Commodore 64 Subroutine Cookbook

David D. Busch

COMMODORE 64 SUBROUTINE COOKBOOK

DAVID D. BUSCH

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

Note to Authors

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

Publishing Director: David Culverwell
Acquisitions Editor: Gisele M. Asher
Production Editor: Janis K. Oppelt
Manufacturing Director: John A. Komsa
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin

Typesetter: PageCrafters, inc. Oswego, New York
Printer: Fairfield Graphics, Fairfield, Pennsylvania
Typefaces: Omega (text); Cheltenham (display)

Commodore 64 Subroutine Cookbook

Library of Congress Cataloging in Publication Data
Busch, David D.
Commodore 64 subroutine cookbook.
Includes index.
1. Commodore 64 (Computer)—Programming. 2. Basic (Computer program language) 3. Subroutines (Computer programs) I. Title: Commodore sixty-four subroutine cook book.
QA76.8.C64B87 1984 001.64'2 84-2775

ISBN 0-89303-383-9

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

Printed in the United States of America

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

Contents

Preface

How many times have you looked over a program listing in a magazine and thought, "Gee, I could have saved a lot of time if I'd used this joystick subroutine in my own game program!"

Did you read an explanation of how to redefine your computer's character set to new and exciting shapes, only to wonder, "Well, I think I understand how it works—but how do I actually do it?"

Worse, do you find that examples are too complex to understand or that tightly packed programs that you try to dissect are so interwoven and poorly commented that it's impossible to extract the purpose of each statement? Have you been reading a lot of useful tips and programming tricks but lost track of them because they were scattered among a few dozen books and magazines?

This book may be the reference you need and may serve as your shortcut to programming proficiency. Herein are a variety of programming "recipes" in the form of BASIC subroutines that, for the most part, perform only a single task. Useful functions are laid out in subroutines that you can transplant directly to your own programs.

In most cases, the routines are presented in simply constructed lines with only one or two statements per line and no extraneous material. That makes it easy for you to look at the routines, and discover on your own the function of each statement. But, to make sure that you grasp each concept, there is a line-by-line description and an explanation of the important variables used in each subroutine.

Some of the information in this book is available elsewhere, but you'd have to compile a huge stack of material to collect all of it in one place. Instead of searching through back issues of magazines, the reader can thumb through the Contents or Index, and find out how to simulate joysticks or paddles, generate specific sound effects, program character sets, or perform various types of sorts.

Most subroutine books concentrate on "general" business or personal routines. Those are included here, too, but we've also emphasized Commodore-specific tips aimed at your special needs. New capabilities have been added to Commodore BASIC. Special features such as repeating keys, PEEKing colors or characters at a specific location, or changing screen and border combinations are covered. Programming the special function keys and using the built in real-time clock are also included.

Whether you're already expert in BASIC programming and looking for a handy reference guide or a new user seeking access to sophisticated subroutine tricks, this book should satisfy your hunger.

Introduction

Be forewarned. This book is unlike any other collection of subroutines that you might have seen before. Herein are 70 useful, ready-to-transplant subroutines and programming tips that you can use to make your own programs sizzle with joystick action or resound with music. These are Commodore 64 specific routines that take the mystery out of using function keys, designing your own character sets, and other special Commodore 64 features.

Most "subroutine" books are 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 64, while it is a powerful, capable microcomputer, is being sold to a broad range of users. Many only want to play games. Many more are interested in learning programming but may have a skimpy technical background otherwise. Then, there are those of you who really do understand computers but would like to avoid reinventing the wheel.

The Commodore 64 Subroutine Cookbook 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 tasks for the Commodore 64 alone.

Nine subroutines are included that add crucial BASIC functions that were left out of PET 2.0 BASIC. You can now simulate the INSTR function of other BASICs, allowing you to search a string of characters for a shorter string. SWAP variable values. Enter commas into strings through an innovative INPUT routine.

Confused even by the most lucid explanations of using the joysticks to manipulate objects on the screen? Just transplant one of FOUR joystick routines included in this book. We even show you how to move your missiles and enemy aliens around on the screen.

If you find that even the Commodore 64'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 64'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.

Or, manipulate your Commodore 64 to your heart's content. One subroutine will automatically set screen and border colors. Blank the computer's screen, while preserving the data stored there. Turn the repeat key feature on and off.

Games players 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 64 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 but which will help you improve your programming immediately.

While several routines explain how to use the joysticks with the Commodore 64, this book does not include a matching paddle routine. Paddle reading is very unreliable from BASIC and is not recommended. Commodore has a nice machine language paddle routine in its Programmer's Reference Guide that you can incorporate into your programs if you wish.

Use of Commodore's special type of character, called "sprites," is not addressed here, either. Sprite graphics allow defining screen objects that can be moved about the screen by the user and are particularly useful for games. However, while the feature is very powerful, BASIC is much too slow to handle moving them around the screen for any serious applications. In addition, designing sprites and moving them is complex, and well beyond the "subroutine" scope of this book. Typical sprite editor programs, such as that found in Tim Onosko's "Commodore 64: Getting the Most From It" (Robert J. Brady Company) are five or six pages long. Commodore also has a very good sprite program in its Disk Bonus pack, and a lengthy discussion of sprites in its Programmer's Reference Guide.

It is possible to write some very sophisticated, fast-moving games with BASIC using user-defined characters, however. This book has a handy subroutine that lets you define five characters of your own, and other modules show you how to manipulate characters on the screen.

Sorting is another task that is typically very slow in BASIC. However, because of the great demand for this routine, two sorts are included here. For limited-size lists, one of them should be entirely acceptable.

How To Use This Book

To best utilize the information contained in this book, it would be helpful if you have some familiarity with BASIC and know what a FOR-NEXT loop is. Ideally, you should have written several programs on your own, and be ready to tackle some more sophisticated programming. You don't NEED to understand exactly how the joysticks are read through the VIA ports on the Commodore 64 to use these subroutines. But, if you are interested, books are available that will explain these complex points to you. More complete explanations of the Commodore 64's special features are contained in another of my books, "Blast Off With BASIC Games for Your Commodore 64", also published by the Robert J. Brady Company.

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.

In doing so, it may be convenient to relocate them. Some utilities are available to renumber programs for you as an aid to relocating them.

To make things easier, the routines are divided into sections. The basic routine itself is clearly labelled. This portion may be renumbered and placed wherever convenient. If renumbering manually, make sure the GOTO's and GOSUB's in the new modules are correct. You don't want a line that reads: "1000 GET A$:IF A$=""GOTO 160".

Another section of each subroutine will usually be labelled "Initialization." These lines will contain values that must be set once during a program, before the routine is run. Or, the variables will be those that must be defined by your program before calling the subroutine. Frequently, these lines can be deleted or an equivalent line placed within your own program. The explanation with each subroutine tells the purpose of the important variables.

The purpose of all the variables that you need to define, as well as the variable returned by the subroutine for your program's use, is explained as well. Because the operation of many subroutines is rather complex, some of the variables used only internally, as well as various operations, may not be explained. This should be rare, as the line-by-line descriptions cover nearly all the functions of every program. However, if this book does not tell you what a variable does, it is information you do not need to know in order to use the subroutine.

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

All the odd special Commodore 64 characters have been left out of this book. In some cases, CHR$(147) is used for the "clear screen" symbol (reversed heart), while CHR$(17) is used for "cursor down" (reversed Q). Both will perform their proper function, the same as the actual symbols. Deleting the extra graphics makes the subroutines shorter and easier to understand. You may want to add various cursor controls and color controls to the routines for your own use as programs are developed. This is a subroutine cookbook; the finishing touches of the meal are up to you.

Variable names have been chosen, when possible, to reflect their functions in the subroutines. In most cases, the variable names from one subroutine do not conflict with those of another. However, when writing a complex program using several of these modules, you should check to see that the same variable is not used twice for different purposes. Keep in mind that only the first two letters are significant in Commodore 64 BASIC. So, PAYMENT, used in one subroutine, is actually the same as PAID, which might be used in a second. You should take this precaution with any program you write, whether "foreign" subroutines are being transplanted or not.

If you are eager to get started and have some experience in programming, you might want to skip ahead to any subroutine that looks tempting. Those who are less sure of themselves will want to start with Chapter 1, which discusses three ways of merging an existing library of subroutines within your own programs.

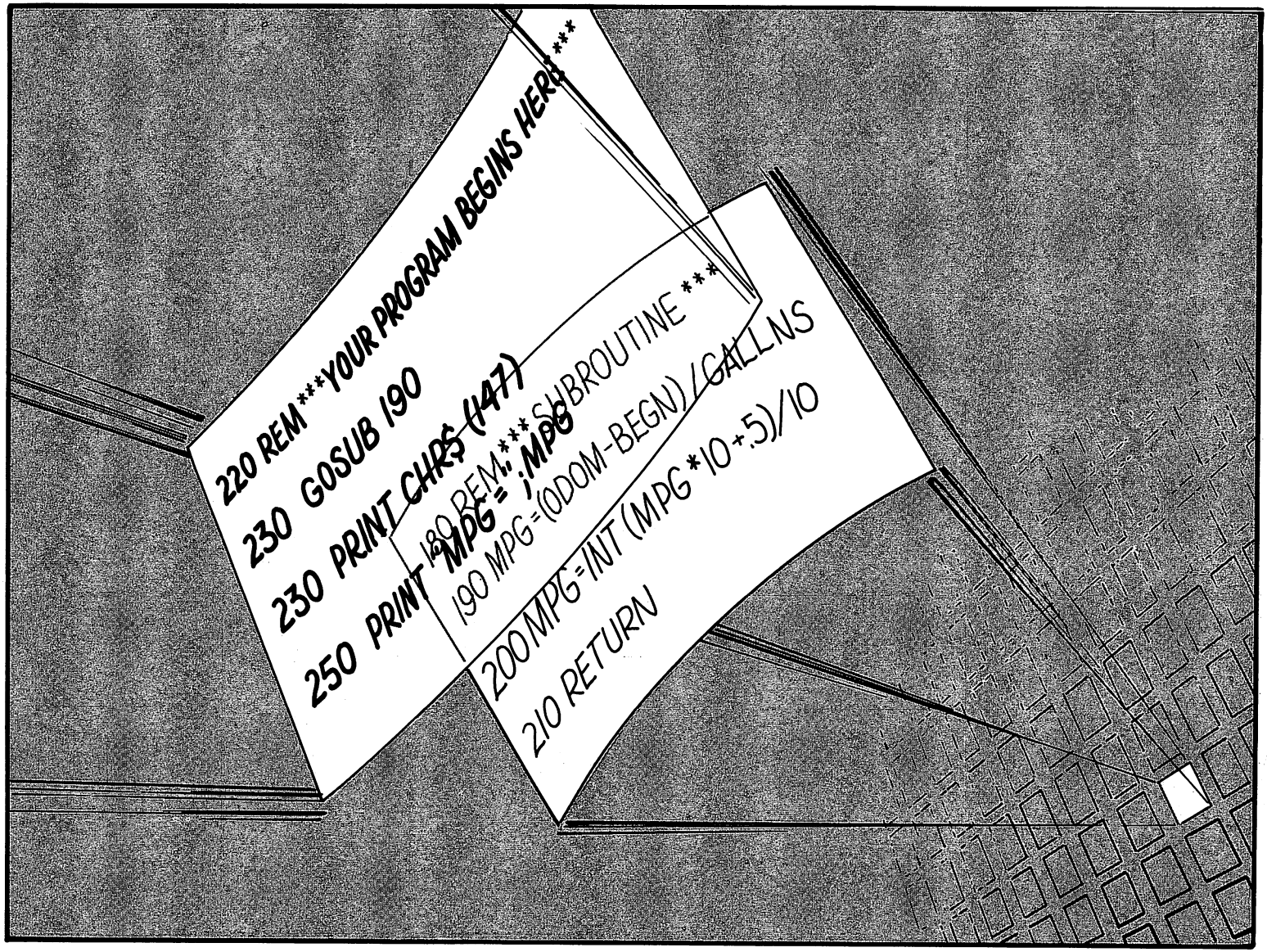
Good luck. You should find this book a shortcut to programming proficiency. To paraphrase a common saying, if you use a subroutine correctly three times, it will be a permanent part of your vocabulary. Given a bit of practice, you can soon have all your friends drooling over your programs and asking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

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the programs to determine their effectiveness. The author and the publisher make no warranty of any kind, expressed or implied, with regard to these programs, the text, or the documentation contained in this book. The author and the publisher shall not be liable in any event for claims of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the text or the programs. The programs contained in this book and on any diskettes are intended for use of the original purchaser-user. The diskettes may be copied by the original purchaser-user for backup purposes without requiring express written permission of the copyright holder.

TRADEMARKS OF MATERIAL MENTIONED IN THIS TEXT

- PAC-MAN: Midway Mfg. Co., a Bally Company
- MASTERMIND: Invicta
- DUNGEONS & DRAGONS: TRS Hobbies, Inc.
- APPLE MACHINES: Apple Computer, Inc.
- RADIO SHACK MACHINES: Tandy Corporation


220 REM***YOUR PROGRAM BEGINS HERE***
230 GOSUB 190
230 PRINT CHR$(147)
250 PRINT "MPG=";MPG
180 REM***SUBROUTINE***
190 MPG=(ODOM-BEGN)/GALLNS
200 MPG=INT(MPG*10+.5)/10
210 RETURN

1 Merging Subroutines With Your Programs

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 stand-alone 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 call them into your programs as needed.

Incorporating existing code into a program is called "merging" and can be accomplished in many different ways. The very simplest can be used if only one stock subroutine will be used in your new program. In such cases, just load the subroutine you want into memory, and write all the other program lines around it.

But, what if you want to incorporate several subroutines into a program, or add them to one which has already been written? Doesn't loading a new subroutine or program destroy anything that is in memory? Not necessarily. This chapter will show you three more ways of merging program lines and subroutines; none of them require a great deal of programming expertise.

First, let's look at the two kinds of merging. In one case, your existing program and the subroutines to be merged have line numbers that do not conflict. Perhaps one or the other has low line numbers, while the code to be merged has high line numbers. That is, your program is numbered from 100 to 1000, while the subroutine(s) to be added all have line numbers higher than 1000. Computerists have a special name for this kind of merge: "appending." One program or module is added to, or appended to, the end of the other. This method is easiest to use, from the standpoint that there is no danger that wanted program lines will be written over with those of the merged program.

However, in the case of true merges, your target program may have program lines that are inclusive of those in the subroutine to be merged. Your program numbered from 100 to 1000 can be merged with a subroutine that is numbered from 500 to 600. If any duplicate lines exist, those of the original program will be replaced by those of the merged program. With some planning, such a merging scheme can also be successful. You would need to make sure that there are no program line numbers in common by, say, purposely leaving a gap between lines 500 to 600 in your original program. Or, perhaps, those lines are occupied by a subroutine that you no longer want. When using this type of merge, be certain that there are no "leftover" lines from the original subroutine or program overlapping with those of your subroutine. For most, the append type of merge is the safest and easiest to implement.

Now, here are three more easy ways to merge any of the subroutines in this book with your own programs.

METHOD #1: SPECIAL UTILITY PROGRAMS

Various vendors sell utility programs for the Commodore 64 which have merging capabilities. Some also include other features, such as FIND or CHANGE commands, which make them indispensable for the serious programmer.

One such utility was used to prepare many of the subroutines in this book. Several components of each subroutine, such as the title block and REMarks, were subroutines of a sort themselves, stored on a Commodore 64 disk in that form. Once a working subroutine was developed, it was renumbered with high line numbers, and the added

lines desired merged, using the programming utility's MERGE"filename",8 command. Subroutines can also be merged from tape using such a utility. For maximum flexibility, this method cannot be beat as a fast, efficient way of adding this book's subroutines to your own programs.

METHOD #2: A FAST, EASY MERGE

Here is a fast merging method that requires no software or programming on your part. Its only limitation is that you can only merge as many lines as will fit on the Commodore 64's screen at once.

If you plan on merging frequently, it will be a good idea to renumber each subroutine in this book with high line numbers that will not conflict with those of your program. With long routines, strip off the unnecessary program lines, such as REMarks, title block, and dummy data.

To merge, load the subroutine first, and LIST it to the screen. Then, LOAD the program you want to merge it with. Your subroutine will be gone, except that the listing will still be in the screen editor's memory. Move the cursor up to each of the remaining subroutine lines so that the cursor is within the line itself. Hit RETURN. The cursor will drop down to the next line. Hit RETURN again. Repeat until all the subroutine lines have been so treated.

Now, list your program. The subroutine will be merged (appended, actually) to the end of your program. You can do an actual merge, with the subroutine located in the middle of your target program, as long as no line numbers conflict. If any do, the subroutine lines will replace those of your program.

Of course, only as many program lines as can fit on the screen, along with the LOAD command, can be merged. But, since many of the routines in this book are very short, you will be able to do the job in nine cases out of ten.

METHOD #3: LONGER MERGES*

This simple tape cassette procedure works for longer subroutines but must be followed carefully and exactly. First, prepare an ASCII version of the subroutine to be appended as follows. LOAD the subroutine and make sure that the line numbers are larger than the line numbers in the programs you intend to append the subroutine to. Place a blank tape in the cassette recorder, and rewind. Next type OPEN 1,1,2,"ASCII-prog. name":CMD1:LIST, and press RETURN. What this step does is open a data channel, number one, with the cassette recorder specified as the output device, using the filename "ASCII-prog.name." You can substitute your own program name for the second part of the file name, but the total filename must be 16 characters or fewer. Then, CMD1 tells the Commodore 64 to route to the tape recorder anything that would have been sent to the CRT screen. The following command, LIST, has the Commodore 64 list the program (or subroutine) in the usual manner—except that the ASCII listing of the module will be written to the tape instead of the screen.

---

*Adapted from a paper presented by Dr. Melvin E. Zandler, Chemistry Department, Wichita State University, Witchita, Kansas (with permission).

Once you have keyed in this command line, and pressed RETURN, the Commodore 64 will ask you to press PLAY and RECORD. Do so, and wait until the tape drive stops. With the PLAY and RECORD buttons still depressed, type PRINT#1:CLOSE1 on the keyboard and press RETURN. The cassette recorder will run for several more seconds as the remaining data in the Commodore 64's buffer is written to tape and the file is closed. You will now have an ASCII version of the subroutine on tape. Rewind the tape and label it.

Next, load your main program into the Commodore 64. Mount the ASCII subroutine tape in the cassette recorder, and TYPE OPEN1,1 on the keyboard. Press RETURN. You will be asked to press the PLAY button on the cassette recorder. When you have done that, the recorder will run until the tape header with the filename is read ("ASCII-prog.name") and the recorder stops.

At this point, leave the PLAY key depressed. Next press CLR to clear the screen, and type POKE 153,1 at the top of the screen, and press RETURN. The cassette drive will run for a few seconds until the buffer is filled with characters. Now, alternately press HOME and RETURN. Each time you do this, one of the subroutine lines from the buffer will be added to the main program in memory. When the buffer is empty, the recorder will start again, and additional data read into the buffer. Repeat pressing HOME and RETURN until the end of the file mark is sensed. At this point, a SYNTAX error will be displayed on the screen.

Congratulations! You now have appended the subroutine to the main program with all the program pointers properly adjusted. LIST the program to check. Rewind your ASCII subroutine tape to make it ready to append to another main program. You can repeat these steps as many times as necessary to add several subroutines to your main program.

SAMPLE MERGING RUN

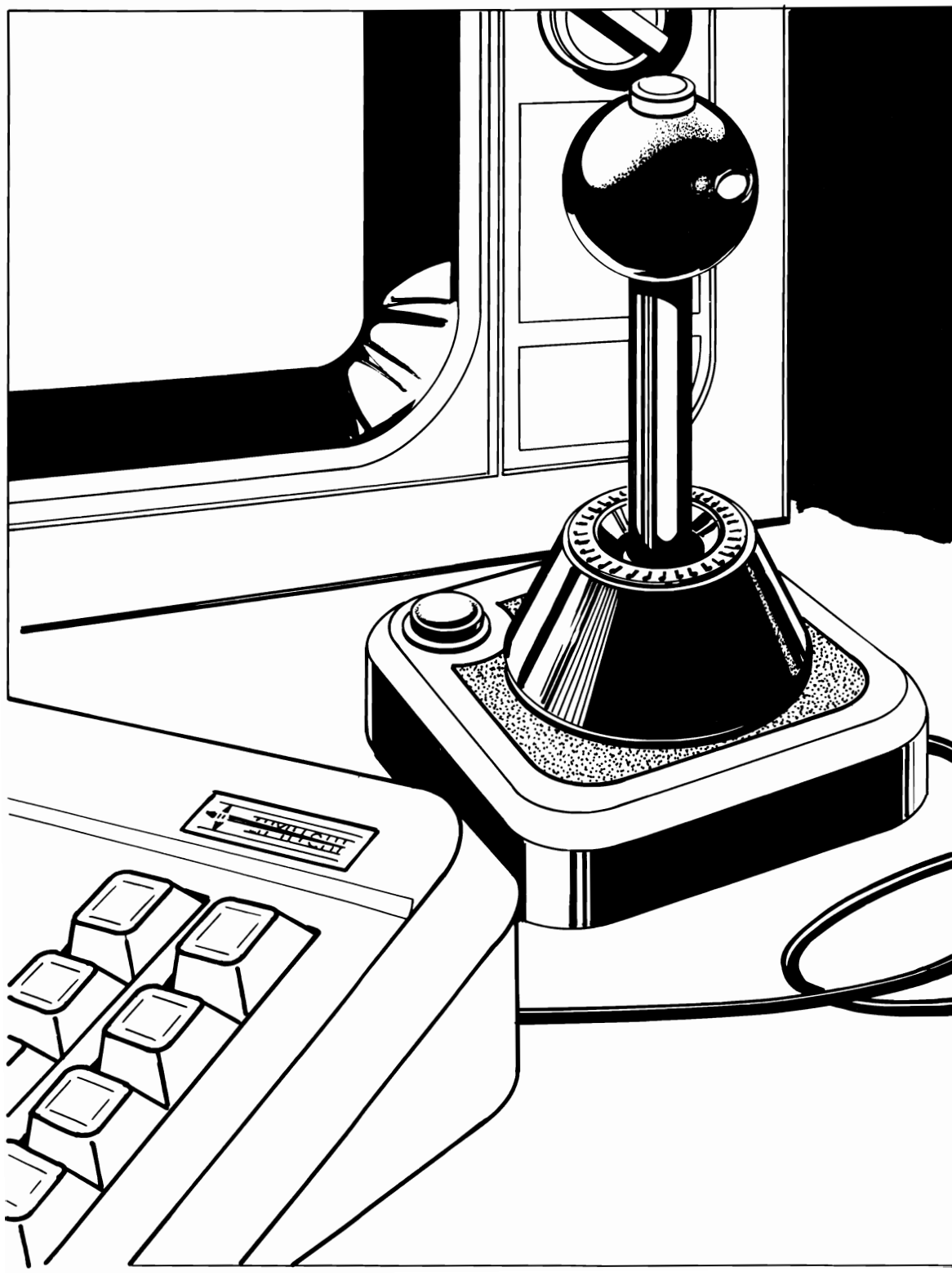
Subroutine (lines 10-250)

```
10 REM **********************
20 REM *                     *
30 REM * COMPOUND INTEREST   *
40 REM *                     *
50 REM **********************

60 REM ---------------------------
70 REM       + + VARIABLES + +
80 REM    RATE: INTEREST RATE
90 REM    YEARS: YEARS COMPOUNDED
100 REM  FUTURE: FUTURE VALUE
110 REM  AMOUNT: AMOUNT TO BE COMPOUNDED
120 REM
130 REM ---------------------------

140 REM *** INITIALIZE ***
```

```
150 RATE=10
160 AMOUNT=1000
170 PERIOD=365
180 YEARS=10
190 GOTO 260

200 REM *** SUBROUTINE ***

210 RATE=RATE/100
220 FUTURE=AMOUNT*(1+RATE/PERIODS)[(PERIODS
    *YEARS):LOAN=PAYMENT*(1-(1+RATE)^-NUMBER)/RATE
230 FUTURE=INT(FUTURE*100+.5)/100
240 RETURN

250 REM *** YOUR PROGRAM STARTS HERE ***
```

Program Merged With Subroutine

```
260 PRINT CHR$(147);CHR$(17)
270 PRINT "COMPOUND INTEREST"
280 PRINT CHR$(17)
290 INPUT "INTEREST RATE";RATE
300 INPUT "YEARS TO BE COMPOUNDED";YEARS
310 INPUT "AMOUNT TO BE COMPOUNDED";AMOUNT
320 INPUT "COMPOUNDING PERIODS/YR";PERIODS
330 GOSUB 210
340 PRINT "FUTURE VALUE:";FUTURE
350 PRINT "DO IT AGAIN?"
360 PRINT TAB(4)"Y/N?"
370 GET A$:IF A$=" " GOTO 370
360 IF A$="Y" THEN GOTO 260
370 IF A$="N" THEN END
380 GOTO 370
```



2 Using Joysticks

Let's start off with one of the most mystifying capabilities of the Commodore 64: joystick manipulation of screen objects. All of us have marveled at arcade-quality games for the Commodore 64 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.

From a programmer's standpoint, the use of the joysticks breaks down into several neat modules. We need to know where on the Commodore 64's screen the object to be moved is. We also must check, as often as possible, the status of 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, erase its image from the old location.

Four subroutines that follow demonstrate alternate ways of moving objects in BASIC, using the joystick as a controller.

The four different joystick routines have been included in this section to allow several different types of object movement in your programs. The first joystick subroutine in this section allows moving an object only in north, south, east, or west directions. The next routine permits movement only left or right, but allows you to pick which row your object will reside in. The third subroutine of the group allows diagonal movement for games or other programs needing that flexibility. All will also tell you if 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 subroutine, DRAW, brings all the elements together in a short program that will allow you to draw on the screen using a joystick. Pressing number keys changes the color of the cursor, while other keys change the cursor character itself.

JOYSTICK SUBROUTINE

WHAT IT DOES

Moves object north, south, east, and west, using joystick.

Variables

- CSCREEN: Start position in memory of a map of the color of each Commodore 64 screen location
- CHAR: Start position in memory of a map of which characters are displayed in each Commodore 64 screen location
- E: End of character memory
- B1: Current position of Object #1
- B2: Current position of Object #2
- DF: The difference between CSCREEN and CHAR
- M1: Direction of move for Object #1, to be used to increment B1

- M2: Direction of move for Object #2, to be used to increment B2
- F1: Status of fire button #1
- F2: Status of fire button #2.

How To Use The Subroutine

The Commodore 64 has two ports which can accommodate, among other accessories, one joystick each. Your games and applications can use the status of the joystick to control the actions of your programs. Usually, the movement of the joystick directs the movement of an object on the screen. That is, when the joystick is pressed left, the object moves left. Motion to the right, up, and down can also be accomplished.

There is no reason why a joystick could not be applied to other program tasks, however. Such input might be appropriate for a very young user who does not know how to type on the keyboard. Moving the joystick to the left might trigger one pictorial "menu" choice; to the right another. Pressing the FIRE button could advance the program to the next screen, and so forth.

This subroutine, while written with object movement in mind, can easily be adapted to that type of application. The basic routine will, if called repeatedly, monitor the status of the joysticks and provide a value that indicates which way the cursor should move. Only north, south, east, or west movement is allowed with this routine, which is best suited for many "maze" and similar games. If you need diagonal movement as well, try the third Commodore 64 joystick subroutine.

Call the subroutine whenever you wish to check on the status of the joysticks. The value of M1 or M2 can be used with B1 and B2 (the current position of each object) to move the object or to direct some other program action. Value of F1 and F2, the fire buttons of each joystick, is also returned, although it is not used in this subroutine. You can make pressing the FIRE button accomplish some other task, such as clearing the screen:

```
465 IF F1=1 THEN PRINT CHR$(147)
```

This subroutine prints a ball character (81 in the Commodore 64 character set) on the screen. You may change the character by substituting some other number for the 81 in line 510. Or, substitute variable CURSR for the 81, and define CURSR to any character you wish.

The routine leaves a "trail" of the character behind it. You can erase this by POKing B1-M1 with 32. Look at the following subroutines for a more complete use of this feature.

Variables Not To Be Changed By User

- CSCREEN: The position in memory of the start of the color memory map.
- CHAR: The position in memory of the start of the character memory map.
- E: End of character memory. You will not want your object to move beyond this value, or else it will be "off" the screen, and could destroy your program!
- DF: The difference between CSCREEN and CHAR. If variable B1 defines the current cursor position in character memory (it will be some number between CHAR and E), then POKing B1 with a number will cause that character to appear on the screen. POKing B1 + DF with a color number (0-7) will change the color of that character position.

Line By Line Description

Line 70: Define the start of the memory map which stores the screen color of each of the 1000 Commodore 64 screen positions.

Line 80: Define the start of the memory map which stores the code number for each character stored in each of the 1000 Commodore 64 screen positions.

Line 90: Initialize the position of objects number 1 and (B1 and B2) to equal the start of character memory.

Line 100: Define the highest and lowest positions that you want an object to move to; beginning and end of screen memory.

Line 110: Calculate the difference between the start of screen memory and character memory.

Lines 290 to 300: Peek registers which store status of joystick 1 and joystick 2.

Lines 310 to 320: Look at bit which tells whether FIRE button of either joystick is pressed. NOTE: Using the AND operator and bit manipulation are explained much more thoroughly in Chapters 5 and 11.

Lines 330 to 340: Determine status of each joystick's switches.

Lines 350 to 420: Depending on value of joystick status variable, J1 and J2, movement of object is defined as +40 (down one whole row), −40 (up one whole row), +1 (one position to the right), or −1 (one position to the left).

Lines 450 to 460: Clear screen and access the subroutine.

Lines 470 to 480: If either FIRE button is pressed, clear the screen. Your program could initiate some other action, such as releasing a missile.

Lines 490 to 500: Update position of object, B1 and B2. If position is less than beginning of character memory, B, or greater than the end, E, negate this move.

Lines 510 to 540: POKE new position of object, B1 or B2, with character desired, in this case the CHR$(81), "ball" character. Then POKE the corresponding position in color memory with a color, different for each joystick.

Line 550: Go back and repeat.

You Supply

- B1 or B2: Current position of object. This will be a number between CHAR and E. Your program should always check to make sure that B1 is never less than CHAR or greater than E. You may want to define B1 = CHAR at the beginning of the program to start in the upper left. That is done in this subroutine. Or, choose some other position. Each movement of the cursor is made by changing the value of B1 or B2, and then POKing B1 or B2 with the number of the character you want. Erase B1 or B2 from its old position by POKing its former value with 32 (a space).

- M1 or M2: Direction of move, to be used to increment B1 or B2. If the cursor is to move to the right one space, then M1 or M2 will equal +1. If the move will be to the left, M1 or M2 will equal −1. Upward movement is produced by making M1 or M2 equal the value of one whole row, −40. Downward movement is accomplished by making M1 or M2 equal +40.

RESULT

Object will move on screen under joystick control, in north, south, east, or west directions.

```
10 REM ************************
20 REM *                      *
30 REM * JOYSTICK SUBROUTINE *
40 REM *                      *
50 REM ************************

60 REM *** INITIALIZE ***

70 CSCREEN=55296
80 CHAR=1024
90 B1=CHAR:B2=CHAR
100 B=CHAR:E=CHAR+1000
110 DF=CSCREEN-CHAR
120 GOTO 450
130 REM ----------------------------
140 REM      ++ VARIABLES ++
150 REM     F1:   STATUS OF FIRE BUTTON 1
160 REM     F2:   STATUS OF FIRE BUTTON 2
170 REM     M1:   MOVE FOR JOY 1
180 REM     M2:   MOVE FOR JOY 2
190 REM     B1:   POSTION FOR OBJECT 1
200 REM     B2:   POSITION FOR OBJECT 2
230 REM     CSCREEN: START OF COLOR MEMORY
240 REM     CHAR:   START OF CHARACTER MEMORY
250 REM     E:  END OF CHARACTER MEMORY
260 REM
270 REM ----------------------------

280 REM *** SUBROUTINE ***
```

```
290 J1=PEEK(56320)
300 J2=PEEK(56321)
310 F1=J1 AND 16
320 F2=J2 AND 16
330 J1=15-(J1 AND 15)
340 J2=15-(J2 AND 15)
350 IF J1=1 THEN M1=-40:GOTO 390
360 IF J1=2 THEN M1=40:GOTO 390
370 IF J1=4 THEN M1=-1:GOTO 390
380 IF J1=8 THEN M1=1:GOTO 390
390 IF J2=1 THEN M2=-40:RETURN
400 IF J2=2 THEN M2=40:RETURN
410 IF J2=4 THEN M2=-1:RETURN
420 IF J2=8 THEN M2=1:RETURN
430 RETURN

440 REM *** YOUR PROGRAM STARTS HERE ***

450 PRINT CHR$(147)
460 GOSUB 290
470 IF F1=0 THEN PRINT CHR$(147)
480 IF F2=0 THEN PRINT CHR$(147)
490 B1=B1+M1:IF B1<B OR B1>E THEN B1=B1-M1
500 B2=B2+M2:IF B2<B OR B2>E THEN B2=B2-M2
510 POKE B1,81
520 POKE B1+DF,3
530 POKE B2,81
540 POKE B2+DF,5
550 GOTO 460
```

MOVE OBJECT LEFT, RIGHT

WHAT IT DOES

Moves object left and right only.

Variables

- ROW: Row to move object in
- B: Beginning of that row
- E: End of that row
- B1: Position of object
- CURSR: Cursor character

- CO: Cursor color
- MOVE: Direction of move
- CSCREEN: Beginning of color memory
- CHAR: Beginning of character memory
- DF: CSCREEN-CHAR
- F1: Status of FIRE button.

How To Use The Subroutine

While the Commodore 64's game port can be used with joysticks, paddles, and other accessories, the joysticks are probably the most flexible controllers for games. They can be used both to control movement in all directions, as well as just from left to right, like a paddle.

A paddle is like "rheostat" or volume control that returns an analog value to the Commodore 64. That is, some number between 0 and 255 is supplied, depending on how far the paddle has been turned. Unfortunately, a small movement of the paddle can result in a large movement of the object on the screen. Paddles are NOT recommended from BASIC by Commodore, because they can be unreliable.

Some games work better when the joystick control is used for movement left and right. In such cases, the joystick will indicate a "zero" when not pressed in a direction and a number when that switch is closed. This approach makes games programming simpler. Move an object one position to the right when the right switch is pressed, and move one position to the left when the left switch is pressed.

This subroutine will tell you whether or not either case has occurred, plus report on the status of the FIRE button.

Your program should repeatedly check to see if MOVE has a value, and take action accordingly. Your object might be located along the bottom of the screen, in a position defined as, say, B1. This will be increased or decreased by one each time the joystick is pressed right or left. Make sure that B1 never exceeds the end of character memory, when POKing objects to the screen. A more complete discussion of moving objects on the screen is included with the previous subroutine.

Line By Line Description

Line 220: Define an array capable of keeping track of ten possible values for two different joysticks, even though only two positions and one joystick are used in this program.

Line 230: Define DATA for direction of movement when joystick is pressed in any of the ten possible values. In this case, only two are used, to provide for movement left and right.

Line 270: Clear screen.

Line 280: Define start of memory map that stores information about what color should be printed in every screen position.

Line 290: Define start of memory map that stores information about what character should be printed in every screen position.

Line 300: Define the row in which the object movement should take place. You may redefine this as any value, 1 to 40.

Line 310: Define starting position of object as the first position in the row chosen.

Line 320: Define maximum position of object as starting position, plus 39.

Line 330: Define minimum position of object as the first position on that row.

Line 340: Calculate difference between the start positions of the two memory maps.

Line 360: Define initial cursor character and color.

Line 390: PEEK joystick register for current value.

Line 400: Calculate switch status.

Line 410: See if FIRE button is depressed.

Line 420: Define next MOVE as J1 element of array. If that array element equals zero, then MOVE will equal zero.

Line 450: Access the subroutine.

Line 460: If MOVE is not 0, then erase object in old position of cursor, by POKing a space, CHR$(32), there.

Line 470: Update position of B1, by adding MOVE.

Line 480: If MOVE would take cursor beyond limits, negate update.

Line 490: POKE cursor position with cursor character.

Line 500: POKE corresponding color location with color value.

Line 510: Go back and repeat.

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, while CO will be the cursor color. CO should be the color value you wish, minus one.

RESULT

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

```
10 REM ******************
20 REM *                 *
30 REM *   MOVE OBJECT   *
40 REM *  LEFT OR RIGHT  *
50 REM *                 *
60 REM ******************
70 REM ------------------------------
80 REM    + + VARIABLES + +
90 REM    ROW:      ROW TO MOVE IN
100 REM   B:        BEGINNING OF THAT ROW
110 REM   E:        END OF THAT ROW
120 REM   B1:       POSITION OF CURSOR
130 REM   CURSR:    CURSOR CHARACTER
140 REM   CO:       CURSOR COLOR
150 REM   MOVE:     DIRECTION OF MOVE
160 REM   CSCREEN:  COLOR MEMORY
170 REM   CHAR:     CHARACTER MEMORY
180 REM   DF:       CSCREEN-CHAR
190 REM
200 REM ------------------------------

210 REM *** INITIALIZE ***

220 DIM D(10,2)
230 DATA 0,0,0,-1,0,0,0,1,0,0
240 FOR X=1 TO 10
250 READ D(X,1)
260 NEXT X
270 PRINT CHR$(147)
280 CSCREEN=55296
290 CHAR=1024
300 ROW=5
310 B1=CHAR+ROW*40
320 E=B1+39
330 B=B1
340 DF=CSCREEN-CHAR
350 POKE 53281,1
360 CURSR=43:CO=2
370 GOTO 450

380 REM *** SUBROUTINE ***

390 JV=PEEK(56320)
400 F1=JVAND16
410 J1=15-(JVAND15)
420 MOVE=D(J1,1)
430 RETURN
```

```
440 REM *** YOUR PROGRAM STARTS HERE ***

450 GOSUB 390
460 IF MOVE<>0 THEN POKE B1,32
470 B1=B1+MOVE
480 IF B1<B OR B1>E THEN B1=B1-MOVE
490 POKE B1,CURSR
500 POKE B1+DF,CO
510 GOTO 450
```

MOVE OBJECT ALL DIRECTIONS

WHAT IT DOES

Moves object diagonally, and north, south, east, and west.

Variables

- CSCREEN: The position in memory of the start of the color memory map
- CHAR: The position in memory of the start of the character memory map
- E: End of screen in memory
- B1: Current position of cursor
- DF: The difference between CSCREEN and CHAR
- MOVE: Direction of move, to be used to increment B1
- CURSR: Cursor character
- CO: Cursor color.

How To Use The Subroutine

Games like Pac-Man(TM) and maze chases work best if the object can only be moved 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 other programs where input with a joystick is desirable.

Call the subroutine every time you wish to check on the status of the joysticks. The value of MOVE can be used with B1 to move the object, or to direct some other program action. Value of F1, the FIRE button, is also returned, although it is not used in this subroutine. You can make pressing the FIRE button accomplish some other task, such as clearing the screen:

```
555 IF F1=0 THEN PRINT CHR$(147)
```

This subroutine prints a plus sign, CHR$(43), in the Commodore 64 character set, on the screen. You may change the character by substituting some other number for the 43 in the definition for CURSR in line 310.

The routine leaves a "trail" of the character behind it. You can erase this by POKing B1-MOVE with 32

Variables Supplied By Subroutine

- CSCREEN: The position in memory of the start of the color memory map.
- CHAR: The position in memory of the start of the character memory map.
- E: End of character memory. You will not want your object to move beyond this value or else it will be "off" the screen, and could destroy your program!
- B1: Current position of cursor. This will be a number between CHAR and E. Your program should always check to make sure that B1 is never less than CHAR or greater than E. You may want to define B1 = CHAR at the beginning of the program to start in the upper left. Or, choose some other position. Each movement of the cursor is made by changing the value of B1, and then POKing B1 with the number of the character you want. Erase B1 from its old position by POKing its former value with 32 (a space).
- DF: The difference between CSCREEN and CHAR. If variable B1 defines the current cursor position in character memory (it will be some number between CHAR and E), then POKing B1 with a number will cause that character to appear on the screen. POKing B1 + DF with a color number (0-7) will change the color of that character position.
- MOVE: Direction of move, to be used to increment B1. If the cursor is to move to the right one space, then MOVE will equal +1. If the move will be to the left, MOVE will equal −1. Upward movement is produced by making MOVE equal the value of one whole row, −40. Downward movement is accomplished by making MOVE equal +40.

An additional feature has been added for diagonal movement. Instead of incrementing MOVE by whole rows, a row MINUS one, or PLUS one is used. That is, adding −41 to MOVE will produce movement up one row, and over one character, or, diagonally to the NorthWest. Adding −39 will accomplish NorthEast movement, and values of 39 and 41, SouthWest and SouthEast, respectively.

Line By Line Description

Line 190: DIMension array to store ten possible values of joystick, with two ROWS to keep track of two joysticks, even though only one is used in this program.

Line 200: Define DATA showing direction of MOVE, with -40 equivalent to North, -39 equal to Northeast, -41 equal to Northwest, etc.

Lines 210 to 230: Read joystick directions into array.

Line 240: Clear screen.

Lines 250 to 260: Define start of character and color memory maps.

Line 270: Define beginning and end positions that cursor will move.

Line 280: Define initial cursor position.

Line 290: Calculate difference between addresses of color and character memory maps.

Line 310: Define cursor and cursor color.

Line 340: PEEK joystick register for current value.

Line 350: See if FIRE button is depressed.

Line 360: Check switch status.

Line 370: Define next MOVE as J1 element of array. If that array element equals zero, then MOVE will equal zero.

Line 400: Access the subroutine.

Line 410: Update position of B1, by adding MOVE.

Line 420: If MOVE would take cursor beyond limits, negate update.

Line 430: POKE cursor position with cursor character.

Line 440: POKE corresponding color location with color value.

Line 450: Go back and repeat.

You Supply

Simply move joysticks. Define CURSR and CO to provide cursor character and color of your choice.

RESULT

Object will move on screen.

```
10 REM ******************
20 REM *                 *
30 REM *  MOVE OBJECT    *
40 REM * ALL DIRECTIONS *
50 REM *                 *
60 REM ******************
70 REM ------------------------------
80 REM   + + VARIABLES + +
90 REM    CSCREEN:  COLOR MEMORY
100 REM   CHAR:     CHARACTER MEMORY
110 REM   DF:       CSCREEN-CHAR
120 REM   B1:       POSITION OF CURSOR
130 REM   MOVE:     DIRECTION OF MOVE
140 REM   CURSR:    CURSOR CHARACTER
150 REM   CO:       CURSOR COLOR
160 REM
170 REM ------------------------------

180 REM *** INITIALIZE ***
```

```
190 DIM D(10,2)
200 DATA -40,40,0,-1,-41,39,0,1,-39,41
210 FOR X=1 TO 10
220 READ D(X,1)
230 NEXT X
240 PRINT CHR$(147)
250 CSCREEN=55296
260 CHAR=1024
270 E=CHAR+1000:B=CHAR
280 B1=CHAR
290 DF=CSCREEN-CHAR
300 POKE 53281,1
310 CURSR=43:CO=2
320 GOTO 400

330 REM *** SUBROUTINE ***

340 JV=PEEK(56320)
350 F1=JVAND16
360 J1=15-(JVAND15)
370 MOVE=D(J1,1)
380 RETURN

390 REM *** YOUR PROGRAM STARTS HERE ***

400 GOSUB 340
410 B1=B1+MOVE
420 IF B1<B OR B1>E THEN B1=B1-MOVE
430 POKE B1,CURSR
440 POKE B1+DF,CO
450 GOTO 400
```

DRAWING SUBROUTINE

WHAT IT DOES

Draws on screen, changes cursor character and color as desired.

Variables

- B1: Current position of cursor
- MOVE: Direction of move, to be used to increment B1
- CURSR: Current cursor character
- CO: Current cursor color.

How To Use The Subroutine

Using the Commodore 64 joystick to sketch on the screen is a natural application. This subroutine, similar to the last, has the added feature of automatically changing the cursor character, as well as the cursor color, under direction of the operator. Pressing the FIRE button will clear the screen.

You can adapt this subroutine to many different types of programs—for drawing floor plans and other design projects.

Pressing any alpha key will cause the cursor to change to that letter or, if shifted, that graphics character. Pressing a number key will cause the cursor to change to that color. The cursor color, CO, is alternated with RED to show movement on the screen more clearly. A short sound routine is also included to mark the movement around the screen audibly.

Variables Supplied By The Subroutine

- B1: Current position of cursor. This will be a number between CHAR and E. Your program should always check to make sure that B1 is never less than CHAR or greater than E. You may want to define B1 = CHAR at the beginning of the program to start in the upper left. Or, choose some other position. Each movement of the cursor is made by changing the value of B1 and then POKing B1 with the number of the character you want. Erase B1 from its old position by POKing its former value with 32 (a space).
- MOVE: Direction of move, to be used to increment B1. If the cursor is to move to the right one space, then MOVE will equal +1. If the move will be to the left, MOVE will equal −1. Upward movement is produced by making MOVE equal the value of one whole row, −40. Downward movement is accomplished by making MOVE equal +40.

Line By Line Description

Line 150: DIMension array to store ten possible values of joystick, with two ROWS to keep track of two joysticks.

Line 160: Define DATA showing direction of MOVE, with −40 equivalent to North, −39 equal to Northeast, −41 equal to Northwest, etc.

Lines 170 to 190: Read joystick directions into array.

Line 210: Clear screen.

Line 220: Define start of character memory map.

Lines 250 to 290: Define registers for volume, voice, wave form, attack, and sustain for sound routines.

Line 300: Define initial values of cursor and cursor color.

Line 330 to 380: PEEK joystick register for current value, then see if FIRE button depressed, and check switch status. Do twice, once for each joystick.

Line 410: Access the subroutine.

Line 420: Look for keyboard input, if none, then move on.

Line 430: Access sound routine.

Line 440: If possible MOVE does not equal 0, give MOVE that value.

Line 450: Update value of object 1, B1, with MOVE.

Lines 460 to 490: If MOVE will take object off the screen, negate update.

Line 500: POKE cursor position with cursor character, then POKE corresponding color location with color value.

Line 510: If cursor color equals WHITE, flash cursor anyway, so user can follow movement.

Line 520: If FIRE button pressed, clear screen.

Lines 540 to 550: If number key pressed, change cursor color.

Line 570: Change cursor to key that was pressed.

Lines 600 to 650: Sound routine. POKE desired VOLUME, WAVE form, ATTACK, SUSTAIN, and note, into VCE.

You Supply

- CURSR: Current cursor character. This number is POKEd into B1 to paint on the screen.
- CO: Current cursor color.

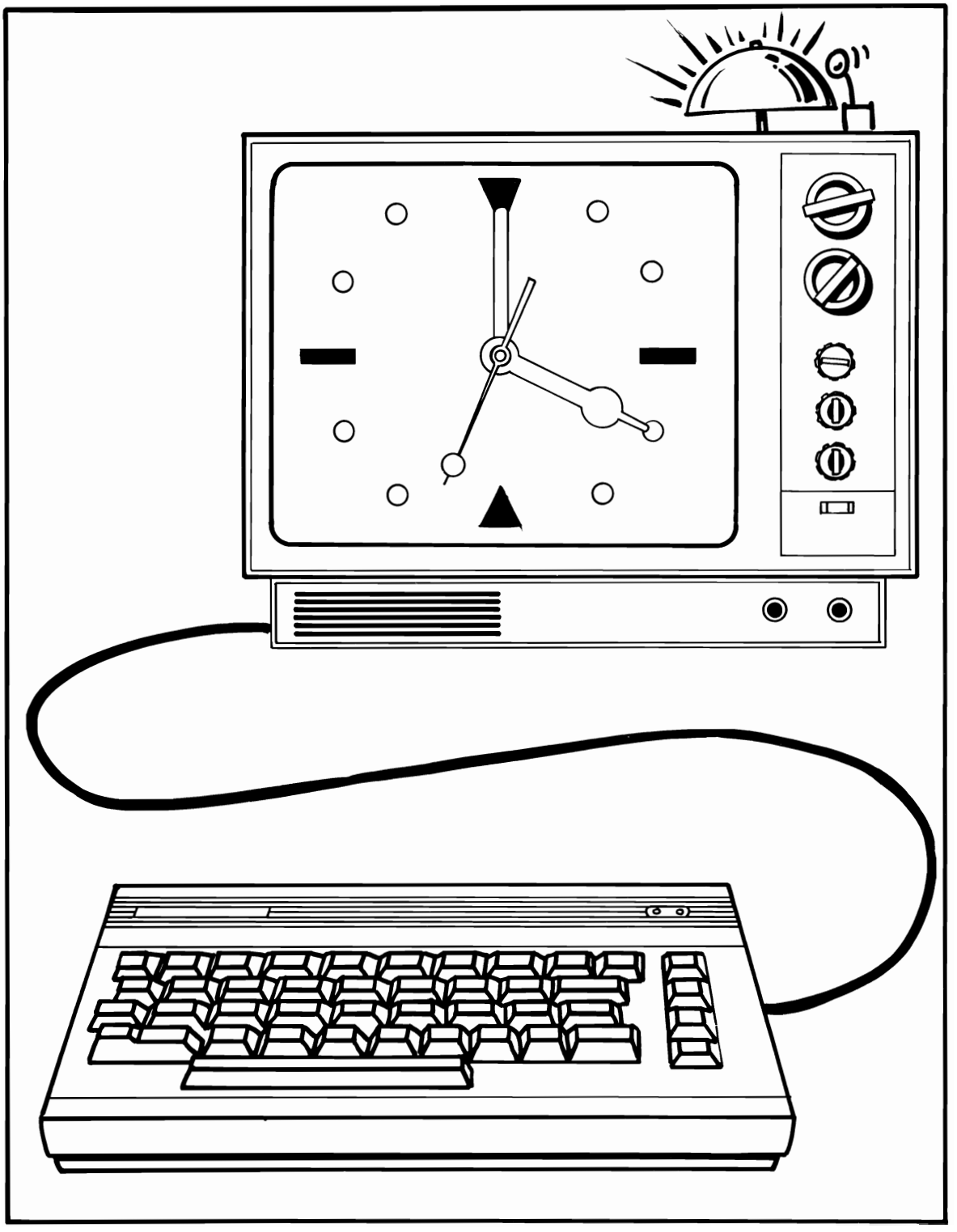
RESULT

Drawing on screen, with variety of colors and cursor characters.

```
10 REM *********
20 REM *       *
30 REM * DRAW  *
40 REM *       *
50 REM *********
60 REM ------------------------------
70 REM       + + VARIABLES + +
80 REM     MOVE:  DIRECTION OF MOVE
90 REM     B1:    POSITION OF CURSOR
100 REM    CURSR: CURSOR CHARACTER
110 REM    CO:    CURSOR COLOR
120 REM
130 REM ------------------------------

140 REM *** INITIALIZE ***
```

```
150 DIM D(10,2)
160 DATA -40,40,0,-1,-41,39,0,1,-39,41
170 FOR X=1 TO 10
180 READ D(X,1)
190 NEXT X
200 V=53248
210 PRINT CHR$(147)
220 B1=1024
230 MOVE=1
240 POKE 53281,1
250 VOLUME=54296
260 VCE=54273
270 WAVE=54276
280 ATTACK=54277
290 SUS=54278
300 CURSR=43:CO=2
310 GOTO 410

320 REM *** READ JOYSTICKS ***

330 JV=PEEK(56320)
340 F1=JVAND16
350 J1=15-(JVAND15)
360 JV=PEEK(56321)
370 F2=JVAND16
380 J2=15-(JVAND15)
390 RETURN

400 REM *** DRAW SUBROUTINE ***

410 GOSUB 330
420 GET A$:IF A$<>" " THEN GOSUB 540
430 GOSUB 600
440 IF D(J1,1)<>0 THEN MOVE=D(J1,1)
450 B1=B1+MOVE
460 IF B1>2022 THEN B1=B1+(D(J1,1)*-1)
470 IF B1<1024 THEN B1=B1+(D(J1,1)*-1)
480 IF B1>2022 THEN B1=B1+(D(J1,1)*-1)
490 IF B1<1024 THEN B1=B1+(D(J1,1)*-1)
500 POKE B1,CURSR:POKE B1+54272,CO
510 IF CO=1 THEN POKE B1+54272,5:POKE B1+54272,CO
520 IF F1=0 THEN PRINT CHR$(147)
530 GOTO 410
540 IF A$<"1" OR A$>"8" GOTO 570
550 CO=VAL(A$)-1
560 RETURN
570 CURSR=ASC(A$)
580 RETURN
```

```
590 REM *** SOUND ***

600 POKE VOLUME,15
610 POKE WAVE,33
620 POKE ATTACK,128
630 POKE SUS,128
640 POKE VCE,72
650 POKE VOLUME,0
660 RETURN
```



3 Using The Clock

The Commodore 64 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 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$, keep track of elapsed time since the computer was turned on or since the clock was last reset by the user. That is, when the computer is first turned on, TI and TI$ equal 000000, and start counting from that point. You may access both TI and TI$ like 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 programing purposes by assigning its value to a variable, as: "CU$ = TI$." You can also set the proper time (the Commodore 64 uses 24-hour, military-style time) by the reverse method: "TI$ = "120000."

Because the Commodore 64 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 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. So, if your program writes data files, it should not depend on TI or TI$ to be 100 percent accurate. And, 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 imbed this routine in your programs when you need to access the time from 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. So long as your program constantly compares TI$ with the finish time specified, you may have it perform other tasks in the meantime, branching to your "time's up!" routine only when the interval has elapsed. As written, this subroutine is much like an alarm clock; however, the computer does nothing else during the timing cycle. Your programs can perform other functions, and simply call the time-check subroutine frequently to see if the desired time interval is up.

TIME SET

WHAT IT DOES

Sets Commodore 64 real-time clock.

Variables

- HR$: Current hours
- MINUTE$: Current minutes
- TI$: Commodore 64 real-time clock value.

How To Use The Subroutine

Many programs can use the built-in real-time clock of the Commodore 64 to measure elapsed time, control events, or simply to keep the operator informed as to what time it is.

The Commodore 64 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 zero is appended automatically to produce, say, 090900.

Line By Line Description

Lines 160 to 200: User enters current hour, which is checked to make sure it is less than 24.

Lines 210 to 250: If number of hours is less than 12, user is asked if A.M. or P.M. is meant. If afternoon is specified, 12 is added to the value of HR, to conform with 24 hour military-style time.

Lines 260 to 290: User enters current minutes, which are checked to see that they do not exceed 59.

Lines 300 to 320: If value of HR or MINUTES is less than ten, a leading zeros are added to fill out the string, e.g., "000900" (for 12:09 A.M.).

Line 330: TI$ is set equal to the hour and minutes input, plus 0 seconds.

You Supply

Subroutine asks for hours and minutes.

RESULT

Internal clock set to correct time.

```
10 REM ************
20 REM *          *
30 REM * TIME SET *
40 REM *          *
50 REM ************
60 REM ----------------------------
70 REM      + + VARIABLES + +
80 REM   HR$       HOURS
90 REM   MINUTE$: MINUTES
100 REM  TI$:     CURRENT TIME
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 GOTO 360

150 REM *** SUBROUTINE ***

160 PRINT "ENTER HOUR :"
170 INPUT HR$
180 HR=VAL(HR$)
190 IF HR>23 THEN PRINT"LESS THAN 24 HOURS,
    PLEASE!":GOTO 170
200 IF HR>12 GOTO 260
210 PRINT"A.M. OR P.M."
220 GET A$:IF A$="" GOTO 220
230 IF A$="P" THEN HR=HR+12:HR$=MID$(STR$(HR),2):GOTO 260
240 IF A$="A" GOTO 260
250 GOTO 220
260 PRINT"ENTER MINUTES :"
270 INPUT MINUTE$
280 MINUTE=VAL(MINUTE$)
290 IF MINUTE>59 THEN PRINT "LESS THAN 60 MINUTES,
    PLEASE!":GOTO 270
300 IF HR<10 THEN HR$="0"+HR$:IF HR<1 THEN HR$="00"
310 IF MINUTE<10 THEN MINUTE$="0"+MINUTE$
320 IF MINUTE<1 THEN MINUTE$="00"
330 TI$=HR$+MINUTE$+"00"
340 RETURN

350 REM *** YOUR PROGRAM STARTS HERE ***

360 GOSUB 160
370 PRINT TI$
```

ELAPSED TIME

WHAT IT DOES

Measures difference between two times.

Variables

- MIN: Elapsed minutes
- SEC: Elapsed seconds
- TS: Jiffies at start
- TF: Jiffies at finish.

How To Use The Subroutine

In addition to TI$, the Commodore 64 also keeps track of TI, which measures the number of 1/60th second intervals that have elapsed since the computer was turned on. Since these increments, called "jiffies," are much more precise than counting seconds, they are especially useful in game programs.

This subroutine takes the number of jiffies at the start 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. Note that the subroutine should not be used to time very long events, because the jiffie counter will reset after "999999," or, about 4.6 hours. Most programs will not need to time that long a period for a single run.

Line By Line Description

Line 160: Set time at finish (TF) to equal current jiffie count, TI.

Line 170: Calculate number of seconds that have elapsed, by dividing the difference between the starting jiffie count, TS, and TF by 60. Since jiffies are 1/60th second intervals, this produces the elapsed seconds.

Line 180: Figure total elapsed minutes by dividing elapsed seconds, FS, by 60. Only integer portion (whole minutes) is used.

Line 190: Remaining seconds are calculated by subtracting the number of seconds in the elapsed minutes (MIN*60) from the total number of seconds elapsed (FS).

Line 200 to 210: Print results to screen.

Line 240: Take initial jiffie reading. This is the start figure for the timer.

Line 250: Wait for end of timing cycle. In this case, a simple GET A$ loop is used. Your program may have a whole program, or some other routine, intervening instead. When your routine or other event that you wish to time is finished, access the subroutine.

Line 260: Go to the subroutine. Figure out how much time has elapsed between the execution of Line 240 and this line.

You Supply

Your program should set TS to equal TI when you wish to start timing. When the end of the timing cycle is over, call the subroutine.

RESULT

Elapsed time is measured.

```
10 REM ****************
20 REM *               *
30 REM * ELAPSED TIME  *
40 REM *               *
50 REM ****************
60 GOTO 240
70 REM ---------------------------
80 REM  + + VARIABLES + +
90 REM      MIN: ELAPSED MINUTES
100 REM     SEC: ELAPSED SECONDS
110 REM     TS  : JIFFIES AT START
120 REM     TF  : JIFFIES AT FINISH
130 REM
140 REM ---------------------------

150 REM *** SUBROUTINE ***

160 TF=TI
170 FS=INT((TF-TS)/60)
180 MIN=INT(FS/60)
190 SEC=FS-(MIN*60)
200 PRINT TAB(2)"IT TOOK YOU ";MIN
210 PRINT TAB(2)"MIN. AND ";SEC;"SEC."
220 RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 TS=TI
250 GET A$:IF A$="" GOTO 250
260 GOSUB 160
270 GOTO 240
```

TIMER

WHAT IT DOES

Sets computer as a timer.

Variables

- FH$: Finish hour
- FM$ Finish minutes
- FS$ Finish seconds
- FT$ Finish time.

How To Use The Subroutine

Having the computer signal us at some future time can be a useful function. This subroutine sets the real-time clock to the correct time, then asks what time we 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.

Note that the computer can't do any other functions while this subroutine is running. You might want to change it so that the line which compares the current time with the target time (IF VAL(TIME$) > VAL(FT$) GOTO...) is imbedded within your main program and checked 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.

Line By Line Description

Lines 160 to 190: User enters current hour, which is checked to make sure it is less than 24.

Lines 200 to 280: If number of hours is less than 12, user is asked if A.M. or P.M. is meant. If afternoon is specified, 12 is added to the value of HR, to conform with 24 hour military-style time.

Lines 290 to 300: User enters current minutes, which are checked to see that they do not exceed 59.

Lines 310 to 320: If value of HR or MINUTES is less than ten, a leading zeros are added to fill out the string, e.g., "000900" (for 12:09 A.M.).

Line 330: TI$ is set equal to the hour and minutes input, plus 0 seconds.

Lines 350 to 390: User enters number of hours and minutes that are to be timed. These values are stored in HR and MN, respectively.

Lines 400 to 410: Timing cycle starts, and current time reading is taken.

Lines 420 to 430: Current hour and minutes figured.

Lines 440 to 450: The finish hour and minutes are figured by adding current hour and minutes to the desired timing interval.

Lines 460 to 480: Finish hour, minutes, and seconds in string form are constructed.

Lines 490 to 500: If either minutes or hours at finish time are less than 10, leading zeroes are added.

Line 510: FT$, the finish time, is constructed.

Lines 520 to 540: Finish time and current time are displayed to the screen.

Lines 550 to 560: If current time, TIME$, does not equal the finish time, FT$, then subroutine goes back to line 520, until time is up. Then, control drops to line 570, where "time is up" message is printed to screen.

You Supply

Answer the requests from the prompts. You also might want to call a sound subroutine of your choice at Line 570, to provide an audible alarm. Several sound routines are provided in the next section of this book.

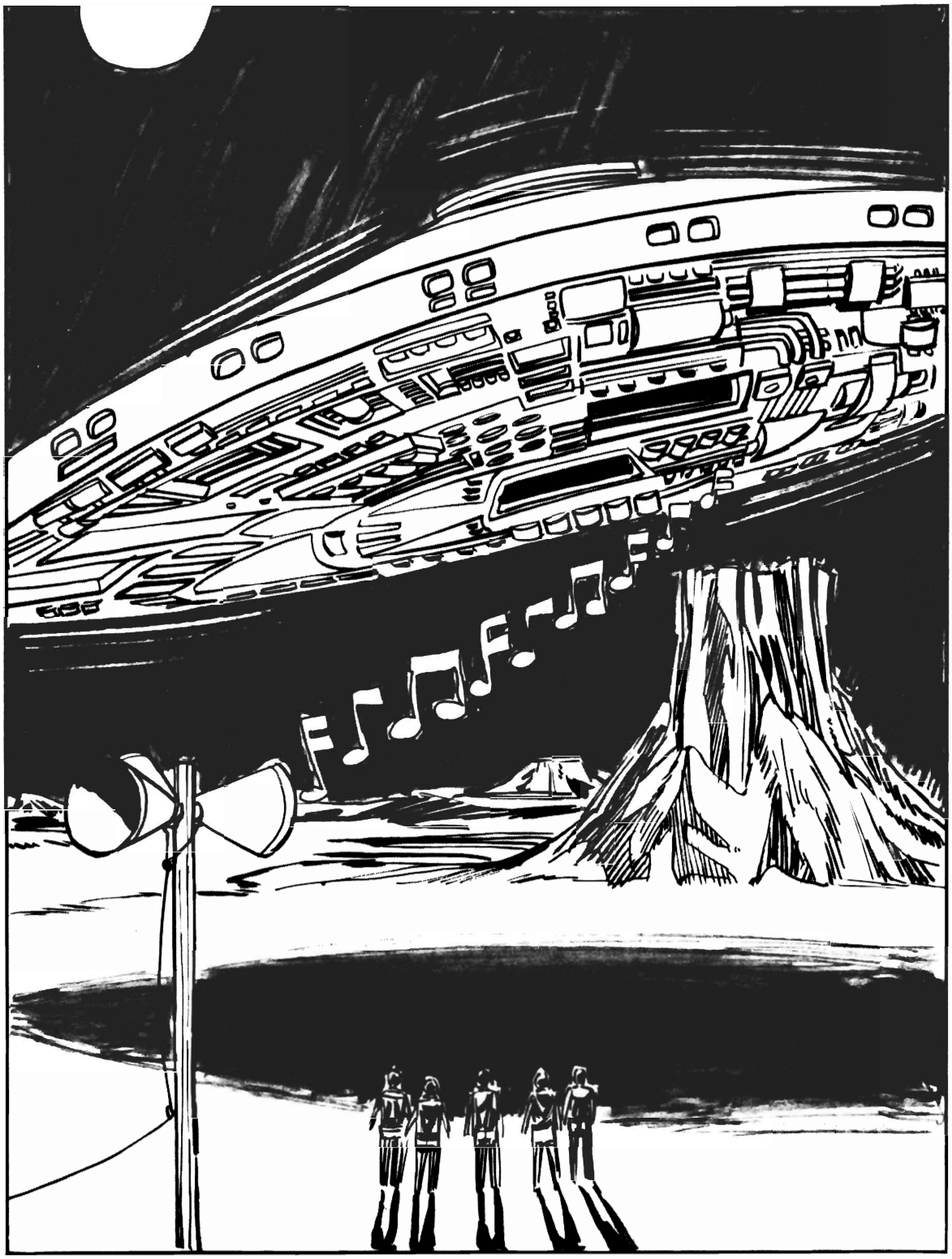
RESULT

Commodore 64 signals at end of requested time interval.

```
10 REM *********
20 REM *       *
30 REM * TIMER *
40 REM *       *
50 REM *********
60 GOTO 590
70 REM ----------------------------
80 REM       + + VARIABLES + +
90 REM     FH$: FINISH HOUR
100 REM    FM$: FINISH MINUTES
110 REM    FS$: FINISH SECONDS
120 REM    FT$: FINISH TIME
130 REM
140 REM ----------------------------

150 REM *** SUBROUTINE ***
```

```
160 PRINT "SET CURRENT TIME"
170 PRINT "ENTER CURRENT HOUR :"
180 INPUT HR$
190 IF VAL(HR$)>23 GOTO 180
200 IF VAL(HR$)>13 GOTO 290
210 PRINT"ENTER A FOR A.M."
220 PRINT"ENTER P FOR P.M."
230 PRINT"CHOICE:"
240 GET A$:IF A$="" GOTO 240
250 IF A$="P" THEN GOTO 280
260 IF A$="A" GOTO 290
270 GOTO 240
280 HR$=MID$(STR$(12+VAL(HR$)),2)
290 PRINT "ENTER CURRENT MINUTES:":INPUT MN$
300 IF VAL(MN$)>59 GOTO 290
310 IF VAL(HR$)<10 THEN HR$="0"+HR$
320 IF VAL(MN$)<10 THEN MN$="0"+MN$
330 T$=HR$+MN$+"00"
340 TIME$=T$

345 REM *** TIME TO BE MEASURED ***

350 PRINT" TOTAL TIME TO BE COUNTED:"
360 INPUT" ENTER HOURS:";HR$
370 INPUT" ENTER MINUTES:";MN$
380 HR=VAL(HR$)
390 MN=VAL(MN$)
400 PRINT TAB(4)"TIMING CYCLE"
410 T$=TIME$
420 HN=VAL(LEFT$(T$,2))
430 MP=VAL(MID$(T$,3,2))
440 FM=MP+MN:IF FM>59 THEN FM=FM-60:HN=HN+1
450 FH=HN+HR:IF FH>23 THEN FH=FH-24
460 FH$=MID$(STR$(FH),2)
470 FM$=MID$(STR$(FM),2)
480 FS$=MID$(T$,5)
490 IF VAL(FH$)<9 THEN FH$="0"+MID$(FH$,2):
    IF VAL(FH)<1 THEN FH$="00"
500 IF VAL(FM$)<9 THEN FM$="0"+MID$(FM$,2):
    IF VAL(FM$)<1 THEN FM$="00"
510 FT$=FH$+FM$+FS$
520 PRINT"(CLR)"
530 PRINT"FINISH TIME: ";FH$;":";FM$
540 PRINT"CURRENT TIME: ";LEFT$
    (TIME$,2);":";MID$(TIME$,3,2);":"
    ;RIGHT$(TIME$,2)
550 IF VAL(TIME$)>VAL(FT$) GOTO 570
560 GOTO 520
570 PRINT"TIME IS UP.":RETURN
```

```
580 REM *** YOUR PROGRAM STARTS HERE ***

590 PRINT
600 GOSUB 160
```



4 Using Sound

The Commodore 64 has some of the best sound capabilities of any home 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 6581 music synthesizer chip built into the Commodore 64. Earlier computers without this chip had much more primitive sound generation capabilities. Some can produce sounds through only a single voice. The VIC-20 has 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 64's synthesizer chip provides.

Yes, the Commodore 64 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 wave form of each musical note that helps give various musical instruments their distinctive "timbre", or sound quality. Your Commodore 64 can play a note with four different waveforms, plus "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," while the speed of decline to the middle range is called "decay." That middle volume, called "sustain," can also be controlled by the Commodore 64. 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 64.

That's the good news. The bad news is that BASIC V.2 does not have any commands to perform this magic, other than tedious POKEs to various memory locations. For example, to play a single note in one voice to your specifications, you might have to include the following program lines:

```
10 POKE 54296,15        (Sets volume)
20 POKE 54273,34        (Plays middle
30 POKE 54272,75        C on Voice 1)
40 POKE 54276,65        (Sets pulse wave form)
50 POKE 54275,12        (Sets hi pulserate)
60 POKE 54274,200       (Sets low pulserate)
70 POKE 54277,130       (Sets attack/decay rate)
80 POKE 54278,66        (Sets sustain/release rate)
```

Length of the note is yet another parameter that must be taken care of by your programming, with a FOR-NEXT delay loop, or some other means. It's nice to have that much control over a sound, isn't it? But who would want to do all that programming three times over (for each voice)? Granted, every line shown above is not needed for every note. Some are required only when you want to change some of the parameters for a given voice. However, keeping track can be a pain, even when "offsets" and special programming tricks are used to reduce the work.

There are several music "synthesizer" programs that will allow you to experiment with the sound capabilities of this computer. These are long, complex, and beyond the scope of this subroutine "cookbook."

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 chapter contains five subroutines that have broad application in many games programs—or which can 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 with 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.

In general, music and sound effects generation on the Commodore 64 is accomplished by POKing various values to several memory locations, or registers. For example, the volume of all three voices can be controlled by POKing values from 0 to 15 in location 54296. Unfortunately, the Commodore 64 does not allow you to control the volume of each voice separately. A 0 will "turn off" all the voices at once, while POKing 15 to 54296 will produce maximum volume.

The three music voice registers are located in pairs at 54272/54273, 54279/54280, and 54286/53287. POKing pairs of values to these registers will produce musical notes that more or less correspond to the notes of the musical scale. The actual values are listed on pages 163-164 of the Commodore 64 User's Guide supplied with the computer.

In addition, you also need to POKE a waveform, to registers located at 54276, 54283, and 54290, for the three voices, respectively. A value of 17 POKed to any of these registers will produce a "triangle" type waveform, while 33 will generate a "sawtooth" waveform. A 65 produces a pulse wave, with 129 delivering nothing but a random-sounding noise. Try substituting each of these in various sound routines to hear the changes for yourself.

But wait! There's more. When using pulse waveforms, each voice has yet one more pair of registers that need to be POKEd to supply pulse rate. Finally, you may choose attack and decay rates. Each are separate values but, since the combinations of the two produce unique numbers, you may add your chosen attack rate to your chosen decay rate and POKE a single number to one register for each voice. Sustain/release are set up in a similar manner.

Another factor, the length of time each note is played, is controlled by your program. A note can begin when you POKE a value 1 to 15 into the volume register, and end when a 0 is POKEd to that location. In between, you might have a FOR-NEXT loop providing the desired amount of delay. This could be varied by your program, as you desire:

```
10 INPUT "LENGTH OF NOTE WANTED (1 TO 10)";DELAY
20 GOSUB 100
...
...
100 POKE VOLUME,15
110 POKE VOICE,128
120 FOR N=1 TO DELAY
130 NEXT N
150 POKE VOLUME,0
160 RETURN
```

In this example, the note will play for as long as the FOR-NEXT loop in lines 120 to 130 takes to execute. Then, the POKE in line 150 will turn volume to 0, effectively silencing the note. Notice that the sound is still playing; we just cannot hear it. To repeat that note, it is necessary only to set the volume again. The POKE to 36874 is not needed.

Therein we have the second way of determining the length of a note. POKE the voice register with a 0 and that note is turned off and inaudible, regardless of what the volume register is set at. Make this substitution in the example above:

```
10 INPUT "LENGTH OF NOTE WANTED (1 TO 10)";DELAY
20 GOSUB 100
...
...
100 POKE VOLUME,15
110 POKE VOICE,128
120 FOR N=1 TO DELAY
130 NEXT N
150 POKE VOICE,0
160 RETURN
```

Now, the volume is still on high, but we cannot hear the note because the voice register is turned off. The first example used is a way of controlling the length of all the notes being played by all the voices simultaneously. This second example shows how to vary the length of a note played by an individual voice, without altering those of the other voices. Combinations of these two can be used to produce sophisticated sounds and melodies.

MUSIC

WHAT IT DOES

Uses keyboard to generate various musical notes.

Variables

- VOLUME: Address to POKE volume
- VCE: Voice currently being used
- LTH: Length of note.

How To Use Subroutine

As written, it will produce a note of the musical scale each time one of the keys of the Commodore 64 are pressed. You could alter to produce those same notes when other keys are DESCRIPTION or when certain program conditions are met.

An array, P(n), stores the numbers which must be POKEd to any sound register to produce a note from the scale. Array N(n) stores the CHR$ value of the keys which correspond to those notes.

Line By Line Description

Lines 140 to 160: Define volume register, voice to be used, and length of note as variables.

Line 170: Set sawtooth waveform.

Line 180: DIMension array to store names of notes, note values, and keys which correspond to them on the Commodore 64 keyboard.

Lines 190 to 270: Read note names, note values, and keys which correspond to them into arrays.

Lines 330 to 340: Set attack/decay and sustain/release.

Line 350: Wait for key to be pressed.

Line 360: If key was RETURN, subroutine is over.

Lines 370 to 390: Check to see if key pressed was legal note.

Line 410: POKE volume to maximum.

Line 420: POKE voice chosen with appropriate note.

Line 430: Delay, providing length of note.

Line 440: Turn the volume off.

Line 450: Turn the note off.

Line 460: Go back and wait for next key to be pressed.

You Supply

LTH controls the length of the note. A larger value produces a longer note. You change to another voice by making the following substitutions:

For Voice #2:

```
150 VCE=54275
170 POKE 54283,33
330 POKE 54284,128
340 POKE 54285,128
```

For Voice #3:

```
150 VCE=54287
170 POKE 54290,33
330 POKE 54291,128
340 POKE 54292,128
```

RESULT

Music played from Commodore 64 synthesizer.

```
10 REM *****************
20 REM *                 *
30 REM * MUSICAL NOTES   *
40 REM *                 *
50 REM *****************
60 REM ----------------------------
70 REM     + + VARIABLES + +
80 REM     VOLUME:    LOUDNESS REGISTER
90 REM     VCE:       VOICE
100 REM    LTH:       LENGTH OF NOTE
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 VOLUME=54296
150 VCE=54273
160 LTH=200
170 POKE 54276,33
180 DIM NME(14),NT$(14),P(14)
190 FOR N=1 TO 14
200 READ NME(N)
210 NEXT N
220 FOR N=1 TO 14
230 READ NT$(N)
240 NEXT N
250 FOR N=1 TO 14
260 READ P(N)
270 NEXT N
280 DATA 3,3,4,4,5,6,6,7,7,1,1,2,3,3
290 DATA A,W,S,E,D,F,T,G,Y,H,U,J,K,O
300 DATA 34,36,38,40,43,45,48,51,54,57,61,64,68,72
310 GOTO 480

320 REM *** SUBROUTINE ***

330 POKE 54277,128
340 POKE 54278,128
350 GET A$:IF A$=" " GOTO 350
360 IF A$=CHR$(13) THEN RETURN
370 FOR N=1 TO 14
380 IF A$=NT$(N) GOTO 410
390 NEXT N
400 GOTO 350
410 POKE VOLUME,15
420 POKE VCE,P(N)
430 FOR DELAY=1 TO LTH:NEXT DELAY
440 POKE VOLUME,0
450 POKE VCE,0
460 GOTO 350
```

```
470 REM *** YOUR PROGRAM STARTS HERE ***

480 GOSUB 330
```

SIREN

WHAT IT DOES

Siren sound routine to produce sounds for games, other applications.

Variables

- Volume: Volume register
- H1: Voice #1, high
- L1: Voice #1, low
- W1: Wave form voice #1
- A1: Attack/decay voice #1
- S1: Sustain/release voice #1
- SAW: Sawtooth waveform.

How To Use Subroutine

Call subroutine when siren sound is desired.

Line By Line Description

Line 160: Define volume register as variable.

Lines 170 to 210: Define registers to POKE notes, waveform, attack/decay and sustain/release parameters.

Line 220: Define sawtooth waveform value as variable, SAW.

Lines 250 to 290: Poke volume, waveform, attack/decay, and sustain/release values to proper registers.

Lines 300 to 390: POKE sounds to produce siren effect.

You Supply

No user changes required. However, voice of siren may be changed by making the following substitutions:

For Voice #2:

H1: 54275
L1: 54274
W1: 54276
A1: 54277
S1: 54278

For Voice #3:

H1: 54280
L1: 54279
W1: 54283
A1: 54284
S1: 54285

RESULT

Siren sound emitted for game play or other applications.

```
10 REM *********
20 REM *       *
30 REM * SIREN *
40 REM *       *
50 REM *********
55 REM ------------------------------------
60 REM    + + VARIABLES + +
70 REM     VOLUME: VOLUME REGISTER
80 REM     H1:     VOICE #1, HIGH
90 REM     L1:     VOICE #1, LOW
100 REM    W1:     WAVE FORM VOICE 1
110 REM    A1:     ATTACK/DECAY VOICE 1
120 REM    S1:     SUSTAIN/RELEASE VOICE 1
130 REM    SAW:    SAWTOOTH WAVEFORM
140 REM ----------------------------------

150 REM *** INITIALIZE ***

160 VOLUME=54296
170 H1=54273
180 L1=54272
190 W1=54276
200 A1=54277
210 S1=54278
220 SAW=33
230 GOTO 410

240 REM *** SUBROUTINE ***
```

```
250 POKE VOLUME,15
260 POKE W1,SAW
270 POKE A1,16
280 POKE S1,16
290 POKE H1,36
300 FOR J=1 TO 5
310 FOR N=34 TO 72
320 POKE H1,N
330 POKE L1,N+75
340 NEXT N
350 FOR N=72 TO 34 STEP -1
360 POKE H1,N
370 FOR I=1 TO 40:NEXT I
380 NEXT N
390 NEXT J

400 REM *** YOUR PROGRAM STARTS HERE ***

410 GOSUB 250
```

COMPUTER SOUND

WHAT IT DOES

Computer-like sound, for games, other applications.

Variables

- VOLUME: Volume register
- H1-H3: Voices #1-3, high
- L1-L3: Voices #1-3, low
- W1-W3: Waveform voice #1-3
- A1-A3: Attack/decay voice #1-3
- S1-S3: Sustain/release voice #1-3
- SAW: Sawtooth waveform.

How To Use Subroutine

Call subroutine when sound is desired. Random, computer-like sound is produced. Subroutine selects random note for all three voices, and plays a number of them, determined by LTH.

Line By Line Description

Line 170: Define length of routine.

Line 180: Define volume register as variable.

Lines 190 to 330: Define registers to POKE notes, waveform, attack/decay and sustain/release parameters, for all three voices..

Line 220: Define triangle and sawtooth waveform value as variables, TRIANGLE and SAW.

Line 360: Set volume to maximum.

Lines 370 to 460: Set envelope parameters for all three voices.

Line 490: Start loop from 1 to LTH.

Lines 500 to 520: Choose random note for each voice.

Lines 530 to 550: POKE the random note to each voice.

Line 560: Wait random period of time.

Line 570: Repeat loop.

Line 580: Turn volume off.

You Supply

LTH may be redefined to produce longer computer sound effect.

RESULT

Computer-like sounds emitted for game play or other applications.

```
10 REM ******************
20 REM *                 *
30 REM * COMPUTER SOUND *
40 REM *                 *
50 REM ******************
60 REM ----------------------------------------
70 REM     + + VARIABLES + +
80 REM     VOLUME: VOLUME REGISTER
90 REM     H1-H3:  VOICES #1-3, HIGH
100 REM    L1-L3:  VOICES #1-3, LOW
110 REM    W1-W3:  WAVE FORM VOICE 1-3
120 REM    A1-A3:  ATTACK/DECAY VOICE 1-3
130 REM    S1-S3:  SUSTAIN/RELEASE VOICE 1-3
140 REM    SAW:    SAWTOOTH WAVEFORM
150 REM ----------------------------------------

160 REM *** INITIALIZE ***
```

```
170 LTH=100
180 VOLUME=54296
190 H1=54273
200 L1=54272
210 W1=54276
220 H2=54280
230 L2=54279
240 W2=54283
250 H3=54287
260 L3=54286
270 W3=54290
280 A1=54277
290 A2=54284
300 A3=54291
310 S1=54278
320 S2=54285
330 S3=54292
340 TRIANGLE=17
350 SAW=33
360 POKE VOLUME,15
370 POKE W1,TRI
380 POKE W2,TRI
390 POKE W3,SAW
400 POKE A1,128
410 POKE S1,128
420 POKE A2,128
430 POKE S2,128
440 POKE A3,128
450 POKE A4,128
460 POKE H1,36
470 GOTO 610

480 REM *** SUBROUTINE ***

490 FOR N=1 TO LTH
500 R1=INT(RND(1)*35)
510 R2=INT(RND(1)*35)
520 R3=INT(RND(1)*35)
530 POKE L1,R1+70:POKE H1,R1+30
540 POKE L2,R2+70:POKE H2,R2+30
550 POKE L3,R3+70:POKE H3,R3+30
560 FOR J=1 TO INT(RND(1)*15):NEXT J
570 NEXT N
580 POKE VOLUME,0
590 RETURN

600 REM *** YOUR PROGRAM STARTS HERE ***

610 GOSUB 490
```

SAUCER SOUND

WHAT IT DOES

Flying saucer-like sound, for games, other applications.

Variables

- VOLUME: Volume register
- H1: Voice #1, high
- L1: Voice #1, low
- W1: Waveform voice #1
- A1: Attack/decay voice #1
- S1: Sustain/release voice #1
- SAW: Sawtooth waveform.

How To Use Subroutine

Call subroutine when sound is desired. Wavering, flying saucer-like sound is produced.

Line By Line Description

Line 170: Define volume register as variable.

Lines 180 to 220: Define registers to POKE notes, waveform, attack/decay and sustain/release parameters.

Line 230: Define sawtooth waveform value as variable, SAW.

Lines 260 to 290: Poke volume, waveform, attack/decay and sustain/release values to proper registers.

Lines 300 to 370: POKE sounds to produce saucer effect.

You Supply

No user changes recommended.

RESULT

Flying saucer-like sounds emitted for game play or other applications.

```
10 REM ****************
20 REM *               *
30 REM * SAUCER SOUND *
40 REM *               *
50 REM ****************
60 REM ---------------------------------
70 REM     + + VARIABLES + +
80 REM     VOLUME: VOLUME REGISTER
90 REM     H1:     VOICE #1, HIGH
100 REM    L1:     VOICE #1, LOW
110 REM    W1:     WAVE FORM VOICE 1
120 REM    A1:     ATTACK/DECAY VOICE 1
130 REM    S1:     SUSTAIN/RELEASE VOICE 1
140 REM    SAW:    SAWTOOTH WAVEFORM
150 REM ------------------------------

160 REM *** INITIALIZE ***

170 VOLUME=54296
180 H1=54273
190 L1=54272
200 W1=54276
210 A1=54277
220 S1=54278
230 SAW=33
240 GOTO 400

250 REM *** SUBROUTINE ***

260 POKE W1,SAW
270 POKE A1,128
280 POKE S1,128
290 POKE VOLUME,15
300 FOR N=1 TO 100
310 POKE H1,N/2+30
320 POKE L1,N/2+75
330 FOR G=1 TO 20:NEXT G
340 NEXT N
350 POKE VOLUME,0
360 POKE W1,0
370 POKE A1,0
380 RETURN

390 REM *** YOUR PROGRAM STARTS HERE ***

400 GOSUB 260
```

KLAXON

WHAT IT DOES

Klaxon sound routine to produce sounds for games, other applications.

Variables

- VOLUME: Volume register
- H1: Voice #1, high
- L1: Voice #1, low
- W1: Waveform voice #1
- A1: Attack/decay voice #1
- S1: Sustain/release voice #1
- SAW: Sawtooth waveform.

How To Use Subroutine

Call subroutine when klaxon is desired.

Line By Line Description

Line 180: Define volume register as variable.

Lines 190 to 250: Define registers to POKE notes, waveform, attack/decay and sustain/release parameters.

Line 260: Define sawtooth waveform value as variable, SAW.

Lines 290 to 320: Poke volume, waveform, attack/decay, and sustain/release values to proper registers.

Lines 330 to 380: POKE sounds to produce klaxon effect.

Line 390: Turn volume off.

You Supply

No user changes required.

RESULT

Klaxon sound emitted for game play or other applications.

```
10 REM **********
20 REM *        *
30 REM * KLAXON *
40 REM *        *
50 REM **********
60 REM --------------------------------
70 REM     + + VARIABLES + +
80 REM     VOLUME:  VOLUME REGISTER
90 REM     H1:      VOICE #1, HIGH
100 REM    L1:      VOICE #1, LOW
110 REM    W1:      WAVE FORM VOICE 1
120 REM    A1:      ATTACK/DECAY VOICE 1
130 REM    S1:      SUSTAIN/RELEASE VOICE 1
140 REM    SAW:     SAWTOOTH WAVEFORM
150 REM --------------------------------------

170 REM *** INITIALIZE ***

180 VOLUME=54296
190 H1=54273
200 L2=54272
210 W1=54276
220 A1=54277
230 S1=54278
240 S2=54285
250 S3=54292
260 SAW=33
270 GOTO 420

280 REM *** SUBROUTINE ***

290 POKE VOLUME,15
300 POKE W1,SAW
310 POKE POKE A1,128
320 POKE S1,128
330 FOR J=1 TO 5
340 FOR N=1 TO 100
350 POKE H1,N/2+30
360 POKE L1,N/2+70
370 NEXT N
380 NEXT J
390 POKE VOLUME,0
400 RETURN

410 REM *** YOUR PROGRAM STARTS HERE ***

420 GOSUB 290
```

GUNFIRE

WHAT IT DOES

Gunfire sound routine to produce sounds for games, other applications.

Variables

- VOLUME: Volume register
- H1: Voice #1, high
- L1: Voice #1, low
- W1: Waveform voice #1
- A1: Attack/decay voice #1
- S1: Sustain/release voice #1
- SAW: Sawtooth waveform.

How To Use Subroutine

Call subroutine when gunfire is desired.

Line By Line Description

Line 150: Define volume register as variable.

Lines 160 to 190: Define registers to POKE notes, waveform, attack/decay parameters.

Lines 220 to 310: POKE sounds to produce gunfire effect.

Line 330: Access the subroutine.

Lines 340 to 360: Choose random delay between shots.

Line 370: Repeat shot.

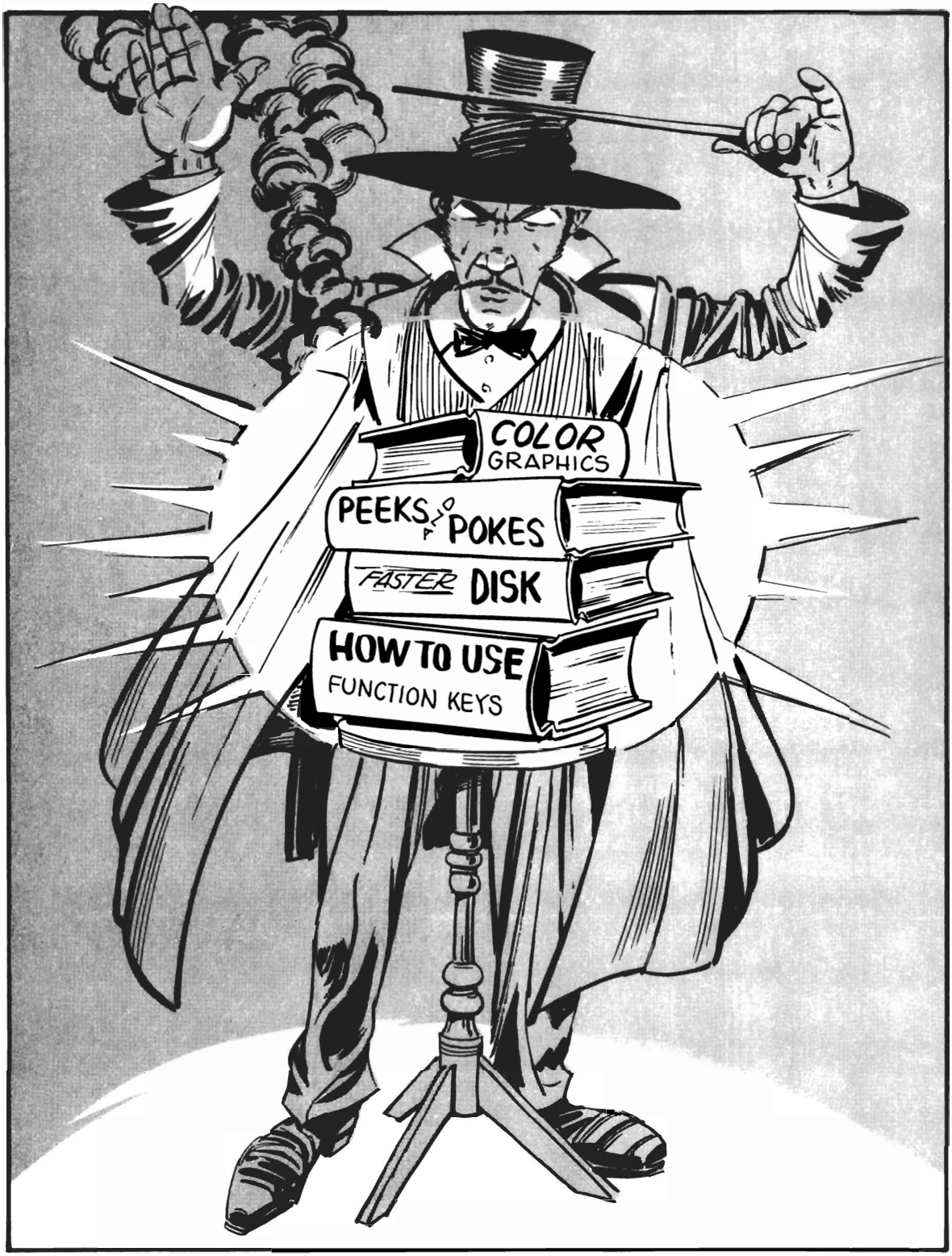
You Supply

No user changes required. Noise voice must be used.

RESULT

Gunfire sound emitted for game play or other applications.

```
10 REM ***********
20 REM *         *
30 REM * GUNFIRE *
40 REM *         *
50 REM **********
55 REM --------------------------------
60 REM     + + VARIABLES + +
70 REM     VOLUME:   VOLUME REGISTER
80 REM     H1:       VOICE #1, HIGH
90 REM     L1:       VOICE #1, LOW
100 REM    W1:       WAVE FORM VOICE 1
110 REM    A1:       ATTACK/DECAY VOICE 1
120 REM    S1:       SUSTAIN/RELEASE VOICE 1
130 REM ------------------------------

140 REM *** INITIALIZE ***

150 VOLUME=54296
160 W1=54276
170 A1=54277
180 H1=54273
190 L1=54272
200 GOTO 330

210 REM *** SUBROUTINE ***

220 FOR N=15 TO 1 STEP-1
230 POKE VOLUME,N
240 POKE W1,129
250 POKE A1,15
260 POKE H1,40
270 POKE L1,200
280 NEXT N
290 POKE W1,0
300 POKE A1,0
310 RETURN

320 REM *** YOUR PROGRAM STARTS HERE ***

330 GOSUB 220
340 R=INT(RND(1)*150)
350 FOR N=1 TO R
360 NEXT N
370 GOTO 330
```


COLOR
GRAPHICS
PEEKS & POKES
FASTER DISK
HOW TO USE
FUNCTION KEYS

5 Other Commodore 64 Tricks

Here is a group of routines that will let you perform specific tasks peculiar to Commodore computers. Some of these consist of a couple simple POKEs and, admittedly, could be carried out almost as easily from the keyboard in direct mode. However, they have been included as subroutines for several reasons.

First, it is typical for users to forget the most needed PEEKs and POKEs particularly at the times when they are most needed. It is time-consuming to look them up. In addition, some POKEs need an argument. For example, you could look up the proper number to POKE to change the screen/border color combination and then type that in from the keyboard. A subroutine in this section will calculate the numbers for you and do all the work. You just press the color key at the top of the keyboard. That subroutine could be used in many different graphics programs. Or, you include it in games.

You may also want to perform a function under program control, such as altering which keys repeat automatically and which don't. By having the proper subroutine available, this task can be carried out quickly by your program, at the user's discretion.

A variety of housekeeping subroutines are included in this section. One will allow you, under program control, to set the color of the screen, border, and characters. Another dissects the proper nybble from a color memory byte and tells you what color has been set in a given screen memory location.

Also included is a subroutine that makes 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, while F3 quits the game entirely. F5 could clear the screen, while F7 would exchange sides. Other function keys could display the score or do some other function.

Remembering all those complex disk commands is made a bit easier by one subroutine that takes care of them for you.

"Display Tester" prints horizontal and vertical lines on the screen, as well as a series of color bars that allow you to adjust your color television or monitor for best image.

Those wanting to use user-defined character sets will like the drop-in module that allows changing up to three characters, while leaving the others unaltered. Some tips on calculating the necessary numbers while designing your characters are also included.

The final module in this group allows turning your computer's repeat key feature on and off under program control.

SCREEN, BORDER, AND COLOR CHANGER

WHAT IT DOES

Changes color combination of screen, border and characters.

Variables

- SCREEN: Screen color
- BRDR: Border color
- CCHARS: Character color.

How To Use Subroutine

One useful feature for games and other programs is the ability of the Commodore 64 to change the colors of the screen, border, and characters as required. In the 64's companion computer, the VIC-20, this can be accomplished by looking up the proper number to POKE to a location in a table. This subroutine takes care of all the POKing automatically. Just specify the colors desired, using actual names, for screen, border, and character colors.

Various colors, corresponding to the colors on the keytops, have been defined as variables. That is, DARK = 1 (black), WHITE = 2, and so forth. DARK is used instead of BLACK because the Commodore 64 recognizes only the first two letters of a variable name. Therefore BLACK and BLUE are the same variable, BL.

In this subroutine the screen has been predefined as red, border as white, and characters as white.

Line By Line Description

Lines 150 to 180: Define the colors as variables.

Lines 190 to 210: Define memory addresses to POKE to change background, border, and character colors.

Line 250: Change background color.

Line 260: Change character color.

Line 270: Change border color.

Lines 300 to 320: Define desired combination.

You Supply

A value for SCREEN in the range 1 through 8. For convenience, the actual color names may be used instead. Also, supply a value for BRDER, using the same scheme. To change the character colors, define CCHAR as the desired color in the same manner.

RESULT

Screen, border, and character colors change.

```
10 REM ******************
20 REM *                *
30 REM * SCREEN, BORDER, *
40 REM *  COLOR CHANGER  *
50 REM *                *
60 REM ******************
70 REM ---------------------------
80 REM       + + VARIABLES + +
90 REM     SCREEN: SCREEN COLOR
100 REM    BRDER:  BORDER COLOR
110 REM    CCHARS: COLOR OF CHARACTERS
120 REM
130 REM ---------------------------

140 REM *** INITIALIZE ***

150 DARK=1:WHITE=2
160 RED=3:CYAN=4
170 PURPLE=5:GREEN=6
180 BLUE=7:YELLOW=8
190 BACKGROUND=53281
200 SURROUND=53280
210 LTTERS=646
220 GOTO 300

240 REM *** SUBROUTINE ***

250 POKE BACKGROUND,SCREEN-1
260 POKE LTTERS,CC-1
270 POKE SURROUND,BRDR-1
280 RETURN

290 REM *** YOUR PROGRAM STARTS HERE ***

300 SCREEN=DARK
312 BRDER=BLUE
323 CCHARS=WHITE
330 GOSUB 250
```

COLOR PEEKER

WHAT IT DOES

Determines character color in a certain memory location.

Variables

- COLR$(n): Color
- ADDRESS: Location to PEEK.

How To Use Subroutine

It may be convenient for some programming applications to determine what color is in a certain spot on the screen. For example, you may write a game program where the obstacles are various characters, but they are all blue, while the player's ship is another color. To detect a collision, check to see if the color of the space about to be occupied is blue.

However, the color memory of the Commodore 64 stores the colors in "nybbles" rather than full bytes. If a full 256 colors were available with the Commodore 64, then an entire byte would be required to assign one number to each color, since the largest number that can be expressed with a single byte is 255. This is not the case. In fact, 16 colors are used by the Commodore 64 (colors 0 through 15), and can be expressed by the first four bits of a given color memory location byte. The other four bits are of no concern and, in fact, contain random information (termed "garbage" in computer slang.) Half a byte is termed, jocularly, a "nybble" and we need a special technique to read only that part of a full byte.

POKing a color to a location produces the expected results. If you want to change the color of location X to white by POKE X,1, the technique works just fine. However, PEEKing that same location could yield several different decimal numbers, all of which, expressed in binary, end with "0001." That is, 10100001, 01100001, and 1110001 might be returned from a color memory location, but a four would mean that the color in that place on the screen would be white. Only the last four bits "count" when working with this type of nybble.

This 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, while NOT complements each bit. Boolean math is discussed in more detail later in this book. For this section, we need to know more of how AND works. The particular bit or bits looked at depends on the number we choose to AND with. You will remember in Chapter 2, various numbers were ANDed with a PEEK to determine the status of a given joystick switch bit. For color memory, we want to know about all of the first four bits (reading left to right) of a byte, so we AND with 15, which is 00001111 in binary. Here are a few examples:

```
COLOR MEMORY BYTE:  11010001
AND with 15         00001111
RESULT:             00000001

COLOR MEMORY BYTE   01100001
AND with 15         00001111
RESULT:             00000001
```

You'll see by following along the columns that the result equals one only when the corresponding bit in both the color memory byte and 15 equal 1. Since the second four bits of the binary equivalent of 15 decimal are always 0, the result will always be 0. Since the first four bits of 15 are all 1's, the result will be 1 only if there is also a 1 in the color memory byte.

Since 11010001 and 01100001 both will produce white in color memory, and we have returned a value of 00000001 by ANDing with 15, the operation has effectively filtered out the unwanted bits.

Line By Line Description

Line 70: DIMension an array to store the names of the colors.

Lines 80 to 110: Read those names into the array from data statements.

Line 130: Find address of color memory.

Line 140: Define CHOICE, location to be PEEKed.

Line 150: Add CHOICE to start of color memory.

Line 160: PEEK CHOICE, then AND it with 15.

Line 290: Print color name from array COLR$(n).

You Supply

The memory location you wish to PEEK, variable ADDRESS. This is determined by adding CHOICE (the offset of the address you wish to look at) to CSCREEN (the start of color memory). In the example, location 25 is used.

RESULT

Variable C will store the number of the key that is the same as that in location ADDRESS. String array COLR$(C) will print the name of that color.

```
10 REM ****************
20 REM *               *
30 REM * COLOR PEEKER *
40 REM *               *
50 REM ****************

60 REM *** INITIALIZE ***
```

```
70 DIM COLR$(15)
80 FOR N=0 TO 15
90 READ COLR$(N)
100 NEXT N
110 DATA BLACK,WHITE,RED,CYAN,PURPLE,GREEN,
    BLUE,YELLOW,ORANGE,LT. ORANGE
120 DATA PINK,LT.CYAN,LT.PURPLE,LT.GREEN,LT.BLUE,LT.YELLOW
130 CSCREEN=55296
140 CHOICE=25
150 ADDRESS=CSCREEN+CHOICE
160 GOTO 270
170 REM ----------------------------
180 REM       + + VARIABLES + +
190 REM    COLR$(N): COLOR
200 REM    ADDRESS:  LOCATION TO PEEK
210 REM
220 REM ----------------------------

230 REM *** SUBROUTINE ***

240 C=PEEK(ADDRESS)AND15
250 RETURN

260 REM *** YOUR PROGRAM STARTS HERE ***

270 PRINT
280 GOSUB 240
290 PRINT"COLOR : ";COLR$(C)
```

DISPLAY TESTER

WHAT IT DOES

Displays color bars, horizontal parallel lines, and vertical parallel lines, to check out video display.

Variables

None.

How To Use Subroutine

This subroutine will provide a quick check of the Commodore 64 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 lines to check your entire screen. Pressing any key will clear the screen and print vertical bars instead. Pressing

the key another time will print color bars, displayed in the same order as the colors arrayed on the number keys on the keyboard. You can then adjust your set or monitor for best color and contrast.

Line By Line Description

Lines 70 to 90: Read color codes into array.

Line 100: DATA of codes to print given colors.

Line 180: Clear screen.

Lines 190 to 220: Print horizontal lines to screen, then wait for a key to be pressed.

Lines 230 to 270: Clear screen, then print vertical bars, and wait for a key to be pressed.

Lines 280 to 350: Print color bars to screen.

You Supply

Set adjustments.

RESULT

Better display quality or rough diagnosis of a problem.

```
10 REM ******************
20 REM *                 *
30 REM * DISPLAY TESTER *
40 REM *                 *
50 REM ******************

60 REM *** INITIALIZE ***

70 FOR N=1 TO 8
80 READ COLR(N)
90 NEXT N
100 DATA 144,5,28,159,156,30,31,158
110 GOTO 410
120 REM ----------------------------
130 REM       + + VARIABLES + +
140 REM          NONE
150 REM
160 REM ----------------------------

170 REM *** SUBROUTINE ***
```

```
180 PRINT CHR$(147)
190 FOR N=1 TO 999
200 PRINT CHR$(196);
210 NEXT N
220 GET A$:IF A$="" GOTO 220
230 PRINT CHR$(147)
240 FOR N=1 TO 999
250 PRINT CHR$(212);
260 NEXT N
270 GET A$:IF A$="" GOTO 270
280 PRINT CHR$(147)
290 FOR ROW=1 TO 24
300 FOR N=1 TO 4
310 FOR CO=1 TO 8
320 PRINT CHR$(COLR(CO));
330 PRINT CHR$(18);CHR$(32);
340 NEXT CO
350 NEXT N
360 PRINT
370 NEXT ROW
380 GET A$:IF A$="" GOTO 380
390 RETURN

400 REM *** YOUR PROGRAM STARTS HERE ***

410 PRINT
420 GOSUB 180
```

SCREEN BLANKER

WHAT IT DOES

Allows blanking the Commodore 64 screen, while preserving data printed there.

Variables

None.

How To Use Subroutine

The normal "clear screen" routine destroys the data printed there. You may want to temporarily clear the screen to hide its contents from the operator. Or, your game may be writing some message to the screen, and you want to reveal it to the player all at one time.

This subroutine will temporarily change the screen and character color to match the border color, obscuring the writing on the screen. Call the first routine when you want screen blanking. Then, have your program call the second routine to restore the image to its original state.

Line By Line Description

Line 130: Blank screen.

Line 160: Restore screen.

Line 200: Access subroutine.

Lines 210 to 220: Delay before restoring screen.

Line 230: Access restore subroutine.

You Supply

No user changes required.

RESULT

Screen image blanked.

```
10 REM ****************
20 REM *               *
30 REM * BLANK SCREEN *
40 REM *               *
50 REM ****************
60 GOTO 190
70 REM ----------------------------
80 REM       + + VARIABLES + +
90 REM          NONE
100 REM
110 REM ----------------------------

120 REM *** BLANK SUBROUTINE ***

130 POKE 53265,PEEK(53265)AND 239
140 RETURN

150 REM *** RESTORE SUBROUTINE ***

160 POKE 53265,PEEK(53265)OR 16
170 RETURN

180 REM *** YOUR PROGRAM STARTS HERE ***
```

```
190 PRINT
200 GOSUB 130
210 FOR N=1 TO 1000
220 NEXT N
230 GOSUB 160
```

PROGRAM CHARACTERS

WHAT IT DOES

Redefine five characters to user-specified set.

Variables

None.

How To Use Subroutine

The Commodore 64 allows redefining its character set. The existing characters are constructed on an eight by eight-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 zero or a one, 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 64 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 64 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 64 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 64 not to look at the normal location for its character information but to start at 12288 decimal instead. Because this location is BASIC RAM, we have to protect it by lowering the top of RAM memory. That is accomplished by POKing 48 into 52 and 56 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, we will copy 64 characters from ROM into the protected RAM locations. This is 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 64 would be obtaining the information from a different place. Instead, we will POKE some new data into the locations for some selected characters that are not needed by our program. These characters are "at" sign (@), the exclamation point (!), and 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.

We POKE those new values, determined by laying out an 8 x 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 120 to 140: Protect memory, and redefine where computer collects its character set from.

Lines 160 to 200: Copy old characters, arrange for new set.

Lines 220 to 250: POKE new data for @ character.

Lines 270 to 300: POKE new data for ! character.

Lines 320 to 350: POKE new data for > character.

Lines 370 to 400: POKE new data for < character.

Lines 420 to 450: POKE new data for = character.

Lines 470 to 510: DATA lines with new character data.

Line 540: Print new characters.

You Supply

New DATA lines corresponding to your redesigned characters. Lay out your characters in an 8 x 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.

RESULT

Pressing "@", "!", "<", "=", and ">" keys, or using them in a program will produce new, redefined characters.

```
10 REM **************
20 REM *              *
30 REM * CHARACTERS *
40 REM *              *
50 REM **************
60 REM --------------------------
70 REM      + + VARIABLES + +
80 REM            NONE
90 REM
100 REM --------------------------

110 REM *** INITIALIZE ***

120 POKE 52,48:POKE 56,48:CLR
130 POKE 56334,PEEK(56334)AND254
140 POKE1,PEEK(1)AND251

150 REM *** COPY OLD CHARS ***

160 FOR I=0 TO 511
170 POKE I+12288,PEEK(I+53248)
180 NEXT I:POKE1,PEEK(1)OR4
190 POKE 56334,PEEK(56334)OR1
200 POKE 53272,(PEEK(53272)AND240)+12

210 REM *** CHANGE @ ***
```

```
220 FOR N=12288 TO 12288+7
230 READ H
240 POKE N,H
250 NEXT N

260 REM *** CHANGE ! ***

270 FOR N=12552 TO 12552+7
280 READ H
290 POKE N,H
300 NEXT N

310 REM *** CHANGE > ***

320 FOR N=12784 TO 12784+7
330 READ H
340 POKE N,H
350 NEXT N

360 REM *** CHANGE < ***

370 FOR N=12768 TO 12768+7
380 READ H
390 POKE N,H
400 NEXT N

410 REM *** CHANGE = ***

420 FOR N=12776 TO 12776+7
430 READ H
440 POKE N,H
450 NEXT N

460 REM *** SUBSTITUTE YOUR DATA ***

470 DATA 226,164,27,40,108,176,175,132
480 DATA 195,195,68,60,24,24,60,24
490 DATA 255,129,189,165,165,189,129,255
500 DATA 195,195,68,60,24,24,60,24
510 DATA 255,129,189,165,165,189,129,255

520 REM *** YOUR PROGRAM STARTS HERE ***

530 PRINT CHR$(147)
540 PRINT "!=><@"
```

REPEAT KEYS

WHAT IT DOES

Changes repeat keys and cursor color

Variables

- YES: Repeat all keys
- NO: Don't repeat
- SHADE: Color for cursor.

How To Use Subroutine

This subroutine allows turning "ON" or "OFF" the repeat key feature of your Commodore 64. Poking memory location AGAIN with variable YES will turn on the repeat feature. POKing AGAIN with NO will turn it off.

As a secondary function, the subroutine will also change the cursor color.

Line By Line Description

Lines 150 to 220: Define colors as vairables.

Lines 230 to 240: Define YES and NO variables.

Lines 250 to 260: Define addresses to POKE for repeat keys and cursor change.

Line 290: Define cursor color.

Line 300: Change cursor color.

Line 310: Change repeat key feature.

You Supply

- Variable YES or NO
- A value for SHADE, the cursor color.

RESULT

Either key repeat feature is turned ON, OFF, and cursor color adjusted.

```
10 REM ****************
20 REM *               *
30 REM * REPEAT KEYS, *
40 REM * CURSOR COLOR *
50 REM *               *
60 REM ****************
70 REM ----------------------------
80 REM      + + VARIABLES + +
90 REM      YES:   REPEAT KEYS
100 REM     NO:    DON'T REPEAT
110 REM     SHADE: NEW COLOR OF CURSOR
120 REM
130 REM ----------------------------

140 REM *** INITIALIZE ***

150 DARK=1
160 WHITE=2
170 RED=3
180 CYAN=4
190 PURPLE=5
200 GREEN=6
210 BLUE=7
220 YELLOW=8
230 YES=128
240 NO=0
250 AGAIN=650
260 CURSR=646
270 GOTO 340

280 REM *** SUBROUTINE ***

290 SHADE=RED
300 POKE CURSR,SHADE-1
310 POKE AGAIN,YES
320 RETURN

330 REM *** YOUR PROGRAM STARTS HERE ***

340 GOSUB 290
```

FUNCTION KEYS

WHAT IT DOES

Sends control to various subroutines, depending on which special function key has been pressed.

Variables

- A: Value of function key pressed.

How To Use Subroutine

The Commodore 64 function keys are nothing more than normal keyboard keys similar to those more commonly used. The difference is that no alphanumeric character is assigned to the function keys, like the letters A to Z, or numbers 0 to 9. Nor is a punctuation mark or other special character alloted to the function keys, like the "." or "," or Control key. The function keys actual do nothing but tell the computer that their particular key was pressed.

To oversimplify things a bit, when you press the letter A, the Commodore 64 knows that CHR$(65) (which corresponds to the letter A) has been pressed. Similarly, when the space bar is pressed, the computer recognizes CHR$(32.) The special function keys just return a CHR$ code of their own which, while not a letter of the alphabet, like CHR$(65), can be processed by the computer just as if a letter or number key were pressed. In the Commodore 64, the function F1, F3, F5, and F7 correspond to CHR$(133), CHR$(134) CHR$(135) and CHR$(136) respectively. The shifted function keys (F2, F4, F6, and F8) return CHR$(137) to CHR$(140).

To "program" them to perform some special function, it is necessary to determine which has been pressed, and then direct control of the program to an appropriate subroutine that actually carries out the function we want.

This FUNCTION KEY subroutine checks to see if one of the function keys has been depressed (CHR$(133) to CHR$(140)) and then sends the program to one of eight dummy subroutines. As written, the subroutine does NOTHING. You should substitute subroutines of your own for the dummies, in order to accomplish the task you want.

For example, one may be to clear the screen; another to change the real-time clock. You may use as many or as few as you wish. To filter out unwanted function keys, change line 170 to send control back to line 140 if an unneeded key is pressed:

```
170 ON A-132 GOSUB 210,230,250,270,290,310
175 GOTO 140
```

The above line will ignore function keys F6 and F8.

Line By Line Description

Line 140: Wait for user to press a key.

Line 150: Find out ASCII, or CHR$ code for that key.

Line 160: If key was not a function key, go back and wait some more.

Line 170: Depending on which key was pressed, access subroutine corresponding to that key.

Lines 210 to 360: Dummy subroutines to be replaced by user.

You Supply

- Subroutines as needed
- Modifications to filter out unwanted keys.

RESULT

Pressing function key leads to various subroutines.

```
10 REM ******************
20 REM *                 *
30 REM * FUNCTION KEYS  *
40 REM *                 *
50 REM ******************
60 GOTO 140
70 REM ----------------------------
80 REM       + + VARIABLES + +
90 REM     A:   VALUE OF FUNCTION
100 REM          KEY PRESSED
110 REM
120 REM ----------------------------

130 REM *** SUBROUTINE ***

140 GET A$:IF A$="" GOTO 140
150 A=ASC(A$)
160 IF A<133 OR A>140  GOTO 140
170 ON A-132 GOSUB 210,230,250,270,290,310,330,350
180 REM *** YOUR PROGRAM STARTS HERE ***
190 GOTO 140
200 REM *** DEFINE FUNCTIONS HERE ***
210 PRINT"F1 PRESSED"
220 RETURN
230 PRINT"F3 PRESSED"
240 RETURN
250 PRINT"F5 PRESSED"
260 RETURN
270 PRINT"F7 PRESSED"
280 RETURN
290 PRINT"F2 PRESSED"
300 RETURN
310 PRINT"F4 PRESSED"
320 RETURN
330 PRINT"F6 PRESSED"
340 RETURN
350 PRINT"F8 PRESSED"
360 RETURN
```

DISK COMMAND

WHAT IT DOES

Displays a menu that user can choose from to carry out formatting of new disk, scratching of unwanted programs from disk, initializing disk, or validating disk.

Variables

- NC: Number of menu choices.

How To Use Subroutine

This subroutine can be appended onto the end of any program, given line numbers that do not conflict. Then, by RUNning that portion, various disk commands can be carried out quickly.

The routines include formatting a new disk and killing unwanted programs from the disk. (This is also recommended over the SAVE"@0:filename",8 command, which can destroy files.) The menu selections also allow validating and initializing a disk.

You may add your own functions by writing additional routines, appending the choice to the menu, and increasing NC to reflect the new number of choices.

Line By Line Description

Lines 150 to 250: Menu of command choices.

Line 260: If number entered larger than NC, number of choices, go back and wait some more.

Line 270: Access desired subroutine.

Lines 320 to 430: Ask for disk name and ID, then format new disk.

Lines 440 to 550: Ask for name of program to be scratched, then erase from disk.

Lines 560 to 610: Initialize disk.

Lines 620 to 670: Validate disk.

You Supply

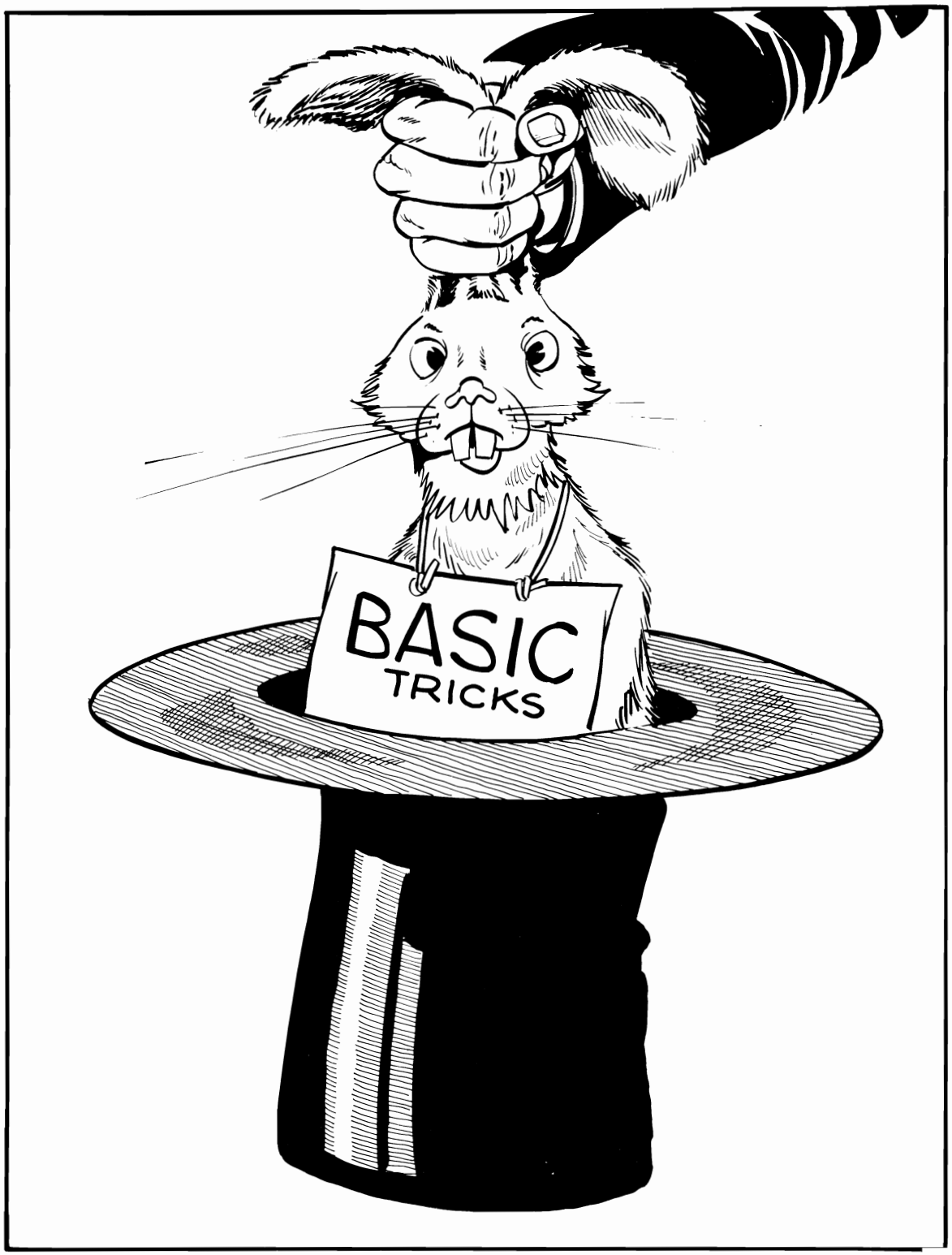
- A value for NC whenever the menu is enlarged.

RESULT

Disk functions carried out automatically.

```
10 REM ****************
20 REM *               *
30 REM * DISK COMMAND  *
40 REM *               *
50 REM ****************

60 REM *** INITIALIZE ***

70 NC=4
80 GOTO 300
90 REM ----------------------------
100 REM       + + VARIABLES + +
110 REM       NC: NUMBER OF MENU CHOICES
120 REM
130 REM ----------------------------

140 REM *** SUBROUTINE ***

150 PRINT CHR$(147)
160 PRINT TAB(6)"** MENU **"
170 PRINT CHR$(17);CHR$(17)
180 PRINT TAB(3)"1. FORMAT NEW DISK"
190 PRINT TAB(3)"2. SCRATCH PROGRAM"
200 PRINT TAB(3)"3. INITIALIZE"
210 PRINT TAB(3)"4. VALIDATE"
220 PRINT CHR$(17)
230 PRINT TAB(6)"ENTER CHOICE"
240 GET A$:IF A$="" GOTO 240
250 A=VAL(A$)
260 IF A<1 OR A>NC GOTO 240
270 ON A GOSUB 330,450,570,630
280 RETURN

290 REM *** YOUR PROGRAM STARTS HERE ***

300 PRINT
310 GOSUB 150

315 REM *** COMMANDS ***
```

```
320 REM -FORMAT-
330 PRINT CHR$(147)
340 PRINT"INSERT DISK TO"
350 PRINT"BE FORMATTED."
360 PRINT
370 INPUT"DISK NAME:";N$
380 INPUT"ID? (2 CHAR):";ID$
390 ID$=LEFT$(ID$,2)
400 OPEN15,8,15
410 PRINT#15,"NO:N$,ID$"
420 CLOSE 15
430 GOTO 310
440 REM -SCRATCH-
450 PRINT CHR$(147)
460 OPEN15,8,15
470 PRINT "NAME OF PROGRAM"
480 PRINT "TO BE SCRATCHED :"
490 PRINT
500 PRINT"<RETURN> TO QUIT."
510 INPUT PROGRAM$
520 IF PROGRAM$="" THEN CLOSE 15:GOTO 310
530 PRINT#15,"SO:PROGRAM$"
540 PROGRAM$=""
550 GOTO 470
560 REM -INITIALIZE DISK-
570 PRINT CHR$(147)
580 OPEN15,8,15
590 PRINT#15,"I"
600 CLOSE 15
610 GOTO 310
620 REM -VALIDATE-
630 PRINT CHR$(147)
640 OPEN15,8,15
650 PRINT#15,"V"
660 CLOSE15
670 GOTO 310
```


BASIC
TRICKS

6 Basic Tricks

Here are some BASIC routines that will make your programming a bit easier. These are general subroutines that can be applied to many different programs. Two 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 numbers only, or alpha characters only, are desired, that is what the routines will accept. All other characters are ignored. By using these, 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.

Three sort routines are included for those who need to rearrange lists of numbers or strings. Two bubble sorts are included, as these are the easiest to understand and to modify.

Loading arrays with data is one of the most frequent requirements for any BASIC program. Beginners are often confused by arrays. Yet, this is one of the most important concepts after FOR-NEXT loops, and program branching (GOTO, GOSUB). The array-loading subroutine is included here primarily for educational value. If you don't understand how to fill an array, you probably couldn't use it properly. The example presented is a fully-working program that the user can RUN and experiment with until arrays are more fully understood.

The array routine can be transplanted to other programs, however, and interfaced with disk or tape read/write routines, provided later in this book, to build a complete data base program with permanent files.

"Automatic Saver" is a program development utility that helps protect the user from data loss due to power outages. It makes periodic saves more convenient, especially when using disk drives, which do not lend themselves to using the same file name over and over easily.

These BASIC tricks are all simple enough to find their way into many of your programs.

NUMBER INPUT

WHAT IT DOES

Allows user to input only numbers

Variables

- I: Number entered
- I$: String entered.

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" will 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 non-numbers; 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 GET A$ 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.

Line By Line Description

Line 140: Wait for user entry.

Line 150: If key pressed was RETURN, then input is finished.

Line 160: If key was less than 0 or greater than 9, go back and wait for another entry.

Line 170: Print acceptable key pressed to screen.

Line 180: Add key to previous entries.

Line 190: Go back for more entries.

Line 200: Variable I equals value of entries.

Line 230: Access the subroutine.

You Supply

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.

RESULT

Only user numeric input, in the form of positive numbers, is allowed.

```
10 REM ****************
20 REM *               *
30 REM * NUMBER INPUT  *
40 REM *               *
50 REM ****************
60 REM ----------------------
70 REM    + + VARIABLES + +
80 REM    I:  NUMBER ENTERED
90 REM    I$: STRING ENTERED
100 REM
110 REM ----------------------
120 GOTO 230
```

```
130 REM *** SUBROUTINE ***

140 GET A$:IF A$="" GOTO 140
150 IF A$=CHR$(13) GOTO 200
160 IF A$<"0" OR A$>"9" GOTO 140
170 PRINT A$;
180 I$=I$+A$
190 GOTO 140
200 I=VAL(I$):PRINT
210 RETURN

220 REM *** YOUR PROGRAM STARTS HERE ***

230 GOSUB 140
```

LETTER INPUT

WHAT IT DOES

Allows user to input only alpha characters.

Variables

- I$: String entered.

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 GET A$ 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

Line 130: Wait for user entry.

Line 140: If key pressed was RETURN, then input is finished.

Line 150: If key was less than A or greater than Z, go back and wait for another entry.

Line 160: Print acceptable key pressed to screen.

Line 170: Add key to previous entries.

Line 180: Go back for more entries.

Line 210: Access the subroutine.

You Supply

User may change the upper and lower limits in line 150 to restrict the range of alpha characters that can be entered. This might be useful when getting input for, say, a game like Mastermind(TM) where only the letters A-E are wanted. All numbers, graphics, and alpha characters larger than E can be ignored.

RESULT

Only user alpha input is allowed.

```
10 REM ****************
20 REM *               *
30 REM * LETTER INPUT *
40 REM *               *
50 REM ****************
60 REM -----------------------
70 REM    + + VARIABLES + +
80 REM    I$: STRING ENTERED
90 REM
100 REM -----------------------
110 GOTO 210

120 REM *** SUBROUTINE ***

130 GET A$:IF A$="" GOTO 130
140 IF A$=CHR$(13) GOTO 190
150 IF A$<"A" OR A$>"Z" GOTO 130
160 PRINT A$;
170 I$=I$+A$
180 GOTO 130
190 RETURN

200 REM *** YOUR PROGRAM STARTS HERE ***

210 GOSUB 130
```

STRING SORT

WHAT IT DOES

Alphabetizes a list.

Variables

- NU: Number of items to be sorted
- US$(n): Array storing list to be sorted.

How To Use Subroutine

Sorting a list is a common need for many programs. Data files, mailing lists, and other groups 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, US$(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 or tape, for example, using one of the routines presented later 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. When 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 the same number of 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 comparision between strings, the Commodore 64 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. Gradually, each member of the list "floats" up to its proper place in the array.

While such sorts are not very fast, small lists of, say, 30 or 40 items, the speed is satisfactory.

Line By Line Description

Line 130: Define NU, the number of units in the array to be sorted.

Line 140: DIMension the array to proper size.

Lines 170 to 200: 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 210: Start loop from 1 to the number of items to be sorted.

Line 220: Start a nested loop from 1 to 1 less than the number of items to be sorted.

Line 230: Make A$ equal to the N1th item of the array.

Line 240: Make B$ equal to the item following A$ in the array.

Line 250: 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 260 to 270: 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 280 to 290: The inner and outer loops are incremented.

Lines 300 to 320: The sorted list is printed to the screen.

You Supply

- Define NU, the number of items to be sorted
- Supply the data for the array, US$(n).

RESULT

List is sorted alphabetically.

```
10 REM ***************
20 REM *             *
30 REM * STRING SORT *
40 REM *             *
50 REM ***************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM     NU:     NUMBER OF ITEMS SORTED
90 REM     US$(N): ARRAY WITH ITEMS
100 REM
110 REM ----------------------------

120 REM *** INITIALIZE ***

130 NU=10
140 DIM US$(NU)
150 GOTO 350

160 REM *** SUBROUTINE ***
```

```
170 FOR ITEM=1 TO NU
180 PRINT"ENTER # ";ITEM
190 INPUT US$(ITEM)
200 NEXT ITEM
210 FOR N=1 TO NU
220 FOR N1=1 TO NU-N
230 A$=US$(N1)
240 B$=US$(N1+1)
250 IF A$<B$ THEN GOTO 280
260 US$(N1)=B$
270 US$(N1+1)=A$
280 NEXT N1
290 NEXT N
300 FOR N=1 TO NU
310 PRINT US$(N)
320 NEXT N
330 RETURN

340 REM *** YOUR PROGRAM STARTS HERE ***

350 GOSUB 170
```

NUMBER SORT

WHAT IT DOES

Sort group of numbers by size.

Variables

- NU: Number of items to be sorted
- US(n): Array storing list to be sorted.

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 one of the routines presented later 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.

Line By Line Description

Line 130: Define NU, the number of units in the array to be sorted.

Line 140: DIMension the array to proper size.

Lines 170 to 200: 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 210: Start loop from 1 to the number of items to be sorted.

Line 220: Start a nested loop from 1 to 1 less than the number of items to be sorted.

Line 230: Make A equal to the N1th item of the array.

Line 240: Make B equal to the item following A in the array.

Line 250: 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 260 to 270: 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 280 to 290: The inner and outer loops are incremented.

Lines 200 to 320: The sorted list is printed to the screen.

You Supply

- Define NU, the number of items to be sorted
- Supply the data for the array, US(n).

RESULT

List of numbers is sorted by size.

```
10 REM ***************
20 REM *               *
30 REM * NUMBER SORT *
40 REM *               *
50 REM ***************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM     NU:    NUMBER OF ITEMS SORTED
90 REM     US(N): ARRAY WITH ITEMS
100 REM
110 REM ----------------------------
```

```
120 REM *** INITIALIZE ***

130 NU=10
140 DIM US(NU)
150 GOTO 350

160 REM *** SUBROUTINE ***

170 FOR ITEM=1 TO NU
180 PRINT"ENTER # ";ITEM
190 INPUT US(ITEM)
200 NEXT ITEM
210 FOR N=1 TO NU
220 FOR N1=1 TO NU-N
230 A=US(N1)
240 B=US(N1+1)
250 IF A<B THEN GOTO 280
260 US(N1)=B
270 US(N1+1)=A
280 NEXT N1
290 NEXT N
300 FOR N=1 TO NU
310 PRINT US(N)
320 NEXT N
330 RETURN

340 REM *** YOUR PROGRAM STARTS HERE ***

350 GOSUB 170
```

ARRAY LOADER

WHAT IT DOES

Load array with data.

Variables

- NROWS: Number of rows in array
- NCOLUMNS: Number of columns in array.

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, date, 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 manipulated, sorted, added to, or entries deleted. This subroutine does exactly that. Although written for a string array, it can be converted to a numeric array simply by 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 140 to 150: Define number of rows and columns in the data file. User should change these numbers to reflect their own data.

Line 160: Dummy data, in this example a name, address, and phone number.

Line 170: DIMension array to size specified by values of NR and NC.

Lines 200 to 210: Begin nested loops which repeat for the number of rows, and the number of columns.

Line 220: READ item of DATA.

Line 230: Place data in current array element, defined by ROW and COLUMN in FOR-NEXT loop. Each time through the inner loop, column will be incremented by one, while ROW remains the same. When finished, ROW is incremented by one, and the inner loop repeats. As a result, DTA$(1,1) is loaded first, followed by DTA$(1,2) and DTA$(1,3). Then, DATA$(2,1), and so forth, are filled from the DATA.

Lines 240 to 250: Increment the FOR-NEXT loops.

Line 290: Access the subroutine.

Lines 300 to 350: Print out the data loaded into the array.

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.

RESULT

Data list is loaded into array.

```
10 REM ****************
20 REM *               *
30 REM * ARRAY LOADER *
40 REM *               *
50 REM ****************
60 GOTO 280
70 REM ----------------------------
80 REM       + + VARIABLES + +
90 REM      NROWS:     NUMBER OF ROWS
100 REM     NCOLUMNS: NUMBER OF COLUMNS
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 NROWS=2
150 NCOLUMNS=3
160 DATA JOE,2 PINE,232-4531,SAM,1 ROE,445-3622
170 DIM DTA$(NR,NC)
180 GOTO 280

190 REM *** SUBROUTINE ***

200 FOR ROW=1 TO NROWS
210 FOR COLUMN=1 TO NCOLUMNS
220 READ A$
230 DTA$(ROW,COLUMN)=A$
240 NEXT COLUMN
250 NEXT ROW
260 RETURN

270 REM *** YOUR PROGRAM STARTS HERE ***

280 PRINT
290 GOSUB 200
300 PRINT''NAME ADDRESS PHONE''
310 PRINT
320 FOR ROW=1 TO NROWS
330 FOR COL=1 TO NCOL
340 PRINT DTA$(ROW,COL);" ";
350 NEXT COL
360 PRINT
370 NEXT ROW
```

SUPER SAVER

WHAT IT DOES

SAVES program to disk.

Variables

- F$: Name of your program.

How To Use Subroutine

Frequent SAVING of a program under construction is good insurance against losing hours of work to a power failure or accidental keyboard lockup. The fast operation of a disk drive makes this step easy.

However, it is sometimes advisable to save a program under a different name each time so that, if necessary, one can go back to an earlier version when a later incarnation has gone awry. In addition, saving a program by writing over the earlier version, using the SAVE"@:filename",8 syntax can be dangerous. Under some conditions, a file can be destroyed. Rather than SCRATCH the old file before resaving, use this subroutine to create a new filename automatically.

The subroutine takes the name of the program you specify, F$, in line 130, and adds to it the current time on the real-time clock, TI$. So, each SAVE during a session will have filenames like "TEST120330", "TEST132015", and so forth. After all work is finished, simply SCRATCH the unwanted versions, and save the final copy.

Renumber the subroutine with high line numbers, say, 30000 and over. Then, each time you wish to SAVE the file, type RUN 30000. The rest is automatic.

Line By Line Description

Line 130: F$ is defined as "TEST" plus the current TI$. User should substitute their own program name for "TEST".

Line 140: If total filename is more than 16 characters, the maximum allowed by the Commodore 64, the rightmost 16 characters are taken and used instead.

Line 150: The file is saved under the filename F$.

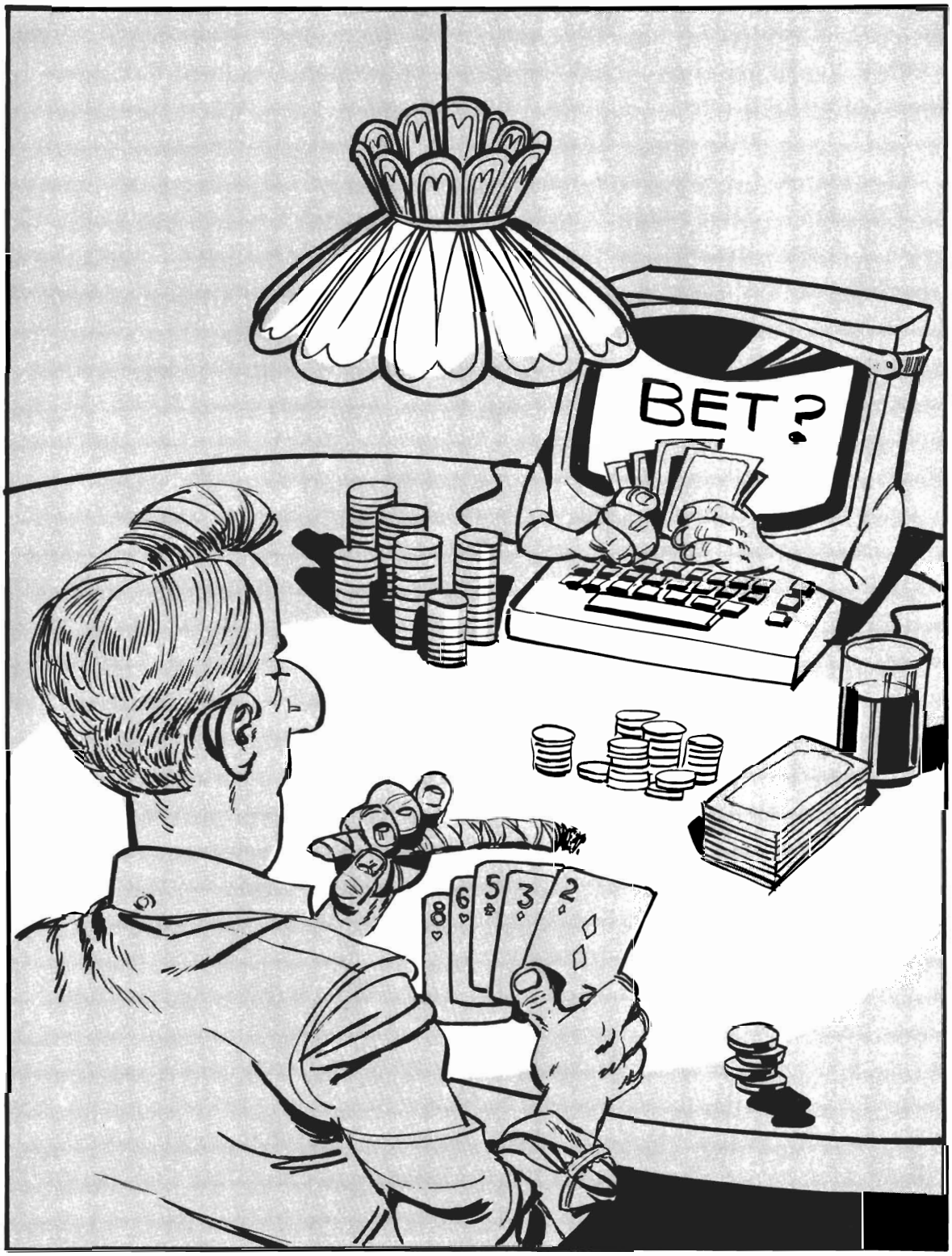
You Supply

Define a filename, which should be 10 characters or less, to allow for the 6 characters of TI$. If your filename is longer than 10 characters, no error will be generated. Instead, it will be shortened. For example, "TEST PROGRAM" might be SAVEd as "ST PROGRAM123412."

WHAT IT DOES

File can be saved repeatedly under updated names.

```
10 REM ***************
20 REM *              *
30 REM * SUPER SAVER  *
40 REM *              *
50 REM ***************
60 GOTO 170
70 REM ----------------------------
80 REM       + + VARIABLES + +
90 REM     F$:    NAME OF YOUR PROGRAM
100 REM
110 REM ----------------------------

120 REM *** SUBROUTINE ***

130 F$="TEST"+TI$
140 IF LEN(F$)>16 THEN F$=RIGHT$(F$,16)
150 SAVE F$,8:END

160 REM *** YOUR PROGRAM STARTS HERE ***

170 PRINT
```


BET?

7 Game Routines

Games are probably one of 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 program. Somewhere short of your first year with the machine, you will probably investigate writing an arcade-quality joystick-activated shoot-em-up.

In either case, you can use 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 so far can be transplanted to games programs. These include all of the routines in Chapters 2, 3, and 4. Many of those in Chapter 5, such as setting screen and border colors, using function keys, or determining screen color at specific locations are handy. User-defined character sets is a popular mainstay of games programming.

Here are five more subroutines that are particularly applicable to games. Three deal with universal "tools" of games: decks of cards, rolling pairs of dice, or flipping coins. The dice module is especially flexible, as it allows you to define how many sides each die will have. Dungeons and Dragons players take note!

Randomness is an essential factor in many types of games, and one subroutine in this section allows you to specify the drawing of random numbers in any range you choose. If your game happens to need random numbers larger than 100 and smaller than 999, the routine can handle that nicely.

Arcade-style programs frequently need a slowdown feature. Delay loops that change in length, getting faster or slower, are a neat way to accomplish this. A subroutine is provided that does all the work for you.

DEAL CARDS

WHAT IT DOES

Shuffle and deal deck of cards.

Variables

- DECK$(n): Deck of cards
- CARD$: Card drawn from deck
- DRAW: Random number.

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 the subroutine beginning at line 370. The deck has already been assembled (lines 210-360) using the Commodore 64 graphics characters for suits, and the numbers or 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.

The routine starts off by setting NC (number of cards) to 52. In line 410, the computer selects a number between 1 and NC (52 this time), and that element of DECK$(n) becomes the card drawn. This leaves a "hole" in the deck at position DRAW. We fill it up by taking the last card in the deck, which is DECK$(NC), and place it in DECK$(DRAW). This leaves the "hole" at the end, but we then change NC to equal NC-1, so the computer will only draw 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 we have mixed up the order of the deck, as we want the cards shuffled in the first place.

Each element of DECK$(n) consists of a number, or face card name, plus the Commodore 64 CHR$ value for the suit (either 193,211,218 or 216—DATA in line 140). This produces a full deck of 52 cards.

Line By Line Description

Line 130: DIMension an array to represent the deck of cards.

Lines 160 to 190: READ CHR$ codes corresponding to the graphics characters for individual suits into array.

Line 210: Begin FOR-NEXT loop from 1 to 4, one trip through for each individual suit.

Line 220: Increment CU, which keeps track of which element of DECK$(n) is being created. CU will range in value from 1 to 52.

Lines 230 to 290: Create the face cards for each suit.

Lines 300 to 330: Create numbered cards for each suit, through a FOR-NEXT loop from 2 to 10 (deuce to ten). A string representation of each number is added to " OF " and the suit symbol, SUIT$(SUIT).

Line 340: Repeat loop.

Line 350: Define number of cards, NC, as initially equalling 52.

Line 360: GOTO main program.

Line 370: If any cards are remaining, access the "draw" routine.

Lines 380 to 390: If none remaining, tell player that the deck has been dealt.

Line 410: Draw a random card number smaller than NC, the number of cards remaining.

Line 420: Make card drawn, CARD$, equal the DRAW element of the array DECK$(n).

Line 430: Place the last card in the array in the hole left behind by the drawn card.

Line 440: Reduce the size of the deck by one card.

Line 480: Access the subroutine.

You Supply

No user input needed.

RESULT

Deck of 52 cards may be dealt out as needed.

```
10 REM **************
20 REM *              *
30 REM * DEAL CARDS *
40 REM *              *
50 REM **************
60 REM -----------------------
70 REM     + + VARIABLES + +
80 REM    DECK$(N): DECK
90 REM    CARD$:    CARD DRAWN
100 REM   DRAW:     RANDOM CARD
110 REM
120 REM -----------------------
130 DIM DECK$(52)
140 DATA 193,211,218,216

150 REM *** READ SUITS ***

160 FOR N=1 TO 4
170 READ A
180 SUIT$(N)=CHR$(A)
190 NEXT N

200 REM *** ASSEMBLE DECK ***

210 FOR SUIT=1 TO 4
220 CU=CU+1
230 DECK$(CU)="ACE OF "+SUIT$(SUIT)
240 CU=CU+1
250 DECK$(CU)="KING OF "+SUIT$(SUIT)
260 CU=CU+1
270 DECK$(CU)="QUEEN OF "+SUIT$(SUIT)
280 CU=CU+1
290 DECK$(CU)="JACK OF "+SUIT$(SUIT)
300 FOR N=2 TO 10
310 CU=CU+1
320 DECK$(CU)=STR$(N)+" OF "+SUIT$(SUIT)
330 NEXT N
340 NEXT SUIT
350 NC=52
360 GOTO 470
370 IF NC<>0 GOTO 410
380 CARD$=""
390 PRINT"DECK GONE!!"
400 RETURN
410 DRAW=INT(RND(1)*NC)+1
420 CARD$=DECK$(DRAW)
430 DECK$(DRAW)=DECK$(NC)
440 NC=NC-1
450 RETURN
```

```
460 REM *** YOUR PROGRAM STARTS HERE ***

470 PRINT
480 GOSUB 370
490 PRINT CARD$
```

RANDOM RANGE

WHAT IT DOES

Allows choosing random numbers in any range.

Variables

- HIGH: Top of range
- MINIMUM: Bottom of range
- DF: Difference
- NU: The number chosen.

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 unadorned Commodore 64 BASIC.

The Commodore 64 can choose pseudorandom numbers. That is, although they appear to be random, the numbers actually are 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 one to seven, for example. The Commodore 64 will generate random numbers larger than zero and smaller than one. So, we might get .74329 or .15832 or some other value. To get numbers in a given range 1 to N, we must multiply the random number by N and add one. That is, INT(RND(1)*7)+1 will produce numbers larger than one and no larger than seven.

But, what if some other range is desired—such as numbers between 43 and 198? This subroutine will pluck them out of randomland for you. From user-supplied minimum and maximum numbers, it will select random integers only in the desired range.

Line By Line Description

Lines 160 to 180: Define the highest random number desired, the lowest, and find the difference between them.

Line 200: Choose a random number in the range 1 to DF, the difference, then add the minimum number to that to produce a number in the desired range.

Line 230: Print the result.

Line 240: Access the subroutine.

You Supply

- Define HIGH and MINIMUM to set the limits for the random range you want.

RESULT

Only random numbers in the specified range will be produced.

```
10 REM ****************
20 REM *               *
30 REM * RANDOM RANGE *
40 REM *               *
50 REM ****************
60 REM ------------------------
70 REM      + + VARIABLES + +
80 REM      HIGH:     TOP OF RANGE
90 REM      MINIMUM:  BOTTOM OF RANGE
100 REM     DF:       DIFFERENCE
110 REM     NU:       NUMBER CHOSEN
120 REM
130 REM ------------------------
140 GOTO 230

150 REM *** INITIALIZE ***

160 HIGH=100
170 MINIMUM=15
180 DF=HIGH-MINIMUM+1

190 REM *** SUBROUTINE ***

200 NU=INT(RND(1)*DF)+MINIMUM
210 RETURN

220 REM *** YOUR PROGRAM BEGINS HERE ***

230 PRINT NU;
240 GOSUB 190
```

COIN FLIP

WHAT IT DOES

Flips coin, producing heads or tails.

Variables

- FLIP: Random value, either one or two
- FLIP$: Name of side chosen
- COIN$(n): Coin array.

How To Use Subroutine

Some beginner level statistical experiments and a few games need to simulate coin flips. For example, you may want to construct a loop that flips a coin 1000 times and adds up the number of heads and tails to check the randomness of your computer.

This subroutine will flip the coin for you, producing the name of the side—either "HEADS" or "TAILS"—after each flip. The module can be adapted to larger ranges of choice, with more than two names to be applied. For example, the array names might be NORTH, SOUTH, EAST, and WEST, and the 2 in line 180 changed to a 4. Then, random directions will be chosen.

Line By Line Description

Lines 140 to 150: Define array as "HEADS" and "TAILS".

Line 180: Produce value for variable FLIP of either 1 or 2.

Line 190: Assign "HEADS" or "TAILS" to FLIP$, depending on which random number was chosen.

Line 220: Access the subroutine.

Line 230: Print result.

You Supply

- No user input needed.

RESULT

Coin flipping simulated.

```
10 REM *************
20 REM *           *
30 REM * COIN FLIP *
40 REM *           *
50 REM *************
60 REM ------------------------
70 REM     + + VARIABLES + +
80 REM     COIN$(N): COIN ARRAY
90 REM     FLIP:     RANDOM VALUE 1-2
100 REM    FLIP$:    SIDE FLIPPED
110 REM
120 REM ------------------------
```

```
130 REM *** INITIALIZE ***

140 COIN$(1) = "HEADS"
150 COIN$(2) = "TAILS"
160 GOTO 220

170 REM *** SUBROUTINE ***

180 FLIP = INT(RND(1)*2) + 1
190 FLIP$ = COIN$(FLIP)
200 RETURN

210 REM *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 180
230 PRINT FLIP$
240 GOTO 220
```

DICE

WHAT IT DOES

Simulates roll of dice.

Variables

- D1: Value of Die #1
- D2: Value of Die #2
- ROLL: Total of roll.

How To Use Subroutine

This dice rolling subroutine includes a short sound module to provide an additional bit of realism. It will roll two dice, each producing a number between one and six. The value of each die, as well as the total roll, is figured.

Dungeons and Dragons players can specify how many sides each die in the pair will have. In adapting this subroutine for that feature, you might want to add a line like INPUT"ENTER NUMBER OF SIDES";SIDE before each roll. If only one die is needed, both will be rolled anyway. Just choose which one will "count" ahead of time, either D1 or D2. Variable ROLL will store the total count.

Line By Line Description

Lines 150 to 160: Roll two dice, each producing numbers in the range 1 to SIDES, with SIDES defined as the number of sides you wish on the dice.

Line 170: Make ROLL the total of the two dice.

Line 190: Beginning of sound routine. Set volume to maximum.

Line 200: Poke a note to sound register.

Line 210: Delay a while.

Line 220: Poke a note to different sound register.

Line 230: Another short delay.

Line 240: Turn both sound registers off.

Line 250: Return to main program.

Line 280: Define number of sides on dice.

Line 290: Access the subroutine.

Lines 300 to 330: Print results of roll.

You Supply

- Number of sides of die.

RESULT

N-sided dice are rolled, and the value of each plus total roll reported.

```
10 REM ********
20 REM *      *
30 REM * DICE *
40 REM *      *
50 REM ********
60 GOTO 270
70 REM ----------------------------
80 REM        + + VARIABLES + +
90 REM     D1:   DIE #1 TOTAL
100 REM    D2:   DIE #2 TOTAL
110 REM    ROLL: TOTAL OF ROLL
120 REM
130 REM ----------------------------

140 REM *** SUBROUTINE ***
```

```
150 D1=INT(RND(1)*SIDES)+1
160 D2=INT(RND(1)*SIDES)+1
170 ROLL=D1+D2
180 REM *** DICE SOUND ***
190 POKE 36878,15
200 POKE 36874,150
210 FOR N=1 TO 100:NEXT N
220 POKE 36877,255
230 FOR N=1 TO 100:NEXT N
240 POKE 36877,0:POKE 36874,0
250 RETURN

260 REM *** YOUR PROGRAM STARTS HERE ***

270 PRINT
280 SIDES=6
290 GOSUB 150
300 PRINT" DIE #1:";D1
310 PRINT" DIE #2:";D2
320 PRINT "TOTAL :";
330 PRINT ROLL
```

DELAY LOOP

WHAT IT DOES

Delays loop changes in length.

Variables

- DELAY: Initial delay
- CHANGE: Amount of change.

How To Use Subroutine

In games, delay loops are frequently used to display messages on the screen for a given length of time. However, another important use is to control the speed of movement or some other play action. By having a FOR-NEXT loop count off between each move, a short delay can be built in. A loop from 1 to 100 might slow things down appreciably, while 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 140: Set initial delay to 1000.

Line 150: Set change factor to .9.

Line 180: Count off the delay.

Line 190: Change value of delay.

Line 230: Access the subroutine.

Line 240: Inform player that delay is finished.

Line 250: Repeat, with shortened delay.

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 one. Setting it to 1.1 will slowly increase the delay each time. Any number larger than 1.5 (such as two or three) will probably slow down the program much more than you desire.

RESULT

Program speeds up, or slows down gradually, at a rate selected by user.

```
10 REM *********
20 REM *       *
30 REM * DELAY *
40 REM *       *
50 REM *********
60 REM ----------------------------
70 REM + + VARIABLES + +
80 REM    DELAY:  INITIAL DELAY
90 REM   CHANGE:  AMOUNT OF CHANGE
100 REM           PLUS OR MINUS
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***
```

```
140 DELAY=1000
150 CHANGE=.90
160 GOTO 230

170 REM *** SUBROUTINE ***

180 FOR N=1 TO DELAY
190 NEXT N
200 DELAY=DELAY*CHANGE
210 RETURN

220 REM *** YOUR PROGRAM STARTS HERE ***

230 GOSUB 180
240 PRINT"FINISHED"
250 GOTO 230
```

8 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. These files can be loaded by the Commodore 64 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.

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), we would have to type the information in everytime we 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.

There are three basic types of data files available to Commodore 64 users: sequential, random access, and relative files. Disk drive users may take advantage of all three types. Those who are confined to cassette data storage may use only sequential files. Because this type is easiest to understand and use, sequential file systems are emphasized in this book.

The Commodore 64 cassette recorder is a good analog to sequential files, i.e., serial 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 is 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 only written or read 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. If your program needs some information for the middle of a data file, it must read in the entire file, make any changes it wants, and then write the entire file back to the tape or disk.

To read a given data file, we first OPEN a channel for that information to be sent. Then, we INPUT# (with the # being followed by the number we have assigned to the input channel, e.g. INPUT#1) data to a variable of our 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 64 keep its files and the devices it uses straight, each of the devices has been assigned a device number. The keyboard is device number zero; the cassette tape recorder device number one; the screen device number three; and so forth. A serial printer is assigned device number four, while the first disk drive in a system is usually assigned device number eight.

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

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

If no device number is indicated, the computer assumes we mean device number one, 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, we would type:

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

Using a numeral four, instead, would send the file to the printer. Logically, we could even list a program to the screen by SAVE"filename",3, except that the Commodore 64 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 will redirect 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 we have OPENed that device first.

As 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.

The particular channel we 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 only be used 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 we are using. So, to open a cassette data file, we 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.

We 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 filename within quotes. However, we will follow the convention of using the same logical file number as the device number.

What about that secondary address number? What other options are available? For tape usage, there are two more. We may specify, "0", which signifies reading a file from the tape, or "2", which will open the channel for writing to the tape, but place a special "end-of-tape" marker at the end of the file. In reading that file, the Commodore 64 will progress no further, until the EOT marker is removed.

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

You may have guessed that SAVE or 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. So, 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, in the device number specification. Disk drive users can also follow the filename with a file type specifier, generally "S" (for "sequential") or "P" (for "program"). You also need to tell the disk drive which direction the information will flow, using "W" for write and "R" for read.

So, 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".
```

You can examine the sequential file tape and disk read/write routines in order to understand this method of data storage better.

The Commodore 64 also can use random access files, when updates to a file are frequent and relative files which are not generally used in BASIC 2.0. Both are types of files that allow accessing any given record within a file, either for 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. Instead, we offer, as the first module in this section, a faster data file routine that can be used by disk or tape owners alike and requires no lengthy accesses to mass storage media. Instead, the entire data file is stored within the program itself.

DATA FILES

WHAT IT DOES

Imbeds data files in program listing.

Variables

- R: Number of items in file
- DTA$(n): Array storing data file
- FIELD$(n): Field names.

How To Use Subroutine

Sometimes we like to keep track of information that is entered into a program permanently. Storing data on cassette or disk with the Commodore 64 is relatively simple, but each method has drawbacks.

To use disk files, you must have a disk drive, which are more expensive than cassette recorders like the Commodore Datasette. Commodore 64 disk files are far from a universal tool, available to everyone. But, because of the header written on tape data files, writing to cassette and loading data from tape is relatively slow. In addition, how do

you know your data was recorded successfully? The Commodore 64 uses special circuitry that makes its cassette recorder much more reliable than those used on some other computer systems. Because of this, it can use much less expensive tapes than other computers. These tapes do have dropouts—places with no iron oxide available to record information. That is why it is wise to VERIFY any program that is SAVEd.

Because it is so simple to check a program, why not simply store data within a program itself? At the end of the RUN, SAVE the new program, VERIFY it, and you can be assured that your information is safely transferred to tape.

This subroutine allows you to do that, storing data in the program in the form of DATA lines. It shows you the lines already entered so that the new one may be placed at the end. It also prompts you to make one other change, which tells the program how many items of data have been entered.

Then, in running the program, you can search for an item of data. The Commodore 64 will loop through all the items in the file, display each field, or tell you that the item is not in the file.

The program first reads the DATA into a two dimensional string array, DTA$(row,column). Two nested FOR-NEXT loops read all these data into the proper places in the array DTA$(row, column). In line 1060, the first loop starts from 1 to R, which is the number of rows and which is defined in line 201. Then, the second loop starts, from 1 to 3, the number of columns. Each time through the second loop, one of the data items is stored in DTA$(ROW,COLUMN).

When all three have been read for a particular record, the program advances to line 1100. This line sends control back to line 1060 for the next ROW. After all the data has been read, a menu is presented which offers the choice of adding a listing or retrieving a record. If retrieval is chosen, the user is asked to enter a string to search for. This string is stored in variable NME$, and a FOR-NEXT loop from 1 to R looks at column #1 of each row until a match is found.

If so, the data are displayed on separate lines. If no match is found, the computer tells us the bad news. In either case, the program returns to the main menu. If we want to add a name, instructions appear telling the user to add the DATA line to the last one shown and to change line 201 so that R is defined as one larger. The new value of R (R + 1) is displayed to make this very simple. The program then lists all the lines from 200 to 999 and turns the computer over to the user in command mode.

The program line number chosen should be one larger than the last used (i.e., on the initial RUN, 206 should be used). This will make it possible to fit the maximum number of DATA lines to be entered. Actually, the DATA can be inserted anywhere in the program, but it is much neater if they are located all in one consecutive block.

Line By Line Description

Line 200: DATA for names of each of the three fields used by this program. User should change these to names of actual fields in his or her data file.

Line 205: Initial, dummy data, showing the format for the first items of data in the file. Additional entries should follow this format.

Lines 1020 to 1040: Read names of fields into array.

Line 1050: Dimension array to R + 5 rows, and three columns, allowing for five new items to be entered in each session.

Lines 1060 to 1100: Read existing data lines into the array.

Lines 1110 to 1180: Print menu, get user choice of add listing or retrieve information.

Lines 1190 to 1260: Instructions for updating data file.

Line 1270: List current data to screen.

Lines 1280 to 1310: User enters name to search for.

Lines 1320 to 1340: Program searches for item desired.

Lines 1350 to 1380: Item was not found.

Lines 1390 to 1450: Item was found and is listed to the screen.

You Supply

Data and program line changes as prompted by the routine are needed. The first three DATA items should be the names of your fields, i.e., NAME, ADDRESS, PHONE.

RESULT

Data file is imbedded in the program and can be SAVEd reliably.

```
10 REM ***************
20 REM *             *
30 REM * DATA FILES  *
40 REM *             *
50 REM ***************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM     R:          ITEMS OF DATA
90 REM     DTA$(N):    DATA FILE
100 REM    FIELD$(N): FIELDS
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***
```

```
200 DATA NAME,ADDRESS,PHONE
201 R=1
205 DATA JOE,1 PINE ST.,232-1347
1020 FOR N=1 TO 3
1030 READ FIELD$(N)
1040 NEXT N
1050 DIM DTA$(R+5,3)
1060 FOR ROW=1 TO R
1070 FOR COLUMN=1 TO 3
1080 READ DTA$(ROW,COLUMN)
1090 NEXT COLUMN
1100 NEXT ROW
1110 PRINT CHR$(147)
1120 PRINT "DATA BASE"
1130 PRINT "1. ADD LISTING"
1140 PRINT "2. RETRIEVE"
1150 GET A$:IF A$=" " GOTO 1150
1160 A=VAL(A$)
1170 IF A<1 OR A>2 GOTO 1150
1180 ON A GOTO 1190,1280
1190 PRINT CHR$(147)
1200 PRINT " TO ADD LISTING ENTER"
1210 PRINT " DATA LINE FOLLOWING"
1220 PRINT " LAST ONE SHOWN."
1230 PRINT "HIT ANY KEY"
1240 GET A$:IF A$="" GOTO 1240
1250 PRINT"TYPE THIS LINE FIRST:"
1260 PRINT" 201 R=";R+1
1270 LIST 200-999
1280 PRINT CHR$(147)
1290 PRINT" ENTER NAME TO"
1300 PRINT" SEARCH FOR:"
1310 INPUT NME$
1320 FOR ROW=1 TO R
1330 IF DTA$(ROW,1)=NME$ THEN GOTO 1390
1340 NEXT ROW
1350 PRINT" == NOT FOUND =="
1360 PRINT"HIT ANY KEY"
1370 GET A$:IF A$="" GOTO 1370
1380 GOTO 1110
1390 PRINT"FOUND:"
1400 PRINT FIELD$(1);":";DTA$(ROW,1)
1410 PRINT FIELD$(2);":";DTA$(ROW,2)
1420 PRINT FIELD$(3);":";DTA$(ROW,3)
1430 PRINT"HIT ANY KEY"
1440 GET A$:IF A$=" " GOTO 1440
1450 GOTO 1110
```

SEQUENTIAL FILE—WRITE TO TAPE

WHAT IT DOES

Writes a sequential data file to tape.

Variables

- NI: Number of items in file
- DTA$(n): Array storing data file.

How To Use Subroutine

Cassette users can only access sequential files. This subroutine provides a sample file writing routine that will take data that has been loaded into a string array, DTA$(n), and write it to tape. The same routine can be used with numeric arrays, simply by removing the variable type specifier, "$", from DTA$(n).

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

Line By Line Description

Line 140: Set number of items in file to 10.

Line 150: DIMension DTA$ to NI elements.

Line 170: OPEN the data file given the filename in quotes. You can substitute your own filename, or a variable, like F$, and then define F$ through user INPUT.

Line 180: Print, as the first item in the data file, the number of items in the file, NI.

Lines 190 to 210: PRINT each of the items in the array to the data file.

Line 220: 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 filename for "filename" in line 170.

RESULT

Data file written to tape.

```
10 REM ********************
20 REM *                  *
30 REM * SEQUENTIAL FILE  *
40 REM *   WRITE TO TAPE  *
50 REM *                  *
60 REM ********************
70 REM --------------------------------
80 REM       + + VARIABLES + +
90 REM    NI:       NUMBER OF ITEMS IN FILE
100 REM   DTA$(N): ARRAY STORING FILE
110 REM
120 REM --------------------------------

130 REM *** INITIALIZE ***

140 NI=10:DIM DTA$(NI)
150 GOTO 240

160 REM *** SUBROUTINE ***

170 OPEN 1,1,1,"FILENAME"
180 PRINT#1,NI
190 FOR N=1 TO NI
200 PRINT#1,DTA$(N)
210 NEXT N
220 CLOSE 1: RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 170
```

SEQUENTIAL FILE—READ FROM TAPE

WHAT IT DOES

Reads a sequential data file from tape.

Variables

- NI: Number of items in file
- DTA$(n): Array storing data file
- AD: Amount of room beyond number of items in file to allow for expansion.

How To Use Subroutine

Cassette users can only access sequential files. This subroutine provides a sample file reading routine that will take data that has been written to tape and load it into a string

array, DTA$(n). The same routine can be used with numeric arrays, simply by removing the variable type specifier, "$", from DTA$(n).

The routine will first read NI, the number of items in the file, from the tape. 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 tape, if you use the Write Routine supplied with this book.

Line By Line Description

Line 140: Set number of items that can be added to the file in one session to 10.

Line 160: OPEN the data file given the filename in quotes. You can substitute your own filename, or a variable, like F$, and then define F$ through user INPUT.

Line 170: INPUT the number of items currently in the file.

Line 180: DIMension the array to NI plus AD, allowing room for additional items.

Lines 190 to 210: INPUT each of the items in the array to the data file.

Line 220: CLOSE the file.

You Supply

- Your program should change the counter NI to reflect the number of items in the file each time an update is made.
- You should subsitute your file name for "filename" in line 160.

RESULT

Data file read from tape.

```
10 REM ********************
20 REM *                    *
30 REM * SEQUENTIAL FILE    *
40 REM *   READ FROM TAPE   *
50 REM *                    *
60 REM ********************
70 REM ----------------------------------
80 REM       + + VARIABLES + +
90 REM    NI:       NUMBER OF ITEMS IN FILE
100 REM   DTA$(N): ARRAY STORING FILE
110 REM
120 REM ----------------------------------

130 REM *** INITIALIZE ***
```

```
140 AD=10:GOTO 240

150 REM *** SUBROUTINE ***

160 OPEN 1,1,0,"FILENAME"
170 INPUT#1,NI
180 DIM DTA$(NI+AD)
190 FOR N=1 TO NI
200 INPUT#1,DTA$(N)
210 NEXT N
220 CLOSE 1:RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 160
```

SEQUENTIAL FILE—WRITE TO DISK

WHAT IT DOES

Writes a sequential data file to disk.

Variables

- NI: Number of items in file
- DTA$(n): Array storing data file.

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 beyond the scope of this book.

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

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

Line By Line Description

Line 140: Set number of items in file to 10.

Line 150: DIMension DTA$ to NI elements.

Line 170: OPEN the data file given the filename in quotes. You can substitute your own filename, or a variable, like F$, and then define F$ through user INPUT.

Line 180: Print, as the first item in the data file, the number of items in the file, NI.

Lines 190 to 210: PRINT each of the items in the array to the data file.

Line 220: 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 filename for "filename" in line 170.

RESULT

Data file written to disk.

```
10 REM ********************
20 REM *                  *
30 REM * SEQUENTIAL FILE  *
40 REM *   WRITE TO DISK  *
50 REM *                  *
60 REM ********************
70 REM -------------------------------------
80 REM       + + VARIABLES + +
90 REM    NI:       NUMBER OF ITEMS IN FILE
100 REM   DTA$(N): ARRAY STORING FILE
110 REM
120 REM -------------------------------------

130 REM *** INITIALIZE ***

140 NI=10
150 DIM DTA$(NI)
160 GOTO 250

170 REM *** SUBROUTINE ***

180 OPEN 8,8,8,"0:FILENAME,S,W"
190 PRINT#8,NI
200 FOR N=1 TO NI
210 PRINT#8,DTA$(N)
220 NEXT N
230 CLOSE 8:RETURN

240 REM *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 180
```

SEQUENTIAL FILE—READ FROM DISK

WHAT IT DOES

Reads a sequential data file from disk.

Variables

- NI: Number of items in file
- DTA$(n): Array storing data file
- AD: Amount of room beyond number of items in file to allow for expansion.

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$(n). The same routine can be used with numeric arrays, simply by removing the variable type specifier, "$", 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.

Line By Line Description

Line 150: Set number of items that can be added to the file in one session to 10.

Line 170: OPEN the data file given the filename in quotes. You can substitute your own filename, or a variable, like F$, and then define F$ through user INPUT.

Line 180: INPUT the number of items currently in the file.

Line 190: DIMension the array to NI plus AD, allowing room for additional items.

Lines 200 to 220: INPUT each of the items in the array to the data file.

Line 230: CLOSE the file.

Lines 260 to 280: Print data file to screen.

You Supply

Your program should change the counter NI, which should be redefined to reflect the number of items in the file each time an update is made. You should subsitute your file name for "filename" in line 160.

RESULT

Data file read from disk.

```
10 REM ********************
20 REM *                  *
30 REM * SEQUENTIAL FILE  *
40 REM * READ FROM DISK   *
50 REM *                  *
60 REM ********************
70 REM ---------------------------------------
80 REM       + + VARIABLES + +
90 REM    NI:       NUMBER OF ITEMS IN FILE
100 REM   DTA$(N): ARRAY STORING FILE
110 REM   AD:       NUMBER OF EMPTY SPACES AT
                    END OF FILE
120 REM
130 REM -------------------------------------

140 REM *** INITIALIZE ***

150 AD=10
160 GOTO 250

170 REM *** SUBROUTINE ***

180 OPEN 8,8,8,"0:FILENAME,S,R"
190 INPUT#8,NI
200 DIM DTA$(NI+AD)
210 FOR N=1 TO NI
220 INPUT#8,DTA$(N)
230 NEXT N
240 CLOSE 8:RETURN


240 REM *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 180
260 FOR N=1 TO NI
270 PRINT DTA$(N)
280 NEXT N
```


LAST FEDERAL
BANK

9 Business And Financial Subroutines

Although business programs have much 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, or sorts, like those in Chapter 6.

Business programs also have a need for keeping records. The results of one session may have to be stored for access in the next session. This brings up a requirement for data files which were discussed in the last section.

The subroutines in this chapter all handle some aspect of business. The first three calculate loan amounts, number of payments, and monthly payment. The number of years required to reach a given savings goal is handled by another subroutine, while compound interest is calculated by an additional module. Correct formatting of dollars-and-cents and dates are also accounted for by another pair. Temperature conversion and figuring miles per gallon (MPG) are also included as examples of the types of algorithms that might be useful in a typical business program.

Making your program easier to use through a clever menu is also explained. You may substitute the tasks of your choice and then write the appropriate branches.

LOAN AMOUNT

WHAT IT DOES

Calculates size of loan, given monthly payment, interest rate, and length of loan.

Variables

- RATE: Interest rate
- LOAN: Amount of loan
- NUMBER: Months of loan
- PAYMENT: Monthly payment.

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 the purchaser 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 the auto be financed by finding the extra $1400 from some other source, such as trading in a third car that the owner 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 150 to 170: 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.

Line 200: Change yearly interest rate in whole percent to decimal figure per month, e.g., 12 percent equals 12/1200 or .01 per month.

Line 210: Calculate loan amount.

Line 220: Round off to two decimal places.

Line 230: Return to main program.

Line 240: Access the subroutine.

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)
- NUMBER (number of months loan will run).

The subroutine will return LOAN, or the maximum loan amount given those parameters.

RESULT
Loan amount calculated.

```
10 REM ***************
20 REM *             *
30 REM * LOAN AMOUNT *
40 REM *             *
50 REM ***************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM     RATE:     INTEREST RATE
90 REM     LOAN:     AMOUNT OF LOAN
100 REM    NUMBER:   MONTHS OF LOAN
110 REM    PAYMENT:  MONTHLY PAYMENT
120 REM
130 REM ----------------------------

140 REM *** INITIALIZE ***

150 RATE=10
160 PAYMENT=10
170 NUMBER=36
180 GOTO 250

190 REM *** SUBROUTINE ***

200 RATE=RATE/1200
210 LOAN=PAYMENT*(1-(1+RATE)^-NUMBER)/RATE
220 LOAN=INT(LOAN*100+.5)/100
230 RETURN

240 REM *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 200
```

PAYMENT AMOUNT

WHAT IT DOES

Calculates monthly payment given interest rate, number of payments, and loan amount.

Variables

- RATE: Interest rate
- LOAN: Amount of loan
- NUMBER: Months of loan
- PAYMENT: Monthly payment.

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 varying the number of payments, you can calculate the affect 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 pay-offs 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 150 to 170: 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.

Line 200: Change RATE to percentage.

Line 210: Calculate PAYMENT.

Line 220: Round off PAYMENT to two decimal places.

Line 270: Print result.

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)
- NUMBER (number of months loan will run).

The subroutine will return PAYMENT, which is the monthly payment, against principal and interest.

RESULT
Loan payment calculated.

```
10 REM ******************
20 REM *                *
30 REM * PAYMENT AMOUNT *
40 REM *                *
50 REM ******************
60 REM ---------------------------
70 REM       + + VARIABLES + +
80 REM      RATE:     INTEREST RATE
90 REM      LOAN:     AMOUNT OF LOAN
100 REM     NUMBER:   MONTHS OF LOAN
110 REM     PAYMENT:  MONTHLY PAYMENT
120 REM
130 REM ---------------------------

140 REM *** INITIALIZE ***

150 LOAN=100
160 RATE=10
170 NUMBER=36
180 GOTO 260

190 REM *** SUBROUTINE ***

200 RATE=RATE/100
210 PAYMENT=LOAN*(RATE/12)/(1-(1+(RATE/12))^-NUMBER)
220 PAYMENT=INT(PAYMENT*100+.5)/100
230 RETURN

250 REM *** YOUR PROGRAM STARTS HERE ***

260 GOSUB 200
270 PRINT PAYMENT
```

NUMBER OF PAYMENTS

WHAT IT DOES

Calculates number of payments given interest rate, monthly payment, and loan amount.

Variables

- RATE: Interest rate
- LOAN: Amount of loan
- NUMBER: Months of loan.

- PAYMENT: Monthly payment
- WP: Number of whole payments
- FP: Amount of final payment.

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 170 to 190: 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.

Line 220: Change RATE to monthly decimal value, that is, 12 percent per year equals 12/1200 or .01 per month.

Line 230: Calculate number of payments.

Line 240: Calculate number of whole payments.

Line 250: Figure amount of final, partial payment.

Line 280: Access the subroutine.

Lines 290 to 310: Print results.

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)
- PAYMENT (the amount of the monthly payment.

The subroutine will return NUMBER, which is the number of monthly payments that will be required.

RESULT
Number of loan payments calculated.

```
10 REM ******************
20 REM *                *
30 REM * NUMBER PAYMENTS *
40 REM *                *
50 REM ******************
60 REM ------------------------------------
70 REM       + + VARIABLES + +
80 REM    RATE:     INTEREST RATE
90 REM    LOAN:     AMOUNT OF LOAN
100 REM   NUMBER:   MONTHS OF LOAN
110 REM   PAYMENT:  MONTHLY PAYMENT
120 REM   WP:       NUMBER OF WHOLE PAYMENTS
130 REM   FP:       AMOUNT OF FINAL PAYMENT
140 REM
150 REM ---------------------------------

160 REM *** INITIALIZE ***

170 LOAN=1500
180 RATE=12
190 PAYMENT=100
200 GOTO 280

210 REM *** SUBROUTINE ***

220 RATE=RATE/1200
230 NUMBER=LOG(PAYMENT/(PAYMENT-AMOUNT*RATE))/LOG(1+RATE)
240 WP=INT(NUMBER)
250 FP=PAYMENT*(NUMBER-WP)
260 RETURN

270 REM *** YOUR PROGRAM STARTS HERE ***

280 GOSUB 200
290 PRINT WP;"PAYMENTS OF $ ";PAYMENT
300 PRINT "PLUS FINAL PAYMENT OF"
310 PRINT "$";FP
```

YEARS TO REACH DESIRED VALUE

WHAT IT DOES

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

Variables

- RATE: Interest rate
- YEARS: Years compounded
- FUTURE: Future value desired
- AMOUNT: Amount to be compounded.

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 160 to 190: 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.

Line 220: Change RATE to decimal figure.

Line 230: Calculate number of years needed to produce the goal value.

Lines 240 to 260: Figure number of whole months and years.

Line 290: Access the subroutine.

Lines 300 to 370: Print results.

You Supply

You must define these variables:

- FUTURE (desired future value)
- RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent)
- PERIODS (number of compounding periods per year).

The subroutine will return YEARS, or the number of years that will be required to reach the desired value.

RESULT
Years calculated.

```
10 REM ******************
20 REM *                *
30 REM *  YEARS TO REACH  *
40 REM *  DESIRED VALUE   *
50 REM *                *
60 REM ******************
70 REM -------------------------------
80 REM       + + VARIABLES + +
90 REM      RATE:     INTEREST RATE
100 REM     YEARS:    YEARS COMPOUNDED
110 REM     FUTURE:   FUTURE VALUE DESIRED
120 REM     AMOUNT:   AMOUNT TO BE COMPOUNDED
130 REM
140 REM -------------------------------

150 REM *** INITIALIZE ***

160 RATE=10
170 AMOUNT=1000
180 PERIOD=365
190 FUTURE=2000
200 GOTO 290

210 REM *** SUBROUTINE ***

220 RATE=RATE/100
230 YEARS=LOG(FUTURE/AMOUNT)/((LOG(1+RATE
    /PERIODS))*PERIODS)
240 MTH=YEARS-INT(YEARS)
250 MTH=INT(MTH*12)
260 YEARS=INT(YEARS)
270 RETURN

280 REM *** YOUR PROGRAM STARTS HERE ***

290 GOSUB 220
300 PRINT CHR$(147)
310 PRINT "$";AMOUNT;" WILL"
320 PRINT "COMPOUND TO $";FUTURE
330 PRINT "IN ";YEARS;" YEARS
340 PRINT MTHS;" MONTHS"
350 PRINT "AT ";RATE*100;" PERCENT"
360 PRINT "COMPOUNDED ";PERIODS
370 PRINT "TIMES A YEAR."
```

COMPOUND INTEREST

WHAT IT DOES

Calculates compounded amount of investment, given original value, interest rate, and time period.

Variables

- RATE: Interest rate
- YEARS: Years compounded
- FUTURE: Future value
- AMOUNT: Amount to be compounded.

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 150 to 180: Define original principal AMOUNT, the interest RATE in whole percent, and the number of YEARS to be compounded.

Line 210: Change RATE to decimal value.

Line 220: Calculate FUTURE value.

Line 230: Round off value to two decimal places.

Line 260: Access the subroutine.

Lines 270 to 300: Print results.

You Supply

You must define these variables:

- AMOUNT (the original amount)
- RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent)
- YEARS (number of years to be compounded).

The subroutine will return FUTURE, or value of the compounded investment.

RESULT

Compound interest calculated.

```
10 REM **********************
20 REM *                    *
30 REM * COMPOUND INTEREST *
40 REM *                    *
50 REM **********************
60 REM ----------------------------------
70 REM       + + VARIABLES + +
80 REM      RATE:    INTEREST RATE
90 REM      YEARS:   YEARS COMPOUNDED
100 REM     FUTURE:  FUTURE VALUE
110 REM     AMOUNT:  AMOUNT TO BE COMPOUNDED
120 REM
130 REM ----------------------------------

140 REM *** INITIALIZE ***

150 RATE=10
160 AMOUNT=1000
170 PERIOD=365
180 YEARS=10
190 GOTO 260

200 REM *** SUBROUTINE ***

210 RATE=RATE/100
220 FUTURE=AMOUNT*(1+RATE/PERIODS)^(PERIODS*YEARS)
230 FUTURE=INT(FUTURE*100+.5)/100
240 RETURN

250 REM *** YOUR PROGRAM STARTS HERE ***

260 GOSUB 210
270 PRINT CHR$(147)
280 PRINT "$";AMOUNT;" LEFT"
290 PRINT YEARS;" YEARS WILL"
300 PRINT " GROW TO $";FUTURE
```

RATE OF RETURN

WHAT IT DOES

Calculates interest rate, given present and future value, and number of compounding periods.

Variables

- RATE: Interest rate
- YEARS: Years compounded

- FUTURE: Future value
- AMOUNT: Amount to be compounded.

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 150 to 180: 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 user enter these values.

Line 210: Figure interest RATE.

Line 220: Change RATE to whole percent.

Line 250: Access the subroutine.

Lines 260 to 310: Print results.

You Supply

You must define these variables:

- AMOUNT (present value)
- FUTURE (future value)
- YEARS (number of years to be compounded)
- PERIODS (number of compounding periods).

The subroutine will return RATE, the interest rate.

RESULT

Interest rate calculated.

```
10 REM ******************
20 REM *                 *
30 REM * RATE OF RETURN *
40 REM *                 *
50 REM ******************
60 REM --------------------------------
70 REM       + + VARIABLES + +
80 REM    RATE:    INTEREST RATE
90 REM    YEARS:   YEARS COMPOUNDED
100 REM   FUTURE:  FUTURE VALUE
110 REM   AMOUNT:  AMOUNT TO BE COMPOUNDED
120 REM
130 REM --------------------------------
```

```
140 REM *** INITIALIZE ***

150 YEARS=10
160 AMOUNT=1000
170 PERIOD=365
180 FUTURE=2000
190 GOTO 250

200 REM *** SUBROUTINE ***

210 RATE=((FUTURE/AMOUNT)^(1/(PERIODS*YEARS))-1)*PERIODS
220 RATE=RATE*100
230 RETURN

240 REM *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 210
260 PRINT CHR$(147)
270 PRINT "$";AMOUNT
280 PRINT "TO PRODUCE $";FUTURE
290 PRINT "IN ";YEARS;"YEARS"
300 PRINT "REQUIRES AN INTEREST"
310 PRINT "RATE OF ";RATE
```

DOLLARS AND CENTS

WHAT IT DOES

Formats dollars and cents.

Variables

- A: Dollars and cents amount to be formatted
- D$: The dollar figure, formatted.

How To Use Subroutine

Business programs frequently work with money, and it is desirable to format the dollars and cents output into conventional form. That is, the decimal point is followed by two numbers only, and a trailing zero or two will be included as needed. A figure should appear as $12.50 or $13.00 rather than $12.5 or $13.

This subroutine will round off any fractional cent larger than one-half cent to the next larger penny and round anything smaller to the next lower amount. The correct number of zeros will be added to amounts evenly divisible by 10.

NOTE: the caret symbol (^) in the program listing stands for the UP ARROW on the Commodore 64 keyboard, the key between RESTORE and "*".

Line By Line Description

Line 140: Define dollars and cents amount to be formatted.

Line 170: Define number of decimal places to round off figures. For dollars-and-cents, P will always equal 2.

Lines 180 to 190: Round off figure by adding 5.5, multiplying it by 100, taking the integer part of the result, and dividing by 100.

Line 200: Produce string representation of result.

Lines 210 to 240: Check D$ for a decimal point.

Line 250: If no decimal point found, add ".00".

Lines 260 to 300: If only one number to right of decimal point, add "0."

Line 330: Access subroutine.

Line 340: Print result.

You Supply

Your program should define A, the dollars-and-cents amount to be formatted. The routine will return D$, which is the formatted figure.

RESULT

Dollar amount is properly formatted for display.

```
10 REM **********************
20 REM *                    *
30 REM *  DOLLARS AND CENTS  *
40 REM *                    *
50 REM **********************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM      A:   DOLLARS AND CENTS
90 REM           AMOUNT TO BE FORMATTED
100 REM    D$:   DOLLAR FIGURE
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 A=55.345
150 GOTO 330

160 REM *** SUBROUTINE ***
```

```
170 P=2
180 C=A+5.5*10^-(P+1)
190 B=INT(C*10^P)/10^P
200 D$="$"+STR$(B)
210 FOR N=1 TO LEN(D$)
220 T$=MID$(D$,N,1)
230 IF T$="." GOTO 270
240 NEXT N
250 D$=D$+".00"
260 RETURN
270 L$=LEFT$(D$,N)
280 R$=MID$(D$,N+1)
290 IF VAL(R$)<10 THEN R$=R$+"0"
300 D$=L$+R$
310 RETURN

320 REM *** YOUR PROGRAM STARTS HERE ***

330 GOSUB 170
340 PRINT D$
```

TEMPERATURE

WHAT IT DOES

Calculates Celsius and Fahrenheit.

Variables

- F: Fahrenheit temperature
- C: Celsius temperature.

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 150 to 160: Figure Fahrenheit temperature.

Lines 170 to 180: Figure Celsius temperature.

Lines 210 to 250: User enters temperature to convert.

Lines 260 to 270: Check to see if Fahrenheit or Celsius.

Lines 280 to 290: If wrong, make user reenter.

Lines 300 to 310: Access proper subroutine.

Line 320: Print results of conversion.

You Supply

- You should define a value for either F or C, depending on which way the conversion will go. This will usually be input from the keyboard.
- The alpha character ending the input, either "F" or "C", should be supplied to determine which type of conversion will be activated.

RESULT

Temperature converted to alternate value.

```
10 REM ***************
20 REM *             *
30 REM * TEMPERATURE *
40 REM *             *
50 REM ***************

60 REM *** INITIALIZE ***

70 GOTO 200
80 REM ----------------------------
90 REM       + + VARIABLES + +
100 REM      F: FAHRENHEIT
110 REM      C: CELSIUS
120 REM
130 REM ----------------------------

140 REM *** SUBROUTINE ***

150 C=VAL(AN$)
160 F=INT((9/5)*C+32):RETURN
170 F=VAL(AN$)
180 C=INT((F-32)*(5/90)):RETURN

190 REM *** YOUR PROGRAM STARTS HERE ***
```

```
200 PRINT
210 PRINT"ENTER TEMPERATURE"
220 PRINT"IN THIS FORM:"
230 PRINT CHR$(34);52C;CHR$(34);" OR "
240 PRINT CHR$(34);98F;CHR$(34);"."
250 INPUT AN$
260 A$=RIGHT$(AN$,1)
270 IF A$="F" OR A$="C" THEN GOTO 310
280 PRINT "WRONG FORMAT."
290 GOTO 220
300 IF A$="F" THEN GOSUB 190
310 IF A$="C" THEN GOSUB 170
320 PRINT F;"F. = ";C;"C."
```

DATE FORMATTER

WHAT IT DOES

Formats dates to MM/DD/YY style.

Variables

- DTE$: Date
- MNTH$: Months
- DAY$: Day
- YEAR$: Year.

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 zero 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

Line 160: Enter month to be formatted.

Lines 170 to 180: Check to see that MNTH is at least 1 but no more than 12.

Line 190: If MNTH is less than 10 then MNTH$ = "0" plus the string representation of MNTH. That is, "9" becomes "09."

Lines 200 to 220: Enter day of month, which must be at least 1 and less than 31.

Line 230 to 250: Check to see if month should have only 30 days, and force user to reenter if an illegal date has been entered.

Line 260: Enter year.

Lines 270 to 310: If leap year, then February may have 29 days, otherwise, only 28 allowed.

Line 320: If DAY is less than 10 then add leading "0."

Line 330: Construct MM/DD/YY string.

Line 370: Access the subroutine.

Line 380: Print result.

You Supply

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

RESULT

Properly formatted date.

```
10 REM ******************
20 REM *                 *
30 REM * DATE FORMATTER *
40 REM *                 *
50 REM ******************
60 GOTO 360
70 REM ---------------------------
80 REM       + + VARIABLES + +
90 REM        DTE$:  DATE
100 REM       MNTH$: MONTHS
110 REM       DAY$:  DAY
120 REM       YEAR$: YEAR
130 REM
140 REM ---------------------------

150 REM *** SUBROUTINE ***
```

```
160 INPUT"ENTER MONTH:";MNTH$
170 MNTH=VAL(MNTH$)
180 IF MNTH<1 OR MNTH>12 GOTO 160
190 IF MNTH<10 THEN MNTH$="0"+RIGHT$(MNTH$,1)
200 INPUT"ENTER DAY : ";DAY$
210 DAY=VAL(DAY$)
220 IF DAY<1 OR DAY>31GOTO 200
230 IF MNTH=4 OR MNTH=6 OR MNTH=9 OR MNTH=11 GOTO 250
240 GOTO 260
250 IF DAY>30 GOTO 200
260 INPUT"ENTER YEAR : ";YEAR$
270 YEAR=VAL(YEAR$)
280 IF YEAR/4<>INT(YEAR/4)GOTO 310
290 IF MNTH=2 AND DAY>29 GOTO 200
300 GOTO 320
310 IF MNTH=2 AND DAY>28 GOTO 200
320 IF DAY<10 THEN DAY$="0"+RIGHT$(DAY$,1)
330 DTE$=MNTH$+"/"+DAY$+"/"+YEAR$
340 RETURN

350 REM *** YOUR PROGRAM STARTS HERE ***

360 PRINT CHR$(147)
370 GOSUB 160
380 PRINT DTE$
```

MENU

WHAT IT DOES

Menu template for user programs.

Variables

- NC$: Number of choices on menu.

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, i.e., those that are out of the allowed range. User input for up to nine selections is 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 1000, 2000, 3000, and 4000; you must write those routines yourself.

Line By Line Description

Line 70: Define number of menu choices available.

Lines 150 to 160: Clear screen, and present menu title. Line 160 may be changed by user to label specific menu.

Lines 180 to 210: Labels for the menu choices.

Line 230: Prompt user choice.

Line 240: Wait for user input.

Lines 250 to 260: If entry is less than 1 or larger than the number of choices available, go back and continue waiting.

Line 270: Access subroutine specified by user, at Lines 1000,2000,3000 or 4000.

You Supply

- Define NC to equal the number of menu choices.
- Write subroutines to accomplish your various tasks, using line 270 as a model to direct control.

RESULT

Operator can select from list of menu choices.

```
10 REM ********
20 REM *      *
30 REM * MENU *
40 REM *      *
50 REM ********

60 REM *** INITIALIZE ***

70 NC=4
80 GOTO 300
90 REM ----------------------------
100 REM       + + VARIABLES + +
110 REM       NC:  NUMBER OF MENU CHOICES
120 REM
130 REM ----------------------------

140 REM *** SUBROUTINE ***
```

```
150 PRINT CHR$(147)
160 PRINT TAB(6)"** MENU **"
170 PRINT CHR$(17);CHR$(17)
180 PRINT TAB(3)"1. FIRST CHOICE"
190 PRINT TAB(3)"2. SECOND CHOICE"
200 PRINT TAB(3)"3. THIRD CHOICE"
210 PRINT TAB(3)"4. FOURTH CHOICE"
220 PRINT CHR$(17)
230 PRINT TAB(6)"ENTER CHOICE"
240 GET A$:IF A$="" GOTO 240
250 A=VAL(A$)
260 IF A<1 OR A>NC GOTO 240
270 ON A GOSUB 1000,2000,3000,4000
280 RETURN

290 REM *** YOUR PROGRAM STARTS HERE ***

300 PRINT
310 GOSUB 150
320 END

990 REM *** FIRST SUBROUTINE ***
1000 RETURN

1990 REM *** SECOND SUBROUTINE ***
2000 RETURN

2990 REM *** THIRD SUBROUTINE ***
3000 RETURN

3990 REM *** FOURTH SUBROUTINE ***
4000 RETURN
```

TIME ADDER

WHAT IT DOES

Totals seconds, minutes, and hours.

Variables

- TM: Total minutes
- TS: Total seconds
- TH: Total hours
- MIN: Minutes to be added in
- HOUR: Hours to be added in
- SECS: Seconds to be added in.

How To Use Subroutine

Various programs, such as timers, must add minutes and 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 160 to 180: Define the current total minutes, hours, and seconds.

Lines 190 to 210: Define the number of hours, minutes, and seconds to be added to the above variables.

Lines 240 to 300: Add current total to additional minutes, seconds, and hours, in form of total number of seconds..

Lines 280 to 290: Figure whole hours, and subtract that number of seconds (hours × 3600) from the total number of seconds.

Line 300 to 310: Figure whole minutes, and subtract that number of seconds (minutes × 60) from total seconds.

Line 340: Access the subroutine.

Lines 350 to 380: Print the results.

You Supply

You must supply start up values for TS, TM, and TH, or else they will default to those shown in lines 160 to 180. You may change these defaults to zeros if you wish. Your program should furnish MIN, HOUR, and SECS values.

RESULT

New total time calculated.

```
10 REM **************
20 REM *              *
30 REM * TIME ADDER *
40 REM *              *
50 REM **************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM        TM:   TOTAL MINUTES
90 REM        TS:   TOTAL SECONDS
100 REM       TH:   TOTAL HOURS
110 REM       MIN:  MINUTES TO BE ADDED
120 REM       HOUR: HOURS TO BE ADDED
130 REM       SECS: SECONDS TO BE ADDED
140 REM
150 REM ----------------------------
```

```
155 REM *** INITIALIZE ***

160 TM=54
170 TH=40
180 TS=30
190 MIN=30
200 HOUR=2
210 SECS=30
220 GOTO 340

230 REM *** SUBROUTINE ***

240 TM=TM+MIN*60
250 TS=TS+SECS
260 TH=TH+HOUR*3600
270 TS=TM+TS+TH
280 TH=INT(TS/3600)
290 TS=TS-TH*3600
300 TM=INT(TS/60)
310 TS=TS-TM*60
320 RETURN

330 REM *** YOUR PROGRAM STARTS HERE ***

340 GOSUB 340
350 PRINT CHR$(147)
360 PRINT"SECONDS:";TS
370 PRINT"MINUTES:";TM
380 PRINT"HOURS:";TH
```

MPG

WHAT IT DOES

Calculates auto miles per gallon.

Variables

- BEGN: Starting odometer reading
- ODOM: Current odometer reading
- GALLNS: Gallons of gas consumed between readings.

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 fill up. In other words, the MPG can be figured for the aggregate of a number of tanksful of gas.

Line By Line Description

Lines 140 to 160: 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.

Line 190: Calculate MPG.

Line 200: Round off MPG.

Line 230: Access the subroutine.

Lines 240 to 250: Print results.

You Supply

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

RESULT
MPG calculated.

```
10 REM *******
20 REM *     *
30 REM * MPG *
40 REM *     *
50 REM *******
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM       BEGN:   STARTING ODOMETER
90 REM       ODOM:   CURRENT ODOMETER
100 REM      GALLNS: GALLONS GAS USED
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 ODOM=36420
150 BEGN=36001
160 GALLNS=13.8
170 GOTO 230

180 REM *** SUBROUTINE ***

190 MPG=(ODOM-BEGN)/GALLNS
200 MPG=INT(MPG*10+.5)/10
210 RETURN

220 REM *** YOUR PROGRAM BEGINS HERE ***

230 GOSUB 190
230 PRINT CHR$(147)
250 PRINT "MPG=";MPG
```

GOSUB

INPUT

RETURN

STR$()

PRINT

LIST

GOTO

POKE

DIM

CHR$()

THEN

RUN

PEEK

LOAD

NEXT

RESTORE

10 Add New Functions To Commodore Basic

When compared to the BASICs supplied with other computers in the same price range, the BASIC 2.0 included in the Commodore 64 stacks up very well indeed. It has a powerful screen editor that allows changing program lines just by typing over them, and most of the features of standard Microsoft BASIC.

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 these important features with your Commodore computer.

If your only familiarity with BASIC is through your use of your Commodore 64, some of these statements and functions may appear strange to you. You may even wonder what they can be used for. A few are rather rare and not commonly found in the majority of BASICS. This section will try to show you how to use the new functions, as well as providing tips on why you'd want to put them in your own programs.

Commodore Basic V.2 has some serious deficiencies in the area of manipulating strings of characters. For example, it is somewhat tedious to find out whether a given string is located within a second string that is the same size or longer. Other BASICs have a statement called INSTR which will do this automatically, returning a value equivalent to the position in which the substring is found (or 0 if it is not found).

This capability might be useful in writing a word processing program or a game program in which we might want to find out if the string entered by the user has a target word or phrase.

For example, MAIN$, as INPUT, might equal "My name is Michael." We wish to know if the substring "name is" (SUB$) is included and, if so, where. Running the INSTR subroutine will tell us that "name is" starts at position 4 in MAIN$. In all probability, the name itself, Michael, begins at position 4 + LEN(SUB$) + 1 or position 12. To extract that name from the string, we take each middle character of MAIN$ until we encounter a space. That might be accomplished like this (with I being the start position of the SUB$, as determined by INSTR):

```
1000 E=LEN(MID$(MAIN$,I))
1010 FOR N=I TO E
1020 T$=MID$(MAIN$,N,1)
1030 IF T$=CHR$(32) GOTO 1060
1040 NME$=NME$+T$
1050 NEXT N
1060 RETURN
```

Sometimes, after extracting a desired string in this manner, you will want to replace it with another string of your choice. This is most common in word processing programs, where a word or phrase will be replaced.

The MID$ = (referred to as "MID$ on the left of the equals sign) function permits you to replace a portion of MAIN$ with SUB$. SUB$ has to be of the same length as the string it replaces. If you want to insert a LONGER string to replace a SHORTER one, use another subroutine, INSERT A STRING.

Another statement likely to be valued by those handling text and other strings using commas (one of several so-called "string delimiters") is the LINEINPUT feature. Your user may type in any characters to the INPUT prompt. Keys will be accepted until RETURN is pressed, as normal. However, the question mark prompt can be changed to any character of your choice—or omitted entirely.

STRING$ adds a feature to BASIC 2.0 that allows easily building a string of any size, using a character of your choice. Say you need a string, BALL$, composed of 20 solid ball graphic characters (CHR$(81)). This subroutine will build it for you automatically.

MOD is a math-type function that returns the remainder when a number is divided by another. There are many applications for this feature in games and other programming.

SWAP is a handy way to exchange the values of two variables, even though the SWAP statement is not included in PET BASIC 2.0. Sorts and other modules that you might design can use this feature.

LISTER and CHR$ are two modules that handle chores that can also be taken care of quickly from command mode, just by typing in the command. However, by including either in your BASIC program with high line numbers, like 30000 and 31000, you can call them at any time by typing the shorter RUN 30000 or RUN 31000.

MOD FUNCTION

WHAT IT DOES

Simulates MOD function found in other BASICs. Returns remainder when one number is divided by another.

Variables

- MOD: Remainder when NU is divided by DIV
- NU: Counter being checked for remainder
- DIV: Divisor.

How To Use Subroutine

Frequently, it will be desirable to know the remainder when one number is divided by another. For example, you may wish to display 16 lines of a data file on the screen of the Commodore 64 and then pause until the user presses a key.

MOD will check to see when a counter, NU, is evenly divisible by another number. When data items number 16, 32, 48, etc., are displayed, the pause routine can be called. If the number is not evenly divisible, MOD will equal something other than zero—the remainder.

So, using this function, lines such as the following are legal:

```
100 IF MOD=0 GOSUB 200
...
200 PRINT"HIT ANY KEY TO CONTINUE"
210 GET A$:IF A$=" " GOTO 210
220 PRINT CHR$(147)
230 RETURN
```

Line By Line Description

Line 140: Divide counter, NU, by DIV. Subtract from that the integer portion of the same result, producing the remainder.

Line 170: Define divisor.

Line 180: Increment counter.

Line 190: Access the subroutine.

Line 200: Test to see if evenly divisible.

Line 210: Return to line 180 to repeat.

You Supply

You should provide a value for the divisor, DIV, each time it is changed. That is, if you always want to check to see if the number can be divided evenly by 16, DIV can be defined once at the beginning of the program. Your counter, NU, should be changed in value by your program so that it can be checked against FLAG. You may check other variables simply by making NU equal to them just before calling the subroutine. NU itself will NOT be altered by the subroutine. For example:

```
100 NU=C
110 GOSUB xxx
```

RESULT
MOD will equal remainder; zero if none.

```
10 REM ***********
20 REM *         *
30 REM * MODULUS *
40 REM *         *
50 REM ***********
60 REM ----------------------
70 REM    ++VARIABLES++
80 REM    MOD: RESULT
90 REM    NU:  COUNTER
100 REM   DIV: DIVISOR
110 REM ----------------------
120 GOTO 180

130 REM *** SUBROUTINE ***

140 MOD=NU/DIV-INT(NU/DIV)
150 RETURN
```

```
160 REM *** YOUR PROGRAM STARTS HERE ***

170 DIV=30
180 NU=NU+1
190 GOSUB 140
200 IF MOD=0 THEN PRINT NU;" EVENLY DIVISIBLE BY";DIV
210 GOTO 180
```

INSTR FUNCTION

WHAT IT DOES

Simulates INSTR function found in other BASICs. Checks string for inclusion of shorter substring.

Variables

- MAIN$: String to be searched for the substring
- SUB$: The substring you are looking for
- BEGN: Starting position for the search
- PLACE: Position where the substring was found, if found.

How To Use Subroutine

Checking one string of alphanumeric characters for the inclusion of a smaller string is done with the INSTR statement in many BASICs. But, even though the Commodore 64 lacks this feature, this subroutine will mimic it quite closely.

It will run through each character in the main string (MAIN$), beginning at position BEGN. The routine then checks to see if the next LTH characters of MAIN$ equal the substring, SUB$. LTH is the length of the substring. If so, then PLACE is given the value of the starting position. If SUB$ is not present, PLACE will equal 0.

Note: this subroutine will find only the FIRST occurrence of the SUB$ in MAIN$. If you want to check further, make BEGN equal PLACE + LTH + 1, and run the subroutine again. In this case, the search will begin after the place where the substring was first located.

If you wish to find a word, rather than a series of characters, you should add spaces to either side of SUB$, when you enter it. That is, entering "and" will locate those letters in "sand" or "inland" as well as " and " used as a distinct word. So, entering " and ", that is, "(SPACE)and(SPACE)" will search only for those occurrences that stand alone. Unfortunately, punctuation marks will count as a character, so words searched for at ends of sentences have to specify only the initial space.

Line By Line Description

Lines 150 to 170: Define main string of characters, the substring to be searched for, and the beginning search point.

Line 200: Determine length of substring.

Line 210: Start FOR-NEXT loop from beginning point, BEGN, to length of the MAIN$.

Line 220: If a string corresponding to the MID$ portion of MAIN$ defined by the FOR-NEXT counter and the length of the substring equals SUB$, then return. That is, when MID$(MAIN$, PLACE, LTH) equals SUB$, then SUB$ has been located in MAIN$ at position PLACE.

Line 230: Repeat loop.

Line 240: If SUB$ was not found, then PLACE equals 0.

Line 270: Access the subroutine.

Line 280: Print result.

You Supply

You should supply a value for MAIN$ and SUB$ each time the routine is run. BEGN will always have a default value of 1, unless you define it as something else just before calling the subroutine.

RESULT

Position of string searched for will be stored in variable PLACE.

```
10 REM *********
20 REM *       *
30 REM * INSTR *
40 REM *       *
50 REM *********
60 REM --------------------------------
70 REM       ++ VARIABLES ++
80 REM    MAIN$: STRING TO BE SEARCHED
90 REM    SUB$:  STRING TO BE SEARCHED FOR
100 REM   BEGN:  START POSITION FOR SEARCH
110 REM   PLACE: POSITION WHERE FOUND
120 REM
130 REM --------------------------------

140 REM *** INITIALIZE ***

150 MAIN$="THIS IS THE STRING"
160 SUB$="THE"
170 BEGN=1
180 GOTO 270

190 REM *** SUBROUTINE ***
```

```
200 LTH=LEN(SUB$)
210 FOR PLACE=BEGN TO LEN(MAIN$)
220 IF MID$(MAIN$,PLACE,LTH)=SUB$ THEN RETURN
230 NEXT PLACE
240 PLACE=0:BEGN=1
250 RETURN

260 REM *** YOUR PROGRAM BEGINS HERE ***

270 GOSUB 200
280 IF PLACE=0 GOTO 310
290 PRINT "FOUND AT POSITION";PLACE
300 END
310 PRINT "NOT FOUND."
```

REPLACE STRING

WHAT IT DOES

Simulates MID$= function found in other BASICs. Replace middle portion of a string with another string.

Variables

- TARGT$: Main string
- SUB$: String to be replaced
- PLACE: Position to put SUB$.

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 three in the target string, and substitute the SUB$.

You Supply

- Define TARGT$, SUB$, and PLACE just before the subroutine is called.

RESULT

TARGT$ will be changed to include SUB$.

```
10 REM ******************
20 REM *                *
30 REM * REPLACE STRING$ *
40 REM *                *
50 REM ******************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM     TARGT$: MAIN STRING
90 REM     SUB$:   STRING TO BE REPLACED
100 REM    PLACE:  POSITION TO PUT SUB$
110 REM
120 REM ----------------------------

130 REM *** INITIALIZE ***

140 SUB$="TEST"
150 TARGT$="REPLACEMENT MADE"
160 PLACE=7
170 GOTO 240

180 REM *** SUBROUTINE ***

190 L$=LEFT$(TARGT$,PLACE)
200 R$=MID$(TARGT$,PLACE+LEN(SUB$)+1)
210 TARGT$=L$+SUB$+R$
220 RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 190
250 PRINT TARGT$
```

INSERT STRING

WHAT IT DOES

Inserts string into another.

Variables

- TARGT$: Main string
- SUB$: String to be inserted into main string
- PLACE: Position to put SUB$.

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 that with a few limitations.

Like all Commodore 64 strings, neither TARGT$ nor SUB$ can be longer than 255 characters. The resulting string with SUB$ inserted must be shorter than 255 characters as well.

In the subroutine as written, the target string is "THIS IS THE MAIN STRING OF CHARACTERS", while the SUB$ is defined as "TEST". Since the PLACE where we want to insert it is position 7, the new string will read: "THIS IS THE TEST MAIN STRING OF CHARACTERS".

Line By Line Description

Lines 140 to 160: Define the SUB$, the TARGT$, and PLACE where the SUB$ will be inserted.

Line 190: Take the leftmost characters in the target string, up to, and including, position PLACE.

Line 200: Take the rightmost characters in the target string, starting with one after position PLACE.

Line 210: Construct new target string, from L$, SUB$, and R$.

Line 240: Access the subroutine.

Line 250: Print result.

You Supply

Values for:

- The main string TARGT$
- The string to be inserted SUB$
- The position where it will be put, PLACE.

RESULT

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

```
10 REM ******************
20 REM *                 *
30 REM * INSERT STRING$ *
40 REM *                 *
50 REM ******************
60 REM ----------------------------
70 REM       ++ VARIABLES ++
80 REM    TARGT$: MAIN STRING
90 REM    SUB$:   STRING TO BE INSERTED
100 REM   PLACE:  POSITION TO PUT SUB$
110 REM
120 REM ----------------------------
```

```
130 REM *** INITIALIZE ***

140 SUB$="TEST "
150 TARGT$="TARGET STRING LETTERS"
160 PLACE=12
170 GOTO 240

180 REM *** SUBROUTINE ***

190 L$=LEFT$(TARGT$,PLACE)
200 R$=MID$(TARGT$,PLACE+1)
210 TARGT$=L$+SUB$+R$
220 RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 190
250 PRINT TARGT$
```

LISTER

WHAT IT DOES

Simulates LLIST function found in other BASICs. Send program listings to serial line printer.

Variables

None.

How To Use Subroutine

Sends current program in memory to line printer. You may renumber and include at the end of your program with, say, line numbers beginning at 30000. Then, type RUN 30000 to produce the listing.

If you are using one of the other merging techniques described in Chapter 1 you may also store this subroutine on disk with a high line number, then type MERGE"LISTER" or MERGE"LISTER",8 to append it to your program. Call it as needed.

This subroutine works only with serial printers, such as the Commodore 1525. Parallel printers require a special hardware adapter and software.

Line By Line Description

Line 130: CLOSE channel 4, just in case it had been left open previously.

Line 140: OPEN fresh channel.

Line 150: Redirect any output normally going to the screen, device #1, to device #4, the printer.

Line 160: LIST program (sends listing to printer).

Line 170: CLOSE channel, end.

You Supply

If you wish, you may edit line 160 to read LIST 0-29999 so that after the subroutine has been renumbered to 30000 or higher, only the program will be listed.

Or, you can edit the LIST line at any time so that selected lines will be listed to the lineprinter. Changing the line to LIST 100-500 will cause the lineprinter to print only those lines, inclusive.

RESULT

Listing sent to line printer.

```
10 REM **********
20 REM *        *
30 REM * LISTER *
40 REM *        *
50 REM **********
60 GOTO 190
70 REM ----------------------------
80 REM      ++ VARIABLES ++
90 REM         NONE
100 REM
110 REM ----------------------------
120 GOTO 210

130 REM *** SUBROUTINE ***

140 CLOSE4
150 OPEN4,4
160 CMD4:LIST 230-
170 PRINT#4:CLOSE4
180 END
190 RETURN

200 REM *** YOUR PROGRAM STARTS HERE ***

210 PRINT
220 GOSUB 140
```

CHR$ VALUE

WHAT IT DOES

Returns Commodore CHR$ code for any key.

Variables

- A: CHR$ value of last key pressed.

How To Use Subroutine

When POKing to character memory, or printing graphics or alphanumerics via PRINT CHR$(n), it is necessary to know the special Commodore CHR$ code for a given key. If many keys are used, looking them all up on a table in a reference book 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 is fed to a Commodore 64, it 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 differing operating systems and languages. Commodore departs somewhat from this code for the Commodore 64, especially when using the graphics characters. PEEKing and POKing characters is done using the Commodore character set listing, not the standard ASCII table. However, the standard alphanumeric symbols are accurately portrayed with the CHR$(n) statement. That is, PRINT CHR$(65) will produce an uppercase "A" in Commodore BASIC, just like in other BASICs.

But even if you don't have a modem and aren't communicating with other computerists, there are many times when it is necessary to translate a string into the corresponding ASCII code, or vice versa. 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 uppercase "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$, and concatenated using CHR$(n), and the ASCII codes:

```
100 DATA 76,111,111,107,32,105,110,32,116,104
    ,101,32,104,111,108,108,111,119,32,
    115,116,117,109,112,32
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 which are difficult to read accidentally. Of course, any knowledgable programmer could pick the BASIC game apart, or enter PRINT H1$ from command mode once the program has been run past the initialization point. But, 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 140: Wait for user to press a key.

Line 150: Find ASCII value of that key.

Lines 160 to 170: Print result.

Line 210: Access the subroutine.

You Supply

Just press the key that you want to check.

RESULT

Variable A will equal CHR$ value of that key.

```
10 REM ***************
20 REM *             *
30 REM * CHR$ VALUE *
40 REM *             *
50 REM ***************
60 GOTO 180
70 REM ----------------------------
80 REM       ++ VARIABLES ++
90 REM     A: CHR$ VALUE OF KEY
100 REM
110 REM ----------------------------
120 GOTO 200

130 REM *** SUBROUTINE ***
```

```
140 GET A$:IF A$="" GOTO 130
150 A=ASC(A$)
160 PRINT"CHR$ VALUE OF KEY";A$
170 PRINT"IS :";A
180 RETURN

190 REM *** YOUR PROGRAM STARTS HERE ***

200 GOSUB 140
210 GOTO 200
```

LINE INPUT

WHAT IT DOES

Simulates LINE INPUT function found in other BASICs.
Allows entering string delimiters into strings.

Variables

- AN$: INPUT returned by user
- PROMPT$: Prompt character.

How To Use Subroutine

If you ask the user of a program to enter a string, and a comma or other character such as a quotation mark is entered, the extra 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 a GET A$ loop, and add any characters to AN$. Commas may be entered, and input can be ended only by pressing RETURN (CHR$(13)). 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 may be 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, which is especially useful for entries for which a question mark is inappropriate.

Line By Line Description

Line 60: Define the character to be used as a prompt.

Line 150: Print the prompt character to the screen.

Line 160: Wait for user input.

Line 170: Only if the key pressed was RETURN, then end the subroutine.

Line 180: Add key pressed to the answer string, AN$.

Line 190: Print the current key pressed to the screen.

Line 200: Go back for another entry.

Line 220: Access the subroutine.

Line 230: Print result, the user entry.

You Supply

- PROMPT$ may be redefined as any character you wish.

RESULT

Your finished INPUT will be stored in AN$.

```
10 REM **************
20 REM *              *
30 REM * LINE INPUT *
40 REM *              *
50 REM **************
60 PROMPT$="?"
70 GOTO 220
80 REM ----------------------------
90 REM       ++ VARIABLES ++
100 REM    AN$:       INPUT RETURNED
110 REM    PROMPT$:   PROMPT CHARACTER
120 REM
130 REM ----------------------------

140 REM *** SUBROUTINE ***

150 PRINT PROMPT$;
160 GET A$:IF A$="" GOTO 160
170 IF A$=CHR$(13) THEN RETURN
180 AN$=AN$+A$
190 PRINT A$;
200 GOTO 160

210 REM *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 150
230 PRINT
240 PRINT AN$
```

SWAP

WHAT IT DOES

Simulates SWAP function found in some other BASICs.

Variables

- A$: First variable
- B$: Second variable.

How To Use Subroutine

Exchanging the value of one variable for that of another cannot be done in one step in Commodore 2.0 BASIC found in the Commodore 64. This feature is useful in sorts and 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, either use a second identical subroutine, with the string identifiers removed (i.e., DUMMY = A, A = B, B = DUMMY). Or, a 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 130 to 140: Define initial value of A$ and B$.

Line 170: Temporarily assign A$ to a dummy variable, DUMMY$.

Line 180: Make A$ equal to B$.

Line 190: Make B$ equal to DUMMY$, which stores the original value of A$.

Line 220: Print original value.

Line 230: Access the subroutine.

Line 240: Print the results.

You Supply

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

RESULT

Values exchanged.

```
10 REM ********
20 REM *      *
30 REM * SWAP *
40 REM *      *
50 REM ********
60 REM ----------------------------
70 REM       + +  VARIABLES  + +
80 REM      A$:    FIRST VARIABLE
90 REM      B$:    SECOND VARIABLE
100 REM
110 REM ----------------------------

120 REM *** INITIALIZE ***

130 A$="FIRST"
140 B$="SECOND"
150 GOTO 220

160 REM *** SUBROUTINE ***

170 DUMMY$=A$
180 A$=B$
190 B$=DUMMY$
200 RETURN

210 REM *** YOUR PROGRAM STARTS HERE ***

220 PRINT "A$=";A$;"B$=";B$
230 GOSUB 170
240 PRINT "A$=";A$;"B$=";B$
```

STRING$

WHAT IT DOES

Simulates STRING$ function found in other BASICs.

Variables

- LTH: Desired length
- COMP$: Character or string used to assemble finished string
- STRNG$: The finished string.

How To Use Subroutine

You may wish to define a string as being composed of 40 spaces, to clear a line. Or, 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 140 to 150: Define the character to be used as the component string and the length of the desired string.

Line 180: Null any previous value of STRNG$.

Lines 190 to 210: Build new string of desired length, using COMP$.

Line 240: Access the subroutine.

Line 250: Assign value of STRNG$ to chosen variable.

Line 260: Print new variable.

You Supply

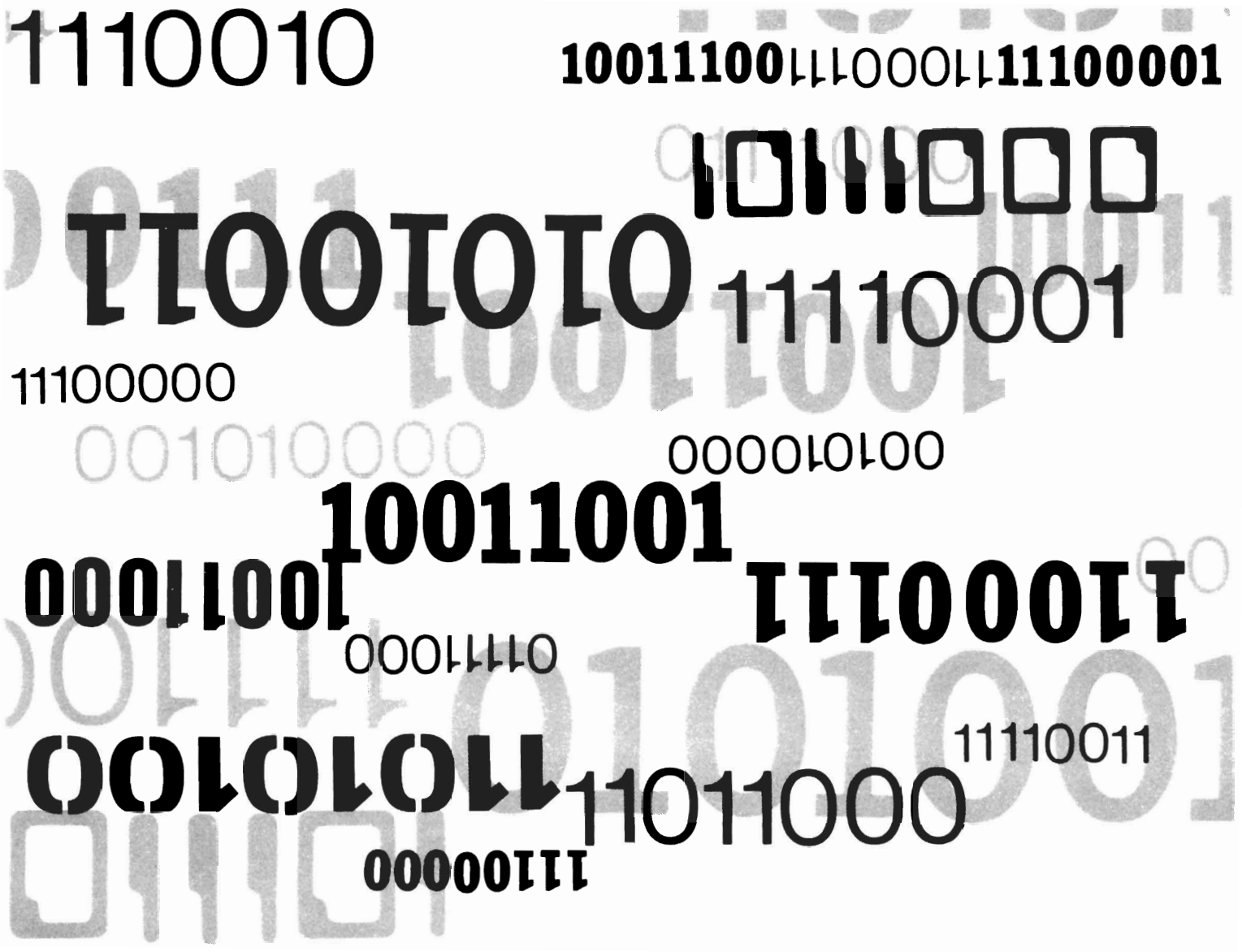
- Define LTH with the desired number of times the component string will be repeated. If the component string has more than one character, this length will NOT be the same as the length of the finished string.
- Supply a definition for COMP$, which may be either the actual characters or their CHR$ codes.

RESULT

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

```
10 REM ***********
20 REM *         *
30 REM * STRING$ *
40 REM *         *
50 REM ***********
60 REM ---------------------------
70 REM       ++ VARIABLES ++
80 REM      LTH:  DESIRED LENGTH
90 REM      COMP$: COMPONENT STRING
100 REM     STRNG$: FINISHED STRING
110 REM
120 REM ---------------------------

130 REM *** INITIALIZE ***

140 COMP$="A"
150 LTH=10
160 GOTO 240

170 REM *** SUBROUTINE ***
```

```
180 STRNG$ = " "
190 FOR N = 1 TO LTH
200 STRNG$ = STRNG$ + COMP$
210 NEXT N
220 RETURN

230 REM *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 180
250 B$ = STRNG$
260 PRINT B$
```


1110010
10011100
11100001
11110001
11100000
10011001
11011000
11110011
11100000

11 Bits and Bytes

This section is for those at the threshold of advanced programming. All but one of the routines in this part of the book deals 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 something like this:

10110111

In many cases, the value of this whole byte is of use to us. For example, in character memory, when we find a "1010001" (81 decimal), we know that the graphic "ball" character has been printed there. Using a full byte allows us 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. We 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 we want to change that condition, we may need to POKE ONLY that bit and leave the others, which pertain to other features, unchanged.

You'll remember in Chapter 5 that Boolean math is used to do certain bit-level operations. ANDing was introduced and used in the examples given for filtering out the "upper" four bits (or nybble) of a color memory byte. The OR and NOT operators were briefly mentioned as two of the other more frequently used Boolean tools. Where AND produced a 1 after each bit-to-bit comparison only when both bits are 1, OR will produce a 1 if either bit is one, while NOT produces the opposite of the value used.

For example:

```
Original byte:     10110110 OR
Comparison byte: 01100011
Result:            11110111
```

IF NOT A = 1 will produce a zero (false) value if A does equal 1. There are a number of other Boolean operators, including exclusive OR (XOR), but none of these are used in this book. What these subroutines let you do is manipulate individual bits, in order to set certain registers which 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 one, turning a feature "on." A third will set any bit to zero, 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 other 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 for you. Another routine, "Bit Displayer", will show the status of all the bits in a byte. In effect, it translates the byte into binary.

The final 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.

Variables

- ADDRESS: Location to PEEK
- BIT: Bit to examine
- V: Value of that bit, either 0 or 1.

How To Use Subroutine

The Commodore 64 gets maximum mileage 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, i.e., "IF V=0 THEN PRINT"THE FEATURE IS OFF."

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 64 keyboard.

Line By Line Description

Line 70: Define ADDRESS to PEEK.

Line 80: Define BIT to look at.

Line 180: Determine number to AND with byte.

Line 190: AND byte with P to determine status of the bit.

Line 220: Access subroutine.

Lines 230 to Line 240: Print results.

You Supply

Define:

- BIT as the bit
- 1-8 that you want to examine
- ADDRESS as the memory location to be PEEKed.

V will indicate whether the bit is on or off, by equaling either 1 or 0.

RESULT

Status of bit displayed.

```
10 REM ************
20 REM *          *
30 REM * PEEK BIT *
40 REM *          *
50 REM ************

60 REM *** INITIALIZE ***

70 ADDRESS=36879
80 BIT=3
90 GOTO 220
100 REM ----------------------------
110 REM       + + VARIABLES + +
120 REM   ADDRESS: LOCATION TO PEEK
130 REM   BIT:     BIT TO EXAMINE
140 REM   V:       VALUE OF BIT
150 REM
160 REM ----------------------------

170 REM *** SUBROUTINE ***

180 P=BIT-1
190 V=(PEEK(ADDRESS)AND(2 P))/(2 P)
200 RETURN

210 REM *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 180
230 PRINT CHR$(147)
240 PRINT V
```

BIT DISPLAYER

WHAT IT DOES

Shows pattern of all eight bits within a byte.
Converts the decimal value to binary.

Variables

- ADDRESS: Location to POKE
- BIT$: Bit pattern.

How To Use Subroutine

This subroutine will display all of the bits within a byte. Each position will be indicated by a one or a zero.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 64 keyboard. 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, e.g., protection purposes.

This subroutine will also serve as a means of converting decimal numbers smaller than 255 to binary. Simply substitute your variable for PEEK(ADDRESS), and define the variable as the decimal number you want to convert.

Line By Line Description

Line 70: Define address to be PEEKed.

Line 170: Null any previous value of BYTE$.

Line 180: Provide TAB to print result.

Lines 190 to 230: Repeat through each bit of byte AND each bit with the next highest power of two, and store the result in G$, which will store the on/off status of each bit. Then, add G$ to BYTE$.

Line 270: Access subroutine.

Line 280: Print result.

You Supply

- 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.

RESULT

All bits within a byte are displayed.

```
10 REM ******************
20 REM *                 *
30 REM * BIT DISPLAYER *
40 REM *                 *
50 REM ******************

60 REM *** INITIALIZE ***

70 ADDRESS=36879
80 GOTO 260
90 REM ----------------------------
100 REM       + + VARIABLES + +
110 REM     ADDRESS: MEMORY BYTE
120 REM              TO DISPLAY
130 REM     BIT$:    BIT PATTERN
140 REM
150 REM ----------------------------
```

```
160 REM *** SUBROUTINE ***

170 BIT$=" "
180 PRINTTAB(4)" ";
190 FOR N=7 TO 0 STEP-1
200 V=(PEEK(ADDRESS)AND(2 N))/(2 N)
210 G$=MID$(STR$(V),2)
220 BIT$=BIT$+G$
230 NEXT N
240 RETURN

250 REM *** YOUR PROGRAM STARTS HERE ***

260 PRINT
270 GOSUB 170
280 PRINT"ADDRESS: ";ADDRESS
290 PRINT PEEK(ADDRESS);" ="
300 PRINT TAB(4)BIT$
```

BIT TO ONE

WHAT IT DOES

Soft POKEs any desired bit within a byte so that it now has the value of one, without changing any other bits.

Variables

- ADDRESS: Location to POKE
- BIT$: Bit to change to one.

How To Use Subroutine

This subroutine will take any bit within a byte and change that value to one, 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 64.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 64 keyboard.

Line By Line Description

Line 70: Define ADDRESS to PEEK and POKE.

Line 80: Define BIT to change to a value of 1.

Line 170: POKE BIT to one.

You Supply

- 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 one.

RESULT

Bit within a byte is changed to one.

```
10 REM ***************
20 REM *             *
30 REM * BIT TO ONE *
40 REM *             *
50 REM ***************

60 REM *** INITIALIZE ***

70 ADDRESS=36878
80 BIT=3
90 GOTO 200
100 REM ----------------------------
110 REM       + + VARIABLES + +
120 REM   ADDRESS: LOCATION TO POKE
130 REM   BIT:     BIT TO CHANGE TO ONE
140 REM
150 REM ----------------------------

160 REM *** SUBROUTINE ***

170 POKE ADDRESS,PEEK(ADDRESS)OR(2 BIT)
180 RETURN

190 REM *** YOUR PROGRAM STARTS HERE ***

200 PRINT
210 GOSUB 170
```

BIT TO ZERO

WHAT IT DOES

Soft POKEs any desired bit within a byte so that it now has the value of zero, without changing any other bits.

Variables

- ADDRESS: Location to POKE
- BIT$: Bit to change to zero.

How To Use Subroutine

This subroutine will take any bit within a byte, and change that value to zero, 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 64.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 64 keyboard.

Line By Line Description

Line 70: Define ADDRESS to PEEK and POKE.

Line 80: Define BIT to change to a value of 0.

Line 170: POKE BIT to one.

You Supply

- 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 zero.

RESULT

Bit within a byte is changed to zero.

```
10 REM ***************
20 REM *             *
30 REM * BIT TO ZERO *
40 REM *             *
50 REM ***************

60 REM *** INITIALIZE ***

70 ADDRESS=36879
80 BIT=3
90 GOTO 200
100 REM ----------------------------
110 REM       + + VARIABLES + +
120 REM   ADDRESS: LOCATION TO POKE
130 REM   BIT:     BIT TO CHANGE TO ZERO
140 REM
150 REM ----------------------------
```

```
160 REM *** SUBROUTINE ***

170 POKE ADDRESS,PEEK(ADDRESS)AND(255-(2 BIT))
180 RETURN

190 REM *** YOUR PROGRAM STARTS HERE ***

200 PRINT
210 GOSUB 170
```

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.

Variables

- ADDRESS: Location to POKE
- BIT: Bit to reverse.

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 one, it will be changed to zero. A zero bit will be given a value of one. 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 64. Using this subroutine, it is not necessary to know whether the feature is on or off. "Reverse Bit" will change it to the other status automatically.

NOTE: The caret symbol (^) indicates the up-arrow key, located between RESTORE and "*" on your Commodore 64 keyboard.

Line By Line Description

Line 70: Define ADDRESS to PEEK and POKE.

Line 80: Define BIT to reverse.

Lines 170 to 180: Find out value of the bit, then reverse that, using OR.

You Supply

- 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.

RESULT

Bit within a byte is reversed.

```
10 REM ***************
20 REM *             *
30 REM * REVERSE BIT *
40 REM *             *
50 REM ***************

60 REM *** INITIALIZE ***

70 ADDRESS=36878
80 BIT=3
90 GOTO 210
100 REM ----------------------------
110 REM       + + VARIABLES + +
120 REM   ADDRESS: LOCATION TO POKE
130 REM   BIT:     BIT TO REVERSE
140 REM
150 REM ----------------------------

160 REM *** SUBROUTINE ***

170 M=1-(PEEK(ADDRESS)AND(2 BIT))/(2 BIT)
180 POKE ADDRESS,PEEK(ADDRESS)AND(255-(2 BIT))OR(M*(2 BIT))
190 RETURN

200 REM *** YOUR PROGRAM STARTS HERE ***

210 PRINT
220 GOSUB 170
```

BINARY TO DECIMAL

WHAT IT DOES

Changes binary number to decimal equivalent.

Variables

- A$: Binary number in string form
- A: Decimal equivalent.

How To Use Subroutine

Several of the subroutines in this book, and many more that you will prepare, will 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 64 keyboard.

Line By Line Description

Lines 150 to 160: Look at each binary character, and raise any 1's to the power of 2 indicated by its position within the byte.

Lines 200 to 210: Ask user for binary number to convert.

Lines 220 to 270: Check for presence of illegal characters.

Line 280: Access subroutine.

Line 290: Print result.

You Supply

- Enter the binary number to be converted.

RESULT
Binary number converted to decimal.

```
10 REM ******************
20 REM *                 *
30 REM * BINARY/DECIMAL *
40 REM *                 *
50 REM ******************

60 REM *** INITIALIZE ***

70 REM GOTO 200
80 REM --------------------------------
90 REM       + + VARIABLES + +
100 REM   A$ : Binary number in string form.
110 REM   A:   Decimal equivalent
120 REM
130 REM ----------------------------
```

```
140 REM *** SUBROUTINE ***

150 FOR N=1 TO LEN(A$)
160 A=A+2^VAL(MID$(A$,N,1)
170 NEXT N
180 RETURN

190 REM *** YOUR PROGRAM STARTS HERE ***

200 PRINT CHR$(147)
210 INPUT "ENTER NUMBER TO CONVERT: ";A$
220 FOR N=1 TO LEN(A$)
230 T$=MID$(A$,N,1)
240 IF T$="0" OR T$="1" GOTO 280
250 PRINT "NOT BINARY NUMBER"
260 PRINT CHR$(17)
270 GOTO 210
280 GOSUB 150
290 PRINT A$"=";A
```

ROUNDER

WHAT IT DOES

Rounds positive number, and cuts off after desired number of decimal places.

Variables

- A: Number to be rounded
- P: Digits desired to right of decimal point
- B: Rounded value.

How To Use Subroutine

The Commodore 64 is sometimes a great deal more accurate than we need. For example, our car may get 24.3459121 miles per gallon, but we 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 64 keyboard.

Line By Line Description

Line 150: Define number to be rounded.

Line 160: Define number of digits to right of decimal desired.

Line 190: Add rounding factor.

Line 200: Take integer portion of number multiplied by 10 raised to P power, and divide that by 10 raised to P power.

Line 230: Access subroutine.

Line 240: Print result.

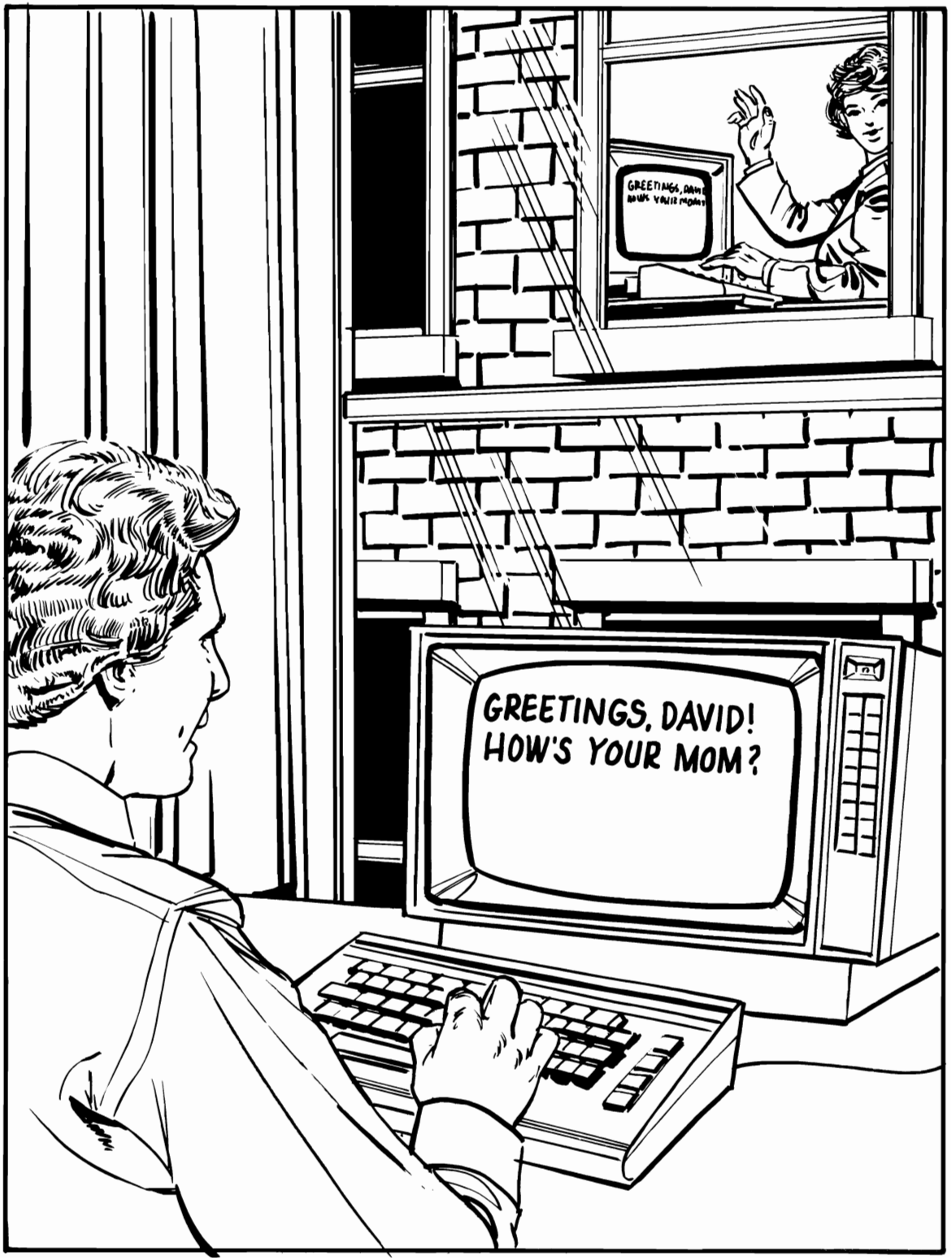
You Supply

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 zero, the zero 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.

RESULT

Number rounded as specified.

```
10 REM ***********
20 REM *         *
30 REM * ROUNDER *
40 REM *         *
50 REM ***********
60 REM ------------------------------
70 REM       + + VARIABLES + +
80 REM     A:   NUMBER TO BE ROUNDED
90 REM     P:   DIGITS DESIRED TO
100 REM         RIGHT OF DECIMAL POINT
110 REM    B:   ROUNDED VALUE
120 REM
130 REM ------------------------------

140 REM *** INITIALIZE ***

150 A=55.534
160 P=2
170 GOTO 230

180 REM *** SUBROUTINE ***

190 C=A+5.5*10^-(P+1)
200 B=INT(C*10^P)/10^P
210 RETURN

220 REM *** YOUR PROGRAM STARTS HERE ***

230 GOSUB 190
240 PRINT B
```


GREETINGS, DAVID!
HOW'S YOUR MOM?

12 Communications and Other Tips

Communications is rapidly becoming one of the most popular applications for computers like the Commodore 64. The machine needs only to be coupled with a low-cost device called a modem. A modem converts the computer's signals into sounds that can be transmitted over the telephone wires. In addition, some sort of communications software needs to be written to allow the Commodore 64 to talk to the other computers.

BASIC is generally too slow to communicate in such a manner, because the data must be sent out in serial fashion, a byte at a time, and received the same way. It is easy enough to send data one way, but when the Commodore 64 tries to talk and listen at the same time, the speed of the BASIC interpreter becomes a limiting factor. However, there are several applications which can use BASIC quite handily, and this section has two subroutines that allow you to experiment with both of them.

The first is a quick way of sending Commodore 64 programs from your computer to another in the same room. You can hard-wire the two together through a null modem adapter (cost: about $30), and an accessory RS232 adapter for your Commodore 64. The other computer also needs to be equipped with an RS232 serial port of its own. This form of communications is one-way only, but allows sharing your BASIC programs with users of other computer systems, particularly those with similar BASICs. The Apple and Radio Shack line both have many similarities with Commodore Basic 2.0, and the author has successfully transferred many programs to each. Some modifications have to be made to make the programs compatible. But, editing out special graphics characters, screen formatting characters, etc. before the program transfer will help a great deal.

The second subroutine—actually a full-fledged program—in this section is a simple BASIC terminal routine. You cannot upload or download text and are limited to 300 baud, but true two-way communications is possible. If you have more advanced requirements, check into one of the many faster machine-language terminal programs available for the Commodore 64.

PROGRAM TRANSFER

WHAT IT DOES

Allows sending program listing out RS232 interface.

Variables

None.

How To Use Subroutine

Because the Commodore 64's BASIC is fairly compatible with that of other computers, it may be desirable to send a program listing to another machine. If a Commodore 64 is equipped with an inexpensive RS-232 interface (about $40.00) and the other computer has the same, a cable, a null modem adapter, and this subroutine are nearly all that is needed. The other computer should have a terminal program that will allow dumping the transmitted file to disk or tape.

Be sure and include the null modem adapter (available from most computer stores) when communicating 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. In truth, the SEND/RECEIVE lines need to be matched. A null modem will do this.

Simply load the program you want to transmit, and either MERGE this subroutine or type in the characters from the keyboard. You should type them all in before hitting RETURN. Baud rate will be set for 300; your receiving computer should be set likewise.

Line By Line Description

Line 130: OPEN RS232, set for 300 Baud.

Line 140: Redirect screen output to RS232 instead.

Line 150: List desired program lines.

Lines 160 to 170: CLOSE RS232 channel.

You Supply

- Program to upload.

RESULT

Program listing is transmitted out RS-232 interface.

```
10 REM ********************
20 REM *                  *
30 REM * PROGRAM TRANSFER *
40 REM *                  *
50 REM ********************
60 GOTO 130
70 REM ---------------------------
80 REM       + + VARIABLES + +
90 REM          NONE
100 REM
110 REM ---------------------------

120 REM *** SUBROUTINE ***

130 OPEN 2,2,3,CHR$(38)+CHR$(160)
140 CMD2
150 LIST
160 PRINT#2
170 CLOSE 2
180 END
```

TERMINAL

WHAT IT DOES

BASIC terminal program.

Variables

- B$: Character sent
- C$: Character received.

How To Use Subroutine

BASIC is just about fast enough to allow the Commodore 64 to communicate at 300 baud if nothing fancy is attempted. This subroutine will fetch characters from the RS232 line, and send keyboard characters out that serial interface. Anything sent or received will be echoed to the screen.

Line By Line Description

Line 130: OPEN RS232 channel for 300 baud communications.

Line 140: GET from channel #2, a character, if available.

Line 150: GET a character from the keyboard, if a key is depressed.

Line 160: If a key was depressed, print that character to the screen.

Line 170: If user enters CHR$(95) (back arrow), stop communications.

Line 180: Get character from RS232, if available.

Line 190: Print keyboard or character from RS232 to screen.

Line 200: Go back and get more characters.

Line 210: CLOSE the RS232 channel.

You Supply

- No user changes needed.

RESULT

Dumb terminal communications using BASIC.

```
10 REM ************
20 REM *          *
30 REM * TERMINAL *
40 REM *          *
50 REM ************
60 REM ----------------------------
70 REM       + + VARIABLES + +
80 REM    B$:  CHARACTER SENT
90 REM    C$:  CHARACTER RECEIVED
100 REM
110 REM ----------------------------

120 REM *** SUBROUTINE ***

130 OPEN2,2,3,CHR$(38)+CHR$(160)
140 GET#2,A$
150 GET B$
160 IF B$<>"" THEN PRINT#2,B$;
170 IF B$=CHR$(95) THEN GOTO 210
180 GET#2,C$
190 PRINT B$;C$;
200 GOTO 150
210 CLOSE 2

220 REM *** YOUR PROGRAM STARTS HERE ***

230 PRINT
```

Glossary

Algorithm: A formula or method for performing a given task, such as MPG = MILES/GALLNS.

Alphanumeric: Characters that are letters or numbers, as opposed to graphics or control characters. Alphanumerics include the upper and lowercase alphabet, as well as the digits 0 to 9.

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.

Array: A method of storing information in the computer's memory. An array can have one dimension, as A$(n), with each element (or "compartment") in the array storing one piece of information. Arrays may also have more than one dimension, e.g., A$(row,col), and store rows and columns of information. Multidimensional arrays are like tables with horizontal and vertical slots.

Arrays may also be either of the numeric or string type. With a numeric array, each element can store one number; a string array can accommodate a single string per compartment but that string can be up to 255 bytes long and, therefore, contain more than one character.

ASCII: American Standard Code for the Interchange of Information. A common code used by most computer systems for storage of information, especially text files. It provides a basis for sharing files between unlike computers. Commodore computers use a modified form of this code, which must be converted to standard ASCII when communicating to other systems.

Binary: The base-two number system used by computers, which consists of 1's and 0's only.

Bit: The smallest unit of information that can be processed by the Commodore 64; short for binary digit. A bit represents either 1 or 0, with eight bits making up a single byte.

Boolean math: A type of algebra named for George Boole, which uses two-valued variables (on/off,true/false) suitable for use by binary computers like the Commodore 64. Certain boolean operators, such as AND and OR, are used with many subroutines to examine memory registers on the bit level.

Channel: A line of communication used by the Commodore 64 to send or receive information.

Character set: A table listing "definitions" of the characters used by the Commodore 64. By changing the definition, the user can make the computer display a different, customized character in place of the standard ones provided.

Color memory: A set of RAM locations that keep track of what color is being displayed in any corresponding screen location.

Cursor: The block character, or any other character,used to mark the current printing position on the screen.

Decimal: Base-10 numbers; the commonly used number system. The Commodore 64 asks for decimal numbers, and returns decimal numbers for PEEK and POKE operations, even though it processes them in binary form internally.

Decrement: To decrease a variable by one. However, the word is also commonly used as a verb when the number is being decreased by some larger amount, as to "decrement by two."

Default: Any value used automatically if no other value is supplied by the program or user.

Download: To capture a file through telecommunications into the Commodore 64's memory buffer, and then write it to tape or disk for permanent storage. Programs or text files can be transmitted to your computer through a modem, and then downloaded.

File: Any collection of information on disk or tape. Basic and machine-language programs, as well as text material, are all files. The Commodore 64 treats program files differently than other files, as they are sent to certain specific memory locations with the correct links so that they can be RUN.

Format: In disk-drive usage, to make a new, unused disk ready for data storage. This is done using the NEW command with the Commodore 64. Certain information is placed on the disk marking the available sectors and readying a directory to store a listing of the SAVEd programs.

Function key: The row of four keys at the extreme right of the Commodore 64 keyboard, which can be used as additional keys to direct control to subroutines or functions of the programmer's choice.

Increment: To increase the value of a variable by one. This is also commonly used as a verb to denote increasing a variable by any amount, such as "to increment by four."

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 zero again.

Jiffie: A 1/60th second interval used by the Commodore 64 to keep track of elapsed time.

Garbage: Random information with no meaning. Every memory location contains something. If it is not meaningful information placed there by the computer or user, it is termed garbage.

Merge: To combine two programs in such a way that their line numbers become mixed. While MERGING may produce interleaved programs, if there are duplicate line numbers, the program added will write over the same lines in the original program.

Modem: Modulator-demodulator. A device that converts the Commodore 64's signals to sounds that can be transmitted over telephone lines. The modem also receives sounds and converts them back for the Commodore 64 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 30000,X
```

```
10 OFFSET=30000
20 POKE OFFSET+1,X
```

The second method is often more clear and can also be used when the memory location defined by OFFSET can vary, as the Commodore 64's color memory changes depending on the amount of RAM installed.

OR: A Boolean operator that is used to compare one byte with another on a bit for bit level. If a bit and the corresponding bit in the other byte are either 1, OR will produce a 1 as the result.

Oscillator: An electronic device that, in the Commodore 64, 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, transmit each of the eight bits one at a time.

Port: One of the "windows" used by the Commodore 64 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 which appear to be random but which 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, which 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 64 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 64's microprocessor and can be accessed only through machine language. Some memory locations in the Commodore 64 perform a register-like function, telling the computer whether a certain feature is ON or OFF, or the volume of a sound oscillator, or some other status.

Rheostat: A variable resistor, like those used in paddles, which lets more electricity flow when turned one way and less when turned the other.

RS-232: A serial interface device that allows the Commodore 64 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 lightswitch.

Upload: To store a file from disk or tape in the Commodore 64'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 four oscillators in the Commodore 64 that produce sounds. Three have overlapping music ranges, while the fourth voice produces "noise."

Index

Whet Your Programming Appetite With a "Cookbook" Approach . . .
Write Quality Programs on the Commodore 64!

Acclaim from prepublication reviewers:

". . . this book offers excellent line-by-line descriptions for each subroutine presented . . . a great aid for beginners who know **BASIC** and for experienced users alike!

". . . very, very accurate . . . here's an author who knows his subject thoroughly!"

COMMODORE 64 SUBROUTINE COOKBOOK

David D. Busch

Here's a programming "cookbook" that offers a potpourri of machine-specific subroutines designed to help improve your programming expertise on the Commodore 64! This unique programming guide includes 70 ready-to-merge subroutines plus programming tips to make your own programs sizzle! These easy-to-follow subroutines are ingredients to "recipes" for your programming proficiency, designed to take the mystery out of designing character sets, using function keys, joystick action, sound, and other special features of the Commodore 64. No more "stewing" over exotic, top-heavy math functions and statistics! Complete with line-by-line descriptions of each subroutine presented, this book also includes:

- Subroutines for generating musical notes or adding sound effects within your own programming!
- Subroutines for business/financial users and advanced programmers as well!
- Subroutines for adding your own special characters when the Commodore 64's are not enough for your needs!
- Tips on routines to make your games of arcade quality!
- Plus—a comprehensive glossary and index!

CONTENTS

ISBN 0-89303-383-9