TRS-80 Portable Computer Subroutine Cookbook

David D. Busch

TRS-80 Portable Computer Subroutine Cookbook

TRS-80 Portable Computer Subroutine Cookbook

DAVID D. BUSCH

Brady Communications Company, Inc., Bowie, Maryland 20715
A Prentice-Hall Publishing Company

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

Typesetter: PageCrafters, inc. Oswego, New York
Printer: R. R. Donnelley & Sons Co., Harrisonburg, Virginia
Typefaces: Novarese (text); Cheltenham (display)

TRS-80 Portable Computer Subroutine Cookbook

Library of Congress Cataloging in Publication Data

Busch, David D.
TRS-80 portable computer subroutine cookbook.
Includes index.
1. TRS-80 Portable Computer (Computer)—Programming. 2. Subroutines (Computer programs)
I. Title. II. Title: TRS-80 portable computer subroutine cookbook.
QA76.8.T184B87 1984 001.64'2 84-11135

ISBN 0-89303-904-7

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

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 simulate joystick action or resound with music. These are TRS-80 Portable Computer specific routines that take the mystery out of using function keys, the built-in clock, interrupt routines, and other special TRS-80 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 TRS-80 Portable Computer, while it is a powerful, capable microcomputer, is being sold to a broad range of users. Some only want to use programs for business. 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 TRS-80 *Portable Computer 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 task for the TRS-80 Portable Computer alone.

Nine subroutines are included that add crucial BASIC functions that were left out of the already feature-packed BASIC ROM. SWAP variable values; insert a larger string within another. Do global search and replace functions.

Interested in using the arrow keys as pseudo-joysticks to manipulate objects on the screen? Just transplant one of the joystick routines included in this book. We even show you how to move your missiles and enemy aliens around on the screen.

Using the TRS-80 Portable Computer'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 the TRS-80 to your heart's content. One subroutine will show you how to invoke reverse character sets.

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 TRS-80 Portable Computer 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.

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

This is not a first programming book. You should 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.

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 A$=INKEY$: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 several joystick routines. Some of the concepts are explained only once. At times, you will be directed to look at previous subroutines for longer explanations. This allows you to access the routines in any order, without reading the entire book.

All the odd special TRS-80 characters have been left out of this book. If a graphics symbol is used in a subroutine, the CHR$ code is substituted instead.

Both the graphics symbol and CHR$ code will perform the proper function. Deleting the extra graphics makes the subroutines shorter and easier to understand. 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 TRS-80 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.

Good luck. You should find this book a short-cut 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.

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 tape, in RAM, or, when available, on disk or some other mass storage medium, 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. The TRS-80 Portable Computer has a powerful, simple, merge command that allows merging program lines and subroutines.

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 because 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 the the program and the routine to be merged have no program line numbers in common. The way to do this is to purposely leave 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 users, the append type of merge is the safest and easiest to implement.

To MERGE with the Portable Computer, go to BASIC, and load the module that you wish to add to memory. The original program lines can be either a .BA or .DO file. The subroutine or program that you wish to MERGE must be stored as a .DO file. You might have created it using TEXT. Or you can load any .BA file and store it as a .DO file by using the .DO extension when doing the SAVE.

To MERGE just type:

```
MERGE"filename.do"
```

That's all there is to it. The TRS-80 Portable Computer assumes that you want to MERGE a RAMfile, and does the work for you. Of course, like all of the Computers 100's I/O-commands, you can also MERGE from tape, COM, or MDM, just by prefacing the filename with "CAS:", "COM:" or "MDM:" (e.g., MERGE"COM:filename.do").

Unlike some other computers, this model makes it easy for you to maintain a large, useful, subroutine library.

Sample Merging Run:

Subroutine (lines 10-250)

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

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

```
140 ' *** INITIALIZE ***

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

200 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***
```

Program merged with subroutine:

```
260 CLS:
270 PRINT "COMPOUND INTEREST"
280 PRINT CHR$(17)
290 INPUT "INTEREST RATE";RATE
300 INPUT "YEARS TO BE COMPOUNDED";YEARS
310 INPUT "AMOUNT TO BE COMPOUNDED";AMOUNT
320 INPUT "COMPOUNDING PERIODS/YR";PERIODS
330 GOSUB 210
340 PRINT "FUTURE VALUE:";FUTURE
350 PRINT "DO IT AGAIN?"
360 PRINT TAB(4)"Y/N?"
370 A$=INKEY$:IF A$="" GOTO 370
380 IF A$="Y" OR A$="y" THEN GOTO 260
390 IF A$="N" OR A$="n" THEN END
400 GOTO 390
```

Limits of Liability and Disclaimer of Warranty

Trademarks of Material Mentioned in this Text

Chapter One

Simulating Joysticks and Paddles

Let's start off by adding a new capability to your Micro Executive Workstation—the simulation of joysticks and paddles. Though these devices are now most frequently seen in arcade-style shoot-em-ups, they have many applications in business, engineering, and science. Joysticks, in particular, make an excellent input device whenever it is necessary to position the cursor on the screen with some accuracy.

While the TRS-80 doesn't pretend to be a computer-assisted drafting tool, its lack of a joystick control is a drawback that can be remedied. This chapter contains four subroutines that allow you to use the arrow keys to simulate a joystick's directional functions. Your program can also use any key of your choice (such as the SPACE bar) to simulate the FIRE button of joysticks and paddles.

From a programmer's standpoint, the use of the simulated joystick breaks down into several neat modules. We need to know where on the TRS-80'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 arrow keys as a controller.

The different routines have been included in this section to allow several different types of object movement in your programs. The first joystick subroutine in this chapter allows moving an object only in north, south, east, or west directions. The next routine permits movement only left or right, like a paddle, but allows you to pick which row your object will reside in. The third subroutine of the group performs a similar paddle-type unidirectional movement but vertically.

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 a non-arrow key changes the cursor character to that of the key pressed.

Joystick Subroutine

WHAT IT DOES

Moves object north, south, east and west using arrow keys as joystick.

Variables

- E: End of screen
- B1: Current position of object

- MOVE. Direction of move for object, to be used to increment B1
- F1: Status of FIRE button (SPACE bar).

How To Use The Subroutine

The arrow keys can be used to simulate the north, south, east, and west movements of a joystick. Your games and applications can use the status of this 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 arrow key joystick 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.

Call the subroutine whenever you wish to check on the status of the joysticks. The value of MOVE can be used with B1 (the current position of the object) to move the object or to direct some other program action. The value of F1, the FIRE button of the 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 CLS:
```

This subroutine prints an asterisk character on the screen. You may change the character by substituting some other character for the asterisk.

The routine leaves a "trail" of the character behind it. You can erase this by PRINT @ B1-MOVE with CHR$(32).

Variables Not To Be Changed By User

- E: End of screen. You will not want your object to move beyond this value or else it will be "off" the screen and will crash your program!
- B: Beginning of screen. Same as above, only for start of screen PRINT @ positions.

Line-By-Line Description

Line 70: Define beginning and end of screen.

Line 250: Wait for user to press key.

Line 260: Determine ASC value of key pressed. If it was SPACE bar, then change FIRE variable, F1, to 1.

Lines 270 to 300: Depending on which arrow key was pressed, change value of the move, MOVE, to −40 (up), 40 (down), 1 (right) or −1, (left).

Line 330: Clear screen.

Line 340: Access the subroutine.

Line 350: Change position of cursor, B1, to account for move, MOVE. If new position is beyond screen limits, change back to former value.

Line 360: Print asterisk at B1.

You Supply

B1: Current position of object. This will be a number between B and E. Your program should always check to make sure that B1 is never less than B or greater than E. You may want to define B1=B 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 and then PRINTing @ B1 with the CHR$ of the character you want. Erase B1 from its old position by PRINTing @, its former value, with CHR$(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.

RESULT

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

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

60 ' *** INITIALIZE ***

70 B=0:E=318
80 GOTO 330
90 ' -------------------------------------
100 '      ++ VARIABLES ++
170 '     MOVE: MOVE FOR JOYSTICK
190 '     B1:   POSITION FOR OBJECT
200 '     B:    BEGINNING OF SCREEN
210 '     E:    END OF SCREEN
220 '     F1:   STATUS OF FIRE (SPACE BAR)
230 ' ------------------------------------
```

```
240 ' *** SUBROUTINE ***

250 A$=INKEY$:IF A$="" GOTO 250
260 J1=ASC(A$):IF J1=32 THEN F1=1:RETURN ELSE F1=0
270 IF J1=30 THEN MOVE=-40:RETURN
280 IF J1=31 THEN MOVE=40:RETURN
290 IF J1=29 THEN MOVE=-1:RETURN
300 IF J1=28 THEN MOVE=1:RETURN
310 RETURN

320 ' *** YOUR PROGRAM STARTS HERE ***

330 CLS
340 GOSUB 250
350 B1=B1+MOVE:IF B1<B OR B1>E THEN B1=B1-MOVE
360 PRINT @ B1,"*";
370 GOTO 340
```

Horizontal Paddle Simulation

WHAT IT DOES

Moves object left and right only.

Variables

- ROW: Row to move object in
- B: Beginning of that column
- E: End of that column
- B1: Position of object
- CURSR: Cursor character
- MOVE: Direction of move
- F1: Status of FIRE button.

How To Use The Subroutine

The Micro Executive Workstation currently has no provision for attaching paddles. However, their movement from side to side, as used in breakout-type games, can be simulated by using just the arrow keys.

In any case, some games work better when the arrow keys are used for movement from side to side. This approach makes game programming simpler. Move an object one column to the right when the right arrow is pressed and move one row left when the left arrow is pressed.

This subroutine will tell you whether or not either case has occurred, plus it will 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 one bottom of the screen in a position defined as, say, B1. This will be increased or decreased by 1 each time an arrow key is pressed from side to side. Make sure that B1 never exceeds the end of the screen when PRINTing objects to the screen. A more complete discussion of moving objects on the screen is included with the previous subroutines.

Line-By-Line Description

Line 180: Define an array capable of keeping track of four possible values for movement, even though only two directions are used in this subroutine. The module represents a second, different technique for passing the value of the arrow key pressed on to MOVE and can be adapted for four-directional play.

Line 190: Define DATA for direction of movement when joystick is pressed in any of the four possible values. In this case, only two are used, to provide for movement from side to side.

Lines 200 to 220: Read move data into array.

Line 240: Define ROW to move cursor in.

Line 250: Determine starting position of cursor.

Line 260: Calculate end PRINT position of that column.

Line 270: Define beginning PRINT position of the column.

Line 280: Define cursor character. User may change.

Line 310: Wait for user to press key.

Line 320: Determine ASCII value of key pressed, and see if FIRE button (SPACE bar) was pressed.

Line 330: If arrow key not pressed, MOVE=0.

Line 340: MOVE equals J1 element of array.

Line 370: Access the subroutine.

Line 380: If MOVE is not zero, erase old character.

Line 390: Update value of B1 by adding MOVE.

Line 400: If new value of B1 is outside COL, change back.

Line 410: PRINT cursor character at B1.

Line 420: Repeat.

You Supply

Just move joystick. COL can be defined as the screen column on which the object moves. CURSR can be defined as any character you wish.

RESULT

Object will move on screen under joystick control, side to side only.

```
10 ' *****************
20 ' *               *
30 ' *  MOVE OBJECT  *
40 ' * LEFT OR RIGHT *
50 ' *               *
60 ' *****************
70 '  --------------------------------
80 '    ++ VARIABLES ++
90 '   ROW:   ROW TO MOVE IN
100 '  B:     BEGINNING OF THAT ROW
110 '  E:     END OF THAT ROW
120 '  B1:    POSITION OF CURSOR
130 '  CURSR: CURSOR CHARACTER
140 '  MOVE:  DIRECTION OF MOVE
150 '  F1:    STATUS OF FIRE (SPACE BAR)
160 ' --------------------------------

170 ' *** INITIALIZE ***

180 DIM MOVE(4)
190 DATA 1,-1,0,0
200 FOR N=1 TO 4
210 READ MOVE(N)
220 NEXT N
230 CHAR=0
240 ROW=5
250 B1=CHAR+ROW*40
260 E=B1+39
270 B=B1
280 CURSR=43
290 GOTO 370

300 ' *** SUBROUTINE ***

310 A$=INKEY$:IF A$="" GOTO 310
320 J1=ASC(A$):IF J1=32 THEN F1=1:RETURN ELSE F1=0
330 IF J1<28 OR J1>31 THEN MOVE=0:RETURN
340 MOVE=MOVE(J1-27)
350 RETURN
```

```
360 ' *** YOUR PROGRAM STARTS HERE ***

370 GOSUB 310
380 IF MOVE< >0 THEN PRINT @ B1,CHR$(32);
390 B1=B1+MOVE
400 IF B1<B OR B1>E THEN B1=B1-MOVE
410 PRINT @ B1,CHR$(CURSR);
410 GOTO 370
```

Vertical Paddle Simulation

WHAT IT DOES

Moves object up and down only.

Variables

- COL: Row to move object in
- B: Beginning of that column
- E: End of that column
- B1: Position of object
- CURSR: Cursor character
- MOVE: Direction of move
- F1: Status of FIRE button.

How To Use The Subroutine

The TRS-80 currently has no provision for attaching paddles. However, their movement up and down, as used in PONG-type games, can be simulated by using just the up-down arrow keys.

In any case, some games work better when the arrow keys are used for movement up and down. This approach makes game programming simpler. Move an object up one row when the up arrow is pressed and move one row down when the down arrow is pressed.

This subroutine will tell you whether or not either case has occurred, plus it will 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 one side of the screen in a position defined as, say, B1. This will be increased or decreased by 40 each time an arrow key is pressed up or down. Make sure that B1 never exceeds the end of the screen when PRINTing objects to the screen. A more complete discussion of moving objects on the screen is included with the previous subroutines.

Line-By-Line Description

Line 180: Define an array capable of keeping track of four possible values for movement, even though only two directions are used in this subroutine. The module represents a second, different technique for passing the value of the arrow key pressed on to MOVE and can be adapted for four-directional play.

Line 190: Define DATA for direction of movement when joystick is pressed in any of the four possible values. In this case, only two are used, to provide for movement up and down.

Lines 200 to 220: Read MOVE data into array. Since the first two elements are both 0, no movement will occur if the right or left arrows are pressed. By substituting −1 and +1 for the zeroes in the data line, four-way control is restored.

Line 240: Define COL to move cursor in.

Line 250: Determine starting position of cursor.

Line 260: Calculate end PRINT position of that column.

Line 270: Define beginning PRINT position of the column.

Line 280: Define cursor character. User may change.

Line 310: Wait for user to press key.

Line 320: Determine ASCII value of key pressed, and see if FIRE button (SPACE bar) was pressed.

Line 330: If arrow key not pressed, MOVE=0.

Line 340: MOVE equals J1 element of array.

Line 370: Access the subroutine.

Line 380: If MOVE is not zero, erase old character.

Line 390: Update value of B1 by adding MOVE.

Line 400: If new value of B1 is outside COL, change back.

Line 410: PRINT cursor character at B1.

Line 420: Repeat.

You Supply

Just move joystick. COL can be defined as the screen column on which the object moves. CURSR can be defined as any character you wish.

RESULT

Object will move on screen under joystick control, up or down only.

```
10 ' *****************
20 ' *               *
30 ' *  MOVE OBJECT  *
40 ' *  UP AND DOWN  *
50 ' *               *
60 ' *****************
70 '  ---------------------------------
80 '   ++ VARIABLES ++
90 '   COL:   COLUMN TO MOVE IN
100 '  B:     BEGINNING OF THAT COLUMN
110 '  E:     END OF THAT COLUMN
120 '  B1:    POSITION OF CURSOR
130 '  CURSR: CURSOR CHARACTER
140 '  MOVE:  DIRECTION OF MOVE
150 '  F1:    STATUS OF FIRE (SPACE BAR)
160 ' ---------------------------------

170 ' *** INITIALIZE ***

180 DIM MOVE(4)
190 DATA Ø,Ø,-4Ø,4Ø
200 FOR N=1 TO 4
210 READ MOVE(N)
220 NEXT N
230 CHAR=Ø
240 COL=5
250 B1=CHAR+COL*8
260 E=B1+7
270 B=B1
280 CURSR=43
290 GOTO 37Ø

300 ' *** SUBROUTINE ***

310 A$=INKEY$:IF A$="" GOTO 31Ø
320 J1=ASC(A$):IF J1=32 THEN F1=1:RETURN ELSE F1=Ø
330 IF J1<28 OR J1>31 THEN MOVE=Ø:RETURN
340 MOVE=MOVE(J1-27)
350 RETURN

360 ' *** YOUR PROGRAM STARTS HERE ***

370 GOSUB 31Ø
380 IF MOVE<>Ø THEN PRINT @ B1,CHR$(32);
390 B1=B1+MOVE
400 IF B1<B OR B1>E THEN B1=B1-MOVE
410 PRINT @ B1,CHR$(CURSR);
420 GOTO 37Ø
```

Drawing Subroutine

WHAT IT DOES

Draws on screen, changing cursor character as desired.

Variables

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

How To Use The Subroutine

Using the computer 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, under direction of the operator. Pressing the FIRE key (SPACE bar) will clear the screen.

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

If you press an alpha key the cursor will change to that letter or, if GRAPH or the CODE key, the graphics character associated with that key. 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 B and E. Your program should always check to make sure that B1 is never less than B or greater than E. You may want to define B1=B 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 PRINTing @ B1 with the the character you want. Erase B1 from its old position by PRINTing its former value with CHR$(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 140: Define an array capable of keeping track of four possible values for movement.

Line 150: Define DATA for direction of movement when joystick is pressed in any of the four possible values.

Lines 160 to 180: Read move data into array.

Lines 190 to 210: Define beginning and end of screen.

Line 220: Define cursor character. Program will change.

Line 250: Wait for user to press key.

Line 260: If ESCape key pressed, then clear screen.

Line 270: Determine ASCII value of key pressed.

Line 280: If arrow key not pressed, MOVE=0, and cursor changes to that character.

Line 290: MOVE equals J1 element of array.

Line 300: Return.

Line 320: Access the subroutine.

Line 330: Make beep.

Line 340: Update B1 by MOVE.

Line 350: Cancel if B1 outside screen limits.

Line 360: PRINT cursor character to screen.

Line 370: Repeat.

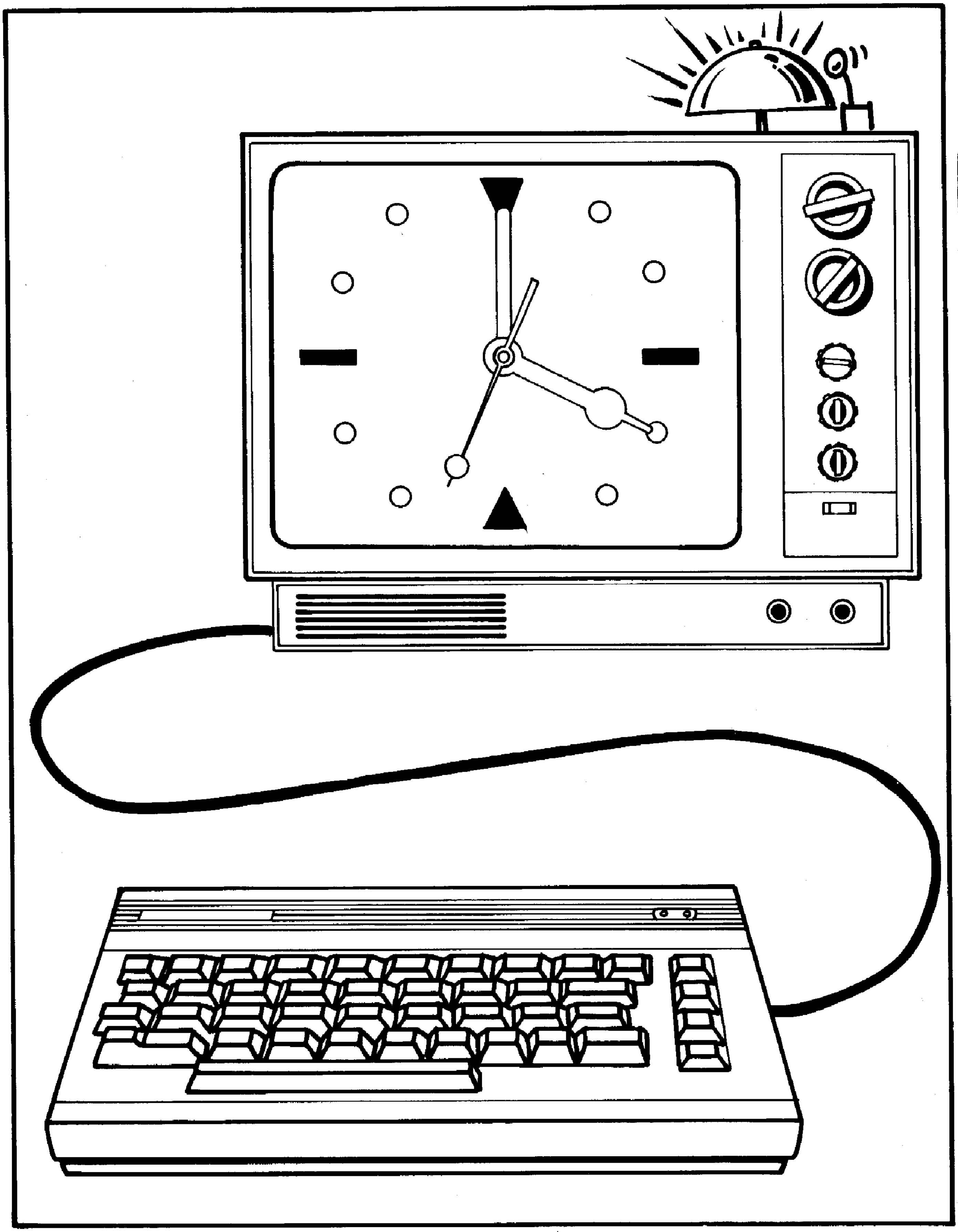
You Supply

CURSR: Current cursor character. This number is PRINTed into position B1 to paint on the screen.

RESULT

Drawing on screen, with variety of cursor characters.

```
10 ' ********
20 ' *      *
30 ' * DRAW *
40 ' *      *
50 ' ********
60 '  ------------------------------
70 '       ++ VARIABLES ++
80 '    MOVE:   DIRECTION OF MOVE
90 '    B1:     POSITION OF CURSOR
100 '   CURSR:  CURSOR CHARACTER
110 '
120 ' ------------------------------.
130 ' *** INITIALIZE ***
140 DIM MOVE(4)
150 DATA 1,-1,-40,40
160 FOR N=1 TO 4
170 READ MOVE(N)
180 NEXT N
190 B1=1:B=B1
200 CLS
210 E=318
220 CURSR=43
230 GOTO 320

240 ' *** SUBROUTINE ***

250 A$=INKEY$:IF A$="" GOTO 250
260 IF A$=CHR$(27)THEN CLS:MOVE=0:RETURN
270 J1=ASC(A$)
280 IF J1<28 OR J1>31 THEN MOVE=0:CURSR=ASC(A$):RETURN
290 MOVE=MOVE(J1-27)
300 RETURN

310 ' *** DRAW SUBROUTINE ***

320 GOSUB 250
330 BEEP
340 B1=B1+MOVE
350 IF B1>E OR B1<B THEN B1=B1-MOVE
360 PRINT @ B1,CHR$(CURSR);
370 GOTO 320
```



Chapter Two

Using the Clock and Interrupts

Many of the previous Radio Shack computers have used the Zilog Z-80 microprocessor chip. The TRS-80 Model 100 was the first to use a related semiconductor, the 80C85. This is a CMOS version of the veteran 8085 microprocessor, a member of the same 8008, 8080, Z-80 family that is so popular among Radio Shack and CP/M-based computers. The CMOS attributes allow the chip to be operated at much lower power levels; hence the viability of the battery-operated Model 100.

While the 8085 shares many of the same instructions as the Z-80 used in other Radio Shack computers, it does have much better "interrupt" features. That is, the microprocessor will, when told, constantly poll various registers for activity. When the right sort of action is found, the chip interrupts whatever else is going on and can, if we tell it to, go do something else instead.

The most commonly used interrupt routine from BASIC is ON ERROR GOTO.... Unlike other commands, we do not have to enter an interrupt-based command at the time we want it to be carried out. Instead, we place ON ERROR at the beginning of a program. The computer remembers that we have turned this feature on, and when an error is encountered, it will send program control where we choose, rather than invoking the normal error routine.

The TRS-80 has a variety of ON...GOSUB routines available. In all cases, your program merely starts the routine at the beginning and then goes on to do other tasks as you wish. For example, ON KEY GOSUB will interrupt whatever you are doing whenever one of the defined function keys is pressed. You can define one or all eight function keys for this routine. The others are ignored.

ON MDM and ON COM will interrupt your program whenever a character is received from the built-in modem or arrives over the RS232 port, respectively. In this way, your computer can carry out one task but still not miss data coming in from another computer through the modem or serial port.

ON TIME$ GOSUB will let you be a clock-watcher, with your program interrupting you when a desired time is reached. This is a handy way of providing for a timer in any BASIC program you might have.

Timing is one of the most interesting interfaces your Portable Computer has with the real world. The TRS-80 has a built-in 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. Some of the subroutines in this book are your keys to using the real-time clock of your TRS-80 computer.

Because the TRS-80's real-time clock is accurate under most circumstances, it can be used to time events fairly precisely. This might be useful in competitive games, typing tutors, and other programs that measure elapsed time accurately.

ELAPSED TIME

WHAT IT DOES

Measures difference between two times.

Variables

- HOUR: Elapsed hours
- MIN: Elapsed minutes
- SEC: Elapsed seconds.

How To Use The Subroutine

This subroutine takes the time when the program starts running and compares it to the current time when you press a key (or perform some other task in the program that you write). It then calculates the elapsed time.

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 clock returns to 00:00:00 at midnight. But for anything short of a day or which does not begin on one day and end the next, the subroutine should do just fine. Most programs will not need to time that long a period for a single run.

Line-By-Line Description

Line 160: Set F$ (finish time) to equal the current time, TIME$. Then determine the value of the hour.

Line 170: Find value of minutes.

Line 180: Find value of seconds.

Lines 190 to 210: Figure starting hour, minutes, and seconds.

Line 220: Figure difference, in seconds, between starting and finishing times.

Line 230: If difference is less than 60 seconds, then elapsed time equals the difference.

Line 240: If difference is less than an hour, go figure minutes and seconds elapsed.

Line 250: Calculate elapsed hours.

Line 260: Calculate elapsed minutes.

Line 270: Calculate elapsed seconds.

Lines 280 to 290: Print results.

Line 320: Take starting time.

Line 330: Wait awhile. Your program could have a game or some other activity replacing this.

Line 340: When it is necessary to calculate the elapsed time (in this case, because a key was pressed, access the subroutine).

You Supply

Your program should set S$ to equal TIME$ when you wish to start timing and then call the subroutine when the end of the timing cycle is over.

RESULT

Elapsed time is measured.

```
10 ' ****************
20 ' *              *
30 ' * ELAPSED TIME *
40 ' *              *
50 ' ****************
60 '
70 '  ---------------------------
80 '       ++ VARIABLES ++
90 '     HOUR:  ELAPSED HOURS
100 '    MIN:   ELAPSED MINUTES
110 '    SEC:   ELAPSED SECONDS
120 '
130 ' ---------------------------
140 GOTO 320

150 ' *** SUBROUTINE ***

160 F$=TIME$:FH=VAL(LEFT$(F$,2))
170 FM=VAL(MID$(F$,4,2))
180 FS=VAL(RIGHT$(F$,2))
190 SH=VAL(LEFT$(S$,2))
200 SM=VAL(MID$(S$,4,2))
210 SS=VAL(RIGHT$(S$,2))
220 DF=(FH*3600+FM*60+FS)-(SH*3600+SM*60+SS)
230 IF DF<60 THEN SEC=DF:GOTO 280
240 IF DF<3600 THEN GOTO 260
250 HOUR=INT(DF/3600):DF=DF-HR*3600
260 MIN=INT(DF/60)
270 SEC=DF-MIN*60
280 PRINT TAB(2)"IT TOOK YOU ";HOUR
290 PRINT TAB(2)MIN;"MIN. AND ";SEC;"SEC."
300 RETURN

310 ' *** YOUR PROGRAM STARTS HERE ***

320 S$=TIME$
330 A$=INKEY$:IF A$="" GOTO 330
340 GOSUB 160
350 GOTO 320
```

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 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. The subroutine's chief value, in fact, is to demonstrate how efficient the ON TIME$ interrupt routine which follows is for BASIC programming.

In order to come close with this subroutine, it would be necessary to change the module 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 170 to 210: User enters amount of time to be measured.

Line 220: Announce timing cycle.

Line 230: Take current time.

Lines 240 to 250: Figure the minutes and hours to start.

Lines 260 to 270: Calculate value of finish hours and minutes.

Lines 280 to 300: Construct string representations of the finish hours, minutes, and seconds.

Lines 310 to 320: Add leading 0s as necessary.

Line 330: Construct finish time string.

Line 340: Clear screen.

Line 350: PRINT finish time to screen.

Line 360: PRINT current time to screen.

Line 370: Check to see if finish time has been reached.

Line 380: Otherwise, repeat.

Line 390: Notify time is up.

You Supply

You enter the time to be counted off.

RESULT

Computer signals at end of requested time interval.

```
10 ' *********
20 ' *       *
30 ' * TIMER *
40 ' *       *
50 ' *********
60 GOTO 410
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '      FH$ FINISH HOUR
100 '     FM$ FINISH MINUTES
110 '     FS$ FINISH SECONDS
120 '     FT$ FINISH TIME
130 '
140 ' ---------------------------

160 ' *** TIME TO BE MEASURED ***
```

```
170 PRINT" TOTAL TIME TO BE COUNTED:"
180 INPUT" ENTER HOURS:";HR$
190 INPUT" ENTER MINUTES:";MN$
200 HR=VAL(HR$)
210 MN=VAL(MN$)
220 PRINT TAB(4)"TIMING CYCLE"
230 T$=TIME$
240 HN=VAL(LEFT$(T$,2))
250 MP=VAL(MID$(T$,4,2))
260 FM=MP+MN:IF FM>59 THEN FM=FM-60:HN=HN+1
270 FH=HN+HR:IF FH>23 THEN FH=FH-24
280 FH$=MID$(STR$(FH),2)
290 FM$=MID$(STR$(FM),2)
300 FS$=MID$(T$,7)
310 IF VAL(FH$)<9 THEN FH$="0"+MID$(FH$,2):
    IF VAL(FH)<1 THEN FH$="00"
320 IF VAL(FM$)<9 THEN FM$="0"+MID$(FM$,2):
    IF VAL(FM$)<1 THEN FM$="00"
330 FT$=FH$+FM$+FS$
340 CLS:
350 PRINT @ 5,"FINISH TIME: ";FH$;":";FM$;
360 PRINT @ 45,"CURRENT TIME: ";TIME$;
370 IF VAL(TIME$)>VAL(FT$) GOTO 390
380 GOTO 350
390 PRINT"TIME IS UP.":BEEP:RETURN

400 ' *** YOUR PROGRAM STARTS HERE ***

410 PRINT
420 GOSUB 170
```

Second Timer

WHAT IT DOES

Counts off seconds.

Variables

- SEC: Number of seconds to count off.

How To Use The Subroutine

This subroutine shows yet a third way of using the computer as a timer. With this method, a FOR-NEXT loop repeats once for each second until the required interval has elapsed. How do we make sure that each iteration of the loop takes exactly one second? At the start of the loop, the current value of TIME$ is taken. Then, the program pauses and compares the new time with the past value of TIME$. If they are identical (meaning less than one second has elapsed), the subroutine continues to wait. Only when an additional second has ticked off will the next trip through the loop take place.

When a certain number of seconds should be counted off, this method is faster than using the Micro Executive Workstation's interrupt feature. This is because the starting time doesn't have to be equated with the time at the finish. This method is very simple for figuring intervals that are measured in whole seconds. To count off 122 seconds, merely enter that amount when prompted. There is no need to convert to minutes or hours.

Like all non-interrupt-driven routines, this one does prevent your program from tackling other chores during the timing interval.

Line-By-Line Description

Line 140: Start subroutine from 1 to number of seconds desired, SEC.

Line 150: Take current time.

Line 160: Compare current time to see if one second has elapsed. Loop if not.

Line 170: Print current time to screen.

Line 180: Count off next second.

Lines 190 to 200: Notify that time is up.

Line 240: Ask user for number of seconds to count.

You Supply

Value for SEC, either through user input, or by defining this variable.

RESULT

TRS-80 signals at end of requested number of seconds.

```
10 ' ***********
20 ' *         *
30 ' * SECOND  *
40 ' * COUNTER *
50 ' *         *
60 ' ***********
70 GOTO 220
80 '  ---------------------------
90 '      ++ VARIABLES ++
100 '      SEC: NUMBER OF SECONDS
110 '           TO BE COUNTED
120 ' ---------------------------

130 ' *** TIME TO BE MEASURED ***

140 FOR N=1 TO SEC
150 T$=TIME$
160 IF T$=TIME$ GOTO 160
170 PRINT @ 60,TIME$;
180 NEXT N
190 CLS
200 PRINT "TIME IS UP."
210 RETURN

220 ' *** YOUR PROGRAM STARTS HERE ***

230 PRINT
240 INPUT"HOW MANY SECONDS TO COUNT";SEC
250 GOSUB 140
```

Time$ Interrupt

WHAT IT DOES

Allows computer to interrupt task at requested time.

Variables

- TIME$: Current time
- FT$: Finish time.

How To Use The Subroutine

This subroutine uses the ON TIME$= interrupt routine to notify you when the desired time interval has elapsed. The module asks for the time when the "alarm" is desired, and sets the interrupt routine to trigger the notification subroutine when that time rolls around. Your program can go on and do other things in the meantime.

Line-By-Line Description

Lines 160 to 190: Ask user for time to be alerted, in HH:MM:SS format.

Lines 200 to 220: Extract hours, minutes, and seconds.

Lines 230 to 240: Check for proper format.

Line 250: Turn on TIME$ interrupt.

Line 260: Set ON TIME$ to requested finish time.

Line 300: Notify that time has expired.

Line 340: Access the subroutine.

Line 350: Print current time.

You Supply

Time when alarm should be triggered.

RESULT

Your program is interrupted when desired time is reached.

```
10 ' *************
20 ' *           *
30 ' *   TIME$   *
40 ' * INTERRUPT *
50 ' *           *
60 ' *************
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '
100 '   TIME$: CURRENT TIME
110 '   FT$:   FINISH TIME
120 '
130 ' ---------------------------
140 GOTO 330

150 ' *** SUBROUTINE ***
```

```
160 CLS
170 PRINT"What time would you like to be alerted:?"
180 PRINT "Use HH:MM:SS format"
190 INPUT FT$
200 A$=LEFT$(FT$,2)
210 B$=MID$(FT$,4,2)
220 C$=RIGHT$(FT$,2)
230 IF VAL(A$)>23 OR VAL(B$)>59 OR VAL(C$)>59 THEN PRINT
    "WRONG FORMAT!":GOTO 180
240 IF MID$(FT$,3,1) <>":" AND MID$(FT$,6,1)<>":" THEN
    PRINT "WRONG FORMAT!":GOTO 180
250 TIME$ ON
260 ON TIME$=FT$ GOSUB 290
270 RETURN

280 ' *** TIME IS UP ***

290 CLS
300 PRINT "TIME IS UP!":BEEP
310 STOP

320 ' *** YOUR PROGRAM STARTS HERE ***

330 PRINT
340 GOSUB 160
350 PRINT @ 10,TIME$;
360 GOTO 350
```

Modem/RS232 Interrupt

WHAT IT DOES

Interrupts program when signal is received over the RS232 or modem.

Variables

- B$: Character received.

How To Use The Subroutine

This subroutine provides a demonstration of the ON MDM and ON COM interrupt routines. As with ON ERROR and ON TIME$, your program can be going about its business while constantly checking the built-in modem or RS232 interface for input. When a character is received, the program can immediately branch to the designated subroutine.

You might want to use this feature while your TRS-80 is connected to another computer. You may run your BASIC program, and be signalled by the other computer when necessary.

The routine was written for RS232 use, with 300 baud communications using a seven-bit word, even parity, one stop bit, and XON/XOFF enabled. That is, the protocol is the "37E1E" setting, as explained in the TRS-80 manual.

You may change the routine to substitute the built-in modem by using MDM wherever COM appears in the listing. That is, line 130 should read OPEN "MDM:37E1E"...etc. Line 140 would be MDM ON, instead of COM ON, and line 150 should read ON MDM GOSUB 180.

You may also change the protocols, following the style used by TELCOM (or, actually, any of the Portable Computer's programs). For example, "58I1D" would activate 1200 baud communications with an 8-bit word, Ignore parity, 1 stop bit, and with XON/XOFF disabled.

Line-By-Line Description

Line 120: Set maximum number of files.

Line 130: Open communications channel to accept input.

Line 140: Turn on interrupt feature.

Line 150: Define subroutine to branch to.

Line 180: Accept one character from the channel.

Line 190: Print character received to screen.

Line 220: Your program starts here.

You Supply

Communications parameters, as desired.

RESULT

Your program is interrupted when signal is received over the modem or RS232.

```
10 ' *********************
20 ' *                   *
30 ' *  RS232 INTERRUPT  *
40 ' *                   *
50 ' *********************
60 '  ---------------------
70 '    +++ VARIABLES +++
80 '   B$ CHARACTER RECEIVED
90 '
100 ' ---------------------
```

```
110 ' *** INITIALIZE ***

120 MAXFILES=1
130 OPEN "COM:37E1E" FOR INPUT AS 1
140 COM ON
150 ON COM GOSUB 180
160 GOTO 220

170 ' *** SUBROUTINE ***

180 B$=INPUT$(1,1)
190 PRINT B$;
200 RETURN

210 ' *** YOUR PROGRAM STARTS HERE ***

220 A$=INKEY$:IF A$="" GOTO 220
230 PRINT A$;
240 GOTO 220
```

Function Key Interrupt

WHAT IT DOES

Sends control to desired subroutine when a function key is pressed.

Variables

- None

How To Use The Subroutine

With ON MDM or ON COM, we had the option of sending control to only one subroutine when the interrupt was triggered. With ON KEY, as many as eight subroutines, one for each of the special function keys, can be designated.

The command allows accounting for as many or as few of the function keys as we wish:

```
ON KEY GOSUB 1000,,2000,,,3000
```

This would activate F1, sending control to line 1000, F3 (line 2000), and F6 (line 3000). The commas designate keys to be skipped. And, since none of the other keys are defined, the TRS-80 will ignore them.

As with all the interrupt routines (except ON ERROR), ON KEY must be activated, in this case with the KEY ON command.

Line-By-Line Description

Line 110: Turn on interrupt routine.

Line 180: Define keys to be used and subroutine lines.

Lines 180 to 220: Subroutines. Replace with the desired functions you want to include.

Line 250: Wait for user to press key.

Line 260: Notify that key was pressed.

You Supply

Subroutines to carry out desired actions.

RESULT

Program interrupted when defined function key pressed.

```
10 ' ***************************
20 ' *                         *
30 ' * FUNCTION KEY INTERRUPT  *
40 ' *                         *
50 ' ***************************
60 '  ----------------------
70 '    +++ VARIABLES +++
80 '         NONE
90 '
100 ' ----------------------
110 KEY ON
120 ON KEY GOSUB 180,200,,220
130 GOTO 250

170 ' *** SUBROUTINES ***

180 PRINT "KEY 1 PRESSED."
190 RETURN
200 PRINT "KEY 2 PRESSED."
210 RETURN
220 PRINT "KEY 4 PRESSED."
230 RETURN

240 ' *** YOUR PROGRAM STARTS HERE ***

250 A$=INKEY$:IF A$="" GOTO 250
260 PRINT A$;"KEY PRESSED."
270 GOTO 250
```

Error Handler

WHAT IT DOES

Provides longer error messages.

Variables

- ERR: Error code
- ERL: Line error is in.

How To Use The Subroutine

You may put this routine within programs you are debugging, as a speedier way of determining what errors have taken place. There is no need to look up the cryptic two-character error codes that the Portable Computer provides.

You may also build in error-trapping routines to help the naive user better operate your program. Once an error has been found and control sent to the proper subroutine, you may correct the problem and then RESUME the program at an appropriate place. The Portable Computer allows sending control back to the same place the error took place (RESUME) or at the following line (RESUME NEXT). In many cases, RESUME is not appropriate, because the error is a fatal one. Some suggested results are included; you may change them to suit your own programs.

Line-By-Line Description

Line 140: Activate ON ERROR interrupt routine.

Line 170: Assign variable E with error code.

Lines 180 to 220: Calculate which subroutine to access.

Line 230: Access error routine.

Lines 250 to 540: Print error messages.

You Supply

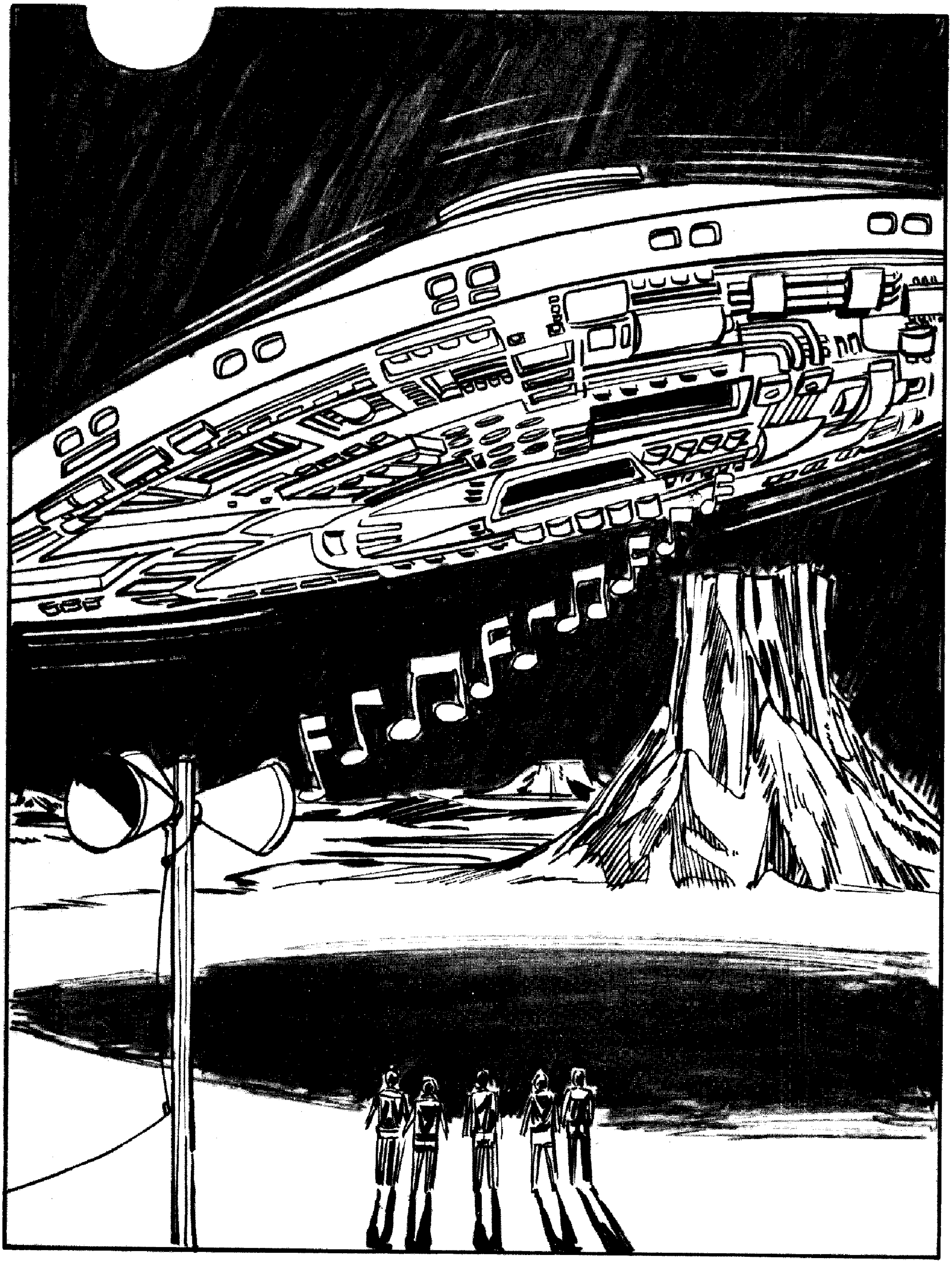
Program to be error-trapped.

RESULT

Longer error messages printed to screen.

```
10 ' ********************
20 ' *                  *
30 ' *  ERROR HANDLING  *
40 ' *                  *
50 ' ********************
60 '  ---------------------
70 '    +++ VARIABLES +++
80 '
90 '   ERR: ERROR CODE
100 '  ERL: LINE ERROR IS IN
110 '
120 ' ---------------------

130 ' *** INITIALIZE ***

140 ON ERROR GOTO 170
150 GOTO 560

160 ' *** SUBROUTINE ***

170 E=ERR
180 IF E>22 AND E<51 THEN E=21:GOTO 220
190 IF E>58 THEN E=21:GOTO 220
200 IF E=57 THEN E=21:GOTO 220
210 IF E>50 AND E<58 THEN E=E-29
220 IF ERL=65535 THEN PRINT "ERROR MADE IN DIRECT MODE"
    :GOTO 230
230 PRINT "ERROR IN LINE";ERL:ON E GOSUB 250,260,270,280,290,
    300,310,320,330,340,350,360,370,380,390,400,410,420,430,
    440,450,460,470,480,490,500,510,520,530,540
240 RESUME
250 PRINT"NEXT without FOR":STOP
260 PRINT"Syntax Error":STOP
270 PRINT"RETURN without matching GOSUB":STOP
280 PRINT"Not enough DATA available.":STOP
290 PRINT"Illegal function call.":STOP
300 PRINT"Overflow error.":STOP
310 PRINT"Out of Memory.":STOP
320 PRINT"Undefined line number.":STOP
330 PRINT"Subscript out of range.":STOP
340 PRINT"Re-dimensioned array.":RESUME NEXT
350 PRINT"Division by Zero":RESUME NEXT
360 PRINT"Illegal direct":STOP
370 PRINT"Mixed string and numeric variable types.":STOP
380 PRINT"Out of string space. CLEAR more.":STOP
390 PRINT"String too long":RESUME
400 PRINT"String formula too complex. Rewrite.":STOP
```

```
410 PRINT"Can't Continue.":STOP
420 PRINT"Input/output error.":STOP
430 PRINT"ON ERROR without RESUME.":RESUME
440 PRINT"RESUME encountered without ERROR.":RESUME
450 PRINT"Undefined Error.":RESUME
460 PRINT"Missing Operand":STOP
470 PRINT"File number unavailable.":RESUME NEXT
480 PRINT"That file not OPEN":STOP
490 PRINT"File not found":FILES:STOP
500 PRINT"That file already open":RESUME NEXT
510 PRINT"Input past end of the file":RESUME NEXT
520 PRINT "Bad file name":RESUME
530 PRINT"Statement without line number in file":STOP
540 PRINT"That file has not been OPENed.":STOP

550 ' *** YOUR PROGRAM STARTS HERE ***

560 PRINT
```



Chapter Three

Using Sound

The TRS-80 Portable Computer has only primitive sound capabilities, when compared to some home computers. It has only two sound commands, BEEP and SOUND, to deliver notes from the built-in speaker. BEEP allows no arguments. With SOUND, you can specify both a note and duration:

```
SOUND 12ØØØ, 1Ø
```

The note can range over five octaves, and the time interval can range from a fraction of a second to several seconds. This chapter includes one routine that will turn your computer keyboard into a piano to allow you to play notes by pressing appropriate keys. The other nine subroutines produce different sound effects which you can drop into your game, personal, or business programs as appropriate.

Music

WHAT IT DOES

Uses keyboard to generate various musical notes.

Variables

- P(n): Note value
- NME$(n): Name of note
- NT$(14): Key corresponding to note
- LTH: Length of note.

How To Use The Subroutine

Use this subroutine where you want to have the Micro Executive Workstation keyboard simulate a piano. Pressing appropriate keys on the home row produces a note from the speaker. In addition, the name of the note (and a sharp symbol if the note is sharp) is printed to the screen. Some of the keys are deactivated because, as music students will know, there is only a half-step between some notes on the scale and others. Therefore, there is no sharped (or flatted) note between them. Music students will also recognize that flats are simply another way of writing a sharp note, (e.g., B-flat is the same as A#).

The user can "learn" the TRS-80 keyboard, and begin to play songs using the sound capabilities of the portable.

Line-By-Line Description

Line 160: Define length of note to be played.

Line 170: DIMension arrays to store names of notes, proper keys, and note values.

Lines 180 to 200: Read note names into array.

Lines 210 to 230: Read acceptable keys into array.

Lines 240 to 260: Read note values into array.

Lines 280 to 310: DATA for above arrays.

Line 330: Wait for user to press key.

Line 340: If key was carriage return, end subroutine.

Line 350 to 370: See if key matches list of acceptable choices.

Line 390: Play note.

Line 400: Erase old note name on screen.

Line 410: Print name of new note.

Line 440: Access subroutine.

You Supply

No user input required, other than to play keyboard.

RESULT

Music played from TRS-80's speaker.

```
10 REM *****************
20 REM *               *
30 REM * MUSICAL NOTES *
40 REM *               *
50 REM *****************
60 REM  ----------------------------
70 REM     ++ VARIABLES ++
80 REM    P(n):     NOTE VALUE
90 REM    NME$(n):  NAME OF NOTE
100 REM   NT$(14):  KEY CORRESPONDING
110 REM               TO NOTE
120 REM   LTH:      LENGTH OF NOTE
130 REM
140 REM ----------------------------

150 REM *** INITIALIZE ***
```

```
160 LTH=200
170 DIM NME$(14),NT$(14),P(14)
180 FOR N=1 TO 14
190 READ NME$(N)
200 NEXT N
210 FOR N=1 TO 14
220 READ NT$(N)
230 NEXT N
240 FOR N=1 TO 14
250 READ P(N)
260 NEXT N
270 DATA G,G#,A,A#,B,C,C#,D,D#,E,F,F#,G,G#
280 DATA A,W,S,E,D,F,T,G,Y,H,J,I,K,O
290 DATA 12538,11836,11172,10544,9952,9394,8866
300 DATA 8368,7900,7456,7032,6642,6269,5918
310 GOTO 440

320 REM *** SUBROUTINE ***

330 A$=INKEY$:IF A$="" GOTO 330
340 IF A$=CHR$(13) THEN RETURN
350 FOR N=1 TO 14
360 IF A$=NT$(N) GOTO 390
370 NEXT N
380 GOTO 330
390 SOUND P(N),LTH
400 PRINT @ 10," ";
410 PRINT @ 10,NME$(N);
420 GOTO 330

430 REM *** YOUR PROGRAM STARTS HERE ***

440 GOSUB 330
```

Siren

WHAT IT DOES

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

Variables

- N1: Number of repeats of sound effect.

How To Use The Subroutine

Call subroutine when siren sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Begin loop of ten repetitions.

Lines 170 to 190: Loop through notes from 400 to 12000, in increments of 500. Produces falling pitch.

Lines 200 to 220: Rising pitch.

Line 250: Access the subroutine.

You Supply

No user changes required.

RESULT

Siren sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' * SIREN *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '       ++ VARIABLES ++
90 '
100 '     N1: NUMBER OF REPEATS
110 '
120 '
130 ' ---------------------------
140 GOTO 260

150 ' *** SUBROUTINE ***
```

```
160 FOR N1=1 TO 10
170 FOR N=400 TO 12000 STEP 500
180 SOUND N,1
190 NEXT N
200 FOR N=12000 TO 400 STEP -500
210 SOUND N,1
220 NEXT N
230 NEXT N1
240 RETURN

250 ' *** YOUR PROGRAM STARTS HERE ***

260 GOSUB 160
```

Bomb

WHAT IT DOES

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

Variables

- None.

How To Use The Subroutine

Call subroutine when bomb sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Lines 160 to 180: Produce falling sound.

Lines 190 to 210: Rising sound, explosion.

Line 240: Access the subroutine.

You Supply

No user changes required.

RESULT

Bomb sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' * BOMB  *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '     ++ VARIABLES ++
90 '
100 '       NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 240

150 ' *** SUBROUTINE ***

160 FOR N=800 TO 8000 STEP 20
170 SOUND N,1
180 NEXT N
190 FOR N=13000 TO 800 STEP -1000
200 SOUND N,1
210 NEXT N
220 RETURN

230 ' *** YOUR PROGRAM STARTS HERE

240 GOSUB 160
```

Alarm

WHAT IT DOES

Alarm sound routine produces sounds for games, other applications.

Variables

- N1: Number of repeats.

How To Use The Subroutine

Call subroutine when alarm sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Repeat sound N1 (10) times.

Lines 170 to 220: Produce rising sound.

Line 240: Access the subroutine.

You Supply

No user changes required.

RESULT

Alarm sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' * ALARM *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '
100 '     N1: NUMBER OF REPEATS
110 '
120 '
130 ' ---------------------------
140 GOTO 230

150 ' *** SUBROUTINE ***
```

```
160 FOR N=1 TO N1
170 FOR N2=8000 TO 11000 STEP 100
180 SOUND N2,1
190 NEXT N2
200 NEXT N
210 RETURN

220 ' *** YOUR PROGRAM STARTS HERE ***

230 GOSUB 160
```

UFO

WHAT IT DOES

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

Variables

- None.

How To Use The Subroutine

Call subroutine when UFO sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Lines 160 to 170: Define two notes for UFO to play.

Lines 180 to 190: Alternate those two notes.

Lines 200 to 210: Reduce A by 100; increase B by 100.

Line 220: If A is out of range, end subroutine.

Line 250: Access subroutine.

You Supply

No user changes required.

RESULT

UFO sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' *  UFO  *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '
100 '         NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 250

150 ' *** SUBROUTINE ***

160 A=13000
170 B=400
180 SOUND A,1
190 SOUND B,1
200 A=A-100
210 B=B+100
220 IF A=<1 THEN RETURN
230 GOTO 180

240 ' *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 160
```

Computer

WHAT IT DOES

Random, computer-like sound routine to produce sounds for games, other applications.

Variables

- None.

How To Use The Subroutine

Call subroutine when computer sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 150: Repeat 100 times.

Line 160: Select random note.

Line 170: Select random length.

Line 180: Play note.

Line 220: Access subroutine.

You Supply

No user changes required.

RESULT

Computer sound emitted for game play or other applications.

```
10 ' ************
20 ' *          *
30 ' * COMPUTER *
40 ' *          *
50 ' ************
60 '
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '
100 '         NONE
110 '
120 ' ---------------------------
130 GOTO 220

140 ' *** SUBROUTINE ***
```

```
15Ø FOR N=1 TO 1ØØ
16Ø  R=RND(1)*13ØØØ
17Ø L=RND(1)*1Ø
18Ø SOUND R,L
19Ø NEXT N
2ØØ RETURN

21Ø ' *** YOUR PROGRAM STARTS HERE ***

22Ø GOSUB 15Ø
```

Laser

WHAT IT DOES

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

Variables

- None.

How To Use The Subroutine

Call subroutine when laser sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Loop from 400 to 8000 in increments of 1000.

Line 170: Make laser sound.

Line 220: Access subroutine.

You Supply

No user changes required.

RESULT

Laser sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' * LASER *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '       ++ VARIABLES ++
90 '
100 '      NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 220

150 ' *** SUBROUTINE ***

160 FOR N=400 TO 8000 STEP 1000
170 SOUND N,1
180 NEXT N
190 SOUND 400,10
200 RETURN

210 ' *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 160
```

Roulette Wheel

WHAT IT DOES

Makes roulette wheel sound, which slows down gradually, for games, other applications.

Variables

- None.

How To Use The Subroutine

Call subroutine when wheel sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Set initial delay at 10.

Line 170: Loop sound from 400 to 12000.

Line 180: Increase length of delay loop.

Line 190: Count off delay.

Line 200: Play sound.

Line 240: Access subroutine.

You Supply

No user changes required.

RESULT

Roulette wheel-type sound emitted for game play or other applications.

```
10 ' ******************
20 ' *                *
30 ' * ROULETTE WHEEL *
40 ' *                *
50 ' ******************
60 '
70 '  ---------------------------
80 '       ++ VARIABLES ++
90 '
100 '          NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 240

150 ' *** SUBROUTINE ***

160 DELAY=10
170 FOR N=400 TO 12000 STEP 200
180 DELAY=DELAY*1.05
190 FOR D=1 TO DELAY:NEXT D
200 SOUND N,1
210 NEXT N
220 RETURN

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 160
```

Heartbeat

WHAT IT DOES

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

Variables

- None.

How To Use The Subroutine

Call subroutine when heartbeat sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Repeat 20 times.

Line 170: Produce first thump.

Line 180: Wait.

Line 190: Produce second thump.

Line 200: Wait slightly longer.

Line 210: Repeat.

Line 240: Access subroutine.

You Supply

No user changes required.

RESULT

Heartbeat sound emitted for game play or other applications.

```
10 ' *************
20 ' *           *
30 ' * HEARTBEAT *
40 ' *           *
50 ' *************
60 '
70 '  ---------------------------
80 '       ++ VARIABLES ++
90 '
100 '     NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 240

150 ' *** SUBROUTINE ***

160 FOR N1=1 TO 20
170 SOUND 14000,1
180 FOR N=1 TO 100:NEXT N
190 SOUND 14000,5
200 FOR N=1 TO 300:NEXT N
210 NEXT N1
220 RETURN

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 160
```

Clock

WHAT IT DOES

Clock ticking sound routine to produce sounds for games, other applications.

Variables

- N2: Number of ticks.

How To Use The Subroutine

Call subroutine when clock sound is desired. You may experiment with loops and actual numbers used to produce a different sound effect. Try varying the values used with the SOUND statement. The first value controls the note produced, while the second adjusts the length of time the note is played. Five octaves are encompassed between the numbers 12538 (at the low end) and 415 (at the high end).

Line-By-Line Description

Line 160: Repeat ten times.

Line 170: Tick.

Line 180: Wait.

Line 190: Tock.

Line 200: Wait again.

Line 210: Repeat

Line 240: Access subroutine.

You Supply

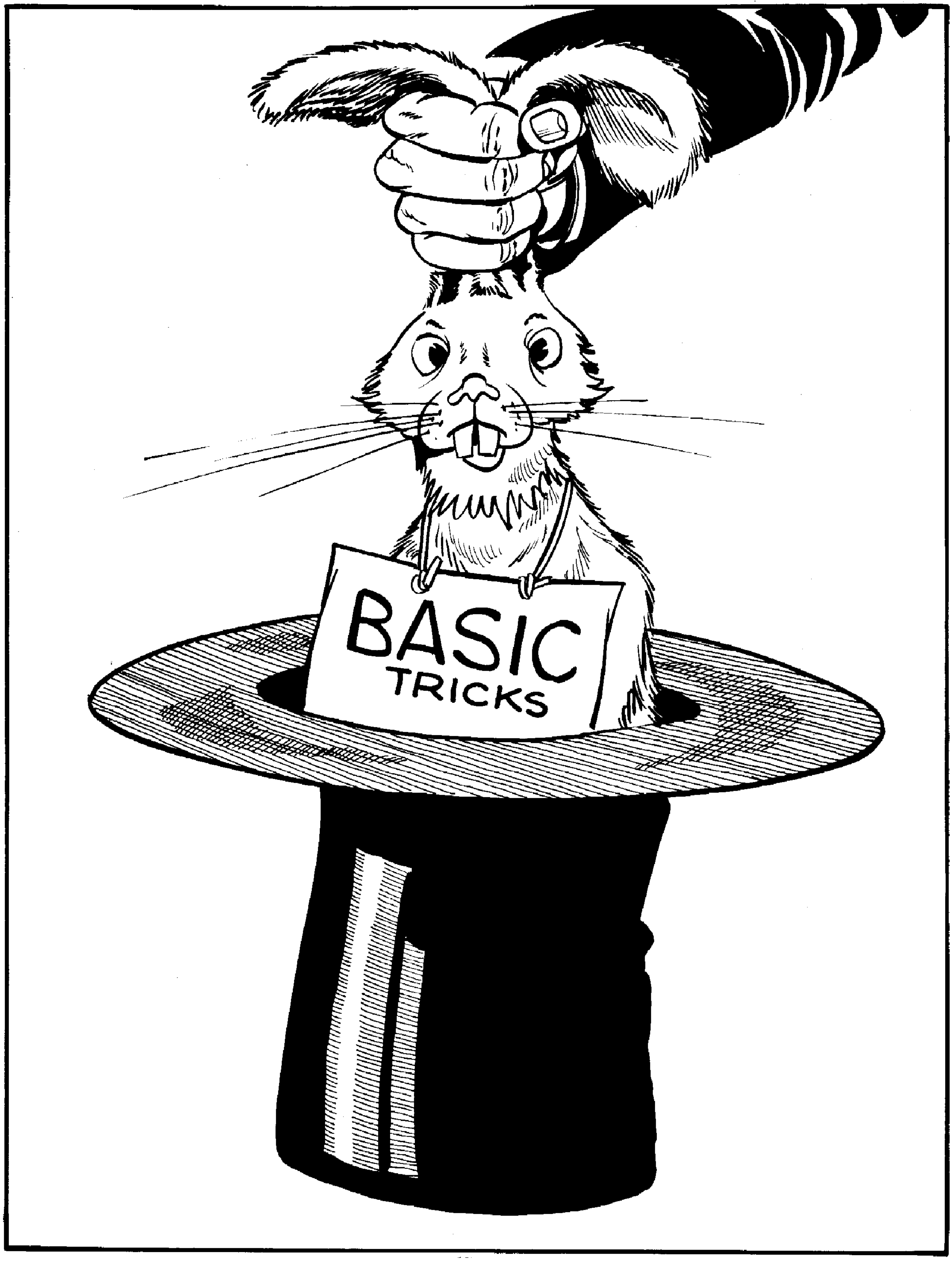
No user changes required.

RESULT

Clock sound emitted for game play or other applications.

```
10 ' *********
20 ' *       *
30 ' * TICK  *
40 ' *       *
50 ' *********
60 '
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '
100 '   N2: NUMBER OF TICKS
110 '
120 '
130 ' ---------------------------
140 N2=10:GOTO 240

150 ' *** SUBROUTINE ***
```

```
160 FOR N1=1 TO N2
170 SOUND 6000,1
180 FOR N=1 TO 200:NEXT N
190 SOUND 8000,1
200 FOR N=1 TO 200:NEXT N
210 NEXT N1
220 RETURN

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 160
```


BASIC
TRICKS

Chapter Four

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. One lets you PEEK the Portable Computer's equivalent of screen memory to see what characters are displayed there. Another lets you define a pair of variables that will turn on and off the reverse screen display attribute.

While redefining the function keys can be accomplished from BASIC just by typing "KEY (number),(string)," there are times when you might want to accomplish this under program control. A subroutine lets you do the redefinition (plus return the keys to normal) from within a program.

Three other modules are "user interface" routines that trap errors by permitting the operator to enter ONLY the type of input that is required by the program. If numbers only, or alpha characters only, are desired, that is what the routines will accept. The third accepts either lowercase or uppercase entries and converts the lowercase characters to upper.

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.

Two sort routines are included for those who need to rearrange lists of numbers or strings. Bubble sorts are used, as these are the easiest to understand and to modify. If you are interested in a faster, more sophisticated, but more difficult to follow, sort, Radio Shack's user's guide has a routine which sorts ".DO" files, using the Shell-Metzner sort.

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 RAM or tape file read/write routines provided later in this book, to build a complete data base program with permanent files.

Function Keys

WHAT IT DOES

Redefines function keys, or restores them to normal, under program control.

Variables

- K: Key to reprogram
- S$: New definition of that key.

How To Use The Subroutine

The TRS-80 Portable Computer doesn't have screen memory like the TRS-80 Models I and III. With those computers, the user can PEEK video memory to see what characters are currently being displayed and can POKE new characters there, as desired. You cannot POKE characters with the Micro Executive Workstation; however, a representation of the screen DOES exist in RAM. It is practical to PEEK those locations to see what is currently being displayed.

This capability is especially useful for games in which we want to see what lies ahead of our moving ship or missile. Placing graphics on the screen has to be done using PRINT @, which is, fortunately, very fast. We can PRINT @ any of 320 locations on the screen, from 0 to 319. If the PRINT @ statement is followed by a semicolon (except for the last screen position), the screen will not SCROLL. Thus, "PRINT@ A" will function nearly the same as POKing to video memory on other computers. PEEKing to A-512 will function the same as PEEKing to video memory.

The following subroutine handles the chore for you automatically. While it asks you which address to PEEK, your actual program will probably define A depending on the movement of a screen object or some other parameter.

Line-By-Line Description

Line 140: Define PRINT @ address to PEEK.

Line 150: Check that RAM location and assign value to CHAR$.

You Supply

PRINT @ address to PEEK.

RESULT

CHAR$ will equal character printed on screen.

```
10 ' ****************
20 ' *              *
30 ' *  PEEK SCREEN *
40 ' *              *
50 ' ****************
60 GOTO 180
70 '  --------------------------
80 '       ++ VARIABLES ++
90 '      A: POSITION TO PEEK
100 '     CHAR$: CHARACTER FOUND
110 '
120 ' --------------------------
```

```
130 ' *** SUBROUTINE ***

140 INPUT"WHICH PRINT @ POSITION?";A
150 CHAR$=CHR$(PEEK(A-512))
160 RETURN

170 ' *** YOUR PROGRAM STARTS HERE ***
180 GOSUB 140
190 PRINT "FOUND :";CHAR$
```

Reverse Screen

WHAT IT DOES

Turns REVERSE SCREEN attribute on and off.

Variables

- REV$: Variable to turn REVERSE on
- FF$: Variable to turn REVERSE off.

How To Use The Subroutine

Although the feature is not well-documented, the TRS-80 is able to turn its reverse character mode on and off from BASIC. The correct sequence is ESC "p" (note: it must be lowercase p, not uppercase P) to turn on the reverse attribute. ESC "q" will turn reverse off. You can't do this from command mode. That is, holding down the escape key and pressing "p" has no effect. However, the feature works fine from within a program. Rather than typing PRINT CHR$(27);"p" repeatedly, (CHR$(27) is the escape character), we can define a variable, REV$, as CHR$(27)+CHR$(112). Then PRINT REV$ will turn the trick for us. This subroutine also defines FF$ as CHR$(27)+CHR$(113) to turn the attribute off when we want.

Line-By-Line Description

Line 140: Define REV$.

Line 150: Define FF$.

Lines 190 to 200: Provide an example of each type of screen attribute.

You Supply

PRINT statements where you desire reverse characters.

RESULT

REVERSE turned on and off.

```
10 ' *****************
20 ' *               *
30 ' * REVERSE SCREEN *
40 ' *               *
50 ' *****************
60 GOTO 180
70 '  ---------------------------
80 '      ++ VARIABLES ++
90 '    REV$: TURNS REVERSE ON
100 '   FF$:  TURNS REVERSE OFF
110 '
120 ' ---------------------------

130 ' *** SUBROUTINE ***

140 REV$=CHR$(27)+CHR$(112)
150 FF$=CHR$(27)+CHR$(113)
160 RETURN

170 ' *** YOUR PROGRAM STARTS HERE ***

180 GOSUB 140
190 PRINT REV$;"THIS IS REVERSED"
200 PRINT FF$;"THIS IS NORMAL"
```

Number Input

WHAT IT DOES

Allows user to input only numbers.

Variables

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

How To Use The 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 an A$=INKEY$ 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 ' ****************
20 ' *              *
30 ' * NUMBER INPUT *
40 ' *              *
50 ' ****************
60 '  ----------------------
70 '    ++ VARIABLES ++
80 '    I:  NUMBER ENTERED
90 '    I$: STRING ENTERED
100 '
110 ' ----------------------
120 GOTO 230

130 ' *** SUBROUTINE ***

140 A$=INKEY$: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 ' *** 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 The 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 an A$=INKEY$ 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 ' ****************
20 ' *              *
30 ' * LETTER INPUT *
40 ' *              *
50 ' ****************
60 '  ----------------------
70 '     ++ VARIABLES ++
80 '    I$: STRING ENTERED
90 '
100 ' ----------------------
110 GOTO 210

120 ' *** SUBROUTINE ***
```

```
130 A$=INKEY$: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 ' *** YOUR PROGRAM STARTS HERE ***

210 GOSUB 130
```

Case Converter

WHAT IT DOES

Changes lowercase input to uppercase.

Variables

- I$: String entered.

How To Use The Subroutine

Many times our error traps in programs check to see if an acceptable key has been pressed. We may want the user to enter "Y" or "N" answers only, or our program will check a name or other entry against a list. However, the TRS-80 can produce uppercase and lowercase. "Y" does not equal "y" and "JONES" is completely different from "Jones" to the computer.

This subroutine will check each character entered, and if it is lowercase, convert it to uppercase. Only letters in the range "a" to "z" are affected. All other input is passed through unchanged.

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 lowercase a through z, change to uppercase.

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

Only user input needed.

RESULT

User alpha input is all uppercase.

```
10 ' ****************
20 ' *              *
30 ' * CASE CONVERT *
40 ' *              *
50 ' ****************
60 '  ----------------------
70 '    ++ VARIABLES ++
80 '    I$: STRING ENTERED
90 '
100 ' ----------------------
110 GOTO 210

120 ' *** SUBROUTINE ***

130 A$=INKEY$:IF A$="" GOTO 130
140 IF A$=CHR$(13) GOTO 190
150 A=ASC(A$):IF A>96 AND A<123 THEN A$=CHR$(A-32)
160 PRINT A$;
170 I$=I$+A$
180 GOTO 130
190 RETURN

200 ' *** 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 The 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 more letters, because it would be placed on an alphabetized list first. In computer terminology, we would say that: "computerization" < "contain" is a true statement. In making the comparision between strings, the computer 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 RAM 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 fewer 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

You should define NU, the number of items to be sorted, as well as supply the data for the array, US$(n).

RESULT

List is sorted alphabetically.

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

120 ' *** INITIALIZE ***

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

160 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***

350 GOSUB 170
```

Number Sort

WHAT IT DOES

Sorts group of numbers by size.

Variables

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

How To Use The 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 RAM 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 RAM 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

You should define NU, the number of items to be sorted, as well as supply the data for the array, US(n).

RESULT

List of numbers is sorted by size.

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

120 ' *** INITIALIZE ***

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

160 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***

350 GOSUB 170
```

Array Loader

WHAT IT DOES

Loads array with data.

Variables

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

How To Use The 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 that 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$(1,3). Then DTA$(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

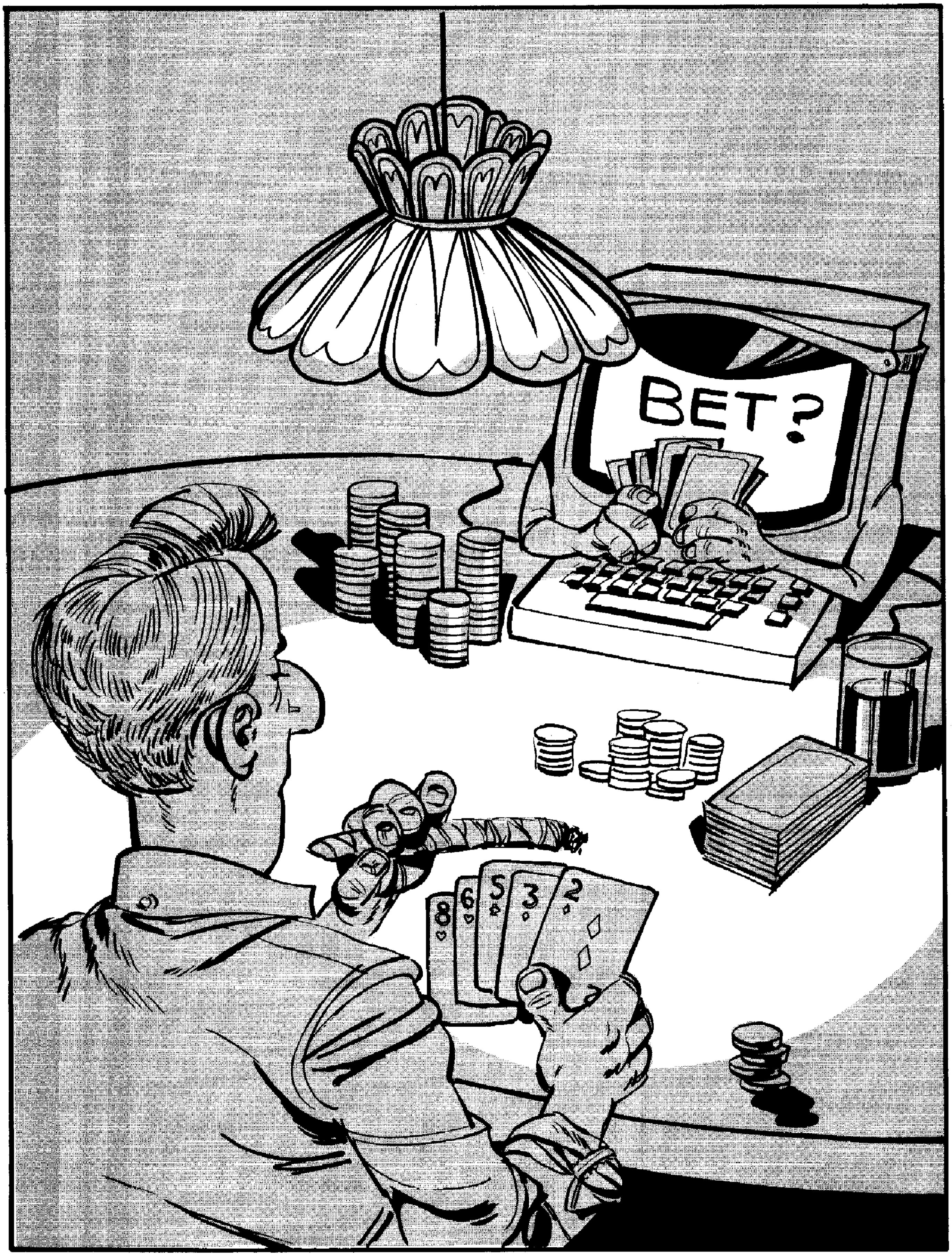
The number of rows, NROWS, and number of columns, NCOLUMNS, should be specified. Data can be supplied from DATA lines, or, better, read in from RAM or tape.

RESULT

Data list is loaded into array.

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

130 ' *** 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 ' *** 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 ' *** 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
```


BET?

Chapter Five

Game Routines

Games are probably among the most popular programming exercises for beginners and advanced users alike. On your first day with a computer you can easily learn to write a "Craps"-playing program. Somewhere short of your first year with the machine, you will probably investigate writing an arcade-type, 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.

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, because it allows you to define how many sides each die will have. Dungeons and Dragons (TM) 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.

One topic that needs to be addressed is the TRS-80's inability to generate true random numbers. The computer instead provides numbers from a pseudorandom list. This list is very long, and thus many numbers can be produced before it repeats. But the TRS-80 will always start at the same position on the list and thus generate the same random numbers each time.

We can compensate for this somewhat by "using up" a random number of the numbers from this list before we start to draw them for our own program. A convenient way to do this is to take the number of seconds currently on the real-time clock, and then take that many random numbers, assigning them to a dummy variable prior to beginning the actual program. The random seeding routine used in this book looks something like this and is taken from the one included in the handbook published by Radio Shack:

```
100 FOR N=1 TO VAL(RIGHT$(TIME$,2))
110 DU=RND(1)
120 NEXT N
```

All the routines that call for random numbers use this subroutine. Of course, the start points in the random sequence will only vary from 0 to 59. If you want additional randomness, use the subroutine which follows, which makes use of a random seed entered by the program operator.

Random Seed

WHAT IT DOES

Sets random start point.

Variables

- DU: Dummy variable to use up pseudorandom numbers.
- SEED: Random seed entered by user.

How To Use The Subroutine

Include in any program where many random numbers will be drawn or which will be run so many times that having only start points from 0 to 59 is not sufficient.

User can enter new random seed each time program is run.

Line-By-Line Description

Line 140: User enters seed number.

Lines 150 to 160: Seed calculated using seconds on TIME$.

Lines 170 to 190: Dummy random numbers expended.

Line 220: Access the subroutine.

You Supply

No changes needed.

RESULT

Random numbers reseeded.

```
10 ' ***************
20 ' *             *
30 ' * RANDOM SEED *
40 ' *             *
50 ' ***************
60 '  --------------------
70 '    ++ VARIABLES ++
80 '    DU:   DUMMY VARIABLE
90 '    SEED: USER INPUT
100 '
110 ' --------------------
120 GOTO 220

130 ' *** SUBROUTINE ***
```

```
140 INPUT "ENTER A NUMBER FROM 1 TO 100";NU$
150 SEED=VAL(NU$)
160 SEED=SEED*VAL(RIGHT$(TIME$,2))
170 FOR N=1 TO SEED
180 DU=RND(1)
190 NEXT N
200 RETURN

210 ' *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 140
```

Deal Cards

WHAT IT DOES

Shuffles and deals deck of cards.

Variables

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

How To Use The 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 to 360) using the 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 placing 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$(n) consists of a number, or face card name, plus the CHR$ value for the suit. This produces a full deck of 52 cards.

Line-By-Line Description

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

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

Lines 170 to 190: Set new random start point.

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 ' **************
20 ' *            *
30 ' * DEAL CARDS *
40 ' *            *
50 ' **************
60 '  ----------------------
70 '    ++ VARIABLES ++
80 '    DECK$(N): DECK
90 '    CARD$:    CARD DRAWN
100 '   DRAW:     RANDOM CARD
110 '
120 ' ----------------------
130 DIM DECK$(52)
140 DATA 156,157,158,159

150 ' *** READ SUITS ***

160 FOR N=1 TO 4:READ A:SUIT$(N)=CHR$(A):NEXT N

170 ' *** SET RANDOM START POINT ***

180 FOR N=1 TO VAL(RIGHT$(TIME$,2))
190 DU=RND(1)
200 NEXT N

200 ' *** ASSEMBLE DECK ***
```

(program continued)

```
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 ' *** 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.

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 Radio Shack BASIC.

The TRS-80 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. The simple command RND(N), is used where N is the upper limit. RND(7) would produce numbers from one to seven, for example. The Micro Executive Workstation will generate random numbers larger than zero and smaller than one. So, we might get .74329, .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

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

Lines 170 to 190: Set random start point.

Line 200: Choose a random number in the range 1 to DF, find 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 ' ****************
20 ' *              *
30 ' * RANDOM RANGE *
40 ' *              *
50 ' ****************
60 '  -----------------------
70 '     ++ VARIABLES ++
80 '
90 '     HIGH:    TOP OF RANGE
100 '    MINIMUM: BOTTOM OF RANGE
110 '    DF:      DIFFERENCE
120 '    NU:      NUMBER CHOSEN
130 '
140 ' -----------------------

150 ' *** INITIALIZE ***

160 HIGH=100:MINIMUM=15:DF=HIGH-MINIMUM+1
170 FOR N=1 TO VAL(RIGHT$(TIME$,2))
180 DU=RND(1)
190 NEXT N

195 ' *** SUBROUTINE ***

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

220 ' *** 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.

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

Line 130: Define array as "HEADS" and "TAILS."

Lines 140 to 160: Set random start point.

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

120 ' *** INITIALIZE ***
```

```
130 COIN$(1)="HEADS":COIN$(2)="TAILS"
140 FOR N=1 TO VAL(RIGHT$(TIME$,2))
150 DU=RND(1)
160 NEXT N
170 GOTO 220

175 ' *** SUBROUTINE ***

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

210 ' *** 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.

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

Line 130 to 150: Set random start point.

Line 170: 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 180: Make ROLL the total of the two dice.

Line 190: Return to main program.

Line 220: Define number of sides on dice.

Line 230: Access the subroutine.

Lines 240 to 260: 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 ' ********
20 ' *      *
30 ' * DICE *
40 ' *      *
50 ' ********
60 GOTO 270
70 '  ----------------------------
80 '      ++ VARIABLES ++
90 '    D1:   DIE #1 TOTAL
100 '   D2:   DIE #2 TOTAL
110 '   ROLL: TOTAL OF ROLL
120 ' ----------------------------

130 FOR N=1 TO VAL(RIGHT$(TIME$,2))
140 DU=RND(1)
150 NEXT N

160 ' *** SUBROUTINE ***

170 D1=INT(RND(1)*SIDES)+1:D2=INT(RND(1)*SIDES)+1
180 ROLL=D1+D2

200 ' *** YOUR PROGRAM STARTS HERE ***

210 PRINT
220 SIDES=6
230 GOSUB 150
240 PRINT" DIE #1:";D1
250 PRINT" DIE #2:";D2
260 PRINT "TOTAL :";
270 PRINT ROLL
```

Delay Loop

WHAT IT DOES

Delays loop changes in length.

Variables

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

How To Use The 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.

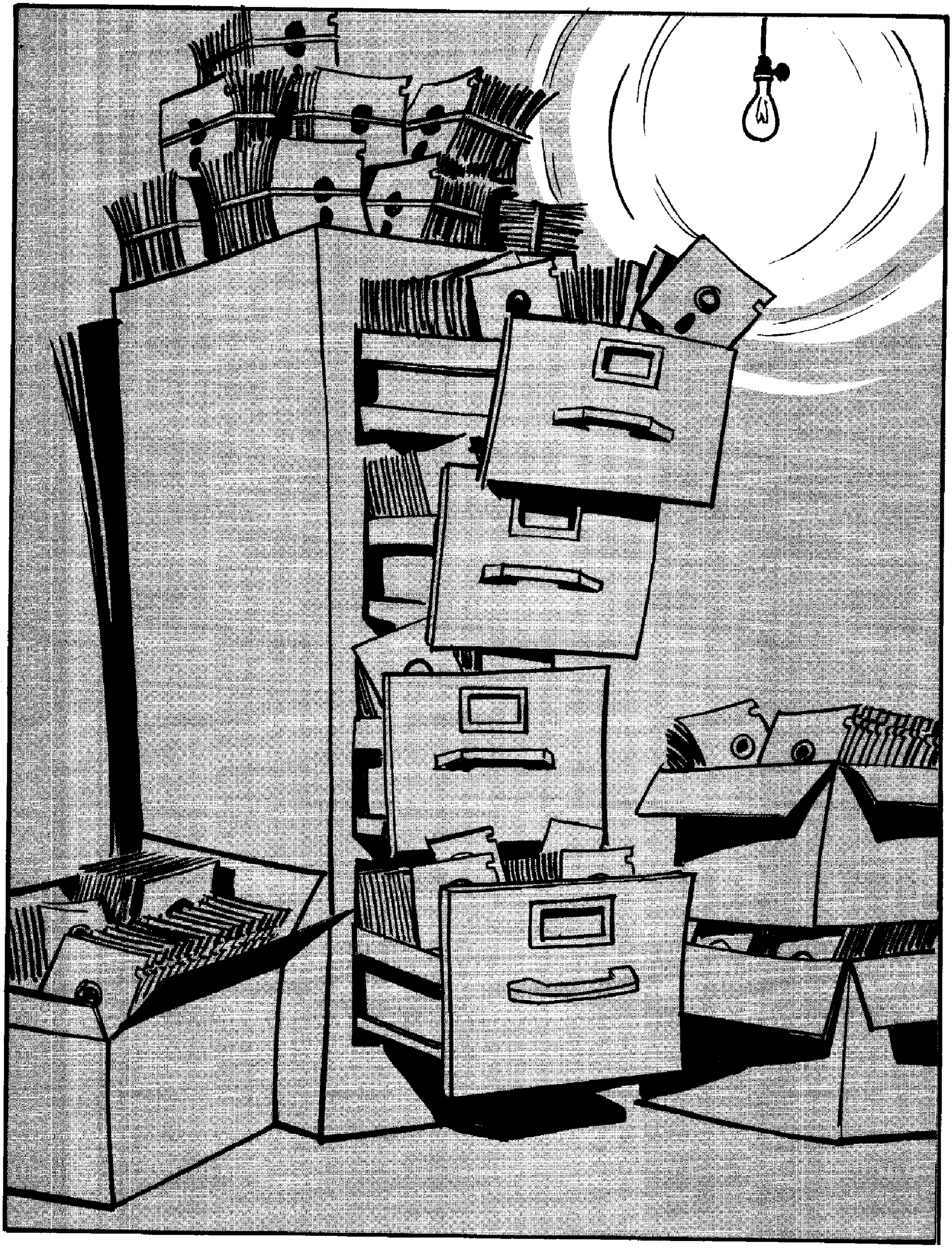
As decimal fractions approach 1.0, the amount of speedup each time will be smaller, producing a slower acceleration. Smaller fractions, such as .75 or even .50 will rev up the speed quite quickly.

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

130 ' *** INITIALIZE ***

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

170 ' *** SUBROUTINE ***

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

220 ' *** YOUR PROGRAM STARTS HERE ***

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



Chapter Six

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 .BA files can be loaded by the Portable Computer 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. If proper line numbers are in place, the computer can load .DO files just as if they were programs.

The use of data files is one of the basic tools of business and personal programming, because data files 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 every time 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.

The most commonly used file is the sequential file. Because this type is easiest to understand and use, sequential file systems are emphasized in this book.

The cassette recorder is a good analog to sequential files, i.e., serial files. A program is a sequential file, which is 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. Programs are stored in RAM the same way.

Sequential data files also operate this 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.

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

So, we might have the following collection of lines:

```
10 MAXFILES =2 '                       We want to have two channels available.
20 OPEN "FILE1" FOR INPUT AS 1 '       Source file.
30 OPEN "FILE2" FOR OUTPUT AS 2 '      New data file.
40 LINEINPUT #1, A$ '                  Load a line from source file.
50 PRINT #2,A$ '                       Print it to data file.
60 CLOSE '                             Close the files.
```

This would be a simple data file routine. The subroutines that follow show you how to write and load files to RAM, from RAM, to tape, and from tape. Examine them carefully until you know how data files operate.

Sequential File—Write to RAM

WHAT IT DOES

Writes a sequential data file to RAM.

Variables

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

How To Use The Subroutine

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

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

130 ' *** INITIALIZE ***

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

160 ' *** SUBROUTINE ***

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

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 170
```

Sequential File—Read From RAM

WHAT IT DOES

Reads a sequential data file from RAM.

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 The Subroutine

This subroutine provides a sample file reading routine that will take data that has been written to RAM 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 RAM. 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 substitue 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 substitue your file name for "filename" in line 160.

RESULT

Data file read from RAM.

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

```
130 ' *** INITIALIZE ***

140 MAXFILES=2:AD=10:GOTO 240

150 ' *** SUBROUTINE ***

160 OPEN "RAM:FILENAME" FOR INPUT AS 2
170 INPUT#2,NI
180 DIM DTA$(NI+AD)
190 FOR N=1 TO NI
200 INPUT#2,DTA$(N)
210 NEXT N
220 CLOSE 2:RETURN

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 160
```

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 The Subroutine

Sequential files are the most popular data files because they are easiest to understand. With the Portable Computer's 1500 baud tape storage speed, such files are even respectably fast.

However, this subroutine provides a sample sequential file writing routine that will take data that has been loaded into a string array, DTA$(n), and write it to tape. The same routine can be used with numeric arrays, simply by removing the variable type specifier, "$", 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 RAM 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 ' ******************
20 ' *                *
30 ' * SEQUENTIAL FILE *
40 ' *  WRITE TO TAPE  *
50 ' *                *
60 ' ******************
70 '  -----------------------------------
80 '      ++ VARIABLES ++
90 '   NI:      NUMBER OF ITEMS IN FILE
100 '  DTA$(N): ARRAY STORING FILE
110 '
120 ' -----------------------------------

130 ' *** INITIALIZE ***

140 MAXFILES=2:NI=10
150 DIM DTA$(NI)
160 GOTO 250

170 ' *** SUBROUTINE ***

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

240 ' *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 180
```

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 The Subroutine

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 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 substitue 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 substitue your file name for "filename" in line 160.

RESULT

Data file read from tape.

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

140 ' *** INITIALIZE ***

150 MAXFILES=2:AD=10
160 GOTO 250

170 ' *** SUBROUTINE ***

180 OPEN "CAS:FILENAME" FOR INPUT AS 2
190 INPUT#2,NI
200 DIM DTA$(NI+AD)
210 FOR N=1 TO NI
220 INPUT#2,DTA$(N)
230 NEXT N
240 CLOSE 2:RETURN


245 ' *** YOUR PROGRAM STARTS HERE ***

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

LAST FEDERAL
BANK

Chapter Seven

Business and Financial Subroutines

The TRS-80 Portable Computer, touted as the "Micro Executive Workstation," is most popular among businesspeople. Much of the early software was aimed at this market. 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 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 4.

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 The 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

Line 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), and 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 ' ***************
20 ' *             *
30 ' * LOAN AMOUNT *
40 ' *             *
50 ' ***************
60 '  ---------------------------
70 '      ++ VARIABLES ++
80 '    RATE:    INTEREST RATE
90 '    LOAN:    AMOUNT OF LOAN
100 '   NUMBER:  MONTHS OF LOAN
110 '   PAYMENT: MONTHLY PAYMENT
120 '
130 ' ---------------------------

140 ' *** INITIALIZE ***

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

190 ' *** SUBROUTINE ***

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

240 ' *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 200
260 PRINT LOAN
```

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 The 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 effect on your monthly payment until a satisfactory amount has been worked out.

The subroutine would also be valuable for those considering consolidating a number of debts. Add up the current 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

Line 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), and 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 ' ******************
20 ' *                *
30 ' * PAYMENT AMOUNT *
40 ' *                *
50 ' ******************
60 ' ----------------------------
70 '        ++ VARIABLES ++
80 '    RATE:    INTEREST RATE
90 '    LOAN:    AMOUNT OF LOAN
100 '   NUMBER:  MONTHS OF LOAN
110 '   PAYMENT: MONTHLY PAYMENT
120 '
130 ' ----------------------------
```

```
140 ' *** INITIALIZE ***

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

190 ' *** 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 ' *** 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 The 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

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

160 ' *** INITIALIZE ***

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

210 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***

280 GOSUB 220
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 The 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), and 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 ' *****************
20 ' *               *
30 ' * YEARS TO REACH *
40 ' * DESIRED VALUE  *
50 ' *               *
60 ' *****************
70 '  -------------------------------
80 '       ++ VARIABLES ++
90 '      RATE:   INTEREST RATE
100 '     YEARS:  YEARS COMPOUNDED
110 '     FUTURE: FUTURE VALUE DESIRED
120 '     AMOUNT: AMOUNT TO BE COMPOUNDED
130 '
140 ' -------------------------------

150 ' *** INITIALIZE ***

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

210 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***

290 GOSUB 220
300 CLS
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 The 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), and YEARS (number of years to be compounded). The subroutine will return FUTURE, or value of the compounded investment.

RESULT

Compound interest calculated.

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

140 ' *** INITIALIZE ***

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

200 ' *** SUBROUTINE ***

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

250 ' *** YOUR PROGRAM STARTS HERE ***

260 GOSUB 210
270 CLS:
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 The 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

Line 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) and PERIODS (number of compounding periods). The subroutine will return RATE, the interest rate.

RESULT

Interest rate calculated.

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

```
140 ' *** INITIALIZE ***

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

200 ' *** SUBROUTINE ***

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

240 ' *** YOUR PROGRAM STARTS HERE ***

250 GOSUB 210
260 CLS
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 The 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 SHIFT 6 on the keyboard.

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

130 ' *** INITIALIZE ***

140 A=55.345
150 GOTO 330

160 ' *** 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 ' *** 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 The 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 ' ***************
20 ' *             *
30 ' * TEMPERATURE *
40 ' *             *
50 ' ***************

60 ' *** INITIALIZE ***

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

140 ' *** 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 ' *** 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$="C" THEN GOTO 300
280 PRINT "WRONG FORMAT."
290 GOTO 220
300 IF A$="F" THEN GOSUB 170
310 IF A$="C" THEN GOSUB 150
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 The 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 12/03/47 or 12/03/1947. 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 ' *****************
20 ' *               *
30 ' * DATE FORMATTER *
40 ' *               *
50 ' *****************
60 GOTO 360
70 '  --------------------------
80 '      ++ VARIABLES ++
90 '       DTE$:  DATE
100 '      MNTH$: MONTHS
110 '      DAY$:  DAY
120 '      YEAR$: YEAR
130 '
140 ' --------------------------

150 ' *** 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$)
280 IF YEAR/4<>INT(YEAR/4)GOTO 310
290 IF MNTH=2 AND DAY>29 GOTO 200
300 GOTO 320
310 IF MNTH=2 AND DAY>28 GOTO 200
320 IF DAY<10 THEN DAY$="O"+RIGHT$(DAY$,1)
330 DTE$=MNTH$+"/"+DAY$+"/"+YEAR$
340 RETURN

350 ' *** YOUR PROGRAM STARTS HERE ***

360 CLS:
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 The 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. Users choose one of up to nine selections 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 160 to 200: Labels for the menu choices.

Line 210: Prompt user choice.

Line 220: Wait for user input.

Lines 230 to 240: If entry is less than 1 or larger than the number of choices available, go back and continue waiting.

Line 250: Access subroutine specified by user, at Lines 1000,2000,3000 or 4000.

You Supply

You should define NC to equal the number of menu choices. You will need to write subroutines to accomplish your various tasks, using line 250 as a model to direct control.

RESULT

Operator can select from list of menu choices.

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

60 ' *** INITIALIZE ***

70 NC=4
80 GOTO 280
90 '  ---------------------------
100 '      ++ VARIABLES ++
110 '      NC: NUMBER OF MENU CHOICES
120 '
130 '  ---------------------------
```

```
140 ' *** SUBROUTINE ***

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

270 ' *** YOUR PROGRAM STARTS HERE ***

280 PRINT
290 GOSUB 150
300 END

990 ' *** FIRST SUBROUTINE ***
1000 RETURN

1990 ' *** SECOND SUBROUTINE ***
2000 RETURN

2990 ' *** THIRD SUBROUTINE ***
3000 RETURN

3990 ' *** 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 The 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

Line 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 ' **************
20 ' *            *
30 ' * TIME ADDER *
40 ' *            *
50 ' **************
60 '  ---------------------------
70 '    ++ VARIABLES ++
80 '  TM:   TOTAL MINUTES
90 '  TS:   TOTAL SECONDS
100 ' TH:   TOTAL HOURS
110 ' MIN:  MINUTES TO BE ADDED
120 ' HOUR: HOURS TO BE ADDED
130 ' SECS: SECONDS TO BE ADDED
140 '
150 ' ---------------------------

155 ' *** INITIALIZE ***

160 TM=54
170 TH=40
180 TS=30
190 MIN=30
200 HOUR=2
210 SECS=30
220 GOTO 340

230 ' *** 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 ' *** YOUR PROGRAM STARTS HERE ***

340 GOSUB 240
350 CLS
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 The 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 ' *******
20 ' *     *
30 ' * MPG *
40 ' *     *
50 ' *******
60 '  ---------------------------
70 ' ++ VARIABLES ++
80 '  BEGN:   STARTING ODOMETER
90 '  ODOM:   CURRENT ODOMETER
100 ' GALLNS: GALLONS GAS USED
110 '
120 ' ---------------------------

130 ' *** INITIALIZE ***

140 ODOM=36420
150 BEGN=36001
160 GALLNS=13.8
170 GOTO 230

180 ' *** SUBROUTINE ***

190 MPG=(ODOM-BEGN)/GALLNS
200 MPG=INT(MPG*10+.5)/10
210 RETURN

220 ' *** YOUR PROGRAM BEGINS HERE ***

230 GOSUB 190
240 CLS
250 PRINT "MPG=";MPG
```

LOAD
RESTORE
NEXT
PEEK
THEN
RUN
DIM
CHR$()
PRINT
POKE
GOTO
LIST
RETURN
STR$()
INPUT
GOSUB

Chapter Eight

Add New Capabilities

When compared to the BASICs supplied with other computers in the same price range, the BASIC included in the Portable Computer stacks up very well indeed. There are many statements and commands that were missing in earlier Radio Shack offerings. The BASIC is more similar to that used in the IBM Personal Computer, and other state-of-the-art micros than it is to the interpreter originally supplied with the TRS-80 Model I and III computers.

There are many important capabilities—such as MOD, INSTR, and MID$= that are not featured in such top selling competing brands as the Commodore VIC-20 and Commodore 64.

Even so, there is room to add some capabilities to your Micro Executive Workstation through this collection of subroutines. One will allow you to remove remarks that have been put in a program, thus making the program more compact, and palatable to the RAM-hungry Portable Computer. Another program lets you center TABBed prompts automatically.

There are several subroutines useful for processing TEXT files and programs which have been saved in .DO form. You may insert strings under program control, perform global search and replaces, and count the number of words in a document.

Finally, we have included two subroutines which provide a quick way of determining the CHR$ code for any key pressed and to swap variables within a program.

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 The 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 TRS-80 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$, the string to be inserted, SUB$, and the position where it will be put, PLACE.

RESULT

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

```
10 ' *****************
20 ' *               *
30 ' * INSERT STRING$ *
40 ' *               *
50 ' *****************
60 '  --------------------------
70 '  ++ VARIABLES ++
80 '   TARGT$: MAIN STRING
90 '   SUB$:   STRING TO BE INSERTED
100 ' PLACE:  POSITION TO PUT SUB$
110 '
120 ' --------------------------

130 ' *** INITIALIZE ***

140 SUB$="TEST "
150 TARGT$="TARGET STRING LETTERS"
160 PLACE=12
170 GOTO 240

180 ' *** SUBROUTINE ***
```

```
190 L$=LEFT$(TARGT$,PLACE)
200 R$=MID$(TARGT$,PLACE+1)
210 TARGT$=L$+SUB$+R$
220 RETURN

230 ' *** YOUR PROGRAM STARTS HERE ***

240 GOSUB 190
250 PRINT TARGT$
```

CHR$ Value

WHAT IT DOES

Returns CHR$ code for any key.

Variables

- A: CHR$ value of last key pressed.

How To Use The Subroutine

When printing graphics or alphanumerics via PRINT CHR$(n), it is necessary to know the CHR$ 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 the computer, 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. Radio Shack departs somewhat from this code for the Portable Computer, especially when using the graphics characters. However, the standard alphanumeric symbols are accurately portrayed with the CHR$(n) statement. That is, PRINT CHR$(65) will produce an uppercase "A" in Radio Shack BASIC, just as 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 ' **************
20 ' *            *
30 ' * CHR$ VALUE *
40 ' *            *
50 ' **************
60 '  ---------------------------
70 '       ++ VARIABLES ++
80 '
90 '     A: CHR$ VALUE OF KEY
100 '
110 ' ---------------------------
120 GOTO 200

130 ' *** SUBROUTINE ***

140 A$=INKEY$:IF A$="" GOTO 130
150 A=ASC(A$)
160 PRINT"CHR$ VALUE OF KEY";A$
170 PRINT"IS :";A
180 RETURN

190 ' *** YOUR PROGRAM STARTS HERE ***

200 GOSUB 140
210 GOTO 200
```

Swap

WHAT IT DOES

Simulates SWAP function found in some other BASICs.

Variables

- A$: First variable
- B$: Second variable.

How To Use The Subroutine

Exchanging the value of one variable for that of another cannot be done in one step in Radio Shack's BASIC. 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 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 ' ********
20 ' *      *
30 ' * SWAP *
40 ' *      *
50 ' ********
60 '  ---------------------------
70 '      ++ VARIABLES ++
80 '     A$: FIRST VARIABLE
90 '     B$: SECOND VARIABLE
100 '
110 ' ---------------------------

120 ' *** INITIALIZE ***

130 A$="FIRST"
140 B$="SECOND"
150 GOTO 220
```

```
160 ' *** SUBROUTINE ***

170 DUMMY$=A$
180 A$=B$
190 B$=DUMMY$
200 RETURN

210 ' *** YOUR PROGRAM STARTS HERE ***

220 PRINT "A$=";A$;"B$=";B$
230 GOSUB 170
240 PRINT "A$=";A$;"B$=";B$
```

Remove Remarks

WHAT IT DOES

Removes remarks from programs.

Variables

- P: Position to begin search
- R: Position REMark found
- Q1-Q2: Position of quotes
- A$: New line built by program.

How To Use The Subroutine

You should store the program that you want to remove remarks from in .DO form. This is accomplished from BASIC by appending the .DO extension to the filename. Your new program, without remarks, will also be stored as a .DO file. You can load this into BASIC and then SAVE as a .BA file that you can load and run.

Line-By-Line Description

Line 160: CLEAR 5000 bytes of string space.

Line 180: Clear screen.

Lines 190 to 200: Build input/output file names.

Lines 230 to 240: OPEN files.

Line 260: If end of program, then end.

Line 270: Read in a program line.

Line 280: Set start position of search at 1.

Line 300: Look for " ' ".

Line 320: Look for "REM".

Lines 350 to 380: Look for quotation marks.

Lines 410 to 440: Parse program line.

Lines 510 to 530: PRINT new line to file, screen, and return for more.

Line 540: Close file.

You Supply

User supplies program to be processed.

RESULT

Remarks removed from program.

```
10 ' ******************
20 ' *                *
30 ' * REMOVE REMARKS *
50 ' *                *
60 ' ******************
70 '  --------------------------------
80 '      ++ VARIABLES ++
90 '
100 '   P:      Position to begin search
110 '   R:      Position REMark found
120 '   Q1-Q2:  Position of quotes
130 '   A$:     New line built by program
140 '
150 ' --------------------------------
160 CLEAR 5000

170 ' *** INPUT PROGRAM TO PROCESS ***

180 CLS:PRINT:PRINT
190 PRINT"Enter name of program to have REMARKS removed:"
200 LINEINPUT F$
210 F1$=LEFT$(F$,4)+"RM.DO"

220 ' *** OPEN FILES ***

230 OPEN F$ FOR INPUT AS 1
240 OPEN F1$ FOR OUTPUT AS 2
```

```
250 ' *** READ IN A LINE ***

260 IF EOF(1) GOTO 540
270 LINE INPUT #1,A$
280 P=1

290 ' *** CHECK FOR ' OR REM ***

300 R=INSTR(P,A$,"'")
310 IF R<>0 GOTO 350
320 R=INSTR(P,A$,"REM")
330 IF R=0 GOTO 510

340 ' *** REMARK FOUND ***

350 Q1=INSTR(P,A$,CHR$(34)):IF Q1=0 GOTO 410
360 Q1=Q1+1
370 Q2=INSTR(Q1,A$,CHR$(34))
380 IF R<Q1 OR R>Q2 GOTO 410
390 P=Q2+1
400 GOTO 300
410 A$=LEFT$(A$,R-1)
420 FOR N=1 TO LEN(A$)
430 B$=MID$(A$,N,1)
440 IF ASC(B$)<48 OR ASC(B$)>57 GOTO 460
450 NEXT N
460 T$=MID$(A$,N)
470 IF T$="" GOTO 260
480 IF T$=" " GOTO 260
490 IF RIGHT$(A$,1)=":" THEN A$=LEFT$(A$,(LEN(A$)-1))

500 ' *** PRINT TO FILE ***

510 PRINT A$
520 PRINT#2,A$
530 GOTO 260
540 CLOSE
```

Center TABS

WHAT IT DOES

Centers TABs in a program.

Variables

- C: Location of TAB marker in line
- C1: Location of quote
- A$: Program line being processed.

How To Use The Subroutine

If you want to have your prompts neatly centered on the screen of your TRS-80, simply substitute the statement "TAB(T)" in your program at any place where you wish to have the prompt that follows centered.

This subroutine will go through your program and calculate the length of the prompt, then rewrite the line so that a new TAB value will be substituted for T. The resulting prompt will be centered.

You should store the program that you want to process in .DO form. This is accomplished from BASIC by appending the .DO extension to the filename. Your new program, with centered TABs, will also be stored as a .DO file. You can load this into BASIC and then SAVE as a .BA file that you can load and run.

Line-By-Line Description

Line 130: CLEAR 1000 bytes of string space.

Line 140: Set maximum number of files at 2.

Line 150: Clear screen.

Lines 200 to 230: Enter names of input and output files.

Lines 240 to 250: OPEN files.

Line 270: Load a program line.

Line 280: Look for TAB(T).

Line 300: Look for quote.

Lines 310 to 320: Parse line.

Line 330: Calculate centering.

Line 340: Convert centering to string.

Line 380: Print line to RAM file.

Line 400: If end of the file, END.

Line 410: Go back for more.

Line 420: CLOSE.

You Supply

User supplies program with TAB(T)'s inserted.

RESULT

Program prompts centered.

```
10 ' ******************
20 ' *                *
30 ' *   CENTER TABS  *
40 ' *                *
50 ' ******************
60 '  --------------------------------
70 '      ++ VARIABLES ++
80 '    C:  Location of TAB marker in line
90 '    C1: Location of quote
100 '   A$: Program line being processed.
110 '
120 ' --------------------------------
130 CLEAR 1000
140 MAXFILES=2
150 CLS:PRINT:PRINT
160 PRINT
170 PRINT

180 ' *** Enter Name of File to Process ***

190 CLS:PRINT:PRINT
200 PRINT "ENTER PROGRAM WITH TABS TO BE CENTERED:"
210 LINEINPUT F$
220 PRINT "ENTER NAME OF OUTPUT FILE :"
230 LINEINPUT F2$
240 OPEN F$ FOR INPUT AS 1
250 OPEN F2$ FOR OUTPUT AS 2

260 ' *** Load a Line ***

270 LINEINPUT#1,A$
280 C=INSTR(A$,"TAB(T)")
290 IF C=0 GOTO 380
300 C1=INSTR(C,A$,CHR$(34))+1
310 B$=MID$(A$,C1)
320 B$=LEFT$(B$,INSTR(B$,CHR$(34))-1)
330 D=INT((40-LEN(B$))/2)
340 D$=MID$(STR$(D),2)
350 A$=LEFT$(A$,C+3)+D$+MID$(A$,C+5)
360 GOTO 280
```

```
370 ' *** Print to RAM ***

380 PRINT#2,A$
390 PRINT A$
400 IF EOF(1) GOTO 420
410 GOTO 270
420 CLOSE
```

Count Words

WHAT IT DOES

Counts words in TEXT file.

Variables

- CHAR: Number of characters in file
- C$: Current character
- L$: Last character.

How To Use The Subroutine

This subroutine will count the number of words in any TEXT file or any other .DO file, such as programs that have been stored in that form. It will also provide the average word length and number of standard five character words in the file.

Line-By-Line Description

Line 120: Clear 4000 bytes of string space, and define variables as integers for extra speed.

Line 130: Set maximum files at 1, and clear screen.

Lines 150 to 160: Enter name of file to process, and OPEN it.

Line 170: Load a line from the file.

Line 190: Add length of characters in line to total.

Line 210: Look at each character in the line.

Line 220: Take one character from the line, determined by FOR-NEXT counter, N.

Line 230: If the character is a space, and the last one was NOT a space, then a word has ended.

Line 240: Make the last character equal current character.

Line 260: If at end of file, END.

Lines 290-350: Print results of count.

You Supply

User supplies text file to count.

RESULT

Number of words in file compiled.

```
10 ' ******************
20 ' *                *
30 ' *   COUNT WORDS  *
40 ' *                *
50 ' ******************
60 '  --------------------------------
70 '       ++ VARIABLES ++
80 '     CHAR: NUMBER OF CHARACTERS IN FILE
90 '     C$:   CURRENT CHARACTER
100 '    L$:  LAST CHARACTER
110 ' --------------------------------
120 CLEAR 4000:DEFINT A-Z
130 MAXFILES=1:CLS:PRINT:PRINT

140 ' *** ACCESS RAM FILE ***

150 PRINT"Enter name of file to count:"
160 LINEINPUT F$:OPEN F$ FOR INPUT AS 1
170 LINEINPUT #1,A$

180 ' *** ADD TO TOTAL ***

190 CHAR=CHAR+LEN(A$)

200 ' *** LOOK AT EACH CHARACTER ***

210 FOR N=1 TO LEN(A$)
220 C$=MID$(A$,N,1)
230 IF C$=CHR$(32) AND L$<>CHR$(32) THEN CU=CU+1
240 L$=C$
250 NEXT N
260 IF EOF(1) THEN GOTO 290
270 GOTO 170

280 ' *** PRINT RESULTS ***
```

```
290 CLS:PRINT:PRINT
300 PRINT"WORDS:";CU
310 AW=CHAR/CU
320 PRINT"AVG. LENGTH:";AW
330 SW=CHAR/5
340 PRINT"NO. 5-CHAR. WORDS:":SW
350 CLOSE
```

Global Search

WHAT IT DOES

Performs global search and replaces.

Variables

- L$: Right portion of program line
- R$: Left portion of program line
- RE$: Replacement string.

How To Use The Subroutine

TEXT's FIND feature does not allow you to do global search and replacements in a file without repeatedly entering commands. This subroutine will go through a file and replace any string with any other string. It will do this globally, or ask you at each occurrence whether or not you want to replace this example. Thus, you can replace some occurrences of a word, but leave others alone.

The program will also work on BASIC programs. You should store the program that you want to do the global search and replace function on in .DO form. This is accomplished from BASIC by appending the .DO extension to the filename. Your new program, with changes, will also be stored as a .DO file. You can load this into BASIC and then SAVE as a .BA file that you can load and run.

Line-By-Line Description

Line 150: Clear 5000 bytes of string space and set maximum files to 2.

Line 170: Clear screen.

Lines 180 to 190: Enter name of file to be searched.

Lines 200 to 220: Enter string to search for.

Lines 230 to 250: Enter string to replace with.

Lines 260 to 300: Replace all?

Line 320: Set up output file name.

Lines 340 to 350: OPEN RAM files.

Line 360: If end of FILE then END processing.

Line 380: Load line of file.

Lines 390 to 420: Look for search string.

Line 430: Take left portion of line.

Line 440: Determine length of search string.

Line 450: Take right portion of line.

Line 460: Add spaces where necessary to fill out line.

Lines 490 to 560: Ask if replace it.

Line 570: Set new search start point.

Line 580: Go back search same line for more.

Line 590: Construct new line.

Line 630: Print new file to RAM.

Line 670: CLOSE files.

You Supply

User supplies TEXT file to be processed. Will also work on programs saved in .DO format.

RESULT

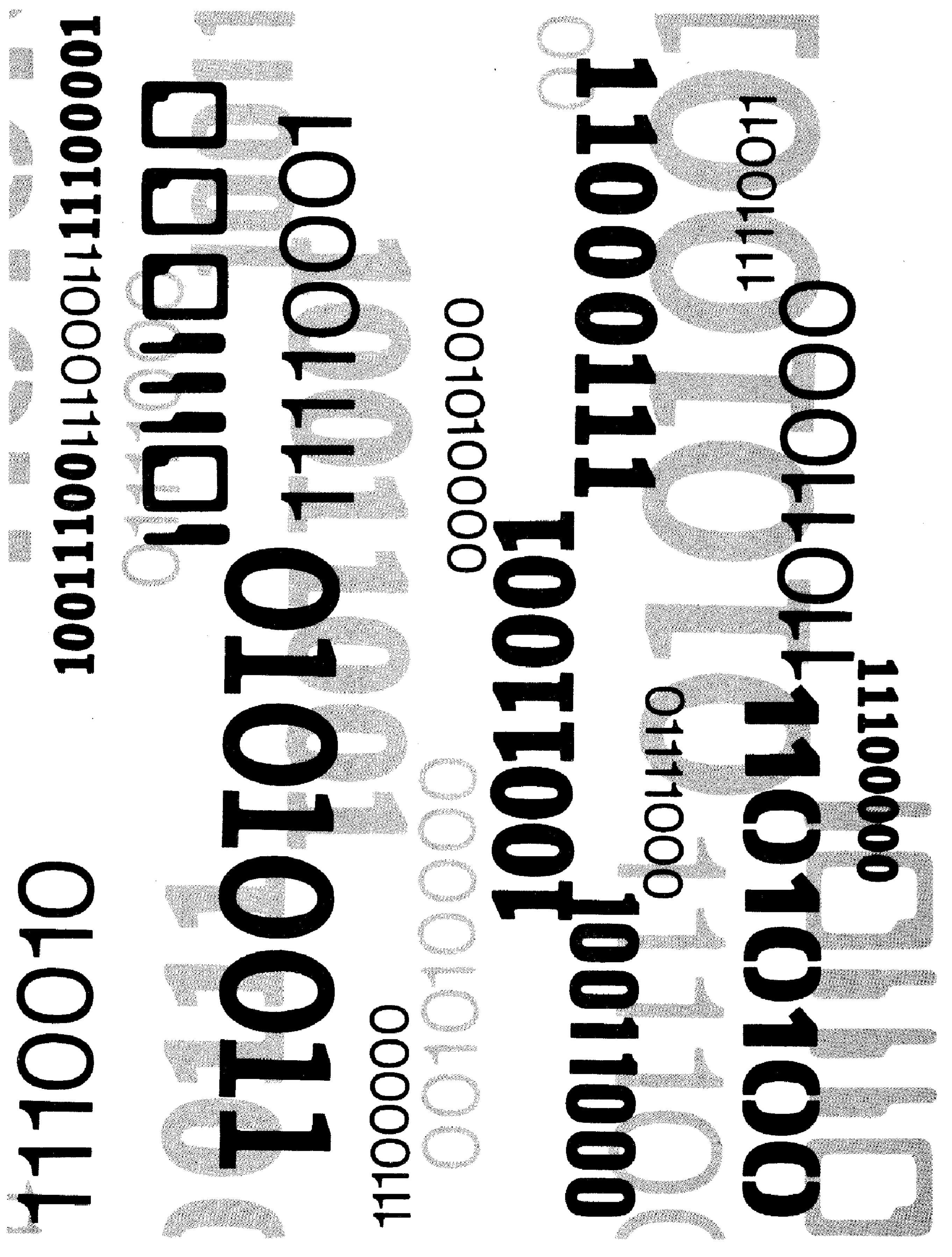
Global search and replace carried out.

```
10 ' ******************
20 ' *                *
30 ' *  GLOBAL SEARCH  *
40 ' *     UTILITY     *
50 ' *                *
60 ' ******************
70 '  --------------------------------
80 '       ++ VARIABLES ++
90 '    R$:  RIGHT PORTION OF PROGRAM LINE
100 '   L$:  LEFT PORTION OF PROGRAM LINE
110 '   RE$: REPLACEMENT STRING
120 '   S$:  STRING BEING SEARCHED FOR
130 '
140 ' --------------------------------
150 CLEAR 5000:MAXFILES=2
```

```
160 ' *** DEFINE FILES ***

170 CLS:PRINT:PRINT
180 PRINT"Enter name of program to be processed :":PRINT
190 LINEINPUT F$
200 CLS:PRINT:PRINT
210 PRINT"Enter string to search for :":PRINT
220 LINEINPUT S$
230 CLS:PRINT:PRINT
240 PRINT"Enter string to replace with :":PRINT
250 LINEINPUT RE$
260 CLS:PRINT:PRINT
270 PRINT"Do you want to choose whether to replace each?"
280 PRINT"(Y/N)"
290 CH$=INKEY$:IF CH$="" GOTO 290
300 IF CH$="Y" OR CH$="y" THEN CH=1
310 CLS
320 F1$=LEFT$(F$,4)+"GB.DO"

330 ' *** OPEN RAM FILES ***

340 OPEN F$ FOR INPUT AS 1
350 OPEN F1$ FOR OUTPUT AS 2
360 IF EOF(1) GOTO 660

370 ' *** LOAD LINE ***

380 LINEINPUT #1,A$
390 IF CH=1 THEN CLS
400 P=1
410 R=INSTR(P,A$,S$)
420 IF R=0 GOTO 630
430 L$=LEFT$(A$,R-1)
440 E=LEN(S$)
450 R$=MID$(A$,R+E)
460 Y$=STRING$(LEN(RE$),32)
470 IF CH=0 THEN GOTO 590

480 ' *** REPLACE? ***
```

(program continued)

```
490 B$=INKEY$
500 PRINT @0,L$;Y$;R$
510 FOR N1=1 TO 50:NEXT
520 PRINT @,L$;RE$;R$;
530 FOR N1=1 TO 50:NEXT
540 PRINT @ 100,"Replace it? (Y/N)"
550 IF B$="" GOTO 490
560 IF B$="Y" OR B$="y" GOTO 590
570 P=INSTR(P,A$,S$)+LEN(S$)-1
580 GOTO 410
590 A$=L$+RE$+R$
600 P=INSTR(P,A$,RE$)+LEN(RE$)-1
610 GOTO 410

620 ' *** PRINT FILE ***

630 PRINT#2,A$
640 IF CH=0 THEN PRINT A$
650 GOTO 360
660 PRINT#2,A$
670 CLOSE
```



Chapter Nine

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.

This book doesn't purport to explain assembly or machine language. However, these routines will be helpful for those who are just beginning to explore this area, as well as those who want to do some sophisticated, memory-efficient BASIC programming that uses various "spare" memory locations to store information (beyond the reach of the casual intruder). Therefore, there won't be a lengthy discussion of how to use these subroutines. If you don't know how already, they probably wouldn't be of much use to you.

As you know, each memory location stores a single, 8-bit byte. The binary numbers look something like this:

```
1Ø11Ø111
```

In many cases, the value of this whole byte is of use to us. Using a full byte allows us to have a total of 256 different "states" in that location and, therefore, 256 different characters or conditions.

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.

Boolean math is a way of performing certain bit-level operations. For example, when two bytes are compared using the OR operator, the result will be a 1 whenever that bit in either byte is a 1.

For example:

Original byte:	1Ø11Ø11Ø OR
Comparison byte:	Ø11ØØØ11
Result:	1111Ø111

AND will produce a 1 when both are 1, as in the example that follows:

Original byte:	1Ø11Ø11Ø AND
Comparison byte:	Ø11ØØØ11
Result:	ØØ1ØØØ1Ø

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 The Subroutine

You can get maximum mileage from your computer's RAM locations by using many for multiple purposes. A given location 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 SHIFT 6 key.

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 240: Print results.

You Supply

Define BIT as the bit, 1-8, that you want to examine and ADDRESS as the memory location to be PEEKed. V will indicate whether the bit is on or off by equaling either 1 or 0.

RESULT

Status of bit displayed.

```
10 ' ************
20 ' *          *
30 ' * PEEK BIT *
40 ' *          *
50 ' ************

60 ' *** INITIALIZE ***

70 ADDRESS=36879
80 BIT=3
90 GOTO 220
100 ' ----------------------------
110 '      ++ VARIABLES ++
120 '   ADDRESS: LOCATION TO PEEK
130 '   BIT:     BIT TO EXAMINE
140 '   V:       VALUE OF BIT
150 '
160 ' ----------------------------

170 ' *** SUBROUTINE ***

180 P=BIT-1
190 V=(PEEK(ADDRESS)AND(2^P))/(2^P)
200 RETURN

210 ' *** YOUR PROGRAM STARTS HERE ***

220 GOSUB 180
230 CLS:
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 The 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 SHIFT 6 key. 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 deliberatly 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

You must define ADDRESS as the memory location, in decimal, that you want to PEEK. The subroutine returns BIT$, which is a representation of all the bits within that byte.

RESULT

All bits within a byte are displayed.

```
10 ' ****************
20 ' *              *
30 ' * BIT DISPLAYER *
40 ' *              *
50 ' ****************

60 ' *** INITIALIZE ***
```

```
70 ADDRESS=36879
80 GOTO 260
90 '  --------------------------
100 '       ++ VARIABLES ++
110 '      ADDRESS: MEMORY BYTE
120 '                  TO DISPLAY
130 '      BIT$:    BIT PATTERN
140 '
150 ' --------------------------

160 ' *** 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 ' *** 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 The 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 Portable Computer.

NOTE: The caret symbol (^) indicates the SHIFT 6 key.

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

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1-8, that you want changed to a value of one.

RESULT

Bit within a byte is changed to one.

```
10 ' **************
20 ' *            *
30 ' * BIT TO ONE *
40 ' *            *
50 ' **************

60 ' *** INITIALIZE ***

70 ADDRESS=36878
80 BIT=3
90 GOTO 200
100 ' ----------------------------
110 '      ++ VARIABLES ++
120 '   ADDRESS: LOCATION TO POKE
130 '   BIT:     BIT TO CHANGE TO ONE
140 '
150 ' ----------------------------

160 ' *** SUBROUTINE ***

170 POKE ADDRESS,PEEK(ADDRESS)OR(2 BIT)
180 RETURN

190 ' *** 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 The 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 TRS-80.

NOTE: The caret symbol (^) indicates the SHIFT 6 key.

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

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1-8, that you want changed to a value of zero.

RESULT

Bit within a byte is changed to zero.

```
10 ' ***************
20 ' *             *
30 ' * BIT TO ZERO *
40 ' *             *
50 ' ***************
```

```
60 ' *** INITIALIZE ***

70 ADDRESS=36879
80 BIT=3
90 GOTO 200
100 ' ---------------------------
110 '      ++ VARIABLES ++
120 '    ADDRESS: LOCATION TO POKE
130 '    BIT:     BIT TO CHANGE TO ZERO
140 '
150 ' ---------------------------

160 ' *** SUBROUTINE ***

170 POKE ADDRESS,PEEK(ADDRESS)AND(255-(2^BIT))
180 RETURN

190 ' *** 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 The 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 TRS-80. Using this subroutine, you do not need 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 SHIFT 6 key.

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

You must define ADDRESS as the memory location, in decimal, that you want to POKE. BIT should be given the value of the bit, 1-8, that you want changed to reverse in value.

RESULT

Bit within a byte is reversed.

```
10 ' ***************
20 ' *             *
30 ' * REVERSE BIT *
40 ' *             *
50 ' ***************

60 ' *** INITIALIZE ***

70 ADDRESS=36878
80 BIT=3
90 GOTO 210
100 ' ----------------------------
110 '      ++ VARIABLES ++
120 '    ADDRESS: LOCATION TO POKE
130 '    BIT:     BIT TO REVERSE
140 '
150 ' ----------------------------

160 ' *** 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 ' *** 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 The 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 SHIFT 6 key.

Line-By-Line Description

Lines 150 to 160: Look at each binary character, and raise any 1's to the power of two 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

You must enter the binary number to be converted.

RESULT

Binary number converted to decimal.

```
10 ' *****************
20 ' *               *
30 ' * BINARY/DECIMAL *
40 ' *               *
50 ' *****************

60 ' *** INITIALIZE ***

70 GOTO 200
80 '  ---------------------------
90 '       ++ VARIABLES ++
100 '  A$ : BINARY NUMBER IN STRING FORM.
110 '  A:   DECIMAL EQUIVALENT
120 '
130 ' ---------------------------

140 ' *** SUBROUTINE ***

150 FOR N=1 TO LEN(A$)
160 A=A+2^VAL(MID$(A$,N,1)
170 NEXT N
180 RETURN

190 ' *** YOUR PROGRAM STARTS HERE ***

200 CLS
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 270
250 PRINT "NOT BINARY NUMBER"
260 GOTO 210
270 NEXT N
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 The Subroutine

The TRS-80 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 SHIFT 6 key.

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  ' ***********
20  ' *         *
30  ' * ROUNDER *
40  ' *         *
50  ' ***********
60  '  ----------------------------
70  '       ++ VARIABLES ++
80  '   A:   NUMBER TO BE ROUNDED
90  '   P:   DIGITS DESIRED TO
100  '       RIGHT OF DECIMAL POINT
110  '  B:   ROUNDED VALUE
120  '
130  ' ----------------------------
```

```
140 ' *** INITIALIZE ***

150 A=55.534
160 P=2
170 GOTO 230

180 ' *** SUBROUTINE ***

190 C=A+5.5*10^-(P+1)
200 B=INT(C*10^P)/10^P
210 RETURN

220 ' *** YOUR PROGRAM STARTS HERE ***

230 GOSUB 190
240 PRINT B
```

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.

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 TRS-80; 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 TRS-80. Certain

Boolean operators, such as AND and OR, are used with many subroutines to examine memory registers on the bit level.

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 TRS-80 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 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 computer's memory buffer and then to 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, tape, or in RAM. BASIC and machine-language programs, as well as text material, are all files.

Function key: The row of eight keys at the extreme top of the computer 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 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.

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 alternating lines in the resulting programs. However, 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 computer's signals to sounds that can be transmitted over telephone lines. The modem also receives sounds and converts them back for the computer to use.

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.

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
```

or

```
10 OFFSET=30000
20 POKE OFFSET+1,X
```

The second method is often clearer and can also be used when the memory location defined by OFFSET can vary.

OR: A Boolean operator that is used to compare one byte with another on a bit for bit level. If a bit and the corresponding bit in the other byte are either 1, OR will produce a 1 as the result.

Parallel: A method of transferring data an entire bit at a time by sending each of the eight bits along a separate parallel address line simultaneously. Serial transfer, on the other hand, transmits each of the eight bits one at a time.

Port: One of the "windows" used by the Portable Computer to talk to the outside world. The RS232 interface is a type of 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 can be made to start 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 some mass storage device, 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 computer that keeps track of elapsed time 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 computer's microprocessor and can be accessed only through machine language. Some memory locations in the computer perform a register-like function, telling the computer whether a certain feature is ON or OFF, telling the volume of a sound oscillator, or supplying some other status report.

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 computer 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 string delimiters are commas and quotation marks.

String variable: A variable that can store alpha information only. Strings can include numbers, punctuation marks, and graphics, but the computer recognizes them only as characters, not as values.

Subroutine: A program module that performs a specific task, called through the GOSUB statement and ending with RETURN, which directs program control back to the instruction following the GOSUB.

Toggle: A feature that can be either ON or OFF is sometimes "toggled" between the two, like a light switch.

Upload: To store a file from disk or tape in the computer'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).

Index

Whet Your Programming Appetite With a "Cookbook" Approach to Writing Quality Programs on the TRS-80 Portable Computer!

Acclaim from prepublication reviewers:

"This book can't miss! It's well-organized and well-written. . .even the novice can handle the programs!"

"The writing style and level of presentation are excellent. . .the author has a natural talent for communicating computer lingo and writing for a specific audience—the experienced beginner!"

TRS-80 PORTABLE COMPUTER 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 TRS-80! 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 using function keys, designing character sets, joystick action, sound, and other special features of the TRS-80 Portable Computer. No more "stewing" over exotic, topheavy math functions and statistics! Complete with line-by-line description 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 TRS-80's are not enough for your needs!
- Tips on routines to make your games of arcade-quality!
- Plus—a comprehensive glossary and index!

CONTENTS

Simulating Joysticks And Paddles/Using The Clock And Interrupts/Using Sound/Basic Tricks/Game Routines/Data Files/Business And Financial Subroutines/Add New Capabilities/Bits and Bytes

ISBN 0-89303-904-7