************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM
180 REM  SUPPLIED BY USER --
190 REM    AMOUNT:  ORIGINAL INVESTMENT
200 REM    RATE:    INTEREST RATE
210 REM    PERIODS: NUMBER OF WITHDRAWALS
220 REM             MADE EACH YEAR
230 REM    YEARS:   NUMBER OF WHOLE YEARS
240 REM    MNTHS:   NUMBER OF WHOLE MONTHS
250 REM  RESULT --
260 REM    PAYMENT: AMOUNT TO BE WITHDRAWN
270 REM
280 REM --------------------------------

290 REM *** INITIALIZE ***

300 AMOUNT=1000
310 RATE=10
320 NUMBER=12
330 YEARS=3
340 MNTHS=6
350 GOSUB 10810
360 PRINT "WITHDRAWING  ";:PRINT USING CASH$;PAYMENT;
370 PRINT NUMBER;" TIMES PER YEAR"
380 PRINT "FOR ";YEARS WILL DEPLETE ";
390 PRINT USING CASH$;AMOUNT
400 END

10800 REM *** SUBROUTINE ***
```

```
10810 CASH$="$####.##"
10820 YEARS=(MNTHS/12)+YEARS
10830 RATE=(RATE/NUMBER)/100
10840 PAYMENT=AMOUNT*(RATE/((1+RATE)^(NUMBER*YEARS)-1)+RATE)
10850 RETURN
```

HOW TO USE SUBROUTINE

Under current law, when a retiree reaches age 70½, he or she must begin making withdrawals from an IRA. Most people choose to begin drawing on their retirement nest eggs sooner than that. Or perhaps a rich uncle has died and left you a small fortune. Maybe you received your lottery winnings in a lump sum. In any of these cases, while the money may still be earning interest in the bank, making regular withdrawals of amounts larger than that being earned in interest will eventually deplete the fund.

This subroutine calculates how much money can be taken out each period if you expect the fund, or annuity, to last for a given amount of time. Say you expect to need income from the fund for 20 years following retirement. Plugging in the proper figures in this module will show how much monthly income you can rely on.

Or, suppose the account is a college fund that must last four years. The subroutine will show the prospective student how much can be withdrawn for tuition, room and board, books, and spending money.

LINE-BY-LINE DESCRIPTION

Lines 300–340: Define the initial AMOUNT of the deposit, the interest RATE being paid during the withdrawal period, the NUMBER of withdrawals per year, as well as the YEARS and MNTHS the fund is expected to last.

Lines 350–400: Access the subroutine and show results.

Line 10810: Define CASH$ as PRINT USING format.

Line 10820: Combine YEARS and MNTHS to one value, YEARS.

Line 10830: Change RATE to decimal fraction, or percentage.

Line 10840: Calculate amount withdrawn each period.

YOU SUPPLY

You must define variables AMOUNT, RATE, PERIODS, YEARS, and MNTHS.

SUGGESTED ENHANCEMENTS: Again, write a program that will demonstrate various "what-if" possibilities. All the recommendations for enhancements so far demonstrate the power of another decision-support tool—the electronic spreadsheet. Advanced users can experiment with writing their own BASIC programs that produce data files that can be imported into spreadsheets. It's not easy—you must thoroughly understand the file structure of the spreadsheet you are using—but it can be done.

RESULT

Amount that can be withdrawn over a given period to deplete a fund is calculated.

SAMPLE VALUE: $28.32

RATE OF RETURN

WHAT IT DOES: Calculates interest rate, given present and future value and the number of compounding periods.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                  *
120 REM * RATE OF RETURN   *
130 REM *                  *
140 REM ********************
150 REM ----------------------------------------
160 REM          ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    AMOUNT:      AMOUNT TO BE COMPOUNDED
190 REM    PERIODS:     NUMBER OF COMPOUNDING PERIODS
200 REM    YEARS:       YEARS COMPOUNDED
210 REM    FUTURE:      FUTURE VALUE OF INVESTMENT
```

```
220 REM  RESULT --
230 REM   RATE:         RATE OF RETURN
240 REM
250 REM ----------------------------------------

260 REM *** INITIALIZE ***
270 AMOUNT=1000   -
280 YEARS=10
290 MNTHS=6
300 PERIODS=365
310 FUTURE=2000
320 GOSUB 10910
330 PRINT USING CASH$;AMOUNT;:PRINT " WILL COMPOUND TO"
340 PRINT USING CASH$;FUTURE;:PRINT " IN ";YEARS;" YEARS
AND";MNTHS;" MONTHS"
350 PRINT "AT ;RATE;" PERCENT COMPOUNDED ";PERIODS;" TIMES"
360 PRINT "A YEAR."
370 END

10900 REM *** SUBROUTINE ***

10910 CASH$="$###,###.##"
10920 RATE=((FUTURE/AMOUNT)^(1/(PERIODS*((YEARS*12)
+MTHS)))-1)*PERIODS
10930 RATE=INT(RATE*120000)/100
10940 RETURN
```

HOW TO USE SUBROUTINE

This routine will calculate the interest rate on an investment, given the present value, future value, years compounded, and number of compounding periods. You could use this to figure what sort of a return your investments are providing you, as a means of deciding whether to continue or look for new investments.

LINE-BY-LINE DESCRIPTION

Lines 270–310: Define the present (or original) value of the investment, the number of YEARS it has or will be compounded, and the FUTURE (or current, if the investment is an old one) value. Your subroutine can substitute INPUT lines to have the user enter these values.

Lines 320–370: Access the subroutine and print results.

Line 10910: Define CASH$ as PRINT USING format.

Line 10920: Figure interest RATE.
Line 10930: Change RATE to whole percent.

YOU SUPPLY

You must define these variables: AMOUNT (present value), FUTURE (future value), YEARS (number of years to be compounded), and PERIODS (number of compounding periods).

SUGGESTED ENHANCEMENTS: Write a program to let you compare the rate of return on several different investments.

RESULT

Interest rate is calculated.

SAMPLE VALUE: 6.93 percent

DEPRECIATION RATE

WHAT IT DOES: Calculates simple straight-line depreciation.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                     *
120 REM * DEPRECIATION RATE   *
130 REM *                     *
140 REM **********************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM
180 REM  SUPPLIED BY USER --
190 REM    PAID:    ORIGINAL PRICE PAID
200 REM    RESALE:  RESALE VALUE
210 REM    YEARS:   NUMBER OF WHOLE YEARS
220 REM    MNTHS:   NUMBER OF WHOLE MONTHS
230 REM  RESULT --
240 REM    DEPRECIATE: RATE OF DEPRECIATION
250 REM
260 REM --------------------------------
```

```
270 REM *** INITIALIZE ***

280 PAID=1000
290 RESALE=500
300 YEARS=3
310 MNTHS=0
320 GOSUB 11010
330 PRINT "DEPRECIATION RATE IS
340 PRINT USING ROUND$;DEPRECIATE;:PRINT" PER CENT PER YEAR"
350 PRINT "OVER ";YEARS;" YEARS."
360 END

11000 REM *** SUBROUTINE ***

11010 ROUND$="##.##"
11020 YEARS=(MNTHS*12)+YEARS
11030 DEPRECIATE=100*(1-(RESALE/PAID)^(1/YEARS))
11040 RETURN
```

HOW TO USE SUBROUTINE

As a practical matter, there are two kinds of depreciation, neither of which usually reflects what happens in real life. The first kind is the simple, straight-line depreciation calculated by this routine. You buy a car for $10,000. Three years later it is worth $7000. You figure that it has depreciated 30 percent, or 10 percent per year.

Actually, the car may have depreciated 15 percent of its value the first five minutes after you drove out of the showroom and it became a "used" car. The second year it might have depreciated an additional 7 to 10 percent of its original value, and only 5 percent the third year. However, real-world depreciation is difficult to calculate. And, after the fact, it makes little difference exactly how the depreciation took place. The car is still worth only $7000.

However, in business it *does* make a difference just what the depreciation schedule is. If you can depreciate an asset more in the early years of its use, you usually gain, because the value of money decreases over time, and a $1000 deduction this year is more valuable than a $1000 deduction next year, after inflation has taken its toll.

Accordingly, there are many rules set up to allow accelerated depreciation for businesses. These often have as little relation-

ship to the real world as straight-line depreciation, and the rules change often. Real estate, which usually increases in value, can nevertheless be depreciated. The allowable time span has changed from 30 or 40 years to 15 and back to 18 or 19 in recent years.

Therefore, this subroutine is intended for personal or casual business use only, not for tax purposes. It will tell you the approximate annual rate of depreciation (calculated on a straight-line basis) if it is supplied with the original purchase price of an asset, the current value, and the number of years and months that have elapsed.

You might use this subroutine to compare the relative advantages of different models of automobile. Scan the used-car ads in your newspaper for asking prices of late-model automobiles. Then figure the approximate depreciation amount by estimating the original price of these cars (you needn't be exact, nor could you be, because of the various options). Since different cars will cost more or less than others, checking the depreciation percentages is the only way to check how much value is lost by each car over a given period.

For example, you may discover that a certain luxury car depreciates only 2 percent per year, compared with 10 percent for another car, even though the amount of depreciation in dollars is exactly the same. In such a case the "cost" to own the luxury car is the same as that of the less expensive car, in terms of what you lose at trade-in time.

LINE-BY-LINE DESCRIPTION

Lines 280–310: Define PAID, RESALE, YEARS, and MNTHS.

Lines 320–360: Access the subroutine and print results.

Line 11010: Define ROUND$ as a PRINT USING format. Note that here PRINT USING is used to round off a number to a desired number of decimal places and not to format dollars-and-cents. The dollar sign has been left out of the ROUND$ definition for that reason, and no dollar sign will appear in the formatted output.

Line 11020: Change depreciation rate to percentage.

Line 11030: Calculate depreciation rate.

YOU SUPPLY

Your program should define the amount PAID, the RESALE price, and YEARS and MNTHS over which the decline in value occurred.

SUGGESTED ENHANCEMENTS: Your program can include error-trapping routines to make sure that the RESALE value is not higher than the amount PAID. In such cases, no depreciation but, rather, appreciation has occurred.

RESULT

Depreciation rate as a percentage is calculated.

SAMPLE VALUE: 20.63 percent

DEPRECIATION AMOUNT

WHAT IT DOES: Calculates the amount of depreciation each year over a designated period, with a given depreciation rate.

LEVEL: Intermediate

```
100 REM ************************
110 REM *                      *
120 REM * DEPRECIATION AMOUNT *
130 REM *                      *
140 REM ************************
150 REM ---------------------------------------
160 REM           ++ VARIABLES ++
170 REM
180 REM  SUPPLIED BY USER --
190 REM    PAID:         ORIGINAL PRICE PAID
200 REM    DEPRECIATE:   DEPRECIATION RATE
210 REM    YEARS:        NUMBER OF WHOLE YEARS
220 REM    MNTHS:        NUMBER OF WHOLE MONTHS
230 REM  RESULT --
240 REM    AMOUNT:       YEARLY AMOUNT DEPRECIATED
250 REM
260 REM ---------------------------------------

270 REM *** INITIALIZE ***
```

```
280 PAID=1000
290 DEPRECIATE=9
300 YEARS=28
310 GOSUB 11110
320 END

11100 REM *** SUBROUTINE ***

11110 CASH$="$##,###.##"
11120 DEPRECIATE=DEPRECIATE/100
11130 FOR N=1 TO YEARS
11140 AMOUNT=PAID*DEPRECIATE*(1-DEPRECIATE)^(N-1)
11150 PRINT "DEPRECIATION IN YEAR # ";N;" :";
11160 PRINT USING CASH$;AMOUNT
11170 IF N/20<>INT(N/20) GOTO 11200
11180 PRINT "-- HIT ANY KEY -- ":GETKEY A$
11190 SCNCLR
11200 NEXT N
11210 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will show you, year by year, how much an asset loses in value if you know the average annual depreciation rate. Again, the figures will not be exact, because few assets depreciate at an exact rate per year.

However, if you know that your $2 million yacht will be worthless in eight years, this subroutine will show you just how much is lost each year at that 12.5 percent annual depreciation rate.

LINE-BY-LINE DESCRIPTION

Lines 280–300: Define amount PAID, the depreciation rate, and number of YEARS and MNTHS to be calculated. If you try to figure depreciation over longer than the depreciable life of an asset, negative values will be produced.

Lines 310–320: Access the subroutine.

Line 11110: Define CASH$ as PRINT USING format. Notice the inclusion of commas to format the dollars-and-cents, since we expect to be working with larger amounts of money in this subroutine.

Line 11120: Change depreciation rate to percentage.

Line 11130: Start FOR-NEXT loop from 1 to number of YEARS.

Line 11140: Calculate amount of depreciation for year N.

Lines 11150–11160: Print result for year N.

Line 11170: Check loop counter to see if it is evenly divisible by 20. This is done by comparing the value of the loop counter divided by 20 (N/20) with the integer value of the same (INT(N/20). Only when there is no remainder will they be the same, that is, when N = 20 or 40 or 60 or 80. At this point, the routine drops down to the following lines. Otherwise, it loops back for the next year.

Lines 11180–11190: When 20 screen lines have been filled up, the routine waits for the user to press a key and then clears the screen and prints the next set. This procedure keeps information from scrolling off the screen faster than the user can read it.

Line 11200: Loop.

YOU SUPPLY

You must define variables PAID, DEPRECIATION, YEARS, and MNTHS.

SUGGESTED ENHANCEMENTS: Write a routine that will direct the output of this module to your printer, or adapt the Write to Disk routine at the end of this chapter to save your results for later study.

RESULT

Depreciation schedule is printed out on screen.

SAMPLE VALUES: Year one: $90.00; year five: $61.72; year ten: $38.57

TEMPERATURE

WHAT IT DOES: Calculates Celsius and Fahrenheit.

LEVEL: Novice

```
100 REM ***************
110 REM *               *
120 REM * TEMPERATURE   *
130 REM *               *
140 REM ***************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM       F:  FAHRENHEIT
190 REM           OR
200 REM       C:  CELSIUS
210 REM   RESULT --
220 REM       F:  FAHRENHEIT
230 REM           OR
240 REM       C:  CELSIUS
250 REM
260 REM ----------------------------

270 REM *** INITIALIZE ***

280 GOSUB 11310
290 PRINT F;"F. = ";C;"C."
300 END


11300 REM *** SUBROUTINE ***

11310 PRINT
11320 PRINT" CONVERT :"
11330 PRINT TAB(4)"1.) FAHRENHEIT TO CELSIUS"
11340 PRINT TAB(4)"2.) CELSIUS TO FAHRENHEIT"
11350 PRINT "ENTER CHOICE :"
11360 GETKEY A$
11370 A=VAL(A$):IF A<1 OR A>2 GOTO 11360
11380 ON A GOTO 11420,11390
11390 INPUT"ENTER TEMPERATURE IN CELSIUS :";C
11400 F=INT((9/5)*C+32)
11410 RETURN
11420 INPUT"ENTER TEMPERATURE IN FAHRENHEIT :";F
11430 C=INT((F-32)*(5/9))
11440 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will convert Celsius temperatures to Fahrenheit and vice versa. The sample routine has a short INPUT section that asks for the temperatures to be entered from the keyboard.

LINE-BY-LINE DESCRIPTION

Lines 280–300: Access the subroutine and print result.
Lines 11310–11350: Present menu of options.
Lines 11360–11370: Accept valid choice only.
Line 11380: Access appropriate input/calculation routine.
Line 11390: Enter temperature in Celsius.
Line 11400: Convert to Fahrenheit.
Line 11410: Return.
Line 11420: Enter temperature in Fahrenheit.
Line 11430: Convert to Celsius.
Line 11440: Return.

YOU SUPPLY

You should respond to the prompts with keyboard entry. Your program can also bypass the input routine and use the subroutines directly.

SUGGESTED ENHANCEMENTS: Your routine could check input for "C" or "F" and automatically determine the conversion needed.

RESULT

Temperature converted to alternate value.

SAMPLE VALUES: 212F = 100C; 32F = 0C; –40F = –40C

DATE FORMATTER

WHAT IT DOES: Formats dates to MM/DD/YY style.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * DATE FORMATTER *
130 REM *                  *
140 REM ******************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM       MNTH$: MONTHS
190 REM       DAY$:  DAY
200 REM       YEAR$: YEAR
210 REM  RESULT --
220 REM       DATE$  FORMATTED DATE
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 GOSUB 11510
270 PRINT DTE$
280 END

11500 REM *** SUBROUTINE ***

11510 INPUT"ENTER MONTH";MNTH$
11520 MNTH=VAL(MNTH$)
11530 IF MNTH<1 OR MNTH>12 GOTO 11510
11540 IF MNTH<10 THEN MNTH$="0"+RIGHT$(MNTH$,1)
11550 INPUT"ENTER DAY : ";DAY$
11560 DAY=VAL(DAY$)
11570 IF DAY<1 OR DAY>31 GOTO 11550
11580 IF MNTH=4 OR MNTH=6 OR MNTH=9 OR MNTH=11 AND DAY>30 THEN
GOTO 11550
11590 INPUT"ENTER YEAR : ";YEAR$
11600 YEAR=VAL(YEAR$)
11610 IF YEAR/4<>INT(YEAR/4)GOTO 11640
11620 IF MNTH=2 AND DAY>29 GOTO 11550
11630 GOTO 11650
11640 IF MNTH=2 AND DAY>28 GOTO 11550
11650 IF DAY<10 THEN DAY$="0"+RIGHT$(DAY$,1)
11660 DTE$=MNTH$+"/"+DAY$+"/"+YEAR$
11670 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will accept input of month, day, and year and format it into MM/DD/YY style. That is, December 3, 1947, will be displayed as 03/12/47 or 03/12/1947. As written, the module prompts the operator to enter the values. It disallows illegal months (smaller than one or larger than 12). Other checks are made to make sure the day of the month is acceptable.

For example, June 31 and February 30 are not allowed. February 29 is permitted only during leap years.

Where needed, a leading 0 is added, along with backslashes to produce the desired format. This subroutine can be used in any business program where the operator is asked the date and it is important to have a uniform format.

LINE-BY-LINE DESCRIPTION

Lines 260–280: Access the subroutine and print results.

Line 11510: Enter month to be formatted.

Lines 11520–11530: Check to see that MNTH is at least 1 but no more than 12.

Line 11540: If MNTH is less than 10 then MNTH$ = "0" plus the string representation of MNTH. That is, "9" becomes "09". We take RIGHT$(MNTH$,1) in case the user has already added the 0 for us. We don't want two of them.

Lines 11550–11570: Enter day of month, which must be at least 1 and less than 31.

Line 11580: Check to see if month should have only 30 days, and force user to reenter if an illegal date has been supplied.

Line 11590–11600: Enter year.

Lines 11610–11640: If leap year, then February may have 29 days; otherwise, only 28 allowed.

Line 11650: If DAY is less than 10, then add leading "0".

Line 11660: Construct MM/DD/YY string.

YOU SUPPLY

The date to be formatted must be supplied from the keyboard.

SUGGESTED ENHANCEMENTS: As written, subroutine will not work for all days of all years. Change the leap year section so it will work for more years, at least using our current calendar. Hint: You must change the divisor to some other value. Find a way to determine whether year entered has two digits or four, and truncate to two so that 12/03/47 will not appear as 12/03/1947.

RESULT

Properly formatted date.

SAMPLE VALUE: 12, 3, 1947 changed to 12/03/1947

NUMBER OF DAYS

WHAT IT DOES: Calculates number of days between two dates in a single year.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * NUMBER OF DAYS *
130 REM *                  *
140 REM ******************
150 REM ------------------------------------
160 REM    ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     DA$:  DATE TO BE COMPARED
190 REM   RESULT --
200 REM     DF:   NUMBER OF DAYS DIFFERENCE
210 REM
220 REM ------------------------------------

230 REM *** INITIALIZE ***

240 DATA 3,0,3,2,3,2,3,3,2,3,2,3
250 DIM M(12)
260 FOR N=1 TO 12:READ M(N):NEXT N
270 GOSUB 11710
280 END

11700 REM *** SUBROUTINE ***
```

```
11710 DA=0
11720 GOSUB 11780
11730 D1=DA
11740 GOSUB 11780
11750 DF=DA-D1
11760 PRINT "DAYS DIFFERENCE: ";DF
11770 RETURN
11780 INPUT "ENTER DATE (MM/DD)";DA$
11790 IF MID$(DA$,3,1)<>"/" THEN PRINT
"USE MM/DD FORMAT!":GOTO 11780
11800 M=VAL(LEFT$(DA$,2)):D=VAL(RIGHT$(DA$,2))
11810 IF M=4 OR M=6 OR M=9 OR M=11 AND D> 30 THEN GOTO 11850
11820 IF M=2 AND D>29 THEN GOTO 11850
11830 GOSUB 11860
11840 RETURN
11850 PRINT "IMPROPER DATE!":PRINT:GOTO 11780
11860 FA=0
11870 FOR N=0 TO M-1
11880 FA=FA+M(N)
11890 NEXT N
11900 DA=28*(M-1)+FA+D
11910 RETURN
```

HOW TO USE SUBROUTINE

Sometimes in figuring penalty charges, or in prorating rents or other amounts, we need to know the number of days between two dates. This subroutine asks for input for the starting and end dates and supplies the number of days between them. It will not span years, but you can calculate the number of days from the start date to December 31, and then from January 1 to the end date, and add them together. If more than one year intervenes, add 365 (366 for leap years) for each year.

LINE-BY-LINE DESCRIPTION

Line 240: DATA line, with number of days' difference between those in each month and the minimum, 28. That is, for January, which has 31 days, the value is 3; for September, the value is 2.

Lines 250–260: Read this DATA into array M(12).

Lines 270–280: Access the subroutine.

Line 11710: Return value of DA to 0.

Line 11720: Access the routine which asks for a date and converts it into a day number.

Line 11730: Take the day number produced and store it in D1.

Line 11740: Access the routine again for the second date to be compared.

Line 11750: Calculate the difference between the two day numbers.

Line 11760: Print the results.

Line 11770: Exit the subroutine.

Line 11780: Ask user to enter date.

Line 11790: Check to see if in MM/DD format.

Line 11800: Extract value for month and day.

Line 11810: If user enters more than 30 days in illegal month, ask for reentry.

Line 11820: If user attempts to enter more than 29 days in February, refuse input.

Line 11830: Access routine to add up days.

Line 11840: Exit "day number subroutine" back to main subroutine.

Line 11850: Print IMPROPER DATE notice.

Line 11860: Initialize FA back to 0.

Line 11870: Start loop from 0 to number of month, less 1.

Line 11880: Add "extra" days to factor. These are the days more than 28 for the given month.

Line 11890: Loop.

Line 11900: Take number of months times 28 and add the extra days calculated above, producing day number.

Line 11910: Go back to previous subroutine.

YOU SUPPLY

Dates supplied from keyboard in response to prompts.

SUGGESTED ENHANCEMENTS: See HOW TO USE SUBROUTINE above for hints on changing routine to allow dates in different years. You'll need to add a year input routine as well. This

routine does not figure for leap years; you'll need to change it during leap years, or provide a year input routine to check and add an extra day for dates after February 28 during those years.

RESULT

Number of days between two dates calculated.

SAMPLE VALUE: Between January 1 and December 31: 365

DAY CONVERTER

WHAT IT DOES: Changes day number into month/day format.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * DAY CONVERTER    *
130 REM *                  *
140 REM ******************
150 REM -----------------------------------
160 REM     ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     DA:     DAY NUMBER TO BE CONVERTED
190 REM   RESULT --
200 REM     M$(M): MONTH NAME
210 REM     D:      DAY OF THE MONTH
220 REM
230 REM -----------------------------------

240 REM *** INITIALIZE ***

250 DIM M$(12)
260 FOR N=1 TO 12
270 READ M$(N)
280 NEXT N
290 DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY
300 DATA JUNE,JULY,AUGUST,SEPTEMBER
310 DATA OCTOBER,NOVEMBER,DECEMBER
320 GOSUB 12010
330 D=DA-F1
340 PRINT "DATE IS ";M$(M);" ";D
350 END

12000 REM *** SUBROUTINE ***
```

```
12010 INPUT "ENTER DAY NUMBER :";DA$
12020 DA=VAL(DA$):IF DA<1 OR DA>366 GOTO 12010
12030 GOSUB 12040:RETURN
12040 IF DA>334 THEN M=12:F1=334:RETURN
12050 IF DA>304 THEN M=11:F1=304:RETURN
12060 IF DA>273 THEN M=10:F1=273:RETURN
12070 IF DA>243 THEN M=9:F1=243:RETURN
12080 IF DA>212 THEN M=8:F1=243:RETURN
12090 IF DA>181 THEN M=7:F1=181:RETURN
12100 IF DA>151 THEN M=6:F1=151:RETURN
12110 IF DA>120 THEN M=5:F1=120:RETURN
12120 IF DA>90 THEN M=4:F1=90:RETURN
12130 IF DA>59 THEN M=3:F1=59:RETURN
12140 IF DA>31 THEN M=2:F1=31:RETURN
12150 M=1:F1=0
12160 RETURN
```

HOW TO USE SUBROUTINE

Computer programs work best with dates that are in an absolute day-of-the-year number format. However, humans relate better to month/day format. This subroutine will take any day number and convert it to a string, such as January 2 or December 3.

LINE-BY-LINE DESCRIPTION

Line 250: DIMension array to store the names of the months.

Lines 260–280: Read the names of the months into the array, M$(n).

Lines 290–310: DATA lines with month names.

Lines 320: Access the subroutine.

Line 330: Calculate day-of-month.

Line 340: Print name of month, and day-of-month.

Line 12010: Ask for day number. If your program is already working with day numbers, you can simply supply the number to the routine instead of including this line.

Line 12020: Check day number for validity.

Line 12030: Access day-number routine. This is treated as a separate subroutine so that an exit by RETURN can be effected when the desired value is determined.

Lines 12040–12140: Compare DA with various values. As soon as a value that DA is greater than is found, then the routine

"knows" that the day number falls within the corresponding month. Variable M is assigned that month number, and the day number of the end of the previous month is assigned to variable F1. In line 330, F1 is subtracted from DA, with the remainder equaling the day-of-the-month.

Line 12150: If DA is less than 31, then control passes to this line, and the month is therefore January.

YOU SUPPLY

You must enter, or your program must supply, the day-of-the-year number to this subroutine.

SUGGESTED ENHANCEMENTS: Provide a way of accounting for leap years, since all day numbers after February 29 are one higher than in non-leap years.

RESULT

Day number converted to month/day format.

SAMPLE VALUE: Day number 33 converted to February 2

MENU

WHAT IT DOES: Serves as menu template for user programs.

LEVEL: Novice

```
100 REM ********
110 REM *       *
120 REM * MENU *
130 REM *       *
140 REM ********
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM       NC:  NUMBER OF MENU CHOICES
190 REM  RESULT --
200 REM       MENU CHOICES
```

```
210 REM
220 REM ----------------------------

230 REM *** INITIALIZE ***

240 NC=4
250 GOSUB 12210
260 END
270 REM -- INSERT FIRST ROUTINE HERE
280 RETURN
290 REM -- INSERT SECOND ROUTINE HERE
300 RETURN
310 REM -- INSERT THIRD ROUTINE HERE
320 RETURN
330 REM -- INSERT FOURTH ROUTINE HERE
340 RETURN

12200 REM *** SUBROUTINE ***

12210 SCNCLR
12220 PRINT TAB(6)"** MENU **"
12230 PRINT CHR$(17);CHR$(17)
12240 PRINT TAB(3)"1. FIRST CHOICE"
12250 PRINT TAB(3)"2. SECOND CHOICE"
12260 PRINT TAB(3)"3. THIRD CHOICE"
12270 PRINT TAB(3)"4. FOURTH CHOICE"
12280 PRINT CHR$(17)
12290 PRINT TAB(6)"ENTER CHOICE :"
12300 GETKEY A$
12310 A=VAL(A$)
12320 IF A<1 OR A>NC GOTO 12300
12330 ON A GOSUB 270,290,310,330
12340 RETURN
```

HOW TO USE SUBROUTINE

Most programs with more than one function feature a menu of choices for the user to select from. This subroutine is a menu "template" that can be fleshed out with choices of your own selection and routines that fulfill each menu item.

If you define the number of selections on the menu at the beginning of your program, the menu will automatically reject illegal choices, that is, those that are out of the allowed range. User input for up to nine selections is accomplished by pressing a single key.

Once the operator has selected a menu item, the routine branches to modules written by the user to carry out the menu functions. To expand the number of menu items, redefine NC. If

more than nine choices are listed, you will have to sacrifice single-key entry. Replace line 240 with INPUT A$. Then any number can be entered.

Note that no menu functions are provided at lines 270, 290, 310, or 330; you must write those routines yourself.

LINE-BY-LINE DESCRIPTION

Line 240: Define number of menu choices available.

Lines 12210–12220: Clear screen and present menu title. Line 12220 may be changed by user to label-specific menu.

Line 12230: Move cursor down two lines.

Lines 12240–12270: Labels for the menu choices.

Line 12280: Move cursor down one more line.

Line 12290: Prompt user choice.

Line 12300: Wait for user input.

Lines 12310–12320: If entry is less than 1 or larger than the number of choices available, go back and continue waiting.

Line 12330: Access subroutine specified by user, at Lines 270, 290, 310, or 330.

YOU SUPPLY

You should define NC to equal the number of menu choices. You will need to write subroutines to accomplish your various tasks, using line 12330 as a model to direct control.

SUGGESTED ENHANCEMENTS: Fancier input routine; sound.

RESULT

Operator can select from list of menu choices.

SAMPLE VALUE: None

TIME ADDER

WHAT IT DOES: Totals seconds, minutes, and hours.

LEVEL: Intermediate

```
100 REM **************
110 REM *            *
120 REM * TIME ADDER *
130 REM *            *
140 REM **************
150 REM ------------------------------
160 REM      ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM       TM:   TOTAL MINUTES SO FAR
190 REM       TS:   TOTAL SECONDS SO FAR
200 REM       TH:   TOTAL HOURS SO FAR
210 REM       MIN:  MINUTES TO BE ADDED
220 REM       HOUR: HOURS TO BE ADDED
230 REM       SECS: SECONDS TO BE ADDED
240 REM  RESULTS --
250 REM      TM:    NEW TOTAL MINUTES
260 REM      TS:    NEW TOTAL SECONDS
270 REM      TH:    NEW TOTAL HOURS
280 REM
290 REM ------------------------------

300 REM *** INITIALIZE ***

310 TM=54
320 TH=40
330 TS=30
340 MIN=30
350 HOUR=2
360 SECS=30
370 GOSUB 12410
380 PRINT "SECONDS :";TS
390 PRINT "MINUTES :";TM
400 PRINT "HOURS :";TH
410 END

12400 REM *** SUBROUTINE ***

12410 TM=(TM+MIN)*60
12420 TS=TS+SECS
12430 TH=(TH+HOUR)*3600
12440 TS=TM+TS+TH
12450 TH=INT(TS/3600)
12460 TS=TS-TH*3600
12470 TM=INT(TS/60)
12480 TS=TS-TM*60
12490 RETURN
```

HOW TO USE SUBROUTINE

Various programs, such as timers, must add minutes, seconds, and hours and come up with a total, despite the clumsy base-60/base-24 numbering system combination.

This subroutine takes the total seconds, minutes, and hours at any time and adds in user-supplied figures, producing a new set of totals.

LINE-BY-LINE DESCRIPTION

Lines 310–330: Define the current total minutes, hours, and seconds.

Lines 340–360: Define the number of hours, minutes, and seconds to be added to the above variables.

Lines 370–410: Access the subroutine and print the results.

Line 12410: Add current total minutes to the minutes to be added, then multiply by 60 to produce the total number of seconds in those minutes.

Line 12420: Add the current total seconds to the seconds to be added.

Line 12430: Add the current total hours to the hours to be added, and then multiply by 3600 to determine the number of seconds in those hours.

Line 12440: Calculate the overall total number of seconds by adding subtotals of seconds, minutes, and hours.

Line 12450: Take the integer portion of this total divided by 3600 to determine the number of whole hours.

Line 12460: Subtract the hours' worth of seconds from the subtotal.

Line 12470; Take the integer portion of the new subtotal divided by 60 to determine the number of whole minutes.

Line 12480: Subtract the minutes' worth of seconds from the subtotal; the remainder is the seconds.

YOU SUPPLY

You must supply start-up values for TS, TM, and TH, or else they will default to those shown in lines 310 to 330. You may change these defaults to zeros if you wish. Your program should furnish MIN, HOUR, and SECS values.

SUGGESTED ENHANCEMENTS: None.

RESULT

New total time calculated.

SAMPLE VALUE: 43 hours, 25 minutes, 0 seconds

MPG

WHAT IT DOES: Calculates auto miles per gallon.

LEVEL: Novice

```
100 REM *******
110 REM *     *
120 REM * MPG *
130 REM *     *
140 REM *******
150 REM ----------------------------
160 REM      ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM      BEGN:   STARTING ODOMETER
190 REM      ODOM:   CURRENT ODOMETER
200 REM      GALLNS: GALLONS GAS USED
210 REM  RESULT --
220 REM      MPG:    MILES PER GALLON
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 ODOM=36420
270 BEGN=36001
280 GALLNS=13.8
290 GOSUB 12510
300 PRINT
```

```
310 PRINT "MPG=";MPG
320 END

12500 REM *** SUBROUTINE ***

12510 MPG=(ODOM-BEGN)/GALLNS
12520 MPG=INT(MPG*10)/10
12530 RETURN
```

HOW TO USE SUBROUTINE

This routine will figure your gas consumption, given the starting and ending odometer readings and number of gallons of gas consumed. To be accurate, you should top off your gas tank before writing down BEGN value, and top it off again when recording ODOM. Any gas put in between those two should be added to the final fillup. In other words, the MPG can be figured for the aggregate of a number of tankfuls of gas.

LINE-BY-LINE DESCRIPTION

Lines 260–280: Define current ODOMeter reading, the BEGN, or initial odometer reading, and the number of gallons, GALLNS of gas used. Your subroutine can use INPUT statements to allow the user to enter these values.

Lines 290–320: Access subroutine and print results.

Line 12510: Calculate MPG.

Line 12520: Round off MPG.

YOU SUPPLY

You need to enter values for BEGN, ODOM, and GALLNS, as outlined above. Variable MPG will store final miles per gallon figure.

SUGGESTED ENHANCEMENTS: Write routines that will store each odometer reading and gallons used and will keep a running log of MPG's for one or a group of automobiles used for business or personal use.

RESULT

MPG calculated.

SAMPLE VALUE: 30.3

ABBREVIATIONS

WHAT IT DOES: Takes user input of state name, and returns two-character post office abbreviation.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * ABBREVIATIONS *
130 REM *                 *
140 REM *****************
150 REM ----------------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     STATE$:        STATE NAME TO ABBREVIATE
190 REM  RESULT --
200 REM       STATE$(N,1): ABBREVIATION OF STATE
210 REM
220 REM ----------------------------------------

230 REM *** DATA ***

240 DATA AL,ALABAMA,AK,ALASKA,AZ,ARIZONA,AR,ARKANSAS
250 DATA CA,CALIFORNIA,CO,COLORADO,CT,CONNECTICUT
260 DATA DE,DELAWARE,DC,DISTRICT OF COLUMBIA,FL,FLORIDA
270 DATA GA,GEORGIA,HI,HAWAII,ID,IDAHO,IL,ILLINOIS
280 DATA IN,INDIANA,IA,IOWA,KS,KANSAS,KY,KENTUCKY
290 DATA LA,LOUISIANA,ME,MAINE,MD,MARYLAND,MA,MASSACHUSETTS
300 DATA MI,MICHIGAN,MN,MINNESOTA,MS,MISSISSIPPI
310 DATA MO,MISSOURI,MT,MONTANA,NE,NEBRASKA,NV,NEVADA
320 DATA NH,NEW HAMPSHIRE,NJ,NEW JERSEY,NM,NEW MEXICO
330 DATA NY,NEW YORK,NC,NORTH CAROLINA,ND,NORTH DAKOTA
340 DATA OH,OHIO,OK,OKLAHOMA,OR,OREGON,PA,PENNSYLVANIA
350 DATA RI,RHODE ISLAND,SC,SOUTH CAROLINA,SD,SOUTH DAKOTA
360 DATA TN,TENNESSEE,TX,TEXAS,UT,UTAH,VT,VERMONT,VA,VIRGINIA
370 DATA WA,WASHINGTON,WV,WEST VIRGINIA,WI,WISCONSIN,WY,WYOMING

380 REM *** INITIALIZE ***

390 DIM STATE$(51,2)
400 FOR N=1 TO 51
410 FOR N1=1 TO 2
```

```
420 READ STATE$(N,N1)
430 NEXT N1
440 NEXT N
450 INPUT"ENTER STATE NAME :";STATE$
460 GOSUB 12610
470 IF N=51 AND STATE$<>"WYOMING" GOTO 490
480 PRINT "ABBREVIATION FOR ";STATE$;" IS ";STATE$(N,1)
490 END

12600 REM *** SUBROUTINE ***

12610 TEMP$=""
12620 FOR N=1 TO LEN(STATE$)
12630 B$=MID$(STATE$,N,1)
12640 B=ASC(B$)
12650 IF B>192 AND B<219 THEN B=B-128
12660 TEMP$=TEMP$+CHR$(B)
12670 NEXT N
12680 STATE$=TEMP$
12690 IF LEFT$(STATE$,2)="S." THEN STATE$="SOUTH"+MID$(STATE$,3)
12700 IF LEFT$(STATE$,2)="N." THEN STATE$="NORTH"+MID$(STATE$,3)
12710 IF LEFT$(STATE$,2)="W." THEN STATE$="WEST VIRGINIA"
12720 FOR N=1 TO 51
12730 IF STATE$(N,2)=STATE$ THEN RETURN
12740 NEXT N
12750 PRINT "YOU SPELLED THE STATE WRONG!"
12760 RETURN
```

HOW TO USE SUBROUTINE

Many times business programs have features that make them easier to use. These features include allowing a variety of user inputs to specific questions. Instead of requiring a user to memorize the correct abbreviations for the states, a program could allow entering the state name and supply the abbreviations automatically.

With this subroutine, users can enter the whole state name, and the routine will supply proper postal abbreviations for any of the 50 states and the District of Columbia. It will also accept N. Dakota as well as North Dakota.

LINE-BY-LINE DESCRIPTION

Lines 240–370: DATA lines with the names and abbreviations for the states and Washington, D.C.

Line 390: DIMension an array to store the state names in one column and the corresponding abbreviation in the second column.

Lines 400–440: Read the state names into the two-dimensional array.

Line 450: Request state name.

Line 460: Access the subroutine.

Line 470: If no match was found, skip next line.

Line 480: Display result.

Line 12610: Null the temporary string, TEMP$.

Line 12620: Begin loop from 1 to the length of the name of the state typed in.

Line 12630: Store in variable B$ the single character string taken from the middle of STATE$ at position N.

Line 12640: Find ASCII value of that string.

Line 12650: If that value is higher than 192 and less than 219, then the character is lowercase. Change to uppercase by subtracting 128. All other characters left unchanged.

Line 12660: Add CHR$(B) to TEMP$.

Line 12670: Loop through all characters in STATE$, changing any lowercase to uppercase.

Line 12680: Change STATE$ to equal TEMP$.

Line 12690–12710: Look for abbreviations of North, South, and West. Change to full word if found.

Lines 12720–12740: Check STATE$ against list of state names in array. If match found, exit subroutine. N will mark place in array where both state name and abbreviation are stored.

Line 12750: If no match found, inform user that state is spelled wrong, and return.

YOU SUPPLY

Your program should call this routine whenever a state name is entered and the abbreviation is needed.

SUGGESTED ENHANCEMENTS: Build a more sophisticated error trap for when the state name is spelled wrong. Subroutine could look at only the first four or five characters of the state name, to allow for misspellings. You would have to allow for states that start with the same characters, such as North Dakota and North

Carolina. As it is written, when no match is found, routine returns to main program with a value for N equal to 51. In other words, a mismatch provides an abbreviation of WY (Wyoming).

RESULT

State name changed to abbreviation.

SAMPLE VALUES: Ohio, OH; N. Carolina, NC; South Dakota, SD

SEQUENTIAL DATA FILES

A *file* is any collection of information that is stored on disk or tape. Computer software is a type of file called a *program file.* On your disk, these files are marked with the PRG designation in the directory. Such files can be loaded by the Commodore 128 and can provide the BASIC interpreter with instructions that can be used to perform a task.

Raw information can also be stored as a file, even though the computer cannot load it and act on it directly. These *data files* must usually be loaded into memory through another program or subroutine which contains the actual instructions for accessing the information. These files are in either serial, or sequential, form (and marked SEQ in the disk directory) or in random access format (marked REL). The latter, relative files, are beyond the scope of this book. For simple BASIC programs, sequential data files are adequate.

The Commodore 128 cassette recorder is a good analog to sequential files. A program is a sequential file, stored one byte at a time, on your program tape or disk in the same order in which it is LISTed. In the case of cassette tapes, the program is continuous on one long piece of tape. On disks, programs are also written or read serially, but the actual sectors in which the information is stored are not always consecutive. If there is not enough room on a single track, the disk drive will often continue on a different

track, called an *extent*, leaving behind a pointer to tell itself where to pick up the next piece of the program.

In either case, however, program files are written or read only from the very first byte to the last, in sequential order, regardless of the physical order of the media. Sequential data files operate exactly the same way.

The disadvantage of sequential files is that, since they must always be read from beginning to end, there is no way to access the information in the middle without reading everything that has come first. Although you may APPEND new information to the end of a sequential file, to make a change in the middle it is necessary to read in the entire file, make the change in the middle, and then write the whole file back out to disk.

Data files are one of the basic tools of business and personal programming, as they let you keep permanent records that can be accessed, printed out, manipulated, and otherwise used in a practical manner. Data files are akin to programs in that, lacking some mass storage for the data (or program), you would have to type the information in every time you turned on the computer. In many ways, however, a computer program is a more complicated file. Programs have line numbers and links that tell the computer where the next line number is. Data files consist of just an ASCII representation of the information as it was written to the disk or tape; they are words, numbers, and punctuation, and almost nothing more.

To read a given data file, you first OPEN a channel for that information to be sent. Then, you INPUT# (with the # being followed by the number you have assigned to the input channel, e.g., INPUT#1) data to a variable of your choice. Writing to a disk or tape file is done by OPENing a channel for output and using the PRINT# statement to print information from a variable to the file.

To help your Commodore 128 keep its files and the devices it uses straight, each of the devices has been assigned a device number. The keyboard is device number 0, the cassette tape recorder device number 1, the screen device number 3, and so

forth. A serial printer is assigned device number 4, and the first disk drive in a system is usually assigned device number 8.

So, by simply substituting one device number for another, you can direct files to where you desire, within limits. For example, to SAVE a program to cassette, you can type:

```
SAVE"filename",1
```

If no device number is indicated, the computer assumes you mean device number 1, the default value. That is why your cassette SAVEs do not include the numeral one. To SAVE the same program to disk, device number 8, you would type:

```
SAVE"filename",8
```

Using a numeral 4 instead would send the file to the printer. Logically, you could even list a program to the screen by SAVE"filename",3, except that the Commodore 128 defines the screen, as well as certain other devices, as "illogical" when used with certain commands, such as SAVE. However, another command is available in BASIC, that of "CMD", which redirects output intended for the screen to another device. Typing CMD4: LIST will cause a program to be listed on the printer instead of the screen, assuming you have OPENed that device first. As was mentioned, it is usually necessary to open a data "channel" to send information from one device to another. This is done with the OPEN command.

If some of this information seems unfamiliar to you, the reason is that Commodore added a number of new commands to BASIC 7.0 that take care of much of this channel-handling automatically. While you *may* type SAVE "filename",8, just using the DSAVE command instead accomplishes the same thing. Understanding how the Commodore 128 sees these channels, devices, and what are called *secondary addresses* can be useful for advanced programming.

The particular channel you use is given a number of its own. Which number is assigned to the data channel is not particularly

important. However, a given channel can be used only to send information to one device at a time. To make the data channels easy to keep track of, it is often convenient to give them the same number as the device you are using. So, to open a cassette data file, you might type:

```
OPEN 1,1,1,"filename"
```

The first 1 is the number of the data channel or, as it is also called, the *logical file number*. The second 1 is the device number or the cassette tape. The third 1 is referred to as the *secondary address*, an instruction to the computer on what to do with the data. In this case, "1" means to write the data.

You could just as easily have used:

```
OPEN 2,1,1,"filename"
```

This would mean that logical file or data channel number 2 was being used with device number 1, to perform task number 1 (write), with the file name within quotes. However, here we will follow the convention of using the same logical file number as the device number. In Commodore 128 mode (but not Commodore 64 mode) you may use the DOPEN command, with its slightly different syntax, as outlined in the System Guide.

What about the secondary address number? What other options are available? For tape usage, there are two more. You may specify "0", which signifies reading a file from the tape, or "2", which will open the channel for writing to the tape but places a special "end-of-tape" marker at the end of the file. In reading that file, the Commodore 128 will progress no further until the EOT marker is removed.

OPEN just prepares the data channel for you, however. To actually read or write data, you must use PRINT# or INPUT#, with each followed by the logical file number you are using.

You may have guessed that SAVE and LOAD are modified forms of the OPEN command, which combine OPEN with PRINT or INPUT in one statement. Since that is true, the secondary

address numbers may be used with them as well. Therefore, it is possible to enter:

```
SAVE"filename",1,2
```

This writes the program to device number 1 (the cassette recorder) and places an end-of-tape marker after it. The numbers have a slightly different meaning when loading a program from the tape.

To load the file back into the memory location from which it was SAVEd, enter:

```
LOAD"filename"1,11
```

Disk files use the same format, with device number 8, the disk drive, substituting for the 1, the cassette drive. Disk drive users can also follow the file name with a file type specifier, generally "S" (for sequential) or "P" (for program). You also need to tell the disk drive in which direction the information will flow, using "W" for write and "R" for read.

Thus, a sequential disk writing OPEN statement might be:

```
OPEN 8,8,8,"0:filename,S,W"
```

The equivalent read statement would be:

```
OPEN 8,8,8,"0:filename,S,R "
```

The following sequential disk write and read programs provide both the older style OPEN statements described above, and the DOPEN equivalents. You will need to understand both if you want to master BASIC 7.0.

NOTE: The Commodore 128 also can use random access files, when updates to a file are frequent, and relative files, which are not yet widely used. Both types of files allow accessing any given record within a file, for either reading or writing, while ignoring

the rest of the file. Random files are much faster than sequential files, because only a small bit of information needs to be read into memory at a time. However, their use is much more complex, and too ambitious for this collection of subroutines.

SEQUENTIAL FILE—WRITE TO DISK

WHAT IT DOES: Writes a sequential data file to disk.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                  *
120 REM * SEQUENTIAL FILE  *
130 REM *  WRITE TO DISK   *
140 REM *                  *
150 REM ********************
160 REM -----------------------------------
170 REM       ++ VARIABLES ++
180 REM   SUPPLIED BY USER --
190 REM     DATA IN ARRAY DTA$(NI)
200 REM     A FILENAME IN LINE 12810
210 REM     NI: NUMBER OF ITEMS IN FILE
220 REM   RESULT --
230 REM     ARRAY DTA$(NI) WRITTEN TO DISK
240 REM
250 REM   NOTE: EITHER LINE 12810 OR 12820
260 REM   MAY BE USED
270 REM -----------------------------------

280 REM *** INITIALIZE ***

290 NI=10
300 DIM DTA$(NI)
310 GOSUB 12810
320 END

12800 REM *** SUBROUTINE ***

12810 REM OPEN 8,8,8,"0:FILENAME,S,W"
12820 DOPEN #8,"FILENAME,S",DO,W
12830 PRINT#8,NI
12840 FOR N=1 TO NI
12850 PRINT#8,DTA$(N)
12860 NEXT N
12870 DCLOSE #8
```

HOW TO USE SUBROUTINE

Disk users can access sequential, random access, and relative files. Sequential files are the most popular because they are easiest to understand. With disk, such files are even respectably fast. Explaining random access and relative files is a task requiring many changes and is beyond the scope of this book.

However, this subroutine provides a sample sequential file-writing routine that will take data that have been loaded into a string array, DTA$(n), and write them to disk. The same routine can be used with numeric arrays, simply by your removing the variable type specifier, "$", from DTA$(n).

Your program should also update NI each time more items are added to the array.

LINE-BY-LINE DESCRIPTION

Line 290: Set number of items in file to 10.

Line 300: DIMension DTA$ to NI elements.

Line 310: Access the subroutine.

Lines 12810–12820: OPEN the data file, given the file name in quotes. You can substitute your own file name, or a variable, like F$, and then define F$ through user INPUT.

Line 12830: Print, as the first item in the data file, the number of items in the file, NI.

Lines 12840–12860: PRINT each of the items in the array to the data file.

Line 12870: CLOSE the file.

YOU SUPPLY

Your program must furnish data for DTA$(n), either from keyboard entry or loaded from some tape or disk file. The counter NI should be redefined to reflect the number of items in the file each time an update is made. You should substitute your file name for "filename" in line 12810 or 12820.

SUGGESTED ENHANCEMENTS: You can write a program that does not need to print the number of items to the disk. Work in conjunction with the next subroutine, Read from Disk.

RESULT

Data file written to disk.

SAMPLE VALUE: None

SEQUENTIAL FILE—READ FROM DISK

WHAT IT DOES: Reads a sequential data file from disk.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                  *
120 REM * SEQUENTIAL FILE  *
130 REM * READ FROM DISK   *
140 REM *                  *
150 REM ********************
160 REM ---------------------------------------
170 REM       ++ VARIABLES ++
180 REM SUPPLIED BY USER --
190 REM     DATA FILE
200 REM     FILENAME IN LINE 12910
210 REM     AD:      NUMBER OF EMPTY SPACES TO ADD
220 REM     NI:      NUMBER OF ITEMS IN FILE
230 REM   RESULT --
240 REM    DTA$(N): ARRAY STORING FILE
250 REM    NI:      NUMBER OF ITEMS IN THE FILE
260 REM
270 REM   NOTE: EITHER LINE 12910 OR 12920
280 REM   MAY BE USED
290 REM ---------------------------------------

300 REM *** INITIALIZE ***

310 AD=10
320 GOSUB 12910
330 FOR N=1 TO NI
340 PRINT DTA$(N)
350 NEXT N
360 END
```

```
12900 REM *** SUBROUTINE ***

12910 REM OPEN 8,8,8,"0:FILENAME,S,R"
12920 DOPEN #8,"FILENAME,S",DO,R
12930 INPUT#8,NI
12940 DIM DTA$(NI+AD)
12950 FOR N=1 TO NI
12960 INPUT#8,DTA$(N)
12970 NEXT N
12980 DCLOSE #8
12990 RETURN
```

HOW TO USE SUBROUTINE

This subroutine provides a sample file reading routine that will take data that has been written to disk and load it into a string array, DTA$(n). The same routine can be used with numeric arrays, simply by your removing the variable type specifier, "$", from DTA$(n).

The routine will first read NI, the number of items in the file, from the disk. Then, the array is DIMensioned to NI + AD. This will allow AD more elements in the array for expansion during the session.

NOTE: You cannot reDIMension the array without generating an error. AD should be defined large enough to allow plenty of space for additions during any one session. Your program should also update NI to equal NI + AD before writing back to disk, if you use the Write Routine supplied with this book.

This routine willl not work unless you have previously written a file for it to read to disk, as with the previous subroutine.

LINE-BY-LINE DESCRIPTION

Line 310: Define the maximum number of items to be added in any session. If you wish, you can make this number very large to allow for lengthy input sections. Keeping it small, and then having your program prompt the user to close the file once the limit has been reached, is a way of automatically protecting your data from losses due to power outages. With sequential files, updates are stored in memory until written to disk, so a lengthy

session could be lost if the program is exited ungracefully (the comptuer crashes, or the power is lost).

Line 320: Access the subroutine.

Lines 330–360: Print the information read from disk to the screen.

Lines 12910–12920: Open the sequential file. Substitute your file name in these lines, as described in the last subroutine.

Line 12930: Read from disk the current number of items in the file. You may also simply read from the disk until the ST (status) variable indicates that the end of file marker has been reached.

As is, the program will read the number of items and then begin the following FOR-NEXT loop.

Line 12940: DIMension an array large enough for NI items, plus AD to be added.

Lines 12950–12980: Read all items in the file into the array.

SUGGESTED ENHANCEMENTS: Eliminate the need for reading NI but reduce the chance of losing data through error or power loss. Consider writing file to disk during an input session.

RESULT

Data file read from disk.

SAMPLE VALUE: None

3 BOMBPROOF DATA INPUT

Data entry may sound like a forbidding computer term, but it refers only to the process of getting new information into the computer. Your programs may ask users questions like, "WHAT IS YOUR NAME?" or "WHAT IS THE LOAN INTEREST RATE?" This information will be required by the program in order to complete its functions successfully, so naturally you will want the information to be as accurate as possible.

Here are some BASIC routines that will streamline the step of entering data and make your programming a bit easier. These are general subroutines that can be applied to many different programs. The modules are "user-interface" routines that trap

errors by permitting the operator to enter ONLY the type of input that is required by the program. If only numbers or only alpha characters are desired, that is what the routines will accept. Another routine accepts either lowercase or uppercase entries and converts the lowercase characters to upper. With Commodore computers uppercase/lowercase is an interesting concern, because the entire keyboard can be toggled between uppercase-only/graphics and uppercase/lowercase by the user's pressing the Shift key and Commodore key simultaneously. Your routines may have to take this dual mode into account in accepting data from the keyboard.

Another data entry routine is the Line Input module. This routine, simulating a function found in some other BASICs, allows the user to enter *anything* into a string, including commas and other so-called string "delimiters." Normally, INPUT will permit the user to enter alpha characters and numbers but will "choke" on commas, quotation marks, and other characters that BASIC normally uses to mark the end of a string. The Line Input routine here accepts a character at a time using GETKEY, and builds a string until RETURN is pressed.

Using these subroutines, you can explore the concept of error traps and see how avoiding improper entries can reduce the frustration of first-time users of your programs.

LINE INPUT

WHAT IT DOES: Simulates LINE INPUT function found in other BASICs. Allows entering string delimiters into strings.

LEVEL: Novice

```
100 REM **************
110 REM *              *
120 REM * LINE INPUT *
130 REM *              *
140 REM **************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
```

```
180 REM     KEYBOARD INPUT
190 REM     PROMPT$:  PROMPT CHARACTER
200 REM   RESULT --
210 REM     AN$:       INPUT RETURNED
220 REM -----------------------------

230 REM *** INITIALIZE ***

240 PROMPT$="?"
250 GOSUB 13010
260 PRINT
270 PRINT AN$
280 END

13000 REM *** SUBROUTINE ***

13010 AN$="":PRINT PROMPT$;
13020 DO WHILE A$<>CHR$(13)
13030 GETKEY A$
13040 AN$=AN$+A$
13050 PRINT A$;
13060 LOOP
13070 RETURN
```

HOW TO USE SUBROUTINE

If you ask the user of a program to enter a string and if a comma or other character such as a quotation mark is entered in the body of the string, the extra character is ignored. These characters that are ignored during string INPUT are known as *string delimiters*. However, sometimes it is desirable to allow such input, as when an entire phrase or sentence is entered. Such a routine will also avoid an error by naive users who don't know enough to avoid pressing string delimiter keys during their input.

This subroutine will repeatedly poll the keyboard using GETKEY and add any characters to AN$. Commas may be entered and input can be ended only by pressing RETURN. The keys pressed are printed to the screen, just as with normal INPUT, and a question mark prompt is displayed. To the user, the change is invisible, except that string delimiters are accepted. If you wish, your program can look for string delimiters that are *not* wanted in the input and eliminate them.

In addition, the prompt character can be changed or eliminated, an especially useful feature for entries for which a question mark is inappropriate.

LINE-BY-LINE DESCRIPTION

Line 240: Define the prompt character.
Line 250: Access the subroutine.
Line 270: Print the resulting AN$.
Line 13010: Print the prompt string.
Line 13020: Start loop while A$ does not equal CHR$(13) (RETURN).
Line 13030: Get character from keyboard.
Line 13040: Add it to AN$.
Line 13050: Print the character.
Line 13060: Loop and repeat.

YOU SUPPLY

Prompt asking for input, a definition for PROMPT$, and keyboard input.

SUGGESTED ENHANCEMENTS: Filter out characters that you do not want. See following routines for examples.

RESULT

String delimiters may be entered into string.

SAMPLE VALUE: None

NUMBER INPUT

WHAT IT DOES: Allows user to input only numbers.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * NUMBER INPUT   *
130 REM *                *
140 REM ****************
150 REM -----------------------
160 REM     ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     KEYBOARD INPUT
190 REM   RESULT --
200 REM     I: NUMBER ENTERED
210 REM
220 REM -----------------------

230 REM *** INITIALIZE ***

240 GOSUB 13110
250 PRINT "NUMBER ENTERED :";I
260 END

13100 REM *** SUBROUTINE ***

13120 I$=""
13120 GETKEY A$:IF A$=CHR$(13) GOTO 13220
13130 IF MINUSFLAG=1 GOTO 13150
13140 IF A$="-" THEN MINUSFLAG=1:GOTO 13180
13150 IF MANTISSAFLAG=1 GOTO 13170
13160 IF A$="." THEN MANTISSAFLAG=1:GOTO 13180
13170 IF A$<"0" OR A$>"9" GOTO 13120
13180 PRINT A$;
13190 I$=I$+A$:MINUSFLAG=1
13200 GOTO 13120
13210 I=VAL(I$)
13220 PRINT:MANTISSAFLAG=0:MINUSFLAG=0:I$=""
13230 RETURN
```

HOW TO USE SUBROUTINE

Well-written programs include features that trap possible errors by the user—or avoid them entirely. When numbers only are expected for INPUT, an elegantly constructed program will accept only numeric entries and reject everything else.

The most common procedures all have drawbacks. A line like "10 INPUT A" willl indeed accept only numbers. However, if a user happens to enter a string instead, only a cryptic "RE-DO FROM START" message will be displayed. That's not much help for a naive operator.

Another less-than-perfect solution is to use a line like "10 INPUT A$:A = VAL(A$):IF A<1 GOTO 10". If the user enters alpha characters, the program loops back and the input must be repeated.

This subroutine takes a different approach. It totally ignores nonnumbers; if the operator presses an illegal key, it isn't even echoed to the screen. The keyboard responds only when numeric keys are pressed.

The secret is a GETKEY loop. If the user presses a number key, that letter is added to I$. When A$ equals CHR$(13), a carriage return, then input is over. Otherwise, the loop repeats, allowing additional numeric entries.

When the subroutine ends, variable I will have the value of the user's entry. Negative numbers are accommodated by this module, but you cannot enter equations or number series such as 4+3–4.

LINE-BY-LINE DESCRIPTION

Lines 240–260: Access the subroutine and display result.

Line 13110: Wait for key to be pressed.

Line 13120: If key was RETURN, then go to end of this subroutine.

Line 13130: If MINUSFLAG=1, meaning that a minus sign has already been pressed once, jump to portion that checks for a decimal point.

Line 13140: Otherwise, if A$ is a minus sign, then set MINUSFLAG to 1, and jump to where minus will be added to the string.

Line 13150: If MANTISSAFLAG=1, meaning that a decimal point has already been pressed once, jump to portion that checks for nonnumeric entries. As in 13130, this ensures that the decimal point can be entered only *once* for each number input.

Line 13160: If decimal point entered, set MANTISSAFLAG to 1. One decimal point, and only one, may be entered anywhere within the number.

Line 13170: If character is not a numerical, ignore it and return for more input.

Line 13180: Print the character, which either will be a number or, if both flags had been zero, may be either a minus sign or decimal point.

Line 13190: Add the character to the string. Change MINUS-FLAG to 1, so that after the first character has been added to the string, a minus sign may not erroneously be placed in the middle of the string.

Line 13200: Return for more input.

Line 13210: Determine value of I, null variables, and return to the main program.

YOU SUPPLY

Keyboard input.

SUGGESTED ENHANCEMENTS: User may change the upper and lower limits in line 160 to restrict the range of numbers to be entered. This might be useful when getting input for, say, a menu with only five choices. All numbers over five and all alpha characters would be ignored. Filter out all numbers too large for the Commodore 128 (approximately 1.9E+19).

RESULT

Only user numeric input, in the form of positive numbers or negative numbers, is allowed. Large numbers entered as numerals will be displayed in scientific notation.

SAMPLE VALUES: –1222.54; 234; 129755.171711

LETTER INPUT

WHAT IT DOES: Allows user to input only alpha characters.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * LETTER INPUT *
130 REM *                *
140 REM ****************
150 REM --------------------------
160 REM    ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    KEYBOARD INPUT
190 REM  RESULT --
200 REM    I$: STRING ENTERED
210 REM    ONLY UPPER AND LOWER CASE
220 REM    CHARACTERS ACCEPTED
230 REM
240 REM  NOTE: TO EXCLUDE SPACES, TOO
250 REM        DELETE LINE 13660
260 REM
270 REM --------------------------

280 REM *** INITIALIZE ***

290 GOSUB 13310
300 PRINT
310 PRINT I$
320 END

13300 REM *** SUBROUTINE ***

13310 I$=""
13320 GETKEY A$
13330 A=ASC(A$):IF A=13 THEN RETURN
13340 IF A>64 AND A<91 GOTO 13380
13350 IF A>192 AND A<219 GOTO 13380
13360 IF A=32 OR A=160 GOTO 13380
13370 GOTO 13320
13380 PRINT A$;
13390 I$=I$+A$
13400 GOTO 13320
```

HOW TO USE SUBROUTINE

At times you will want only alpha characters to be input in a program, with all other entries, such as numbers or graphics characters, to be ignored. For example, word games might allow only the 26 letters A–Z while rejecting other keys entirely.

This subroutine does exactly that. The user may enter any alpha character. Others are ignored. If the operator presses an illegal key, it isn't even echoed to the screen. The keyboard responds only when alpha keys are pressed.

The secret is a GETKEY loop. If the user presses a letter key,

that letter is added to I$. When A$ equals CHR$(13), a carriage return, then input is over. Otherwise, the loop repeats, allowing additional alphabetic entries.

When the subroutine ends, variable I$ will have the value of the user's entry.

LINE-BY-LINE DESCRIPTION

Lines 290–320; Access the subroutine and print result.

Line 13310: Null any previous value of I$, in case subroutine has been called before during this program run.

Line 13320: Wait for user entry.

Line 13330: If key pressed was RETURN, then input is finished. It also determines ASCII value of key pressed.

Lines 13340–13390: If key was uppercase or lowercase character or a space, add to I$. Otherwise go back for more entries.

YOU SUPPLY

Keyboard input.

SUGGESTED ENHANCEMENTS: User may change the upper and lower limits to restrict the range of alpha characters that can be entered. This might be useful when getting input for, say, a game like Mastermind, where only the letters A–E are wanted. All numbers, graphics, and alpha characters larger than E can be ignored. You can also expand the allowable characters to permit entry of commas, periods, or other characters.

RESULT

Only user alpha input is allowed.

SAMPLE VALUES: This is allowed; SO IS THIS

CASE CONVERTER

WHAT IT DOES: Changes lowercase input to uppercase, and uppercase to lower.

LEVEL: Intermediate

```
100 REM ****************
110 REM *                *
120 REM * CASE CONVERT   *
130 REM *                *
140 REM ****************
150 REM --------------------------
160 REM    ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    A$:    STRING TO CONVERT
190 REM  RESULT --
200 REM    I$:    CONVERTED STRING
210 REM
220 REM  NOTE: PRESS SHIFT AND C=
230 REM        (COMMODORE) KEY FOR
240 REM        UPPER AND LOWERCASE
250 REM
260 REM --------------------------

270 REM *** INITIALIZE ***

280 A$="NOW IS THE TIME FOR ALL GOOD MEN"
290 GOSUB 13510
300 PRINT
310 PRINT I$
320 END

13500 REM *** SUBROUTINE ***

13510 I$=""
13520 FOR N=1 TO LEN(A$)
13530 A=ASC(MID$(A$,N,1))
13540 IF A>192 AND A<219 THEN A=A-128:GOTO 13560
13550 IF A>64 AND A<91 THEN A=A+128
13560 I$=I$+CHR$(A)
13570 NEXT N
13580 RETURN
```

HOW TO USE SUBROUTINE

Because the Commodore 128 does not see "YES" as equal to "yes", it is often convenient to filter user input to change all uppercase entries to lowercase, or vice versa. A word processing

program, for example, will want to allow a mixture of uppercase and lowercase text but may be confused if commands are typed that way.

Many times our error traps in programs check to see if an acceptable key has been pressed. We may want the user to enter "Y" or "N" answers only. Or our program will check a name or other entry against a list, as was done with Abbreviations in the last chapter. The Commodore 128 powers up in the uppercase/graphics mode but can be toggled to the uppercase/lowercase mode by the user's pressing the Commodore and Shift keys at the same time.

As written, this program will change uppercase entries to lowercase, and lowercase to upper. You can delete either line 13530 or line 13540 to force the routine to do one but not the other.

LINE-BY-LINE DESCRIPTION

Line 280: Define a string to be converted.

Lines 290–320: Access the subroutine and print result.

Line 13510: Start loop from 1 to the length of the string being converted.

Line 13520: Find ASCII value of a single character in the middle of the string, at position N.

Lines 13530–13540: If character is uppercase or lowercase, either add or subtract 128 to reverse it.

Line 13350: Add CHR$(A) to the string.

Line 13360: Loop until entire string is examined.

YOU SUPPLY

A string to be converted.

SUGGESTED ENHANCEMENTS: None.

RESULT

Lowercase and uppercase are reversed.

SAMPLE VALUE: now is the time for all good men

4 STRING HANDLING

When compared with the BASICs supplied with other personal computers, the BASIC 7.0 included in the Commodore 128 stacks up very well indeed. It has a powerful screen editor that allows changing program lines just by typing over them, and it has most of the features of standard Microsoft BASIC. Music and graphics capabilities, in particular, have been made much easier by the addition of special commands that carry out functions that demanded complex POKEs with the Commodore 64.

However, a few statements found in most BASICs have been omitted. Some of these are handy to have, others almost crucial for serious programming. The subroutines in this section show

you how to simulate some of these important features with your Commodore computer. Then we take you a few steps *beyond* basic BASIC with some new commands not available in any interpreter.

If your only familiarity with BASIC is through use of your Commodore 128, some of these statements and functions may appear strange to you. You may even wonder what they can be used for. A few are not commonly found in the majority of BASICs. This section will show you how to use the new functions, as well as provide tips on why you might want to put them in your own programs.

One of the key functions found in BASIC 7.0 that you'll want to familiarize yourself with is INSTR. It is used to find out whether a given string you are looking for is located within a second string of characters that is the same size or longer. INSTR returns the starting position of that target string. You might have a program that looks like this:

```
100 A$="COMMODORE 128"
110 B$="VIC-20"
120 C$="MY COMMODORE 128 IS A POWERFUL COMPUTER"
130 A=INSTR(C$,A$)
140 B=INSTR(C$,B$)
150 C=INSTR(C$,"COMMODORE",8)
160 PRINT A,B,C
```

When this program is run, the screen will display:

```
4,0,0
```

In the first instance, the value 4 is produced because INSTR has found A$, "COMMODORE" within the string C$ starting at the fourth character in C$. In the second case, a zero is returned because B$, "VIC-20", is *not* found within C$.

Line 150 shows another format for INSTR. Instead of using a variable for the string to be searched for, we've inserted a string constant, "COMMODORE", instead. We could have also used a string constant in place of C$. Neither option is usually taken because using variables is more flexible. Note the ",8" in line 150. This parameter tells INSTR to *start* its search at the eighth

position in the string, ignoring everything up to that point. Even though "COMMODORE" does appear in C$, line 150 returns a zero because the complete string is not found from position 8 onward. You would use this format when you want to search only part of a string. For example, you might have already found the target string, left it as is, and desired to look for the next occurrence.

The first two subroutines in this chapter, Replace String and Insert String, will take a string of your choice and either replace the equivalent number of characters in the target string with the second, shorter string, or insert them into the middle of the target string. You need to have an understanding of INSTR to use them most effectively, since INSTR will let you examine the target string and decide exactly where to insert or replace the shorter string.

STRING$ adds a feature to BASIC 7.0 that allows you easily to build a string of any size, using a character of your choice. Say you need a string, SPACE$, composed of 20 spaces (CHR$(32)). This subroutine will build it for you automatically.

Exchange is a handy way to exchange the values of two variables and is available in some BASICs under the name SWAP. Commodore 128 BASIC has a different SWAP command, so we call this one EXCHANGE. It is a rudimentary function included to get novice programmers thinking about the processes needed for sorts. Two sorts are included in this chapter, along with some routines to encode and decode strings and format them on the screen.

REPLACE STRING

WHAT IT DOES: Simulates MID$= function found in other BASICs. Replaces middle portion of a string with another string.

LEVEL: Novice

```
100 REM ******************
110 REM *                  *
120 REM * REPLACE STRING *
130 REM *                  *
140 REM ******************
150 REM ----------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    TARGT$:   MAIN STRING
190 REM    SUB$:     STRING TO BE REPLACED
200 REM    PLACE:    POSITION TO PUT SUB$
210 REM  RESULT --
220 REM    FINISH$:  COMPLETED STRING
230 REM
240 REM ----------------------------------

250 REM *** INITIALIZE ***

260 SUB$="TEST"
270 TARGT$="REPLACEMENT MADE"
280 PLACE=7
290 GOSUB 13610
300 PRINT FINISH$
310 END

13600 REM *** SUBROUTINE ***

13610 L$=LEFT$(TARGT$,PLACE)
13620 R$=MID$(TARGT$,PLACE+LEN(SUB$)+1)
13630 FINISH$=L$+SUB$+R$
13640 RETURN
```

HOW TO USE SUBROUTINE

Replacing the middle portion of a string with another string can be useful, especially in word processing programs. This subroutine will allow replacing any number of characters in a main string, TARGT$, with an equal number of characters, SUB$. If you want to insert a string that is larger or smaller than the one replaced, you should use the next subroutine.

This routine will replace only an equal number of characters. For example, you may take a string such as RETAIN and change it to REPAIR by making TARGT$="RETAIN", SUB$="PAIR", and PLACE=3. The subroutine will start at position 3 in the target string and substitute the SUB$.

LINE-BY-LINE DESCRIPTION

Lines 260–280: Define the SUB$, TARGT$, and PLACE.

Lines 290–310: Access the subroutine and print results.

Line 13610: Define L$ as left portion of string up to and including the character at position PLACE.

Line 13620: Define R$ as the remaining characters in the target string, from position following PLACE, *plus* the length of the string to be substituted, to the end. R$ will not include any of the characters that will be replaced by those in SUB$.

Line 13630: Define FINISH$ as L$, plus the string to be substituted, plus R$.

YOU SUPPLY

You should define TARGT$, SUB$, and PLACE just before the subroutine is called.

SUGGESTED ENHANCEMENTS: Use INSTR to determine where PLACE should be.

RESULT

TARGT$ will be changed to include SUB$.

SAMPLE VALUE: REPLACETEST MADE

INSERT STRING

WHAT IT DOES: Inserts a string into another.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * INSERT STRING   *
130 REM *                 *
140 REM *****************
150 REM ----------------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     TARGT$:    MAIN STRING
190 REM     SUB$:      STRING TO BE INSERTED
200 REM     PLACE:     POSITION TO PUT SUB$
210 REM   RESULT --
220 REM     FINISH$:   COMPLETED STRING
230 REM
240 REM ----------------------------------

250 REM *** INITIALIZE ***

260 SUB$="TEST"
270 TARGT$="TARGET STRING LETTERS"
280 PLACE=7
290 GOSUB 13710
300 PRINT FINISH$
310 END

13700 REM *** SUBROUTINE ***

13710 L$=LEFT$(TARGT$,PLACE)
13720 R$=MID$(TARGT$,PLACE+1)
13730 FINISH$=L$+SUB$+R$
13740 RETURN
```

HOW TO USE SUBROUTINE

Sometimes a string to be inserted may need to be longer or shorter than the string replaced. This subroutine takes care of doing that with a few limitations.

Like all Commodore 128 strings, neither TARGT$ nor SUB$ can be longer than 255 characters (and keep in mind that you can enter a string only up to 160 characters at the keyboard). The resulting string with SUB$ inserted must be shorter than 255 characters as well.

In the subroutine as written, the target string is "TARGET STRING LETTERS", while the SUB$ is defined as "TEST". Since the PLACE where we want to insert it is position 7, the new string will read: "TARGET TESTSTRING LETTERS".

LINE-BY-LINE DESCRIPTION

Lines 260–280: Define the SUB$, the TARGT$, and PLACE where the SUB$ will be inserted.

Lines 290–310: Access the subroutine and display results.

Line 13710: Define L$ as left portion of string up to and including the character at position PLACE.

Line 13720: Define R$ as the remaining characters in the target string, from position following PLACE to the end.

Line 13730: Define FINISH$ as L$, plus the string to be inserted, plus R$.

YOU SUPPLY

Values for the main string, TARGT$, the string to be inserted, SUB$, and the position where it will be put, PLACE.

SUGGESTED ENHANCEMENTS: In your program have INSTR determine exactly where the insertion will take place.

RESULT

TARGT$ will have SUB$ inserted in it, at position PLACE.

SAMPLE VALUE: TARGET TESTSTRING LETTERS

CHR$ VALUE

WHAT IT DOES: Returns Commodore CHR$ code for any key.

LEVEL: Novice

```
100 REM **************
110 REM *            *
120 REM * CHR$ VALUE *
130 REM *            *
140 REM **************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     KEYBOARD INPUT
190 REM  RESULT --
200 REM     A:    CHR$ VALUE OF KEY
210 REM     A$:   KEY PRESSED
220 REM ----------------------------

230 REM *** INITIALIZE ***

240 GOSUB 13810
250 PRINT "CONTINUE ? (Y/N)"
260 GETKEY AN$
270 IF AN$="Y" GOTO 240
280 END

13800 REM *** SUBROUTINE ***

13810 GETKEY A$
13820 A=ASC(A$)
13830 PRINT"CHR$ VALUE OF KEY ";A$;" IS ";A
13840 RETURN
```

HOW TO USE SUBROUTINE

It is often necessary to know the special Commodore CHR$ code for a given key. If many keys are used, looking them all up on a table in your System Guide can be time-consuming. Instead, add this subroutine to the end of your program and call it as needed.

Just why would you want this capability? The answer lies in the differences in the ways computers and human beings like to process information. People are comfortable handling mixtures of alpha and numeric characters; computers recognize just binary numbers—ones and zeros. When string data are fed to a Commodore 128, they must be converted to a series of numbers that the processor can handle.

ASCII, or American Standard Code for Information Interchange, is one standard of communication that has been agreed upon so that computers can exchange alphanumeric information in a form that is common to processors with different operating

systems and languages. Commodore departs somewhat from this code for the Commodore 128, especially when using the graphics characters. PEEKing and POKing characters are done using the Commodore character set listing, not the standard ASCII table. However, the standard alphanumeric symbols can be represented by use of the correct CHR$(n) statement. In some cases, only a few characters need to be converted, so a table of codes and their string values will do the job. Other times, longer messages must be deciphered.

One good application for ASCII characters in programs is in game writing. Writers of BASIC Adventure-style programs may wish to "hide" messages from those casually LISTing the program. The CHR$(n) function can be used to assign the desired string values to string variables that are called at appropriate points in the program. CHR$(n) returns a one-character string that corresponds to the ASCII code of *n*. For example, PRINT CHR$(65) will produce an "A" on the screen.

A BASIC Adventure might have use for a message such as:

```
"LOOK IN THE HOLLOW STUMP."
```

This hint could be labeled H1$, and concatenated with CHR$(n) and the ASCII codes:

```
100 DATA 76,79,79,75,32,73,78,32,84,72,69,32
72,79,76,76,79,87,32,83,84,85,77,80
110 FOR N=1 to 25:READ A
120 H1$=H1$+CHR$(A)
130 NEXT A
```

Additional DATA lines and FOR-NEXT loops could be used to put any number of messages into string variables that are difficult to read accidentally. Of course, any knowledgeable programmer could pick the BASIC game apart, or enter PRINT H1$ from command mode once the program has been run past the initialization point. However, this technique assumes that the object is to protect the game player who innocently LISTS the program and doesn't want to spoil the fun. The same method can be used to hide program credits within BASIC code.

LINE-BY-LINE DESCRIPTION

Line 240: Access the subroutine.
Lines 250–280: See if user wants to access again.
Line 13810: Wait for user to press a key.
Line 13820: Find ASCII value of that key.
Line 13830: Print result.

YOU SUPPLY

Just press the key that you want to check.

SUGGESTED ENHANCEMENTS: Write a routine to store the results in a data file, and call them into your program as needed for display on the screen. Separating the program that writes the CHR$ codes from the program that uses them provides extra security. You can change the messages accessed by your second program simply by changing the DATA lines in the first.

RESULT

Variable A will equal CHR$ value of that key.

SAMPLE VALUE: A = CHR$(65)

EXCHANGE

WHAT IT DOES: Simulates SWAP function found in some other BASICs.

LEVEL: Novice

```
100 REM ************
110 REM *          *
120 REM * EXCHANGE *
130 REM *          *
140 REM ************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     A$:   FIRST VARIABLE
190 REM     B$:   SECOND VARIABLE
200 REM  RESULT --
210 REM     A$:   SECOND VARIABLE
220 REM     B$:   FIRST VARIABLE
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 A$="FIRST"
270 B$="SECOND"
280 PRINT "VALUE OF A$=";A$
290 PRINT "VALUE OF B$=";B$
300 GOSUB 13910
310 PRINT "VALUE OF A$=";A$
320 PRINT "VALUE OF B$=";B$
330 END

13900 REM *** SUBROUTINE ***

13910 DUMMY$=A$
13920 A$=B$
13930 B$=DUMMY$
13940 RETURN
```

HOW TO USE SUBROUTINE

Exchanging the value of one variable for that of another cannot be done in one step in Commodore 7.0 BASIC found in the Commodore 128. This feature is useful in sorts and in some other types of programming where the value of one variable needs to be traded with the value of another.

This subroutine will do that for you for any two string variables. To use it with numeric variables, you can use a second identical subroutine, with the string identifiers removed (i.e., DUMMY = A, A = B, B = DUMMY). A second, less elegant way is to change the numeric variables to strings before calling the routine. This can be done as follows: A$ = STR$(A): B$ = STR$(B): GOSUB xxx: A = VAL(A$):B = VAL(B$)

Not exactly efficient, right? Use two subroutines instead, one for strings and one for numbers.

LINE-BY-LINE DESCRIPTION

Lines 260–270: Define initial value of A$ and B$.
Lines 280–290: Show their values prior to the exchange.
Line 300: Access the subroutine.
Lines 310–320: Show values after the exchange.
Line 13910: Temporarily assign A$ to a dummy variable, DUMMY$.
Line 13920: make A$ equal to B$.
Line 13930: Make B$ equal to DUMMY$, which stores the original value of A$.

YOU SUPPLY

Values for A$ and B$, or A and B, the two variables that must be swapped.

SUGGESTED ENHANCEMENTS: It is usually simpler to perform this function in place without calling a subroutine.

RESULT

Values exchanged.

SAMPLE VALUES: A$ = FIRST; B$ = SECOND to A$ = SECOND; B$ = FIRST

STRING$

WHAT IT DOES: Simulates STRING$ function found in other BASICs.

LEVEL: Novice

```
100 REM ***********
110 REM *         *
120 REM * STRING$ *
130 REM *         *
140 REM ***********
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    LTH:      DESIRED LENGTH
190 REM    COMP$:    COMPONENT STRING
200 REM  RESULT --
210 REM    STRNG$:   FINISHED STRING
220 REM
230 REM ----------------------------

240 REM *** INITIALIZE ***

250 COMP$="A"
260 LTH=10
270 GOSUB 14010
280 PRINT STRNG$
290 END

14000 REM *** SUBROUTINE ***

14010 STRNG$=""
14020 DO WHILE LTH>0
14030 STRNG$=STRING$+COMP$:LTH=LTH-1
14040 LOOP
14050 RETURN
```

HOW TO USE SUBROUTINE

You may wish to define a string as being composed of 40 or 80 spaces, to clear a line. Or you may want to build a string made up of the word "HI" repeated 12 times. Both can be accomplished with this subroutine. The main limitation is that the resulting string must be 255 characters or shorter.

LINE-BY-LINE DESCRIPTION

Lines 250–260: Define the character to be used as the component string and the length of the desired string.

Lines 270–290: Access subroutine and print result.

Line 14010: Null any previous value of STRNG$.

Line 14020: Start procedure to continue while the length of STRNG$ is less than LTH, the desired length.

Line 14030: Add COMP$ to STRNG$.
Line 14040: Loop to repeat.
Line 250: Assign value of STRNG$ to chosen variable.

YOU SUPPLY

You need to define LTH with the desired number of times the component string will be repeated. If the component string has more than one character, this length will *not* be the same as the length of the finished string.

You also must supply a definition for COMP$, which may be either the actual characters or their CHR$ codes.

SUGGESTED ENHANCEMENTS: Add a routine to check to see if LTH*LEN(COMP$)>255, and, if so, require that LTH be reduced, or post a notice to that effect to the operator.

RESULT

Variable STRNG$ will be assembled using LTH copies of COMP$.

SAMPLE VALUE: AAAAAAAAAA

STRING SORT

WHAT IT DOES: Alphabetizes a list.

LEVEL: Intermediate

```
100 REM ***************
110 REM *               *
120 REM * STRING SORT *
130 REM *               *
140 REM ***************
150 REM -------------------------------------
160 REM       ++ VARIABLES ++
170 REM SUPPLIED BY USER --
180 REM      NUMBER :   NUMBER OF ITEMS SORTED
```

```
190 REM      D$(N):      ARRAY WITH ITEMS
200 REM   RESULT --
210 REM      D$(N) SORTED
220 REM
230 REM ------------------------------------

240 REM *** INITIALIZE ***

250 NUMBER=10
260 DIM D$(NUMBER)
270 GOSUB 14110
280 FOR N=1 TO NUMBER
290 PRINT D$(N)
300 NEXT N
310 END

14100 REM *** SUBROUTINE ***

14110 FOR ITEM=1 TO NUMBER
14120 PRINT"ENTER #";ITEM
14130 INPUT D$(ITEM)
14140 NEXT ITEM
14150 FOR N=1 TO NUMBER
14160 FOR N1=1 TO NUMBER-N
14170 A$=D$(N1)
14180 B$=D$(N1+1)
14190 IF A$<B$ THEN GOTO 14220
14200 D$(N1)=B$
14210 D$(N1+1)=A$
14220 NEXT N1
14230 NEXT N
14240 RETURN
```

HOW TO USE SUBROUTINE

Sorting a list is a common need for many programs. Data files, mailing lists, and other collections of information may be more easily handled when sorted. This routine is a simple bubble sort which will alphabetize any list that has been loaded into an array, D$(n).

Although as written the subroutine asks the user to enter the list from the keyboard, any means can be used to load the array. The file may also be read from disk, for example, via one of the routines presented in Chapter 2.

The *bubble sort* is so-called because each entry in the array is examined and then allowed to rise up past the one below until it encounters a "smaller" item. When one is comparing strings, *smaller* is defined as an entry that, when alphabetized, comes

before the larger entry. That is, "computerization" is smaller than "contain" even though it has more letters, because it would be placed on an alphabetized list first. In computer terminology we would say that: "computerization"<"contain" is a true statement. In making the comparison between strings, the Commodore 128 will look at as many characters in the string as necessary to differentiate. For example, "contain"<"contains."

In the bubble sort, each element of the array will gradually rise until it encounters a smaller item. Eventually, each member of the list "floats" up to its proper place in the array. While such sorts are not very fast, for small lists of, say, 30 or 40 items the speed is satisfactory.

LINE-BY-LINE DESCRIPTION

Line 250: Define NU, the number of units in the array to be sorted.

Line 260: DIMension the array to proper size.

Lines 270–310: Access the subroutine and print results.

Line 14100: Begin loop from 1 to number of items to be sorted.

Lines 14110–14140: Get the items from user keyboard input. A disk or tape file read routine could be substituted for these lines to sort an existing string file.

Line 14150: Start loop from 1 to the number of items to be sorted.

Line 14160: Start a nested loop from 1 to 1 less than the number of items to be sorted.

Line 14170: Make A$ equal to the N1th item of the array.

Line 14180: Make B$ equal to the item following A$ in the array.

Line 14190: If the "higher" element, A$, is already smaller than B$, then B$ remains where it is and the inner loop steps off the next value of N1.

Lines 14200–14210: If B$ is smaller than A$, then the two strings are swapped, with B$ moving ahead one element and A$ being pushed down one.

Lines 14220–14230: The inner and outer loops are incremented.

YOU SUPPLY

You should define NUMBER, the number of items to be sorted, and also supply the data for the array, D$(n).

SUGGESTED ENHANCEMENTS: Interface this routine with data files produced by your program.

RESULT

List is sorted alphabetically.

SAMPLE VALUES: APPLE,ORANGE,PEAR,WATERMELON

SHELL/METZNER SORT

WHAT IT DOES: Sorts a list.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                     *
120 REM * SHELL-METZNER SORT *
130 REM *                     *
140 REM **********************
150 REM ------------------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     NUMBER:    NUMBER OF ITEMS OF DATA
190 REM     FIRST:     START POSITION OF SORT
200 REM     LAST:      END POSITION OF SORT
210 REM     D$(N):     DATA ARRAY TO BE SORTED
220 REM   RESULT --
230 REM     D$(N) SORTED
240 REM
250 REM ------------------------------------

260 REM *** DATA ***

270 DATA COMMODORE,VIC,APPLE,ORANGE
280 DATA MONITOR,KEYBOARD,PRINTER
290 DATA LISTING,PROGRAM,INSERT

300 REM *** INITIALIZE ***
```

```
310 NUMBER=10
320 FIRST=1
330 LAST=10
340 DIM D$(NUMBER)
350 FOR I=1 TO NUMBER
360 READ D$(I)
370 NEXT I
380 GOSUB 14310
390 FOR N=FIRST TO LAST
400 PRINT D$(N)
410 NEXT N
420 END

14300 REM *** SUBROUTINE ***

14310 T1=NUMBER
14320 T1=INT(T1/2)
14330 IF T1=0 THEN RETURN
14340 T2=1
14350 T3=NUMBER-T1
14360 T5=T2
14370 T4=T5+T1
14380 S1$=(MID$(D$(T5),FIRST,(LAST-FIRST)+1))
14390 S2$=(MID$(D$(T4),FIRST,(LAST-FIRST)+1))
14400 IF S1$<S2$ GOTO 14470
14410 T6$=D$(T5)
14420 D$(T5)=D$(T4)
14430 D$(T4)=T6$
14440 T5=T5-T1
14450 IF T5<1 THEN GOTO 14470
14460 GOTO 14370
14470 T2=T2+1
14480 IF T2>T3 THEN GOTO 14320
14490 GOTO 14360
14500 RETURN
```

HOW TO USE SUBROUTINE

The Shell-Metzner Sort is considered more efficient for slightly longer lists than the bubble sort. There are many other sorts, such as Quicksort and Heapsort, but this sort is easy to understand without a lengthy explanation and is entirely adequate for most BASIC programs.

LINE-BY-LINE DESCRIPTION

Lines 270–290: Data to be sorted.

Lines 310–330: Define the NUMBER of items to be sorted, the position of the FIRST item to be sorted, and the LAST item

to be sorted. With this routine you may sort only *part* of a list if you wish.

Line 340: DIMension an array to hold the data items.

Lines 350–370: Read data to the array.

Lines 380–420: Access the subroutine and print the sorted list to the screen.

Lines 14310–14330: Divide the list in half.

Lines 14340–14390: Extract two elements from each half.

Line 14400: Compare, and if first is "smaller," bypass the swap routine.

Lines 14410–14460: Exchange the elements.

Lines 14470–14490: Increment the pointers and repeat.

YOU SUPPLY

Either data in DATA lines, or data file to be sorted.

SUGGESTED ENHANCEMENTS: Interface with your own data files.

RESULT

List sorted alphabetically.

SAMPLE VALUES: APPLE,COMMODORE,INSERT,KEYBOARD, LISTING,MONITOR,ORANGE,PRINTER,PROGRAM,VIC

ARRAY LOADER

WHAT IT DOES: Loads array with data.

LEVEL: Novice

```
100 REM **************
110 REM *              *
120 REM * LOAD ARRAY *
130 REM *              *
140 REM **************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NROWS:       NUMBER OF ROWS
190 REM    NCOLUMNS:    NUMBER OF COLUMNS
200 REM  RESULT --
210 REM    DTA$(NR,NC) LOADED WITH DATA
220 REM
230 REM ----------------------------

240 REM *** DATA ***

250 DATA DAVE,1 PINE ST.,445-1881
260 DATA FRED,3 HIGH ST.,232-4444

270 REM *** INITIALIZE ***

280 NROWS=2
290 NCOLUMNS=3
300 DIM DTA$(NR,NC)
310 GOSUB 14610
320 FOR ROW=1 TO NROWS
330 FOR COLUMN=1 TO NCOLUMNS
340 PRINT DTA$(ROW,COLUMN)
350 NEXT COLUMN
360 NEXT ROW
370 END

14600 REM *** SUBROUTINE ***

14610 FOR ROW=1 TO NROWS
14620 FOR COLUMN=1 TO NCOLUMNS
14630 READ A$
14640 DTA$(ROW,COLUMN)=A$
14650 NEXT COLUMN
14660 NEXT ROW
14670 RETURN
```

HOW TO USE SUBROUTINE

An array is a table with rows and columns storing lists of data. In a checkbook register each row might contain information about a single check/deposit transaction. The columns would contain specific entries, such as check number, payee, data, and amount.

Once a data file has been assembled with such information, a routine is needed to load it into an array where it can be manipu-

lated, sorted, added to, or have entries deleted. This subroutine does exactly that. Although written for a string array, it can be converted to a numeric array simply by the user's deleting the variable type specifier, "$". That is, DTA$(row,column) should become DTA(row,column), and A$ should be changed to A.

Study this example to learn more of how arrays work, as they are one of the most important concepts in BASIC programming.

LINE-BY-LINE DESCRIPTION

Lines 250–260: Data to be loaded.

Line 280: Define number of rows in the array.

Line 290: Define number of columns in the array.

Line 300: DIMension an array to hold the data.

Lines 310–370: Access the subroutine and then print the results, using nested FOR-NEXT loops. (See below.)

Line 14610: Begin a FOR-NEXT loop from 1 to the number of rows.

Line 14620: Begin a second FOR-NEXT loop nested inside the first one, from 1 to the number of columns.

Line 14630: READ a data item from the DATA lines.

Line 14640: Store that value in DTA$(row,col), in the positions marked by the value of ROW,COLUMN. The first time through the loop, ROW will equal 1 and COLUMN will equal 1. Then control will drop to Line 14650 below, where COLUMN will be incremented to 2. So, the next DATA item will be stored in DTA$(1,2). When all the columns for a given row have been filled, ROW will be incremented and the next data item will be stored in DTA$(2,1).

Line 14650: Next COLUMN.

Line 14660: Next ROW.

YOU SUPPLY

The number of rows, NROWS, and number of columns, NCOLUMNS, should be specified. Data can be supplied from DATA lines, or, better, read in from disk or tape.

SUGGESTED ENHANCEMENTS: Adapt this subroutine for your own programs.

RESULT

Data list is loaded into array.

SAMPLE VALUE: The DATA will be printed in order shown in listing

DESPACER

WHAT IT DOES: Removes spaces from a string.

LEVEL: Intermediate

```
100 REM *************
110 REM *           *
120 REM * DESPACER  *
130 REM *           *
140 REM *************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    S$:       STRING TO DESPACE
190 REM  RESULT --
200 REM    RESULT$: DESPACED STRING
210 REM
220 REM --------------------------------

230 REM *** INITIALIZE ***

240 INPUT "ENTER STRING TO DESPACE :";S$
250 GOSUB 14710
260 PRINT "NEW STRING :";RESULT$
270 END

14700 REM *** SUBROUTINE ***

14710 RESULT$=S$
14720 P=1
14730 B=INSTR(RESULT$,CHR$(32),P)
14740 IF B=0 THEN RETURN
14750 RESULT$=LEFT$(RESULT$,B-1)+MID$(RESULT$,B+1)
14760 P=B
14770 GOTO 14730
```

HOW TO USE SUBROUTINE

This subroutine demonstrates how to manipulate strings to add or subtract from them what we want. Earlier in this chapter you saw how to insert a new string in the middle of a string or to replace a portion of a string with another. To continue in that vein, you can simply remove all the spaces from a string using this subroutine. Your programming may have the need for this technique, as when compacting a data file in which spaces are not desired or needed to delimit characters.

The subroutine looks at each character in the string in turn, using a FOR-NEXT loop from 1 to the length of the string. If a space is found, the routine skips over it. All other characters are added to a temporary string, T$, which is finally used to redefine the original string when all the spaces have been removed.

LINE-BY-LINE DESCRIPTION

Line 240: Ask user for string to despace. Strings entered from the keyboard can be only 160 characters long. Strings you define in a program can be combined up to 255 characters.

Line 250: Access the subroutine.

Line 260: Print result.

Line 14710: Store value of string input in RESULTS.

Line 14720: Define search position as 1.

Line 14730: Store in variable B the position of the first space in the string following position P.

Line 14740: If B is 0, then no more spaces remain: RETURN to main program.

Line 14750: Take the left and right portions of the string, excluding the space found, and combine them. Doing this removes the space from the string.

Line 14760: Redefine P as the place where the space was found, so that the next search will start there, looking for additional spaces.

Line 14770: Repeat.

YOU SUPPLY

String to despace.

SUGGESTED ENHANCEMENTS: Interface with your program.

RESULT

String has spaces removed.

SAMPLE VALUE: THISSTRINGHASBEENDESPACED

CENTER STRING

WHAT IT DOES: Centers string on the screen.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * CENTER STRING *
130 REM *                 *
140 REM *****************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     A$:     STRING TO BE CENTERED
190 REM     WIDE:   SCREEN WIDTH
200 REM   RESULT --
210 REM     L:      AMOUNT TO TAB THE STRING
220 REM
230 REM --------------------------------

240 REM *** INITIALIZE ***

250 WIDE=80
260 INPUT "ENTER STRING TO BE CENTERED :";A$
270 GOSUB 14810
280 IF L=0 THEN GOTO 260
290 PRINT TAB(L)A$
300 END

14800 REM *** SUBROUTINE ***

14810 L=0
14820 IF LEN(A$)>WIDE-2 THEN PRINT "STRING TOO LONG!":RETURN
14830 L=INT((WIDE-LEN(A$)))/2
14840 RETURN
```

HOW TO USE SUBROUTINE

You may want your program to center prompts or other material on the screen for a neat appearance. This subroutine works with either 80- or 40-column screens. It calculates the length of the string and subtracts that amount from the number of spaces available to determine the amount left over. This figure is divided by two so this extra space is arranged as equally as possible before and after the string. If an uneven number of spaces remains, the string will be "centered" one character to the left, which should be almost an unnoticeable difference.

Call the subroutine each time you want to center a string on the screen. The module is handy because it can be used for prompts that you don't know the length of ahead of time. For example:

```
1000 INPUT "ENTER YOUR NAME :";NME$
1010 S$="HELLO THERE, "+NME$
1020 WIDE=80
1030 GOSUB 14810
1040 PRINT TAB(L)S$
1050 END
```

LINE-BY-LINE DESCRIPTION

Line 250: Define whether program is using the 40-column or 80-column screen.

Line 260: User inputs line to be centered.

Line 270: Access the subroutine.

Line 280: If string was too long (equal to or greater than WIDE-1) then go back and ask for new input.

Line 290: Otherwise tab and print the string.

Line 14810: Set tab value to 0, in case this subroutine has been called previously by the program.

Line 14820: See if string is too long.

Line 14830: Calculate value for L, as half of the extra spaces left on the line.

Line 14840: Return.

YOU SUPPLY

String to be centered.

SUGGESTED ENHANCEMENTS: None.

RESULT

String is centered on the screen.

SAMPLE VALUE: Not applicable

FLUSH RIGHT STRING

WHAT IT DOES: Prints string flush right on the screen.

LEVEL: Novice

```
100 REM ***************
110 REM *               *
120 REM * FLUSH RIGHT *
130 REM *               *
140 REM ***************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    A$:    STRING TO BE FLUSH RIGHT
190 REM    WIDE:  SCREEN WIDTH
200 REM  RESULT --
210 REM    R:     AMOUNT TO TAB THE STRING
220 REM
230 REM --------------------------------

240 REM *** INITIALIZE ***

250 WIDE=80
260 INPUT "ENTER STRING  :";A$
270 GOSUB 14910
280 IF R=0 THEN GOTO 260
290 PRINT TAB(R)A$;
300 END

14900 REM *** SUBROUTINE ***

14910 R=0
14920 IF LEN(A$)>WIDE THEN PRINT "STRING TOO LONG!":RETURN
14930 R=WIDE-LEN(A$)
14940 RETURN
```

HOW TO USE SUBROUTINE

Use when you wish to print a string flush right on the screen. This may be the case with columns of numbers or other material. The routine works with either 40- or 80-column screens.

LINE-BY-LINE DESCRIPTION

Line 250: Define WIDE as width of screen being used with the subroutine.

Line 260: Enter string to be printed flush right.

Line 270: Access the subroutine.

Line 280: If string was too long, go back and ask for new string.

Line 290: TAB to print string at right side of screen.

Line 14910: Return R to 0, in case routine has been called before during this program run.

Line 14920: Check to see if string is too long to display on a single line.

Line 14930: Calculate amount to TAB.

Line 14940: Return.

YOU SUPPLY

String to print flush right.

SUGGESTED ENHANCEMENTS: Modify the routine so the string can be printed flush right in a column *other* than the far right of the screen. HINT: Change the value of WIDE.

RESULT

String is printed at the right side of the screen.

SAMPLE VALUE: Not applicable

ENCODE STRING

WHAT IT DOES: Encodes strings to prevent reading.

LEVEL: Intermediate

```
100 REM *****************
110 REM *                 *
120 REM * ENCODE STRING   *
130 REM *                 *
140 REM *****************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     S$:        STRING TO BE ENCODED
190 REM  RESULT --
200 REM     RESULT$:   ENCODED STRING
210 REM
220 REM --------------------------------

230 REM *** INITIALIZE ***

240 INPUT "ENTER STRING TO BE ENCODED :";S$
250 INPUT "ENTER CODE NUMBER (1 TO 25)";CODE
260 GOSUB 15010
270 PRINT "ENCODED STRING :";RESULT$
280 END

15000 REM *** SUBROUTINE ***

15010 TEMP$=""
15020 FOR N=1 TO LEN(S$)
15030 B$=MID$(S$,N,1)
15040 B=ASC(B$)
15050 IF B>192 AND B<219 THEN B=B-128
15060 IF B>64 AND B<91 THEN BEGIN
15070 C=B+CODE
15080 IF C>90 THEN C=C-26
15090 BEND
15100 IF B<65 OR B>92 THEN C=B
15110 TEMP$=TEMP$+CHR$(C)
15120 NEXT N
15130 RESULT$=TEMP$
15140 RETURN
```

HOW TO USE SUBROUTINE

This subroutine introduces the concept of encrypting data to prevent unauthorized access by individuals. You could encode each string of a data file prior to writing it to disk. No one else

could LIST that file and read it without having your decoding subroutine (which follows). You might keep the decoding program on a separate disk for added security.

There are many different methods of encrypting information. A substitution cipher, like the one in this subroutine, is the easiest to break, since the frequency of words and letters in English is well-known. In fact, this particular scheme is easier than most, because it merely moves the entire alphabet one or more characters to the right, using a code number from 1 to 25 entered by the user. It would be necessary only to figure out the number and then subtract that from each letter to decode the data.

However, the object of this subroutine is to provide a simple encoding procedure that can be slipped into any program and used to make it more difficult for the casual intruder to read the data. If you are serious about security, investigate special encrypting programs, which will doubtless be adapted to the Commodore 128 by the time this book is published.

LINE-BY-LINE DESCRIPTION

Lines 240–250: Ask for string to encode, and a code number.

Lines 260–280: Access the subroutine and print results.

Line 15010: Null the temporary string, TEMP$.

Line 15020: Begin loop from 1 to length of S$, the string to be encoded.

Line 15030: Extract middle character from string, at position N.

Line 15040: Calculate ASCII value of that character.

Lines 15050–15060: Change lowercase to upper, then test to see if it is an alpha character (other characters are left unchanged).

Line 15070: Add the code to the ASCII value of the character.

Lines 15080–15090: If new value is higher than Z, then wrap around to beginning of alphabet.

Line 15100: If nonalpha character found, simply pass it through.

Line 15110: Add character to TEMP$.

Line 15120: Loop to next character.

Lines 15130–15140: After all characters are examined, store result in RESULT$, and return.

YOU SUPPLY

String to encode.

SUGGESTED ENHANCEMENTS: Add lines to encode *all* characters, in addition to A–Z. Make the encoding algorithm more complex, so that a different number may represent the same character. For example, you might make all two-digit numbers higher than 26 that are multiples of 11 represent spaces: 33, 44, 55, 66, etc. Read up on data encryption to see how truly sophisticated methods can be.

RESULT

String encoded.

SAMPLE VALUE: THIS IS A SAMPLE = YMNX NX F XFRUQZ (Code 5)

DECODE STRING

WHAT IT DOES: Decodes strings encoded with previous subroutine.

LEVEL: Intermediate

```
100 REM *****************
110 REM *                 *
120 REM * DECODE STRING   *
130 REM *                 *
140 REM *****************
150 REM --------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    S$:        STRING TO BE DECODED
190 REM  RESULT --
```

```
200 REM     RESULT$:  DECODED STRING
210 REM
220 REM -------------------------------

230 REM *** INITIALIZE ***

240 INPUT "ENTER STRING TO BE DECODED :";S$
250 INPUT "ENTER CODE NUMBER (1 TO 25)";CODE
260 GOSUB 15210
270 PRINT "DECODED STRING :";RESULT$
280 END

15200 REM *** SUBROUTINE ***

15210 TEMP$=""
15220 FOR N=1 TO LEN(S$)
15230 B$=MID$(S$,N,1)
15240 B=ASC(B$)
15250 IF B>192 AND B<219 THEN B=B-128
15260 IF B>64 AND B<91 THEN BEGIN
15270 C=B-CODE
15280 IF C<65 THEN C=C+26
15290 BEND
15300 IF B<65 OR B>92 THEN C=B
15310 TEMP$=TEMP$+CHR$(C)
15320 NEXT N
15330 RESULT$=TEMP$
15340 RETURN
```

HOW TO USE SUBROUTINE

This is the reverse of the previous subroutine. It takes the code number you enter and decodes the strings you provide using the same formula. If you forget your code number, you can try all 25 until your encrypted text makes sense.

LINE-BY-LINE DESCRIPTION

Lines 240–250: Ask for string to decode, and a code number.

Lines 260–280: Access the subroutine and print results.

Line 15210: Null the temporary string, TEMP$.

Line 15220: Begin loop from 1 to length of S$, the string to be decoded.

Line 15230: Extract middle character from string, at position N.

Line 15240: Calculate ASCII value of that character.

Lines 15250–15260: Change lowercase to upper, then test to see if it is an alpha character (other characters are left unchanged).

Line 15270: Subtract the code from the ASCII value of the character.

Lines 15280–15290: If new value is less than "A", then wrap around to end of alphabet.

Line 15300: If nonalpha character found, simply pass it through.

Line 15310: Add character to TEMP$.

Line 15320: Loop to next character.

Lines 15330–15340: After all characters are examined, store result in RESULT$, and return.

YOU SUPPLY

String to decode.

SUGGESTED ENHANCEMENTS: Write routine so program will attempt to decode using several different code numbers automatically, to help user decode strings where code number is not known.

RESULT

String decoded.

SAMPLE VALUE: YMNX NX F XFRUQZ = THIS IS A SAMPLE (Code 5)

WORD COUNTER

WHAT IT DOES: Counts the words in a sequential file.

LEVEL: Intermediate

```
100 REM ****************
110 REM *                *
120 REM * WORD COUNTER *
130 REM *                *
140 REM ****************
150 REM -----------------------------------
160 REM       ++ VARIABLES ++
170 REM SUPPLIED BY USER --
180 REM    DATA FILE
190 REM    FILENAME IN LINE 15410 OR 15420
200 REM  RESULT --
210 REM    CU:      NUMBER WORDS IN FILE
220 REM    CH:      NUMBER CHARACTERS IN FILE
230 REM    AV:      AVERAGE WORD LENGTH
240 REM    SW:      NUMBER STANDARD WORDS
250 REM
260 REM  NOTE: EITHER LINE 15410
270 REM  NOTE: OR 15420 MAY BE USED
280 REM -----------------------------------

290 REM *** INITIALIZE ***

300 GOSUB 15410
310 PRINT "WORDS :";
320 PRINT USING PLACE$;CU
330 PRINT "AVERAGE WORD LENGTH :";
340 PRINT USING PLACE$;AV
350 PRINT "NUMBER OF FIVE-CHARACTER WORDS :";
360 PRINT USING PLACE$;SW
370 END

15400 REM *** SUBROUTINE ***

15410 REM OPEN 1,8,8,"0:FILENAME,S,R"
15420 DOPEN #1,"FILENAME,S",D0,R
15430 DO
15440 GET#1,A$
15450 CH=CH+LEN(A$)
15460 FOR N=1 TO LEN(A$)
15470 C$=MID$(A$,N,1)
15480 IF C$=CHR$(32) AND L$<>CHR$(32) THEN CU=CU+1
15490 L$=C$
15500 NEXT N
15510 LOOP UNTIL ST
15520 DCLOSE #1
15530 AV=CHAR/CU
15540 SW=CHAR/5
15550 PLACE$="##.#"
15560 RETURN
```

HOW TO USE SUBROUTINE

Some word processing programs produce simple sequential text files that can be read into BASIC and manipulated easily.

This subroutine demonstrates what can be done, using word counting as an example. It will read in the file, count the number of words, and display the words counted, the average word length, and the number of "average" five-character words in the file. The latter measure, when compared with average word length, helps show just how difficult the text is. The shorter the average word length, presumably the easier the text is to read. This subroutine considers any space that is preceded by a character that is *not* a space to mark the end of a word. This system works pretty well; double spaces are not counted as word markers, but spaces at the ends of sentences or at the ends of words do count.

LINE-BY-LINE DESCRIPTION

Line 310: Access the subroutine.

Line 320: Print number of words counted.

Lines 330–340: Print average word length.

Lines 350–360: Print number of five-character words.

Lines 15410–15420: Open the sequential file.

Line 15430: While STATUS variable indicates there is still information in the file, DO the following loop.

Line 15440: GET a string from the file.

Line 15450: Assign length of the new string to CH, which keeps track of the total number of characters in the file.

Line 15460: Start FOR-NEXT loop to look at each character in the string.

Line 15470: Extract single character from middle of the string at position N.

Line 15480: If the character is a space and the *last* character looked at, L$, was not a space, then increment the word counter, CU.

Line 15490: Assign value of C$ to L$, making it the new last character read prior to looping again.

Line 15500: Next N.

Line 15510: Loop while ST remains true.

Line 15520: After all information in file has been processed, close the file.

Line 15530: Divide the number of characters in the file by the number of words found to figure average word length.

Line 15540: Calculate standard words by dividing character total by 5.

Line 15550: Define PLACE$ as PRINT USING format.

YOU SUPPLY

Data file to count.

RESULT

Words in the file counted.

SAMPLE VALUE: Does not apply

GLOBAL SEARCH

WHAT IT DOES: Searches a file for a string and replaces each occurrence with a new string.

LEVEL: Intermediate

```
100 REM *****************
110 REM *                 *
120 REM * GLOBAL SEARCH   *
130 REM *                 *
140 REM *****************
150 REM -----------------------------------
160 REM       ++ VARIABLES ++
170 REM SUPPLIED BY USER --
180 REM    DATA FILE
190 REM    FILENAMES IN LINE 15610 AND 15620
200 REM  RESULT --
210 REM    S$:   STRING TO SEARCH FOR
220 REM    RE$:  STRING TO REPLACE WITH
230 REM
240 REM -----------------------------------

250 REM *** INITIALIZE ***

260 GOSUB 15610
270 PRINT "SEARCH/REPLACE COMPLETED"
280 END
```

```
15600 REM *** SUBROUTINE ***

15610 DOPEN #1,"FILENAME"
15620 DOPEN #2,"FILE2NAME,S",W
15630 INPUT "ENTER STRING TO SEARCH FOR :";S$
15640 INPUT "ENTER STRING TO REPLACE WITH :";RE$
15650 DO
15660 INPUT#1,A$
15670 P=1:R=INSTR(A$,S$,P)
15690 IF R=0 GOTO 15750
15700 L$=LEFT$(A$,R-1)
15710 E=LEN(S$)
15720 R$=MID$(A$,R+E)
15730 A$=L$+RE$+R$
15740 P=INSTR(A$,RE$,P)+LEN(RE$)-1
15750 PRINT #2,A$;
15760 LOOP UNTIL ST
15770 DCLOSE #2
15780 DCLOSE #1
15790 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will search through a file, changing all occurrences of the string you specify with the one you wish to replace it with. Useful for word processing and other applications.

LINE-BY-LINE DESCRIPTION

Line 260: Access the subroutine.
Line 270: Notify operator of completion of task.
Line 15610: Open file name to read.
Line 15620: Open file name for new file with changes.
Line 15630: Ask for string to search for.
Line 15640: Ask for replacement string.
Line 15650: Start DO loop.
Line 15660: GET string from the file.
Line 15670: Define position to begin search, P, as 1 for start of search of this string.
Line 15690: If it is not present in string, go to the file write routine.
Line 15700: Take left portion of the string, up to position where the search string was found (this is the same as the Replace String subroutine).

Line 15710: Define the position of the search string as E.

Line 15720: Take the right portion of the string.

Line 15730: Assemble new string.

Line 15740: Redefine P to position following where search string was found.

Line 15750: Print new string to new file.

Line 15760: Loop until STATUS indicator shows that no more information is in the file.

Lines 15770–15780: Close both files.

YOU SUPPLY

Sequential file to process.

SUGGESTED ENHANCEMENTS: You may write the new string segment to the file immediately and thus avoid the problem of having A$ possibly longer than 255 characters. Use of a semicolon at the end of the PRINT # statement will allow you to put each successive part of the current string being processed back as a continuous stream of characters. Only the CHR$(13) (carriage returns) that marked the original file will be transported over to the new one. Study sequential files in the *Commodore 128 Programmer's Reference Guide* for more information on this.

RESULT

File searched and replacement made.

SAMPLE VALUE: Does not apply

5 GAME ROUTINES

Games are probably among the most popular programming exercises for beginners and advanced users alike. On your first day with a computer you can easily learn to write a "craps"-playing computer program. Those Commodore 128 users who are not all business will probably at some time write an arcade-quality joystick-activated, shoot-em up. What they learn in writing games may have broad applications in other areas of programming as well.

In any case you can use some of the routines in this book to avoid reinventing the wheel. Actually, the subroutines useful for games programming are not confined to this section. Many of the modules presented in other chapters can be transplanted to games

programs. Those here are labeled "games routines" because they are particularly apt.

Three routines deal with universal tools of games: decks of cards, rolling dice, and flipping coins. We've tried to take even these basic routines beyond the basic. The dice module allows you to specify more than two dice; you can roll five or six or more during one roll. The dice can be *n*-sided as well; they are not limited to the standard six sides. Dungeon and Dragons players take note.

The coin-flip (coins are actually just two-sided dice, if you think about it) has been spiced up by an animation routine that makes it seem as if the coin is actually being thrown up in the air prior to landing heads or tails. While a simple subroutine isn't sufficient to provide full-fledged graphics images of real playing cards, our subroutine shows how to deal a deck efficiently and uses the Commodore 128's graphics characters to portray suits.

Randomness is briefly explored in a pair of routines, and the chapter starts off with a comprehensive look at accessing the joysticks to manipulate objects on the screen. Subroutines are provided to move objects on both the 40-column and 80-column screens, as is a keyboard-joystick simulation module for those who want to use the cursor keys instead of hunting up a joystick to plug in. Several of the routines allow you to draw on the screen in colors, an introduction to the more sophisticated graphics work that follows in Chapter 6.

USING JOYSTICKS

Let's start off with one of the most mystifying capabilities of the Commodore 128: joystick manipulation of screen objects. All of us have marveled at arcade-quality games for the Commodore 128 that let the player use a joystick to move a space ship, racing car, or other token around on the screen while firing "missiles" at oncoming attackers.

Unfortunately, getting objects, sprites, or other things to move is not just a matter of joystick action directly causing movement

in a desired direction. The process is a bit more complex than that, and this chapter includes no less than ten joystick subroutines to explore different aspects of it.

From a programmer's standpoint, the use of the joysticks breaks down into several neat modules. When dealing with nonsprite objects, we need to know where on the Commodore 128's screen the object to be moved is. We also must check, as often as possible, the status of the player's joystick to see if it is pressed in any direction, or if the FIRE button has been depressed. If so, the programmer must update the location of the object on the screen. This is usually done by changing the value of the location where the object is printed, by adding or subtracting. We need to put the object in the new position and, if we do not want to leave a "trail" behind the object, we need to erase its image from the old location.

The joystick routines have been included in this section to allow several different types of object movement in your programs. Some allow moving an object only in north/south, east/west directions. Others allow diagonal movement as well and tell you when the fire button has been pressed.

One subroutine demonstrates how to move an object by erasing its old position after the move has been made. The others illustrate leaving a trail behind the moving object. Both types of movement will be useful for game programs.

A final type of subroutine brings all the elements together in a short program that will allow you to draw on the screen using a joystick.

80-COLUMN JOYSTICK—HORIZONTAL MOVEMENT

WHAT IT DOES: Moves object left and right only.

LEVEL: Advanced

```
100 REM ************************
110 REM *                      *
120 REM * 80 COLUMN JOYSTICK   *
130 REM * HORIZONTAL MOVEMENT  *
140 REM *                      *
150 REM ************************
160 REM --------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:   CURSOR CHARACTER
210 REM    ROW:  ROW DESIRED
220 REM    JOYSTICK MOVEMENT
230 REM  RESULT --
240 REM    CURSOR MOVES ACROSS SCREEN
250 REM    IN DESIRED ROW.  FLASHES RED
260 REM    WHEN FIRE BUTTON IS PRESSED.
270 REM
280 REM --------------------------

290 REM *** INITIALIZE ***

300 ROW=20
310 CURSR$=CHR$(209)
320 GOTO 17010

17000 REM *** SUBROUTINE ***

17010 PRINT CHR$(154)
17020 P=1
17030 DATA -0,-0,+1,0,0,0,-1,0
17040 FOR N=1 TO 8
17050 READ P(N)
17060 NEXT N
17070 SCNCLR
17080 P$(3)=CHR$(29)
17090 P$(7)=CHR$(157)
17100 FOR N=1 TO ROW:PRINT CHR$(17):NEXT N
17110 GOTO 17130
17120 PRINT CHR$(157);
17130 A=JOY(1)
17140 PRINT CHR$(154);
17150 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
17160 IF FLAG=1 THEN PRINT CHR$(42);CHR$(157);CHR$(150);
17170 P=P+P(A)
17180 PRINT CHR$(32);CHR$(157);
17190 IF P<1 OR P>79 THEN P=P-P(A):GOTO 17130
17200 B$=P$(A)
17210 PRINT B$;CURSR$;
17220 GOTO 17120
```

HOW TO USE THE SUBROUTINE

While the Commodore 128's ports can be used with joysticks, paddles, light pens, and other accessories, the joysticks are proba-

bly the most flexible controllers for games. They can be used to control movement in all directions as well as just from left to right, like a paddle.

A paddle is like a potentiometer or volume control that returns an analog value, i.e., some number you supply between 0 and 255, depending on how far the paddle has been turned. It provides an absolute indication of where the object should be on the screen, the horizontal or vertical coordinate, so to speak, relative to the paddle's position. Unfortunately, a small movement of the paddle can result in a large movement of the object on the screen. The use of paddles from BASIC is *not* recommended by Commodore, because they can be unreliable.

Some games work better when the joystick control is used for movement left and right. The joystick is not an absolute equivalent of a paddle. The nonanalog joysticks used with the Commodore 128 have no way of indicating exactly where on the screen an object should be. Instead, the joystick will indicate a zero when not pressed in a direction and a number when that switch is closed.

This approach makes game programming simpler. When the joystick is pressed to the right, move the object one position to the right. Keep doing that as long as the joystick remains pressed rightward.

This subroutine will tell you whether or not the joystick is pressed to the right or left, plus report on the status of the FIRE button. Your program should repeatedly check to see which direction (if any) is indicated and take action accordingly. Sample movement is provided and explained below under Line-by-Line Description.

Because of the complexity of this topic, some of the descriptions in this chapter will be a bit more verbose than those in earlier parts of the book.

LINE-BY-LINE DESCRIPTION

Line 300: You'll be moving the object only in a single row on the screen, simulating a paddle such as used in games like Breakout. Here the row is defined as row 20.

Line 310: The object being moved, which we'll call CURSR$, is defined as CHR$(209), a solid ball.

Line 320: The subroutine is accessed from here. In this case the routine is a standalone program and not a subroutine, so calling it is not really necessary.

Line 17010: Change character color to light gray.

Line 17020: Define initial position of cursor as 1.

Line 17030: DATA lines showing direction cursor should move when a given value of JOY(n) is returned.

Lines 17040–17060: Read those values to the array P(n).

Line 17070: Clear the screen.

Lines 17080–17090: Define P$(3) as CHR$(29), the CURSOR RIGHT character, and P$(7) as the CURSOR LEFT character.

Line 17100: Move cursor down ROW rows, by printing CHR$(17), which is the cursor down character.

Line 17110: Skip to the "read joystick" line.

Line 17120: Move the cursor to the left, using CHR$(157), the CURSOR LEFT character. This is done at the start of each loop through the joystick routine. The reason will become apparent shortly.

Line 17130: Store in variable A the value returned by reading Joystick 1. This value will vary depending on how the joystick is pressed. If it is set to neutral position, a 0 will be returned. If it is pressed up, a value of 1 will be returned. As it proceeds around the "compass" to northeast, east, southeast, south, etc., the value will range from 2 to 3 to 4 to 5, and so forth. If the FIRE button is also pressed, these numbers are increased by 128.

Line 17140: Change character color back to light gray.

Line 17150: If A is greater than 127, then the joystick button is pressed too. In that case, set FLAG to equal 1 to show that the joystick button is pressed. Then subtract 128 from A to reveal the actual direction, 0–8, that the stick is oriented toward. If A is not more than 127, then set FLAG to 0 to show that the joystick button is *not* pressed.

Line 17160: If FLAG equals 1, we want to show the user that the FIRE button has been pressed. In your programs you may have some action take place. A sound will make noise. Perhaps a

missile will be released. For this subroutine's demonstration, you'll only make the cursor flash red. So, print CHR$(42), an asterisk, to make the cursor "glimmer" a little. Then, because the cursor has moved one space to the right after printing the asterisk, move it back to its original position by printing CHR$(157), the CURSOR LEFT symbol. Next, print CHR$(150) to change the cursor to red. Notice that semicolons separate each character printed to make sure they are printed one after another on the same line. Of course, only the asterisk is an actual printable character, so the CURSOR LEFT character is required *only* after it.

Line 17170: P represents the horizontal position on the row where the cursor resides. If the joystick is oriented left or right, you need to change the value of P. You do this by taking P and adding to it the value stored in P(A). Notice from the data lines that P(A) has 0s, except in positions 3 and 7, which correspond to right and left. If JOY(1) returns a 3, then the joystick is pressed right and the value of P(3), which is +1, will be added to P. If JOY(1) returns a 7, then the joystick is pressed left and a –1 will be added to P. Any other orientation adds a 0 to P and leaves it in the same place.

Line 17180: Since you don't want to leave a trail behind the moving object, before printing an image of the cursor in the new location, you first need to erase it from its old location. That is done here, by printing a space, CHR$(32), followed by the CHR$(157) that moves it back to the now-empty old location.

Line 17190: You don't want the cursor to move beyond the limits of the screen. Here you check to see that P is not less than 1 or more than 79, which would take it to the limits of the 80-column screen. If it is, then reverse the move (which hasn't been implemented yet) by *subtracting* from P the value you have just added.

Line 17200: Variable B$ takes on the value stored in P$(A), which, like P(A) will have either a left- or right-moving value. You PRINT B$, which moves the cursor left or right (or leaves it in place, if A = 0), followed by the CURSR$.

Line 17210: The whole process repeats to account for the next move.

YOU SUPPLY

Just move joystick. ROW can be defined as the screen row on which the object moves. CURSR$ can be defined as any character you wish.

SUGGESTED ENHANCEMENTS: None.

RESULT

Object will move on screen under joystick control, left or right only.

80-COLUMN JOYSTICK—VERTICAL MOVEMENT

WHAT IT DOES: Moves object in north and south directions only.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                      *
120 REM * 80 COLUMN JOYSTICK *
130 REM * VERTICAL MOVEMENT  *
140 REM *                      *
150 REM **********************
160 REM --------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:    CURSOR CHARACTER
210 REM    COL:       COLUMN DESIRED
220 REM    JOYSTICK MOVEMENT
230 REM  RESULT --
240 REM    CURSOR MOVES UP AND DOWN
250 REM    IN DESIRED COLUMN.  FLASHES RED
260 REM    WHEN FIRE BUTTON IS PRESSED.
270 REM
280 REM --------------------------

290 REM *** INITIALIZE ***

300 COLUMN=5
310 CURSR$=CHR$(209)
320 GOTO 17310

17300 REM *** SUBROUTINE ***
```

```
17310 PRINT CHR$(154)
17320 P=1
17330 DATA -1,0,0,0,+1,0,0,0
17340 FOR N=1 TO 8
17350 READ P(N)
17360 NEXT N
17370 SCNCLR
17380 P$(1)=CHR$(145)
17390 P$(5)=CHR$(17)
17400 FOR N=1 TO COL:PRINT CHR$(29);:NEXT N
17410 GOTO 17430
17420 PRINT CHR$(157);
17430 A=JOY(1)
17440 PRINT CHR$(154);
17450 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
17460 IF FLAG=1 THEN PRINT CHR$(42);CHR$(157);CHR$(150);
17470 P=P+P(A)
17480 PRINT CHR$(32);CHR$(157);
17490 IF P<1 OR P>24 THEN P=P-P(A):GOTO 17430
17500 B$=P$(A)
17510 PRINT B$;CURSR$;
17520 GOTO 17420
```

HOW TO USE SUBROUTINE

This is the same as the last routine, but with up/down movement only. You may define the column you want the object to move in.

LINE-BY-LINE DESCRIPTION

This routine is basically identical to the last, with the following exceptions:

Line 300: COLUMN is defined, rather than ROW.

Line 17330: DATA elements are arranged to reflect up and down motion instead of side to side.

Lines 17380–17390: The up/down cursor movement characters are defined instead of the side-to-side characters.

Line 17490: The limits of movement are from 1 to 24, the height of the screen.

YOU SUPPLY

Joystick movement.

SUGGESTED ENHANCEMENTS: None.

RESULT

Object moves up and down only, under joystick control.

80-COLUMN JOYSTICK—ALL DIRECTIONS

WHAT IT DOES: Moves object diagonally, north, south, east, and west.

LEVEL: Advanced

```
100 REM ************************
110 REM *                        *
120 REM * 80 COLUMN JOYSTICK     *
130 REM *   -- ALL DIRECTIONS    *
140 REM *                        *
150 REM ************************
160 REM --------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:   CURSOR CHARACTER
210 REM  JOYSTICK MOVEMENT
220 REM  RESULT --
230 REM    CURSOR MOVES AROUND SCREEN
240 REM    FLASHES RED WHEN FIRE BUTTON
250 REM    IS PRESSED.
260 REM
270 REM --------------------------

17600 REM *** SUBROUTINE ***

17610 PRINT CHR$(154)
17620 CURSR$=CHR$(209)
17630 DATA -80,-79,+1,+81,+80,+79,-1,-81
17640 FOR N=1 TO 8
17650 READ P(N)
17660 NEXT N
17670 P=1
17680 SCNCLR
```

```
17690 P$(1)=CHR$(145)
17700 P$(2)=CHR$(145)+CHR$(29)
17710 P$(3)=CHR$(29)
17720 P$(4)=CHR$(17)+CHR$(29)
17730 P$(5)=CHR$(17)
17740 P$(6)=CHR$(17)+CHR$(157)
17750 P$(7)=CHR$(157)
17760 P$(8)=CHR$(157)+CHR$(145)
17770 GOTO 17790
17780 PRINT CHR$(157);
17790 A=JOY(1)
17800 PRINT CHR$(154);
17810 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
17820 IF FLAG=1 THEN PRINT CHR$(42);CHR$(157);CHR$(150);
17830 P=P+P(A)
17840 PRINT CHR$(32);CHR$(157);
17850 IF P<1 OR P>1920 THEN P=P-P(A):GOTO 17790
17860 B$=P$(A)
17870 PRINT B$;CURSR$;
17880 GOTO 17780
```

HOW TO USE SUBROUTINE

Games like PacMan (TM) and maze chases work best if the object can be moved only in a north, south, east, and west direction. Some programs need diagonal movement as well.

This subroutine will allow moving an object in any direction with a joystick. The movement can be adapted to many types of games as well as to other programs where input with a joystick is desirable. Call the subroutine every time you wish to check on the status of the joysticks.

LINE-BY-LINE DESCRIPTION

Key differences between the previous two routines will be pointed out. A main difference here is that P, instead of representing a position within a row or column, will represent an absolute screen location from 1 (the upper-left-hand corner of the screen) to 1920 (in the lower right of the next-to-last row).

Line 17630: Here in the DATA lines, values are stored for all eight possible nonzero numbers returned from JOY(1). For example, if the joystick is pressed up, then JOY(1) will equal 1, and P(1) equals –80. Since you are using an 80-column screen, subtracting 80 from P will move the object upward one whole row. Adding 80

to P will move it one whole row downward. Adding 81 would move it one row down and one place to the right, while 79 would move it down one row but one to the left (+80 plus –1 equals 79).

Lines 17690–17760: Here values are produced for P$(n) using CHR$ codes that correspond to the cursor movement you want to achieve. That is, CHR$(145)+CHR$(29) moves the cursor in a northeast direction. Note that P$(n) actually does the movement of the cursor; P only keeps track of where the cursor is on the screen for you so you can keep it from moving off the screen area.

Line 17850: Here is where you check on the position of the cursor to make sure it isn't outside the screen limits. Actually, the cursor would not move from the screen; however you would lose track of where it really was if you didn't impose this limit.

YOU SUPPLY

Simply move joysticks.

SUGGESTED ENCHANCEMENTS: None.

RESULT

Object will move on screen.

80-COLUMN JOYSTICK—COLOR DRAWING

WHAT IT DOES: Draws on screen in color.

LEVEL: Intermediate

```
100 REM ************************
110 REM *                      *
120 REM * 80 COLUMN JOYSTICK   *
130 REM *   COLOR DRAWING      *
140 REM *                      *
150 REM ************************
160 REM --------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:   CURSOR CHARACTER
210 REM  JOYSTICK MOVEMENT
220 REM  RESULT --
230 REM    CURSOR MOVES AROUND SCREEN
240 REM    FIRE BUTTON CHANGES COLORS
250 REM
260 REM --------------------------

17900 REM *** SUBROUTINE ***

17910 DIM C(16)
17920 PRINT CHR$(154)
17930 CURSR$=CHR$(209)
17940 DATA -80,-79,+1,+81,+80,+79,-1,-81
17950 DATA 149,150,151,152,153,154,155,156,158,
          159,144,5,28,30,31
17960 FOR N=1 TO 8
17970 READ P(N)
17980 NEXT N
17990 FOR N=1 TO 15
18000 READ C(N)
18010 NEXT N
18020 COLR=1
18030 PRINT CHR$(COLR);
18040 P=1
18050 SCNCLR
18060 P$(1)=CHR$(145)
18070 P$(2)=CHR$(145)+CHR$(29)
18080 P$(3)=CHR$(29)
18090 P$(4)=CHR$(17)+CHR$(29)
18100 P$(5)=CHR$(17)
18110 P$(6)=CHR$(17)+CHR$(157)
18120 P$(7)=CHR$(157)
18130 P$(8)=CHR$(157)+CHR$(145)
18140 GOTO 18160
18150 PRINT CHR$(157);
18160 A=JOY(1)
18170 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
18180 IF FLAG=1 THEN BEGIN
18190 COLR=COLR+1
18200 IF COLR=16 THEN COLR=1
18210 PRINT CHR$(C(COLR));
18220 FLAG=0
18230 BEND
18240 P=P+P(A)
18250 REM PRINT CHR$(32);CHR$(157);
18260 IF P<1 OR P>1920 THEN P=P-P(A):GOTO 18160
```

```
18270 IF P/80=INT(P/80) THEN P=P-P(A):GOTO 18160
18280 B$=P$(A)
18290 PRINT B$;CURSR$;
18300 GOTO 18150
```

HOW TO USE SUBROUTINE

Use the joystick to move the object around on the screen. Pressing the FIRE button changes the cursor color. You may cycle through all the colors available and return to the original.

LINE-BY-LINE DESCRIPTION

This routine is similar to the previous one, with the following additions:

Line 17910: This array stores the CHR$ codes for the individual colors available.

Line 17950: The CHR$ codes are listed here and read into the array by lines 17990–18010.

Line 18020: The initial color used is defined here.

Lines 18180–18230: When the FIRE button is pressed, COLR is incremented by 1. IF COLR reaches a value of 16, it is returned to 1, thus wrapping around through all available colors.

Line 18250: Note that the line that printed CHR$(32) to erase the character left behind has been deactivated with a REM statement. You may delete it entirely. Instead of moving the object on the screen, you also leave a trail of that object behind, thus drawing on the screen.

Line 18270: This is a new feature. In this line, the program checks to see if the cursor happens to be at the extreme edge of the screen. If so, the move is disallowed. This keeps the cursor from moving off the edge and wrapping around to the next line, which was allowed with the previous subroutines. You may or may not want this feature in your own program.

YOU SUPPLY

Move object under joystick control, changing colors and drawing on screen.

SUGGESTED ENHANCEMENTS: You can adapt this or any of the 80-column joystick routines to the 40-column screen by adjusting a few values. For example, the 80's in line 18270 should be changed to 40. The 1920 screen positions in line 18260 should be reduced to account for the fewer (40 × 24 or 40 × 25, depending on whether you want to leave the last line of the screen clear for text) printing positions on the 40-column screen.

RESULT

Images drawn on screen.

40-COLUMN JOYSTICK—HORIZONTAL MOVEMENT (Commodore 64 Compatible)

WHAT IT DOES: Moves object left or right on 40-column screen.

LEVEL: Advanced

```
100 REM ******************************
110 REM *                            *
120 REM *  40 COLUMN JOYSTICK        *
130 REM *  MOVE LEFT OR RIGHT        *
140 REM * (COMMODORE 64 COMPATIBLE)  *
150 REM *                            *
160 REM ******************************
170 REM ------------------------------
180 REM   ++ VARIABLES ++
190 REM  ROW:    ROW TO MOVE IN
200 REM  B:      BEGINNING OF THAT ROW
210 REM  E:      END OF THAT ROW
220 REM  B1:     POSITION OF CURSOR
230 REM  CURSR:  CURSOR CHARACTER
240 REM  CO:     CURSOR COLOR
250 REM  MOVE:   DIRECTION OF MOVE
260 REM  CH:     CHARACTER MEMORY
270 REM  DF:     CSCREEN-CH
280 REM  CSCREEN: COLOR MEMORY
290 REM
300 REM ------------------------------

310 REM *** INITIALIZE ***

320 DIM D(10)
330 DATA 0,0,0,-1,0,0,0,1,0,0
```

```
340 FOR X=1 TO 10
350 READ D(X)
360 NEXT X
370 PRINT CHR$(147)
380 CSCREEN=55296
390 CH=1024
400 ROW=5
410 B1=CH+ROW*40
420 E=B1+39
430 B=B1
440 DF=CSCREEN-CH
450 POKE 53281,1
460 CURSR=43:CO=2
470 GOTO 18470

18400 REM *** SUBROUTINE ***

18410 JV=PEEK(56320)
18420 J1=JVAND16
18430 F1=15-(JVAND15)
18440 MOVE=D(F1)
18450 RETURN

18460 REM *** MOVE CURSOR ***

18470 GOSUB 18410
18480 IF MOVE<>0 THEN POKE B1,32
18490 B1=B1+MOVE
18500 IF B1<B OR B1>E THEN B1=B1-MOVE
18510 POKE B1,CURSR
18520 POKE B1+DF,CO
18530 GOTO 18470
```

HOW TO USE SUBROUTINE

Previous subroutines have dealt with the 80-column Commodore 128 text screen. Many times your programs will be written only for the 40-column screen. If you want the program to be compatible with the broadest range of users, the 40-column screen is your best bet, since not all Commodore 128 owners will have the Commodore dual-mode monitor or another RGB monitor necessary for 80-column display. In addition, your program may want to manipulate objects on the 40-column bit-mapped screen for high-resolution graphics.

This subroutine was written for the 40-column text screen (a bit-map joystick routine is provided in the next chapter). It is *not* based on the techniques used for the previous subroutines; hints

have already been provided to let you convert those routines to 40-column use.

Instead, this subroutine was written without using the Commodore 128's JOY(n) commands. We've used only the PEEKing and POKing techniques that were previously necessary with the Commodore 64. As a result, the subroutines that follow are compatible with the Commodore 128 (which accepts most Commodore 64 commands) in either 128 mode or 64 mode. It is useful for those who are writing programs to be run on both machines with the 40-column screen. If you avoid using other BASIC 7.0 commands, your software can be used by either machine interchangeably.

LINE-BY-LINE DESCRIPTION

Lines 320–360: DIMension an array for the movement information, and load it into array D(n).

Line 370: Clear the screen (SCNCLR is not available in Commodore 64 mode).

Line 380: Define the position of color memory. This routine does not use cursor movement commands to position the object on the screen. Instead, you will POKE characters directly to character memory and screen memory. CSCREEN is the start of a 1000-position memory map that keeps track of the *color* of the character printed at that position.

Line 390: Define the position of the character memory. This is another 1000-position memory map. Each memory location stores what *character* is printed there.

Line 400: Define the row for movement.

Line 410: B1 will keep track of the position in character memory of the object. Here its initial value is calculated as the start of character memory, plus the row number times 40.

Line 420: E is the farthest right the object will be allowed to move, to the end of ROW.

Line 430: B is the farthest left the object will be allowed to move, defined as B1 here, since the value of B1 will change.

Line 440: DF is the difference between the two memory

addresses. You need then only keep track of one of them, POKE to that location, and then POKE to that value plus the difference between them to take care of the second map.

Line 450: This POKE changes the screen color.

Line 460: Define the cursor character as a cross-hair, and the initial color as red.

Line 18410: This short routine takes the place of the JOY(n) function in BASIC 7.0. JV captures the information found in the joystick register.

Line 18420: The Commodore computer uses several different bits within that byte to mean different things, so you must use Boolean logic (discussed more completely in Chapter 9) to extract the value of the relevant bit. J1 stores the value that indicates whether the FIRE button has been pressed.

Line 18430: F1 tells you the direction the joystick has been pressed.

Line 18440: MOVE will equal one of the values in the array D(n). Since you want only horizontal movement, all but two of the values will be 0, so you can have only +1 and –1 movement.

Line 18470: Here the joystick-reading subroutine is called each time through the loop.

Line 18480: If a new MOVE is indicated, POKE the old location of the cursor with a space to erase it.

Line 18490: Add MOVE to B1.

Line 18500: If this would take the cursor beyond the limits set up, nullify the move.

Line 18510: POKE B1 (character memory) with the cursor character.

Line 18520: POKE B1 + DF (color memory) with the color, CO.

Line 18530: Repeat.

YOU SUPPLY

Joystick movement and a definition for CURSR and CO if you wish.

RESULT

Object moves left/right on screen.

40-COLUMN JOYSTICK—VERTICAL MOVEMENT

WHAT IT DOES: Moves object up and down on screen.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                    *
120 REM * 40 COLUMN JOYSTICK *
130 REM * VERTICAL MOVEMENT  *
140 REM *                    *
150 REM **********************
160 REM --------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:   CURSOR CHARACTER
210 REM    COL:      COLUMN DESIRED
220 REM    JOYSTICK MOVEMENT
230 REM  RESULT --
240 REM    CURSOR MOVES UP AND DOWN
250 REM    IN DESIRED COLUMN.  FLASHES RED
260 REM    AND BEEPS WHEN FIRE PRESSED.
280 REM --------------------------

290 REM *** INITIALIZE ***

300 COLUMN=2
310 CURSR$=CHR$(161)
320 GOTO 18610

18600 REM *** SUBROUTINE ***

18610 PRINT CHR$(154)
18620 P=1
18630 DATA -1,0,0,0,+1,0,0,0
18640 FOR N=1 TO 8
18650 READ P(N)
18660 NEXT N
18670 SCNCLR
18680 P$(1)=CHR$(145)+CHR$(145)
18690 P$(5)=CHR$(17)+CHR$(17)
18700 FOR N=1 TO COL:PRINT CHR$(29);:NEXT N
18710 GOTO 18730
18720 PRINT CHR$(157);
18730 A=JOY(1)
18740 PRINT CHR$(154);
18750 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
18760 IF FLAG=1 THEN BEGIN
18770 PRINT CHR$(42);CHR$(157);CHR$(150);
18780 VOL 5
18790 SOUND 1,4096,10
18800 BEND
```

```
18810 P=P+P(A)
18820 PRINT CHR$(32);CHR$(157);
18830 IF P<1 OR P>24 THEN P=P-P(A):GOTO 18730
18840 B$=P$(A)
18850 PRINT B$;CURSR$;
18860 GOTO 18720
```

HOW TO USE SUBROUTINE

The Commodore 64-compatible routine just presented can be adapted for vertical movement if you change the data lines and make a few other modifications.

So, return to Commodore 128 mode with this routine for vertical movement on the 40-column screen. It is based on the 80-column vertical subroutine, with the addition of sound. A beep is heard when the joystick button is pressed.

LINE-BY-LINE DESCRIPTION

There are the following differences:

Line 18780: Turn volume to level 5.

Line 18790: Use voice 1 to make a sound at frequency 4096 for 10 "jiffies," or about one-sixth second.

YOU SUPPLY

Column to move in, cursor character, joystick movement.

RESULT

Object moves up and down on screen in designated column.

40-COLUMN JOYSTICK—MOVE ALL DIRECTIONS (Commodore 64 Compatible)

WHAT IT DOES: Moves object all directions.

LEVEL: Advanced

```
100 REM ****************************
110 REM *                          *
120 REM *  40 COLUMN JOYSTICK      *
130 REM *   MOVE ALL DIRECTIONS    *
140 REM * (COMMODORE 64 COMPATIBLE) *
150 REM *                          *
160 REM ****************************
170 REM ----------------------------
180 REM   ++ VARIABLES ++
190 REM  CSCREEN:  COLOR MEMORY
200 REM  CH:       CHARACTER MEMORY
210 REM  DF:       CSCREEN-CH
220 REM  B1:       POSITION OF CURSOR
230 REM  MOVE:     DIRECTION OF MOVE
240 REM  CURSR:    CURSOR CHARACTER
250 REM  CO:       CURSOR COLOR
260 REM
270 REM ----------------------------

280 REM *** INITIALIZE ***

290 DIM D(10)
300 DATA -40,40,0,-1,-41,39,0,1,-39,41
310 FOR X=1 TO 10
320 READ D(X)
330 NEXT X
340 PRINT CHR$(147)
350 CSCREEN=55296
360 CH=1024
370 E=CH+1000
380 B1=CH
390 DF=CSCREEN-CH
400 POKE 53281,1
410 CURSR=43:CO=2
420 GOTO 18870

18800 REM *** SUBROUTINE ***

18810 JV=PEEK(56320)
18820 J1=JVAND16
18830 F1=15-(JVAND15)
18840 MOVE=D(F1)
18850 RETURN

18860 REM *** MOVE CURSOR ***

18870 GOSUB 18810
18880 B1=B1+MOVE
18890 IF B1<B OR B1>E THEN B1=B1-MOVE
18900 POKE B1,CURSR
18910 POKE B1+DF,CO
18920 GOTO 18870
```

HOW TO USE SUBROUTINE

This is an enhancement of the 40-column Commodore 64-mode–compatible routine presented earlier. This one allows moving the object in vertical, horizontal, and diagonal directions.

LINE-BY-LINE DESCRIPTION

The changes are as follows:

Line 300: DATA is provided for other directions as well as horizontal.

Lines 370–380: The beginning and end of allowable movement are defined as the beginning and ending positions of the screen.

YOU SUPPLY

Joystick movement.

SUGGESTED ENHANCEMENTS: None.

RESULT

Object moves all directions under joystick control.

40-COLUMN JOYSTICK—COLOR DRAWING

WHAT IT DOES: Allows drawing on the screen in color, changing the cursor color and character.

LEVEL: Intermediate

```
100 REM ************************
110 REM *                      *
120 REM * 40 COLUMN JOYSTICK   *
130 REM *   COLOR DRAWING      *
140 REM *                      *
150 REM ************************
160 REM ---------------------------
170 REM      ++ VARIABLES ++
180 REM
190 REM  USER SUPPLIES --
200 REM    CURSR$:   CURSOR CHARACTER
210 REM    JOYSTICK MOVEMENT
220 REM  RESULT --
230 REM    CURSOR MOVES AROUND SCREEN
240 REM    FIRE BUTTON CHANGES COLORS
250 REM    AND CURSOR CHARACTER
260 REM
270 REM ---------------------------

280 REM *** INITIALIZE ***

290 DIM C(16)
300 PRINT CHR$(154)
310 CURSR=209
320 CURSR$=CHR$(CURSR)
330 DATA -40,-39,+1,+41,+40,+39,-1,-41
340 DATA 149,150,151,152,153,154,155,156,158,159,144,5,28,30,31
350 FOR N=1 TO 8
360 READ P(N)
370 NEXT N
380 FOR N=1 TO 15
390 READ C(N)
400 NEXT N
410 COLR=1
420 PRINT CHR$(COLR);
430 P=1
440 SCNCLR
450 P$(1)=CHR$(145)
460 P$(2)=CHR$(145)+CHR$(29)
470 P$(3)=CHR$(29)
480 P$(4)=CHR$(17)+CHR$(29)
490 P$(5)=CHR$(17)
500 P$(6)=CHR$(17)+CHR$(157)
510 P$(7)=CHR$(157)
520 P$(8)=CHR$(157)+CHR$(145)
530 GOTO 19310

19290 REM *** SUBROUTINE ***

19300 PRINT CHR$(157);
19310 A=JOY(1)
19320 IF A=129 THEN BEGIN
19330 CURSR=CURSR+1
19340 IF CURSR>255 THEN CURSR=255
19350 CURSR$=CHR$(CURSR)
19360 BEND:GOTO 19310
19370 IF A=133 THEN BEGIN
```

```
19380 CURSR=CURSR-1
19390 IF CURSR<32 THEN CURSR=32
19400 CURSR$=CHR$(CURSR)
19410 BEND:GOTO 19310
19420 IF A>127 THEN FLAG=1:A=A-128:ELSE FLAG=0
19430 IF FLAG=1 THEN BEGIN
19440 COLR=COLR+1
19450 IF COLR=16 THEN COLR=1
19460 PRINT CHR$(C(COLR));
19470 FLAG=0
19480 BEND
19490 P=P+P(A)
19500 REM PRINT CHR$(32);CHR$(157);
19510 IF P<1 OR P>1000 THEN P=P-P(A):GOTO 19310
19520 IF P/40=INT(P/40) THEN P=P-P(A):GOTO 19310
19530 B$=P$(A)
19540 PRINT B$;CURSR$;
19550 SOUND 1,3000,2
19560 GOTO 19300
```

HOW TO USE SUBROUTINE

This routine lets you draw on the screen in color, with the added enhancement of letting you change the cursor color *and* the cursor character with the FIRE button.

To cycle through the available colors, press the joystick button while the stick itself is centered in the neutral position. When the joystick is pressed upward while the FIRE button is pressed, the cursor character will be changed to the next higher ASCII value instead.

LINE-BY-LINE DESCRIPTION

The operation of this drawing subroutine is similar to previous examples. However, the following additions have been made:

Line 19370: If A equals 133, then the FIRE button was pressed while the joystick was oriented upward. So, begin the following loop:

Line 19380: Reduce CURSR character to 1 less than the current value.

Line 19390: However, don't let it become less than CHR$(32), a space. A space will cause the cursor to print blanks along the

screen. Instead change to 127. Note that with this routine, as written, only characters from 32 to 127 may be used.

Line 19400: Change cursor character to CHR$(CURSR).

Line 19410: Loop.

YOU SUPPLY

Joystick movement, and changes of color and cursor.

SUGGESTED ENHANCEMENTS: Change routine to allow for all character graphics, but still bypass values that change colors and do other tests.

RESULT

Color drawing on the screen.

40-COLUMN JOYSTICKS—FOR TWO JOYSTICKS (Commodore 64 Compatible)

WHAT IT DOES: Allows moving two objects on the screen at once, with two joysticks.

LEVEL: Intermediate

```
100 REM ******************************
110 REM *                            *
120 REM *   40 COLUMN JOYSTICKS      *
130 REM *    FOR TWO JOYSTICKS       *
140 REM * (COMMODORE 64 COMPATIBLE) *
150 REM *                            *
160 REM ******************************
170 REM      ++ VARIABLES ++
180 REM    F1:   STATUS OF FIRE BUTTON 1
190 REM    F2:   STATUS OF FIRE BUTTON 2
200 REM    M1:   MOVE FOR JOY 1
210 REM    M2:   MOVE FOR JOY 2
220 REM    B1:   POSITION FOR OBJECT 1
230 REM    B2    POSITION FOR OBJECT 2
240 REM    CH:   START OF CHARACTER MEMORY
250 REM    E:  END OF CHARACTER MEMORY
```

```
260 REM     CSCREEN: START OF COLOR MEMORY
270 REM
280 REM ---------------------------

290 REM *** INITIALIZE ***

300 CSCREEN=55296
310 CHAR=1024
320 B1=CH
330 B2=CH
340 B=CH
350 E=CH+1000
360 DF=CSCREEN-CH
370 GOTO 19170

19000 REM *** SUBROUTINE ***

19010 J1=PEEK(56320)
19020 J2=PEEK(56321)
19030 F1=J1 AND 16
19040 F2=J2 AND 16
19050 J1=15-(J1 AND 15)
19060 J2=15-(J2 AND 15)
19070 IF J1=1 THEN M1=-40:GOTO 19110
19080 IF J1=2 THEN M1=40:GOTO 19110
19090 IF J1=4 THEN M1=-1:GOTO 19110
19100 IF J1=8 THEN M1=1:GOTO 19110
19110 IF J2=1 THEN M2=-40:RETURN
19120 IF J2=2 THEN M2=40:RETURN
19130 IF J2=4 THEN M2=-1:RETURN
19140 IF J2=8 THEN M2=1:RETURN
19150 RETURN

19160 REM *** MOVE CURSOR ***

19170 PRINT CHR$(147)
19180 GOSUB 19010
19190 IF F1=0 THEN PRINT CHR$(147)
19200 IF F2=0 THEN PRINT CHR$(147)
19210 B1=B1+M1:IF B1<B OR B1>E THEN B1=B1-M1
19220 B2=B2+M2:IF B2<B OR B2>E THEN B2=B2-M2
19230 POKE B1,81
19240 POKE B1+DF,3
19250 POKE B2,81
19260 POKE B2+DF,5
19270 GOTO 19180
```

HOW TO USE SUBROUTINE

Many programs with two players will use both joysticks at once. This routine, an adaptation of the previous Commodore 64-mode–compatible module, allows moving two objects under independent control.

LINE-BY-LINE DESCRIPTION

The following changes have been made:

Line 19020: Instead of just PEEKing the status of J1, also PEEK a value for J2, and calculate the orientation of the second joystick in the same manner.

Lines 19110–19140: The direction of the move is calculated for J2 as well as J1. Note that this variation allows *only* movement in north/south/east/west directions, which may be useful for maze-type games where diagonal movement is undesirable.

Lines 19190–19220: In this variation, pressing either FIRE button causes the screen to be cleared.

YOU SUPPLY

Movement from two joysticks.

SUGGESTED ENHANCEMENTS: Change to allow movement in any direction, if desired.

RESULT

Objects move on screen.

KEYBOARD JOYSTICK

WHAT IT DOES: Allows moving object on screen using keyboard cursor keys, without a joystick.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                    *
120 REM *     40 COLUMN      *
130 REM * KEYBOARD JOYSTICK *
140 REM *                    *
150 REM **********************
160 REM ----------------------------
170 REM        ++ VARIABLES ++
180 REM  SUPPLIED BY USER --
190 REM    CURSR$:  CURSOR CHARACTER
200 REM  RESULT --
210 REM    CURSOR MOVES AROUND SCREEN
220 REM    UNDER CURSOR KEY CONTROL
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 CURS$=CHR$(209)
270 SCNCLR

19600 REM *** SUBROUTINE ***

19610 GETKEY A$
19620 IF A$=CHR$(29) OR A$=CHR$(17) OR A$=CHR$(157) OR
A$=CHR$(145) GOTO 19630:ELSE GOTO 19610
19630 PRINT CHR$(32);CHR$(157);
19640 PRINT A$;CURSR$;CHR$(157);
19650 GOTO 19610
```

HOW TO USE SUBROUTINE

Programs can be written to allow moving objects on the screen using only the cursor keys. This is especially practical for the Commodore 128, since four separate cursor keys are provided, one for each major compass direction. This subroutine simulates a joystick using those keys.

LINE-BY-LINE DESCRIPTION

Line 260: Define cursor character as CHR$(209).

Line 270: Clear screen.

Line 19610: Get a key from keyboard.

Line 19620: If key pressed was a cursor key, proceed to next line, otherwise ignore and go back for more keyboard input. You could also include here another value, such as the space bar, which you would designate as the representation of the FIRE button.

Line 19630: Erase old character and backspace.

Line 19640: Print the cursor-movement key pressed, followed by the cursor character, and then backspace.

Line 19650: Loop and repeat.

YOU SUPPLY

Cursor character definition.

SUGGESTED ENHANCEMENTS: Incorporate some action to be followed when the "fire" button (space bar or whatever you designate) is pressed.

RESULT

Object will move on screen under cursor key control, in north, south, east, or west directions.

KEYBOARD DRAWING

WHAT IT DOES: Draws on screen, changing cursor character and color as desired, using cursor keys for control.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                    *
120 REM *     40 COLUMN      *
130 REM * KEYBOARD DRAWING  *
140 REM *                    *
150 REM ********************
160 REM ----------------------------
170 REM         ++ VARIABLES ++
180 REM  SUPPLIED BY USER --
190 REM    CURSR$:  CURSOR CHARACTER
200 REM    BORDER COLOR
210 REM    CHARACTER COLOR
220 REM    SCREEN COLOR
230 REM  RESULT --
240 REM    CURSOR DRAWS ON SCREEN.
250 REM    CURSOR AND COLOR CAN BE
260 REM    CHANGED FROM KEYBOARD
270 REM
280 REM ----------------------------
```

```
290 REM *** INITIALIZE ***

300 DARK=0:WHITE=1
310 RED=2:CYAN=3
320 PURPLE=4:GREEN=5
330 BLUE=6:YELLOW=7
340 PUMPKIN=8:BROWN=9
350 LRED=10:DGRAY=11
360 MGRAY=12:LGREEN=13
370 LBLUE=14:LIGHT=15
380 BACKGROUND=53281
390 SURROUND=53280
400 POKE BACKGROUND,PUMPKIN
410 POKE SURROUND,MGRAY
420 TRAIL$=CHR$(209)
430 SCNCLR

19700 REM *** SUBROUTINE ***

19710 GETKEY A$
19720 A=ASC(A$)
19730 IF A=29 THEN CURSR$=CHR$(171):GOTO 19850
19740 IF A=157 THEN CURSR$=CHR$(179):GOTO 19850
19750 IF A=17 THEN CURSR$=CHR$(178):GOTO 19850
19760 IF A=145 THEN CURSR$=CHR$(177):GOTO 19850
19770 IF A>154 AND A<160 GOTO 19840
19780 IF A>148 AND A<156 GOTO 19840
19790 IF A>27 AND A<32 GOTO 19840
19800 IF A=144 OR A=5 OR A=129 GOTO 19840
19810 IF A=142 OR A=141 OR A=146 OR A=13 OR A=18 OR A=19 OR A=20
GOTO 19710
19820 TRAIL$=A$
19830 GOTO 19710
19840 PRINT A$;
19850 PRINT TRAIL$;CHR$(157);
19860 PRINT A$;CURSR$;CHR$(157);
19870 GOTO 19710
```

HOW TO USE SUBROUTINE

This routine carries the last module one step farther, providing drawing on the screen. The cursor character and color can be changed from the keyboard.

As you know, the number keys at the top of the keyboard can be used in combination with the control key and the Commodore key to change the colors of the characters being printed. There are also CHR$ equivalents of those key combinations that can be used. In this book we haven't yet explained the use of available POKEs to specific memory registers to change the color of screen background and border.

The memory location for the background screen color is 53281, while the location for the border is 53280. You can POKE the color number 0–15 to these registers to make the color changes you want.

However, it can be difficult to remember that 0 equals black and 6 equals blue. One easy way to determine the first half of the color spectrum is to subtract 1 from the key number where that color resides. However, for the upper eight colors you must add 7 instead. Why not simply define the color values as variables that have some relation to the color name?

That's what is done in this program.

LINE-BY-LINE DESCRIPTION

Lines 300–370: Define numbers 0 through 15 to equal color names of the available hues. Note that BLACK and BLUE would be the same color name to the Commodore 128, since only BL would be significant. Therefore, use DARK for black, instead. Along the same lines, GRAY and GREEN would be the same, so LIGHT is used for the former. ORANGE causes a problem, since ORANGE contains the reserved word OR. Use PUMPKIN instead, following the precedent of naming the color after an edible.

Lines 380–390: Define BACKGROUND and SURROUND as the memory locations for the screen and border colors.

Lines 400–410: POKE the background orange, and the border medium gray.

Line 420: As a change of pace, make the trail left by the cursor be different from the cursor character itself. Define TRAIL$ as CHR$(209).

Line 430: Clear the screen.

Line 19710: Get a key from keyboard.

Line 19720: Find its ASCII value.

Lines 19730–19760: Depending on the direction of movement, the cursor character is changed to a little graphics symbol pointing in the direction of movement.

Lines 19770–19810: Check to see if key pressed was an allowable one. If so, go print it.

Line 19820: If key was not a color key, change TRAIL$ to that key.

Line 19830: Return for more keyboard input.

Lines 19840–19870: Print the TRAIL$ character, backspace, and return.

YOU SUPPLY

Keyboard input to draw, change color, and cursor character. You may also redefine screen and border color.

SUGGESTED ENHANCEMENTS: None.

RESULT

Drawing on screen under keyboard control.

PADDLES

WHAT IT DOES: Reads four paddles to move objects on screen.

LEVEL: Intermediate

```
100 REM ************
110 REM *           *
120 REM *  PADDLES  *
130 REM *           *
140 REM ************
150 REM -------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    PADDLE MOTION
190 REM  RESULT --
200 REM    LETTERS A,B,C,D
210 REM    MOVE ACROSS SCREEN
220 REM    IN RESPONSE TO PADDLES
230 REM
240 REM -------------------------------

19900 REM *** SUBROUTINE ***

19910 SCNCLR
19920 A=POT(1)
19930 B=POT(2)
19940 C=POT(3)
19950 D=POT(4)
```

```
19960 IF A>255 THEN AFLAG=1:A=((A/255)-INT(A/255))*255
:ELSE AFLAG=0
19970 IF B>255 THEN BFLAG=1:B=((B/255)-INT(B/255))*255
:ELSE BFLAG=0
19980 IF C>255 THEN CFLAG=1:C=((C/255)-INT(C/255))*255
:ELSE CFLAG=0
19990 IF D>255 THEN DFLAG=1:D=((D/255)-INT(D/255))*255
:ELSE DFLAG=0
20000 A1=((1+INT(A/6.4))-40)*-1
20010 B1=((1+INT(B/6.4))-40)*-1
20020 C1=((1+INT(C/6.4))-40)*-1
20030 D1=((1+INT(D/6.4))-40)*-1
20040 PRINT TAB(LD);CHR$(32);CHR$(19);
20050 PRINT TAB(D1)"D";CHR$(19);
20060 LD=D1
20070 PRINT TAB(LA);CHR$(32);CHR$(19);
20080 PRINT TAB(A1)"A";CHR$(19);
20090 LA=A1
20100 PRINT TAB(LB);CHR$(32);CHR$(19);
20110 PRINT TAB(B1)"B";CHR$(19);
20120 LB=B1
20130 PRINT TAB(LC);CHR$(32);CHR$(19);
20140 PRINT TAB(C1)"C";CHR$(19);
20150 LC=C1
20160 GOTO 19920
```

HOW TO USE SUBROUTINE

Paddle reading from BASIC is less than satisfactory for a number of reasons. Commodore recommends against it, claiming that the technique is unreliable. In any case, the usefulness of the value returned by POT(n) is reduced by the fact that the full "travel" of the paddle is not used. Depending on what paddle you use, you may find that a half or quarter turn will be sufficient to cover the full range from 0 to 255. Therefore, precise movements may be difficult. This subroutine is included to show that paddle reading can be done.

Paddles are unlike joysticks in that they return a continuous value that corresponds to how far the paddle has been returned. The nonanalog joysticks used by the Commodore 128, on the other hand, merely tell us which way the joystick is oriented. Paddles contain a potentiometer, similar to the volume control on your stereo, that provides a reading of the variable resistance produced by your turning the paddle. While this reading is continuous, the Commodore 128 translates it into a series of whole numbers from 0 to 255.

To use paddles effectively in this program, you need to translate that value into a position on the screen. For horizontal movement on a 40-column screen, you would want 0 to equal column 1, and a value of 255 to equal column 40. That is approximately what is done with this subroutine. Values for all four paddles are produced, so you can move four objects. Note that movement is possible only in an east/west or north/south direction, although a clever programmer could write a routine to move the object back and forth diagonally as well.

LINE-BY-LINE DESCRIPTION

Line 19910: Clear screen.

Lines 19920–19950: Obtain orientation of paddles A–D.

Lines 19960–19990: If a paddle value is greater than 255, then the FIRE button has been pressed. These lines set the fire flag for each paddle and then return the value of A to an equivalent value less than 255.

Lines 20000–20030: Calculate position on screen.

Lines 20040–20150: TAB to position indicated by paddle, print space to erase old character, then print letter corresponding to the paddle in the new position. Lx (LA, LB, LC, or LD) equals the last position of the character, while x1 (A1, B1, C1, or D1) equals the current position.

YOU SUPPLY

Paddle movement.

SUGGESTED ENHANCEMENTS: Create some action to result when the FIRE button is pressed.

RESULT

Four objects move on screen under paddle control.

RANDOM INTEGER

WHAT IT DOES: Provides a random integer within a number range specified.

LEVEL: Novice

```
100 REM ******************
110 REM *                  *
120 REM * RANDOM INTEGER *
130 REM *                  *
140 REM ******************
150 REM ------------------------
160 REM     ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    HIGH:    TOP OF RANGE
190 REM    MINIMUM: BOTTOM OF RANGE
200 REM  RESULT --
210 REM    NU:      NUMBER CHOSEN
220 REM
230 REM ------------------------

240 REM *** INITIALIZE ***

250 HIGH=100
260 MINIMUM=15
270 GOSUB 20210
280 PRINT "RANDOM INTEGER :";NU
290 END

20200 REM *** SUBROUTINE ***

20210 DF=HIGH-MINIMUM+1
20220 NU=INT(RND(0)*DF)+MINIMUM
20230 RETURN
```

HOW TO USE SUBROUTINE

What makes a game a game and not a test? Randomness is one element found in many, but not all, games. Random numbers selected by the computer determine the changes in some games that the player must contend with. Lacking randomness, a game is either a test of memory or a contest of strategy. A little of all three elements makes for a good game, and this subroutine lets you get greater control over randomness than with unadorned BASIC 7.0.

The Commodore 128 can choose pseudorandom numbers. That is, although they appear to be random, the numbers are actually drawn from a long list. Even though the sequence is the same each time, the list of numbers is very long and the starting position is usually different, so the numbers appear to be random to the player.

Some BASICs allow choosing a random number larger than one but smaller than another integer with the simple command RND(N) where N is the upper limit. RND(7) would produce numbers from 1 to 7, for example. The Commodore 128, on the other hand, generates random numbers larger than 0 and smaller than 1, so we might get .74329 or .15832 or some other value. To get the numbers in a given range 1 to N, we must multiply the random number by N and add 1. That is, INT(RND(0)*7)+1 will produce numbers larger than 1 and no larger than 7.

But what if some other range is desired, such as the numbers between 43 and 198. This subroutine will pluck them out of randomland for you. From the user-supplied minimum and maximum numbers, it will select random integers only in the desired range.

LINE-BY-LINE DESCRIPTION

Line 250: Define upper limit as 100.

Line 260: Define lower limit as 15.

Lines 270–290: Access the subroutine and print results.

Line 20210: Calculate the difference between the upper and lower limits.

Line 20220: Generate a random number in the range 0 to the difference between the limits, and then add the MINIMUM value to that to ensure that the number is at least the minimum but no more than the maximum.

YOU SUPPLY

Upper and lower limits.

SUGGESTED ENHANCEMENTS: Figure a way to allow generating real numbers—numbers with both an integer and fractional part.

RESULT

Random integer generated in the desired range.

SAMPLE VALUES: 65, 33, 98, 16

RANDOM SETS

WHAT IT DOES: Chooses a set of random numbers.

LEVEL: Novice

```
100 REM **************
110 REM *              *
120 REM * RANDOM SET   *
130 REM *              *
140 REM **************
150 REM -------------------------------
160 REM      ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM    HIGH:         TOP OF RANGE
190 REM    MINIMUM:      BOTTOM OF RANGE
200 REM    NUMBER:       SIZE OF SET
210 REM   RESULT --
220 REM    SET(NUMBER): RANDOM INTEGERS
230 REM
240 REM -------------------------------

250 REM *** INITIALIZE ***

260 NUMBER=10
270 DIM SET(NUMBER)
280 HIGH=100
290 MINIMUM=15
300 GOSUB 20310
310 FOR N=1 TO NUMBER
320 PRINT "RANDOM INTEGER #";N;" : ";SET(N)
330 NEXT N
340 END

20300 REM *** SUBROUTINE ***

20310 DF=HIGH-MINIMUM+1
```

```
20320 FOR N=1 TO NUMBER
20330 RN=INT(RND(0)*DF)+MINIMUM
20340 FOR N1=1 TO N
20350 IF RN=SET(N1) GOTO 20330
20360 NEXT N1
20370 SET(N)=RN
20380 NEXT N
20390 RETURN
```

HOW TO USE SUBROUTINE

Intended for the beginning programmer, this routine merely demonstrates how a group of random numbers can be generated quickly with one call to a subroutine. Each number generated is different from all others in the set.

LINE-BY-LINE DESCRIPTION

Line 260: Define number of random integers wanted.
Line 270: DIMension an array to store those integers.
Lines 280–290: Define upper and lower limits.
Lines 300–340: Access subroutine and print results.
Line 20310: Determine the difference between the limits.
Line 20320: Start loop from 1 to NUMBER to produce desired quantity of random integers.
Line 20330: Generate random number in the desired range.
Lines 20340–20360: Check previous random numbers generated to see if latest one is a duplicate. If so, return to Line 20330 and try again.
Line 20370: Otherwise, store the number in SET(n).

YOU SUPPLY

Upper and lower limit, and number of random integers wanted.

SUGGESTED ENHANCEMENTS: Install an error trap to make sure the user does not request more unique random numbers than are available in the range specified. For example, if NUMBER = 20, and HIGH = 30 while MINIMUM = 15, then the subroutine will hang.

RESULT

Set of random integers generated, each unique.

SAMPLE VALUES: 45, 98, 32, 16, 84, 21, 99, 33, 46, 75

ANIMATED COIN FLIP

WHAT IT DOES: Produces image of coin flipping, plus heads or tails value.

LEVEL: Novice/Intermediate

```
100 REM *************
110 REM *           *
120 REM * COIN FLIP *
130 REM *           *
140 REM *************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NONE
190 REM  RESULT --
200 REM    GRAPHIC COIN FLIP
210 REM    FLIP$:  HEADS OR TAILS
220 REM
230 REM ------------------------------

240 REM *** INITIALIZE ***

250 GOSUB 20410
260 END

20400 REM *** SUBROUTINE ***

20410 SCNCLR
20420 DATA 119,113,99
20430 FOR N=1 TO 3
20440 READ CN(N)
20450 NEXT N
20460 PRINT:PRINT:PRINT:PRINT:PRINT:PRINT:PRINT
20470 FOR N2=1 TO 7
20480 PRINT CHR$(32);CHR$(157);
20490 PRINT CHR$(145);
20500 PRINT TAB(10);"";
20510 FOR N=1 TO 3
20520 PRINT CHR$(CN(N));
20530 PRINT CHR$(157);
20540 FOR N1=1 TO 3:NEXT N1
```

```
20550 NEXT N
20560 NEXT N2
20570 FOR N3=1 TO 2
20580 FOR N=1 TO 3
20590 PRINT CHR$(CN(N));
20600 PRINT CHR$(157);
20610 FOR N1=1 TO 3:NEXT N1
20620 NEXT N
20630 NEXT N3
20640 FOR N2=1 TO 7
20650 PRINT CHR$(32);CHR$(157);
20660 PRINT CHR$(17);
20670 FOR N1=1 TO 3:NEXT N1
20680 FOR N=1 TO 3
20690 PRINT CHR$(CN(N));
20700 PRINT CHR$(157);
20710 FOR N1=1 TO 3:NEXT N1
20720 NEXT N
20730 NEXT N2
20740 FLIP=INT(RND(0)*2)+1
20750 PRINT CHR$(17);CHR$(157);
20760 IF FLIP=1 THEN FLIP$="HEADS":ELSE FLIP$="TAILS"
20770 PRINT FLIP$
20780 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will produce a simulation of a coin being tossed in the air, followed by a notice of whether it fell heads or tails. More advanced programmers can change the location of the coin flip to suit their own software.

This subroutine is useful for some beginner-level statistical experiments, and a few games. You might want to construct a loop that flips the coin 1000 times and adds up the number of heads and tails to check the randomness of your computer.

LINE-BY-LINE DESCRIPTION

Line 250: Access the subroutine.

Line 20410: Clear screen.

Line 20420: DATA for graphics characters representing coin image. These are an open circle, a filled-in circle, and a straight line to indicate the heads, tails, and edge of a spinning coin.

Lines 20430–20450: Read the data into array CN(N).

Line 20460: Move the cursor down seven lines. You can

substitute one of the cursor movement techniques used in this book, such as Cursor Mover (Chapter 8).

Line 20470: Start loop from 1 to 7. Each increment of the loop will move the coin up the screen one position.

Line 20480: Print a space and backspace to erase old image of coin.

Line 20490: Move cursor up one line.

Line 20500: Start loop from 1 to 3. Each of the three coin images, heads, tails, and edge, will be displayed at each position the coin takes on its flip upwards.

Line 20510: TAB out to coin position.

Line 20520: Print coin image N.

Line 20530: Backspace.

Line 20540: Delay slightly.

Line 20550: Next coin image.

Line 20560: Next position of the coin.

Lines 20570–20630: Spin the coin in the air at the top of its arc.

Lines 20640–20730: Show the coin falling, basically the reverse of the routine used to show it rising.

Line 20740: Produce a random value for coin flip.

Line 20750: Half the time will be heads, the other half tails.

Line 20760: Print heads or tails.

YOU SUPPLY

No user input needed.

SUGGESTED ENHANCEMENTS: None.

RESULT

Coin flipped.

SAMPLE VALUES: HEADS, TAILS

N-SIDED DICE

WHAT IT DOES: Rolls variable number of dice, with variable number of sides.

LEVEL: Intermediate

```
100 REM ****************
110 REM *                *
120 REM * N-SIDED DICE *
130 REM *                *
140 REM ****************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    SIDES: NUMBER OF SIDES
190 REM    DICE:  NUMBER OF DICE
200 REM  RESULT --
210 REM    GRAPHIC DICE ROLL
220 REM    TT:     TOTAL OF ROLL
230 REM    D(N):   INDIVIDUAL DICE
240 REM
250 REM ------------------------------

260 REM *** INITIALIZE ***

270 SIDES=6
280 DICE=2
290 SCNCLR
300 DATA 108,111,112,186
310 PRINT:PRINT:PRINT
320 FOR N=1 TO 4
330 READ DICE(N)
340 NEXT N
350 GOSUB 20810
360 END

20800 REM *** SUBROUTINE ***

20810 FOR N3=1 TO DICE
20820 FOR N2=1 TO 3
20830 FOR N=1 TO 4
20840 PRINT CHR$(32);CHR$(29);CHR$(DICE(N));CHR$(157);
20850 FOR N1=1 TO 3:NEXT N1
20860 NEXT N
20870 NEXT N2
20880 PRINT CHR$(166);"  ";
20890 D(N3)=INT(RND(0)*SIDES)+1
20900 PRINT D(N3)
20910 PRINT:PRINT:PRINT
20920 NEXT N3
20930 FOR N4=1 TO DICE
20940 TT=TT+D(N4)
```

```
20950 NEXT N4
20960 PRINT "TOTAL :";TT
20970 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will roll as many dice as you specify, with the number of sides you request. Dungeons and Dragons players in particular can specify how many sides each die will have. You might want to add an input routine that will request the number of sides prior to a roll, as well as the number of dice to be rolled. All the dice in the set must have the same number of sides.

LINE-BY-LINE DESCRIPTION

Line 270: Define number of sides.

Line 280: Define number of dice.

Line 290: Clear screen.

Lines 300–340: Read into data CHR$ codes for character graphics that will be used to represent rolling and resting dice.

Line 350: Access the subroutine.

Line 20810: Begin loop from 1 to number of dice to be rolled.

Lines 20820–20870: Print image of dice rolling across screen. Final resting image of die is of generic die only and does not have the same number of spots as the actual die rolled. That number is printed to the screen.

Line 20890: Select the value rolled.

Line 20900: Print the value to the screen.

Line 20910: Move down three lines to allow for next die to be rolled.

Line 20920: Repeat the next die.

Lines 20930–20960: Add up the total of the dice thrown.

Line 20970: Return.

YOU SUPPLY

Number of sides and dice wanted.

SUGGESTED ENHANCEMENTS: Produce a graphic image of each die rolled that will have the correct number of spots. Allow users to enter how many dice and how many sides prior to each roll. Allow having each die include a different number of sides, so a mixed set of dice can be rolled.

RESULT

N-sided dice rolled.

DEAL CARDS

WHAT IT DOES: Deals a deck of cards.

LEVEL: Intermediate

```
100 REM **************
110 REM *              *
120 REM * DEAL CARDS *
130 REM *              *
140 REM **************
150 REM -------------------------
160 REM    ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM   NC:          NUMBER CARDS
190 REM  RESULT --
200 REM   DECK$(N):  DECK
210 REM   CARD$:     CARD DRAWN
220 REM   DRAW:      RANDOM CARD
230 REM
240 REM -------------------------

250 REM *** INITIALIZE ***

260 DIM DECK$(52)
270 DATA 193,211,218,216

280 REM *** READ SUITS ***

290 FOR N=1 TO 4
300 READ A
310 SUIT$(N)=CHR$(A)
320 NEXT N

330 REM *** ASSEMBLE DECK ***

340 FOR SUIT=1 TO 4
350 CU=CU+1
```

```
360 DECK$(CU)="ACE OF "+SUIT$(SUIT)
370 CU=CU+1
380 DECK$(CU)="KING OF "+SUIT$(SUIT)
390 CU=CU+1
400 DECK$(CU)="QUEEN OF "+SUIT$(SUIT)
410 CU=CU+1
420 DECK$(CU)="JACK OF "+SUIT$(SUIT)
430 FOR N=2 TO 10
440 CU=CU+1
450 DECK$(CU)=STR$(N)+" OF "+SUIT$(SUIT)
460 NEXT N
470 NEXT SUIT
480 NC=52
490 GOSUB 21010
500 PRINT CARD$
510 END

21000 REM *** SUBROUTINE ***

21010 IF NC<>0 GOTO 21050
21020 CARD$=""
21030 PRINT"DECK GONE!!"
21040 RETURN
21050 DRAW=INT(RND(1)*NC)+1
21060 CARD$=DECK$(DRAW)
21070 DECK$(DRAW)=DECK$(NC)
21080 NC=NC-1
21090 RETURN
```

HOW TO USE SUBROUTINE

Many game programs require dealing a deck of cards. Your own programs may simulate drawing from a randomly shuffled deck simply by calling this subroutine. The deck has already been assembled (Lines 210–360) using the Commodore 128 graphics characters for suits, and the numbers of words for the value of the cards.

If you need to determine the rank of the card for your program, all cards through the 10 may be ascertained by a line such as: "V = VAL(CARD$)".

IF V = 0 then four more lines are needed, such as, "IF LEFT$(CARD$,1) = "J" THEN V = 11" or "IF LEFT$(CARD$,1) = "Q" THEN V = 12".

This is a very fast shuffling routine, which requires only 52 tries to deal 52 cards. Some slower algorithms (a formula for performing a task or computing a result) may repeatedly access "empty" deck positions when looking for the remaining cards.

LINE-BY-LINE DESCRIPTION

Lines 260–270: DIMension a deck, with room for 52 cards, and the DATA lines for the character graphics representing the suits.

Lines 290–320: The suit characters are read into the array SUIT$(n).

Line 340: Start assembling an unshuffled deck of cards, repeating four times, once for each suit.

Line 350: Counter CU keeps track of which card has been assembled so far.

Lines 360–420: Add the face cards and ace to the deck.

Lines 430–470: Assemble the other cards, deuce through 10, and repeat.

Line 480: Set initial number of cards remaining in the deck at 52.

Line 490: Access the subroutine.

Line 500: Print the card selected.

Line 21010: Check to see if any cards are left in the deck; if not, go to card-drawing routine.

Line 21020: Otherwise, print DECK GONE.

Lines 21050–21060: The computer selects a number between 1 and NC (52 this time), and that element of DECK$(n) becomes the card drawn.

Line 21070: This leaves a "hole" in the deck at position DRAW. You fill it up by taking the last card in the deck, which is DECK$(NC), and placing it in DECK$(DRAW).

Line 21080: This leaves the "hole" at the end, but you then change NC to equal NC–1, so the computer will draw only from the elements 1 through 51 on the next time through. Third time, it will choose 1 through 50, and so forth. It does not matter that you have mixed up the order of the deck, since you want the cards shuffled in the first place.

Each element of DECK$(n) consists of a number, or face card name, plus the Commodore 128 CHR$ value for the suit (either 193, 211, 218, or 216). This produces a full deck of 52 cards.

YOU SUPPLY

Draw cards as needed.

SUGGESTED ENHANCEMENTS: Draw actual images of the cards as each is drawn.

RESULT

Shuffled deck of cards is dealt.

DELAY LOOP

WHAT IT DOES: Provides a delay loop that changes in length.

LEVEL: Novice

```
100 REM *********
110 REM *       *
120 REM * DELAY *
130 REM *       *
140 REM *********
150 REM ----------------------------
160 REM      ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM   DELAY:   INITIAL DELAY
190 REM   REPEATS: TIMES TO REPEAT
200 REM   CHANGE:  AMOUNT OF CHANGE
210 REM            PLUS OR MINUS
220 REM  RESULT --
230 REM   LOOP CHANGES IN
240 REM   DELAY EACH TIME
250 REM
260 REM ----------------------------

270 REM *** INITIALIZE ***

280 DELAY=1000
290 CHANGE=.90
300 REPEATS=4
310 GOSUB 21210
320 PRINT"FINISHED"
330 END

21200 REM *** SUBROUTINE ***
```

```
21210 FOR N=1 TO REPEATS
21220 FOR N1=1 TO DELAY
21230 NEXT N1
21240 DELAY=DELAY*CHANGE
21250 PRINT "THROUGH LOOP ";N
21260 NEXT N
21270 RETURN
```

HOW TO USE SUBROUTINE

In games, delay loops are frequently used to display messages on the screen for a given length of time. The Commodore 128 has the SLEEP command to help to make this easier. However, another important use is to control the speed of movement or some other play action not easily controlled through SLEEP.

By having a FOR-NEXT loop count off between each move, you can build in a short delay. A loop from 1 to 100 might slow things down appreciably, whereas setting the upper limit to 10 would produce only a negligible impact.

This subroutine allows the user to vary the length of the delay loop so that action will get faster and faster, until the FOR-NEXT loop is performed only once each time and therefore has almost no effect on the program.

Alternatively, the loop can get longer and longer, so the program will slow down. You might want to place some upper limit so that the action doesn't stop completely after a few minutes.

LINE-BY-LINE DESCRIPTION

Line 280: Set initial delay to 1000.

Line 290: Set change factor to .9.

Line 300: Set number of repeats to 4.

Lines 310–330: Acccess the subroutine and print FINISHED when done.

Line 21210: Start loop from 1 to number of repeats.

Line 21220: Start delay.

Line 21230: End delay.

Line 21240: Change value of delay.

Line 21250: Show iteration through loop, that is, how many times the loop has been run.

Line 21260: Do the loop again with the new delay factor.

YOU SUPPLY

An initial value is needed for DELAY. A high number will start the program off very slowly. A lower number will produce a more moderate beginning speed. You also must define the amount of CHANGE. Fractional numbers will cause DELAY to get smaller each time. That is, if DELAY is 1000 at first, and CHANGE is .90, then DELAY will be set to 900 on the second time through the loop, 810 the third time, and 729 the third time.

As decimal fractions approach 1.0, the amount of speedup each time will be smaller, producing a slower acceleration. Smaller fractions, such as .75 or even .50, will rev up the speed quite quickly.

CHANGE can also be defined as a number larger than 1. Setting it to 1.1 will slowly increase the delay each time. Any number larger than 1.5 (such as 2 or 3) will probably slow down the program more than you desire.

SUGGESTED ENHANCEMENTS: Use sound to show how the delay loop is being counted off.

RESULT

Program speeds up or slows down gradually, at a rate selected by user.

6 INTRODUCTION TO GRAPHICS

Graphics have broad applications in programming, and some users choose their hardware at least partly on the basis of the graphics capabilities of the machine. Complex arrays of numbers and figures can be more easily understood if they are portrayed in chart or graph form. Design work that was done manually can be performed more rapidly with computer graphics. Games, of course, depend heavily on graphic images for their space ships, race cars, and other screen objects.

The Commodore 128 has all the impressive graphics abilities of the Commodore 64, plus a host of new features. Leading the way are 13 new specialized graphics commands that allow the

programmer to draw, capture, fill with color, and move objects on the screen in true high-resolution bit-mapped mode. Eight programmable movable objects called *sprites,* similar to those available with the Commodore 64, are even easier to design and use with the Commodore 128, because of a handy sprite-definition mode and special sprite saving and moving commands.

Many computers, including the Commodore 64, let you use bit-mapped graphics or text, but not both. The Commodore 128 has special commands that allow printing text on a graphics screen. In addition, you may split the screen to use bit-map graphics on one part and ordinary text on the other. Multicolor mode allows drawing of objects and sprites in different colors from a palette of 16.

This introduction to Commodore 128 graphics will be brief. The subject deserves a book of its own and is covered somewhat comprehensively in the *Commodore 128 Programmers' Reference Guide.* Fortunately, with so many graphics commands already in place, there is less need for subroutines, such as those required to effectively use graphics with the Commodore 64.

What you might find beneficial are some simple programs that demonstrate the use of some of the Commodore 128's graphics features. That's what are provided in this chapter. These programs let you use the joystick to draw on the screen in high resolution. Another demonstrates using commands like CIRCLE or BOX to plot on the screen. A third provides a shortcut in generating your own character sets for more sophisticated text graphics. The final program in this chapter shows how the Commodore can capture and move *any* bit-map image drawn on the screen (not just sprites), within a certain size limitation.

COMMODORE 128 GRAPHICS

Throughout this book we've sometimes treated the Commodore 128 as if it were two computers in one. As you probably know, it is actually *three* computers in one package: the Commodore 128, the Commodore 64, and a CP/M machine. With its 40-

and 80-column modes, the computer could be treated as five different PCs.

When it comes to graphics, even in Commodore 128 mode you must treat the computer as if it were two different machines. Each screen format is controlled by a separate specialized microprocessor chip. The 80-column chip, the 8563, offers 16 colors and, at least for the purposes of this book, can be used only for text and character graphics. In 80-column mode the Commodore 128 treats the screen as 2000 separate locations (80 columns × 25 lines) and can place any of the characters in its character set within each of those 2000 locations.

In one sense, bit-mapped mode is less flexible. Instead of printing characters in its screen locations, bit-map mode can only switch each of them on or off. However, you have a great many more locations to work with—64,000 instead of 2000. So, in practice, bit-mapped graphics is *more* flexible because you can print a great deal more detail on the screen. Bit-map mode is also slightly more complex because of the amount of information you are working with.

The 40-column Commodore 128 is controlled by the Video Interface Controller, or VIC chip. The same 16 colors available from 80-column mode are possible, plus bit-mapped graphics, sprites, and a host of new graphics commands.

The key to using graphics is to select the proper graphics mode. Six are available:

Graphics 0: This is standard 40-column text. Only alphanumerics and special symbols in the Commodore 128 character set are allowed. This is the mode the computer boots up in when you have selected the 40-column display.

Graphics 1: This is the full-screen bit-mapped display allowing you to draw on a screen that is 320 dots or picture elements wide by 200 dots tall. These picture elements are usually called *pixels* for short.

Graphics 2: This is a bit-map screen that can be split to allow ordinary text on the lower portion. You control how many lines are set aside for text when the mode is invoked. A single line

can be set aside, or most of the screen. The default value is five lines. Of course, the area used for bit-map graphics is reduced in split-screen mode.

Graphics 3: This is the multicolor bit map. Each pixel doubles in width, but you may mix colors. Resolution is half as good, but the flexibility in using color increases.

Graphics 4: This is the split-screen version of multicolor bit-mapped graphics.

Graphics 5: This is the 80-column text and character-graphics mode—the default mode when the Commodore 128 is booted up with the 80-column RGB display.

Graphics modes are entered through the GRAPHIC command:

GRAPHIC <mode number>,<0 to leave the existing graphics screen or 1 to clear it>, <0–25, to indicate the line on which you want the text to start. This option is valid only for split-screen modes 2 and 4.>

The other basic graphics command is COLOR, which allows you to set various color combinations on the screen. This command has two arguments, the source or section of the screen to be colored, and the color code. The sources available are:

Source 0: This controls the 40-column background color.

Source 1: This controls the foreground color for the graphics screen. This is the color of the objects you might be drawing with other graphics commands.

Source 2: In multicolor mode, each of the two foreground colors can be specified independently. Source 2 controls one of them.

Source 3: This controls the second foreground color in multicolor mode.

Source 4: This controls the border surrounding the 40-column screen, whether in text or graphics mode. Previous subroutines in this book have used POKEs to make this change. The POKEs will work in either Commodore 128 or Commodore 64 mode. COLOR 4,x, where x is a color number 1–16, will work only in Commodore 128 mode.

Source 5: This changes the character color for either the 40- or 80-column text screen. Again, a POKE or printing the color's CHR$ value was used previously, where COLOR 5,x may be used instead.

Source 6: This is used to change the 80-column background color.

Note that the colors produced by the 16 color numbers differ slightly between 40- and 80-column mode. Pages 97 and 98 of your *Commodore 128 System Guide* provide lists of the two color sets.

There are six other nonsprite graphics commands. BOX allows drawing rectangles, including squares, with a single statement at the location of your choice. CIRCLE permits quick drawing of ovals, including circles, and other polygons at the coordinates you select. Figures drawn with either can be filled in with PAINT. DRAW is used to plot from a specific point (or the last location of the pixel cursor) to the points you designate. All these commands have a number of options; consult your System Guide or the Programmer's Reference Guide for a more complete discussion of these, as well as SCALE, which sets the relative size of the bits while maintaining their proportions to one another.

BIT-MAP DRAWING

WHAT IT DOES: Draws on bit-mapped screen in high resolution.

LEVEL: Intermediate

```
100 REM ******************
110 REM *                  *
120 REM * BIT MAP DRAWING *
130 REM *                  *
140 REM ******************
150 REM ------------------------------
160 REM     ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    SCREEN AND LINE COLOR CAN
190 REM    BE CHOSEN BY USER FROM
200 REM    COLOR LIST
210 REM  RESULT --
```

```
220 REM     DRAWING ON BIT MAPPED
230 REM     SCREEN USING JOYSTICK
240 REM
250 REM ------------------------------

260 REM *** INITIALIZE ***

270 DARK=1:WHITE=2
280 RED=3:CYAN=4
290 PURPLE=5:GREEN=6
300 BLUE=7:YELLOW=8
310 PUMPKIN=9:BROWN=10
320 LRED=11:DGRAY=12
330 MGRAY=13:LGREEN=14
340 LBLUE=15:GRAY=16
350 COLOR 0,WHITE
360 COLOR 1,RED
370 GRAPHIC 1,1
380 SOURCE=1
390 X1=1:Y1=1

22000 REM *** SUBROUTINE ***

22010 A=JOY(1)
22020 ON A GOSUB 22050,22070,22120,22150,22200,22230,22280,22310
22030 DRAW SOURCE,X1,Y1
22040 GOTO 22010
22050 Y1=Y1-1:IF Y1<0 THEN Y1=0
22060 RETURN
22070 Y1=Y1-1
22080 X1=X1+1
22090 IF X1>319 THEN X1=319
22100 IF Y1<0 THEN Y1=0
22110 RETURN
22120 X1=X1+1
22130 IF X1>319 THEN X1=319
22140 RETURN
22150 Y1=Y1+1
22160 X1=X1+1
22170 IF Y1>199 THEN Y1=199
22180 IF X1>319 THEN X1=319
22190 RETURN
22200 Y1=Y1+1
22210 IF Y1>199 THEN Y1=199
22220 RETURN
22230 Y1=Y1+1
22240 X1=X1-1
22250 IF Y1>199 THEN Y1=199
22260 IF X1<0 THEN X1=0
22270 RETURN
22280 X1=X1-1
22290 IF X1<0 THEN X1=0
22300 RETURN
22310 X1=X1-1
22320 Y1=Y1-1
22330 IF X1<0 THEN X1=0
22340 IF Y1<0 THEN Y1=0
22350 RETURN
```

HOW TO USE SUBROUTINE

This very simple routine allows using the joystick to draw objects on the bit-mapped screen. You may substitute one of the colors from the color list provided to change the color of the foreground and background.

LINE-BY-LINE DESCRIPTION

Lines 270–340: Define a set of variable names with the 16 color numbers.

Line 350: Set the background color to white.

Line 360: Set the foreground color to red.

Line 380: Define the source to be drawn on as the foreground. You may also draw on the background by changing SOURCE to equal 0.

Line 390: Define the initial coordinates of the pixel cursor, X1 and Y1 as 1,1—the upper left-hand corner of the screen. If you wish the drawing to begin somewhere else on the screen, you may redefine these values.

Line 22010: Read Joystick 1.

Line 22020: Depending on the value returned by Joystick 1, access one of eight movement routines.

Line 22030: Draw a single pixel at the current values of X1 and Y1.

Line 22040: Return for more joystick input.

Lines 22050–22350: All these lines are basically the same routine, modified to account for the direction the joystick was pressed. If the joystick was pressed in a north, northeast, or northwest direction, then Y1 is decremented by one. If Y1 would then become less than 0, then it is returned to a 0 value, to keep the pixel cursor from attempting to move off the top of the screen.

If the joystick was pressed in a south, southeast, or southwest direction, then Y1 is incremented and kept from becoming more than 199, which would take it off the bottom of the screen.

A similar routine takes place for northeast, east, and south-

east directions (X1 is incremented), as well as northwest, west, and southwest indications (X1 is decremented). In no case is X1 permitted to decrease to less than 0, or to more than 319.

YOU SUPPLY

Color choices and joystick movement.

SUGGESTED ENHANCEMENTS: Add a way to erase the screen or to change colors during drawing. Hints may be found in drawing routines in Chapter 5.

RESULT

Drawing on high-resolution bit-mapped screen.

GRAPHICS PLOTTING

WHAT IT DOES: Draws figures on screen from user input.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                     *
120 REM * GRAPHICS PLOTTING   *
130 REM *                     *
140 REM **********************
150 REM ------------------------------
160 REM      ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     SCREEN AND LINE COLOR CAN
190 REM     BE CHOSEN BY USER FROM
200 REM     COLOR LIST
210 REM   RESULT --
220 REM     PLOTTING ON BIT MAPPED
230 REM     SCREEN USING COORDINATES
240 REM
250 REM ------------------------------

260 REM *** INITIALIZE ***

270 DARK=1:WHITE=2
280 RED=3:CYAN=4
290 PURPLE=5:GREEN=6
```

```
300 BLUE=7:YELLOW=8
310 PUMPKIN=9:BROWN=10
320 LRED=11:DGRAY=12
330 MGRAY=13:LGREEN=14
340 LBLUE=15:GRAY=16
350 COLOR 0,WHITE
360 COLOR 1,RED
370 GRAPHIC 2,1

22400 REM *** SUBROUTINE ***

22410 SCNCLR
22420 PRINT:PRINT:PRINT:PRINT
22430 PRINT" 1. DRAW CIRCLE OR POLYGON"
22440 PRINT" 2. DRAW BOX  3. CHANGE COLOR"
22450 PRINT TAB(4)"ENTER CHOICE :"
22460 GETKEY A$
22470 A=VAL(A$)
22480 IF A<1 OR A>3 GOTO 22460
22490 ON A GOTO 22500,22850,23030
22500 PRINT "CENTER COORDINATES OF SHAPE  (X,Y)"
22510 INPUT X$,Y$
22520 X=VAL(X$):Y=VAL(Y$)
22530 PRINT "X RADIUS?"
22540 INPUT XR$
22550 XR=VAL(XR$)
22560 PRINT "Y RADIUS :"
22570 INPUT YR$
22580 YR=VAL(YR$)
22590 IF YR=0 THEN YR=XR
22600 PRINT "STARTING ARC ANGLE?"
22610 PRINT "(DEFAULT IS 0)"
22620 INPUT SA$
22630 SA=VAL(SA$)
22640 PRINT "ENDING ARC ANGLE?"
22650 PRINT "DEFAULT IS 360"
22660 INPUT EA$
22670 EA=VAL(EA$)
22680 IF EA=0 THEN EA=360
22690 PRINT "ROTATION (DEFAULT 0)"
22700 INPUT AN$
22710 AN=VAL(AN$)
22720 PRINT "DEGREES BETWEEN SEGMENTS"
22730 PRINT "(DEFAULT IS 2 DEGREES)"
22740 INPUT IC$
22750 IC=VAL(IC$)
22760 IF IC=0 THEN IC=2
22770 PRINT "PAINT IT? (Y/N)"
22780 GETKEY A$
22790 IF A$="Y" THEN PN=1:GOTO 22820
22800 IF A$="N" THEN PN=0:GOTO 22820
22810 GOTO 22780
22820 CIRCLE 1,X,Y,XR,YR,SA,EA,AN,IC
22830 IF PN=1 THEN PAINT 1,X,Y,0
22840 GOTO 22420
22850 PRINT "ENTER TOP LEFT CORNER"
22860 PRINT "COORDINATES (X,Y) :"
22870 INPUT X1$,Y1$
```

```
22880 X1=VAL(X1$):Y1=VAL(Y1$)
22890 PRINT "ENTER BOTTOM RIGHT CORNER"
22900 PRINT "COORDINATES (X,Y) :"
22910 INPUT X2$,Y2$
22920 X2=VAL(X2$):Y2=VAL(Y2$)
22930 PRINT "ENTER ANGLE OF ROTATION :"
22940 INPUT AN$
22950 AN=VAL(AN$)
22960 PRINT "PAINT THE BOX? (Y/N)"
22970 GETKEY A$
22980 IF A$="Y" THEN PN=1:GOTO 23010
22990 IF A$="N" THEN PN=0:GOTO 23010
23000 GOTO 22970
23010 BOX 1,X1,Y1,X2,Y2,AN,PN
23020 GOTO 22420
23030 PRINT:PRINT:PRINT
23040 PRINT "CHANGE :"
23050 PRINT "1. FOREGROUND COLOR"
23060 PRINT "2. BACKGROUND COLOR"
23070 GETKEY A$
23080 A=VAL(A$)
23090 IF A<1 OR A>2 GOTO 23070
23100 ON A GOTO 23110,23160
23110 PRINT:PRINT:PRINT
23120 PRINT "ENTER FOREGROUND COLOR NUMBER:";
23130 INPUT FG
23140 COLOR 1,FG
23150 GOTO 22420
23160 PRINT:PRINT:PRINT
23170 PRINT "ENTER BACKGROUND COLOR NUMBER :"
23180 INPUT BG
23190 COLOR 0,BG
23200 GOTO 22420
```

HOW TO USE SUBROUTINE

This routine is a miniprogram in its own right that allows using the CIRCLE and BOX commands to plot various-sized objects on the screen. You can fill them with color. The routine demonstrates the split-screen bit-map mode and allows experimenting to see what different radii, arc angles, and so forth do to change the shape of a figure on the screen.

As the program is written, you may specify coordinates that would draw a figure on the text portion of the screen. If you do so, the figure is obscured by the text. The routine was written this way to allow drawing figures up to the very edge of the text screen.

Review your System Guide for a more complete discussion of what each of the parameters of the CIRCLE and BOX command does.

LINE-BY-LINE DESCRIPTION

Lines 270–340: Define a set of variable names as the 16 color numbers.

Lines 350–360: Set foreground and background colors.

Line 370: Enter split screen bit-map mode and clear the screen.

Lines 22410–22420: Clear the text screen and scroll down four lines.

Lines 22430–22480: Present menu of choices, get user choice.

Line 22490: Branch to choice selected by user.

Lines 22500–22520: Ask for coordinates of the center of the oval or polygon.

Lines 22530–22590: Ask for x and y radii of the shape. If both are the same, the figure will be a circle or a perfectly symmetrical polygon. Otherwise, it will be an oval or a figure that is longer in one direction than another.

Lines 22600–22680: Ask for beginning and ending arc angles.

Lines 22690–22710: Ask for rotation of the figure. This will be meaningful only when the radii are different. A rotated circle or perfectly symmetrical polygon looks exactly the same.

Lines 22720–22760: Ask for degrees between segments. Specifying larger increments turns the object from a circle or oval into a polygon, since a circle is a polygon with an infinite number of sides. (Actually, with the Commodore 128 or any personal computer, the number of "sides" in an arc on the screen is finite. The higher the resolution, the smaller they are and the more arclike the curve looks.)

Lines 22770–22810: Ask if figure should be painted with color. If so, set the paint flag, PN, to 1.

Line 22820: Draw the figure using the parameters entered.

Line 22830: If PN set to 1, then also paint the figure.

Line 22840: Return to the menu.

Lines 22850–22920: Enter top left corner of the box to be drawn, and the bottom left corner.

Lines 22930–22950: Get the angle of rotation desired.

Lines 22960–23000: Ask if box should be painted.

Line 23010: Draw the box using the parameters specified.
Line 23020: Return to the menu.
Lines 23030–23190: Ask if foreground or background color is to be changed, and get the color number. Then, change the color and return.

YOU SUPPLY

Initial screen color, drawing color during program run.

SUGGESTED ENHANCEMENTS: Change to provide better error traps for illegal values during the input. Warn the user when a figure will be obscured by the text lines. Let the user enter color names as defined earlier in the program instead of color numbers.

RESULT

Color plotting on screen.

PROGRAMMING CHARACTERS

WHAT IT DOES: Redefines five characters to user-specified set.

LEVEL: Intermediate

```
100 REM **********************
110 REM *                     *
120 REM * PROGRAM CHARACTERS *
130 REM *                     *
140 REM **********************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    DATA LINES FOR DESIRED
190 REM    CHARACTER SET
200 REM  RESULT --
210 REM    CHARACTERS REDEFINED
220 REM
230 REM ------------------------------

240 REM *** DATA ***
```

```
250 DATA 226,164,27,40,108,176,175,132
260 DATA 195,195,68,60,24,24,60,24
270 DATA 255,129,189,165,165,189,129,255
280 DATA 195,195,68,60,24,24,60,24
290 DATA 255,129,189,165,165,189,129,255
300 REM *** INITIALIZE ***
310 PRINT CHR$(142)
320 GOSUB 23310
330 SCNCLR
340 PRINT "!=><@"
350 END

23300 REM *** SUBROUTINE ***

23310 POKE 54,48
23320 POKE 58,48
23330 CLR
23340 BANK 14
23350 FOR N=1 TO 511
23360 POKE N+12288,PEEK(N+53248)
23370 NEXT N
23380 BANK 15
23390 POKE 2604,(PEEK(2604)AND240)+12
23400 FOR N=12288 TO 12295
23410 READ H
23420 POKE N,H
23430 NEXT N
23440 FOR N=12552 TO 12559
23450 READ H
23460 POKE N,H
23470 NEXT N
23480 FOR N=12784 TO 12791
23490 READ H
23500 POKE N,H
23510 NEXT N
23520 FOR N=12768 TO 12775
23530 READ H
23540 POKE N,H
23550 NEXT N
23560 FOR N=12776 TO 12783
23570 READ H
23580 POKE N,H
23590 NEXT N
23600 RETURN
```

HOW TO USE SUBROUTINE

The Commodore 128 allows redefining its character set. The existing characters are constructed on an 8 by 8 dot matrix, with the first and last columns usually empty and the bottom row empty. That arrangement leaves space between each character

and the next. Look at how a letter "A" is put together. Each "0" is considered a blank space, and each "1" a dot that is filled in:

```
0 0 0 1 1 0 0 0
0 0 1 0 0 1 0 0
0 1 0 0 0 0 1 0
0 1 1 1 1 1 1 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0
```

See the letter "A" in that pattern? Since each row is eight characters across, and each character is either a 0 or a 1, it is convenient to think of each row as a byte and to store it that way in memory. Eight consecutive bytes will store the eight rows needed to describe a given character.

This is exactly what the Commodore 128 does. The information on the letter "A" begins at memory location 53256 and continues for eight bytes. The first line above, 00011000, in binary, is 24 in decimal. Similar eight-byte groups are found in memory to tell the computer how to form all the alpha and graphics characters, including reversed characters.

Unfortunately, characters are actually stored in ROM. We can READ the information but not change it. However, when the Commodore 128 wants to find out how to build a given character, it does not go directly to the proper ROM location. Instead, it checks a RAM location, which tells it where to find the beginning of the character memory.

If you change this, you must arrange to have *all* the characters you want to use moved to the new location. The Commodore 128 will not go back and forth, looking in ROM for some characters and RAM for others. Normally, this is accomplished by copying from ROM the information about all the characters you want to use and then modifying only those you want changed.

That is what is done in this subroutine. The first step is to tell the Commodore 128 not to look at the normal location for its character information but to start at 12288 decimal instead. Because this location is BASIC RAM, you have to protect it by

lowering the top of RAM memory. You accomplish that by POKing 48 into 54 and 58 decimal, which are the registers that keep track of how much RAM is available for programs. Once those pointers have been changed, your program will not use any of the memory set aside for characters. The new character set will be safe.

Next, you will copy 64 characters from ROM into the protected RAM locations. This is done with a FOR-NEXT loop, which PEEKs in the ROM, extracts a byte, and POKEs it in the next location of the protected area.

If the program did nothing more than that, then the character set would look exactly the same, except that the Commodore 128 would be obtaining the information from a different place. Instead, you will POKE some new data into the locations for some selected characters that are not needed by your program. These characters are the "at" sign (@), the exclamation point (!), the greater than symbol (>), the less than symbol (<), and the equals sign (=). The characters chosen now are defined beginning at 12288, 12552, 12784, 12768, and 12776, respectively.

You POKE those new values, determined by laying out an 8 by 8 dot grid. Some sample characters are supplied as DATA lines. To form your own, change the binary values obtained to the decimal equivalent, and substitute in the DATA lines.

LINE-BY-LINE DESCRIPTION

Lines 250–290: DATA to form new characters.

Lines 310–350: Access subroutine and print new characters.

Lines 23310–23320: Protect memory, and redefine where computer collects its character set from.

Line 23330: Clear the memory

Line 23340: Select Bank 14 to read ROM.

Lines 23350–23370: Copy old characters from ROM, arrange for new set.

Line 23380: Select Bank 15 to POKE new charaters.

Lines 23390–23600: POKE data for new characters, !,=,>,<, and @.

YOU SUPPLY

Your own DATA lines for characters of your choice. Construct these new DATA lines corresponding to your redesigned characters as follows: Lay out your characters in an 8 × 8 grid, as shown above, and convert each byte to binary. This can be done by taking each of the eight bits, from right to left, and multiplying by 2 to the P power, where P is the position, from the right, of that bit.

For example, 10010111 would be:

1 times 2 to the zeroth power	(1)
1 times 2 to the first power	(2)
1 times 2 to the second power	(4)
0 times 2 to the third power	(0)
1 times 2 to the fourth power	(16)
0 times 2 to the fifth power	(0)
0 times 2 to the sixth power	(0)
1 times 2 to the seventh power	(128)
Total:	151 decimal

Repeat for each byte, to the total of eight in the matrix. You may also use the Binary to Decimal subroutine in Chapter 9.

SUGGESTED ENHANCEMENTS: Write a program to let you draw on the screen in an 8 × 8 matrix, and then translate the binary values to decimal automatically.

RESULT

Pressing "@", "!", "<", "=", and ">" keys or using them in a program will produce new, redefined characters.

SHAPE MOVER

WHAT IT DOES: Allows capturing a shape you draw on the screen in a string variable.

LEVEL: Intermediate

```
100 REM ***************
110 REM *               *
120 REM * SHAPE MOVER *
130 REM *               *
140 REM ***************
150 REM ------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    DRAW SHAPE WITH CURSOR KEYS
190 REM    PRESS ESCAPE WHEN DONE
200 REM    X1:  X COORDINATE TO APPEAR
210 REM    Y1:  Y COORDINATE TO APPEAR
220 REM  RESULT --
230 REM    SHAPE APPEARS AT SPECIFIED
240 REM    LOCATION ON BIT-MAP SCREEN
250 REM
260 REM ------------------------------

270 REM *** INITIALIZE ***

280 COLOR 0,1
290 GRAPHIC 1,1
300 X1=10:Y1=10
310 X=110:Y=100
320 COLOR 1,2
330 BOX 1,108,98,157,142

23700 REM *** SUBROUTINE ***

23710 GETKEY A$
23720 IF A$=CHR$(27) GOTO 23960
23730 IF A$=CHR$(29) THEN BEGIN
23740 X=X+1
23750 IF X>155 THEN X=155
23760 GOTO 23940
23770 BEND
23780 IF A$=CHR$(157) THEN BEGIN
23790 X=X-1
23800 IF X<110 THEN X=110
23810 GOTO 23940
23820 BEND
23830 IF A$=CHR$(145) THEN BEGIN
23840 Y=Y-1
23850 IF Y<100 THEN Y=100
23860 GOTO 23940
23870 BEND
```

```
23880 IF A$=CHR$(17) THEN BEGIN
23890 Y=Y+1
23900 IF Y>140 THEN Y=140
23910 GOTO 23940
23920 BEND
23930 GOTO 23710
23940 DRAW 1,X,Y
23950 GOTO 23710
23960 SSHAPE C$,110,100,155,140
23970 GRAPHIC 1,1
23980 GETKEY B$
23990 GSHAPE C$,X1,Y1,0
```

HOW TO USE SUBROUTINE

Sprites are objects drawn on the screen within a matrix 24 pixels wide and 21 pixels tall. Once they are defined, you can move sprites with powerful Commodore 128 sprite commands as if they were discrete objects. BASIC 7.0 has a sprite definition mode (SPRDEF) that lets you define sprites, store that shape in a string variable, and then define the shape in that variable as one of eight sprites.

This routine serves as an introduction to how sprite shapes are captured and moved. However, to show that the techniques used are not limited to sprites, we'll let you define and move a shape *larger* than that permitted for sprites. You may use the cursor keys to draw on the screen. Press ESCAPE when done. The shape will be stored in string variable C$ and moved to the coordinates specified.

LINE-BY-LINE DESCRIPTION

Lines 280–290: Set graphic mode and colors.

Lines 300–310: Define the limits of the area to be drawn.

Line 330: Draw box to show drawing area.

Line 23710: Get a key.

Line 23720: If ESCAPE is pressed, go to capture routine.

Lines 23730–23930: If key is cursor movement key, move cursor, keeping within limits of drawing box.

Line 23940: Plot a point at current pixel cursor location.

Line 23950: Return for more input.

Line 23960: Store the shape within the coordinates in C$.
Line 23970: Clear the graphics screen.
Line 23980: Wait for user to press a key.
Line 23990: Print the shape stored at the specified coordinates.

YOU SUPPLY

You can change the values for where the shape is printed.

SUGGESTED ENHANCEMENTS: This is a demonstration program. Your program would supply coordinates to print the shape. Or you might ask the user where to put it. Such a program would allow drawing shapes in one location, assigning them to a series of variables (C$(1), C$(2), etc.), and moving them around the screen at will. Note that these shapes are *larger* than sprites but must be moved by your program lines, rather than automatically as are sprites. In addition, sprites allow assigning priorities so that one sprite will appear to move in front of or behind other objects on the screen.

RESULT

Image drawn on screen captured in string variable and moved.

7 USING SOUND

The Commodore 128 has some of the best sound capabilities of any personal computer. When the machine is connected to a good-quality stereo system, the sounds duplicate that of music synthesizers costing many times what the entire computer sells for.

The secret, of course, is the music synthesizer chip built into the Commodore 128. Earlier computers without the chip had much more primitive sound-generation capabilities. Some could produce sounds through only a single voice. The VIC-20 had three musical voices and one "noise" generator with overlapping octaves. By comparison, the VIC-20's musical capabilities don't

hold a candle to the many areas of control that the Commodore 128's synthesizer chip provides.

The Commodore 128 also has three voices, but in addition to pitch and duration, the user can specify the waveform of each voice, varying it from "sawtooth" to "pulse" to "triangle" to "noise." These terms may not mean much to you. However, it is the waveform of each musical note that helps give various musical instruments their distinctive "timbre," or sound quality. Your Commodore 128 can play a note with four different waveforms and can also "filter" the waveform through a selection of "modulators."

Also important is the sound *envelope*. This is controlled by the attack/decay and sustain/release parameters. When a note is first played, it rises from zero volume to its peak volume, then falls back to some middle range. The rate of rise is called *attack*, and the speed of decline to the middle range is called *decay*. That middle volume, called *sustain*, can also be controlled by the Commodore 128. When the note finally stops playing, its rate of decline to zero volume is called *release*. The attack/decay and sustain/release properties of a trumpet note differ from those produced by a piano. If we know each, we can duplicate the sound fairly closely with the Commodore 128.

Unlike the Commodore 64, which required complex POKEs to a multitude of registers to produce sounds, the Commodore 128 has new sound and music commands that take care of most of the work for you. These include SOUND, ENVELOPE, VOL, TEMPO, PLAY, and FILTER. Consult your System Guide or the Programmer's Reference Guide for an in-depth discussion of each. In this chapter we have some plug-in programs and demonstrations that use three of these six commands:

SOUND: Allows you to produce a sound using one of three voices, at a frequency you specify, for a duration you request, and using the waveform and pulse width you enter. The sound can also be swept through a range of frequencies with other parameters.

VOL: Sets the volume level from 0 to 15.

PLAY: Will allow you to enter actual note names, such as A or C#, and will play them.

The intent is to provide you with "plug-in" subroutines that you can use in your own programs immediately, even if you can't tell a synthesizer from photosynthesis. This section contains routines that have broad application in many games programs; they can also be used to spice up your general programming efforts.

The first routine generates musical notes when the keys of the home row and the row above are pressed. Others produce a grating siren sound, a madcap computer gone berserk, eerie flying saucer noises, and other effects.

You may want to experiment with each of these, using some of the suggestions provided or ideas of your own. Change the values of FOR-NEXT loops. Use different voices, as suggested. You should be able to develop new sounds on your own, until a whole library of sound effects is available.

COMMODORE 128 ORGAN

WHAT IT DOES: Uses keyboard to generate various musical notes.

LEVEL: Intermediate

```
100 REM ************************
110 REM *                      *
120 REM *  COMMODORE 128 ORGAN *
130 REM *                      *
140 REM ************************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    VOICE:  VOICE USED
190 REM    LE$:    LENGTH OF NOTE
200 REM    L:      VOLUME
210 REM    OTHER PARAMETERS CHANGED
220 REM    FROM KEYBOARD
230 REM  RESULT --
240 REM ------------------------------
250 REM    MUSIC PLAYED
260 REM
270 GOTO 24060

24000 REM *** POSITION CURSOR ***

24010 PRINT HME$;
24020 PRINT LEFT$(R$,COL);
```

```
24030 PRINT LEFT$(D$,ROW);
24040 RETURN

24050 REM *** INITIALIZE ***

24060 POKE 53281,1
24070 KEY 1,CHR$(133)
24080 KEY 3,CHR$(134)
24090 KEY 5,CHR$(135)
24100 KEY 7,CHR$(136)
24110 DWN$=CHR$(17)
24120 UP$=CHR$(145)
24130 RV$=CHR$(18)
24140 FF$=CHR$(146)
24150 RED$=CHR$(28)
24160 GREEN$=CHR$(30)
24170 YELLOW$=CHR$(158)
24180 BLUE$=CHR$(31)
24190 HME$=CHR$(19)
24200 VOICE=1
24210 O=1
24220 T=6
24230 L=15

24240 REM *** READ DATA ***

24250 LE$="Q"
24260 DIM NME$(20),NT$(20),P(20)
24270 FOR N=1 TO 20
24280 READ NME$(N)
24290 NEXT N

24300 FOR N=1 TO 20
24310 READ NT$(N)
24320 NEXT N
24330 FOR N=0 TO 9
24340 READ TUNE$(N)
24350 NEXT N

24360 REM *** BUILD CURSOR STRING ***

24370 FOR N=1 TO 40
24380 R$=R$+CHR$(29)
24390 D$=D$+CHR$(17)
24400 SPACE$=SPACE$+CHR$(32)
24410 NEXT N

24420 REM *** SET UP SCREEN ***

24430 SCNCLR
24440 COL=1:ROW=20:GOSUB 24010
24450 PRINT TAB(2);RV$;BLUE$;"F1 ";FF$;" OCTAVE UP";
24460 PRINT TAB(20);RV$;YELLOW$;"F3";FF$" OCTAVE DOWN"
24470 PRINT TAB(2);RV$;RED$;"F5 ";FF$;" INSTRUMENT UP";
24480 PRINT TAB(20);RV$;GREEN$;"F7";FF$" INSTRUMENT DOWN"
24490 ROW=4:COL=10
24500 GOSUB 24010
24510 PRINT RV$;RED$;"COMMODORE 128 ORGAN";FF$
```

```
24520 REM *** DATA ***

24530 DATA C,#C,D,#D,E,F,#F,G,#G,A,#A,B,C,#C,D,#D,E,F,#F,G
24540 DATA A,W,S,E,D,F,T,G,Y,H,U,J,K,O,L,P,":",";","*","="
24550 DATA PIANO,ACCORDION,CALLIOPE,DRUM
24560 DATA FLUTE,GUITAR,HARPSICHORD,ORGAN
24570 DATA TRUMPET,XYLOPHONE

24580 REM *** SUBROUTINE ***

24590 COL=13:ROW=10:GOSUB 24010
24600 PRINT SPACE$;
24610 COL=13:ROW=10:GOSUB 24010
24620 F=INT((11-LEN(TUNE$(T)))/2)
24630 PRINT LEFT$(R$,F);
24640 PRINT TUNE$(T);
24650 GETKEY A$
24660 IF A$=CHR$(13) THEN END
24670 IF A$=CHR$(133) THEN O=O+1:IF O>6 THEN O=6:GOTO 24590
24680 IF A$=CHR$(134) THEN O=O-1:IF O<0 THEN O=0:GOTO 24590
24690 IF A$=CHR$(135) THEN T=T+1:IF T>9 THEN T=9:GOTO 24590
24700 IF A$=CHR$(136) THEN T=T-1:IF T<0 THEN T=0:GOTO 24590
24710 FOR N=1 TO 20
24720 IF A$=NT$(N) GOTO 24750
24730 NEXT N
24740 GOTO 24590
24750 ROW=15:COL=18:GOSUB 24010
24760 N$=NME$(N)
24770 IF LEFT$(N$,1)="#" THEN N$=RIGHT$(N$,1)+LEFT$(N$,1)
24780 PRINT N$;CHR$(32);
24790 IF N>12 THEN BEGIN
24800 O=O+1
24810 IF O>6 THEN O=6:GOTO 24840
24820 FLAG=1
24830 BEND
24840 P$="V"+MID$(STR$(VOICE),2)+"O"+MID$(STR$(O),2)
+"T"+MID$(STR$(T),2)+"U"+MID$(STR$(L),2)+LE$+NME$(N)
24850 PLAY P$
24860 IF FLAG=1 THEN O=O-1:IF O<0 THEN O=0
24870 FLAG=0
24880 GOTO 24590
```

HOW TO USE SUBROUTINE

This is another subroutine that got out of hand and ended up as a 100-line program in its own right. Although packed with features, this program only begins to tap the power of the Commodore 128's music synthesizer capabilities.

You may define the voice used and length of the note, as well as the volume. The octave and instrument can be changed from the keyboard. You could add routines to allow using more than one voice and varying the tempo.

LINE-BY-LINE DESCRIPTION

Lines 24010–24040: This is a cursor-positioning routine used here for the first time in the book. It is described completely in Chapter 8 under Cursor Mover.

Line 24060: Change color of the screen; COLOR could also be used here.

Lines 24070–24100: Change the definitions of the Function Keys 1, 3, 5, and 7 (the unshifted keys) so they return the CHR$ values assigned to them in Commodore 64 mode.

Lines 24110–24140: Define string variables with the characters that print cursor down, cursor up, reverse on, and reverse off characters.

Lines 24150–24180: Assign other variable names with color characters.

Line 24190: Define HME$ as HOME.

Line 24200: Set VOICE to 1.

Line 24210: Set initial octave to 1.

Line 24220: Set initial instrument to 6.

Line 24230: Set loudness to 15.

Line 24250: Define the length of the notes played to quarter note.

Line 24260: DIMension two arrays, NME$(n) for the names of the notes, NT$(n) for the corresponding keyboard keys.

Lines 24270–24320: Read those values into the arrays.

Lines 24330–24350: Read the names of the instruments into TUNE$(n).

Lines 24370–24410: Build the cursor movement strings, as discussed in Cursor Mover in Chapter 8.

Lines 24430–24510: Print screen instructions.

Lines 24530–24570: The DATA.

Lines 24590–24640: Erase old instrument printed to the screen and print the new data currently in effect.

Lines 24650: Get a key from the keyboard.

Line 24660: If RETURN was pressed, END. You may delete this if you would rather exit the program by hitting RUN/STOP and RESTORE.

Lines 24670–24680: If F1 is pressed, raise octave by one. If F3 is pressed, decrease by one. In no case allow octave to be less than 0 or higher than 6.

Lines 24690–24700: If F5 is pressed, raise instrument value to next higher instrument. If F7 is pressed, lower. In no case allow instrument to be less than 0 or higher than 9.

Lines 24710–24730: Compare key pressed with allowable keys to see if a legal note has been pressed.

Line 24740: If not, go back for more input.

Lines 24750–24780: Print the name of the note pressed to the screen.

Lines 24790–24830: If note is in the next highest octave, increase octave. However, if the highest octave is already being accessed, do not increase. This allows playing more than one octave on the keyboard, except when we have already changed the lower end of the keyboard to play the highest available octave.

Line 24840: Assemble a PLAY string from the current parameters.

Line 24850: Play the string.

Lines 24860–24870: If note was at upper end of keyboard, reduce octave to next lowest to return to normal.

Line 24880: Return for more input.

YOU SUPPLY

Definitions for voice and note length, plus keyboard input.

SUGGESTED ENHANCEMENTS: Write a routine that will display a keyboard on the screen, and indicate which key is being pressed. Allow changing voices. Store the notes pressed for playback. This is relatively simple, since the PLAY strings may be stored in a string array. Allow changing the length of notes.

RESULT

Music played from Commodore 128 synthesizer.

SIREN

WHAT IT DOES: Produces siren sound.

LEVEL: Novice

```
100 REM *********
110 REM *       *
120 REM * SIREN *
130 REM *       *
140 REM *********
150 REM ------------------------
160 REM    ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    REPEATS:  REPETITIONS
190 REM  RESULT --
200 REM    SIREN SOUND
210 REM
220 REM ---------------------

24990 *** SUBROUTINE ***

25000 REPEATS=4
25010 FOR N=1 TO REPEATS
25020 FOR I=100  TO 20000 STEP 100
25030 SOUND 1,I,1,1,1,1,1,0
25040 NEXT I
25050 FOR I=20000 TO 100 STEP -100
25060 SOUND 1,I,1,0,0,0,1,0
25070 NEXT I
25080 NEXT N
25090 END
```

HOW TO USE SUBROUTINE

Call the routine when a siren sound is desired for games or other applications. The remaining routines in this chapter are written as programs stopping with END rather than RETURN, since there is no initialization required. You will want to add RETURN statements if you use them as subroutines in your own programs. They are short enough to be used as standalone lines within programs.

LINE-BY-LINE DESCRIPTION

Line 25000: Define number of REPEATS.
Line 25010: Start loop to repeat the sound.

Lines 25020–25040: Produce rising sound.
Lines 25050–25070: Produce falling sound.
Line 25080: Repeat.

YOU SUPPLY

Number of repeats.

SUGGESTED ENHANCEMENTS: Experiment to change the sound.

RESULT

Siren sound for games or other programs.

FLYING SAUCER

WHAT IT DOES: Produces eerie flying saucer sound.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * FLYING SAUCER *
130 REM *                 *
140 REM *****************
150 REM ----------------------------
160 REM      ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     NONE
190 REM  RESULT --
200 REM     FLYING SAUCER SOUND
210 REM
220 REM ----------------------------

25100 REM *** SUBROUTINE ***

25110 FOR I=1 TO 65000 STEP 100
25120 SOUND 1,I,1,0,0,3,0
25130 NEXT I
```

HOW TO USE SUBROUTINE

Plug into your program where you want flying saucer sound, or add RETURN to use as subroutine.

LINE-BY-LINE DESCRIPTION

Lines 25110–25130: Produce saucer sound.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Experiment to produce different saucer effects. Change the step value and other parameters.

RESULT

Flying saucer sound for games or other applications.

BURGLAR ALARM

WHAT IT DOES: Produces burglar alarm sound.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * BURGLAR ALARM   *
130 REM *                 *
140 REM *****************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NONE
190 REM  RESULT --
200 REM    BURGLAR ALARM SOUND
210 REM
220 REM ----------------------------

25200 REM *** SUBROUTINE ***

25210 FOR I=1 TO 10
25220 SOUND 1,49000,300,2,32000,3000,1
25230 NEXT I
```

HOW TO USE SUBROUTINE

Insert in your program where you want burglar alarm sound.

LINE-BY-LINE DESCRIPTION

Lines 25210–25230: Produce burglar alarm sound.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Play with it!

RESULT

Burglar alarm sound produced for games, or possibly home control applications with a real world interface.

ALARM SOUND

WHAT IT DOES: Produces different alarm sound.

LEVEL: Novice

```
100 REM ***************
110 REM *               *
120 REM * ALARM SOUND *
130 REM *               *
140 REM ***************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    REPEATS: NUMBER O REPEATS
190 REM  RESULT --
200 REM    ALARM SOUND
210 REM
220 REM ----------------------------
230 REPEATS=5
240 GOSUB 25300
250 END
```

```
25300 REM *** SUBROUTINE ***

25310 FOR I=1 TO REPEATS
25320 SOUND 1,65000,300,0,32000,3000,2,2600
25330 NEXT I
25340 RETURN
```

HOW TO USE SUBROUTINE

Insert in your program where you want an alarm sound different from the last. You may differentiate between two actions with the distinctly different alarms.

LINE-BY-LINE DESCRIPTION

Line 230: Number of REPEATS defined.
Lines 25310–25330: Produce alarm sound.

YOU SUPPLY

Number of REPEATS.

SUGGESTED ENHANCEMENTS: None.

RESULT

Different alarm sound produced.

PLANE ENGINE

WHAT IT DOES: Produces sound of plane engine starting and leveling off.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * PLANE ENGINE *
130 REM *                *
140 REM ****************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    V:  VOICE USED
190 REM  RESULT --
200 REM    SOUND OF PLANE ENGINE
210 REM
220 REM ----------------------------
230 V=1
240 GOSUB 25410
250 END

25400 REM *** SUBROUTINE ***

25410 D=1200
25420 F=12000
25430 DIR=0
25440 S=1
25450 W=1
25460 P=1000
25470 SOUND V,F,D,DIR,M,S,W,P
25480 RETURN
```

HOW TO USE SUBROUTINE

You might want to use this subroutine at the beginning of programs dealing with airplane flight or travel. This program demonstrates the use of the sweep effect with the Commodore 128.

LINE-BY-LINE DESCRIPTION

Line 230: Define voice as 1.

Lines 25410–25460: Define other SOUND parameters.

Line 25470: Produce the sound, sweeping through a range of frequencies.

YOU SUPPLY

Definition for voice to be used.

SUGGESTED ENHANCEMENTS: Try different sweep effects.

RESULT

Sound of airplane engine starting up and leveling off.

BOMB DROPPING

WHAT IT DOES: Produces sound of bomb dropping.

LEVEL: Novice

```
100 REM *************
110 REM *           *
120 REM * BOMB DROP *
130 REM *           *
140 REM *************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NONE
190 REM  RESULT --
200 REM    SOUND OF BOMB DROPPING
210 REM
220 REM ----------------------------
230 GOSUB 25510
240 END

25500 REM *** BOMB DROPPING ***

25510 VOL 15
25520 SOUND 1,49000,480,1,0,100,1,0
25530 RETURN
```

HOW TO USE SUBROUTINE

Call during bombs dropping in your games program.

LINE-BY-LINE DESCRIPTION

Line 25510: Set volume to maximum.

Line 25520: Sweep through range of frequencies to produce effect of bomb dropping.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Change the speed of the bomb dropping. Add an explosion at the end.

RESULT

Sound of bomb dropping produced.

HELICOPTER

WHAT IT DOES: Produces sound of helicopter.

LEVEL: Novice

```
100 REM **************
110 REM *              *
120 REM * HELICOPTER *
130 REM *              *
140 REM **************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     NONE
190 REM   RESULT --
200 REM     HELICOPTER SOUND
210 REM
220 REM ------------------------------

25600 REM *** SUBROUTINE ***

25610 VOL 15
25620 CU=1000
25630 FOR N=100 TO 1000
25640 SOUND 1,N,1
25650 S=N-CU:IF S<38 THEN S=CU-N
25660 SOUND 1,S,1
25670 CU=CU-1
25680 NEXT N
```

HOW TO USE SUBROUTINE

Plug into your programs where you want the sound of a helicopter rotor.

LINE-BY-LINE DESCRIPTION

Line 25610: Set volume to maximum.
Line 25620: Set counter to initial value of 1000.
Line 25630: Loop from 100 to 1000.
Line 25640: Produce sound using frequency N.
Line 25650: Change value of S.
Line 25660: Produce sound using frequency S.
Line 25670: Change value of counter.
Line 25680: Loop and repeat.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: None.

RESULT

Sound of helicopter rotor produced.

COMPUTER SOUND

WHAT IT DOES: Produces random, computerlike sound.

LEVEL: Novice

```
100 REM ******************
110 REM *                  *
120 REM * COMPUTER SOUND *
130 REM *                  *
140 REM ******************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     NONE
190 REM  RESULT --
200 REM     COMPUTER SOUND
210 REM
220 REM ------------------------------

25700 REM ** SUBROUTINE ***

25710 FOR N=1 TO 10
25720 R=RND(0)*60000
25730 L=RND(0)*10
25740 SOUND 1,R,L,0,0,0,2
25750 R=RND(0)*60000
25760 L=RND(0)*20
25770 SOUND 2,R,L,0,0,0,1
25780 R=RND(0)*10000
25790 L=RND(0)*10
25800 SOUND 3,R,L,0,0,0,0
25810 NEXT N
```

HOW TO USE SUBROUTINE

Computers don't really sound like this, of course, but we have come to connect the random production of beeping sounds with computers. You can probably blame this on bad science fiction movies, but the sound effect may come in useful nevertheless. Let this sound emit continuously from your Commodore 128 when friends visit, and see if they are impressed.

LINE-BY-LINE DESCRIPTION

Line 25710: Repeat 10 times.

Lines 25720–25800: Choose random frequency and lengths, and make that sound. Do it for all three voices.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: None.

RESULT

Random computer sound produced.

DISASTER SOUND

WHAT IT DOES: Produces vague, threatening disaster sound.

LEVEL: Novice

```
100 REM ************
110 REM *          *
120 REM * DISASTER *
130 REM *          *
140 REM ************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM      NONE
190 REM  RESULT --
200 REM      ELABORATE EXPLOSION
210 REM
220 REM ------------------------------

25900 REM *** SUBROUTINE ***

25910 VOL 1
25920 SOUND 1,100,10,0,0,0,3
25930 SOUND 1,40,1,0,0,0,3
25940 FOR N=65000 TO 1000 STEP -5000
25950 V=V+1:VOL V
25960 SOUND 1,N,60,0,0,0,3
25970 NEXT N
25980 V=15:VOL V
25990 SOUND 1,8000,1500,1,0,10,3
26000 SOUND 2,2000,750,1,0,10,3
26010 SOUND 3,200,375,1,0,10,3
26020 END
```

HOW TO USE SUBROUTINE

We're not sure what this sound is—maybe an explosion, maybe a volcano erupting. It certainly sounds frightening.

LINE-BY-LINE DESCRIPTION

Line 25910: Set volume low, to 1.
Lines 25920–25930: Play sounds.
Line 25940: Start decreasing FOR-NEXT loop.
Line 25950: First 15 times through the loop, increase volume.
Line 25960: Produce sound, based on frequency N.
Lines 25970–26010: Produce "explosion."

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Fix the explosion to sound better and more explosive.

RESULT

Disaster sound produced.

ROULETTE WHEEL

WHAT IT DOES: Produces roulette wheel sound that gradually slows down.

LEVEL: Novice

```
100 REM ******************
110 REM *                  *
120 REM * ROULETTE WHEEL *
130 REM *                  *
140 REM ******************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NONE
190 REM  RESULT --
200 REM    ROULETTE WHEEL SOUND
210 REM
220 REM ------------------------------
```

```
26100 REM *** ROULETTE WHEEL ***

26110 DELAY=20
26120 F=1.001
26130 F1=.0005
26140 FOR N=1 TO 100
26150 DELAY=DELAY*F
26160 FOR D=1 TO DELAY:NEXT D
26170 SOUND 1,1000,1
26180 F=F+F1
26190 NEXT N
26200 SOUND 1,1000,1
```

HOW TO USE SUBROUTINE

Insert in games where roulette wheel or wheel of fortune type of sound is needed.

LINE-BY-LINE DESCRIPTION

Line 26110: Set initial delay to 20.
Line 26120: Set factor to 1.001.
Line 26130: Set second factor to .0005.
Line 26140: Start loop from 1 to 100.
Line 26150: Increase delay slightly.
Line 26160: Count off the delay.
Line 26170: Make a click sound.
Line 26180: Increase the factor.
Line 26190: Repeat.
Line 26200: Make final click.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: None.

RESULT

Roulette wheel slows down and stops.

CLOCK TICKING

WHAT IT DOES: Produces clock ticking sound.

LEVEL: Novice

```
100 REM **************
110 REM *              *
120 REM * CLOCK TICK   *
130 REM *              *
140 REM **************
150 REM ------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NONE
190 REM  RESULT --
200 REM    CLOCK TICKING SOUND
210 REM
220 REM ------------------------------

26300 REM *** CLOCK TICK ***

26310 VOL 15
26320 FOR N1=1 TO 100
26330 SOUND 1,20000,1
26340 FOR N=1 TO 300:NEXT N
26350 SOUND 1,40000,1
26360 FOR N=1 TO 300:NEXT N
26370 NEXT N1
```

HOW TO USE SUBROUTINE

Insert in your programs where clock ticking needed. You might use this subroutine to represent the passage of time, or during "thinking" periods in games.

LINE-BY-LINE DESCRIPTION

Line 26310: Set volume to maximum.
Line 26320: Repeat 100 times.
Line 26330: Make tick sound.
Line 26340: Delay slightly.
Line 26350: Make tock sound.
Line 26360: Delay again between ticks.
Line 26370: Repeat.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Modify to allow user to vary number of ticks.

RESULT

Clock ticking produced.

8 SOFTWARE TRICKS

Here is a group of routines that will let you perform some tasks specific to the Commodore 128 computer. Included is a simple routine to display color bars on the screen, in case you want to adjust the color balance of your monitor or television set. Another is provided to help you set the Commodore 128's built-in real-time clock. The next measures elapsed time using that clock, and a third lets you set the Commodore 128 as a timer.

Three other subroutines make using the special function keys much easier. These keys can have a variety of tasks assigned to them in your programs. Games, for example, commonly perform some chore when a certain function key is pressed. You might

have F1 pause the game, whereas F3 quits the game entirely. F5 could clear the screen, and F7 would exchange sides. Function keys were used to change the octave and instrument voice in Commodore 128 Organ in Chapter 7. In other applications, function keys could display the score or do some other function. The Function Keys routine in this chapter shows you how to have your own programs branch to specific subroutines when a function key is pressed. Program Keys automates assigning a string of characters to a function key. Utility Keys demonstrates one set of uses for the keys.

The Cursor Mover routine used several times in this book is explained in this chapter. We wind up with a pair of routines that use the Commodore 128's RS-232 port. Program Transfer is a simple one-line transfer program that will send a program listing out the serial port to another computer. Terminal is a dumb terminal program that lets you communicate two ways.

THE REAL-TIME CLOCK

The Commodore 128 has a built-in timing mechanism that allows it to measure seconds, minutes, and hours. This feature is known as the real-time clock, and it can be used by the programmer to keep track of events, such as the length of time needed to complete games. The three subroutines that follow are your key to using the real-time clock of your Commodore computer.

The clock measures time in 1/60th-second intervals. Each of these tiny time increments is popularly called a *jiffie*. Two counters, TI and TI$, keep track of elasped time since the computer was turned on or since the clock was last reset by the user. That is, when the computer is first turned on, TI and TI$ equal 000000 and start counting from that point. You may access both TI and TI$ as you do any other variable. TI numbers the jiffies, while TI$ keeps the actual seconds.

To find out how many jiffies have elapsed since the jiffie counter was last reset, assign the value of TI to some other variable, as: "CU=TI." Similarly, you can take the present time

and use it for programming purposes by assigning its value to a variable, as: "CU$=TI$." You can also set the proper time (the Commodore 128 uses 24-hour, military-style time) by the reverse method: "TI$="120000."

Because the Commodore 128 real-time clock is accurate under most circumstances, TI can be used to time events fairly precisely (one exception is noted below) with approximately 1/60th-second accuracy. This feature might be useful in competitive games, typing tutors, and other programs that measure elapsed time accurately. TI$, which can be set to the current time, keeps track of hours, minutes, and seconds.

Neither TI nor TI$ allows for time lost during input-output functions, such as loading programs from tape or disk, or writing data to tape or disk. Therefore, if your program writes data files, it should not depend on TI or TI$ to be 100 percent accurate. Also, don't expect either to be correct after you have loaded or saved several programs.

The first subroutine of this group sets the real-time clock to the current time, given in 24-hour military style. You can embed this routine in your programs when you need to access the time for your program. The next routine measures actual elapsed time. This is accurate to the second (with the exceptions noted above) and can be used to measure time to that degree of precision.

The final routine uses the computer as a timer. You may set the current time and the time you want to be alerted. The routine will measure that interval and alert you. It uses the SLEEP command to put the Commodore 128 on standby during the required interval. Hints are provided for writing your own subroutine that can be called from time to time while your programs do other tasks.

CLOCK SETTER

WHAT IT DOES: Sets Commodore 128 real-time clock.

LEVEL: Novice

```
100 REM ****************
110 REM *              *
120 REM * CLOCK SETTER *
130 REM *              *
140 REM ****************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    NONE
190 REM  RESULT --
200 REM    INTERNAL CLOCK SET
210 REM
220 REM --------------------------------
230 GOSUB 27210
240 END

27200 REM *** SUBROUTINE ***

27210 PRINT"ENTER HOUR :"
27220 INPUT HR$
27230 HR=VAL(HR$)
27240 IF HR>23 THEN PRINT "LESS THAN 24 HOURS, PLEASE!":GOTO 230
27250 IF HR>12 GOTO 27310
27260 PRINT "A.M. OR P.M. ?"
27270 GETKEY A$
27280 IF A$="P" THEN HR=HR+12:HR$=MID$(STR$(HR),2):GOTO 27310
27290 IF A$="A" GOTO 27310
27300 GOTO 27270
27310 PRINT "ENTER MINUTES :"
27320 INPUT MINUTE$
27330 MINUTE=VAL(MINUTE$)
27340 IF MINUTE>59 THEN PRINT "LESS THAN 60 MINUTES,
PLEASE!":GOTO 27310
27350 IF HR<10 THEN HR$="0"+HR$:IF HR<1 THEN HR$="00"
27360 IF MINUTE<10 THEN MINUTE$="0"+MINUTE$:IF MINUTE<1 THEN
MINUTE$="00"
27370 TI$=HR$+MINUTE$+"00"
27380 RETURN
```

HOW TO USE SUBROUTINE

Many programs can use the built-in real-time clock of the Commodore 128 to measure elapsed time, control events, or simply keep the operator informed as to what time it is.

The Commodore 128 clock keeps 24-hour military time and stores it in a variable, TI$, which can be called from a program at any time. One-thirty P.M. would be stored as "133000." This subroutine prompts the user for the current hours and minutes. If fewer than 12 hours are entered, the routine asks if the time is A.M. or P.M. Illegal time entries, such as 256300, are not allowed.

Minutes and hours less than 10 must be entered with a single digit; the added 0 is appended automatically to produce, say, 090900.

LINE-BY-LINE DESCRIPTION

Lines 27210–27230: User enters hour.
Line 27240: Check to see if less than 24 hours.
Line 27250: If hour is later than 12, skip check to see if it is A.M. or P.M.
Lines 27260–27300: Ask if A.M. or P.M. and change hour to military time if answer is P.M.
Lines 27310–27330: Get current minutes.
Line 27340: Make sure less than 60 minutes entered.
Line 27350: If hour less than 10, add leading 0.
Line 27360: If minutes less than 10, add leading 0.
Line 27370: Define TI$ as hours, plus minutes, plus 0 seconds.

YOU SUPPLY

Subroutine asks for hours and minutes.

SUGGESTED ENHANCEMENTS: None.

RESULT

Internal clock set to correct time.

ELAPSED TIME

WHAT IT DOES: Measures difference between two times.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * ELAPSED TIME *
130 REM *                *
140 REM ****************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    NONE
190 REM  RESULT --
200 REM    ELAPSED TIME MEASURED
210 REM
220 REM --------------------------------

230 REM *** INITIALIZE ***

240 GOSUB 27410
250 END

27400 REM *** SUBROUTINE ***

27410 TS$=TI$
27420 GETKEY A$
27430 TF$=TI$
27440 TS=VAL(TS$)
27450 TF=VAL(TF$)
27460 ET=TF-TS
27470 EM=INT(ET/60)
27480 ES=ET-(EM*60)
27490 PRINT "ELAPSED TIME :"
27500 PRINT " MINUTES :";EM
27510 PRINT " SECONDS :";ES
```

HOW TO USE SUBROUTINE

This subroutine takes the number of seconds at the start of an operation and compares that with the number at the finish, in order to determine the elapsed minutes and seconds. It is *not* necessary to set the real-time clock to the correct time to run this routine. If you want to measure an event that lasts less than a second, you could modify this subroutine to use jiffies instead for 1/60th-second accuracy.

LINE-BY-LINE DESCRIPTION

Line 27410: Get time at start of operation.

Line 27420: Wait for user to press key. This line simulates the operation that you want to time. In your own program, you

should have line 27410 located at the *start* of your operation and then proceed to the following line, 27430, at the point where you wish to measure the time that has elapsed.

Line 27430: Take the current time.

Line 27440: Get value of the time at start.

Line 27450: Get value of the time at finish.

Line 27460: Figure elapsed seconds by subtracting time at start from time at finish.

Line 27470: Figure elapsed minutes by dividing the elapsed seconds by 60.

Line 27480: Figure elapsed seconds by taking the seconds that remain after the elapsed minutes are subtracted from the total seconds elapsed.

Lines 27490–27510: Print results.

YOU SUPPLY

Your program should set TS$ to equal TI$ when you wish to start timing, and then call the subroutine when the end of the timing cycle is over.

SUGGESTED ENHANCEMENTS: Notice that the subroutine will report only elapsed minutes and seconds. Using the routines from Chapter 2, you should be able to modify this one to measure hours as well. Build in a way to handle time that spans a day.

RESULT

Elapsed time is measured.

TIMER

WHAT IT DOES: Sets computer as a timer.

LEVEL: Intermediate

```
100 REM *********
110 REM *       *
120 REM * TIMER *
130 REM *       *
140 REM *********
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    NONE
190 REM  RESULT --
200 REM    TIMER SET
210 REM
220 REM --------------------------------
230 GOSUB 27610
240 END

27600 REM *** SUBROUTINE ***

27610 PRINT "TOTAL TIME TO BE COUNTED :"
27620 PRINT "ENTER HOURS :";
27630 INPUT HR$
27640 H1=VAL(HR$)
27650 PRINT "ENTER MINUTES :";
27660 INPUT MN$
27670 M1=VAL(MN$)
27680 PRINT "ENTER SECONDS :";
27690 INPUT S1
27700 T=(H1*3600)+(M1*60)+S1
27710 IF T>65535 THEN PRINT"SORRY, MUST BE LESS THAN 18
HOURS!":RETURN
27720 SCNCLR
27730 SLEEP T
27740 SOUND 1,1000,10
27750 PRINT "TIME IS UP!"
27760 RETURN
```

HOW TO USE SUBROUTINE

Having the computer signal you at some future time can be a useful function. This subroutine sets the real-time clock to the correct time, then asks what time you want to be alerted. It will then constantly compare the updated current time with the calculated finish time, and when that time is reached, signal.

You are prompted for all the information needed.

LINE-BY-LINE DESCRIPTION

Lines 27610–27690: Get hours, minutes, and seconds to be timed.

Line 27700: Calculate total seconds that will be timed.

Line 27710: If out of SLEEP range, end routine.
Line 27720: Clear screen.
Line 27730: SLEEP for T seconds.
Line 27740: Beep when time is up.
Line 27750: Print message.

YOU SUPPLY

Answer the requests from the prompts. You also might want to call a more "alarming" sound subroutine of your choice at Line 27740 to provide an audible alarm. Several such sound routines are provided in Chapter 7.

SUGGESTED ENHANCEMENTS: Note that the computer can't do any other functions while this subroutine is running, since SLEEP is used. You might want to change it so that a finish hour (say, FH$) is constructed and then constantly compared against the current time. Your line might be something like, IF VAL(TI$) >VAL(FT$) GOTO. . . in your main program. Make this comparison before branching to each new function. Or, you could write a GOSUB line that goes to that line repeatedly during FOR-NEXT or GET loops (use GET A$ instead of GETKEY A$).

RESULT

Commodore 128 signals at end of requested time interval.

COLOR CHECKER

WHAT IT DOES: Displays color bars, to check out video display.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * COLOR CHECKER   *
130 REM *                 *
140 REM *****************
150 REM -------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    NONE
190 REM  RESULT --
200 REM    COLOR BARS DISPLAYED
210 REM
220 REM -------------------------------

230 REM *** INITIALIZE ***

240 GOSUB 27110
250 END

27100 REM *** SUBROUTINE ***

27110 SCNCLR
27120 FOR N1=1 TO 16
27130 COLOR 5,N1
27140 PRINT CHR$(18);
27150 FOR N=1 TO 40
27160 PRINT CHR$(32);
27170 NEXT N
27180 PRINT
27190 NEXT N1
27200 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will provide a quick check of the Commodore 128 video monitor or your color television. Some misadjustments may produce bending horizontal or vertical lines or poor color reproduction.

Running the subroutine will produce parallel, horizontal color bars to check your entire screen. You can then adjust your set or monitor for best color and contrast.

This subroutine demonstrates the use of the COLOR command to change colors on the screen. We have used the CHR$ color codes in some previous subroutines. CHR$ is convenient where you want to use keys that the operator presses to control color, since you can ask the user to press the key he or she wants. For other applications using COLOR may be a better choice, since it is not necessary to build data lines with the CHR$ color codes. The numbers 1–16 can be used instead.

LINE-BY-LINE DESCRIPTION

Line 27110: Clear screen.
Line 27120: Start loop from 1 to 16 colors.
Line 27130: Change character color to the value of N1.
Line 27140: Print the REVERSE ON symbol.
Line 27150: Start loop from 1 to 40 characters, the width of the 40-column screen.
Line 27160: Print a space. Since reverse is on, this will appear as a solid block of the current color.
Line 27170: Repeat for next column.
Line 27180: Move cursor down to next line.
Line 27190: Next color.

YOU SUPPLY

Set adjustments.

SUGGESTED ENHANCEMENTS: Change to provide vertical color bars as well.

RESULT

Better display quality, or rough diagnosis of a problem.

PROGRAM KEYS

WHAT IT DOES: Redefines function keys.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * PROGRAM KEYS   *
130 REM *                *
140 REM ****************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM     NONE
190 REM  RESULT --
```

```
200 REM     FUNCTION KEYS PROGRAMMED
210 REM
220 REM --------------------------------

230 REM *** INITIALIZE ***

240 GOSUB 27810
250 END

27800 REM *** SUBROUTINE ***

27810 PRINT "ENTER KEY TO DEFINE (1-8) :"
27820 GETKEY K$
27830 K=VAL(K$):IF K<1 OR K>8 GOTO 27820
27840 PRINT "ENTER STRING FOR THE KEY :"
27850 GETKEY A$
27860 IF A$=CHR$(13)THEN GOTO 27900
27870 I$=I$+A$
27880 PRINT A$;
27890 GOTO 27850
27900 PRINT
27910 PRINT "END WITH C/R?  (Y/N)"
27920 GETKEY AN$
27930 IF AN$="Y" THEN I$=I$+CHR$(13):GOTO 27950
27940 IF AN$<>"N" GOTO 27920
27950 KEY K,I$
27960 I$=""
27970 RETURN
```

HOW TO USE SUBROUTINE

The Commodore 128 has a much more powerful function key capability than the Commodore 64. With the 64 it was not possible for BASIC 2.0 to assign a string of characters to a key. Instead, the function keys served as any other key: when pressed, they returned a CHR$ code (133–140). Your program could check for any of those keys (as with a GET loop) and branch to a subroutine of your choice that would carry out the actual "function" of the function key.

BASIC 7.0 allows you to assign actual strings to the keys and, in fact, provides a selection of default strings, such as DLOAD" and LIST.

While you may type "KEY,string" from BASIC command mode and assign key definitions, this subroutine may be easier for novices. If you are writing a program with your subroutine library loaded into memory, you need only type GOSUB 27810 to access this module for instant key redefinition. The routine might

also supply you with ideas for redefining keys from within your own program. See the Utility Keys subroutine for an example of a key redefinition program.

NOTE: The *total* length of all the key definitions in force at one time cannot be more than 246 characters.

LINE-BY-LINE DESCRIPTION

Line 27810: Ask for key to be redefined.

Lines 27820–27830: Get response, and check to make sure it is a legal key.

Lines 27840–27860: Get string for the key, stopping when RETURN (CHR$(13)) is pressed.

Lines 27870–27890: Print characters as they are entered, and add them to I$, which stores the key definition.

Lines 27900–27920: Ask if the function key definition should be ended with a CHR$(13) (RETURN) when it is sent.

Line 27930: If yes, add RETURN to the string.

Line 27950: Redefine the key.

Line 27960: Null I$, in case routine is called again later.

YOU SUPPLY

Key definition.

SUGGESTED ENHANCEMENTS: Incorporate subroutine in your own programs so the key definitions you want will be made when the program is run.

RESULT

Keys redefined.

FUNCTION KEYS

WHAT IT DOES: Changes function keys to Commodore 64 mode.

LEVEL: Novice

```
100 REM *****************
110 REM *                 *
120 REM * FUNCTION KEYS *
130 REM *                 *
140 REM *****************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    NONE
190 REM  RESULT --
200 REM    FUNCTION KEYS BEHAVE LIKE
210 REM    COMMODORE 64'S
220 REM
230 REM --------------------------------

240 REM *** INITIALIZE ***

250 GOSUB 28010
280 END

28000 REM *** SUBROUTINE ***

28010 CU=0
28020 FOR N=1 TO 8 STEP 2
28030 CU=CU+1
28040 KEY N,CHR$(132+CU)
28050 KEY N+1,CHR$(136+CU)
28060 NEXT N
28070 RETURN
```

HOW TO USE SUBROUTINE

When you want more complex functions than can be stored in the function keys as strings, it is convenient to use a key-press to trigger a subroutine. As was described in the previous subroutine, with the Commodore 64, when any of the eight function keys are pressed, a single-character string ranging from CHR$(133) to CHR$(140) is returned. Your programs can check for these keys and then branch to subroutines of your choice:

```
100 GETKEY A$
110 A=ASC(A$)
120 IF A<133 or A>140 GOTO 100
130 ON A-132 GOTO 500,600,700,800,900,1000,1100,1200
```

This subroutine returns the Commodore 128 function keys to the single-character strings of the Commodore 64. Note that the eight CHR$ codes chosen were selected because they are otherwise unassigned. If you wanted to assign the characters A–H to the

keys, you could have; however, pressing the main keyboard keys A–H would produce exactly the same effect.

LINE-BY-LINE DESCRIPTION

Line 28010: Initialize count to 0.
Line 28020: Start loop from 1 to 8, stepping by twos.
Line 28030: Increment counter.
Line 28040: Redefine Key N as CHR$(132+CU).
Line 28050: Redefine Key N+1 as CHR$(136+CU).
Line 28060: Repeat.

YOU SUPPLY

No user input required.

SUGGESTED ENHANCEMENTS: Add programs to perform functions when keys are pressed.

RESULT

Commodore 128 function keys redefined to Commodore 64 mode.

UTILITY KEYS

WHAT IT DOES: Examples of key redefinition files.

LEVEL: Novice

```
100 REM ****************
110 REM *                *
120 REM * UTILITY KEYS   *
130 REM *                *
140 REM ****************
150 REM -----------------------------
160 REM      ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM     NONE
```

```
190 REM  RESULT --
200 REM    EXAMPLE PROGRAMMING AID
210 REM
220 REM ----------------------------

28100 REM *** SUBROUTINE ***

28110 KEY 1,"RENUMBER 100,10,1"+CHR$(13)
28120 KEY 6,CHR$(145)+CHR$(145)+CHR$(145)+CHR$(145)+CHR$(13)
28130 KEY 7,CHR$(147)+"LIST"+CHR$(13)
```

HOW TO USE SUBROUTINE

If you use given sets of function key definitions frequently, you can write them as a program and store the program for use as you wish. For example, in the writing of this book several key definitions were used repeatedly. Examples of several of them are described below.

LINE-BY-LINE DESCRIPTION

Line 28110: As subroutines were written, it was sometimes necessary to make some additional space between line numbers. This definition caused the Commodore 128 to renumber all the program lines, with a starting line number of 100 and an increment of 10, starting with the first program line. The CHR$(13) tacked on the end invoked the renumbering as soon as the F1 key was pressed.

Line 28120: As each finished subroutine was saved to disk, the DSAVE step was repeated two times to provide backup copies of the routines. Instead of the author's retyping the DSAVE command and the file name, this key was pressed. It moves the cursor up four lines (back to where the original DSAVE command was) and adds a CHR$(13) to reinvoke the command on the line to where the cursor has been moved. This works only if you have not moved the cursor since the last SAVE and if no error was generated by the SAVE.

Line 28130: To view each subroutine, it was often necessary to move the cursor down to a clear area of the screen and then press the F7, LIST function key. Instead, this key was defined. It clears the screen first (so the cursor does not have to be moved) and then LISTS the entire program.

YOU SUPPLY

Substitute your own key definitions.

SUGGESTED ENHANCEMENTS: Come up with programming shortcuts of your own.

RESULT

Keys redefined to a set of utility functions.

CURSOR MOVER

WHAT IT DOES: Duplicates the LOCATE function of IBM BASIC.

LEVEL: Intermediate

```
100 REM ****************
110 REM *                *
120 REM * CURSOR MOVER   *
130 REM *                *
140 REM ****************
150 REM --------------------------------
160 REM         ++ VARIABLES ++
170 REM  USER SUPPLIED --
180 REM    ROW:     ROW FOR CURSOR
190 REM    COL:     COL FOR CURSOR
200 REM  RESULT --
210 REM    CURSOR MOVED TO ROW, COL
220 REM
230 REM --------------------------------

240 REM *** INITIALIZE ***

250 FOR N=1 TO 80
260 R$=R$+CHR$(29)
270 IF N<26 THEN D$=D$+CHR$(17)
280 NEXT N
290 WIDE=40
300 COL=3:ROW=10
310 SCNCLR
320 GOSUB 28210
330 PRINT "HERE I AM IN ROW ";ROW;" COLUMN ";COL;"!"
340 END

28200 REM *** SUBROUTINE ***
```

```
28210 PRINT CHR$(19);
28220 PRINT LEFT$(R$,COL);
28230 PRINT LEFT$(D$,ROW);
28240 RETURN
```

HOW TO USE SUBROUTINE

Some BASICs, such as that used in the IBM Personal Computer, allow moving the cursor to any location on the screen with a LOCATE (row,col) command.

This subroutine duplicates that function for the 40- or 80-column screen. Prior to calling it, you should define ROW and COL. Your program should define WIDE as the width of your screen, either 40 or 80 columns. If you exceed allowable values on the 80-column screen, the cursor moves to its limits and no farther. With a 40-column screen the cursor will wrap around to the next line, up to the last column in that second row. ROW and COL can be computed values, if you wish, but see that they don't fall outside the limits.

LINE-BY-LINE DESCRIPTION

Lines 250–280: Define R$ as 80 CURSOR RIGHT characters. Define D$ as 25 CURSOR DOWN characters.

Lines 290–300: Define ROW, COL, and screen width, WIDE. Your program will define WIDE once but redefine ROW and COL as you move the cursor around on the screen.

Line 310: Clear screen.

Line 320: Access the subroutine.

Line 330: Show result.

Line 28210: Move cursor to HOME position.

Line 28220: Move cursor right the number of columns in COL by taking the LEFT$ portion of R$ up to COL.

Line 28230: Move cursor down the number of lines in ROW by taking the LEFT$ portion of D$ up to ROW.

Line 28240: Return.

YOU SUPPLY

Definition for screen width, ROW and COL.

SUGGESTED ENHANCEMENTS: Add lines to keep cursor from moving beyond the 40-column point with a 40-column screen. Example:

```
28215 IF COL>40 AND WIDE=40 THEN COL=40
```

RESULT

Cursor moved to ROW and COL specified by user.

PROGRAM TRANSFER

WHAT IT DOES: Allows sending a program listing out the RS-232 interface, if one is installed.

LEVEL: Intermediate

```
100 REM ********************
110 REM *                    *
120 REM * PROGRAM TRANSFER *
130 REM *                    *
140 REM ********************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM          NONE
190 REM  RESULT --
200 REM    LISTING DIRECTED TO
210 REM    RS-232 PORT
220 REM
230 REM ----------------------------

28300 REM *** SUBROUTINE ***

28310 OPEN 2,2,3,CHR$(38)+CHR$(160)
28320 CMD2
28330 LIST
28340 PRINT#2
28350 CLOSE 2
28360 PRINT"TRANSFER COMPLETE"
```

HOW TO USE SUBROUTINE

Because the Commodore 128's BASIC is fairly compatible with that of other computers, it may be desirable to send a program listing to another machine, even though the two have incompatible disk formats. If the Commodore 128 is equipped with an inexpensive RS-232 interface and the other computer has the same, then a cable, a null modem adapter, and this subroutine are nearly all that are needed. The other computer should have a terminal program that will allow dumping the transmitted file to disk or tape.

Be certain to include the null modem adapter, available from most computer stores, when communication is directly between two computers with no modem in between. Otherwise, each will be sending to the other's SEND line and trying to receive from the connected computer's RECEIVE line.

Simply load the program you want to transmit, and type in the lines in this subroutine as one long line before hitting return. Your baud rate will be set for 300, even parity, and a 7-bit word.

This routine was actually used in the preparation of this book. The program listings were sent from the author's Commodore 128 to another computer used for word processing. However, a key was redefined to provide the program lines listed here. After a given routine was loaded, pressing the function key initiated the transmission.

LINE-BY-LINE DESCRIPTION

Line 28310: OPEN the RS-232 line and set for 300 baud, even parity, 7-bit word.

Line 28320: Redirect screen output to the RS-232 line.

Line 28330: LIST the program. The listing goes out the RS-232 instead of to the screen.

Lines 28340–28350: Return status to normal.

Line 28360: Notify that transmission is complete.

YOU SUPPLY

Program to transmit.

SUGGESTED ENHANCEMENTS: Incorporate features from the next subroutine, Terminal, to allow for two-way transmissions.

RESULT

Program listing sent out RS-232 port.

TERMINAL

WHAT IT DOES: BASIC dumb terminal program.

LEVEL: Intermediate

```
100 REM ************
110 REM *          *
120 REM * TERMINAL *
130 REM *          *
140 REM ************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    KEYBOARD INPUT
190 REM  RESULT --
200 REM    COMMUNICATE WITH OTHER
210 REM    COMPUTER AT 300 BAUD,
220 REM    EVEN PARITY, 7-BIT WORD
230 REM
240 REM ----------------------------

28500 REM *** SUBROUTINE ***

28510 OPEN 2,2,3,CHR$(38)+CHR$(160)
28520 GET #2,A$
28530 GET B$
28540 IF B$<>"" THEN PRINT#2,B$;
28550 IF B$=CHR$(95) THEN GOTO 28610
28560 GET #2,C$
28570 A=ASC(C$)
28580 IF A>90 THEN C$=CHR$(A-32)
28590 PRINT B$;C$;
28600 GOTO 28530
28610 CLOSE 2
```

HOW TO USE SUBROUTINE

BASIC is just about fast enough for the Commodore 128 to communicate at 300 baud if nothing fancy is attempted. This

subroutine will fetch characters from the RS-232 line and send keyboard characters out that interface. Anything sent or received is echoed to the screen. Lowercase from the other computer is translated to uppercase.

LINE-BY-LINE DESCRIPTION

Line 28510: OPEN RS-232 channel for 300 baud communications.

Line 28520: Get from channel number 2, a character, if available.

Line 28530: GET a character from the keyboard, if a key is depressed.

Line 28540: If a key was depressed, print that character to the screen.

Line 28550: If user enters CHR$(95) (back arrow), cease communications.

Line 28560: Get character from RS-232, if available.

Line 28570: Determine ASCII value of key received.

Line 28580: If key was lowercase, reduce to uppercase.

Line 28590: Print keyboard or character from RS-232 to screen.

Line 28600: Go back and get more characters.

Line 28610: CLOSE the RS-232 channel.

YOU SUPPLY

No user changes needed.

SUGGESTED ENHANCEMENTS: Add routine to upload a file.

RESULT

Dumb terminal communications in BASIC.

9 BITS AND BYTES

This section is for those at the threshold of advanced programming. All but one of the routines in this chapter deal with viewing and manipulating the individual bits within single bytes in your computer's memory.

As you know, each memory location stores a single, 8-bit byte. The binary numbers look like this:

```
10110111
```

In many cases the value of this whole byte is of use to you. For example, in character memory, when you find a "1010001"

(81 decimal), you know that an uppercase "Q" has been printed there. Using a full byte allows you to have a total of 256 combinations in that location and therefore 256 different characters.

However, some functions do not have that many possibilities. A feature may be on or off, for example. You could store a "1" in that location (00000001 in binary) if the feature is on, and a "0" (00000000 in binary) if it is off. You can see, though, that the other seven bits will never be used.

The Commodore computer makes multiple use of many memory locations by using individual bits to represent different conditions. It is necessary, then, to look at one bit within a byte to see whether a feature is on or off. Similarly, when you want to change that condition, you may need to POKE ONLY that bit and leave the others, which pertain to other features, unchanged.

The Peek Bit subroutine filters out the undesired bits by using a technique known as Boolean logic. Boolean math compares each bit of one byte with the corresponding bit of another byte. The result depends on what type of operator is used, the most common being AND, OR, and NOT. With the AND operation, if both bits are 1, the result is 1; all other comparisons produce a value of 0. The OR operation produces a 1 if either bit is 1. NOT complements each bit.

Boolean math is discussed in more detail shortly. First you need to know more of how AND works. The particular bit or bits looked at depends on the number you choose to AND with. You will remember in the Commodore 64-compatible joystick routines in Chapter 5 various numbers were ANDed with a PEEK to determine the status of a given joystick switch bit. In some cases, you may want to know about all of the first four bits (reading from right to left, as is the convention with numbers) of a byte, so you AND with 15, which is 00001111 in binary, to "mask" the last four bits. Here are a few examples:

```
MEMORY BYTE:          11010001
AND with 15:          00001111
RESULT                00000001


MEMORY BYTE:          01100001
AND with 15:          00001111
RESULT                00000001
```

You'll see by following along the columns that the result equals 1 only when the corresponding bit in both the color memory byte and 15 equal 1. Since the second (left-hand) four bits of the binary equivalent of 15 decimal are always 0, the result will always be 0. Since the first (right-hand) four bits of 15 are all 1's, the result will be 1 only if there is also a 1 in the memory byte.

The OR and NOT operators are two of the other more frequently used Boolean tools. Whereas AND produces a 1 after each bit-to-bit comparison only when both bits are 1, OR produces a 1 if either bit is 1. For example:

```
Original byte:          10110110  OR
Comparison byte:        01100011
Result:                 11110111
```

NOT produces the opposite of the value used: IF NOT A=1 will produce a 0 (false) value if A does equal 1.

There are a number of other Boolean operators, including exclusive OR (XOR), but none of these is used in this book. What these subroutines let you do is manipulate individual bits in order to set certain registers that may not require an entire byte. Rather than POKing a number into a memory location and changing the contents of bits that do not concern you, use the "soft" POKing routines presented here to alter only the desired bit.

One of the subroutines in this section will allow PEEKing at any given bit within a byte. Another will set any chosen bit to 1, turning a feature "on." A third will set any bit to 0, turning that feature "off." What if you don't care whether the bit is on or off but would like to set it to the reverse condition? In computer programming this is known as a *toggle*. Hitting the switch one time turns the feature on or off, depending on its previous condition. Hitting it again does the reverse. The Reverse Bit subroutine will toggle any bit you like. Another routine, Bit Displayer, will show the status of all the bits in a byte. In effect, it translates the byte into binary.

Another subroutine rounds off numbers to any specified degree of precision. While not dealing with bits, it is included in this section as a general number crunching utility.

PEEK BIT

WHAT IT DOES: Looks at status, 0 or 1, of any selected bit in a given byte.

LEVEL: Advanced

```
100 REM ************
110 REM *          *
120 REM * PEEK BIT *
130 REM *          *
140 REM ************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM    ADDRESS: LOCATION TO PEEK
190 REM    BIT:     BIT TO EXAMINE
200 REM   RESULT --
210 REM    V:       VALUE OF BIT
220 REM
230 REM ------------------------

240 REM *** INITIALIZE ***

250 ADDRESS=36879
260 BIT=5
270 GOSUB 29010
280 PRINT "VALUE IN BIT ";BIT;"=";V
290 END

29000 REM *** SUBROUTINE ***

29010 BIT=8-BIT
29020 P=BIT-1
29030 V=(PEEK(ADDRESS)AND(2^P))/(2^P)
29040 BIT=8-BIT
29050 RETURN
```

HOW TO USE SUBROUTINE

The Commodore 128 gets maximum mileage out of its RAM locations by using many for multiple purposes. A given register has eight bits, making up its byte. The status of one bit might be used to indicate whether a certain feature is on or off. Another bit in the same byte might be used to toggle some entirely different function.

Accordingly, it is useful to look at just one bit in a byte, to

see its status. Your program may take some action based on what is found, that is, "IF V=O THEN PRINT "THE FEATURE IS OFF."

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 250: Define ADDRESS to PEEK.
Line 260: Define BIT to look at.
Lines 270–290: Access the subroutine and print out results.
Line 29010: Change BIT number. Although we count bits from right to left, this routine processes them from left to right.
Line 29020: Determine number to AND with byte.
Line 29030: AND byte with P to determine status of the bit.
Line 29040: Change BIT value back.

YOU SUPPLY

Define BIT as the bit, 1–8 counting from the right, that you want to examine, and ADDRESS as the memory location to be PEEKed. V will indicate whether the bit is on or off, by equaling either 1 or 0.

SUGGESTED ENHANCEMENTS: None.

RESULT

Status of bit displayed.

BIT DISPLAYER

WHAT IT DOES: Shows pattern of all eight bits within a byte. Converts the decimal value to binary.

LEVEL: Advanced

```
100 REM *****************
110 REM *                 *
120 REM * BIT DISPLAYER *
130 REM *                 *
140 REM *****************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     ADDRESS: MEMORY BYTE
190 REM              TO DISPLAY
200 REM  RESULT --
210 REM     BYTE$:   BIT PATTERN
220 REM
230 REM ----------------------------

240 REM *** INITIALIZE ***

250 ADDRESS=36879
260 PRINT
270 GOSUB 29110
280 PRINT"ADDRESS: ";ADDRESS
290 PRINT TAB(4)BYTE$
300 END

29100 REM *** SUBROUTINE ***

29110 BYTE$=""
29120 PRINT TAB(4)"";
29130 FOR N=7 TO 0 STEP-1
29140 V=(PEEK(ADDRESS)AND(2^N))/(2^N)
29150 G$=MID$(STR$(V),2)
29160 BYTE$=BYTE$+G$
29170 NEXT N
29180 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will display all of the bits within a byte. Each position will be indicated by a 1 or a 0.

You could also use this subroutine to provide a quick way of converting a number from decimal (in the range 0 to 255 only) to binary. Simply POKE the number to an unused memory location and then immediately call this subroutine to PEEK that address. Quite a roundabout way of performing the task, but useful if you are writing software that you deliberately want to be difficult to change, such as for protection purposes.

This subroutine will also serve as a means of converting decimal number smaller than 255 to binary. Simply substitute

your variable for PEEK(ADDRESS) and define the variable as the decimal number you want to convert.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 250: Define address to be PEEKed.
Lines 260–300: Access the subroutine and print result.
Line 29110: Null any previous value of BYTE$.
Line 29120: Provide TAB to print result.
Lines 29140–29150: Repeat through each bit of byte, AND each bit with the next highest power of 2, and store the result in G$, which will store the on/off status of each bit. Then add G$ to BYTE$.

YOU SUPPLY

You must define ADDRESS as the memory location, in decimal, that you want to PEEK. The subroutine returns BIT$, which is a representation of all the bits within that byte.

SUGGESTED ENHANCEMENTS: None.

RESULT

All bits within a byte are displayed.

BIT TO ONE

WHAT IT DOES: Soft POKEs any desired bit within a byte so that it now has the value of 1, without changing any other bits.

LEVEL: Advanced

```
100 REM **************
110 REM *            *
120 REM * BIT TO ONE *
130 REM *            *
140 REM **************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM   ADDRESS: LOCATION TO POKE
190 REM   BIT:     BIT TO CHANGE TO ONE
200 REM  RESULT --
210 REM    BIT CHANGED TO 1
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 ADDRESS=36878
270 BIT=3
280 GOSUB 29210
290 END

29200 REM *** SUBROUTINE ***

29210 BIT=8-BIT
29220 POKE ADDRESS,PEEK(ADDRESS)OR(2^BIT)
29230 BIT=8-BIT
29240 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will take any bit within a byte and change its value to 1, regardless of what it was before. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the Commodore 128.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 260: Define ADDRESS to PEEK and POKE.
Line 270: Define BIT to change to a value of 1.
Line 29220: POKE BIT to 1.

YOU SUPPLY

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1–8, that you want changed to a value of 1.

SUGGESTED ENHANCEMENTS: None.

RESULT

Bit within a byte is changed to 1.

BIT TO ZERO

WHAT IT DOES: Soft POKEs any desired bit within a byte so that it now has the value of 0, without changing any other bits.

LEVEL: Advanced

```
100 REM ***************
110 REM *               *
120 REM * BIT TO ZERO *
130 REM *               *
140 REM ***************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM   SUPPLIED BY USER --
180 REM    ADDRESS: LOCATION TO POKE
190 REM    BIT:     BIT TO CHANGE TO ZERO
200 REM   RESULT --
210 REM     BIT CHANGED TO ZERO
220 REM
230 REM ----------------------------

240 REM *** INITIALIZE ***

250 ADDRESS=36879
260 BIT=3
270 GOSUB 29310
280 END

29300 REM *** SUBROUTINE ***

29310 BIT=8-BIT
29320 POKE ADDRESS,PEEK(ADDRESS)AND(255-(2^BIT))
29330 BIT=8-BIT
29340 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will take any bit within a byte and change its value to 0, regardless of what it was before. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the Commodore 128.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 250: Define ADDRESS to PEEK and POKE.
Line 260: Define BIT to change to a value of 0.
Line 29320: POKE BIT to 1.

YOU SUPPLY

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1–8, that you want changed to a value of 0.

SUGGESTED ENHANCEMENTS: None.

RESULT

Bit within a byte is changed to 0.

REVERSE BIT

WHAT IT DOES: Soft POKEs any desired bit within a byte so that it now has the opposite value without changing any other bits.

LEVEL: Advanced

```
100 REM ***************
110 REM *               *
120 REM * REVERSE BIT *
130 REM *               *
140 REM ***************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM   ADDRESS: LOCATION TO POKE
190 REM   BIT:     BIT TO REVERSE
200 REM  RESULT --
210 REM    BIT CHANGED TO OPPOSITE
220 REM
230 REM ----------------------------

240 REM *** INITIALIZE ***

250 ADDRESS=36878
260 BIT=3
270 GOSUB 29410
280 END

29400 REM *** SUBROUTINE ***

29410 BIT=8-BIT
29420 M=1-(PEEK(ADDRESS)AND(2^BIT))/(2^BIT)
29430 POKE ADDRESS,PEEK(ADDRESS)AND(255-(2^BIT))OR(M*(2^BIT))
29440 BIT=8-BIT
29450 RETURN
```

HOW TO USE SUBROUTINE

This subroutine will take any bit within a byte and change that value to the opposite of what it was before. If the bit was 1, it will be changed to 0. A 0 bit will be given a value of 1. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the Commodore 128. Using this subroutine, it is not necessary to know whether the feature is on or off. The routine will change it to the other status automatically.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 250: Define ADDRESS to PEEK and POKE.
Line 260: Define BIT to reverse.
Lines 29420–29430: Find out value of the bit, then reverse that, using OR.

YOU SUPPLY

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1–8, that you want changed to reverse in value.

SUGGESTED ENHANCEMENTS: None.

RESULT

Bit within a byte is reversed.

BINARY TO DECIMAL

WHAT IT DOES: Changes binary number to decimal equivalent.

LEVEL: Advanced

```
100 REM ******************
110 REM *                  *
120 REM * BINARY/DECIMAL *
130 REM *                  *
140 REM ******************
150 REM ---------------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM   A$:    BINARY NUMBER AS STRING
190 REM  RESULT --
200 REM   A:     DECIMAL EQUIVALENT
210 REM
220 REM ---------------------------------
230 SCNCLR
240 PRINT "ENTER BINARY NUMBER TO CONVERT":INPUT A$
250 FOR N=1 TO LEN(A$)
260 T$=MID$(A$,N,1)
```

```
270 IF T$="0" OR T$="1" GOTO 310
280 PRINT " NOT A BINARY NUMBER!"
290 PRINT
300 GOTO 240
310 GOSUB 29510
320 PRINT A$;"=";A
330 END
340 A=0

29500 REM *** SUBROUTINE ***

29510 FOR N=1 TO LEN(A$)
29520 P=LEN(A$)-N
29530 A=A+2^P*VAL(MID$(A$,N,1))
29540 NEXT N
29550 RETURN
```

HOW TO USE SUBROUTINE

Several of the subroutines in this book, and many more that you will prepare, require supplying decimal equivalents of binary numbers. For example, producing programmed character sets involves setting each bit of a byte either on or off depending on the desired status of the equivalent picture element. Once the binary number has been "designed," the user needs the decimal equivalent for the appropriate POKE statement.

This routine will calculate the decimal numbers for you. Just enter the binary number when asked. The routine will check to see that ONLY 1's and 0's have been entered, then figure the result.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Lines 230–240: Ask user for binary number to convert.

Lines 250–300: Check for presence of illegal characters.

Line 310: Access subroutine.

Line 320: Print result.

Lines 29510–29540: Look at each binary character, and raise any 1's to the power of 2 indicated by its position within the byte.

YOU SUPPLY

You must enter the binary number to be converted.

SUGGESTED ENHANCEMENTS: None.

RESULT

- Binary number converted to decimal.

ROUNDER

WHAT IT DOES: Rounds positive number, and cuts off after desired number of decimal places.

LEVEL: Intermediate

```
100 REM ***********
110 REM *         *
120 REM * ROUNDER *
130 REM *         *
140 REM ***********
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM      A:   NUMBER TO BE ROUNDED
190 REM      P:   DIGITS DESIRED TO
200 REM           RIGHT OF DECIMAL POINT
210 REM  RESULT --
220 REM      B:   ROUNDED VALUE
230 REM
240 REM ----------------------------

250 REM *** INITIALIZE ***

260 A=55.534
270 P=2
280 GOSUB 29620
290 PRINT B
300 END

29610 REM *** SUBROUTINE ***

29620 C=A+5.5*10^-(P+1)
29630 B=INT(C*10^P)/10^P
29640 RETURN
```

HOW TO USE SUBROUTINE

The Commodore 128 is sometimes a great deal more accurate than you need. For example, your car may get 24.3459121 miles per gallon, but you would be happy to know that it is close to 24.3. This subroutine can be used to produce the desired degree of precision while still rounding the numbers so that the figure is as accurate as the significant digits reflect.

NOTE: The caret symbol (^) in the program listing stands for the up-arrow key, located between RESTORE and "*" on your Commodore 128 keyboard.

LINE-BY-LINE DESCRIPTION

Line 260: Define number to be rounded.
Line 270: Define number of digits to right of decimal desired.
Line 280: Access the subroutine.
Line 290: Print results.
Line 29620: Add rounding factor.
Line 29630: Take integer portion of number multiplied by 10 raised to P power, and divide that by 10 raised to P power.

YOU SUPPLY

You should define A to be the number to be rounded. P will equal the number of digits to the right of the decimal point that you want. The subroutine will return B, the rounded value. If B has a fractional decimal part that ends in 0, the 0 will not be printed, even though that many decimal places have been requested. For example, if two decimal places are desired, 55.344 and 55.399 will be returned as 55.34 and 55.4 respectively.

SUGGESTED ENHANCEMENTS: None.

RESULT

Number rounded as specified.

PRIME NUMBERS

WHAT IT DOES: Finds prime numbers.

LEVEL: Intermediate

```
100 REM *****************
110 REM *                 *
120 REM * PRIME NUMBERS *
130 REM *                 *
140 REM *****************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM     U:   MAX TO SEARCH
190 REM  RESULT --
200 REM    LAST:  LAST PRIME FOUND
210 REM    P:     NEXT TO CHECK
220 REM ----------------------------

230 REM *** INITIALIZE ***

240 INPUT "HOW HIGH TO SEARCH";U
250 GOSUB 29710
260 END

29700 REM *** SUBROUTINE ***

29710 DIM PR(5000)
29720 PR(1)=3
29730 PR(2)=5
29740 LAST=2
29750 P=7
29760 NU=0
29770 IF PR(T)<SQR(P) THEN T=T+1:GOTO 29770
29780 FOR N=1 TO T
29790 G=P/PR(N)-INT(P/PR(N)):IF G=0 THEN NU=1
29800 NEXT N
29810 IF NU=1 THEN GOTO 29850
29820 LAST=LAST+1
29830 PR(LAST)=P
29840 PRINT P
29850 P=P+2
29860 IF P=>U THEN RETURN
29870 GOTO 29760
```

HOW TO USE SUBROUTINE

This subroutine is just for fun but may also be of some use to those who need to locate a list of prime numbers. It will generate

a list of prime numbers smaller than the number you specify. If you have the patience to wait for more than 500, enlarge the array to make room.

LINE-BY-LINE DESCRIPTION

Line 240: Ask how high to search.

Line 250: Access the subroutine.

Line 29710: DIMension an array to hold the first 5000 prime numbers.

Lines 29720–29730: Tell the subroutine the first two prime numbers.

Line 29740: Define last number checked as 2.

Line 29770: See if the last prime found is less than the square of the current number being checked. Only numbers less than the square root of P will be checked.

Lines 29780–29800: See if number is evenly divisible.

Line 29810: If even divisor was found, try next number.

Line 29820: If no even divisor is found, then number is prime.

Lines 29830–29840: Store new prime in array and print results to screen.

Line 29850: Increment P to look for next prime.

Line 29860: Check to see if upper desired limit reached.

Line 29870: Otherwise, repeat.

NUMBER SORT

WHAT IT DOES: Sorts group of numbers by size.

LEVEL: Intermediate

```
100 REM ***************
110 REM *               *
120 REM * NUMBER SORT   *
130 REM *               *
140 REM ***************
150 REM ----------------------------
160 REM       ++ VARIABLES ++
170 REM  SUPPLIED BY USER --
180 REM    NU:    NUMBER OF ITEMS SORTED
190 REM    US(N): ARRAY WITH ITEMS
200 REM  RESULT --
210 REM     SORTED ARRAY
220 REM
230 REM ----------------------------

240 REM *** INITIALIZE ***

250 NU=10
260 DIM US(NU):GOSUB 29610
270 FOR N=1 TO NU
280 PRINT US(N)
290 NEXT N
300 END

29600 REM *** SUBROUTINE ***

29610 FOR ITEM=1 TO NU
29620 PRINT"ENTER #";ITEM
29630 INPUT US(ITEM)
29640 NEXT ITEM
29650 FOR N=1 TO NU
29660 FOR N1=1 TO NU-N
29670 A=US(N1)
29680 B=US(N1+1)
29690 IF A<B THEN GOTO 29720
29700 US(N1)=B
29710 US(N1+1)=A
29720 NEXT N1
29730 NEXT N
29740 RETURN
```

HOW TO USE SUBROUTINE

Sorting a list of numbers is a common need for many programs. Checking account files and other groups of numbers often have to be sorted to be most useful. This routine is a simple bubble sort, which will sort any group of numbers that have been loaded into an array, US(n).

Although as written the subroutine asks the user to enter the number list from the keyboard, any means can be used to load the array. The file may also be read from disk or tape, for example,

using the bubble sort routines presented in this book. The bubble sort is so called because each entry in the array is examined and then allowed to rise up past the one below until it encounters a "smaller" item. Numeric sorts are easier to understand than string sorts, because simple number comparisons are used. That is, 1237 is always larger than 32.6 and smaller than 7844. Gradually, each member of the list "floats" up to its proper place in the array.

While such sorts are not very fast, with small lists of, say, 30 or 40 items, the speed is satisfactory. This routine is basically the same as the String Sort presented in Chapter 4.

LINE-BY-LINE DESCRIPTION

Line 250: Define NU, the number of units in the array to be sorted.

Line 260: DIMension the array to proper size.

Lines 270–290: Print results.

Lines 29610–29640: User enters each array item in random order. A disk or tape file read routine could be substituted for these lines to sort an existing string file.

Line 29650: Start loop from 1 to the number of items to be sorted.

Line 29660: Start a nested loop from 1 to 1 less than the number of items to be sorted.

Line 29670: Make A equal to the N1th item of the array.

Line 29680: Make B equal to the item following A in the array.

Line 29690: If the "higher" element, A, is already smaller than B, then B remains where it is, and the inner loop steps off the next value of N1.

Lines 29700–29710: If B is smaller than A, then the two numbers are swapped, with B moving ahead one element and A$ being pushed down one.

Lines 29720–29730: The inner and outer loops are incremented.

YOU SUPPLY

You should define NU, the number of items to be sorted, as well as supply the data for the array, US(n).

SUGGESTED ENHANCEMENTS: None

RESULT

List of numbers is sorted by size.

GLOSSARY

Algorithm: A formula or method for performing a given task, such as MPG = MILES/GALLNS.

Alphanumeric: Characters that are letters, numbers, punctuation marks, or other symbols, as opposed to graphics or control characters. Alphanumerics include the upper- and lowercase alphabet, as well as the digits 0 to 9 and common punctuation symbols.

AND: Boolean operator that compares each bit of a byte with the corresponding bit in another byte and produces a 1 if both are equal to 1.

Append: To add to the end of, as to append one file onto another.

ory location contains something. If it is not meaningful information placed there by the computer or user, it is termed garbage.

Increment: To increase the value of a variable by 1. This is also commonly used as a verb to denote increasing a variable by any amount, such as "to increment by 4."

Initialize: To set variables to a desired beginning value at the start of a program or at the beginning of a subroutine. For example:

```
10 B=0
20 INPUT A
30 B=B+A
40 PRINT B
50 GOTO 20
```

You would want to initialize B, as in line 10, each time the subtotal should be eliminated and the addition started from 0 again.

Jiffie: A 1/60th-second interval used by the Commodore 128 to keep track of elapsed time.

Modem: Modulator-demodulator. A device that converts the Commodore 128's signals to sounds that can be transmitted over telephone lines. The modem also receives sounds and converts them back for the Commodore 128 to use.

Monitor: The television-like device used to display video information.

Null modem: An adapter plug or cable that reverses the SEND and RECEIVE lines of two RS-232 serial interface devices. It enables two computers to be wired directly together to communicate without one computer's SEND signals being sent to the SEND lines of the other and RECEIVE trying to RECEIVE from the other.

Nybble: Four bits; half a byte. Often used when a feature has only 16 possible conditions and therefore can be expressed in four bits instead of the full eight in a byte. The rest of the byte (the other nybble) can be used by the computer for other data registers or left as random garbage.

Offset: A way of addressing memory through the use of a relative address rather than an absolute address. If a certain

memory block is located between 30000 and 31000, we can POKE the first location in that block by either of the two methods following:

```
10 POKE 30001,X
```

```
10 OFFSET=30000
20 POKE OFFSET+1,X
```

The second method is often clearer and can also be used when the memory location defined by OFFSET can vary.

OR: A Boolean operator that is used to compare one byte with another on a bit-for-bit level. If either a bit or the corresponding bit in the other byte is 1, OR will produce a 1 as the result.

Oscillator: An electronic device that, in the Commodore 128, produces a sound when the proper POKEs are performed to the volume and sound registers.

Parallel: A method of transferring data an entire bit at a time by sending each of the eight bits along a separate parallel address line simultaneously. Serial transfer, on the other hand, transmits each of the eight bits one at a time.

Port: One of the "windows" used by the Commodore 128 to talk to the outside world. The joysticks send information to the computer through a port.

Prompt: A message to the computer user asking for information. The following INPUT statement includes a prompt.

```
10 INPUT "ENTER YOUR NAME";A$
```

Pseudo-random: Numbers that appear to be random but that are actually taken from a very long list of numbers. The list is so long that it takes a great deal of time before it repeats, and since the computer usually starts at a different position in the list each time, the series seems to be different.

Random access: A method of getting data, either from memory or from disk, that allows going directly to the information required and using it, without accessing any of the other information in the file or memory.

Real-time clock: The built-in clock in the Commodore 128 that keeps track of elapsed time since the computer was turned on or since the clock was last reset by the user.

Register: A location storing a status of some type. Some types of registers are located in the Commodore 128's microprocessor and can be accessed only through machine language. Some memory locations in the Commodore 128 perform a registerlike function, telling the computer whether a certain feature is on or off, or telling the volume of a sound oscillator, or some other status.

RS-232: A serial interface device that allows the Commodore 128 to communicate with devices like printers or modems one bit at a time.

Sequential: A serial file access method in which each piece of information is stored after another and must be written or accessed in that fashion.

Serial: Sequential data storage or transfer.

String delimiter: A character that the computer recognizes as the "end" of a given string input. The most common are commas and quotation marks.

String variable: A variable that can store alpha information only. Strings can include numbers, punctuation marks, and graphics, but the computer recognizes them only as characters, not as values.

Subroutine: A program module that performs a specific task, called through the GOSUB statement and ending with RETURN, which directs program control back to the instruction following the GOSUB.

Toggle: A feature that can be either on or off is sometimes "toggled" between the two, like a light switch.

Upload: To store a file from disk or tape in the Commodore 128's memory buffer and then send it through telecommunications to another computer, which can then write it to tape or disk for permanent storage (downloading).

Voice: One of three oscillators in the Commodore 128 that produce sounds.

INDEX

C

D

E

F

G

N

O

P

R

S

T

V

W

ABOUT THE AUTHOR

A former associate editor of RUN, a leading publication for Commodore 128 users, David D. Busch has written seven books about the Commodore line of computers. He has also published more than 300 articles on computer-oriented topics for *Personal Computing, Creative Computing, Interface Age,* and a dozen other publications. Busch is also a contributing editor and monthly columnist for five of those magazines, and *The Commodore 128 Subroutine Library* is his eighteenth book. *Sorry About the Explosion: A Humorous Guide to Computers,* a book of computer humor by Busch, was a nominee for best computer-oriented fiction of 1985 in the Computer Press Association competition.

A full-time writer since 1965, Busch's career has included work as a newspaper reporter-photographer, a college sports information director, a photo-posing instructor for a Barbizon-affiliated modeling agency, and an account executive for a New York-based public relations firm. His articles have appeared in publications as diverse as *Adam, Petersen's PhotoGraphic, Income Opportunities,* and *Writer's Yearbook.*

AVAILABLE NOW

FOR YOUR COMMODORE 128

Award-Winning Software from Bantam Electronic Publishing

Bantam's sophisticated, state-of-the-art interactive fiction will intrigue and challenge you. Try:

THE FOURTH PROTOCOL
and
SHERLOCK HOLMES IN "ANOTHER BOW"

You can explore the uncharted territories of your own mind with this fascinating, innovative pair of programs (on one disk):

KNOW YOUR OWN I.Q./ KNOW YOUR OWN PERSONALITY

Younger computer enthusiasts will spend hours with Bantam's absorbing MicroWorkshop Series and Choose Your Own Adventure® Series—programs that make learning fun:

ROAD RALLY U.S.A
(Ages 10 and up)
CREATIVE CONTRAPTIONS
(Ages 7 and up)
FANTASTIC ANIMALS
(Ages 4 to 9)
ESCAPE (CYOA)
(Ages 10 and up)
and
THE CAVE OF TIME (CYOA)
(Ages 10 and up)

Bantam software is available at computer stores, book stores, and anywhere that Commodore software is sold, or call Bantam direct at 800-223-6834 ext. 479 and order them. (N.Y. & N.J. residents call 212-765-6500 ext. 479.)

Commodore Software from Bantam.

BANTAM SOFTWARE™

The Commodore 128 Subroutine Library

Subroutine books for BASIC have been around as long as personal computers. Subroutines are helpful because each one saves a programmer's precious time—time often spent reinventing the wheel! In **The Commodore 128 Subroutine Library** you'll find nearly 100 useful, ready-to-transplant BASIC subroutines and programming tips. Use them to improve your overall programming skills, customize your business programs, and make your games resound with music or sizzle with joystick action.

Grouped by function, annotated carefully, and arranged to be readily dropped into your own BASIC software, these subroutines are ready to use. Beginning programmers will find these subroutines helpful in refining their skills, while intermediate and advanced programmers will appreciate having all these efficient subroutines in a single book.

Find out how you can make your BASIC programs fly by including:

- A Day Converter—Calculating elapsed time for more precise interest calculations.
- Guaranteed Data Input—Eliminating data entry problems by streamlining your input responses.
- Case Converter—Making upper- and lowercase characters respond to your commands.
- Joystick Programs—Speeding up cursor movement.
- Sound Programs—Creating realistic aural environments.
- Software Tricks—Customizing programs through function keys.
- Find out about all this and more!

N 0-553-34308-4>1695

34308-4 ■ IN U.S. $16.95 (IN CANADA $18.95) ■ BANTAM COMPUTER BOOKS