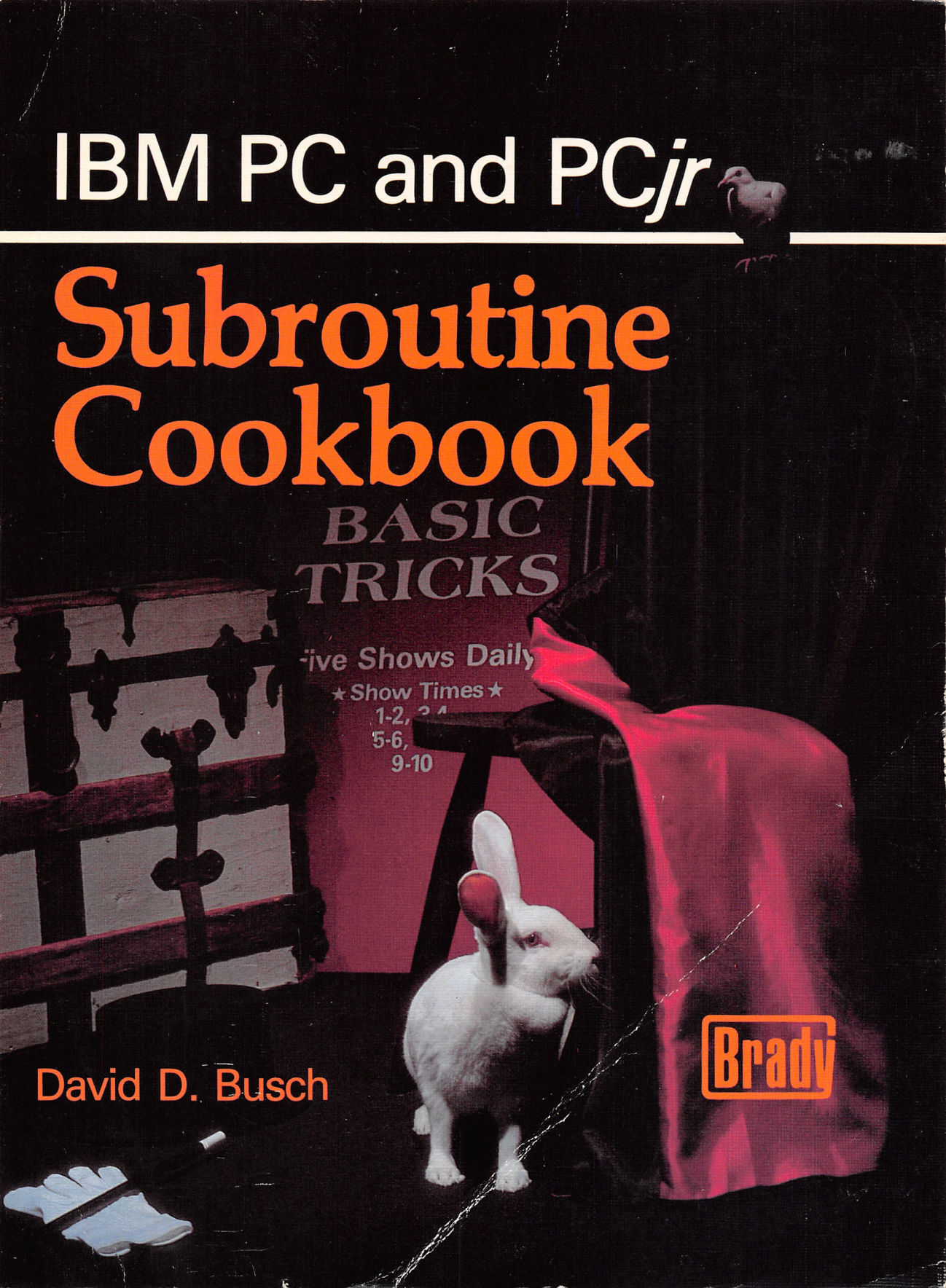
# IBM PC and PC*jr*

# Subroutine Cookbook




David D. Busch

Brady

# IBM PC AND PCjr 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 Designer: Janis K. Oppelt
Art Director/ : Don Sellers
Assistant Art Director: Bernard Vervin
Cover Photograph: George D. Dodson
Manufacturing Director: John A. Komsa

Typesetter: Emerald Graphic Systems, Syracuse, NY
Printer: R. R. Donnelley & Sons Co., Harrisonburg, Virginia
Typefaces: Garamond

Acknowledgments to Dave Kisser's rabbit, Fluffy, for appearing on the front cover.



**Library of Congress Cataloging in Publication Data**

Busch, David D.
**IBM PC and PCjr subroutine cookbook.**

Includes index.
1. IBM Personal Computer—Programming. 2. IBM PCjr (Computer)—Programming. 3. BASIC (Computer program language) 4. Subroutines (Computer programs) I. Title.
II. Title: I.B.M. P.C. and P.C. jr subroutine cookbook.
QA76.8.I2594B87 1985 001.64ffl2 84-11079

ISBN 0-89303-542-4

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

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

# CONTENTS

# PREFACE

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

Did you read an explanation of how to use your IBM Personal Computer's PLAY statement, only to wonder, "Well, I think I understand how it works—but how do I actually do it?"

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

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

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

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

Most subroutine books concentrate on "general" business or personal routines. Those are included here, too, but we've also emphasized IBM-specific tips aimed at your special needs. New capabilities have been added to IBM BASIC. Special features such as key trapping, use of asyncronous communications, generating sounds, programming the special function keys, and using the built in real-time clock are covered.

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

# INTRODUCTION

Be forewarned. This book is unlike any other collection of subroutines that you might have seen before. Herein are nearly six dozen 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 IBM PC and PCjr-specific routines that take the mystery out of using function keys, the built-in clock, interrupt routines, and other special IBM PC and PCjr features.

Most "subroutine" books are top-heavy with exotic math functions and rarely used statistical programs. If you've ever picked up one of those volumes, you were probably dismayed to find that most of the subroutines were not very useful. Most of us don't really use higher mathematics in our everyday work. That type of subroutine was 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 things like Fast Fourier transforms.

However, the IBM PC and PCjr, while they are powerful, capable microcomputers, are being sold to a broad range of users. Some buy PCs or XTs, and only want to use their computers for business. Other users of both the PC and PCjr are more interested in learning programming and may have a limited technical background. Then, there are those of you who really do understand computers but would like to avoid reinventing the wheel.

The *IBM PC and PCjr Subroutine Cookbook* is meant for all of you. There are some useful, general routines included here. This book also bristles with modules designed specifically to perform some sorely needed task for the IBM PC and PCjr alone.

Interested in using the cursor pad 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 IBM PC and PCjr'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.

Games players on both the PC or PCjr 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 IBM PC and PCjr register.

We've gone light on the simpler 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. There are dozens of books that can teach you BASIC. However, there are fewer volumes like this one that can help you go the next step—beyond basic BASIC to true proficiency. If you already know what a FOR-NEXT loop is and what happens when your PC encounters GOTO, you are ready for the lessons contained here. Ideally, you should have written several programs on your own, and be ready to tackle some more sophisticated programming. Those who need this book most will know who they are. You're the programmers who need some gentle guidance, a few inside tips, and the luxury of not having to reinvent the wheel.

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. Because they begin their search for a line number at the beginning of a program, all IBM PCs work fastest when accessing subroutines located there. So, you will probably want to deposit yours there. Using the BASIC RENUM facility will make this task easy.

To make things simpler, 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 many of the subroutines are 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. You will be directed to look at previous subroutines for longer explanations at times. This allows you to access the routines in any order, without reading the entire book.

In most cases, the subroutines will work equally well in machines equipped with either a color/graphics adapter or the IBM monochrome adapter. Therefore, in most, but not all, of the subroutines, appropriate SCREEN statements have been left out. If you have both types of monitor adapters (like the author) you will need to set the MODE yourself. To make these subroutines easily adaptable for both 80-column and 40-column screen widths, they have been written somewhat generically. You can add screen formatting touches, including more refined LOCATE statements, to suit your particular configuration. 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, unlike many other BASICs, up to 40 characters are significant for variable names in IBM PC and PCjr BASIC. So, PAYMENT, used in one subroutine, is actually a different variable name than PAID, which might be used in a second. But, if your subroutine and program both make independent use of the variable PAID, conflicts resulting in incorrect answers could result. You should take this precaution with any program you write, whether "foreign" subroutines are being transplanted or not. Variable names may contain reserved words but may not consist of reserved words alone. Therefore, "TOTAL" is fine as a variable, but "TO" is not.

Although most of the subroutines will operate properly with any of the available IBM BASIC's, they were written specifically for Advanced Basic. Those that require special features of BASIC 2.0 are noted.

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.

Each subroutine explanation includes a list of three sample applications for the subroutine to get you started. The descriptions usually include other hints on where you can use them in your programs. But, ideas should not be in short supply.

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

## LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

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

## TRADEMARKS OF MATERIAL MENTIONED IN THIS TEXT

- IBM MACHINES—International Business Machines Corporation
- APPLE MACHINES—Apple Computer, Inc.
- COMMODORE MACHINES—Commodore Business Machines, Inc.
- RADIO SHACK MACHINES—Tandy Corporation
- MASTERMIND—Invicta
- MICROSOFT—Microsoft Corporation
- BREAKOUT—Atari, Inc.
- PONG—Atari, Inc.
- PACMAN—Midway Mfg. Co., A Bally Company
- HOMEWORD—Sierra On-Line, Inc.
- ATARI MACHINES—Atari, Inc.
- TEXAS INSTRUMENTS—Texas Instruments, Inc.
- DUNGEONS & DRAGONS—TRS Hobbies, Inc.


MERGE
RENUM
APPENDING
HOW TO USE
SUBROUTINES

# 1

# Subroutine Magic

One of the best things about subroutines is that they can be reused many times within an existing program and put to work in many different pieces of software as well. Once you have typed in, say, a joystick routine from this book, you will not need to retype it every time you write a new program requiring joystick handling. Because the subroutines in this book have been designed as stand-alone modules, with both the input and output clearly defined, they can be recycled quite easily. You will want to store your subroutine "library" on disk and call it 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 PC and PCjr have 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, from the standpoint that there is no danger that wanted program lines will be written over with those of the merged program.

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

To MERGE with the PC, go to BASIC and load the module that you wish to add to memory. The subroutine or program that you wish to MERGE must be stored in ASCII form, which is accomplished by appending ",A" to the filename when storing, e.g., SAVE"A:filename.BAS",A

To MERGE just type:

```
MERGE"filename.BAS"
```

That's all there is to it. The PC does the work for you.

Another BASIC command that will be useful to you in using this book is RENUM. This command will renumber your program lines so that a subroutine can be fitted into an existing program easily. The syntax for RENUM is as follows:

```
RENUM [new beginning line number ] [ ,old line number][,
increment]
```

That is, you can specify the first line number to be used in the new renumbering sequence, the line in the current program where renumbering will start, and the increment used for each new line number. For example:

```
RENUM 10Ø,3Ø,5
```

In this case, the newly numbered program will have its first new line number begin with 100. The renumbering will commence at line 30 (so any lines prior to 30 will remain untouched). The new line numbers will be created with an increment of 5, that is, 100, 105, 110, 115, etc.

Any of these values can be left out, and the PC will use the default values. For new line number, the default is 10. So:

```
RENUM ,3Ø,5
```

will perform as described above, only the new lines will start at 10 instead of 100. If no starting line number in the current program is specified, the default is the first line in the program. Thus:

```
RENUM 10Ø,,5
```

will start with the first line, not line 30, as in the original example. Finally, a default increment of 10 is used. Therefore:

```
RENUM 10Ø,3Ø
```

will renumber from line 30, starting with a new line number of 100, and with increments of 10. If we want, we can leave all of the arguments off:

```
RENUM
```

This command will renumber from start to end, with increments of 10, beginning with a new line number of 10.

Of course, RENUM also will change all the line number references, so your GOTO's and GOSUB's will still be correct, as long as the line number specified exists. While RENUM is helpful in making additional space in your programs, it can also make inserting subroutines easier.

Simply RENUM your program to leave a blank where you want the subroutine to go after the merge. Say you want the subroutine to go after the current line 130 in your program. The subroutine is 50 lines long (to use an extreme case). First enter a new line:

```
135 REMARK
```

Then renumber the program in increments of 100:

```
RENUM 1ØØ,1,1ØØ
```

Your program will now have spaces of 100 between lines. Most likely line 130 will now be 1300, and line 135 will now be line 1400. You can delete the REMARK at line 1400. That will leave a gap from 1301 to 1499, which will hold the longest subroutine imaginable. SAVE your main program (in ASCII form, using the ",A" option). Now, load the subroutine, and renumber it so that the first line number will be between 1301 and 1499, leaving enough space for the entire routine. An increment of 1 will usually do this:

```
RENUM 13Ø1,1,1
```

Then type MERGE"program.BAS"

Presto! Your original program, with lines up to 1300, and from 1500 and beyond, is now merged with the subroutine in the proper location. Renumber one more time:

```
RENUM
```

Now your program, with the included subroutine, will be numbered from 10 on in increments of 10.

RENUM can also be used to move a subroutine that is already in a program, but which you would like to relocate. Do as previously described to make a "hole" in your program. Then LIST the lines containing the subroutine. Using the cursor keys, move up to the lines you want to move, and type over the existing line numbers with the new line numbers. A copy of the lines will appear in the new location. Delete the subroutine in the old location. You must then make sure that GOTO and GOSUB references to the subroutine are changed in your program before re-

numbering again. One fast way to do this is to save the program, then RENUM, noting the "Undefined line number" errors. Then, reload and make the corrections.

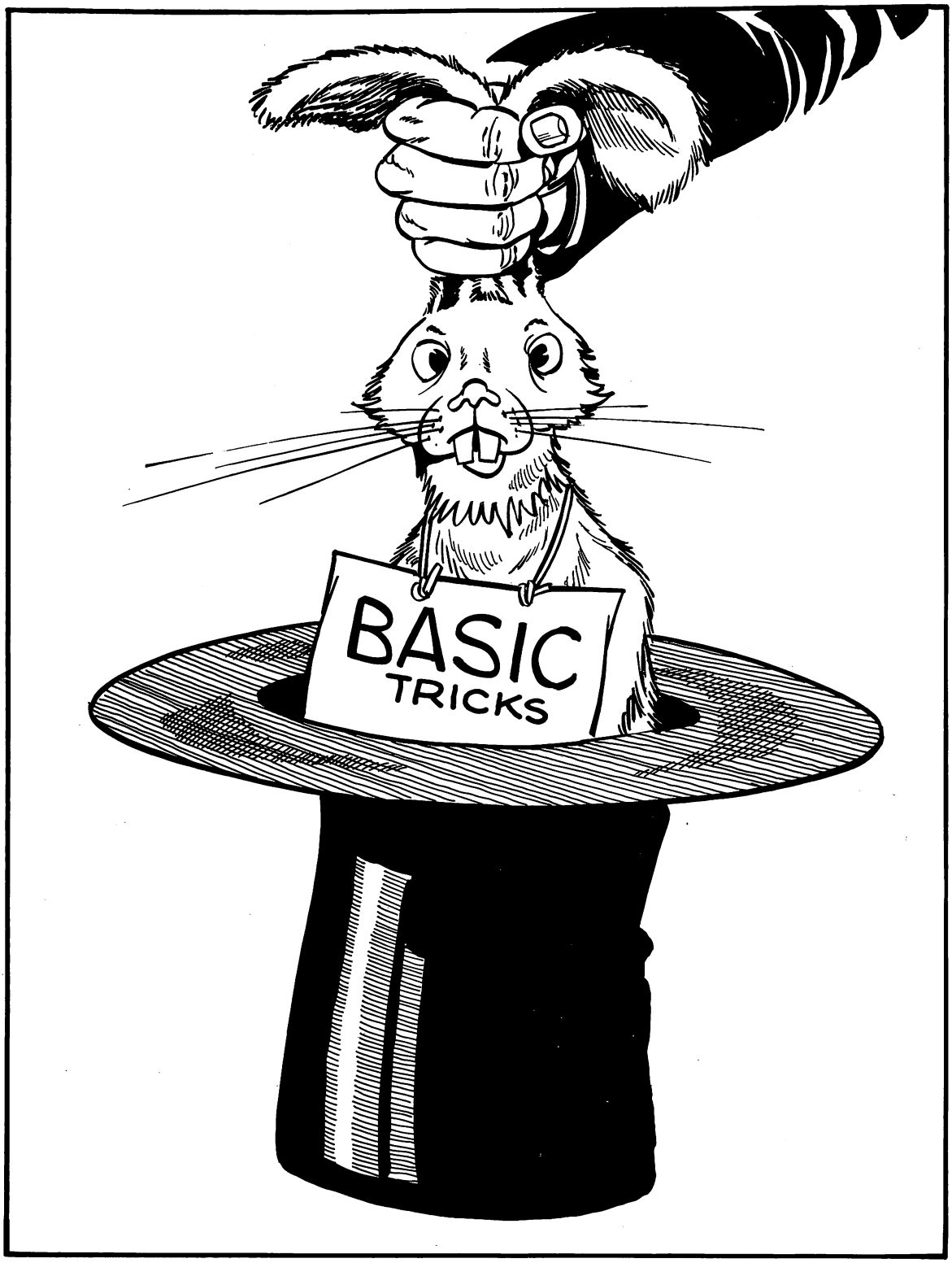
## SAMPLE MERGING RUN

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
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 370
```


BASIC
TRICKS

# 2

# 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 check the IBM PC and PCjr's equivalent of screen memory to see what characters are displayed there. While redefining the IBM PC and PCjr's 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 from within a program.

Many first-time users of computers fear that they will somehow "hurt" their machines by typing some incorrect command at the keyboard. This is, of course, generally impossible, although there was at least one instance of a fabled POKE that would destroy the video board of a non-IBM computer. We don't personally know of anyone who was brave enough to try it and so must say that any computer is generally safe from mischief at the hands of those who only sit and type.

However, there are a number of things you CAN do with hardware from your keyboard. For those with both the monochrome and color/graphics adapter boards in their IBM PC's, it is possible to change from one to the other from DOS by using the MODE command. However, there may be times when you will want to do the same thing from BASIC under program control. The Switch Displays subroutine checks to see which of your two boards is activated and toggles to the other one. There is no need to tell the subroutine which board you want to use.

The Test Adapter subroutine will only check which adapter board is in use, for use when you are writing a program that functions differently with the two types of video displays, and you want to check on which is being used.

With color graphics and some types of video displays, the screen may not be properly centered. If your video monitor does not have a horizontal adjustment, a subroutine is included in this chapter to allow you to move the display and thereby center your screen.

One popular piece of hardware installed in a large majority of IBM computers is the Asynchronous Adapter, which permits communications through a modem or hardwiring. A good basic terminal package is included with DOS, but it has no provision for uploading or downloading files. A subroutine in this chapter will allow plain vanilla program or data transfer between two computers, one of which should be your IBM PC. The routine will also serve as your introduction to programming the computer's RS232 port.

# FUNCTION KEYS

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

The commands entered by the IBM PC and PCjr function keys, which can be up to 15 characters long, are stored in RAM. Normally, the ten keys are assigned functions like "List", "Load", "Save", "Run" or "Key." You may change any of these to definitions more useful to you. Common functions might include "?FRE(0)" and "?TIME$", which report on the amount of free memory and the current clock setting, respectively.

Of course, you can quite easily redefine the function keys from the BASIC command mode by typing "KEY (key number to redefine),(new definition)". This string will be invoked by the IBM PC and PCjr from that point on, even if you exit BASIC, turn off the computer, and come back at some later time.

You may have a program that needs to use special function definitions only for that program run, and you wish to return the IBM PC and PCjr to normal function after. One way to accomplish this would be to use the ON KEY interrupt, described in Chapter 4. However, you may also wish just to have the function key deliver a string.

This subroutine will do that for you. One module allows redefining any of the ten keys. This can be accomplished through user input, as shown, or by defining the variable within the program itself.

### Line-by-Line Description

**Line 140:** Enter key to be reprogrammed.

**Line 150:** Check to make sure key is in range 1-10.

**Line 160:** Enter new definition.

**Lines 170 to 190:** See if it should be ended with carriage return.

**Line 200:** See if definition is too long.

**Line 210:** Redefine key.

### You Supply

Definitions for function keys.

## Sample Applications

- Programming shortcut for writing keywords
- Log-ons when used with BASIC communications program
- Short programs and subroutines stored in key.

*RESULT...*

**Pressing function key summons desired string, up to 15 characters long.**

```
10 ' *****************
20 ' *               *
30 ' * FUNCTION KEYS *
40 ' *               *
50 ' *****************
60 GOTO 140
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '     K:  KEY TO PROGRAM
100 '    S$: STRING DEFINED
110 '
120 ' ---------------------------

130 ' *** PROGRAM KEYS ***

140 INPUT ''PROGRAM WHICH KEY (1-10);K
150 IF K<1 OR K>10 GOTO 140
160 LINEINPUT ''WHAT STRING? (HIT <ENTER> ONLY TO
    NULL)";S$
170 PRINT "END WITH CARRIAGE RETURN (Y/N)"
180 A$=INKEY$:IF A$="" GOTO 180
190 IF A$="Y" OR A$="y" THEN S$=S$+CHR$(13)
200 IF LEN(S$)>15 THEN PRINT"TOO LONG. 15 CHARACTERS
    ONLY.":GOTO 110
210 KEY K,S$
220 RETURN

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

240 GOSUB 140
```

# CHECK SCREEN

*WHAT IT DOES...*

**Tells what character is printed on a certain screen location.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- ROW: Position to check
- COL: Column to check
- CHAR$: Character found
- FG: Foreground color
- BG: Background color
- BLINK$: Blinking state.

### How to Use Subroutine

The IBM PC and PCjr, unlike some other computers, do not have screen memory that is easily accessible by the novice. The older PC has its screen memory tucked away in a special location. Consecutive locations store the character being displayed, attributes, and so forth. The PCjr takes up some of user RAM for its screen memory but tells the operating system exactly where the information is stored so that DOS can "look" just as if the PCjr were a PC and be automatically directed to the proper location even though the two computers use different addresses.

Fortunately, the BASIC programmer does not have to bother with this shifting screen memory. BASIC has a very nice SCREEN function that will return the character currently printed at any ROW or COLUMN, or the foreground and background color there, and whether or not the character is blinking.

This capability is especially useful for games in which we want to see what lies ahead of our moving ship or missile. Graphics can be placed on screen using LOCATE, which is, fortunately, very fast. We can LOCATE the cursor at any row or column on the screen. If the PRINT statement is followed by a semicolon (except for the last screen position), the screen will not SCROLL. Thus, LOCATE will function nearly the same as POKEing to video memory on other computers.

The following subroutine handles the chore for you automatically. While it asks you which address to check, your actual program will probably define ROW and COL depending on the movement of a screen object or some other parameter. For tips on moving screen objects, read Chapter 7.

## Line-by-Line Description

**Lines 170 to 180:** Enter row and column to check.

**Line 190:** Ask if character or color should be returned.

**Line 200:** If invalid choice, go back and ask again.

**Line 210:** If choice was 1, make equal to 0 so that character, not color, will be returned by SCREEN function.

**Line 220:** Find out character or color.

**Line 230:** Determine blinking attribute.

**Line 250:** Calculate foreground color.

**Line 260:** Calculate background color.

**Lines 300 to 310:** Print results.

## You Supply

ROW and COL to check.

## Sample Applications

- Game program to detect collision
- Applications generator to see what has been typed on screen
- User input routine allowing multiline inputs and editing.

*RESULT...*

**CHAR$ will equal character printed on screen.**
**FG and BG will equal foreground and background color numbers, and blinking state will be revealed.**

```
10 ' ******************
20 ' *                *
30 ' *  CHECK SCREEN  *
40 ' *                *
50 ' ******************
60 GOTO 290
70 ' ------------------------------
80 '     ++ VARIABLES ++
90 '     ROW:    POSITION TO CHECK
100 '    COL:    COLUMN TO CHECK
110 '    CHAR$:  CHARACTER FOUND
120 '    FG:     FOREGROUND COLOR
130 '    BG:     BACKGROUND COLOR
140 '    BLINK$: BLINKING STATE
150 ' ------------------------------

160 ' *** SUBROUTINE ***

170 INPUT"WHICH ROW :";ROW
180 INPUT"WHICH COLUMN :";COL
190 INPUT "1. CHARACTER OR 2. COLOR :";C
200 IF C<1 OR C>2 GOTO 190
210 IF C=1 THEN C=0
220 CHAR=SCREEN(ROW,COL,C):IF C=0 THEN
    CHAR$=CHR$(CHAR):RETURN
230 IF CHAR>127 THEN BLINK$="BLINKING" ELSE
    BLINK$="NOT BLINKING"
250 FG=CHAR MOD 16
260 BG=(((CHAR-FG)/16)MOD 128)
270 RETURN

280 ' *** YOUR PROGRAM STARTS HERE ***

290 GOSUB 170
300 PRINT ''FOUND :";CHAR$;" ";BLINK$
310 IF C=2 THEN PRINT"FOREGROUND COLOR :";FG
    :PRINT"BACKGROUND :";BG
```

# SWITCH DISPLAYS

*WHAT IT DOES...*

**For computers with BOTH color/graphics adapter and monochrome adapter, changes from one to the other.**

## Versions: IBM PC, Advanced BASIC, both types of video adapters

### Variables

None.

### How to Use Subroutine

If you have a program which you would like to alternate between one display and the other, you can use this subroutine to make the switch automatically. As noted in the IBM BASIC manual, the screen you switch TO will be cleared, and you will have to keep track of cursor position.

### Line-by-Line Description

**Line 140:** Define PEEK and POKE offset.

**Line 150:** Check to see which adapter is currently being used.

**Lines 170 to 200:** Change to monochrome adapter and adjust screen.

**Lines 220 to 250:** Change to color/graphics adapter and adjust screen.

### You Supply

Access to routine as needed.

### Sample Applications

- Programs using more than one display board
- Programs designed to run on one board only
- Rapid switching of MODE without going to DOS.

*RESULT...*

**Display will be switched to the other adapter board.**

```
10 ' ********************
20 ' *                  *
30 ' * SWITCH DISPLAYS *
40 ' *                  *
50 ' ********************
60 ' ---------------------------
70 ' ++ VARIABLES ++
80 '
90 '       NONE
100 '
110 ' ---------------------------

120 GOTO 280

130 ' *** SUBROUTINE ***

140 DEF SEG=0
150 IF (PEEK(1040) AND 48)=48 THEN ADAPTER=1 ELSE
    ADAPTER=0
160 IF ADAPTER=1 THEN GOTO 220
170 POKE 1040,(PEEK(1040) OR 48)
180 SCREEN 0
190 WIDTH 80
200 LOCATE ,,1,12,13
210 RETURN
220 POKE 1040(PEEK(1040) AND 207) OR 16
230 SCREEN 1,0,0,0
240 SCREEN 0
250 LOCATE ,,1,6,7
260 RETURN

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

280 GOSUB 140
```

# TEST ADAPTER

*WHAT IT DOES...*

**For programs that may run on computers with either color/graphics adapter or monochrome adapter, finds out which is activated.**

## Versions: IBM PC, Advanced BASIC

## Variables

None.

## How to Use Subroutine

If your program takes different action depending on adapter type, use this subroutine to determine which is in use.

## Line-by-Line Description

**Line 140:** Set PEEK offset.

**Line 150:** Check to see which adapter is used.

**Line 180:** Access subroutine.

**Lines 190 to 200:** Display results.

## You Supply

Access to subroutine as needed.

## Sample Applications

- Programs that have different actions for different adapters.

*RESULT...*

**Type of adapter in use will be returned.**

```
10 ' ********************
20 ' *                  *
30 ' *   TEST ADAPTER   *
40 ' *                  *
50 ' ********************
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '
90 '       NONE
100 '
110 ' --------------------------

120 GOTO 180

130 ' *** SUBROUTINE ***

140 DEF SEG=0
150 IF (PEEK(1040) AND 48)=48 THEN ADAPTER=1 ELSE
    ADAPTER=0
160 RETURN

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

180 GOSUB 140
190 PRINT"CURRENTLY ACTIVATED ADAPTER IS ";
200 IF ADAPTER=1 THEN PRINT"MONOCHROME" ELSE PRINT"COLOR"
```

## CLEAR KEYBOARD BUFFER

*WHAT IT DOES...*

**Clears remaining characters in keyboard buffer.**

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

None.

## How to Use Subroutine

Use after INKEY$ routines to clear keyboard buffer, which will hold only 15 characters.

## Line-by-Line Description

**Line 140:** Define POKE offset.

**Line 150:** Clear buffer.

## You Supply

Access to subroutine as required.

## Sample Applications

- Keeping wrong key input from getting into INKEY$ loops
- Games that use key presses to move objects, etc.
- Insuring that computer only receives latest input.

*RESULT...*

**Keyboard buffer cleared.**

```
10 ' ********************
20 ' *                  *
30 ' *   CLEAR BUFFER   *
40 ' *                  *
50 ' ********************
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '
90 '       NONE
100 '
110 ' ---------------------------

120 GOTO 180

130 ' *** SUBROUTINE ***

140 DEF SEG=0
150 POKE 1050,PEEK(1052)
160 RETURN
```

```
17Ø ' *** YOUR PROGRAM STARTS HERE ***

18Ø GOSUB 14Ø
```

# CENTER SCREEN

*WHAT IT DOES...*

**Centers monitor display.**

## Versions: IBM PC, Advanced BASIC

### Variables

None.

### How to Use Subroutine

Access this subroutine to center your monitor display. Press SPACE to move to right, ENTER when finished.

### Line-by-Line Description

**Lines 140 to 160:** Display line of asterisks.

**Lines 170 to 200:** Instructions.

**Lines 210 to 240:** If SPACE pressed, adjust screen.

### You Supply

Center the screen.

### Sample Applications

- Adjust screen for proper viewing
- Align text
- Make all lines visible.

*RESULT...*

**Screen is centered for 40 column display.**

```
10 ' ********************
20 ' *                  *
30 ' *  CENTER SCREEN   *
40 ' *                  *
50 ' ********************
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '
90 '       NONE
100 '
110 ' ---------------------------

120 WIDTH 40:GOTO 270

130 ' *** SUBROUTINE ***

140 FOR N=1 TO 80
150 PRINT"*";
160 NEXT N
170 PRINT
180 PRINT"PRESS SPACE TO MOVE SCREEN"
190 PRINT"TWO CHARACTERS TO THE RIGHT."
200 PRINT"PRESS <ENTER> TO QUIT."
210 A$=INKEY$:IF A$="" GOTO 210
220 IF A$=CHR$(13)THEN RETURN
230 IF A$<>CHR$(32) GOTO 210
240 OUT 980,2:OUT 981,43
250 RETURN

260 ' *** YOUR PROGRAM STARTS HERE ***

270 GOSUB 140
```

# FILE TRANSFER

*WHAT IT DOES...*

**Allows sending program listing or data file out Asynchronous Adapter port.**

## Versions: IBM PC and PCjr with Asynchronous Adapter, Advanced BASIC

### Variables

- F$: Name of file
- INFO$: Data received
- BYTES: Number bytes sent.

### How to Use Subroutine

Communications is rapidly becoming one of the most popular applications for computers like the IBM PC and PCjr. The machine needs only to be coupled with a low-cost device called a modem. A modem converts the computer's signals into sounds that can be transmitted over the telephone wires. In addition, some sort of communications software needs to be written to allow the IBM to talk to the other computers. A fine BASIC program, COMM.BAS, is included with DOS. However, this does not have the capability of up or downloading files — that is, sending something stored on your disk drive to another computer, or storing on disk a file sent to you.

This subroutine is a quick way of sending files or programs from your computer to another in the same room. You can hard-wire the two together through a null modem adapter (cost: about $30) and the proper cables. The other computer also needs to be equipped with an RS232 serial port of its own and software that will allow it to upload and download. If that computer is also an IBM, this routine will work fine.

The routine does not allow other forms of "talking" back and forth between two computers (COMM.BAS is fine for that) but can otherwise be used over phone lines. Note that the communications protocols have been defined as 300 baud, seven bit word, even parity. You can change to some other Asynchronous Adapter setting simply by putting the desired combination in the appropriate OPEN program lines.

This routine will allow sharing your BASIC programs with users of other computer systems, particularly those with similar BASICs. The Apple, Commodore,

and Radio Shack line all have many similarities with IBM BASIC 2.0, and the author has successfully transferred many programs to them. Some modifications have to be made to make the programs compatible, but, editing out special graphics characters, screen formatting characters, etc., before the program transfer will help a great deal.

NOTE: You will have to SAVE programs in ASCII form (using the ",A" option) in order to transmit them successfully. For example:

```
SAVE"filename.bas",A
```

Received program files will load just fine, as the BASIC interpreter can handle programs in either ASCII or the normal "compressed" format.

This subroutine will receive and save a file of any length. Some programs for other computers use a buffer that has a limited size. BYTES keep track of how many bytes have been transmitted. You may specify an upper limit for this, at which point the computer will BEEP and stop, allowing you to dump what you have sent to the buffer of the other machine, and reopen the buffer to accept additional data from the IBM computer.

## Line-by-Line Description

**Line 170:** Position cursor.

**Lines 180 to 260:** Determine mode.

**Line 270:** User enters file to upload.

**Line 280:** That file opened to be loaded into the PC.

**Line 290:** Communications line number one opened. User may change this line to specify COM2 instead, and any baud, parity, or word size desired.

**Line 300:** One line loaded from the file.

**Line 310:** Total bytes loaded so far increased.

**Line 320:** If more than 20000 bytes sent, pause. User may change 20000 if receiving computer has larger or smaller buffer.

**Line 330:** Send the line out the port.

**Line 340:** If that is all of the file, close and return to menu.

**Line 350:** Go back and load another line.

**Line 360:** Enter name of file to be stored on the disk.

**Line 370:** OPEN COM line #1.

**Line 380:** OPEN file for output as data is received.

**Lines 390 to 400:** Activate COM line trapping.

**Line 410:** Send program back to wait for data to arrive over comm line.

**Line 420:** Load data from buffer into INFO$.

**Line 430:** Print the data to the screen.

**Line 440:** Print it to the disk file.

## You Supply

Program or file to upload.

## Sample Applications

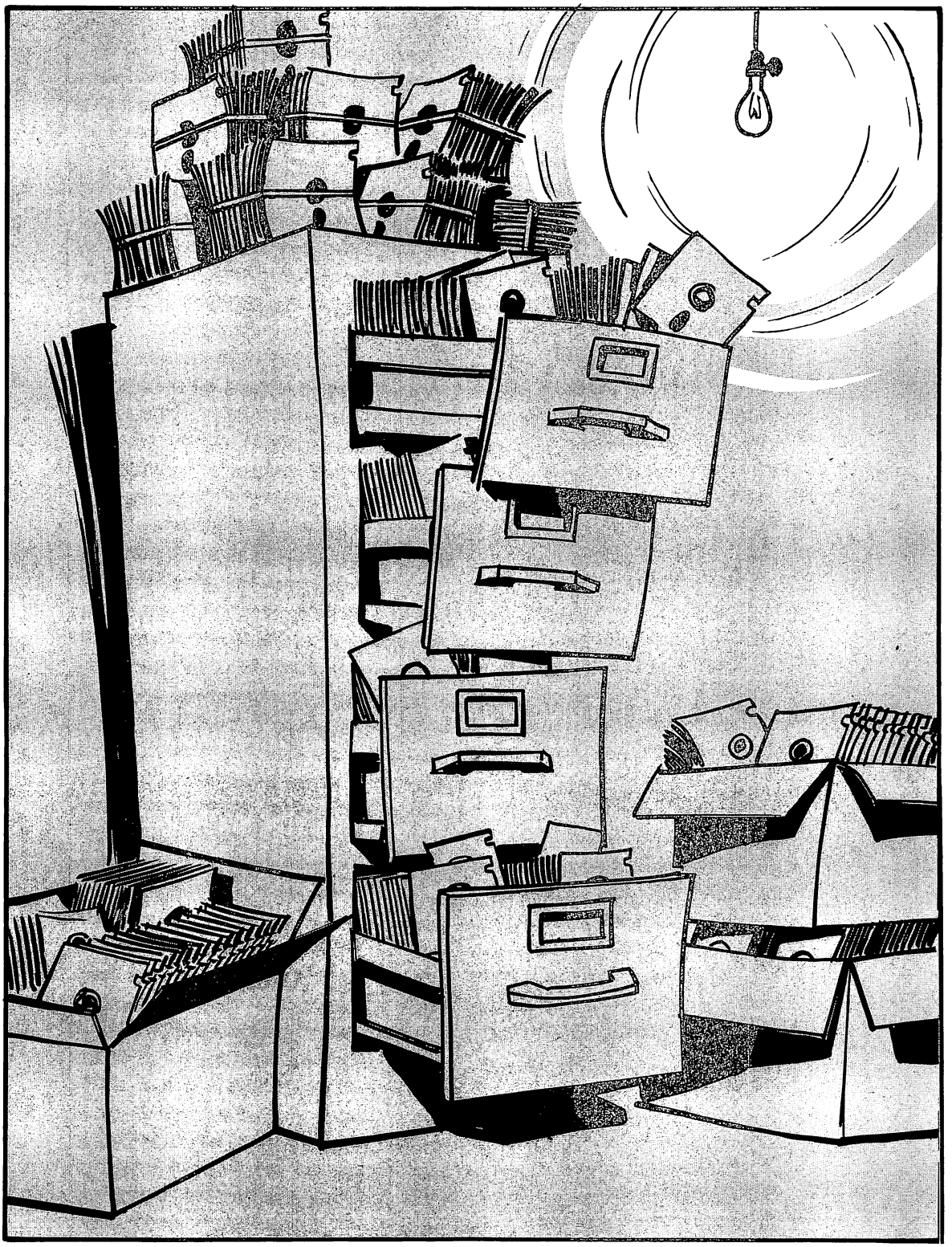
- Send program to another computer
- Share data files
- Use electronic mail.

*RESULT...*

**File is received or transmitted out Asynchronous Adapter port.**

```
10 ' *****************
20 ' *                 *
30 ' * FILE TRANSFER *
40 ' *                 *
50 ' *****************
60 '
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '     F$:    NAME OF FILE
110 '     INFO$: DATA RECEIVED
120 '     BYTES: NUMBER BYTES SENT
130 ' ---------------------------
140 ON KEY(10) GOSUB 460:KEY(10)ON
150 GOTO 480
```

```
16Ø ' *** SUBROUTINE ***

17Ø LOCATE 15,1
18Ø PRINT"Do you want to:
19Ø PRINT"1.) Send a file."
2ØØ PRINT"2.) Receive a file."
21Ø PRINT
22Ø PRINT TAB(6)"Enter choice:"
23Ø A$=INKEY$:IF A$="" GOTO 23Ø
24Ø A=VAL(A$)
25Ø IF A<1 OR A>2 GOTO 23Ø
26Ø ON A GOTO 27Ø,36Ø
27Ø INPUT "Enter filename to transmit -";F$
28Ø OPEN F$ FOR INPUT AS 1
29Ø OPEN "COM1:3ØØ,E,7" AS #3
3ØØ LINE INPUT #1,A$
31Ø BYTES=BYTES+LEN(A$)
32Ø IF BYTES>2ØØØØ THEN BEEP:IF INKEY$=""GOTO 32Ø
    ELSE BYTES=Ø
33Ø PRINT #3,A$
34Ø IF EOF(1) THEN CLOSE:GOTO 17Ø
35Ø GOTO 3ØØ
36Ø INPUT"Enter filename to store on disk -";F$
37Ø OPEN "COM1:3ØØ,E,7" AS #3
38Ø OPEN F$ FOR OUTPUT AS #1
39Ø ON COM(1) GOSUB 42Ø
4ØØ COM(1) ON
41Ø GOTO 39Ø
42Ø INFO$=INPUT$(LOC(3),#3)
43Ø PRINT INFO$;
44Ø PRINT #1,INFO$;
45Ø RETURN
46Ø CLOSE:END

47Ø ' *** YOUR PROGRAM STARTS HERE ***

48Ø GOSUB 17Ø
```



# 3

# Data Input, Editing, and Output

These subroutines will allow you to have a greater amount of control over the data that is entered by users into your programs.

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

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

Once data has been entered, it frequently must be sorted into some semblance of order. Of course, you can use DOS to sort for you, but it is sometimes useful to be able to sort from BASIC. 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.

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

Finally, we show you how to load data to an array from disk and how to write from an array to disk as an introduction to BASIC data files.

## NUMBER INPUT

*WHAT IT DOES...*

**Allows user to input only numbers.**

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

## How to Use Subroutine

Well-written programs include features that trap possible errors by the user—or avoid them entirely. When numbers 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 GO TO 10". If the user enters alpha characters, the program loops back and the input must be repeated.

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

The secret is a 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.

## Sample Applications

- Programs where only numbered menu choices allowed
- Insure user enters only numbers to variables
- Filter out accidental key depressions.

*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$=IF VAL(I$)>1.7E+36 THEN PRINT "NUMBER TOO
    LARGE FOR SINGLE PRECISION VARIABLE":GOTO 140
190 GOTO 140
200 I=VAL(I$):PRINT
210 RETURN

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

230 I$="":GOSUB 140
```

# LETTER INPUT

*WHAT IT DOES...*

**Allows user to input only alpha characters.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- I$: String entered.

### How to Use Subroutine

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

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

The secret is A$=INKEY$ loop. If the user presses a letter key, that letter is added to I$. A$ 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 where only the letters A-E are wanted. All numbers, graphics, and alpha characters larger than E can be ignored.

## Sample Applications

- Menus with alpha character choices
- Keep numbers out of user-entered filenames, etc.
- Insure user enters only alpha characters to string variables.

> *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$>"Z" GOTO 130 ELSE IF A>96 AND
    A<123 GOTO 160 ELSE GOTO 130
160 PRINT A$;
170 I$=I$+A$:IF LEN(I$)=255 THEN PRINT:PRINT "MAXIMUM
    STRING LENGTH REACHED!":RETURN
180 GOTO 130
190 RETURN

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

210 I$="":GOSUB 130
```

# CASE CONVERTER

*WHAT IT DOES...*

**Changes lowercase input to uppercase.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- I$: String entered.

### How to Use 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 IBM PC and PCjr can produce uppercase and lowercase. "Y" does not equal "y" and "JONES" is completely different from "Jones" to the IBM PC and PCjr.

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.

## Sample Applications

- Error trap in programs where answers will be compared with a correct answer
- Change all lowercase words to uppercase
- Can adapt to change to all lowercase, too.

> *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 I$="":GOSUB 130
```

# STRING SORT

*WHAT IT DOES...*

**Alphabetizes a list.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

Although as written the subroutine asks the user to enter the list from the keyboard, any means can be used to load the array. The file may also be read from disk or tape, for example, using one of the routines presented later in this book.

The bubble sort is so called because each entry in the array is examined and then allowed to rise up past "larger ones" 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 IBM PC and PCjr 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 for small lists, say, 30 or 40 items, the speed is satisfactory.

### Line-by-Line Description

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

**Line 140:** DIMension the array to proper size.

**Lines 170 to 200:** User enters each array item in random order. A disk or tape file read routine could be substituted for these lines to sort an existing string file.

**Line 210:** Start loop from 1 to the number of items to be sorted.

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

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

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

**Line 250:** If the "higher" element, A$, is already smaller than B$, then B$ remains where it is, and the inner loop steps off the next value of N1.

**Lines 260 to 270:** If B$ is smaller than A$, then the two strings are swapped, with B$ moving ahead one element, and A$ being pushed down one.

**Lines 280 to 290:** The inner and outer loops are incremented.

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

## You Supply

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

## Sample Applications

- Sort a data file for database management program
- Sort words for index or dictionary
- Sort names for greeting card list.

*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 ' ---------------------------
```

```
12Ø ' *** INITIALIZE ***

13Ø NU=1Ø
14Ø DIM US$(NU)
15Ø GOTO 35Ø

16Ø ' *** SUBROUTINE ***

17Ø FOR ITEM=1 TO NU
18Ø PRINT"ENTER #";ITEM
19Ø INPUT US$(ITEM)
2ØØ NEXT ITEM
21Ø FOR N=1 TO NU
22Ø FOR N1=1 TO NU-N
23Ø A$=US$(N1)
24Ø B$=US$(N1+1)
25Ø IF A$<B$ THEN GOTO 28Ø
26Ø US$(N1)=B$
27Ø US$(N1+1)=A$
28Ø NEXT N1
29Ø NEXT N
3ØØ FOR N=1 TO NU
31Ø PRINT US$(N)
32Ø NEXT N
33Ø RETURN

34Ø ' *** YOUR PROGRAM STARTS HERE ***

35Ø GOSUB 17Ø
```

## NUMBER SORT

*WHAT IT DOES...*

**Sorts group of numbers by size.**

**Versions: IBM PC and PCjr, Advanced BASIC**

## Variables

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

## How to Use Subroutine

Sorting a list of numbers is a common need for many programs. Checking account files and other groups of numbers often have to be sorted to be most useful. This routine is a simple bubble sort that will sort any group of numbers that have been loaded into an array, US(n).

Although as written the subroutine asks the user to enter the number list from the keyboard, any means can be used to load the array. The file may also be read from disk or tape, for example, using one of the routines presented later in this book.

With the bubble sort, each entry in the array is examined, and then allowed to rise up past the one below until it encounters a "smaller" item. Numeric sorts are easier to understand than string sorts, because simple number comparisons are used. That is, 1237 is always larger than 32.6, and smaller than 7844. Gradually, each member of the list "floats" up to its proper place in the array.

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

## Line-by-Line Description

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

**Line 140:** DIMension the array to proper size.

**Lines 170 to 200:** User enters each array item in random order. A disk or tape file read routine could be substituted for these lines to sort an existing string file.

**Line 210:** Start loop from 1 to the number of items to be sorted.

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

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

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

**Line 250:** If the "higher" element, A, is already smaller than B, then B remains where it is, and the inner loop steps off the next value of N1.

**Lines 260 to 270:** If B is smaller than A, then the two numbers are swapped, with B moving ahead one element, and A$ being pushed down one.

**Lines 280 to 290:** The inner and outer loops are incremented.

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

## You Supply

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

## Sample Applications

- Sort list by zip code
- Sort check list by check number
- Sort page numbers in index.

*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.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

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

## How to Use Subroutine

An array is a table with rows and columns storing lists of data. In a checkbook register, each row might contain information about a single check/deposit transaction. Each column would contain a specific type of entry, 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) should become DTA(row,column), and A$ should be changed to A.

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

## Line-by-Line Description

**Lines 140 to 150:** Define number of rows and columns in the data file. User should change these numbers to reflect their own data.

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

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

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

**Line 220:** READ item of DATA.

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

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

**Line 290:** Access the subroutine.

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

## You Supply

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

## Sample Applications

- Load array for database program
- Load an array from DATA lines for quiz, question and answer program
- Load array with variables to be used in program.

*RESULT...*

**Data list is loaded into array.**

```
10 ' ****************
20 ' *              *
30 ' * ARRAY LOADER *
40 ' *              *
50 ' ****************
60 GOTO 280
70 ' ---------------------------
80 '    ++ VARIABLES ++
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$(NROWS,NCOLUMNS)
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
```

```
27Ø ' *** YOUR PROGRAM STARTS HERE ***

28Ø PRINT
29Ø GOSUB 2ØØ
3ØØ PRINT"NAME ADDRESS PHONE"
31Ø PRINT
32Ø FOR ROW=1 TO NROWS
33Ø FOR COL=1 TO NCOLUMNS
34Ø PRINT DTA$(ROW,COL);" ";
35Ø NEXT COL
36Ø PRINT
37Ø NEXT ROW
```

# INSERT STRING

*WHAT IT DOES...*

**Inserts string into another.**

## Versions: IBM PC and PCjr, Advanced BASIC.

### Variables

- TARGT$: Main string
- SUB$: String to be inserted into main string
- PLACE: Position to put SUB$.

### How to Use Subroutine

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

Like all IBM PC and PCjr 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 200:** Take the leftmost characters in the target string up to, and including position PLACE.

**Line 210:** Take the rightmost characters in the target string, starting with one after position PLACE.

**Line 220:** 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.

## Sample Applications

- Word processing or text editing program
- Special text formatting program to do global searches and replaces
- Insertion of a larger string that has more characters than the letters replaced.

> *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=14
170 GOTO 240

180 ' *** SUBROUTINE ***

190 IF LEN(TARGT$)+LEN(SUB$)>255 THEN PRINT "CANNOT
    INSERT ";SUB$;" INFO ";TARGT$;".":PRINT "RESULTING
    STRING WOULD BE MORE THAN 255 CHARACTERS!":RETURN
200 L$=LEFT$(TARGT$,PLACE)
210 R$=MID$(TARGT$,PLACE+1)
220 TARGT$=L$+SUB$+R$:RETURN
220 RETURN

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

240 GOSUB 190
250 PRINT TARGT$
```

# CHR$ VALUE

*WHAT IT DOES...*

**Returns CHR$ code for any key.**

## Versions: IBM PC and PCjr, Advanced BASIC.

### Variables

- A: CHR$ value of last key pressed.

### How to Use 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 an IBM PC or PCjr, it must be converted to a series of these binary 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. IBM departs somewhat from this code for the IBM PC and PCjr, 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 IBM BASIC, just like in other BASICs.

Even if you don't have a modem and aren't communicating with other computers, 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. At 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.

Sometimes a key pressed will be outside the range 0-255, as when ALT plus some other key or a key like HOME, Cursor Up, Pg Up, etc., is pressed. In this case, INKEY$ will return a two-character string with the first character being null. The second character tells which of the extended codes has been produced. This subroutine will allow you to determine these as well.

## Line-by-Line Description

**Line 130:** Wait for a key to be pressed.

**Line 140:** Determine ASC value of that key.

**Line 150:** If null returned, access extended code module.

**Lines 160 to 170:** Otherwise print ASC code.

**Line 190:** Determine extended code.

**Line 200:** Print result.

## You Supply

Just press the key that you want to check.

## Sample Applications

- Check large number of ASCII codes without table
- Hide messages in games programs
- Hide program credits for security reasons.

*RESULT...*

**Variable A will equal CHR$ value or extended code of that key.**

```
10 ' **************
20 ' *              *
30 ' * CHR$ VALUE *
40 ' *              *
50 ' **************
60 GOTO 230
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '     A: CHR$ VALUE OF KEY
100 '
110 ' ---------------------------

120 ' *** SUBROUTINE ***

130 A$=INKEY$:IF A$="" GOTO 130
140 A=ASC(A$)
150 IF A=0 GOTO 190
160 PRINT"CHR$ VALUE OF KEY ";A$
170 PRINT" IS :";A
180 RETURN
190 A=ASC(RIGHT$(A$,1))
200 PRINT "EXTENDED CODE IS 000 +";A
210 RETURN

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

230 GOSUB 130
240 GOTO 230
```

## SEQUENTIAL FILE—WRITE TO DISK

*WHAT IT DOES...*

**Writes a sequential data file to disk.**

**Versions: IBM PC and PCjr, Advanced BASIC**

## Variables

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

## How to Use Subroutine

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 .BAS files can be loaded by the IBM PC and PCjr 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 that contains the actual instructions for accessing the information.

Data files are one of the basic tools of business and personal programming, as they let you keep permanent records that can be accessed, printed out, manipulated, and otherwise used in a practical manner. Data files are akin to programs in that, lacking some mass storage for the data (or program), we would have to type the information in 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 IBM PC and PCjr cassette recorder is a good analog to sequential files, i.e., serial files. A program is a sequential file stored one byte at a time on your program tape or disk in the same order in which it is LISTed. In the case of cassette tapes, the program is continuous on one long piece of tape. Programs are stored sequentially in RAM.

Sequential data files also operate this way. If your program needs some information from 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 PRINT#, WRITE#, LINE INPUT#, or INPUT#, with each followed by the logical file number we are using.

So, we might have the following collection of lines:

```
1Ø OPEN "FILE1" FOR INPUT AS 1 ' Source file.
2Ø OPEN "FILE2" FOR OUTPUT AS 2 ' New data file
3Ø LINE INPUT#1, A$ ' Load a line from source file
4Ø PRINT#2,A$ ' Print it to data file.
5Ø CLOSE ' Close the files.
```

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

WRITE# is essentially the same as PRINT#, except that the former inserts commas between items as they are written and puts strings inside quotation marks. WRITE# also does not put a leading blank ahead of a positive number. With PRINT# you must explicitly separate items and strip off the blank if it is not desired.

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

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

## Line-by-Line Description

**Line 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:** WRITE 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 disk file. The counter NI should be redefined to reflect the number of items in the file each time an update is made. You should substitute your filename for "filename" in line 170.

## Sample Applications

- Data files of all types
- Keeping track of appointments
- Personal or business data files.

*RESULT...*

**Data file written to disk.**

```
10 ' ********************
20 ' *                  *
30 ' * SEQUENTIAL FILE *
40 ' *  WRITE TO DISK  *
50 ' *                  *
60 ' ********************
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '    NI:      NUMBER OF ITEMS IN FILE
100 '   DTA$(N): ARRAY STORING FILE
110 '
120 '
130 ' ---------------------------

140 ' *** INITIALIZE ***

150 DIM DTA$(NI)
160 GOTO 250

170 ' *** SUBROUTINE ***

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

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

250 GOSUB 180
```

# SEQUENTIAL FILE—READ FROM DISK

*WHAT IT DOES...*

**Reads a sequential data file from disk.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

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

## How to Use Subroutine

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

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

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

## Line-by-Line Description

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

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

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

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

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

**Line 230:** CLOSE the file.

**Lines 260 to 280:** Print data file to screen.

## You Supply

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

## Sample Applications

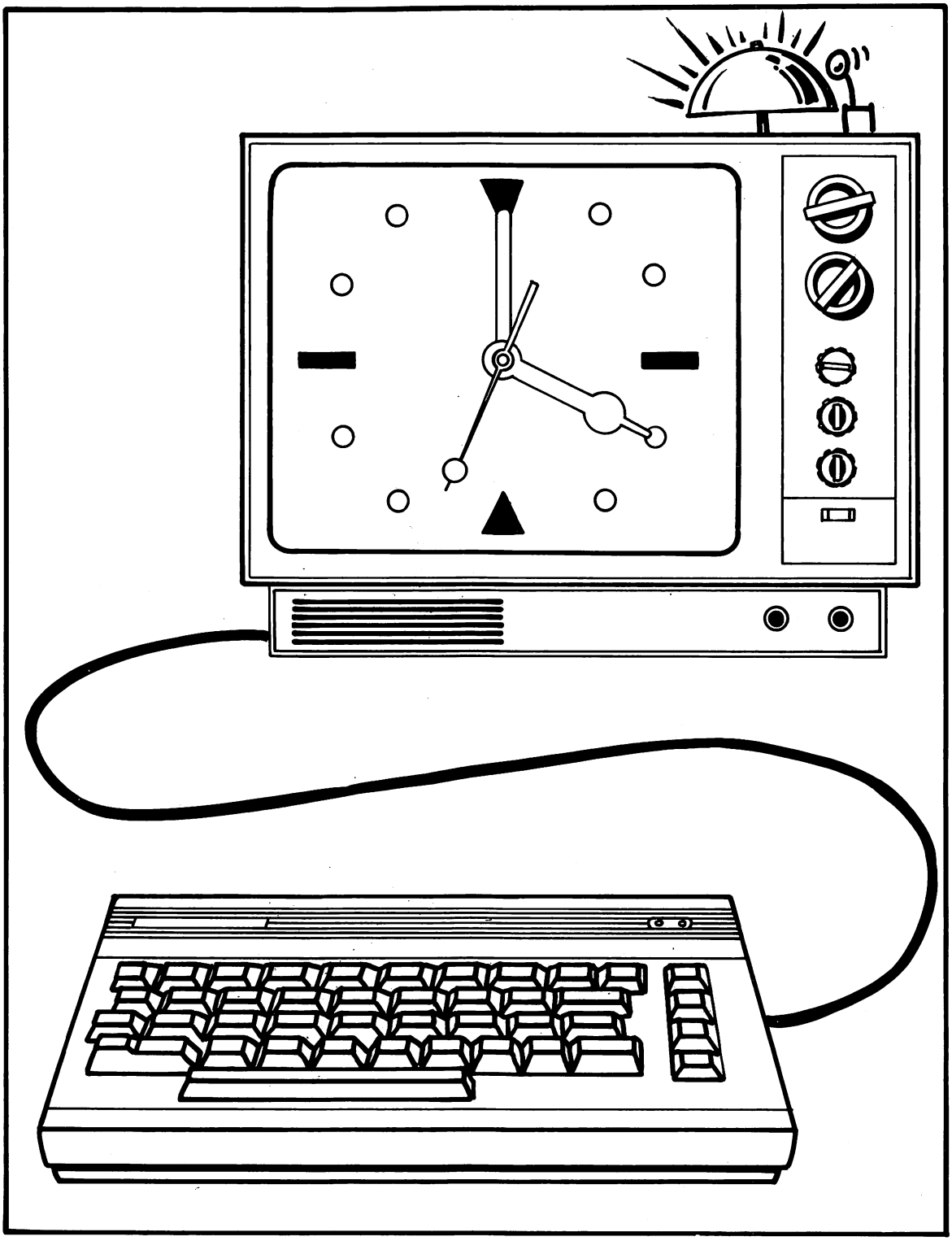
- Loading a data file from disk
- Checkbook data file
- Inventory.

*RESULT...*

**Data file read from disk.**

```
10 ' ********************
20 ' *                  *
30 ' * SEQUENTIAL FILE *
40 ' * READ FROM DISK  *
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
120 '               END OF FILE
130 '
140 ' ---------------------------------------


150 ' *** INITIALIZE ***


160 GOTO 250
```

```
17Ø ' *** SUBROUTINE ***

18Ø OPEN "FILENAME" FOR INPUT AS 2
19Ø INPUT#2,NI
2ØØ DIM DTA$(NI+AD)
21Ø FOR N=1 TO NI
22Ø INPUT#2,DTA$(N)
23Ø NEXT N
24Ø CLOSE 2:RETURN

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

25Ø GOSUB 18Ø
26Ø FOR N=1 TO NI
27Ø PRINT DTA$(N)
28Ø NEXT N
```



# 4

# Using the Clock and Interrupts

Many times, software can give computers features that the hardware manufacturer never thought possible. Other times, capabilities are entirely hardware dependent; if your computer is not capable of a given task, there is no way even the most ingenious software designer can get around that.

The IBM PC and PCjr's built-in clock and interrupt features are bonuses provided by the hardware, including the Intel 8088 CPU chip. This powerful 16-bit microprocessor has a number of capabilities that have been translated into features at the most basic level, i.e., incorporated into the BASIC language itself. A built-in real-time clock keeps track of the time elapsed since powerup or reset, while other "interrupts" check to see if a variety of events have taken place in between execution of other instructions.

This means that the computer does not have to follow the logic of a BASIC program exactly as written without variation. Instead, we can tell it to watch for certain events, and, if they occur, go do something else instead. The "key trapping" routines used in the first chapter were of this kind. Our program went about its business until one of the activated keys were pressed. Then we changed the value of the ROW and COLUMN used in LOCATE statements that printed our cursor character to the screen.

Prior to the latest generation of microprocessors, the most commonly used interrupt routine from BASIC was ON ERROR GOTO.... This interrupt is widely used in many different Microsoft-compatible versions of BASIC, extending back to the 8-bit computers of the dark ages of microcomputing (last year).

Unlike other commands, we do not have to enter an event-trapping 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 IBM PC and PCjr have a variety of ON...GOSUB routines available. We have already used ON KEY(n) and ON STIG(n). Others include ON PEN, which sends control to a specified subroutine whenever a light pen is activated. ON PLAY(n), available from BASIC 2.0 only, allows continuous background music to be played during program execution. An especially useful statement is ON TIMER, which will send the computer to the desired location when a desired amount of time has elapsed. ON COM will interrupt your program whenever a character is received over the asynchronous communications line. 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.

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(n) GOSUB will interrupt whatever you are doing whenever one of the defined function keys (or other keys) is pressed. You can define one or all ten function keys for this routine. Plus, you may add the cursor pad keys and, under BASIC 2.0, six other keys as well. Only those activated are trapped. The others are ignored.

Several of these event-trapping routines are addressed in this chapter. Timing is one of the most interesting interfaces your IBM PC and PCjr have with the real world. The computer 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 key to using real-time clock of your computer.

Because the IBM PC and PCjr 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. Your programs can access the variable TIMER, which is a number ranging from 1 to 86,400, or the number of seconds in 24 hours. If the real-time clock has been set accurately, variable TIMER will reflect the number of seconds that have elapsed since midnight, or since the last system reset. Note that this feature is available only in BASIC Versions 2.0 and beyond.

You can also use the ON TIMER(n) interrupt, which will send control to a desired subroutine after a specified, n, number of seconds have elapsed. The number n can range from 1 to 86,400. Thus, the computer timer can be set from one second to 24 hours.

## ELAPSED TIME

*WHAT IT DOES...*

**Tells how many hours, minutes, and seconds have elapsed.**

### Versions: IBM PC and PCjr, Advanced BASIC 2.0.

### Variables

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

### How to Use Subroutine

Your program should set variable S to equal TIMER at the beginning of the timing cycle that will be measured. Then access the subroutine at the end of the interval. A new value for TIMER will be stored in variable F and compared with S to determine how many hours, minutes, and seconds have elapsed in the interim.

## Line-by-Line Description

**Line 160:** Take current reading of TIMER.

**Line 170:** Calculate difference between F and S.

**Line 180:** Figure number of elapsed hours.

**Line 190:** Figure elapsed minutes.

**Line 200:** Figure elapsed seconds.

**Lines 210 to 220:** Print results.

**Line 250:** Take initial time reading.

**Line 260:** Wait awhile, until key pressed. Your program goes here.

## You Supply

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

## Sample Applications

- Determine elapsed time for reading or writing a file
- Measure elapsed time for a sort
- Time a game or program to compare performance.

*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 250
```

```
15Ø ' *** SUBROUTINE ***


16Ø F=TIMER
17Ø DF=F-S
18Ø HOURS=INT(DF/36ØØ)
19Ø MIN=INT((DF-(HOURS*36ØØ))/6Ø)
2ØØ MN$=VAL(MN$*6Ø)
21Ø PRINT"IT TOOK YOU ";HOURS;"HOURS"
22Ø PRINT MIN;"MIN. AND ";SEC;"SEC."
23Ø RETURN


24Ø ' *** YOUR PROGRAM STARTS HERE ***


25Ø S=TIMER
26Ø A$=INKEY$:IF A$="" GOTO 26Ø
27Ø GOSUB 16Ø
28Ø GOTO 25Ø
```

# TIMER

*WHAT IT DOES...*

**Sets computer as a timer.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

- HR$: Finish hour
- MN$: Finish minutes
- SEC$: Finish time

## How to Use Subroutine

This subroutine asks the user how much elapsed time should be allowed to pass before the ON TIMER(n) interrupt breaks into the program with the "time is up" notification. This will let you know when a given number of hours, minutes, or seconds have passed (somewhat like timing a roast in the oven). The real-time clock does not have to be set to the current time for this subroutine to perform correctly. Nor does your program have to have anything to do with timing anything. You could include this routine in a business program to let you know when a given period of time had passed. However, it could also be used in a game program to limit play to a certain number of minutes or seconds. If you need to be alerted at a particular time, use the TIMER INTERRUPT routine later in this chapter.

## Line-by-Line Description

**Lines 150 to 180:** Ask user for total hours, minutes, and seconds to be counted.

**Lines 190 to 220:** Calculate total number of seconds to be counted off.

**Line 230:** Turn on timer.

**Line 240:** Set routine for time to be counted.

**Line 270:** Notify that time is up.

## You Supply

Your program that operates while timing takes place.

## Sample Applications

- Remind user of appointment
- Set time limit on game or program
- Send program to another routine at given time limit.

*RESULT...*

**IBM PC and PCjr signals with a beep at end of requested time interval.**

```
10 ' *********
20 ' *       *
30 ' * TIMER *
40 ' *       *
50 ' *********
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '
90 '     HR$:  FINISH HOUR
100 '    MN$:  FINISH MINUTES
110 '    SEC$: FINISH TIME
120 '
130 ' ---------------------------


140 ' *** TIME TO BE MEASURED ***


150 PRINT" TOTAL TIME TO BE COUNTED:"
160 INPUT" ENTER HOURS:";HR$
170 INPUT" ENTER MINUTES:";MN$
180 INPUT "ENTER SECONDS:";SEC$
190 HR=VAL(HR$)*3600
200 MN=VAL(MN$*60)
210 SEC=VAL(SEC$)
220 TOTAL=HR+MN+SEC
230 TIMER ON
240 ON TIMER(TOTAL) GOSUB 270
250 GOTO 290


260 ' *** TIME UP ***


270 PRINT"TIME IS UP.":BEEP:RETURN


280 ' *** YOUR PROGRAM STARTS HERE ***


290 PRINT
```

# SECOND TIMER

*WHAT IT DOES...*

**Counts off seconds.**

## Versions: IBM PC and PCjr, Disk or Advanced BASIC

## Variables

- SEC: Number of seconds to count off.

## How to Use Subroutine

This subroutine shows yet a third way of using the computer as a timer. With this method, a WHILE-WEND 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 no faster than using the IBM PC and PCjr's interrupt feature. However, it is presented for those programmers who want an alternate method available under earlier versions of BASIC. This is a very simple method 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 noninterrupt-driven routines, this one prevents your program from tackling other chores during the timing inteval.

## Line-by-Line Description

**Line 140:** Start subroutine WHILE SEC > 0.

**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.

## Sample Applications

- Time number of seconds for a game
- Delay a program a fixed amount of time
- Use computer as a clock.

> *RESULT...*
>
> **IBM PC and PCjr 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 WHILE SEC
150 T$=TIME$
160 IF T$=TIME$ GOTO 160
170 LOCATE 15,10:PRINT TIME$
180 SEC=SEC-1
190 WEND
200 CLS:PRINT "TIME IS UP."
210 RETURN
```

```
22Ø ' *** YOUR PROGRAM STARTS HERE ***
23Ø CLS:PRINT
24Ø INPUT"HOW MANY SECONDS TO COUNT";SEC
25Ø GOSUB 14Ø
```

# TIME$ INTERRUPT

*WHAT IT DOES...*

**Allows computer to interrupt task at requested time.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

None.

## How to Use Subroutine

This subroutine uses the ON TIMER 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 170 to 190:** User enters time to be alerted.

**Lines 200 to 220:** Hours, minutes, seconds parsed out.

**Lines 230 to 240:** Format checked.

**Lines 250 to 270:** Hours, minutes, and seconds of finish time calculated.

**Line 280:** Current time taken.

**Lines 290 to 310:** Current hours, minutes, seconds parsed out.

**Lines 320 to 340:** Figure total seconds elapsed.

**Line 350:** Calculate total seconds to time.

**Line 360:** Turn timer ON.

**Line 370:** Set timer for required number of seconds.

**Lines 400 to 420:** Notify time is up.

**Line 460:** Print current time to screen.

## You Supply

Time when alarm should be triggered.

## Sample Applications

- Use as alarm clock
- End program by certain time
- Remind user of appointment.

*RESULT...*

**Your program is interrupted when desired time is reached.**

```
10 ' *************
20 ' *           *
30 ' *   TIMER   *
40 ' * INTERRUPT *
50 ' *           *
60 ' *************
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '       NONE
110 '
120 '
130 ' ---------------------------
140 GOTO 440
```

```
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 FH=VAL(A$)*3600
260 FM=VAL(B$)*60
270 FS=VAL(C$)
280 CT$=TIME$
290 A$=LEFT$(CT$,2)
300 B$=MID$(CT$,4,2)
310 C$=RIGHT$(CT$,2)
320 CH=VAL(A$)*3600
330 CM=VAL(B$)*60
340 CS=VAL(C$)
350 SECONDS=(FH+FM+FS)-(CH+CM+CS):IF SECONDS<1 THEN
    PRINT " TIME IS ALREADY PAST!":GOTO 180
360 TIMER ON
370 ON TIMER(SECONDS) GOSUB 400
380 RETURN

390 ' *** TIME IS UP ***

400 CLS
410 PRINT "TIME IS UP!":BEEP
420 STOP

430 ' *** YOUR PROGRAM STARTS HERE ***

440 PRINT
450 GOSUB 160
460 LOCATE 10,15:PRINT TIME$;
470 GOTO 460
```

# ON COM(n) INTERRUPT

*WHAT IT DOES...*

**Interrupts program when signal received over asynchronous port.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

- B$: Character received.

## How to Use Subroutine

This subroutine demonstrates the ON COM(n) interrupt routines. As with ON ERROR and ON TIMER, your program can be going about its business while the IBM PC or PCjr constantly checks the COM port 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 IBM PC and PCjr is connected to another computer. You may run your BASIC program and be signalled by the other computer when necessary.

## Line-by-Line Description

**Lines 130 to 150:** User enters which COM line to open.

**Line 160:** Specified COM line interrupt activated.

**Line 170:** Trapping turned on.

**Line 190:** One character retrieved from input buffer

**Line 200:** That character printed to screen.

**Line 230:** Your program starts here

## You Supply

Program.

## Sample Applications

- BASIC communications program with other functions
- Running other application, but alerting the user when communications are received
- Program transfer between computers.

*RESULT...*

**Your program is interrupted when signal received over the COM line specified.**

```
10 ' **********************
20 ' *                    *
30 ' *   COM INTERRUPT    *
40 ' *                    *
50 ' **********************
60 ' --------------------------
70 '    +++ VARIABLES +++
80 '     B$: CHARACTER
90 '         RECEIVED
100 '
110 ' --------------------------

120 ' *** INITIALIZE ***

130 INPUT "ENTER COM LINE TO OPEN :"
140 A$=INKEY$:IF A$="" GOTO 140
150 A=VAL(A$):IF A<1 OR A>2 GOTO 140
160 ON COM(A) GOSUB 190
170 COM(A) ON:GOTO 230

180 ' *** SUBROUTINE ***

190 B$=INPUT$(1,#A)
200 PRINT B$;
210 RETURN
```

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

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

# FUNCTION KEY INTERRUPT

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

None.

## How to Use Subroutine

With ON TIMER or ON COM, we had the option of sending control to only one subroutine when the interrupt was triggered. With ON KEY and BASICs before BASIC 2.0, as many as ten subroutines, one for each of the IBM PC and PCjr's special function keys, can be designated. Of course, the arrow keys can also be trapped as well as six other keys (under BASIC 2.0), but these are not often used to access subroutines by the majority of programmers.

The command allows monitoring as many of the function keys as we wish by including an ON KEY(n) statement for each, where n is the function key to be trapped:

```
10 ON KEY(1) GOSUB 1000
20 ON KEY(3) GOSUB 2000
30 ON KEY(6) GOSUB 3000
40 KEY(1) ON
50 KEY(3) ON
60 KEY(6) ON
```

This would activate F1, sending control to line 1000, F3 (line 2000), and F6 (line 3000). Since none of the other keys are defined, the IBM PC or PCjr will ignore them.

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

## Line-by-Line Description

**Lines 110 to 130:** Turn on desired function keys.

**Lines 140 to 160:** Tell BASIC where to branch.

**Lines 190 to 240:** Dummy subroutines carrying out functions.

**Lines 260 to 280:** Dummy program.

## You Supply

Subroutines to carry out desired actions.

## Sample Applications

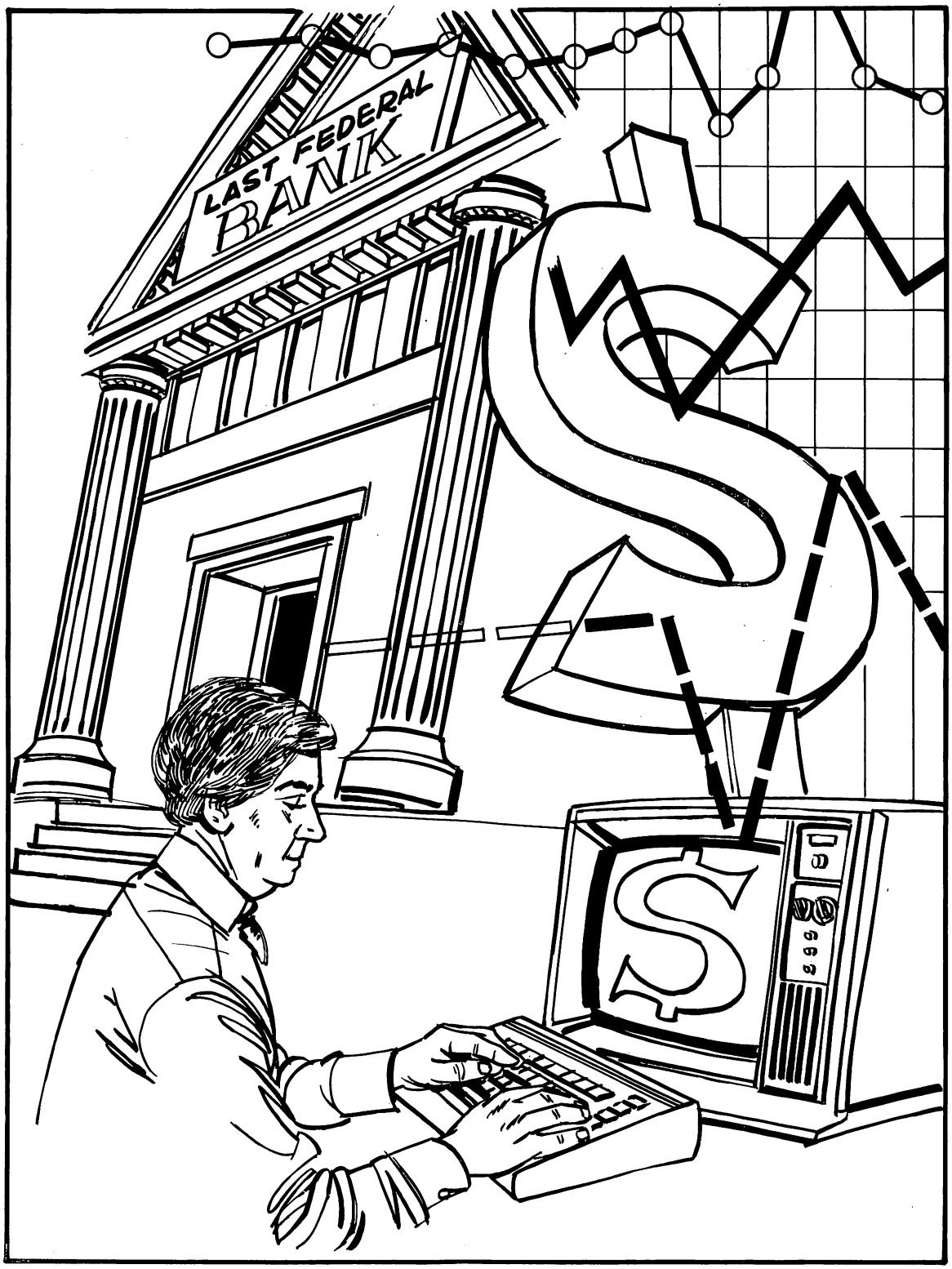
- Allow user to escape from unwanted menu
- Present a Help screen at any time
- Go to another subroutine, such as siren or communications, at any time.

*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(1) ON
120 KEY(2) ON
130 KEY(4) ON
```

```
14Ø ON KEY(1) GOSUB 19Ø
15Ø ON KEY(2) GOSUB 21Ø
16Ø ON KEY(4) GOSUB 23Ø
17Ø GOTO 26Ø

18Ø ' *** SUBROUTINES ***

19Ø PRINT "KEY 1 PRESSED."
2ØØ RETURN
21Ø PRINT "KEY 2 PRESSED."
22Ø RETURN
23Ø PRINT "KEY 4 PRESSED."
24Ø RETURN

25Ø ' *** YOUR PROGRAM STARTS HERE ***

26Ø A$=INKEY$:IF A$="" GOTO 26Ø
27Ø PRINT A$;"KEY PRESSED."
28Ø GOTO 26Ø
```


LAST FEDERAL
BANK

# 5

# Business and Financial Subroutines

The IBM PC is most popular among businesspeople. The PCjr, while aimed at the home market, is also used as a second computer by business folks. Much of the early software for both of these computers was aimed at the businessperson. Although business programs have much in common with games and utilities in BASIC, they also have their own special requirements. A business application will rarely deal with RND but will often have to handle dollars-and-cents. Money matters—figuring loan amounts, monthly payments, interest—and formatting of the output are all important considerations. Business applications also involve recording the date or time when a transaction took place.

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 treated 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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

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

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

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

## Line-by-Line Description

**Lines 150 to 170:** Define the interest RATE, monthly PAYMENT you can afford, and the NUMBER of payments to be made. Your program can substitute INPUT lines to receive these figures from the user.

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

**Line 210:** Calculate loan amount.

**Line 220:** Round off to two decimal places.

**Line 230:** Return to main program.

**Line 240:** Access the subroutine.

## You Supply

You must define these variables:

- PAYMENT (the monthly payment desired)
- RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent)
- NUMBER (number of months loan will run).

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

## Sample Applications

- Figure maximum loan given fixed payments
- Calculate how much down payment needed to reach desired loan given fixed payment amount.

> *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
```

# PAYMENT AMOUNT

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

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

## How to Use Subroutine

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

You might use this subroutine to calculate your monthly auto payment given, say, a 36-month repayment period, a 15 percent interest rate, and an amount to be financed of, say, $8000. It will produce the answer, $277. By shopping around for different interest rates, or varying the number of payments, you can calculate the 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

**Lines 150 to 170:** Define the amount of the LOAN, the interest RATE in whole percent per year, and the NUMBER of monthly payments. Your program can substitute INPUT lines to have this information entered by the user.

**Line 200:** Change RATE to percentage.

**Line 210:** Calculate PAYMENT.

**Line 220:** Round off PAYMENT to two decimal places.

**Line 270:** Print result.

## You Supply

You must define these variables:

- LOAN (the original amount to be financed)
- RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent)
- NUMBER (number of months loan will run).

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

## Sample Applications

- Find out how much your loan payment will be
- Find out how much you will save in monthly payments if you consolidate debts.

> *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
```

```
19Ø ' *** SUBROUTINE ***

2ØØ RATE=RATE/1ØØ
21Ø PAYMENT=LOAN*(RATE/12)/(1-(1+(RATE/12))^-NUMBER)
22Ø PAYMENT=INT(PAYMENT*1ØØ+.5)/1ØØ
23Ø RETURN

25Ø ' *** YOUR PROGRAM STARTS HERE ***

26Ø GOSUB 2ØØ
27Ø PRINT PAYMENT
```

# NUMBER OF PAYMENTS

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- RATE: Interest rate
- LOAN: Amount of loan
- NUMBER: Months of loan
- PAYMENT: Monthly payment
- WP: Number of whole payments
- FP: Amount of final payment.

### How to Use Subroutine

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

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

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

## Line-by-Line Description

**Lines 170 to 190:** Define the amount of LOAN, the interest RATE in whole percent, and the monthly PAYMENT desired. Your subroutine can substitute INPUT statements to have this information supplied by the user.

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

**Line 230:** Calculate number of payments.

**Line 240:** Calculate number of whole payments.

**Line 250:** Figure amount of final, partial payment.

**Line 280:** Access the subroutine.

**Lines 290 to 310:** Print results.

## You Supply

You must define these variables:

- LOAN (the original amount to be financed)
- RATE (interest rate in percent, i.e., 10.5 equals 10.5 percent)
- PAYMENT (the amount of the monthly payment).

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

## Sample Applications

- Find out how long it will take to pay off a loan
- Compare loan lengths at different interest rates
- See how long it will take to pay off a given loan given different payment amounts.

*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-LOAN*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.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

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

## How to Use Subroutine

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

## Line-by-Line Description

**Lines 160 to 190:** Define FUTURE, desired future value, the interest RATE in whole percent per year, and the PERIODS, the number of compounding periods per year. Your subroutine can substitute INPUT statements to allow the user to enter these figures.

**Line 220:** Change RATE to decimal figure.

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

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

**Line 290:** Access the subroutine.

**Lines 300 to 370:** Print results.

## You Supply

You must define these variables:

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

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

## Sample Applications

- See how long it will take to save a certain amount
- Compare to see how long it will take to get a certain amount at different interest rates.

> *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 PERIODS=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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

## How to Use Subroutine

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

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

## Line-by-Line Description

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

**Line 210:** Change RATE to decimal value.

**Line 220:** Calculate FUTURE value.

**Line 230:** Round off value to two decimal places.

**Line 260:** Access the subroutine.

**Lines 270 to 300:** Print results.

## You Supply

You must define these variables:

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

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

## Sample Applications

- Figure future value of savings account
- Calculate future values at compared interest rates.

*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 PERIODS=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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

### Line-by-Line Description

**Lines 150 to 180:** Define the present (or original) value of the investment, the number of YEARS it has or will be compounded, and the FUTURE (or current if the investment is an old one) value. Your subroutine can substitute INPUT lines to have user enter these values.

**Line 210:** Figure interest RATE.

**Line 220:** Change RATE to whole percent.

**Line 250:** Access the subroutine.

**Lines 260 to 310:** Print results.

### You Supply

You must define these variables:

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

The subroutine will return RATE, the interest rate.

## Sample Applications

- Find out the interest rate
- Find the rate of return of an investment.

*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 PERIODS=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
```

# TEMPERATURE

*WHAT IT DOES...*

**Calculates Celsius and Fahrenheit.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- F: Fahrenheit temperature
- C: Celsius temperature.

### How to Use Subroutine

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

### Line-by-Line Description

**Lines 150 to 160:** Figure Fahrenheit temperature.

**Lines 170 to 180:** Figure Celsius temperature.

**Lines 210 to 250:** User enters temperature to convert.

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

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

**Lines 300 to 310:** Access proper subroutine.

**Line 320:** Print results of conversion.

## You Supply

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

## Sample Applications

- Programs which compare temperatures
- Conversion routines.

*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)*(5/9)):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$,1):A=AS(A$):IF A<96 AND A>123 THEN
    A$=CHR$(A-32)
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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

## How to Use Subroutine

This subroutine will accept input of month, day, and year and format it into MM/DD/YY style. That is, December 3, 1947, will be displayed as 03/12/47 or 03/12/1947. As written, the module prompts the operator to enter the values. It disallows illegal months (smaller than one or larger than 12.) Other checks are made to make sure the day of the month is acceptable. For example, June 31 and February 30 are not allowed. February 29 is permitted only during leap years.

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

## Line-by-Line Description

**Line 160:** Enter month to be formatted.

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

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

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

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

**Line 260:** Enter year.

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

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

**Line 330:** Construct MM/DD/YY string.

**Line 370:** Access the subroutine.

**Line 380:** Print result.

## You Supply

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

## Sample Applications

- Business program in which user may enter wrongly formatted date
- Programs in which consistent input is crucial.

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

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

360 CLS:
370 GOSUB 160
380 PRINT DTE$
```

# NUMBER OF DAYS

*WHAT IT DOES...*

**Figures difference between days.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

- M(n): Days more than 28 in each month
- DA: Day number
- DF: Difference

## How to Use Subroutine

Business programs frequently have to calculate the difference between two dates, for example, to figure interest due on an account. This subroutine will figure the number of days between two days you enter in MM/DD format.

An array keeps track of the number of days more than 28 each month has if not a leap year. So, to figure the day of the year for March 3, we multiply 2×28, and then add in M(1), 3, and M(2), 0, plus 3 for the three days elapsed in March, to come up with Day 62. The same thing is done for the second date, and the difference between the two figures is the difference in days.

To keep things simple, this subroutine calculates differences in days within the same year only. To span a year or more, find the day number for the first date, subtract that from 365 (if a not a leap year) to determine the days elapsed in the beginning year. Then, add that figure to the day number of the ending date. If the interval spans more than two years, add 365 or 366 for each additional year.

## Line-by-Line Description

**Line 130:** Dimension array.

**Lines 140 to 160:** Read day data to array.

**Line 190:** Set DA difference to 0.

**Line 200:** Access day calculating module.

**Line 210:** Set start day, D1, to that day number, DA.

**Line 220:** Access day calculating module again.

**Line 230:** Figure difference.

**Line 240:** Display results.

**Line 260:** User enters date.

**Lines 270 to 300:** Extract day and month.

**Lines 340 to 370:** Add up day number.

## You Supply

- Dates to calculate.

## Sample Applications

- Programs which are asked to find data entered between two dates
- Figuring interest or service charges.

*RESULT...*

**Difference between two dates figured.**

```
10 ' ********************
20 ' *                  *
30 ' * NUMBER OF DAYS   *
40 ' *                  *
50 ' ********************
60 ' ---------------------------------------
70 '    ++ VARIABLES ++
80 '    M(n): DAYS MORE THAN 28 IN EACH MONTH
90 '    DA:   DAY NUMBER
100 '   DF:   DIFFERENCE
110 ' ---------------------------------------
```

```
120 DATA 3, 0, 3, 2, 3, 2, 3, 3, 2, 3, 2, 3
130 DIM M(12)
140 FOR N=1 TO 12
150 READ M(N)
160 NEXT N
170 GOTO 400


180 ' *** SUBROUTINE ***


190 DA=0
200 GOSUB 260
210 D1=DA
220 GOSUB 260
230 DF=DA-D1
240 PRINT "Days difference: ";DF
250 RETURN
260 INPUT"ENTER DATE (MM/DD)";DA$:IF MID$(DA$,3,1)<>"/"
    THEN PRINT "USE MM/DD FORMAT!":GOTO 260
270 M=(VAL(LEFT$(DA$, 2)):D=VAL(RIGHT$(DA$, 2))
280 IF M=4 OR M=6 OR M=9 OR M=11 AND D>30 THEN GOTO 300
290 IF D>31 THEN GOTO 300 ELSE IF M<>2 OR M<29 THEN GOTO
    310
300 PRINT"Improper date!":GOTO 260
310 GOSUB 330
320 RETURN
330 FA=0:IF M=1 GOTO 370
340 FOR N=1 TO M-1
350 FA=FA+M(N)
360 NEXT N
370 DA=28*(M-1)+FA+D
380 RETURN


390 ' *** YOUR PROGRAM STARTS HERE ***


400 GOSUB 190
```

# DAY CONVERTER

*WHAT IT DOES...*

**Translates day number to date.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- M$(N): Name of the months
- DA: Day number.

### How to Use Subroutine

Use to convert day number back into calendar date. As written, routine produces full month name and day.

### Line-by-Line Description

**Line 120:** Dimension array to store month names.

**Lines 130 to 150:** Read month names into array.

**Line 210:** User enters day number.

**Line 220:** Access date calculator routine.

**Line 230:** Subtract number of days to that month from total day number to produce day of the month.

**Line 240:** Print results.

**Lines 250 to 390:** Figure month from days elapsed.

### You Supply

Day numbers to convert.

### Sample Applications

- Programs which access data in day number form, but user desires to see conventional date.

*RESULT...*

**Day number converted to date.**

```
10 ' ********************
20 ' *                  *
30 ' *  DAY CONVERTER   *
40 ' *                  *
50 ' ********************
60 ' ----------------------------
70 '    ++ VARIABLES ++
80 '    M$(N): NAME OF THE MONTHS
90 '    DA:    DAY NUMBER
100 '
110 ' ----------------------------
120 DIM M$(12)
130 FOR N=1 TO 12
140 READ M$(N)
150 NEXT N
160 DATA January, February, March, April, May
170 DATA June, July, August, September
180 DATA October, November, December
190 GOTO 410

200 ' *** SUBROUTINE ***

210 INPUT"Enter day number :";DA
220 GOSUB 260
230 D=DA-F1
240 PRINT "Date is ";M$(M);" ";D
250 RETURN
260 IF DA>334 THEN M=12: F1=334: RETURN
270 IF DA>304 THEN M=11: F1=304: RETURN
280 IF DA>273 THEN M=10: F1=273: RETURN
290 IF DA>243 THEN M=9: F1=243: RETURN
300 IF DA>212 THEN M=8: F1=212: RETURN
310 IF DA>181 THEN M=7: F1=181: RETURN
320 IF DA>151 THEN M=6: F1=151: RETURN
330 IF DA>120 THEN M=5: F1=120: RETURN
```

```
340 IF DA>90 THEN M=4: F1=90: RETURN
350 IF DA>59 THEN M=3: F1=59: RETURN
360 IF DA>31 THEN M=2: F1=31: RETURN
370 M=1
380 F1=0
390 RETURN

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

410 GOSUB 210
```

# MENU

*WHAT IT DOES...*

**Menu template for user programs.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

- NC$: Number of choices on menu.

## How to Use Subroutine

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

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

Once the operator has selected a menu item, the routine branches to modules written by the user to carry out the menu functions. To expand the number of menu items, redefine NC. If more than nine choices are listed, you will have to sacrifice single key entry. Replace line 240 with INPUT A$. Then any number can be entered.

Note that no menu functions are provided at lines 1000, 2000, 3000, and 4000; you must write those routines yourself.

## Line-by-Line Description

**Line 70:** Define number of menu choices available.

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

**Lines 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.

## Sample Applications

- Any program which uses menus instead of commands
- Program where you wish to limit choice of actions
- Simple programs that are friendly to new users.

*RESULT...*

**Operator can select from list of menu choices.**

```
1Ø ' ********
2Ø ' *      *
3Ø ' * MENU *
4Ø ' *      *
5Ø ' ********

6Ø ' *** INITIALIZE ***

7Ø NC=4
8Ø GOTO 28Ø
```

```
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$="" GOTO 220
230 A=VAL(A$)
240 IF A<1 OR A>NC GOTO 210
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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

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

### Line-by-Line Description

**Lines 160 to 180:** Define the current total minutes, hours, and seconds.

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

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

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

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

**Line 340:** Access the subroutine.

**Lines 350 to 380:** Print the results.

## You Supply

You must supply startup 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 zero if you wish. Your program should furnish MIN, HOUR, and SECS values.

## Sample Applications

- Programs where two times are supplied and the total of the two are desired
- Finding out the total elapsed time from several timings
- Timing record album selections to see how long of a tape to use to record them.

*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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- BEGN: Starting odometer reading
- ODOM: Current odometer reading
- GALLNS: Gallons of gas consumed between readings.

### How to Use Subroutine

This routine will figure your gas consumption given the starting and ending odometer readings and number of gallons of gas consumed. To be accurate, you should top off your gas tank before writing down BEGN value and top it off again

when recording ODOM. Any gas put in between those two should be added to the final fill-up. In other words, the MPG can be figured for the aggregate of a number of tanksful of gas.

## Line-by-Line Description

**Lines 140 to 160:** Define current ODOMeter reading, the BEGN or initial odometer reading, and the number of gallons, GALLNS, of gas used. Your subroutine can use INPUT statements to allow the user to enter these values. 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.

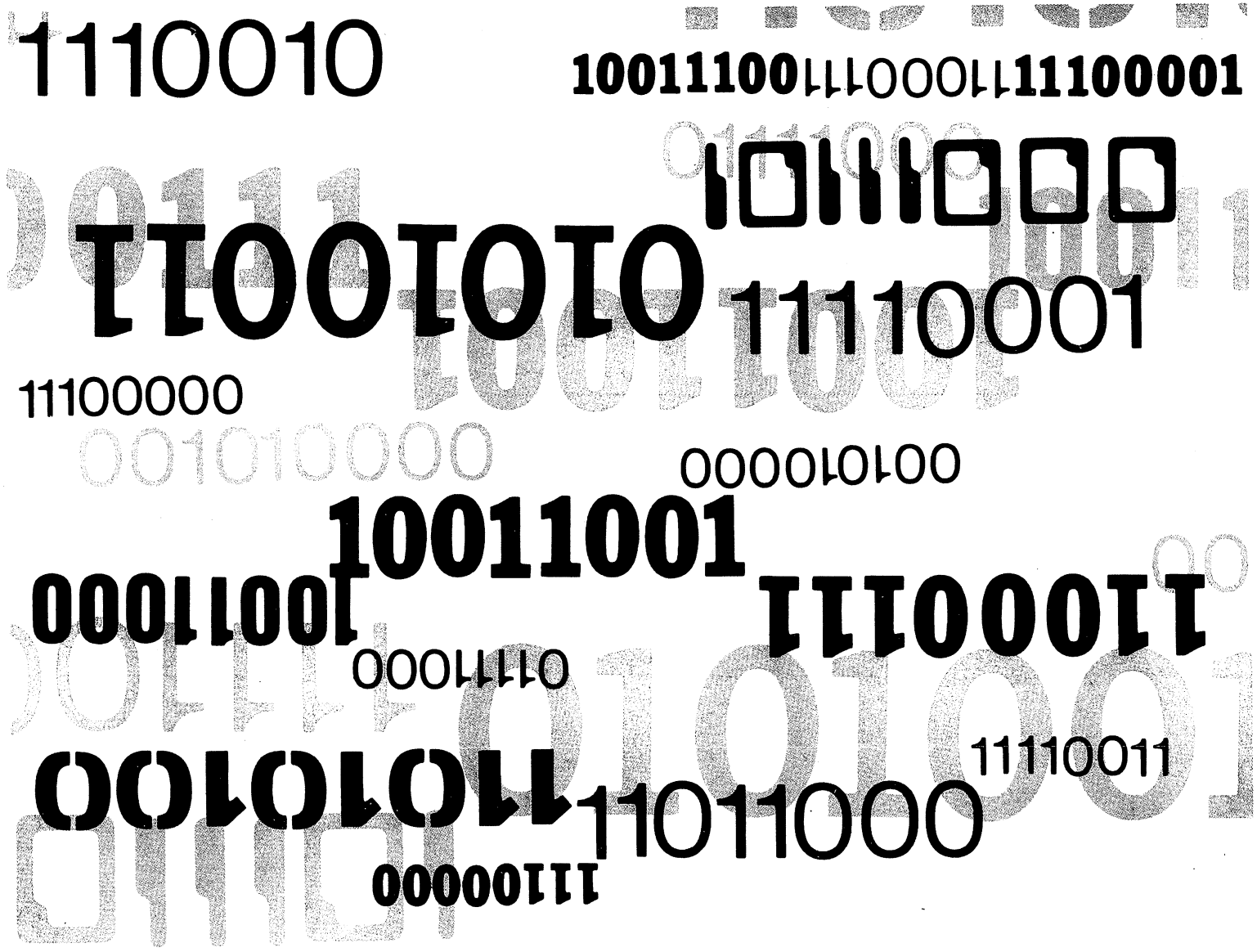
## Sample Applications

- Figuring out how economical a car is
- Comparing economy of two or more cars.

> *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 STARTS HERE ***

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


1110010
10011100
11100001
11100000
10011001
11011000
11110011

# 6

# 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 POKEing a number into a memory location and changing the contents of bits that do not concern you, use the "soft" POKEing 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 choose. Another routine, "Bit Displayer", prints the eight bits of 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.

This chapter is the only one in the book that does not include separate SAMPLE APPLICATIONS. For the most part, the routines in this suggestion all perform a bit manipulation of some type. When you need one, you'll know what for. These applications can include performing bit-level operations on various PC and PCjr registers. For example, the tone generators of the PCjr use the status of individual bits to control pitch and volume of each voice. While you can control these with BASIC statements, more advanced programmers not interested in doing a program 100 percent in machine language might want to do some playing with these bits from a BASIC program for various special effects.

## PEEK BIT

*WHAT IT DOES...*

**Looks at status, 0 or 1, of any selected bit in a given byte.**

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- ADDRESS: Location to PEEK
- BIT: Bit to examine
- V: Value of that bit, either 0 or 1.

## How to Use Subroutine

You can get maximum mileage from your IBM PC and PCjr'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 Line 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.**

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- ADDRESS: Location to POKE
- BIT$: Bit pattern.

## How to Use Subroutine

This subroutine will display all of the bits within a byte. Each position will be indicated by a one or a zero.

NOTE: The caret symbol (^) indicates the 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 deliberately want to be difficult to change, e.g., protection purposes.

This subroutine will also serve as a means of converting positive integers smaller than 256 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 ' ****************
```

```
6Ø ' *** INITIALIZE ***

7Ø ADDRESS=36879
8Ø GOTO 26Ø
9Ø ' ---------------------------
1ØØ '    ++ VARIABLES ++
11Ø '     ADDRESS: MEMORY BYTE
12Ø '              TO DISPLAY
13Ø '     BIT$:   BIT PATTERN
14Ø '
15Ø ' ---------------------------

16Ø ' *** SUBROUTINE ***

17Ø BIT$=""
18Ø PRINTTAB(4)"";
19Ø FOR N=7 TO Ø STEP-1
2ØØ V=(PEEK(ADDRESS)AND(2^N))/(2^N)
21Ø G$=MID$(STR$(V),2)
22Ø BIT$=BIT$+G$
23Ø NEXT N
24Ø RETURN

25Ø ' *** YOUR PROGRAM STARTS HERE ***

26Ø PRINT
27Ø GOSUB 17Ø
28Ø PRINT"ADDRESS: ";ADDRESS
29Ø PRINT PEEK(ADDRESS);"="
3ØØ 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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- ADDRESS: Location to POKE
- BIT$: Bit to change to one.

### How to Use Subroutine

This subroutine will take any bit within a byte, and change that value to one, regardless of what it was before. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the IBM PC and PCjr.

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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- ADDRESS: Location to POKE
- BIT$: Bit to change to zero.

## How to Use Subroutine

This subroutine will take any bit within a byte and change that value to zero regardless of what it was before. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the IBM PC and PCjr.

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 ' ---------------------------
```

```
16Ø ' *** SUBROUTINE ***

17Ø POKE ADDRESS,PEEK(ADDRESS)AND(255-(2^BIT))
18Ø RETURN

19Ø ' *** YOUR PROGRAM STARTS HERE ***

2ØØ PRINT
21Ø GOSUB 17Ø
```

# 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.**

## Versions: IBM PC and PCjr, Advanced BASIC

## Variables

- ADDRESS: Location to POKE
- BIT: Bit to reverse.

## How to Use Subroutine

This subroutine will take any bit within a byte and change that value to the opposite of what it was before. If the bit was one, it will be changed to zero. A zero bit will be given a value of one. None of the other bits within the byte will be altered. This ability is useful for toggling certain features within a multipurpose byte that may also be used to control other parameters of the IBM PC and PCjr. Using this subroutine, it is not necessary to know whether the feature is on or off. "Reverse Bit" will change it to the other status automatically.

NOTE: The caret symbol (^) indicates the 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.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- A$: Binary number in string form
- A: Decimal equivalent.

### How to Use Subroutine

Several of the subroutines in this book—and many more that you will prepare—will require supplying decimal equivalents of binary numbers. 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 the 1's position within the byte.

**Lines 200-210:** Ask user for binary number to convert.

**Lines 220 to 270:** Check for presence of illegal characters.

**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 A=0
160 FOR N=1 TO LEN(A$):P=LEN(A$)-N:A=A+2^P*
    VAL(MID$(A$,N,1))
170 NEXT N
180 RETURN
```

```
19Ø ' *** YOUR PROGRAM STARTS HERE ***

2ØØ CLS
21Ø INPUT "ENTER NUMBER TO CONVERT: ";A$
22Ø FOR N=1 TO LEN(A$)
23Ø T$=MID$(A$,N,1)
24Ø IF T$="Ø" OR T$="1" GOTO 27Ø
25Ø PRINT "NOT BINARY NUMBER"
26Ø GOTO 21Ø
27Ø NEXT N
28Ø A=Ø:GOSUB 15Ø
29Ø PRINT A$"=";A
```

# ROUNDER

*WHAT IT DOES...*

**Rounds positive number, and cuts off after desired number of decimal places.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- A: Number to be rounded
- P: Digits desired to right of decimal point
- B: Rounded value.

### How to Use Subroutine

The IBM PC and PCjr sometimes provide a great deal more precision in calculations 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

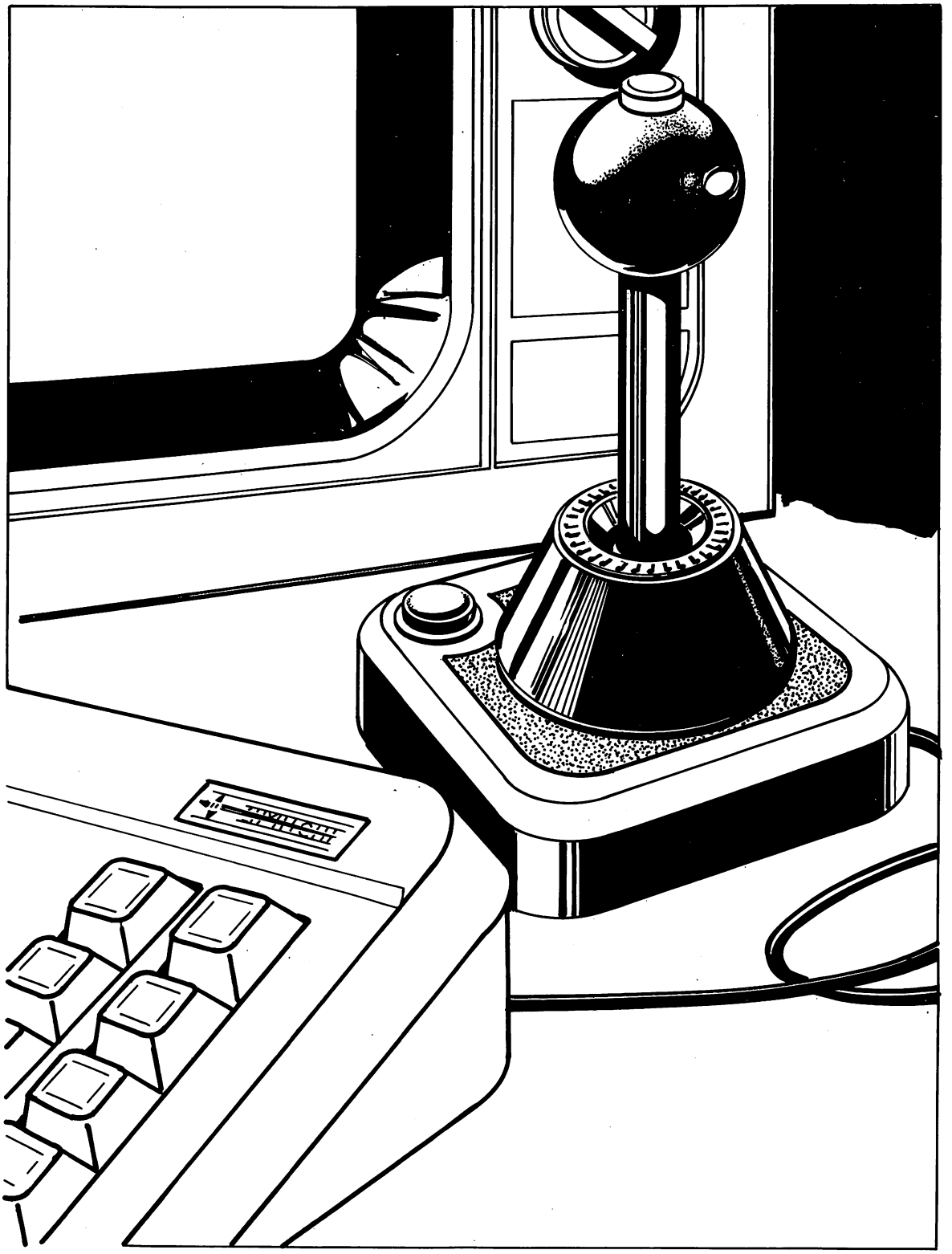
You should define A to be the number to be rounded. P will equal the number of digits to the right of the decimal point that you want. The subroutine will return B, the rounded value. If B has a fractional decimal part that ends in zero, the zero will not be printed, even though that many decimal places 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
```

```
18Ø ' *** SUBROUTINE ***

19Ø C=A+5.5*1Ø^-(P+1)
2ØØ B=INT(C*1Ø^P)/1Ø^P
21Ø RETURN

22Ø ' *** YOUR PROGRAM STARTS HERE ***

23Ø GOSUB 19Ø
24Ø PRINT B
```



# 7

# Joysticks and Paddles

No, joysticks are not just for games. Though these devices are now most frequently seen in arcade-style shoot-em-ups, they have many applications in business, engineering, and science. A joystick is an excellent input device for positioning the cursor on the screen with some accuracy.

Joysticks have important potential uses in many business and other programs. Like mice and digital pads, a joystick is just another useful input device for your IBM computer. The PCjr, of course, has a joystick port built in. This feature can be added to the PC or XT by plugging in a so-called "games" adapter.

This chapter covers the basics of using joysticks and joystick-like input to control objects on the screen of your computer. Six subroutines are provided. Four are aimed at you PC and XT users who have no need or desire to use true joysticks but would still like to simulate their action. These subroutines use the numeric/cursor pad to allow directional input. 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 ersatz joystick breaks down into several neat modules. We need to know where on the IBM PC and PCjr's screen the object to be moved is. We also must check, as often as possible, the status of the player's joystick to see if it is pressed in any direction, or if the FIRE button has been depressed. If so, the programmer must update the location of the object on the screen. This is usually done by changing the value of the LOCATE statement where the object is printed by adding or subtracting from the rows and columns. 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.

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 from side to side to simulate a horizontal paddle (like Breakout-type games). Another mimics the original PONG game, with its vertical paddles. A third subroutine allows moving an object in the four primary directions of a compass—north, south, east, and west. This would be helpful for maze-type games of the PacMan ilk. Those of you with BASIC 2.0 can also gain northeast, southeast, northwest, and southwest input through a special keytrapping subroutine.

Bringing all these functions together is a drawing subroutine—usable on monochrome monitors—that allows drawing on the screen with any alphanumeric or other IBM character.

Finally, a true joystick reading routine is presented that returns the value of the IBM joysticks as well as the FIRE buttons. Another routine is included that allows interrupting a program and branching to a subroutine of your choice whenever any of the four FIRE buttons are pressed.

# HORIZONTAL PADDLE SIMULATION

*WHAT IT DOES...*

**Moves object side to side only.**

## Versions: IBM PC, Advanced BASIC

### Variables

- ROW: Row of cursor
- COL: Column of cursor
- CURSR: Cursor character
- W: Width of screen.

### How to Use Subroutine

Currently, few paddles are marketed for the IBM PC and PCjr. However, the PC has a cursor pad that can be used to simulate side-to-side movement as used in Breakout-type games. The PCjr has no numeric pad although some programs, such as Homeword, use the Pg Up, Pg Dwn, Home, and End keys to simulate directional arrows.

Although true paddles use a potentiometer to constantly tell the computer the proper position of the cursor, some games and applications work better when only the direction of movement, not actual position, is indicated, using the arrow keys. This approach makes games programming simpler. Move an object one character to the right when the right arrow is pressed and one character to the left when the left arrow is pressed.

This subroutine will tell you whether or not either case has occurred. Your program should repeatedly check to see if the value of COL has changed and take action accordingly. Your object might be located in one position, defined as ROW and COL. The appropriate variable, in this case COL, can be updated each time a right or left arrow key is depressed. Make sure that COL never exceeds the end of the screen when PRINTing objects to the screen.

### Line-by-Line Description

**Lines 160 to 180:** Clear screen, define width of screen.

**Line 190:** Turn off 25th line display.

**Line 200:** Define cursor as CHR$(220) in the IBM character set.

**Lines 210 to 220:** Define initial column as 1 and row for movement as 10.

**Line 230:** Activate trapping for left and right arrow key, (KEY(12) and KEY(13)).

**Lines 240 to 250:** Tell program where to go when those keys are pressed.

**Lines 280 to 290:** Check to see that the current cursor column is not more than W, or less than 1, and fix if it is.

**Line 310:** LOCATE cursor at ROW,COL, and print a space to erase old character at that position.

**Line 320:** Reduce COL by 1, moving object to left.

**Line 340:** LOCATE cursor at ROW,COL, and print a space to erase old character at that position.

**Line 350:** Increase COL by 1, moving object right.

**Line 380:** Locate cursor at new value of ROW,COL, and print cursor character, CHR$(CURSR), there.

## You Supply

Just press arrow keys. ROW can be defined as the screen column on which the object moves. CURSR can be defined as any character you wish.

## Sample Applications

- Breakout-type games
- Moving cursor for horizontal menus
- Editing functions within applications program.

*RESULT...*

**Object will move on screen under cursor pad control from side to side only.**

```
10 ' ******************
20 ' *                *
30 ' *  MOVE OBJECTS  *
40 ' *  SIDE TO SIDE  *
50 ' *                *
60 ' ******************
```

```
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '     ROW:   ROW OF CURSOR
110 '     COL:   COLUMN OF CURSOR
120 '     CURSR: CURSOR CHARACTER
130 '
140 ' ---------------------------

150 ' *** INITIALIZE ***

160 CLS
170 W=79
180 SCREEN 0,0,0
190 KEY OFF
200 CURSR=220
210 COL=1
220 ROW=10
230 KEY(12) ON:KEY(13) ON
240 ON KEY(12) GOSUB 310
250 ON KEY(13) GOSUB 340
260 GOTO 380

270 ' *** SUBROUTINE ***

280 IF COL>W THEN COL=W
290 IF COL<1 THEN COL=1
300 GOTO 380
310 LOCATE ROW,COL:PRINT CHR$(32);
320 COL=COL-1
330 RETURN 280
340 LOCATE ROW,COL:PRINT CHR$(32);
350 COL=COL+1
360 RETURN 280

370 ' *** YOUR PROGRAM STARTS HERE ***

380 LOCATE ROW,COL:PRINT CHR$(CURSR)
390 GOTO 380
```

# VERTICAL PADDLE SIMULATION

*WHAT IT DOES...*

**Moves object up and down only.**

## Version: IBM PC, Advanced BASIC

### Variables

- ROW: Row object moves in
- COL: Column to move object in
- CURSR: Cursor character.

### How to Use Subroutine

There are currently few paddles marketed for the IBM PC and PCjr. However, the PC has a cursor pad that can be used to simulate up and down movement, as used in PONG-type games. The PCjr has no numeric pad, although some programs, such as Homeword, use the Pg Up, Pg Dwn, Home, and End keys to simulate directional arrows.

Although true paddles use a rheostat to constantly tell the computer the proper position of the cursor, some games and applications work better when only the direction of movement, not actual position, is indicated using the arrow keys. This approach makes games programming simpler. Move an object up one row when the up arrow is pressed and down one row when the down arrow is pressed.

This subroutine will tell you whether or not either case has occurred. Your program should repeatedly check to see if ROW has changed and take action accordingly. Your object might be located along one side of the screen, in a position defined by the variables ROW and COL. ROW can be changed each time an up or down arrow key is pressed. Make sure that ROW is never less than 1 or larger than the bottom row of the screen (usually 24 or 25).

### Line-by-Line Description

**Lines 160 to 180:** Clear screen, define width of screen.

**Line 190:** Turn off 25th line display.

**Line 200:** Define cursor as CHR$(220) in the IBM character set.

**Lines 210 to 220:** Define initial row as 1, and column for movement as 10.

**Line 230:** Activate trapping for up and down arrow key, (KEY(11) and KEY(14)).

**Lines 240 to 250:** Tell program where to go when those keys are pressed.

**Lines 280 to 290:** Check to see that the current cursor row is not more than 24, or less than 1, and fix if it is.

**Line 310:** LOCATE cursor at ROW,COL, and print a space to erase old character at that position.

**Line 320:** Reduce ROW by 1, moving object up.

**Line 340:** LOCATE cursor at ROW,COL, and print a space to erase old character at that position.

**Line 350:** Increase ROW by 1, moving object down.

**Line 380:** Locate cursor at new value of ROW,COL, and print cursor character, CHR$(CURSR) there.

## You Supply

Just press arrow keys. COL can be defined as the screen column on which the object moves. CURSR can be defined as any character you wish.

## Sample Applications

- Pong-type games
- Moving cursor through vertical menus
- Editing functions within applications program.

*RESULT...*

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

```
10 ' ******************
20 ' *                *
30 ' *  MOVE OBJECTS  *
40 ' *  UP AND DOWN   *
50 ' *                *
60 ' ******************
```

```
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '     ROW:   ROW OF CURSOR
110 '     COL:   COLUMN OF CURSOR
120 '     CURSR: CURSOR CHARACTER
130 '
140 ' ---------------------------

150 ' *** INITIALIZE ***

160 CLS
170 W=79
180 SCREEN 0,0,0
190 KEY OFF
200 CURSR=220
210 COL=10
220 ROW=1
230 KEY(11) ON:KEY(14) ON
240 ON KEY(11) GOSUB 310
250 ON KEY(14) GOSUB 340
260 GOTO 380

270 ' *** SUBROUTINE ***

280 IF ROW>24 THEN ROW=24
290 IF ROW<1 THEN ROW=1
300 GOTO 380
310 LOCATE ROW,COL:PRINT CHR$(32);
320 ROW=ROW-1
330 GOTO 380
340 LOCATE ROW,COL:PRINT CHR$(32);
350 ROW=ROW+1
360 RETURN 280

370 ' *** YOUR PROGRAM STARTS HERE ***

380 LOCATE ROW,COL:PRINT CHR$(CURSR);
390 GOTO 380
```

# CURSOR PAD JOYSTICK—N,S,E,W

*WHAT IT DOES...*

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

## Versions: IBM PC, Advanced BASIC

### Variables

- ROW: Current cursor row
- COL: Current cursor column
- W: Width of screen.

### How to Use Subroutine

The cursor pad 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-type input to control the actions of your programs. Usually, the input from the cursor pad keys 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 keys 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.

The subroutine uses Advanced BASIC's "key trapping" capabilities. The Intel 8088 microprocessor used in the IBM PC and PCjr computers can detect certain events, and interrupt a program if directed to. (Interrupts are discussed more thoroughly in Chapter 2.) This feature is activated by including a KEY(n) ON statement where n equals the number assigned to that key. Keys 1-10 are the function keys, while keys 11-14 are the cursor pad arrow keys. By specifying the proper KEY(n) ON statements and then supplying an appropriate ON KEY(n) GOSUB... statement, we can tell the program where to go whenever that key is pressed. Then, no

matter what the program is doing at that point, it will be interrupted and control will pass to the designated subroutine. Since we usually have no way of knowing where the program was at the time, it is desirable to send the program back to a specific place, using "RETURN linenumber" when finished with the interrupt routine.

Call the subroutine whenever you wish to check on the status of the joysticks. 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.

## Line-by-Line Description

**Lines 150 to 170:** Clear screen, define width of screen.

**Line 180:** Turn off 25th line display.

**Line 190:** Define initial row and column as 1 (upper left hand corner of screen.)

**Line 200:** Activate trapping for up, right, left, and down arrow keys, (KEY(11), KEY(12), KEY(13) and KEY(14)).

**Lines 210 to 240:** Tell program where to go when those keys are pressed.

**Lines 270 to 300:** Check to see that the current cursor row is not more than 24, or less than 1, nor that the current column is less than 1 or greater than the width of the screen, and fix if it is.

**Line 320:** Reduce ROW by 1, moving object up.

**Line 340:** Reduce COL by 1, moving object left.

**Line 360:** Increase ROW by 1, moving object down.

**Line 380:** Increase COL by 1, moving object right.

**Line 410:** Locate cursor at new value of ROW,COL, and print cursor character there.

## You Supply

Just press arrow keys to move object.

## Sample Applications

- Maze games with four directional movement
- Drawing of right angles on screen
- Line-oriented cursor movement.

*RESULT...*

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

```
10 ' ********************
20 ' *                  *
30 ' *  MOVE OBJECTS    *
40 ' *  N,S,E, and W    *
50 ' *                  *
60 ' ********************
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '     ROW: ROW OF CURSOR
110 '     COL: COLUMN OF CURSOR
120 '
130 ' --------------------------


140 ' *** INITIALIZE ***


150 CLS
160 W=79
170 SCREEN 0,0,0
180 KEY OFF
190 ROW=1:COL=1
200 KEY(11) ON:KEY(12) ON:KEY(13) ON:KEY(14) ON
210 ON KEY(11) GOSUB 320
220 ON KEY(12) GOSUB 340
230 ON KEY(13) GOSUB 360
240 ON KEY(14) GOSUB 380
250 GOTO 410
```

```
26Ø ' *** SUBROUTINE ***

27Ø IF ROW>24 THEN ROW=24
28Ø IF ROW<1 THEN ROW=1
29Ø IF COL>W THEN COL=W
3ØØ IF COL<1 THEN COL=1
31Ø GOTO 41Ø
32Ø ROW=ROW-1
33Ø RETURN 27Ø
34Ø COL=COL-1
35Ø RETURN 27Ø
36Ø COL=COL+1
37Ø RETURN 27Ø
38Ø ROW=ROW+1
39Ø RETURN 27Ø

4ØØ ' *** YOUR PROGRAM STARTS HERE ***

41Ø LOCATE ROW,COL:PRINT"*";
42Ø GOTO 41Ø
```

## CURSOR PAD JOYSTICK—ALL DIRECTIONS

*WHAT IT DOES...*

**Moves object in eight compass points.**

### Versions: IBM PC, Advanced BASIC 2.0

### Variables

- ROW: Current cursor row
- COL: Current cursor column.

## How to Use Subroutine

In the previous subroutine, Advanced BASIC's key trapping routine was used to send control to various direction-governing modules depending on which of the four arrow keys were pressed. This subroutine allows movement in eight compass directions.

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.

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. All eight compass directions are permitted with this routine.

The subroutine uses Advanced BASIC's "key trapping" capabilities. The Intel 8088 used in the IBM computers has the ability to look for certain events, and interrupt a program if directed to. (Interrupts are discussed more thoroughly in Chapter 2.) The feature is activated by including a KEY(n) ON statement where n equals the number assigned to that key. Keys 1-10 are the function keys, while keys 11-14 are the cursor pad arrow keys. By specifying the proper KEY(n) ON statements, and then supplying an appropriate ON KEY(n) GOSUB... statement, we can tell the program where to go whenever that key is pressed. Then, no matter what the program is doing at that point, it will be interrupted and control will pass to the designated subroutine. Since we usually have no way of knowing where the program was at the time, it is desirable to send the program back to a specific place, using "RETURN linenumber" when finished with the interrupt routine.

BASIC 2.0 also has the capability of trapping six other keys, KEY(15-20), but with the added feature of allowing the user to decide which keys will be trapped. In other words, we may define Key 15 as any key or key combination on the keyboard, including shift, control, alternate, or a combination of these. Each of the so-called "shift" type keys has a code, and the normal keys have "scan codes" of their own, which can be found in Appendix K of the BASIC manual.

We redefine KEY(15-20) by assigning two strings to it, the first being the "shift" state, and the second being the scan code of the key itself. Here is an example:

```
KEY(15),CHR$(&H04)+CHR$(30)
```

The code for the control key is &H04, and 30 is the scan code (different from ASCII code) for the "A" key. So, we can trap CTRL-A using KEY(15) ON and ON KEY(15) GOSUB...

Other shift-type keys have codes of their own, with CAPS LOCK being &H40; NUM LOCK, &H20; Alt, &H08; and SHIFT either &H01, &H02, or &H03. Scan codes are included for all the keys on the keyboard, including Pg Up, etc. (which roughly correspond to northeast, and so forth, on the cursor pad). So we can trap for these as well. No "shift" is desired, so CHR$(0) is used in place of one of the other shift-type codes.

Once we have set up the interrupt routines for the four "corner keys" on the cursor pad, the routine proceeds exactly like the previous one, except that diagonal movement produces a change in both ROW and COL to account for movement to, say, both the right and up when the Pg Up key is pressed. Call the subroutine whenever you wish to check on the status of the joysticks. This subroutine prints an asterisk character on the screen. You may change the character by substituting some other character for the asterisk.

## Line-by-Line Description

**Lines 150 to 170:** Clear screen, set width.

**Line 180:** Turn off 25th line display.

**Line 190:** Set initial row and column to 1.

**Lines 200 to 210:** Turn on KEY(11-18).

**Lines 220 to 250:** Redefine KEY(15-18) as cursor corner keys.

**Lines 270 to 330:** Tell program which subroutines to use when keys are trapped.

**Lines 360 to 390:** If ROW or COL are outside screen boundaries, fix it.

**Line 410:** Decrease ROW by one, moving object up.

**Line 430:** Decrease COL by one, moving object left.

**Line 450:** Increase COL by one, moving object right.

**Line 480:** Increase ROW by one, moving object down.

**Line 490:** Decrease ROW by one, increase COL by one, moving object northeast.

**Line 510:** Decrease both ROW and COL by one, moving object northwest.

**Line 530:** Increase ROW by one, decrease COL by one, moving object southwest.

**Line 550:** Increase both ROW and COL by one, moving object southeast.

**Line 580:** Locate cursor at updated position of ROW, COL, and print asterisk there.

## You Supply

Just use cursor pad keys to move object.

## Sample Applications

- Games requiring full screen movement
- Drawing programs, computer design
- Fast positioning of cursor.

> *RESULT...*
>
> **Object will move on screen under cursor pad control in all eight compass directions.**

```
10 ' ********************
20 ' *                  *
30 ' *  MOVE OBJECTS    *
40 ' *  ALL DIRECTIONS  *
50 ' *                  *
60 ' ********************
70 ' --------------------------
80 '    ++ VARIABLES ++
90 '
100 '     ROW: ROW OF CURSOR
110 '     COL: COLUMN OF CURSOR
120 '
130 ' --------------------------

140 ' *** INITIALIZE ***

150 CLS
160 W=79
170 SCREEN 0,0,0
180 KEY OFF
190 ROW=1:COL=1
```

```
200 KEY(11) ON:KEY(12) ON:KEY(13) ON:KEY(14) ON
210 KEY(15) ON:KEY(16) ON:KEY(17) ON:KEY(18) ON
220 KEY 15,CHR$(0)+CHR$(&H49)
230 KEY 16,CHR$(0)+CHR$(&H47)
240 KEY 17,CHR$(0)+CHR$(&H4F)
250 KEY 18,CHR$(0)+CHR$(&H51)
260 ON KEY(11) GOSUB 410
270 ON KEY(12) GOSUB 430
280 ON KEY(13) GOSUB 450
290 ON KEY(14) GOSUB 470
300 ON KEY(15) GOSUB 490
310 ON KEY(16) GOSUB 510
320 ON KEY(17) GOSUB 530
330 ON KEY(18) GOSUB 550
340 GOTO 580


350 ' *** SUBROUTINE ***


360 IF ROW>24 THEN ROW=24
370 IF ROW<1 THEN ROW=1
380 IF COL>W THEN COL=W
390 IF COL<1 THEN COL=1
400 GOTO 580
410 ROW=ROW-1
420 RETURN 360
430 COL=COL-1
440 RETURN 360
450 COL=COL+1
460 RETURN 360
470 ROW=ROW+1
480 RETURN 360
490 ROW=ROW-1:COL=COL+1
500 RETURN 360
510 ROW=ROW-1:COL=COL-1
520 RETURN 360
530 ROW=ROW+1:COL=COL-1
540 RETURN 360
550 ROW=ROW+1:COL=COL+1
560 RETURN 360
```

```
570 ' *** YOUR PROGRAM STARTS HERE ***

580 LOCATE ROW,COL:PRINT"*";
590 GOTO 580
```

# DRAWING SUBROUTINE

*WHAT IT DOES...*

**Draws on screen, changing cursor character as desired.**

## Versions: IBM PC, Advanced BASIC or BASIC 2.0

### Variables

- ROW: Current cursor row
- COL: Current cursor column
- CURSR: Current cursor character.

### How to Use Subroutine

This subroutine, similar to the last, has the added feature of automatically changing the cursor character, under direction of the operator.

You can adapt this subroutine to many different types of programs, such as drawing floor plans, and other design projects.

Pressing any alpha key will cause the cursor to change to that letter or, with the shift or control key, other characters of the IBM set can be used.

### Line-by-Line Description

**Lines 150 to 180:** Initialize cursor character, screen width, and clear screen.

**Line 190:** Turn off 25th line display.

**Line 200:** Set initial row and column to 1.

**Line 210:** Turn on Keys 11-14 (arrow keys).

**Lines 220 to 250:** See if BASIC 2.0 is running.

**Line 260:** If so, turn on Keys 15-18.

**Lines 270 to 300:** Redefine Keys 15-18 as cursor pad corner keys.

**Lines 310 to 380:** Tell program which subroutines to access if keys are pressed.

**Lines 410 to 440:** If cursor would go beyond screen limits, fix.

**Line 460:** Decrease ROW by one, moving object up.

**Line 480:** Decrease COL by one, moving object left.

**Line 500:** Increase COL by one, moving object right.

**Line 520:** Increase ROW by one, moving object down.

**Line 540:** Decrease ROW by one, increase COL by one, moving object northeast.

**Line 560:** Decrease both ROW and COL by one, moving object northwest.

**Line 580:** Increase ROW by one, decrease COL by one, moving object southwest.

**Line 600:** Increase both ROW and COL by one, moving object southeast.

**Line 630:** Locate cursor at updated position of ROW, COL, and print CHR$(CURSR) there.

**Line 640:** Wait for other key to be pressed.

**Line 650:** Change cursor to key pressed.

**Line 660:** If non-character key pressed, restore cursor to original value.

## You Supply

Move cursor with cursor controls, change it by pressing other keys.

## Sample Applications

- Computer-assisted drafting
- Children's educational programs
- Laying out rough screens and menus.

*RESULT...*

**Drawing on screen with variety of cursor characters.**

```
10 ' **********
20 ' *        *
30 ' *  DRAW  *
40 ' *        *
50 ' **********
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '
90 '      ROW:    ROW OF CURSOR
100 '     COL:    COLUMN OF CURSOR
110 '     CURSR:  CURSOR CHARACTER
120 '
130 ' --------------------------

140 ' *** INITIALIZE ***

150 CLS
160 CURSR=8
170 W=79
180 SCREEN 0,0,0
190 KEY OFF
200 ROW=1:COL=1
210 KEY(11) ON:KEY(12) ON:KEY(13) ON:KEY(14) ON
220 PRINT"Are you running BASIC 2.0?"
230 A$=INKEY$:IF A$="" GOTO 230
240 CLS
250 IF A$="Y" OR A$="y" THEN GOTO 260 ELSE GOTO 350
260 KEY(15) ON:KEY(16) ON:KEY(17) ON:KEY(18) ON
270 KEY 15,CHR$(0)+CHR$(&H49)
280 KEY 16,CHR$(0)+CHR$(&H47)
290 KEY 17,CHR$(0)+CHR$(&H4F)
300 KEY 18,CHR$(0)+CHR$(&H51)
310 ON KEY(15) GOSUB 540
320 ON KEY(16) GOSUB 560
330 ON KEY(17) GOSUB 580
340 ON KEY(18) GOSUB 600
350 ON KEY(11) GOSUB 460
360 ON KEY(12) GOSUB 480
370 ON KEY(13) GOSUB 500
380 ON KEY(14) GOSUB 520
390 GOTO 630
```

```
400 ' *** SUBROUTINE ***

410 IF ROW>24 THEN ROW=24
420 IF ROW<1 THEN ROW=1
430 IF COL>W THEN COL=W
440 IF COL<1 THEN COL=1
450 GOTO 630
460 ROW=ROW-1
470 RETURN 410
480 COL=COL-1
490 RETURN 410
500 COL=COL+1
510 RETURN 410
520 ROW=ROW+1
530 RETURN 410
540 ROW=ROW-1:COL=COL+1
550 RETURN 410
560 ROW=ROW-1:COL=COL-1
570 RETURN 410
580 ROW=ROW+1:COL=COL-1
590 RETURN 410
600 ROW=ROW+1:COL=COL+1
610 RETURN 410

620 ' *** YOUR PROGRAM STARTS HERE ***

630 LOCATE ROW,COL:PRINT CHR$(CURSR);
640 A$=INKEY$:IF A$="" GOTO 630
650 CURSR=ASC(A$)
660 IF CURSR=0 THEN CURSR=8
670 GOTO 630
```

# READ JOYSTICKS

*WHAT IT DOES...*

**Returns value of IBM joysticks.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- AX: X coordinate Joystick A
- AY: Y coordinate Joystick A
- BY: Y coordinate Joystick B
- BX: X coordinate Joystick B
- F1: Status Fire Button A
- F2: Status Fire Button B.

### How to Use Subroutine

The true joysticks are somewhat trickier to use with the IBM PC and PCjr, because, unlike the methods presented so far, the joysticks return an actual screen coordinate position rather than simply the desired direction of movement. The ersatz joystick routines provided so far have simulated Atari-type joysticks, which consist of four switches that indicate in which direction the joystick is pressed. IBM joysticks, on the other hand, use a pair of rheostats which, depending on the resistance of each, can show the exact orientation, from center, of the sticks. Once properly calibrated (that is, the "center" position of the joystick is the "center" resistance value), these analog type joysticks can indicate movement more rapidly and precisely than the Atari type.

Assigning a value to a variable using the STICK(0) function causes the IBM computer to read the present x and y coordinates of the two joysticks. The value of the x coordinate for joystick A will be assigned to n. To assign the values of the joystick A y coordinate and both x and y of joystick B, it is necessary to carry out v=STICK(1), v=STICK(2), and v=STICK(3) statements.

Note that only the v=STICK(0) statement will cause the sticks to actually be read, even though v will contain just one value. In other words, STICK(0) retrieves all four values so that STICK(1)...etc. can be used. Each time you want a new value of any of the coordinates, it is necessary to use STICK(0).

STRIG is a slightly different statement and function. Unlike STICK, STRIG must be turned ON at the time it will be used with the STRIG ON statement. The function v=STRIG(n) can return several different values. If n=0, then v will equal −1 if joystick button A was pressed since the last time STRIG(0) was called. If n=1 then v will equal −1 only if the button is currently being pressed down.

The function n=2 is similar to N=0, but for joystick button B, while n=3 is equivalent to n=1 for the second joystick.

You may use this routine to position an object on the screen, depending on the coordinates of x and y, using the LOCATE statement. You'll have to convert the resistance values (usually from 0 to 255) to some number proportionate to the screen.

## Line-by-Line Description

**Line 180:** Clear screen.

**Line 200:** Turn off 25th line display.

**Line 210:** Turn ON STRIG feature.

**Line 240:** Retrieve all four joystick values and store value of joystick A's x coordinate in variable AX.

**Lines 250 to 270:** Assign remaining x and y coordinates to variables.

**Line 280:** If fire button A has been pressed, assign F1 with value of 1.

**Line 290:** If fire button B has been pressed, assign F1 with value of 1.

**Line 330:** Print value labels.

**Line 340:** Print values of joysticks and fire buttons.

## You Supply

Modules to provide screen movement.

## Sample Applications

- Simple BASIC games using true joysticks
- Menu selection for very young or for non-typists
- Computer design.

*RESULT...*

**Values of true joysticks returned.**

```
1Ø ' ********************
2Ø ' *                  *
3Ø ' *  READ JOYSTICKS *
4Ø ' *                  *
5Ø ' ********************
```

```
60 ' ---------------------------------
70 '    ++ VARIABLES ++
80 '
90 '      AX: X COORDINATE JOYSTICK A
100 '     AY: Y COORDINATE JOYSTICK A
110 '     BY: Y COORDINATE JOYSTICK B
120 '     BX: X COORDINATE JOYSTICK B
130 '     F1: STATUS FIRE BUTTON A
140 '     F2: STATUS FIRE BUTTON B
150 '
160 ' ---------------------------------


170 ' *** INITIALIZE ***


180 CLS
190 SCREEN 0,0,0
200 KEY OFF
210 STRIG ON
220 GOTO 320


230 ' *** SUBROUTINE ***


240 AX=STICK(0)
250 AY=STICK(1)
260 BX=STICK(2)
270 BY=STICK(3)
280 IF STRIG(0)=-1 OR STRIG(1)=-1 THEN F1=1 ELSE F1=0
290 IF STRIG(2)=-1 OR STRIG(3)=-1 THEN F2=1 ELSE F2=0
300 RETURN


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


320 GOSUB 240
330 PRINT "A-X","A-Y","B-X","B-Y","Fire A ";"Fire B"
340 PRINT AX,AY,BX,BY,F1;" ";F2
350 GOTO 320
```

# JOYSTICK BUTTON INTERRUPT

*WHAT IT DOES...*

**Interrupts program when joystick buttons are pressed.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

None.

### How to Use Subroutine

Write your own subroutines to perform the desired action whenever any of the four joystick buttons are pressed. You may temporarily disable the STRIG(n) function with STRIG(n) STOP. However, the computer remembers when the designated button is pressed in the interim and will activate the interrupt routine when STRIG(n) ON is next used during that program run.

### Line-by-Line Description

**Lines 140 to 170:** Tell program where to branch for each joystick button.

**Lines 180 to 190:** Activate trapping for all four buttons.

**Lines 220 to 290:** Subroutines activated by joystick buttons. User substitutes desired action here.

### You Supply

Desired program actions when button pressed.

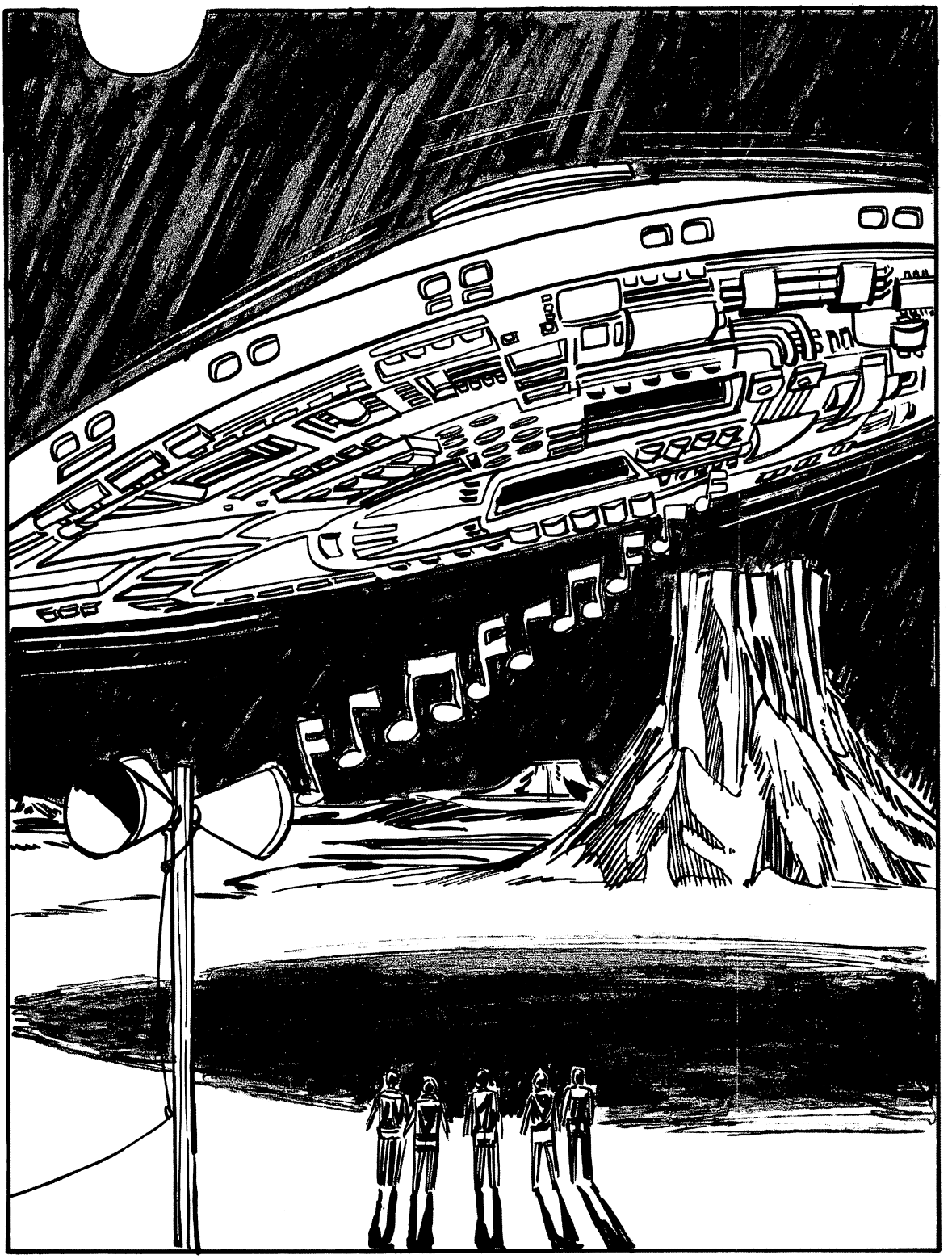
### Sample Applications

- Break out of program
- Clear screen in games
- Fire missile in games.

*RESULT...*

**Program will be interrupted when joystick buttons pressed.**

```
10 ' ********************
20 ' *                  *
30 ' *  JOYSTICK BUTTON *
40 ' *    INTERRUPT     *
50 ' *                  *
60 ' ********************
70 ' --------------------------
80 '    ++ VARIABLES ++
90 '
100 '          NONE
110 '
120 ' --------------------------

130 ' *** INITIALIZE ***

140 ON STRIG(0) GOSUB 220
150 ON STRIG(2) GOSUB 240
160 ON STRIG(4) GOSUB 260
170 ON STRIG(6) GOSUB 280
180 STRIG(0) ON:STRIG(2) ON
190 STRIG(4) ON:STRIG(6) ON
200 GOTO 310

210 ' *** YOUR SUBROUTINES GO HERE ***

220 PRINT "BUTTON A1 PRESSED."
230 RETURN
240 PRINT "BUTTON B1 PRESSED."
250 RETURN
260 PRINT "BUTTON A2 PRESSED."
270 RETURN
280 PRINT "BUTTON B2 PRESSED."
290 RETURN

300 ' *** YOUR PROGRAM STARTS HERE ***

310 GOTO 310
```



# 8

# Using Sound

The sound capabilities of the IBM PC and PCjr are much better than some home computers (because specific note frequencies can be generated precisely), but not so sophisticated as others. The three primary sound commands are BEEP, SOUND, and PLAY, which deliver notes from the built-in speaker. BEEP allows no arguments and produces nothing more than a simple tone. It is not used much in this book and not addressed in this chapter because of this limitation. With SOUND, you can specify both a note and duration:

```
SOUND 1ØØØ, 1Ø
```

The first figure is the number of cycles per second of the note, with the pitch increasing as the number grows larger. The second number is the length of time the note will be played, with 18 the approximate equivalent of one second.

The PCjr has three voices and allows two additional parameters for the SOUND command:

```
SOUND 1ØØØ,1Ø,1,1
```

The third number specifies the voice to be used, while fourth determines the volume of that voice. Both the PC and PCjr are very flexible in allowing any musical note that the built-in speaker can, or cannot, reproduce.

The note can range over seven octaves. . . and the time interval from a fraction of a second to several seconds. Each number in the duration argument represents about 1/18th second. The lowest possible note available with the SOUND function is 38 cycles per second. The IBM will generate notes as high as 32767 cps. This is far beyond the range of human hearing and certainly beyond the capabilities of the tiny speaker built into the computer. Any notes much above 10000 will generally be inaudible.

The third sound command is PLAY. This allows us to write musical routines using the names of the notes. Instead of having to know the frequency of middle C, we can play it by telling the IBM to PLAY "C." A C# is generated with the string "C#" and so on. Octave may be specified as well, along with the length of the note and other parameters.

Those with BASIC 2.0 can access the ON PLAY interrupt routine. That is, the IBM PC or PCjr can be running a program while playing music in the background. It will interrupt processing to fetch notes from the pool you supply as they are used up. This will take place so fast that, for most purposes, the music is continuous.

This chapter includes two routines that will turn your IBM PC or PCjr keyboard into a piano or organ to allow you to play notes by pressing appropriate keys. One demonstrates the use of SOUND to produce musical notes, while the second illustrates the PLAY statement. Ten more subroutines produce different sound effects, which you can drop into your game, personal, or business programs as appropriate. At the end of the chapter is a PCjr subroutine for producing white noise with the special NOISE function.

# MUSIC

*WHAT IT DOES...*

**Uses keyboard to generate various musical notes.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

Use where you want to have the IBM PC or PCjr keyboard simulate a piano. Pressing appropriate keys on the home row produces a note from the IBM PC or PCjr 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. 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 IBM PC or PCjr keyboard and begin to play songs using their sound capabilities.

In this subroutine, the SOUND command is used to play notes, with values provided in the BASIC reference manual used to determine the frequencies to be played. This particular method is a bit more flexible than the one that follows, because you are not limited to absolute note values such as A or A#. If you wish, you may change the numbers in the data statements to produce notes that are a microtone higher or lower than absolute notes. Thus, you can "tune" your IBM instrument in experimental ways.

### Line-by-Line Description

**Line 160:** Define length of note as 1/18th second. User may change this value as desired.

**Line 170:** Clear screen.

**Line 180:** Define arrays to store names of notes, the equivalent keys, and the SOUND values needed to reproduce those notes.

**Lines 190 to 210:** Read note names into array.

**Lines 220 to 240:** Read valid key names into array.

**Lines 250 to 270:** Read SOUND values into array.

**Line 340:** Wait for a key to be pressed.

**Line 350:** If key pressed was ENTER, then return.

**Lines 360 to 380:** Check to see if valid key was pressed.

**Line 400:** If so, then produce that note.

**Lines 410 to 420:** Erase old note name, print new on screen.

## You Supply

No user input required other than to play keyboard.

## Sample Applications

- Learning music
- Diversion in games
- Playing songs in games.

*RESULT...*

**Music played from IBM PC and PCjr speaker.**

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

```
150 ' *** INITIALIZE ***
160 LTH=1
170 CLS
180 DIM NME$(20),NT$(20),P(20)
190 FOR N=1 TO 20
200 READ NME$(N)
210 NEXT N
220 FOR N=1 TO 20
230 READ NT$(N)
240 NEXT N
250 FOR N=1 TO 20
260 READ P(N)
270 NEXT N
280 DATA C,C#,D,D#,E,F,G,G#,A,A#,B,C,C#,D,D#,E,F,F#,G
290 DATA A,W,S,E,D,F,T,G,Y,H,U,J,IK,O,L,P,;,',[,`
300 DATA 262,278,294,312,330,349,370,392,416,440,467,494,
    523,555,587,623,6659,698,741,784
320 GOTO 450
330 ' *** SUBROUTINE ***
340 A$=INKEY$:IF A$="" GOTO 340
345 IF ASC(A$)>96 AND ASC(A$)<121 THEN
    A$=CHR$(ASC(A$)-32)
350 IF A$=CHR$(13) THEN RETURN
360 FOR N=1 TO 16
370 IF A$=NT$(N) GOTO 400
380 NEXT N
390 GOTO 340
400 SOUND P(N),LTH
410 LOCATE 12,25:PRINT"     ";
420 LOCATE 12,25:PRINT NME$(N);
430 GOTO 340
440 ' *** YOUR PROGRAM STARTS HERE ***
450 GOSUB 340
```

# IBM ORGAN

*WHAT IT DOES...*

**Uses keyboard to generate various musical notes.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

This is a second method for producing music, demonstrating the PLAY command. With PLAY, notes may be specified using their names, such as A or A#. PLAY will generate concatenated strings of notes, which can also include symbols that change the octave, length of note, or whether it is played staccato or legato. These more complex capabilities are beyond the scope of this book. You may also want to investigate the capability of ON PLAY..., which enables you to have music in the background while your IBM computer goes on with other tasks.

This particular subroutine simply plays notes as you type them on the keyboard and serves as an introduction to the PLAY command. You can raise and lower the octave in which the IBM Organ is playing by pressing the F1 and F2 function keys.

Use where you want to have the IBM PC and PCjr keyboard simulate a piano. Pressing appropriate keys on the home row produces a note from the IBM PC and PCjr speaker. In addition, the name of the note (and a sharp symbol if the note is sharp) is printed to the screen.

The user can "learn" the IBM PC and PCjr keyboard and begin to play songs using their sound capabilities.

### Line-by-Line Description

**Line 160:** Set length of note to 1/18th second.

**Line 170:** Turn off 25th line display.

**Lines 180 to 190:** Turn on trapping for F1 and F2.

**Lines 200 to 210:** Tell BASIC where to branch on F1 and F2.

**Line 220:** Clear screen.

**Line 230:** Dimension array to store names of notes and corresponding keys.

**Lines 240 to 260:** Read note names into array.

**Lines 270 to 290:** Read keys into array.

**Line 340:** Wait for key to be pressed.

**Line 350:** If key was ENTER then RETURN.

**Lines 360 to 380:** Check to see if valid key was pressed.

**Line 400:** Produce P$, which corresponds to "O" (for octave, OC, which is the current octave), and NME$(N), which is the note to be played.

**Line 410:** Play the note.

**Lines 420 to 430:** Erase old note name and print new one to screen.

**Lines 450 to 460:** If F1 pressed, raise octave by one, up to a limit of six.

**Lines 470 to 480:** If F2 pressed, then reduce octave by one until 0 is reached.

## You Supply

No user input required, other than to play keyboard.

## Sample Applications

- Background music in games
- Music signal as to program status
- Learning musical scale.

*RESULT...*

**Music played from IBM PC and PCjr speaker.**

```
10 ' ****************
20 ' *                *
30 ' *   IBM ORGAN    *
40 ' *                *
50 ' ****************
```

```
6Ø ' ------------------------------------
7Ø '    ++ VARIABLES ++
8Ø '    P$:       STRING TO PLAY
9Ø '    NME$(n): NAME OF NOTE
1ØØ '   NT$(n):  KEY CORRESPONDING TO NOTE
11Ø '   OC:       CURRENT OCTAVE
12Ø '   LTH:      LENGTH OF NOTE
13Ø '
14Ø ' ------------------------------------

15Ø ' *** INITIALIZE ***

16Ø LTH=1
17Ø KEY OFF
18Ø KEY(1) ON
19Ø KEY(2) ON
2ØØ ON KEY(1) GOSUB 45Ø
21Ø ON KEY(2) GOSUB 47Ø
22Ø CLS
23Ø DIM NME$(20),NT$(20),P(20)
24Ø FOR N=1 TO 20
25Ø READ NME$(N)
26Ø NEXT N
27Ø FOR N=1 TO 20
28Ø READ NT$(N)
29Ø NEXT N
3ØØ DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B,C,C#,D,D#,E,F,F#,G
31Ø DATA A,W,S,E,D,F,T,G,Y,H,U,J,K,O,L,P,;,',[,`
32Ø GOTO 5ØØ
```

```
330 ' *** SUBROUTINE ***

340 A$=INKEY$:IF A$="" GOTO 340
350 IF A$=CHR$(13) THEN RETURN
355 IF ASC(A$)>96 AND ASC(A$)<121 THEN
    A$=CHR$(ASC(A$-32))
360 FOR N=1 TO 20
370 IF A$=NT$(N) GOTO 400
380 NEXT N
390 GOTO 340
400 IF LAST<13 AND N>12 THEN OC=OC+1 ELSE IF LAST>12
    AND N<13 THEN OC=OC-1
405 IF OC>6 THEN OC=6 ELSE IF OC<0 THEN OC=0
410 LAST=N:P$="O"+STR$(OC)+NME$(N):PLAY P$
420 LOCATE 12,25:PRINT"      ";
430 LOCATE 12,25:PRINT NME$(N);
440 GOTO 340
450 OC=OC+1
460 RETURN 340
480 RETURN 340

490 ' *** YOUR PROGRAM STARTS HERE ***

500 GOSUB 340
```

## SIREN

*WHAT IT DOES...*

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

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- N: Number of repeats of sound effect.

## How to Use Subroutine

Call the subroutine when a 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. Seven octaves are encompassed between the high and low ends, although the IBM computer will produce notes far beyond the audible hearing range.

## Line-by-Line Description

**Line 160:** Begin loop of 100 repetitions.

**Lines 170 to 190:** Loop through notes from 100 to 200, in increments of 50. Produces rising pitch.

**Lines 200 to 230:** Falling pitch.

**Line 250:** Access the subroutine.

## You Supply

No user changes required.

## Sample Applications

- Simulate police car in game
- Alert user of problem
- General game sound effects.

> *RESULT...*
>
> **Siren sound emitted for game play or other applications.**

```
10 ' *********
20 ' *       *
30 ' * SIREN *
40 ' *       *
50 ' *********
60 '
70 ' ---------------------------
80 '    ++ VARIABLES ++
90 '
100 '     N: NUMBER OF REPEATS
110 '
120 '
130 ' ---------------------------
140 GOTO 270
```

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

160 FOR N=1 TO 100
170 FOR N1=100 TO 200 STEP 50
180 SOUND N1,1
190 NEXT N1
200 FOR N2=1 TO 20:NEXT N2
210 FOR N1=1000 TO 900 STEP -50
220 SOUND N1,1
230 NEXT N1
240 NEXT N
250 RETURN

260 ' *** YOUR PROGRAM STARTS HERE ***

270 GOSUB 160
```

## BOMB

*WHAT IT DOES...*

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

### Versions: IBM PC and PCjr

### Variables
None.

### How to Use Subroutine
Call subroutine when bomb sound is desired.

### Line-by-Line Description

**Lines 160 to 180:** Produce falling sound.

**Lines 190 to 270:** Explosion.

**Line 300:** Access the subroutine.

## You Supply

No user changes required.

## Sample Applications

- Signal when program bombs
- Produce explosion in game programs
- Signal wrong answer in application.

*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 300

150 ' *** SUBROUTINE ***

160 FOR N=300 TO 10 STEP -3
170 SOUND N*10,N/100
180 NEXT N
190 FOR N=100 TO 38 STEP -10
200 CU=38
210 SOUND N,1
220 SOUND CU,1
230 CU=CU+5
240 NEXT N
250 FOR N=38 TO 4000 STEP 500
```

```
26Ø SOUND N,.5
27Ø NEXT

29Ø ' *** YOUR PROGRAM STARTS HERE ***

3ØØ GOSUB 16Ø
```

# ALARM

*WHAT IT DOES...*

**Alarm sound routine to produce sounds for games, other applications.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- N1: Number of repeats.

### How to Use 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.

### Line-by-Line Description

**Line 160:** Repeat sound N1 (10) times.

**Lines 170 to 230:** Produce rising sound.

**Line 260:** Access the subroutine.

### You Supply

No user changes required.

### Sample Applications

- Inform operator of time up
- Signal wrong response in game program
- Indicate problem with program in ON ERROR routine.

*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 260

150 ' *** SUBROUTINE ***

160 FOR N1=1 TO 10
170 FOR N=600 TO 900 STEP 20
180 SOUND N,1
190 NEXT N
200 FOR N=900 TO 600 STEP -20
210 SOUND N,1
220 NEXT N
230 NEXT N1
240 RETURN

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

260 GOSUB 160
```

# KLAXON

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- N1: Number of repeats

### How to Use Subroutine

Call subroutine when klaxon sound is desired.

### Line-by-Line Description

**Line 160:** Repeat sound N1 (10) times.

**Lines 170 to 220:** Produce rising sound.

**Line 260:** Access the subroutine.

### You Supply

No user changes required.

### Sample Applications

- Bridge of ship in game program
- Signal of different problem with program than ALARM
- Attention-getting device in displays, timer programs.

*RESULT...*

**Klaxon sound emitted for game play or other applications.**

```
10 ' **********
20 ' *        *
30 ' * KLAXON *
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=1000 TO 5000 STEP 1000
180 SOUND N,1
190 NEXT N
200 FOR N=5000 TO 1000 STEP -1000
210 SOUND N,1
220 NEXT N
230 NEXT N1
240 RETURN

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

260 GOSUB 160
```

# UFO

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

None.

### How to Use 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.

### Line-by-Line Description

**Line 160:** Define first note, CU, as 1000.

**Line 170:** Loop between 100 and 1000.

**Line 180:** Play note equal to N.

**Line 190:** Play note equal to CU.

**Line 200:** Make S the difference between CU and N.

**Line 210:** Make sure S is never less than 38.

**Line 220:** Play note equal to S.

**Line 230:** Reduce CU by one.

**Line 270:** Access the subroutine.

### You Supply

No user changes required.

### Sample Applications

- Flying saucer or alien sound in game

- Signal "weird" response in application program
- Alert operator of impending disaster.

*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 270

150 ' *** SUBROUTINE ***

160 CU=1000
170 FOR N=100 TO 1000
180 SOUND N,.05
190 SOUND CU,.05
200 S=N-CU:IF S<38 THEN S=CU-N
210 IF S<38 THEN S=38
220 SOUND S,.05
230 CU=CU-1
240 NEXT N
250 RETURN

260 ' *** YOUR PROGRAM STARTS HERE ***

270 GOSUB 160
```

# COMPUTER

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables
None.

### How to Use Subroutine
Call the subroutine when a computer-like sound is desired.

### 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.

### Sample Applications
- Signal of computer "thinking" during number crunching
- Computer sound for games
- Attention-getting device to impress friends.

*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 ***

150 FOR N=1 TO 100
160 R=RND(1)*3000+38
170 L=RND(1)*5
180 SOUND R,L
190 NEXT N
200 RETURN

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

220 GOSUB 150
```

## LASER

*WHAT IT DOES...*

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

### Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- N1: Number of repeats.

## How to Use Subroutine

Call the subroutine when the 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.

## Line-by-Line Description

**Lines 160 to 170:** Initial laser sound.

**Lines 180 to 210:** End laser sound.

**Line 240:** Access subroutine.

## You Supply

No user changes required.

## Sample Applications

- Laser gun sound for games
- Sound effect in applications program when deleting or removing something
- Different alert sound for operator.

*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 240
```

```
15Ø ' *** SUBROUTINE ***

16Ø SOUND 1ØØØ,.Ø1
17Ø SOUND 4Ø,.Ø1
18Ø FOR N=6ØØØ TO 1ØØØ STEP -1ØØ
19Ø SOUND N,.1
2ØØ NEXT N
21Ø SOUND 1ØØØ,.Ø5
22Ø RETURN

23Ø ' *** YOUR PROGRAM STARTS HERE ***

24Ø GOSUB 16Ø
```

# ROULETTE WHEEL

*WHAT IT DOES...*

**Roulette wheel sound, which slows down gradually, for games, other applications.**

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

None.

### How to Use Subroutine

Call the subroutine when the 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 to 20.

**Line 170:** Set initial multiplier to 1.001

**Line 180:** Set factor to increase multiplier to .0005

**Line 190:** Start 120 repetitions.

**Line 200:** Increase delay by multiplying times F.

**Line 210:** Count off the delay.

**Line 220:** Make click sound.

**Line 230:** Enlarge multiplier.

**Line 250:** Play final click.

## You Supply

No user changes required.

## Sample Applications

- Game sound for wheel of fortune
- Slowing car engine
- Signal impending end of an operation.

*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 280

150 ' *** SUBROUTINE ***

160 DELAY=20
170 F=1.001
```

```
18Ø F1=.ØØØ5
19Ø FOR N=1 TO 12Ø
2ØØ DELAY=DELAY*F
21Ø FOR D=1 TO DELAY:NEXT D
22Ø SOUND 3ØØ,.Ø5
23Ø F=F+F1
24Ø NEXT N
25Ø SOUND 3ØØ,1
26Ø RETURN

27Ø ' *** YOUR PROGRAM STARTS HERE ***

28Ø GOSUB 16Ø
```

# HEARTBEAT

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

None.

### How to Use Subroutine

Call the subroutine when the 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.

### 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.

## Sample Applications

- Scary heartbeat sound for games
- Evidence that computer is "thinking"
- Background noise.

> *RESULT...*
>
> **Heartbeat sound emitted for game play or other applications.**

```
10 ' *********
20 ' *       *
30 ' * HEART *
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 40,.5
180 FOR N=1 TO 200:NEXT N
190 SOUND 40,3
200 FOR N=1 TO 600:NEXT N
```

```
21Ø NEXT N1
22Ø RETURN

23Ø ' *** YOUR PROGRAM STARTS HERE ***

24Ø GOSUB 16Ø
```

# CLOCK

*WHAT IT DOES...*

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

## Versions: IBM PC and PCjr, Advanced BASIC

### Variables

- N2: Number of ticks.

### How to Use Subroutine

Call the subroutine when the sound of a ticking clock 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.

### 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.

## Sample Applications

- Count off "waiting" time in game while player thinks
- Produce background noise during timer subroutine
- Use to show that program is awaiting input.

*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 4000,.05
180 FOR N=1 TO 300:NEXT N
190 SOUND 6000,.05
200 FOR N=1 TO 300:NEXT N
210 NEXT N1
220 RETURN

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

240 GOSUB 160
```

# NOISE

*WHAT IT DOES...*

**Generates white noise for use in games, other applications.**

## Versions: IBM PCjr, Advanced BASIC

### Variables

- N2: Number of repetitions
- T3: Voice number 3
- V3: Volume of voice 3
- SOUND3: Sound register
- LGTH: Length of sound.

### How to Use Subroutine

So called nonmusical "white noise" is useful in game programs for explosions, gunshots, and other nonspecific sounds. The PCjr's BASIC has a NOISE command that can create these sounds.

You can vary the pitch, length, frequency and volume of the noise, by varying the values for SOUND, LGTH, T3, and V3, respectively.

### Line-by-Line Description

**Lines 140 to 180:** Define variables for noise.

**Line 210:** Start loop from 1 to N2.

**Line 220:** Make noise.

**Line 230:** Change sound of noise.

**Line 240:** Repeat.

### You Supply

No user changes required.

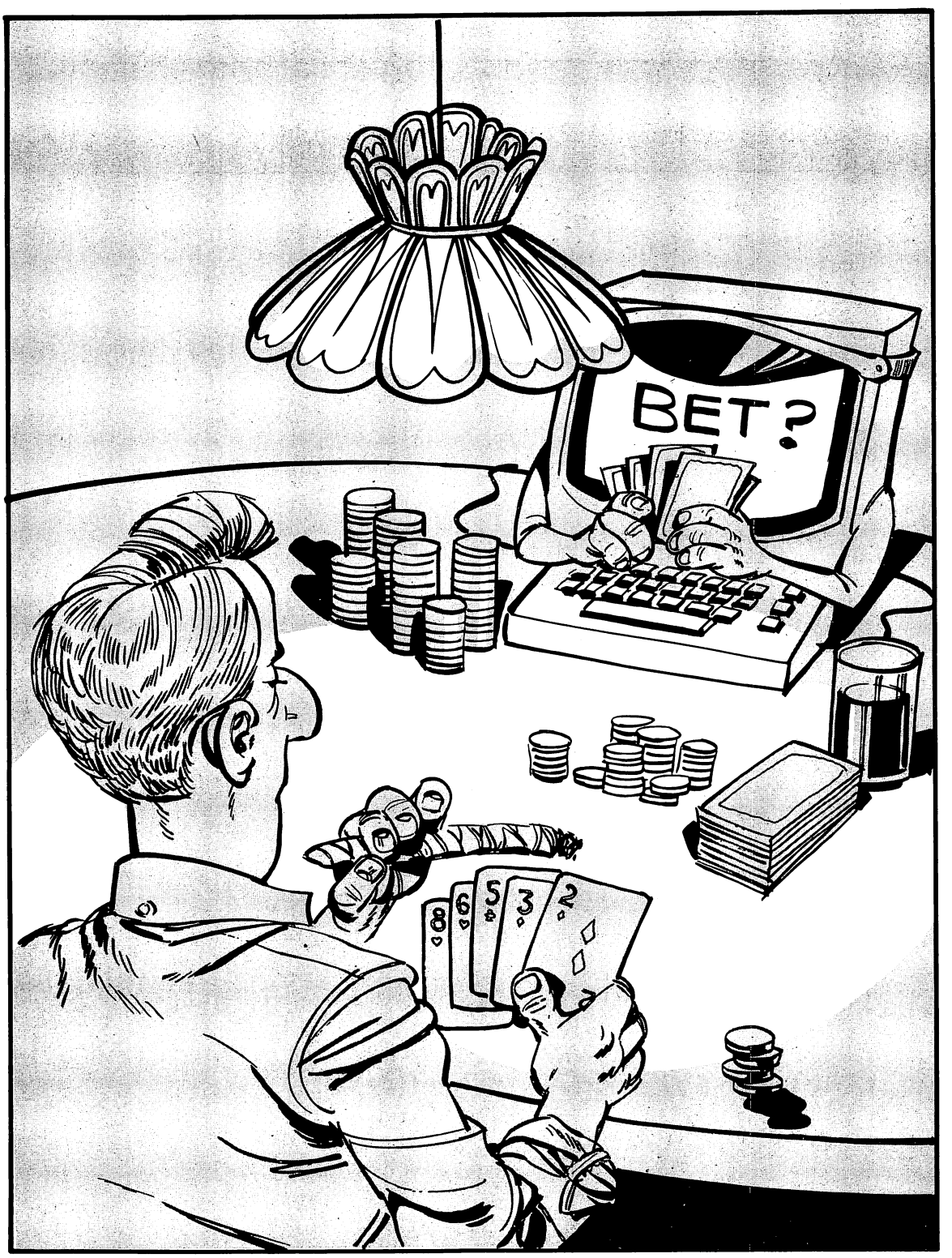
### Sample Applications

- Gunfire
- Space ship engine
- Storm, other noise.

*RESULT...*

**Noise sound emitted for game play or other applications.**

```
10 ' *********
20 ' *       *
30 ' * NOISE *
40 ' *       *
50 ' *********
60 ' ---------------------------
70 '    ++ VARIABLES ++
80 '    N2: REPETITIONS
90 '    T3: VOICE 3
100 '   V3: VOLUME OF VOICE 3
110 '   SOUND3: NOISE NOTE
120 '   LGTH: LENGTH OF NOISE
130 ' ---------------------------
140 N2=10
150 SOUND3=440
160 LGTH=18
170 T3=3
180 V3=3
190 GOTO 270

200 ' *** SUBROUTINE ***

210 FOR N1=1 TO N2
220 NOISE 1
230 SOUND SOUND3,LGTH,T3,V3
240 NEXT N1
250 RETURN

260 ' *** YOUR PROGRAM STARTS HERE ***

270 GOSUB 210
```


BET?

# 9

# Game Routines

Games are probably one of the most popular programming exercises for beginners and advanced users alike. On your first day with a computer you can easily learn to write a "Craps"-playing program. Those IBM PC users who are not all business and many of the owners of the PCjr will probably someday investigate writing an arcade-quality, joystick-activated shoot-em-up.

In either case, you can use the routines in this book to avoid reinventing the wheel. Actually, the subroutines useful for games programming are not confined to this section. Many of the modules presented so far can be transplanted to games programs. These include all of the routines in Chapters 1, 2, and 3.

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

Randomness is an essential factor in many types of games, and one subroutine in this section allows you to specify the drawing of random numbers in any range you choose. If your game happens to need random numbers larger than 100 and smaller than 999, the routine can handle that nicely. Note that the IBM PC and PCjr do not return true random numbers, but rather a "pseudorandom" series. To keep a game from using the same "random" numbers each time, you need to reseed the number generator; that is, tell it to start at a new point in its list of pseudorandom numbers. This is most easily done with the IBM computers by the RANDOMIZE TIMER function, which uses the current value of TIMER as the seed.

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

# DEAL CARDS

*WHAT IT DOES...*

**Shuffles and deals deck of cards.**

## Versions: IBM PC AND PCjr,Advanced BASIC

### Variables

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

## How to Use Subroutine

Many game programs require dealing a deck of cards. Your own programs may simulate drawing from a randomly shuffled deck simply by calling the subroutine beginning at line 370. The deck has already been assembled (lines 210-360) using the IBM PC and PCjr 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$(CARD$)="Q" THEN V=12."

This is a very fast shuffling routine, which requires only 52 tries to deal 52 cards. Some slower algorithms (a formula for performing a task, or computing a result) may repeatedly access "empty" deck positions when looking for the remaining cards.

The routine starts off by setting NC (number of cards) to 52. In line 410, the computer selects a number between 1 and NC (52 this time), and that element of DECK$(n) becomes the card drawn. This leaves a "hole" in the deck at position DRAW. We fill it up by taking the last card in the deck, which is DECK$(NC), and place it in DECK$(DRAW). This leaves the "hole" at the end, but we then change NC to equal NC-1 so the computer will only draw from the elements 1 through 51 on the next time through. Third time, it will choose 1 through 50, and so forth. It does not matter that we have mixed up the order of the deck, as we want the cards shuffled in the first place.

Each element of DECK$(n) consists of a number, or face card name, plus the IBM PC and PCjr CHR$ value for the suit (either 3,4,5 or 6). This produces a full deck of 52 cards.

## Line-by-Line Description

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

**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.

## Sample Applications

- Card games
- Any game in which sets must be "shuffled," such as cryptograms. Use 26 "cards," the alphabet, instead of a deck of cards
- Producing any random number or string from a fixed set of choices.

*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):RANDOMIZE TIMER
```

```
150 ' *** READ SUITS ***

160 FOR N=3 TO 6
180 SUIT$(N)=CHR$(N)
190 NEXT N

200 ' *** ASSEMBLE DECK ***

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

460 ' *** YOUR PROGRAM STARTS HERE ***

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

# RANDOM RANGE

*WHAT IT DOES...*

**Allows choosing random numbers in any range.**

## Versions: IBM PC AND PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

What makes a game a game and not a test? Randomness is one element found in many, but not all, games. Random numbers selected by the computer determine the changes in some games that the player must contend with. Lacking randomness, a game is either a test of memory or a contest of strategy. A little of all three elements makes for a good game, and this subroutine lets you get greater control over randomness than unadorned IBM PC and PCjr BASIC.

The IBM PC and PCjr can choose pseudorandom numbers. That is, although they appear to be random, the numbers actually are drawn from a long list. Even though the sequence is the same each time, the list of numbers is very long, and the starting position is usually different, so the numbers appear to be random to the player.

Some BASICs allow choosing a random number larger than one but smaller than another integer with the simple command RND(N), where N is the upper limit. RND(7) would produce integers from one to seven, for example. The IBM PC and PCjr will generate random numbers larger than zero and smaller than one. So, we might get .74329 or .15832 or some other value. To get numbers in a given range 1 to N, we must multiply the random number by N and add one. That is, INT(RND(1)*7)+1 will produce numbers in the range one to seven.

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

## Line-by-Line Description

**Lines 150 to 170:** Define the highest random number desired and the lowest, and find the difference between them.

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

**Line 230:** Print the result.

**Line 240:** Access the subroutine.

## You Supply

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

## Sample Applications

- Games in which numbers must be in a certain range—like "prices" between $10 and $100
- Other random results in games.

> *RESULT...*
>
> **Only random numbers in the specified range will be produced.**

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

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

```
150 HIGH=100
160 MINIMUM=15
170 DF=HIGH-MINIMUM+1
180 RANDOMIZE TIMER:GOTO 230

190 ' *** SUBROUTINE ***

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

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

230 GOSUB 190
240 PRINT NU;
```

# COIN FLIP

*WHAT IT DOES...*

**Flips coin, producing heads or tails.**

## Versions: IBM PC AND PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

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

## Line-by-Line Description

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

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

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

**Line 220:** Access the subroutine.

**Line 230:** Print result.

## You Supply

No user input needed.

## Sample Applications

- Games in which two choices are involved
- Programs where two variations on a single result are wanted, as for ON A..GO-SUB... with A chosen by coin flip.

*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 ' -------------------------

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

```
14Ø COIN$(1)="HEADS"
15Ø COIN$(2)="TAILS"
16Ø RANDOMIZE TIMER:GOTO 22Ø

17Ø ' *** SUBROUTINE ***

18Ø FLIP=INT(RND(1)*2)+1
19Ø FLIP$=COIN$(FLIP)
2ØØ RETURN

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

22Ø GOSUB 18Ø
23Ø PRINT FLIP$
24Ø GOTO 22Ø
```

# DICE

*WHAT IT DOES...*

**Simulates roll of dice.**

## Versions: IBM PC AND PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

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

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

## Line-by-Line Description

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

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

**Line 180:** Return to main program.

**Line 210:** Define number of sides on dice.

**Line 220:** Access the subroutine.

**Lines 230 to 260:** Print results of roll.

## You Supply

Number of sides of die.

## Sample Applications

- Dungeons and Dragons games
- Any games using dice
- Programs where dice odds are wanted in selection of choices at random.

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

```
10 ' ********
20 ' *      *
30 ' * DICE *
40 ' *      *
50 ' ********
60 RANDOMIZE TIMER:GOTO 200
70 ' --------------------------
80 '    ++ VARIABLES ++
90 '     D1:   DIE #1 TOTAL
100 '    D2:   DIE #2 TOTAL
110 '    ROLL: TOTAL OF ROLL
120 '
130 ' --------------------------

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

```
150 D1=INT(RND(1)*SIDES)+1
160 D2=INT(RND(1)*SIDES)+1
170 ROLL=D1+D2
180 RETURN

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

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

# DELAY LOOP

*WHAT IT DOES...*

**Delays loop changes in length.**

## Versions: IBM PC AND PCjr, Advanced BASIC

### Variables

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

### How to Use Subroutine

In games, delay loops are frequently used to display messages on the screen for a given length of time. 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 set some upper limit, so that the action doesn't appear to 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 fourth 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.

## Sample Applications

- Speed up—or slow down—screen movement of an object
- Change speed of a sound
- Make game harder as player continues.

*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
```

# GLOSSARY

**Algorithm:** A formula or method for performing a given task, such as MPG=MILES/GALLNS.

**Alphanumeric:** A character that is a letter or number, as opposed to a graphics or control character. 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 only if both are equal to 1.

**Append:** To add to the end of, 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.

**Baud:** A measurement of serial communications speed. Baud roughly equals bits per second, up to about 1200 baud.

**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 IBM PC and PCjr; 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, that uses two-valued variables (on/off,true/false) suitable for use by binary computers like the IBM PC and PCjr. 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 IBM PC and PCjr ask for decimal numbers and return decimal numbers for PEEK and POKE operations, even though it processes them in binary form internally. They will, of course, also PEEK and POKE with other number bases, such as hexadecimal (base-16).

**Decrement:** To decrease a variable by one. However, the word is also commonly used as a verb when the number is being decreased by some larger amount, as to "decrement by two."

**Default:** Any value used automatically if no other value is supplied by the program or user.

**Download:** To capture a file through telecommunications into the IBM PC and PCjr's memory buffer, and then write it to tape or disk for permanent storage. Programs or text files can be transmitted to your computer through a modem and then downloaded.

**File:** Any collection of information on disk or tape. BASIC and machine-language programs, as well as text material, are all files.

**Function key:** One of the 10 keys at the side of the IBM PC and at the top of the PCjr keyboard, which can be used as to direct control to subroutines or functions of the programmer's choice.

**Increment:** To increase the value of a variable by one. This is also commonly used as a verb to denote increasing a variable by any amount, such as "to increment by four."

**Initialize:** To set variables to a desired beginning value at the start of a program or at the beginning of a subroutine. For example:

```
10 B=0
20 INPUT A
30 B=B+A
40 PRINT B
50 GOTO 20
```

You would want to initialize B, as in line 10, each time the subtotal should be eliminated and the addition started from zero again.

**Garbage:** Random information with no meaning. Every memory location contains something. If it is not meaningful information placed there by the computer or user, it is termed garbage.

**Merge:** To combine two programs in such a way that their line numbers become mixed. While MERGING may produce interleaved programs, if there are duplicate line numbers, the program added will write over the same lines in the original program.

**Modem:** Modulator-demodulator. A device that converts the IBM PC and PCjr's signals to sounds that can be transmitted over telephone lines. The modem also receives sounds and converts them back for the IBM PC and PCjr 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. We use DEF SEG to tell the IBM PC and PCjr which memory block we want to address during PEEKs and POKEs.

**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 byte 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 byte one bit at a time.

**Port:** One of the "windows" used by the IBM PC and PCjr 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.

```
1Ø INPUT "ENTER YOUR NAME";A$
```

**Pseudorandom:** 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 appears random.

**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 IBM PC and PCjr 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 IBM PC and PCjr's microprocessor and can be accessed only through machine language. Some memory locations in the IBM PC and PCjr perform a register-like function. The status of these registers tells the computer whether a certain feature is ON or OFF, or the volume of a sound oscillator, or some other status.

**Rheostat:** A variable resistor, like those used in paddles, which lets more electricity flow when turned one way and less when turned the other.

**RS-232:** The asynchronous adapter port, a serial interface device that allows the IBM PC and PCjr to communicate with devices like printers or modems one bit at a time.

**Sequential:** A serial file access method in which each piece of information is stored after another and must be written or accessed in that fashion.

**Serial:** Sequential data storage or transfer.

**String delimiter:** A character that the computer recognizes as the "end" of a given string input. The most common are commas and quotation marks.

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

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

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

**Upload:** To store a file from disk or tape in the IBM PC and PCjr'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

## O



## P



## R



## S



## T



## U



## V



## X

# The Brady IBM PC Library

## Inside the IBM PC: Access to Advanced Features and Programming

**Peter Norton**

The most widely read author on the IBM PC explains the workings of the computer.

*1983/320 pp/ISBN 0-89303-556-4*
*Book/diskette package ISBN 0-89303-561-0*

## IBM PC: An Introduction to the Operating System, BASIC Programming and Applications—Revised and Enlarged

**Larry Joel Goldstein and Martin Goldstein**

An updated and expanded version of what has become the classic self-study book for the IBM PC.

*1983/392 pp/ISBN 0-89303-530-0*
*Book/diskette package ISBN 0-89303-527-0*

## IBM PC & XT Assembly Language: A Guide for Programmers

**Leo J. Scanlon**

An introduction to the principles of microprocessors (specifically the 8088), numbering systems, and assemblers.

*1983/320 pp/ISBN 0-89303-241-7*
*Book/diskette package ISBN 0-89303-535-1*

## MS-DOS and PC-DOS: User's Guide

**Peter Norton**

The authority, Peter Norton, gives an introduction and explanation of the Microsoft Disk Operating System. Explains DOS 1.0 and 2.0 versions.

*1984/266 pp/ISBN 0-89303-645-5*

## 8087 Applications and Programming for the IBM PC and Other PCs

**Richard Startz**

Starts with a non-technical introduction and turns into a very detailed description of the myriad applications for the 8087 microprocessor.

*1983/276 pp/ISBN 0-89303-420-7*
*Diskette only ISBN 0-89303-421-5*
*Book/diskette package ISBN 0-89303-425-8*

## Communications and Networking for the IBM PC

**Larry E. Jordan and Bruce W. Churchill**

Brings together data communications applications and the IBM PC. Includes asynchronous and synchronous communications and a complete study of local area networking.

*1983/237 pp/ISBN 0-89303-385-5*

## Handbook of BASIC for the IBM PC

**David I. Schneider**

Clearly translates the BASIC reference manual supplied with the IBM PC into understandable terms. Organized by BASIC programming statements.

*1983/498 pp/ISBN 0-89303-506-8*
*Book/diskette package ISBN 0-89303-508-4*

## IBM PC and XT Owner's Manual: A Practical Guide to Operations

**Barbara Lee Chertok, Dov Rosenfeld, and James H. Stone**

Provides easy instructions on the operation of the IBM PC. Helps to unravel the jargon found in the supplied manual to allow for easy access to operation.

*1984/200 pp/ISBN 0-89303-531-9*

## Advanced BASIC and Beyond for the IBM PC

**Larry Joel Goldstein**

A complete guide to the advanced skills of BASIC programming, files, graphics, event-trapping, machine language, and subroutines. A must for the IBM PC programmer.

*1984/360 pp/ISBN 0-89303-324-3*
*Book/diskette package ISBN 0-89303-325-1*

## Business Problem Solving with the IBM PC & XT

**Leon A. Wortman**

Business professionals will use the dozens of programs specifically designed for decision-making and problem-solving. Source codes are included in BASIC and many are in Pascal.

*1983/324 pp/ISBN 0-89303-282-1*
*Book/diskette package ISBN 0-89303-342-1*

## Business Applications for the IBM Personal Computer

**Steven Zimmerman and Leo Conrad**

Offers step-by-step intructions on the use and customization of existing business software programs.

*1983/300 pp/ISBN 0-89303-243-3*
*Book/diskette package ISBN 0-89303-351-0*

## Games, Graphics, and Sound for the IBM PC

**Dorothy Strickland, Dennis Rockwell, and Kevin Bowyer**

Teaches how to proram in BASIC, Pascal, and FORTRAN to create graphics and sound for the IBM PC. Illustrates how to integrate sound and graphics into animation.

*1983/256 pp/ISBN 0-89303-469-3*
*Book/diskette package ISBN 0-89303-470-3*

## BASIC Engineering and Scientific Programs for the IBM PC

**Philip M. Wolfe and C. Patrick Koelling**

A source of BASIC programs for on-the-job use by engineers and scientists to provide important computer techniques for problem-solving and data manipulation.

*1983/358 pp/ISBN 0-89303-330-8*
*Book/diskette package ISBN 0-89303-331-6*

## Pascal for the IBM PC: IBM DOS Pascal and UCSD p-System Pascal

**Kevin W. Bowyer and Sherryl J. Tomboulian**

The first word written on combining the IBM PC with Pascal programming. An emphasis on sound and graphics applications is provided.

*1983/416 pp/ISBN 0-89303-280-8*
*Book/diskette package ISBN 0-89303-761-3*

## WordStar for the IBM PC: A Self-Guided Tutorial

**Micro Workshop of Cambridge**

Now WordStar can be mastered with this step-by-step journey into it's many features.

*1984/280 pp/ISBN 0-89303-956-X*

## The C Programmer's Handbook

**Thom Hogan**

The ultimate encyclopedia of information necessary to use C. Choose other books to learn C, choose this book to USE C.

*1984/288pp/ISBN 0-89303-365-0*

## The C Programming Tutor

**Leon A. Wortman and Thomas O. Sidebottom**

Here is a tutorial that goes beyond the standard guides to C language. Contains a variety of C programming examples in a self-study format.

*1984/274pp/ISBN 0-89303-364-2*

## Programming the IBM PC & XT: A Guide to Languages

**Clarence B. Germain**

A gold mine of information on programming the PC. An in-depth reference guide for programmers seeking skills necessary to operate and run programs on the IBM PC. Covers PC-DOS 1.1 through 2.0.

*1984/338 pp/ISBN 0-89303-783-4*

**Available at your local bookstore or computer retailer. Or write to Brady Communications Company, Inc., Bowie, MD 20715. Phone (301) 262-6300.**

*Whet Your Programming Appetite with a "Cookbook" Approach . . .*
*Writing Quality Programs on the IBM PC and PCjr!*

*". . .the lightly written text and the line-by-line description coupled with the subroutines themselves will definitely be well received by novices everywhere!"*

*"This book fills a very definite need in the PC and PCjr book marketplace!"*

## IBM PC AND PCjr 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 IBM PC and PCjr! 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, joysticks, sound, and other special features of the IBM PC and PCjr (what's more, the book specifies which subroutines apply to which machine). No more "stewing" over exotic, top-heavy math functions and statistics! Complete with line-by-line descriptions of each subroutine presented, this book also includes:

- Subroutines for generating musical notes or adding sound effects within your own programming
- Subroutines for business/financial users and advanced programmers as well
- Tips on routines to make your games of arcade quality
- Plus—a comprehensive glossary and index!

### CONTENTS




ISBN 0-89303-542-4