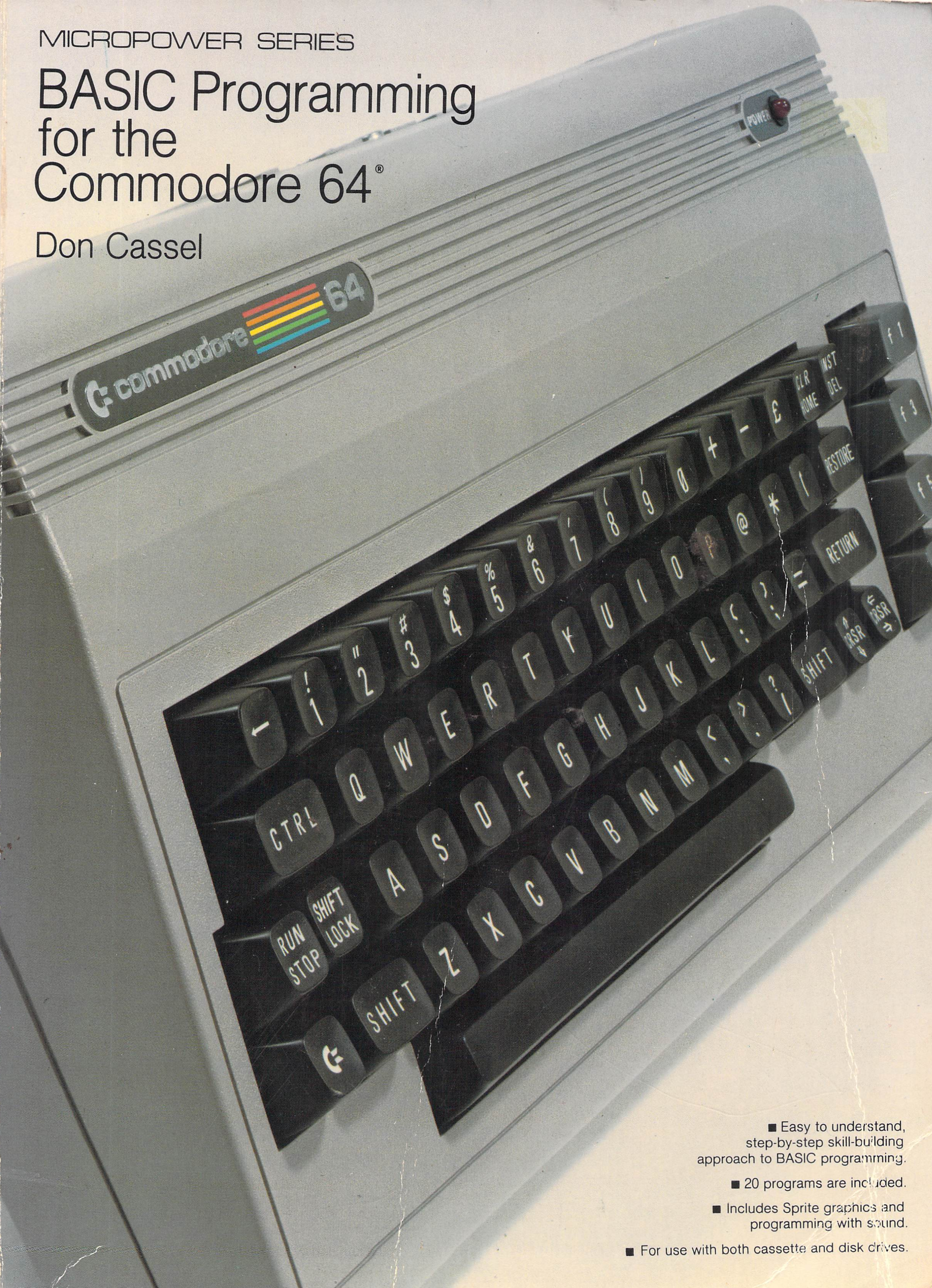

MICROPOWER SERIES
BASIC Programming for the Commodore 64®
Don Cassel
commodore 64
POWER
■ Easy to understand, step-by-step skill-building approach to BASIC programming.
■ 20 programs are included.
■ Includes Sprite graphics and programming with sound.
■ For use with both cassette and disk drives.

BASIC Programming for the Commodore 64®

MICROPOWER SERIES

BASIC Programming for the Commodore 64®

Don Cassel
Humber College

wcb
Wm. C. Brown Publishers
Dubuque, Iowa

Cover photo is provided courtesy of Bob Coyle

Figures 1.1, 2.1, 2.2, 2.3, 2.4, and Appendix F are reproduced courtesy of Commodore Business Machines.

Edouard J. Desautels, University of Wisconsin-Madison
Consulting Editor

Copyright © 1984 by Wm. C. Brown Publishers. All rights reserved.

Library of Congress Catalog Card Number: 83—73414

ISBN 0—697—09912—1

2—09912—01

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Printed in the United States of America.

Contents

Preface xi

1 *Introduction to the Commodore 64* 1

COMMODORE 64 HARDWARE 1
MEMORY 2
BASIC 2
DOS (DISK OPERATING SYSTEM) 3
REVIEW QUESTIONS 3

2 *Getting Started on the Commodore 64* 5

TURNING ON THE COMMODORE 64 5
KEYBOARD CHARACTERS 6
- Commodore 64 Graphics Keyboard 6

USING THE CASSETTE TAPE 7
- Inserting a Cassette 7
- Tape Controls 8
- Loading a Program 8
- Saving a Program 8

USING THE FLOPPY DISK DRIVE 9
- Loading a Program 9
- Saving a Program 10
- Replacing a Program 10
- Directories 10
- Formatting a Diskette 10
- Backups 11

REVIEW QUESTIONS 11

3 *Elementary BASIC Programming* 13

WHAT IS BASIC? 13
CALCULATOR MODE 13

HOW TO USE CURSOR CONTROLS 14
INSERT/DELETE 15
CLEAR/HOME 15
RUN/STOP 15
LIST COMMAND 16
BASIC NUMBERS 16
- *Integers* 17
- *Real Numbers* 17
- *Scientific Notation* 19

STRINGS 20
VARIABLES 20
MULTIPLE STATEMENTS PER LINE 22
USING COLOR 22
ARITHMETIC STATEMENTS 23
- *Add (+)* 23
- *Subtract (−)* 24
- *Multiply (*)* 24
- *Divide (/)* 24
- *Exponentiation (↑)* 24

HIERARCHY AND PARENTHESES 24
PRINT 25
INPUT 27
GOTO 28
REM 29
SIMPLE CALCULATION PROGRAM 29
FAHRENHEIT—CELSIUS PROGRAM 30
IMPROVING YOUR SOLUTION 32
ELEMENTARY GRAPHICS—SAILBOAT 33
WEIGHTED AVERAGE PROGRAM 33
COMPUTING LOAN PAYMENTS 36
REVIEW QUESTIONS 37

4 *Not So Basic BASIC* 39

DIM 39
- *Storing Values in an Array* 40
- *Multidimensional Arrays* 41

IF—THEN 42
FOR—NEXT 44
GOSUB—RETURN 45
STOP 47
GENERATING RANDOM NUMBERS 47
NUMBER GUESSING GAME 48
TIME DELAYS 50
IMPROVED WEIGHTED AVERAGE PROGRAM 52
ESTIMATING TRIP COSTS 53
REVIEW QUESTIONS 57

5 *More on Input and Output* 59

READ—DATA 59
CREATING A BAR GRAPH 60
WEIGHTED AVERAGE WITH DATA 61

RESTORE *62*
MORE ABOUT PRINT *63*
- ***Cursor Controls*** *63*
- ***TAB*** *65*
- ***SPC*** *65*

METRIC CONVERSION PROGRAM *66*
SIMPLE PAYROLL PROGRAM *74*
REVIEW QUESTIONS *77*

6 ***Advanced BASIC*** *79*

GET *79*
ON—GOSUB *80*
ON—GOTO *81*
POKE *83*
PEEK *85*
RANDOM STARS *86*
TI AND TI$ FUNCTIONS *86*
- ***TI or TIme Function*** *86*
- ***Reaction Timer Program*** *86*
- ***TI$ or TIme$*** *87*

DEF FN *88*
ARITHMETIC FUNCTIONS *90*
- ***ABS*** *90*
- ***ATN*** *90*
- ***COS*** *90*
- ***EXP*** *91*
- ***INT*** *91*
- ***LOG*** *91*
- ***RND*** *91*
- ***Creating a Specific Set of Random Numbers*** *92*
- ***SGN*** *92*
- ***SIN*** *92*
- ***SQR*** *93*
- ***TAN*** *93*
- ***Converting Radians to Degrees*** *93*

STRING FUNCTIONS *93*
- ***ASC*** *93*
- ***CHR$*** *94*
- ***LEFT$*** *94*
- ***LEN*** *95*
- ***MID$*** *95*
- ***RIGHT$*** *97*
- ***STR$*** *97*
- ***VAL*** *97*

CONCATENATION *97*
PLOTTING GRAPHS *98*
CONTROLLING DECIMAL POSITIONS *102*
- ***Driver Routines*** *104*

CAI CHESS AND PROGRAM GENERALIZATION *105*
- ***The Data*** *105*
- ***The Program*** *107*

REVIEW QUESTIONS *112*

7 *Interacting with the User of Your Program* 113

USER LEVEL 113
1. Casual 113
2. Trained 113
3. Programming Skills 113
USER DIALOGUES 114
Prompting 114
DEFAULT RESPONSES 115
MENUS 116
MULTILEVEL MENUS 117
FORM FILLING 119
COMMAND LANGUAGES 119
REVIEW QUESTIONS 120

8 *Graphics, Animation, Sound, and Music* 121

GRAPHIC CHARACTER SET 121
REACTION TIMER WITH GRAPHICS 124
USING POKE TO PRODUCE A GRAPHIC 126
Lunar Lander 126
Chessboard 127
ANIMATION 128
Rocket 1 128
Rocket 2 129
Egg Timer 130
Train—Animating Multiple Objects 131
HIGH RESOLUTION ON THE COMMODORE 64 132
SPRITE GRAPHICS 135
Sprite Design 135
Sprite Registers 136
Sprite Movement 137
Moving a Sprite Horizontally 138
Moving a Sprite Vertically 138
Sprite Size 139
Sprite Color 139
GENERATING SOUND AND MUSIC 139
Volume 140
Attack/Decay Setting 140
Sustain/Release Setting 141
Frequency 141
Waveform Settings 142
Playing a Single Note 143
Take Me Out to the Ballgame 143
Note Duration 143
Sound Characteristics 144
REVIEW QUESTIONS 145

9 Tape Files Extend Your Reach 147

CONCEPTS 147
OPEN AND CLOSE 148
PRINT# 149
WRITING BUDGET NAMES ON TAPE 149
INPUT# 150
READING THE BUDGET NAMES TAPE 151
HOW TO HANDLE RECORDS WITH MULTIPLE FIELDS 152
UPDATING A TAPE FILE 154
THE CASH-FLOW PROBLEM 156
- New File 157
- Old File 157
- Enter/Review Data 157
- Display Balance 158

SUMMARY OF TAPE FILES 164
REVIEW QUESTIONS 164

10 Disk Files 165

SEQUENTIAL FILES 166
OPEN AND CLOSE 166
PRINT# AND INPUT# 168
WRITING BUDGET NAMES ON DISK 168
READING BUDGET NAMES FROM DISK 169
HOW TO HANDLE MULTIPLE FIELDS ON DISK FILES 169
DETECTING DISK ERRORS 171
MAINTAINING THE CHECKBOOK 172
REPLACING A CURRENT FILE 172
USING VARIABLE FILENAMES 172
ENGLISH CODE 173
DISPLAY SCREEN PAGE 174
DELETING TABLE ENTRIES 175
REVIEW QUESTIONS 179

11 How to Debug Your Programs 181

DESK CHECKING 182
SYNTAX ERRORS 182
TEST DATA PREPARATION 184
IMMEDIATE MODE DEBUGGING 184
TRACING PROGRAM LOGIC 185
- Manual Tracing 185
- Automatic Program Tracing 188

REVIEW QUESTIONS 194

Appendices *195*

- **A** ***Basic Operating Commands: CLR, LIST, LOAD, NEW, RUN, SAVE, VERIFY*** *195*
- **B** ***Reserved Words*** *198*
- **C** ***Abbreviations*** *199*
- **D** ***Disk Error Messages*** *200*
- **E** ***C-64 ASCII and PEEK/POKE Codes*** *201*
- **F** ***ASCII and CHR$ Codes*** *202*

Index *205*

Preface

This book introduces the BASIC language used on the Commodore 64 microcomputer. If you are a beginner at programming in BASIC, this material will give you a gradual introduction to the language, stressing a hands-on approach with your Commodore 64.

The BASIC used here is BASIC 2.0, released by Commodore since 1981. It is virtually identical to the BASIC used on Commodore PET and CBM computers and the Commodore VIC-20, with a few slight differences where disk applications are concerned. Major differences between the VIC and C-64 are the color graphics and music capabilities, which are far more extensive on the Commodore 64.

The first chapter is a brief introduction to the C-64 hardware with BASIC and DOS for the floppy disk. Chapter 2 discusses the use of the keyboard, tape, and floppy disk. If you have used the C-64 before, you may skip these chapters or read them quickly.

Chapter 3 begins with an elementary introduction to BASIC , stressing a hands-on approach. By entering the program code on your C-64 as you read, you will get immediate feedback from your computer. Enough BASIC statements are introduced in this chapter for you to begin to write some useful programs at this early stage.

Chapter 3 ends with six complete programs that may be run on the C-64. In these six programs, two things are emphasized.

One is the need for planning a program. The approach taken shows how input and output must be defined before the program is written and then how to make use of an English code (Pseudo code) to develop the general program logic. Later the flowcharts concept is also discussed.

The second emphasis in the sample programs is how to apply the language statements just discussed in the chapter to a variety of situations. Each program is designed to expose you to realistic situations requiring the use of the BASIC statements you have just read about.

Subsequent chapters follow the same pattern as chapter 3 but go into more depth in the language, as indicated in the table of contents. Each chapter has numerous examples and ends with several programs that apply the new features of the language. Each program has been completely developed using the techniques of program design to establish the program logic.

Chapter 7 deviates slightly from this pattern to discuss ways we can communicate with our program users. Since you will sooner or later write programs for other people, this chapter considers effective interaction with users of your program and how to implement these concepts on the C-64.

Next, in chapter 8, we look at graphics, animation, and sound. These topics will be of particular interest since these are powerful capabilities for such an inexpensive computer. Chapter 9 covers the use of sequential files on tape, and chapter 10 explains sequential files on disk. Again, there are examples and programs to try as we make the concepts clear.

Finally, chapter 11 discusses the procedure for debugging your programs. As you will no doubt discover very early, programs don't just work immediately after you have written them. In fact, a program that works perfectly the first time is the exception rather than the rule. So this chapter discusses some techniques for finding your bugs and correcting them.

Remember, this is an introduction to BASIC programming on the Commodore 64. The book does not pretend to be an exhaustive treatment of programming, and there is much more to be learned about the C-64 beyond this level. However, I hope you will find the book instructive and helpful in your quest for learning to program, and I trust it will provide the necessary foundation for you to move on to more advanced programming on your Commodore 64.

1 Introduction to the Commodore 64

BASIC is the language the majority of microcomputers use, so by learning it on your C-64 you will be prepared to apply programming to a wide variety of micros.

Possibly you purchased your C-64 to play some of the exciting games available on cartridge or tape but now you want to explore some of the C-64's further capabilities. Many people have used the C-64 for a wide range of solutions to problems, running from home finance, record keeping, and investment to music skills and child education. The list is endless. However, before you can apply the computer to your interests, you must learn to use the tool effectively. After studying the contents of this book you will be equipped to solve many of your programming applications with BASIC. But first let's look at the hardware.

COMMODORE 64 HARDWARE

Figure 1.1 shows the C-64 microcomputer with the floppy disk drive, printer, and tape used with the Commodore computer. Hidden inside the C-64's case are the electronic components, a microprocessor and a memory, necessary for its operation.

To understand C-64 basics, it is useful to compare the computer to parts of the human body. For example, when you want to learn something, you might read a book, as you are doing now. The new information becomes input to your brain, where it is stored in memory. Similarly, you can enter new information into the computer through the keyboard and then store it temporarily in the C-64's electronic memory or storage.

After you have read something, you may think about it by processing it mentally. Then you might discuss your conclusions with someone or maybe simply repeat verbally what you have read. Your voice is then output from the brain. When information has been processed by the computer's processor (brain?), that same information or a new arrangement of it may be displayed on the TV screen, which is the C-64's output device.

These physical components of the computer—display, keyboard, processor, and so on—are called the hardware. The program used to describe the processing steps is called the software. The program, residing in the computer's memory when it is being used, describes in precise steps how the computer is to process the data.

If your brain is anything like mine you will not likely remember everything you have read. One solution to this is to make notes, another kind of output. Although the computer never forgets anything entered into its memory, it does have limited capacity. Therefore another form of output is often

Figure 1.1 C-64 hardware

used to record or store information outside the computer. One of these is the tape cassette. Another is the floppy disk or diskette. Information recorded on the tape or disk can be stored and read back into the computer at a later time when needed, just as you can read your notes to jog your memory. Tape and disk are commonly used to store programs as well as other data.

MEMORY

When you type something on the keyboard or read from tape or disk, the information goes into the C-64's memory or RAM (Random Access Memory) as shown in figure 1.2. RAM is a solid-state memory device that is a part of the C-64's circuitry, and programs or data may be read or written into it. Normally only one program at a time goes into memory; when a new program is read from tape or disk it replaces the previous program in memory.

Memory in the C-64, as the name implies, is 64K or 64,000 storage positions. Each storage position may hold only one character (i.e., a letter or a number), so a 64K Commodore 64 can store 64,000 characters, or bytes in computer lingo. (Technically, 1K bytes is 1,024 bytes, so 64K is actually 65,536 bytes.) BASIC programmers have access to only 38,911 bytes since the computer uses the remaining memory. But even 38,911 bytes will give you lots of room for some sophisticated programs.

Also shown in figure 1.2 is a component called ROM, for Read Only Memory. As the name suggests, ROM may be read only by the computer. The most significant part of ROM is the BASIC interpreter it contains. This interpreter is a program that has been prerecorded in the C-64's ROM, which the computer uses to run your BASIC programs.

BASIC

BASIC is the primary language for programming the C-64, the means by which you give instructions to the computer to solve a particular problem. BASIC, like human languages, has rules that must be followed to use the language effectively. Fortunately, as the name BASIC suggests, it is not nearly as complicated as English or French. In fact, BASIC is one of the easiest of all computer languages to learn.

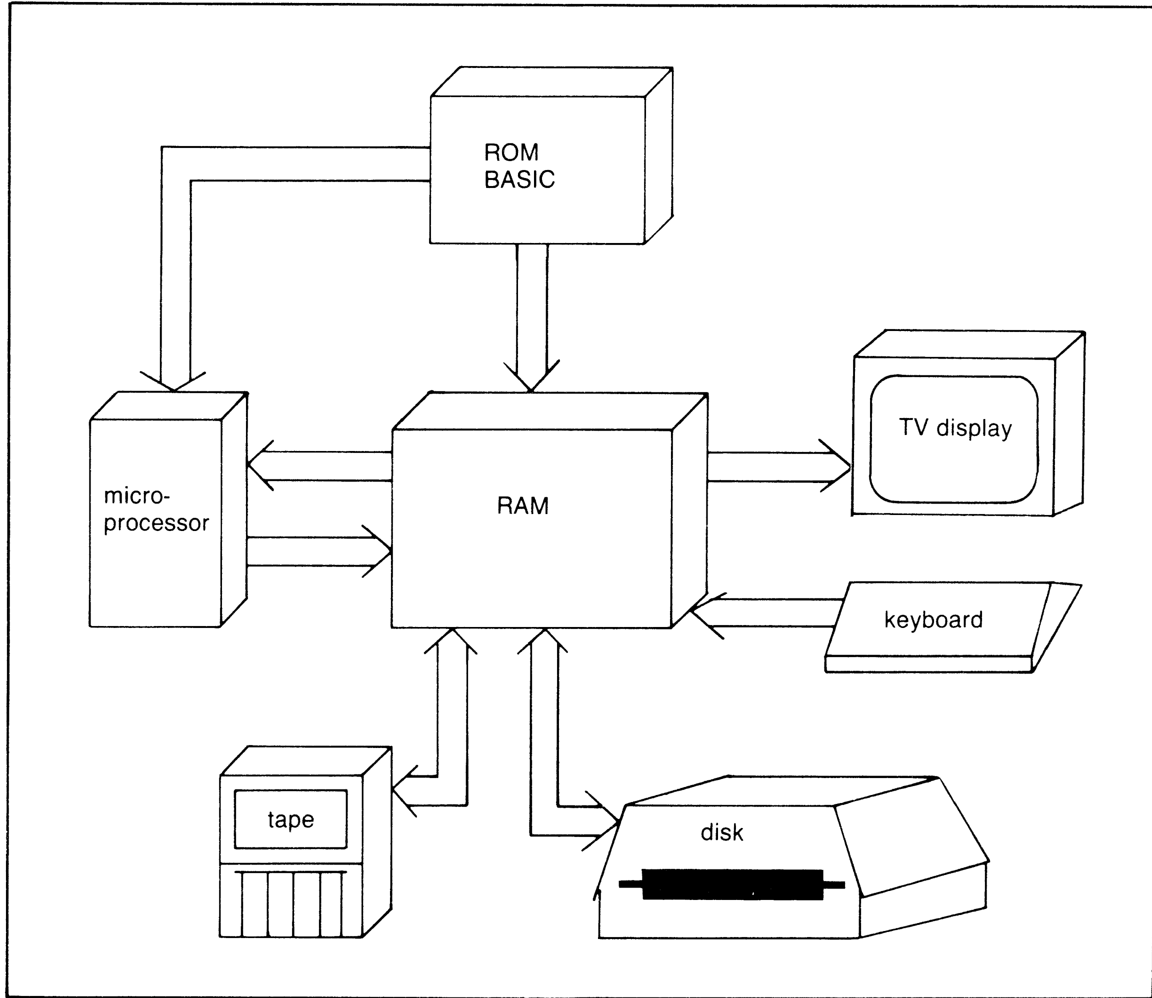


Figure 1.2 Components of a Commodore 64 system

DOS (DISK OPERATING SYSTEM)

DOS is the program in the floppy disk drive that controls the reading and writing of disk files. Without DOS, working with disk would be terribly complex. Although we will introduce the use of DOS at appropriate times, most of our discussion will refer to tape as the primary storage device. So if you don't have a disk drive, don't worry; tape will work equally well.

REVIEW QUESTIONS—CHAPTER 1

1. Name some of the components of the Commodore 64 microcomputer.
2. Compare the concepts of computer input, process, and output to the human body. Can you think of any other analogy?
3. What is hardware? Give some examples.
4. What is software? What purpose does it serve in the microcomputer?
5. What does RAM mean?
6. Discuss BASIC's function on the C-64.
7. What device must your computer have if you need to use DOS?

2 Getting Started on the Commodore 64

*I*f you are familiar with the C-64, you might wish to skip ahead to chapter 3. Otherwise you should read this chapter to learn its basic operation.

The C-64 (figure 2.1) comes with the full-sized graphics keyboard and 64K of RAM. A TV is used for the display, which shows 40 characters per line. The most basic configuration for the C-64 is with a cassette tape; the floppy disk and/or printer are possible additions.

Figure 2.1 C-64 microcomputer

TURNING ON THE COMMODORE 64

First be sure the C-64's power supply is plugged into a wall outlet and that the other end of the power supply cord is plugged into the power supply socket on the side of the C-64. A video cable plugs into the back of the C-64 and the other end connects to the RF modulator. Then attach the modulator wire to the VHF antenna on your TV set and tune the TV to channel 3. Turn on the TV. If channel 3 has interference on your set, try changing the channel selector on the back of the C-64 and set the TV to channel 4.

Now press the power switch on the right side of the C-64 to the ON position. In a few seconds the TV screen will come to life and display the following characters:

```
****COMMODORE 64 BASIC V2****

64K RAM SYSTEM 38911 BYTES FREE

READY.
```

KEYBOARD CHARACTERS

Commodore 64 Graphics Keyboard

Figure 2.2 illustrates the C-64's keyboard. It is arranged with 62 keys for alphabetic, numeric, special, and graphic characters. The keys select capital letters from A to Z and such special characters as $ % '' , and ( ), which are used mainly as special programming symbols. At the right of the keyboard are four function keys that may be programmed to perform special activities.

The keyboard contains most of the C-64 graphic symbols on the front of each key. To select the graphic symbol on the right of the key, you must hold down one of the two shift keys or the shift/lock while you also press the appropriate graphic key. Try holding the shift while you press the Q key. You should get a solid light blue circle on the screen.

To get the graphics on the left of the keys, hold the Commodore key (the one with the Commodore logo beside the shift key) down while you press the graphic key. Try holding the Commodore key while you press the * key. This time you should get a light blue triangle on the screen.

Figure 2.2 Commodore 64 keyboard

At the right of the main keyboard is the Return key. When you make an entry to the C-64, such as a BASIC statement or a response to a question from a program, pressing the Return key will enter that line into memory.

On the left of the keyboard, above the Commodore key, is the RUN/STOP key. This key may be used to STOP a program or to RUN (by holding shift and pressing RUN) a program from tape.

On the left of the keyboard is the CTRL (control) key. When this key is held down, other keyboard features are activated. Try this: Hold down the CTRL and simultaneously press the 9 key. Now type some characters. Notice that they are now dark blue on a light-blue background. This is called a reverse character, which explains the RVS ON on the 9 key. You have activated the reverse character feature of the C-64. Now hold down the CTRL key and press 0 (zero), which turns the RVS OFF. Now type the same characters and notice the difference.

The CTRL key also may be used to change the color of the characters you type. Hold CTRL and press the RED key. First the cursor, that blinking rectangular symbol flashing on the screen, changes to red. Now any characters you type will also be in red. Holding CTRL and pressing another color will change the cursor once again. Use RUN/STOP and the RESTORE keys simultaneously to get things back to normal.

Below the Return key are two cursor (CRSR) control keys to control cursor movement. Using these keys with the shift key will cause the cursor to move up, down, left, or right.

Press the down-pointing arrow and then the right-pointing arrow and see what happens. Notice how the cursor moves in the direction of the arrow each time you press the key. Now press the HOME key. The cursor should return immediately to the upper-left corner, or home position, of the screen.

In the next chapter we will examine these keys and others in more detail. For now, experiment with the keys and see what effects you can get. Don't worry about damaging your C-64. Nothing you can do on the keyboard will harm it. If it locks so nothing seems to work properly, all you need to do is turn the C-64 power off and then on again. Everything will come back to normal, although anything you have typed will be lost.

USING THE CASSETTE TAPE

If you plan to use tape to store and load your programs, you will want to read this section. However, if you have the disk drive, skip this section and go on to the following section, on using disk.

Using the cassette drive (figure 2.3) is a convenient low-cost way to load a program that has been previously written on the C-64 and saved on cassette tape. The C-64's RAM can hold only one program at a time, but the only limit to the number of programs you may have depends on the number of cassettes and their length.

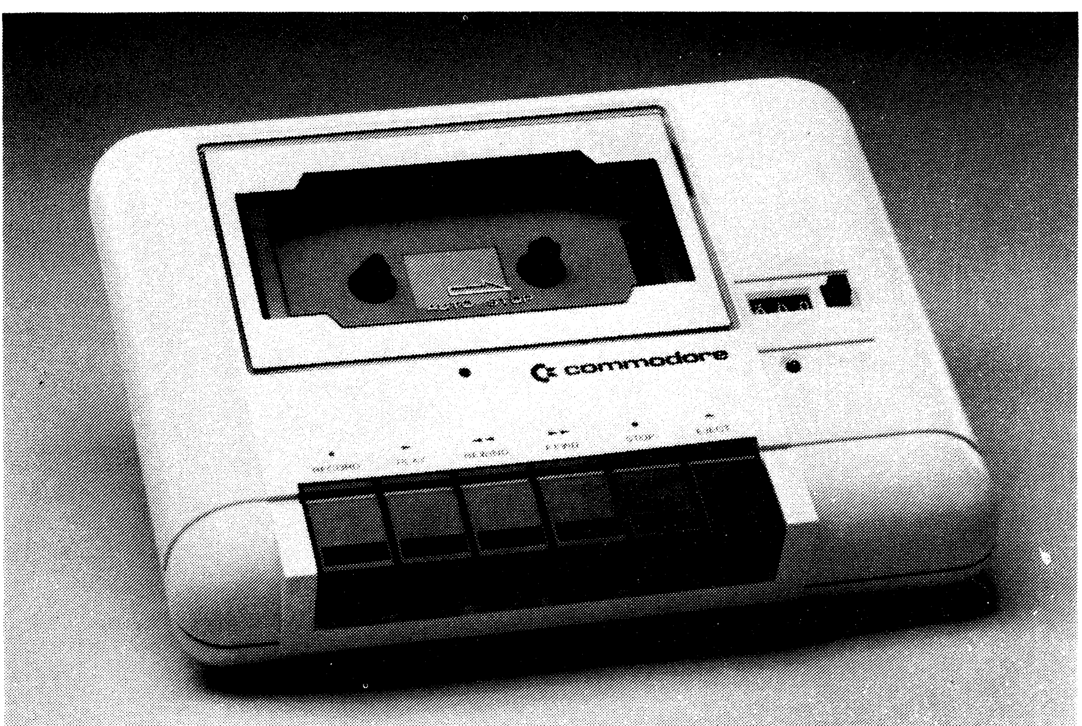


Figure 2.3 Cassette tape drive

When purchasing tapes for the C-64, it is best to use relatively short tapes C-10 to C-30 and store only one or two programs per side. If you store many more programs on a single tape, the waiting time for a program to load becomes too long. Tapes should be of the low-noise, high-quality, and high-output variety. Avoid the cheap brands.

Inserting a Cassette

Inserting a tape in the C-64's drive is no different from inserting a music or audio cassette in a recorder. First press the EJECT key to open the door. Now insert the tape with the open end toward you and the label facing up. If it won't go in, don't force it. You are probably putting it in the wrong way, so turn it around and try again. Now press the door closed.

Tape Controls

The tape controls are like other cassettes except the C-64's tape does not have volume or tone controls. The controls you will need are:

1. RECORD—This is the record key used simultaneously with the play key when you want to save a program from the C-64's RAM onto a cassette tape.
2. REWIND—The rewind key ensures that the tape begins at the leader before any program is reached. It will also be used to rewind the tape after the desired program has been read.
3. FFWD—The fast-forward key lets you advance the tape.
4. PLAY—Press this key when you load a program into RAM from the tape.
5. STOP—This key will stop reading or saving, but you use it normally to stop the tape when you are finished with it.

Loading a Program

To load a program from tape, simply type the command

```
LOAD
```

and press RETURN. If the name of the program is known, such as LOW BALL, then you could type

```
LOAD"LOW BALL"
```

and press RETURN.

In the first case the computer would simply look for the first program on the tape and load it. When you supply the program's name, the computer will load only a program with that name, bypassing any other program it happens to find on the tape.

After you press Return, the C-64 will respond with

```
PRESS PLAY ON TAPE
```

When you press the play button, the screen will go blank as the C-64 looks for your program. After a few seconds it should display

```
FOUND LOW BALL
```

or a different program name depending on the tape you are using. Now press the Commodore key to load the program. Once the program is loaded, which might take as long as a minute, the message

```
READY
```

will appear. If you get a Load Error, rewind the tape and start again from where we typed LOAD.

Now type

```
RUN
```

and press Return. Your program will now begin to run.

Saving a Program

When you write your own programs, you may save them for future use on a cassette tape. First, the program must be in the C-64's memory. This could be a program you have typed in or it could have

been loaded from another tape. The C-64 really can't tell the difference. To save the program, type

```
SAVE"LOW BALL"
```

which means we name the program LOW BALL. Make sure a blank cassette is in the drive and then press Return on the C-64. The C-64 displays

```
PRESS RECORD & PLAY ON TAPE
```

Do this, being sure both buttons are pressed simultaneously. The screen now will go blank as the C-64 saves the program on the tape. When it is finished saving the program, the message

```
READY.
```

will appear. To ensure that the save worked, you can now rewind the tape and type VERIFY. Press Return and follow instructions. The C-64 now checks each character on the tape with each character in its memory. If all went well, the message

```
OK
READY.
```

will appear on the screen.

USING THE FLOPPY DISK DRIVE

The advantage of having a disk drive (figure 2.4) is its greater speed in reading programs or data. A program that takes a minute to load from tape will require only two or three seconds from disk.

Loading a Program

First, be certain the disk power is on. Never turn the power on or off while a diskette is in the drive. Now, to use disk, insert a diskette (with the oblong- or oval-shaped opening pointing away from you) containing the program you want to run into the drive, and gently close the door.

Figure 2.4 C-64 floppy disk drive

Now type the following command on the C-64 keyboard:

```
LOAD"program name",8
```

If your program is called BUDGET, then type

```
LOAD"BUDGET",8
```

and press Return.

The program will now load into memory. When the C-64 has loaded the program, the message READY will appear. Now type RUN to run the program.

Saving a Program

To save a program on diskette from the C-64's memory, use the SAVE command:

```
SAVE"program name",8
```

Originally the BUDGET program could be saved on disk by typing the command

```
SAVE"BUDGET",8
```

and pressing Return.

After you have saved programs on the floppy disk, place an adhesive tape over the notch on the floppy's container. This tape will protect the disk from writing over your programs or data accidentally. If you wish to add additional programs at a later time, the tape must first be removed before inserting the diskette into the drive.

Replacing a Program

Sometimes you already have a program on the disk and because of changes you want to replace the disk copy with the copy in memory. If the program is called CHECKS, use the following command:

```
SAVE"@CHECKS",8
```

By using the ''at'' sign (@) before the name, we are telling the disk drive to replace the program's old version with the new.

Directories

Diskettes also contain directories that list their contents. If at any time you want to know what is on a disk, list the directory with the commands:

```
LOAD"$",8
LIST
```

During the display of a directory, the scrolling may be stopped by pressing the Space bar to let you more easily read its contents. Pressing Space again will continue the display.

Formatting a Diskette

When you start with a blank diskette, or if you want to erase a diskette you wish to use over again, you must format the diskette. Be careful when you use this procedure, for it will completely erase all files and programs on a disk. It is, however, a necessary operation for all new diskettes.

The following statements show how to format.

```
OPEN1,8,15
PRINT#1,"N0:diskname,id"
CLOSE1
```

The OPEN statement prepares a disk channel for use. The PRINT#1 identifies the operation, "N" meaning "new," the disk drive "0," a diskname that may be up to 16 characters, and an ID of two characters. The ID is usually used to indicate whether there is more than one disk of this diskname.

As an example, if you are creating a disk to store all your financial records, the following formatting may be done.

```
OPEN1,8,15
PRINT#1,"N0:FINANCES,01"
CLOSE1
```

This command identifies the diskette as FINANCES and uses an ID of 01, indicating that it is the first diskette with this name. If you needed two diskettes for storing your financial information, the ID for the second diskette could be 02, and so on.

Backups

When you have stored a number of programs on disk, you might occasionally destroy them. Usually this happens inadvertently, but the loss can be shattering, especially if it represents hours of your time. Keeping a second copy on another diskette as a backup is well worth the small amount of time and expense necessary.

Create a backup by first loading a program into memory from the master diskette. Then save the program on the backup diskette. This means a lot of inserting and removing of disks, but it's worth the extra effort.

REVIEW QUESTIONS—CHAPTER 2

1. Describe the basic characteristics of the Commodore 64 microcomputer.
2. When the power to the C-64 is turned on, what does the message "38911 BYTES FREE" mean?
3. How are the graphic characters on the front of the keys selected on the C-64 keyboard?
4. What is the purpose of the RUN/STOP key?
5. How do the RVS ON and RVS OFF keys work?
6. How can you change the color of the screen characters?
7. Explain the commands for loading a program from tape into memory and running it.
8. If you have a program currently in memory, how could you save this program on a cassette tape with the name C-64 BOOKS?
9. Explain how to load and save programs to disk using the name C-64 BOOKS.
10. What is the purpose of a directory? How can you get one from the disk?

3 Elementary BASIC Programming

WHAT IS BASIC?

The name BASIC is a term copyrighted by the trustees of Dartmouth College and is an acronym for Beginner's All-Purpose Symbolic Instruction Code. Professors John G. Kemeny and Thomas E. Kurtz developed BASIC on a time-sharing computer system in 1963. They designed it with the purpose of making programming an easier task for the average person who had little computer background or knowledge.

As the name implies, BASIC is relatively easy to learn, but since 1963 many enhancements have been added to the language to extend its usefulness. These developments have led to many new versions of which Microsoft BASIC for the C-64 is one. Learning BASIC for the C-64 gives excellent grounding for using it in most other computers.

CALCULATOR MODE

One of the C-64's features is its ability to perform as a calculator, a sophisticated one. In general, calculator or immediate mode is in use whenever a BASIC statement is typed without a statement number. For instance, type the statement:

```
PRINT 5*2.7
```

and press Return. This statement is in calculator mode and the answer

```
13.5
```

is displayed immediately. If, however, you type the statement

```
10 PRINT 5*2.7
```

the machine is in program mode because a statement number (10) is used. The C-64 stores this statement in its memory instead of immediately calculating the answer.

Now type

```
?14/2
```

and press Return. Did the answer 7 appear? The question mark is an abbreviation for the command PRINT and is a much more convenient form of the command.

Complex calculations may be done in immediate mode by combining several arithmetic operations together. For example, try entering the following statement on your Commodore 64.

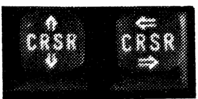
```
PRINT 15000*(.01/(1-(1+.01) -48))
```
← press RETURN

When you press Return, the answer given is

```
395.007536
```

Try a few more calculations to get the feel of the C-64's capability. Don't worry about making mistakes or doing something that might damage the machine. Nothing you type in can harm it.

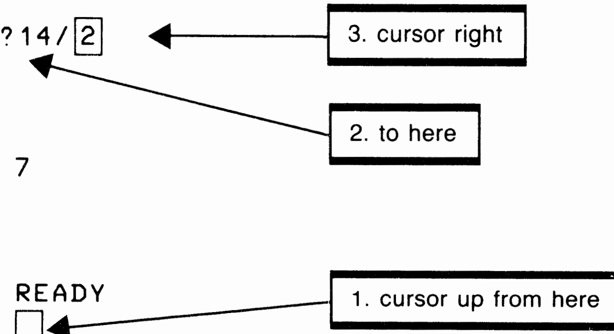
HOW TO USE CURSOR CONTROLS



Do you recall the cursor controls at the bottom right of the keyboard? These are the arrows that point up, down, left, and right. Now reenter the command

```
?14/2
```

and press Return to get the answer. Next use the up cursor control to move back to the command and the right cursor to move to the 2. Like this:



Notice the cursor moves over the characters without changing them in any way. Now type a 5 on top of the 2. When you press Return the answer given is 2.8, replacing the previous answer.

If you press any cursor key once, it will simply move the cursor one position on the screen. But if you hold it down, "auto repeat" is in effect. The cursor will continue to move across the screen until the key is released. This can be useful when you want to move across several characters at once.

Now try the statement

```
?245*598
```

The answer should be 146510

INSERT/DELETE

The Insert/Delete keys are situated at the upper-right corner of the keyboard. This key allows you to make space in a line to insert characters that were missed or to delete characters that should not be there.

Move the cursor up to the 9 of 598 from the previous exercise. The screen looks like this:

Hold the shift key and simultaneously press INST once. Now the screen shows

leaving a space between the 5 and the 9. Type a decimal point (.) so you get

```
?245*5.98
```

and then press Return. The answer 1465.1 replaces the previous result.

To use the Delete, move the cursor to the character to the right of the one you wish to delete and press the DEL key. Try this to change the 5.98 to 5.8, and then press Return to get your answer.

Did you move the cursor to the 8? If so, the answer you received should be 1421.1.

Cursor controls can be used to correct any entry on a line. For instance, you might try typing a sentence and then make changes in it using the cursor controls with the Insert and Delete.

CLEAR/HOME

This key is located at the upper-right corner of the keyboard beside the Insert/Delete key. Pressing the HOME key causes the cursor to move immediately to the upper-left corner of the screen, called the Home position. Pressing shift and CLR together clears the screen and homes the cursor. This is sometimes useful if you wish to remove the displayed results from a program before going on to the next one. Clearing the screen does not clear program memory, only the screen itself. Memory may be cleared by using the NEW command.

Enter the BASIC code:

```
10 PRINT10/5+35
```

When you press Return no answer is given since this is a BASIC statement in program mode, not immediate mode.

Hold shift and press CLR, clearing the screen. Now type the command LIST and notice what happens. The C-64 displays the command you previously entered. Statement 10 is still available since it was stored in memory. Type the command RUN and the program runs, displaying the result 37.

RUN/STOP

Although well-written programs will always provide a way to terminate the procedure when you are finished, occasionally you may get into a program that has not provided a way to get out. If so, you can exit from a program by pressing STOP. The C-64 confirms your instruction by displaying

```
READY.
```

followed by the flashing cursor.

Now try this program:

```
10 PRINT I
20 I=I+1
30 GO TO 10
```

Type RUN and after a few seconds press STOP. This will bring the program to a screeching halt. In rare cases a programmer may have disabled the STOP. If this happens, the only alternative you have is to turn off the power and then turn it on again. Since this action erases program memory, use it only as a last resort. You can hope that the program was saved on tape so it can be loaded once again.

A better solution is to press the RUN/STOP and RESTORE keys simultaneously. This has the same effect as turning the power off and on, but it does not destroy the contents of memory.

The RUN key (shifted) lets you load and run a program from tape. This is equivalent to typing a LOAD and RUN except the RUN key does not permit the entry of a program name.

LIST COMMAND

Now, while you still have the above program in your computer, try the LIST command as follows:

```
LIST  ←——— press RETURN
```

This command will cause the program to be listed on the screen as it was originally typed into memory. LIST works fine if the program is short, like this one, but if it is much longer the entire program will not fit on the screen at one time. Instead, it will quickly scroll past your eyes.

One solution to this problem is to hold the CTRL key down when a program is listing. This action slows down the display's speed so you can see each line as it appears. You must still be somewhat of a speed reader if you expect to read each line.

A better solution is to list a range of lines in the program. For example, type

```
LIST 10-20
```

This command will list all lines between 10 and 20, which in the case of the above program includes only two lines. Typing the command

```
LIST -20
```

will list all lines up to and including line 20. The command

```
LIST 10-
```

lists lines 10 and above. In this example the entire program is again listed. By using this form of the LIST, a range of lines that will fit on the screen at one time may be selected for listing.

BASIC NUMBERS

A lot of programs are number oriented and those that are not, such as games and graphic programs, usually need some numbers for their operation. BASIC uses two types of numbers: Integers and Real (Floating Point). Another type of program uses Scientific Notation, which is simply a variation of the Real number. Numbers may be used in programs for calculations, comparisons, as data in DATA statements, or as input from the keyboard. If theory doesn't turn you on, skim the following pages.

Integers

An integer is a number with no fractional part, thus no decimal point. The integer may have a sign for negative (−) or positive (+) numbers. Some examples of integers are:

```
     1
    12
 -2358        INTEGERS
   -25
+31278
```

Integers have a maximum range from −32768 to +32767. An integer uses only two bytes of memory and can therefore save a lot of memory space in large programs. Why two bytes? Well, a byte occupies eight bits, or binary digits. Two bytes have 2 × 8 or 16 bits available for use. One of these bits is used for a sign, leaving 15 for storing numbers. Now each binary digit can store two possible values: 1 or 0. With 15 bits available, this allows a total of 15 to the power of 2 numbers, which is 32,768. If one of these numbers is reserved for the value zero, this leaves 32,767 as the maximum. Makes sense, doesn't it?

Integers may also be represented as real numbers.

Real Numbers

A real number can be an integer, a fractional number, or a combination of both. Real numbers, like integers, may also be signed. For example:

```
  1
  1.5
276.075        REAL
   .0125       NUMBERS
 -123456789
 -.0000001
```

A real number in the C-64 uses five bytes of memory and can have 8 or sometimes 9 digits, not counting the sign or decimal point.

If excessive digits are to the right of the decimal point, roundoff occurs. Try entering

```
?.9876543216        ROUNDING
.987654322
```

The 6 is truncated and the last digit, 1, rounds up to 2.

```
?.9876543211        TRUNCATING
.987654321
```

In this case, the last 1 is truncated but the remaining 1 is not rounded since the next digit is less than 5.

```
?.9876543214
.987654322
```

This number rounds incorrectly due to the conversion to floating point done internally in the C-64's memory.

Real numbers are useful for the majority of applications on the C-64, whether for business, school, or the home. Real numbers can represent dollars, quantities, marks, account numbers, and so on.

Now try this interesting exercise:
Light travels at a speed of 186,200 miles per second. How many miles does it travel in a minute? In an hour? A day? A year? The distance light travels in a year is called a light year. So using the C-64 as a calculator, try these calculations. Using the cursor to change the Print command each time will make this easier.

Distance light travels in a minute

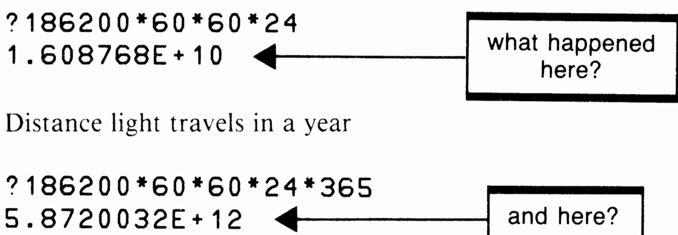
```
?186200*60
11172000
```

Distance light travels in an hour

```
?186200*60*60
670320000
```

Distance light travels in a day

```
?186200*60*60*24
1.608768E+10
```



Distance light travels in a year

```
?186200*60*60*24*365
5.8720032E+12
```

Real numbers can also be small fractional values. For instance, computers do things in very small fractions of a second. These measurements are in milliseconds (1/1000 of a second), microseconds (1/1,000,000), nanoseconds (1/1,000,000,000), and the really fast ones are in picoseconds (1/1,000,000,000,000). C-64s basically operate in microseconds. Now let's find out how far light or electricity travels in these small periods of time.

Distance light travels in a millisecond

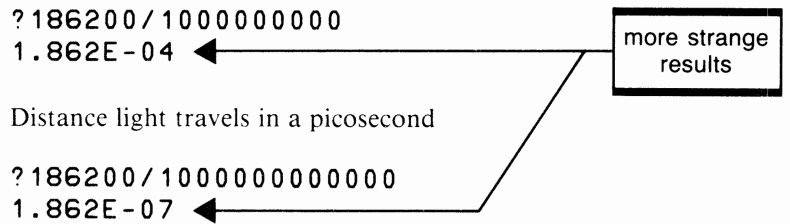
```
?186200/1000
186.2
```

Distance light travels in a microsecond

```
?186200/1000000
.1862
```

Distance light travels in a nanosecond

```
?186200/1000000000
1.862E-04
```



Distance light travels in a picosecond

```
?186200/1000000000000
1.862E-07
```

Did you know that the nearest galaxy to Earth is Andromeda? It is 2,300,000 light years away. Can you use your C-64 to find out how many miles that is?

Now let's find out about those weird numbers we got in some of the previous calculations.

Scientific Notation

Real numbers may also be represented in scientific notation if they exceed their defined maximum size. The result of the calculation for the number of miles light travels in one day exceeded the maximum size for a real number, so the C-64 automatically converted it to scientific notation. The nanosecond result was smaller than the smallest real number, so it was converted to a small fractional number in scientific notation. Try typing a 1 followed by 9 zeros.

```
?1000000000
1E+09
```

The result is a number (1E + 09) in scientific notation that represents the original number in powers of 10. This number may be thought of as

$1 X 10^9$

Some valid scientific notations are:

Scientific Notation	Actual Value
2.7385E8	273850000
1.085215E – 7	.0000001085215
– 45E12	– 45000000000000
– 21E – 15	– .000000000000021

The maximum ranges for floating point numbers in scientific notation are

Largest:	± 1.70141183E + 38
Smallest:	± 2.93873588E – 39

Try typing the following:

```
?1.70141183E+38
```

When you press Return, the same number will display. You have just used the maximum floating-point value for the C-64. Now use the cursor to change the value to

```
?1.70141184E+38
```

which is 1 larger than the largest value permitted. This time when you press Return, the message

```
OVERFLOW ERROR
```

will be displayed. This is the C-64's way of telling you an error has occurred and it cannot accept the value you entered.

Now we can understand the previous results. The number of miles light travels in a light year was

5.8720032E + 12

which we now know to be equivalent to

5872003200000 miles

and the picosecond distance of

1.862E – 07

is

.0000001862 miles

How many feet? inches? centimeters? millimeters?

Do you know about the Googol? It is a 1 followed by 100 zeros.

STRINGS

A string permits BASIC programs to manipulate data that are not used for arithmetic calculations. Strings are enclosed in double quotes ('') and may contain any letter, number, special character, color control symbol, graphic symbol, or cursor-control character. Some strings are

```
"SPACE INVADERS"
"DIFFICULTY LEVEL 1 TO 9"
"__________"
"Want to try again? (Y/N)"
```

Strings may be up to 255 characters in length, which is more than adequate for most applications.

VARIABLES

A variable is a name you give to a memory location capable of storing a value. It might help to think of memory as consisting of a number of small boxes (figure 3.1). Every time you need a place to store a number or name, you put a label on one of the boxes and place your value in the box. Suppose your program requires values to represent speed and complexity. Then one box could be given the variable name S (speed) and the other C (complexity). To place the numbers in the box (memory), you use an assignment statement.

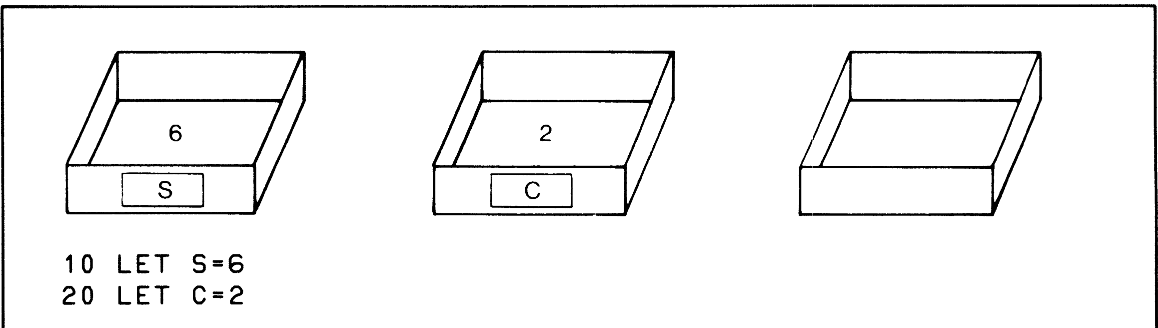


Figure 3.1 Use of variables

Statements 10 and 20 are assignments that give location S a value of 6 and C a value of 2. The C-64 also lets you do this without the keyword LET, as follows:

```
10 S=6
20 C=2
```

Variable names for real numbers may be a single letter, a letter followed by a number, or two letters. Applying these rules gives the following valid names:

A
A5
T
SM
X1

Integers may be defined by placing a percentage sign (%) after the variable name.

A%
D4%
HP%
C%

INTEGER NAMES

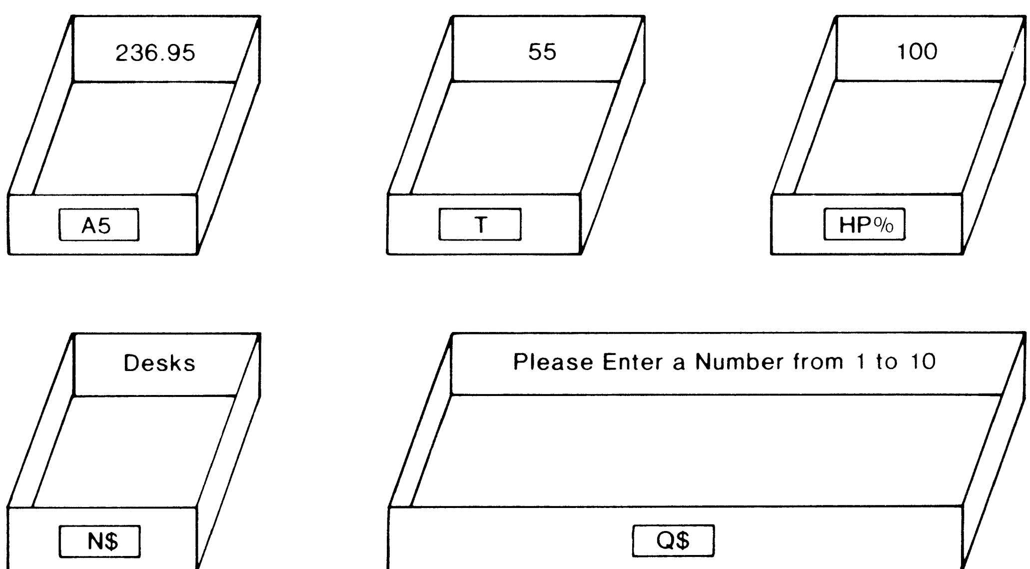
Variable names for strings also follow the above rules except that a string name ends with a dollar sign ($). Thus some valid string names are:

N$
AD$
R9$
Q$

STRING NAMES

Some valid assignments for these names are:

```
10 A5=236.95
20 T=55
30 HP%=100
40 N$="DESKS"
50 Q$="PLEASE ENTER A NUMBER FROM 1 TO 10"
```



When choosing variable names, take care to avoid selecting names that are also reserved words. This problem is particularly prevalent with two-letter variables. When you're unsure of a word, a quick look at Appendix B can tell you if your choice is a reserved word or not.

MULTIPLE STATEMENTS PER LINE

The C-64 has a feature that permits you to enter several commands on one line. This feature is handy when several small steps are required and it seems unnecessary to type several lines. It also saves storage and may be useful to conserve space in longer programs. To place several statements on a line, simply separate each statement with a colon (:).

```
10 N=1:K=25:X=0
```

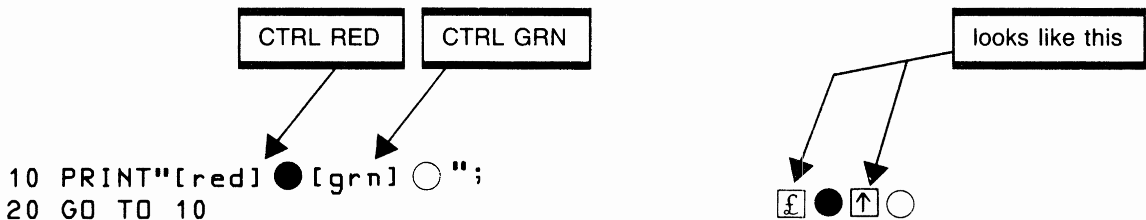
USING COLOR

One of the exciting C-64 features is its color capability. This feature may be included in programs to dress up the results you display on the screen. Try typing the following program:

Now type RUN.

Did you get rows of alternating red and white circles? The C-64 will continue displaying these circles until you stop it. So when you're ready, press the RUN/STOP key.

Now press RUN/STOP and RESTORE together to get back to the normal colors. Type LIST to see your program again. Now try this program code.



Now if the program is RUN, you get alternating red and green circles. Notice that each color-control character has its own special reverse symbol to tell the C-64 what color it represents. Try a few more color combinations and graphics symbols.

Now let's try a special command called POKE. Add the following statement to the previous program by simply typing it and pressing Return.

```
5 POKE 53280,6:POKE 53281,1
```

Now if you LIST the program, statement 5 will be included at the beginning. RUN the program. What happened? If all is well, the screen and border colors will have changed. The screen is now white and the border blue.

POKE is a special BASIC command that tells memory locations 53280 and 53281 the colors you want for the border and background. The following table gives the values the C-64 uses to identify screen and border colors. Restore your program, list it, and try some other combinations.

Border color	POKE 53280,X
Background color	POKE 53281,Y

Values for X and Y

Black	0	Orange	8
White	1	Brown	9
Red	2	Light red	10
Cyan	3	Gray 1	11
Purple	4	Gray 2	12
Green	5	Light green	13
Blue	6	Light blue	14
Yellow	7	Gray 3	15

Table of Screen and Border Colors

ARITHMETIC STATEMENTS

statement number LET variable = { constant / variable / expression }

We have already seen the use of the LET for assigning values to such a variable as

```
10 LET N=25
```

The 10 in this statement is called a statement number. Each statement in BASIC requires a unique statement number except when multiple statements per line are defined. Statement numbers increase in value through the program. The value of the increment is up to the programmer, but a value of 10 or more is advised since this will leave space to insert additional statements if they are eventually required.

The LET statement also permits the calculation of values using the common arithmetic operations as well as a number of mathematical functions. In every case the calculation is specified on the right of the equal sign, and the variable to which the result is assigned is on the left of the equal sign.

Add (+)

Addition is the same as the immediate mode except for the statement number. The plus sign specifies the addition of two numeric values, which may be integer, real, or scientific notation.

```
10 N=25              Result
20 K=14.5
30 J=N+K          J is 39.5
40 N=N+1          N is 26
50 K=K+.25        K is 14.75
60 L=J+20         L is 59.5
70 L=L+N+K        L is 100.25
```

Subtract (–)

Handle subtraction the same way as addition. The only real difference is that the result can sometimes be a negative number. In this case the C-64 automatically retains the sign on the result.

```
10 M=40            Result
10 J=55.5
30 F=J-M         F is 15.5
40 G=M-J         G is -15.5
50 H=J-M-20      H is -4.5
```

Multiply (*)

To multiply, the C-64 uses the asterisk symbol. All the usual rules in mathematics apply.

```
10 A=12            Result
20 B=5.3
30 C=A*B         C is 63.6
40 D=A*-5        D is -60
50 E=A*B*D       E is -3816
```

Divide (/)

Division follows normal rules of mathematics, with the dividend to the left of the slash and the divisor to the right.

```
10 W=48            Result
20 X=12
30 Y=W/X         Y is 4
40 Z=W/-4.8      Z is -10
50 Z=Y/W/.25     Z is 1
```

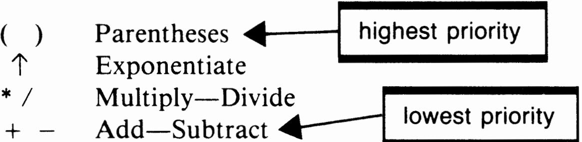
Exponentiation (↑)

Exponentiation is simply raising a number to the power of a second number. This is most common when we square a number or cube it, multiplying the number by itself two or three times respectively. But the C-64 can raise a number to any power, even fractional or negative powers.

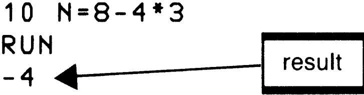
```
10 P=5             Result
20 Q=25
30 R=P↑2         R is 25
40 S=Q↑3         S is 15625
50 T=P↑1.6       T is 13.132639
60 U=Q↑-2.5      U is 3.2E-04
```

HIERARCHY AND PARENTHESES

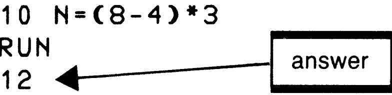
Most arithmetic operations are straightforward, but what happens when there is a mixture of different operations in one statement? When any of the five operators are mixed, BASIC applies a rule of hierarchy to the expression. This rule assigns a certain priority to the operators and defines which goes first.



Normally, arithmetic operations proceed left to right, but when operators are mixed the rules of priority take effect. For instance, the expression



must be evaluated with the multiply operation first. Therefore the correct answer to this calculation is −4, not 12, which would be the case if the C-24 evaluated from left to right. To change the order, parentheses may be used. Since parentheses have the highest priority, an operation within the brackets will be done first.



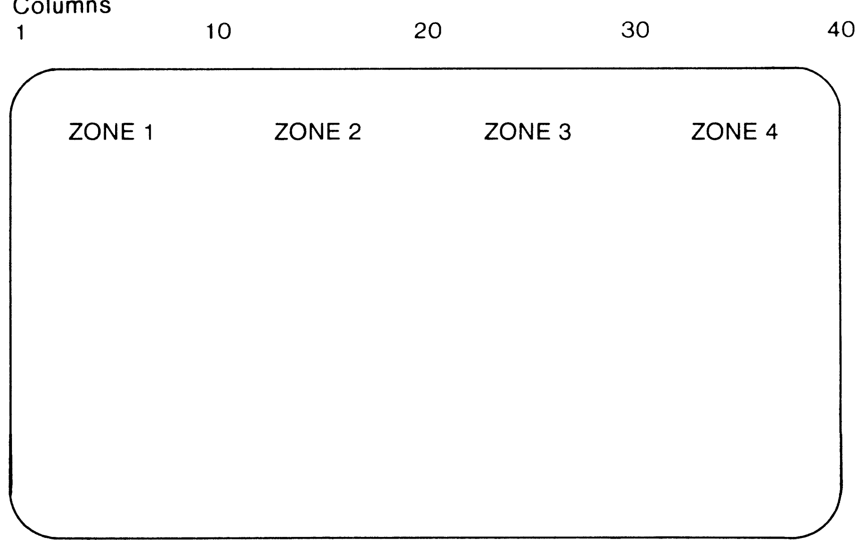
PRINT

statement number PRINT data list

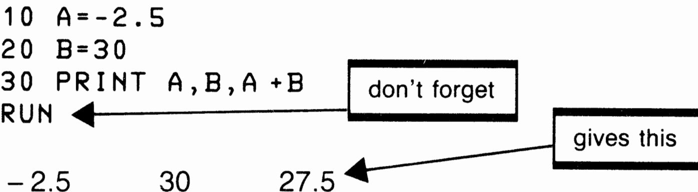
The PRINT statement is used to display information on the screen. Usually the Print displays one line at a time from left to right, 40 characters per line, and from screen top to bottom. When the bottom line (line 25) is reached, all lines are scrolled up by one line and the first line disappears from the top of the screen. The format of the Print is the keyword followed by a list of variables, constants, or expressions separated by commas or semicolons.

```
100 PRINT A,B
200 PRINT N,25
300 PRINT L,"SUM OF A AND B",A+B
```

When the parameters are separated by commas, each value is displayed in a separate 10-position zone on the screen. The C-64 therefore has four zones on one line and can have one value in each of these zones.



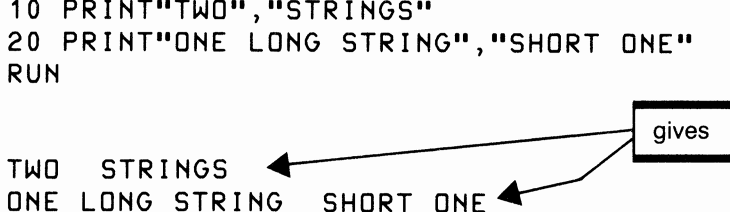
Now try this program:



Each of the three values are allocated 10 spaces. The first of these spaces is either blank or, in the case of a negative number, shows the sign. If a Print has more than four values to print, they will overflow to the next line.

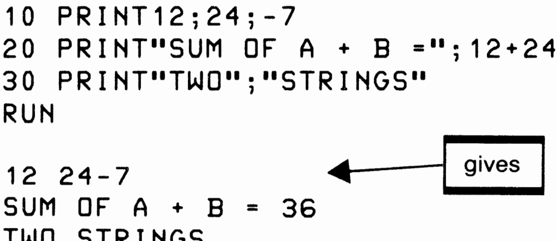
Alphanumeric strings are also given a 10-character zone in which to print.

Now try this program:

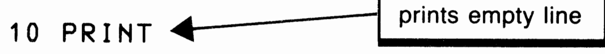


Since strings have no sign, the first character prints in position 1 of the zone. In some cases, such as statement 20, a string may exceed the length of a zone. When this happens, the string continues into the second zone without interruption. A subsequent string or value will start in the next available zone.

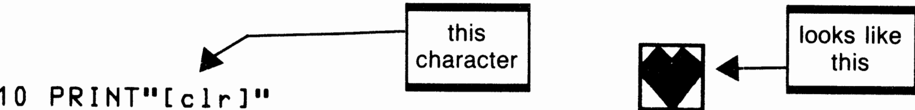
You can achieve closer spacing of values by using the semicolon between values. When a semicolon is used instead of a comma for numeric values, one space is left between each value plus a second space for the sign if the number is negative. Alpha fields have no space when a semicolon is used. Example:



Some additional useful features of the Print are the use of a single Print statement to leave a blank line. This is a command like



with no variables listed. A second useful feature, which will be considered in more depth later, is the machine's ability to place cursor controls in a string. For instance, the statement:



where [clr] represents the shifted CLR control, which causes the screen to be cleared and the cursor brought home before subsequent printing is done. This command is great for clearing extraneous characters from the screen and gives the following output a clear and uncluttered place to display.

Try this program:

```
10 PRINT"CLUTTER THE SCREEN"
20 PRINT"CLUTTER"
30 PRINT"CLUTTER"
40 PRINT"CLUTTER"
50 FOR I=1 TO 1000:NEXT I
60 PRINT"[clr]"
70 PRINT"ON A CLEAR SCREEN YOU CAN SEE YOUR OUTPUT"
RUN
```

slows things down (line 50)

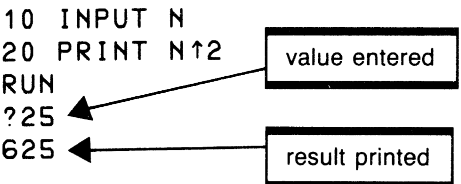
press shift clear (line 60)

INPUT

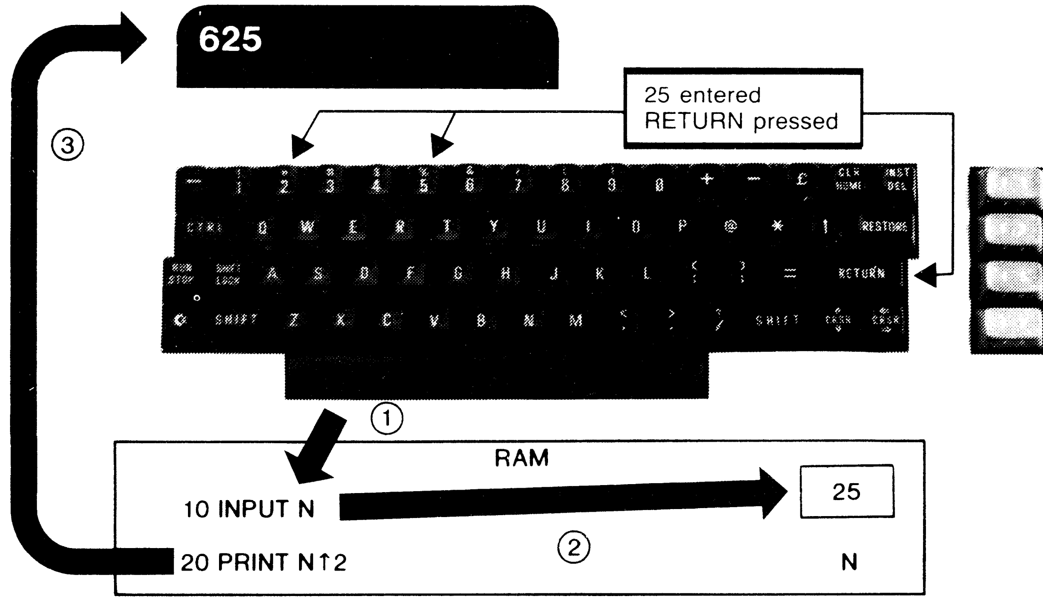
statement number INPUT one or more variables

The INPUT statement is used to let the user supply data for the program to process. When an INPUT is encountered during program execution, the computer waits for the user to type in a value or values and then press Return. When Return is pressed, the values are entered into the program's variables and the program continues with these values.

For example,



When this program is RUN, a question mark appears indicating the program is waiting for a value for N. If 25 is typed and Return pressed, the value 25 is now contained in the variable N and used for the calculation in statement 20.



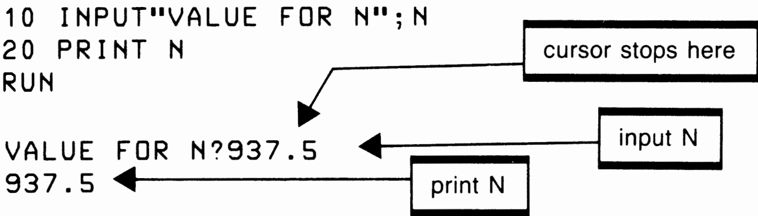
More than one value may be entered by separating them with commas. For instance:

```
10 INPUT A,B
20 PRINT A*B
```

expects two values to be entered, one for A and the second for B. These values must be entered with a comma between them, as follows:

```
?12,25
```

One problem with the above examples is that the user may not know what is expected as input when the program is run. For this purpose, the C-64 will let you use a character string in the INPUT statement. In the first example we could have written




```
10 INPUT"VALUE FOR N";N
20 PRINT N
RUN

VALUE FOR N?937.5
937.5
```

Notice the use of the semicolon in statement 10 to separate the string from the variable N. The semicolon is necessary to avoid a syntax error and it also lets the user type N's value directly following the message that asks for it.

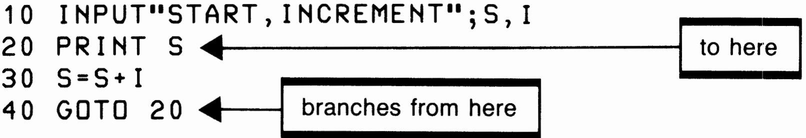
GOTO

statement number GOTO statement number

A GOTO is a branching statement that causes the program to change the flow of execution and continue at the statement number identified in the GOTO. GOTO may be written with or without the space. The statement

```
100 GO TO 20
```

causes the program to branch from statement 100, where it is currently, to statement 20 where the program continues. This statement is called an unconditional branch since branching occurs regardless of what happened up to this point in the program.




```
10 INPUT"START,INCREMENT";S,I
20 PRINT S
30 S=S+I
40 GOTO 20
```

This program will accept a starting value and a value by which S is to be increased. The program proceeds to print S and then increases it by the increment. Statement 40 then branches to 20 where the new value is printed and the process continues. Since a GOTO is unconditional, this program will run forever, well, not quite, since eventually S will run out of space and an illegal quantity error will terminate the program.

REM

```
statement number REM comment
```

The REMark statement simply makes life easier for you as a programmer. It has no effect on your program but prints whatever REM includes within the program. REM is useful for including comments to yourself or anyone else who may read your program. Ideally these comments should make the program more readable and understandable to anyone who reads it. Don't make the mistake of thinking you understand what you have written and don't need to use REMs. It's surprising how much you will forget after a few days or weeks.

```
10 REM DISPLAYS SCORE
40 REM CALCULATES TRAJECTORY
```

SIMPLE CALCULATION PROGRAM

For our first program, let's try something fairly simple. Suppose we want a program that takes two numbers and adds, subtracts, multiplies, and divides them and then prints the results.

Since this is a relatively small program, we might simply sit down at the C-64 and write it. However, to develop good program design habits we should engage in some planning before actually writing the program. An effective tool for this is called English code or Pseudo code. This approach involves writing down a few basic steps constituting a correct general solution to the problem. For instance, we might write:

1. Input values
2. Do calculations
3. Print results

These steps are called the top level of our solution. If the problem is more complex, these steps may be broken down into smaller parts. Step 1 could break down to:

1. Input values
   - 1.1 Input first value
   - 1.2 Input second value

Step 2 could break down into

2. Do calculations
   - 2.1 Calculate sum
   - 2.2 Calculate difference
   - 2.3 Calculate product
   - 2.4 Calculate quotient

Now if we combine all this English code, the solution looks like figure 3.2.

This may seem like a lot of work to prepare for a simple program, but as programs you write become longer and more complex this approach will help you organize and produce much more successful programs. More on this later.

1. Input values
   1.1 Input first value
   1.2 Input second value
2. Do calculations
   2.1 Calculate sum
   2.2 Calculate difference
   2.3 Calculate product
   2.4 Calculate quotient
3. Print results

Figure 3.2 Simple calculation program English code

Now by following the English code, we can write the C-64 BASIC program in figure 3.3. Statements 110 and 120 use INPUT to accept values for A and B, the first and second values. These values are then used in lines 140 to 170 to find the sum, difference, product, and quotient of A and B. Finally, statements 190 to 220 print the answers. For good measure 230 goes back to 110 for a new set of values. We'll discuss looping in English code in later programs. Pay close attention to the method used to print the results, such as:

```
190 PRINT A;"+";B;"=";S
```

The semicolons are used to concatenate, on the output line, the values for A and B with the symbols + and =, followed by the calculated result S. The effect of this statement is an output that looks like an arithmetic expression.

```
100 REM SIMPLE ARITHMETIC CALCULATIONS
110 INPUT "FIRST VALUE";A
120 INPUT "SECOND VALUE";B
130 REM DO CALCULATIONS +,-,*,/
140 S=A+B
150 D=A-B
160 P=A*B
170 Q=A/B
180 REM PRINT RESULTS
190 PRINT A;"+";B;"=";S
200 PRINT A;"-";B;"=";D
210 PRINT A;"*";B;"=";P
220 PRINT A;"/";B;"=";Q
230 GOTO 110
```

Figure 3.3 Simple calculation program

FAHRENHEIT—CELSIUS PROGRAM

This program is a bit more practical but is also easily developed. We would like to input either Fahrenheit or Celsius degrees and have the computer give us the alternative temperature figure. Since we haven't yet learned decision-making statements for the C-64, we will write this as two separate programs combined into one.

The first will be program 100 and will convert Celsius to Fahrenheit. Here is the English code:

Program 100

1. Input Celsius
2. Calculate Fahrenheit
3. Print results

The second is program 200, which converts Fahrenheit to Celsius. Here is its English code:

Program 200

1. Input Fahrenheit
2. Calculate Celsius
3. Print results

The program is shown in figure 3.4. The first part is in statements 100 to 150, converting Celsius to Fahrenheit degrees, the second part, from 200 to 250, converts Fahrenheit to Celsius. To run part 1 you may enter either

```
RUN
```

or

```
RUN 100
```

Either command starts the program at statement 100. The program asks for a Celsius degree, which it then converts to Fahrenheit. This process continues as long as you like. To stop it, press RUN/STOP and RESTORE together.

Now to convert the other way, enter

```
RUN 200
```

which starts the program at statement 200 and proceeds to do the other conversion.

```
100 REM CONVERT CELSIUS TO FAHRENHEIT
110 INPUT"ENTER CELSIUS";C
120 F=C*9/5+32
130 PRINT C; "DEGREES CELSIUS IS";F;"FAHRENHEIT"
140 PRINT
150 GOTO 110
200 REM CONVERT FAHRENHEIT TO CELSIUS"
210 INPUT "ENTER FAHRENHEIT";F
220 C=(F-32)*5/9
230 PRINT F;"DEGREES FAHRENHEIT IS";C;"CELSIUS"
240 PRINT
250 GOTO 210
```

Figure 3.4 Fahrenheit—Celsius program

IMPROVING YOUR SOLUTION

After you have used this program awhile, you realize that some improvements could be made. One problem is, conversions to Celsius often come out as long fractional values. If you look up the INT function in the index, you will discover a way to improve this output, implemented in figure 3.5 in line 220.

INT is a language feature called a function that extracts the integer part of the number or variable contained within its brackets. For example, the expression INT(22.7777778) gives the result of 22. Normally the number is in a variable and appears as follows in the INT function: INT(C). If, in this example, we wanted to round the result, giving 23 instead of the unrounded answer 22, the expression INT(C + 0.5) could be used. An arithmetic expression may also be used in the function, as shown in figure 3.5.

```
RUN 200

ENTER FAHRENHEIT ? 73

73 DEGREES FAHRENHEIT
IS 22.7777778 CELSIUS
```

this would look better if INT were used

```
73 DEGREES FAHRENHEIT
IS 22 CELSIUS
```

A second irritation is that whenever you change from part 1 of the program to part 2, as described earlier, the output from previous operations remains on the screen. This output is cleared in statements 105 and 205, which reflect the use of the CLR control character in the character string of the Print statement as follows:

```
105 PRINT "[clr]"
```

```
100 REM CONVERT CELSIUS TO FAHRENHEIT
105 PRINT"[clr]"
110 INPUT"ENTER CELSIUS";C
120 F=C*9/5+32
130 PRINT C;"DEGREES CELSIUS IS";F;"FAHRENHEIT"
140 PRINT
150 GOTO 110
200 REM CONVERT FAHRENHEIT TO CELSIUS"
205 PRINT"[clr]"
210 INPUT "ENTER FAHRENHEIT";F
220 C=INT((F-32)*5/9)
230 PRINT F;"DEGREES FAHRENHEIT IS";C;"CELSIUS"
240 PRINT
250 GOTO 210
```

Figure 3.5 Improved Fahrenheit—Celsius program

ELEMENTARY GRAPHICS—SAILBOAT

Here is a fun program (figure 3.6) that displays a graphic of a sailboat against an azure blue (well, cyan) sky. The key to entering this program successfully is to make sure all of the spaces, color control characters, and reverse characters are each entered as shown. Notice the use of the POKE to set both the screen and border colors to cyan. This gives a nice background for a blue sailboat with a red sail.

When you get tired of looking at this work of art, press RUN/STOP and RESTORE to get the screen back to normal. Now LIST the program. Your task and challenge is to try to add a bird to the sky above the boat. Use line 18 for this. Use your imagination to add other features to this graphic. How about a flag, or a tiller?

```
10 PRINT "[clr]":PRINT:PRINT:PRINT:PRINT
15 POKE 53280,3:POKE 53281,3
20 PRINT"       [red] [rvs]◢◣ '
30 PRINT"      [red] [rvs]◢     ◣'
40 PRINT"     [red] [rvs] ◢       ◣'
50 PRINT"    [red] [rvs] ◢         ◣'
60 PRINT"          [red] ▌ "
70 PRINT" [blu]◣[rvs]               [off] ◢ "
```

Figure 3.6 Sailboat graphic

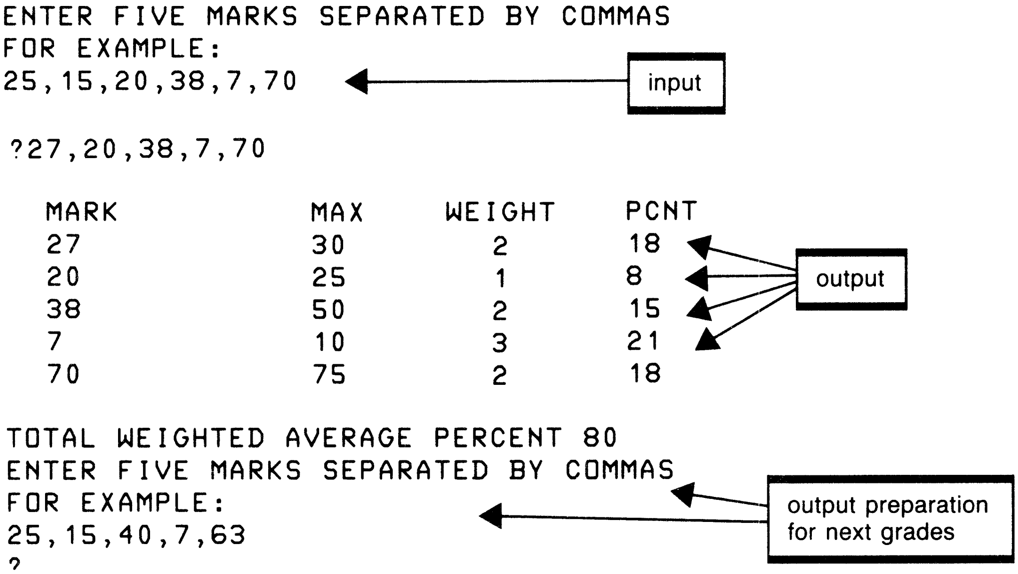
WEIGHTED AVERAGE PROGRAM

A common need in education is to apply different weights to grades depending on the importance of a particular project. Grades themselves may also be given different marking schemes, so that determining a student's final mark becomes a difficult task. This program will make a start at solving this problem, although in a somewhat limited way.

To develop any program, we need a clear mental picture of what we want to accomplish. In the previous programs this has been easy, but real life is not always so. For the weighted average problem, a diagram of what we wish to do will help us to get a grasp of the problem. This is what the picture looks like:

Test	1	2	3	4	5
Maximum Marks	30	25	50	10	75
Weight Factor	2	1	2	3	2

This information gives us some idea about the types of processing we will need in our program. In addition to this, we must define the input expected and output required. The following illustrates our requirements for input and output:



Once we have designed the required input and output and considered the processing requirements, we are ready to develop the English code. Be careful not to do too much at this stage of development. Keep it simple and quite general. In other words, do an overview.

1. Set up initial values
2. Input five marks
3. Calculate percentages
4. Print results
5. Calculate total WA percent

As usual, each necessary step of the English code is developed further. Step 1 sets up each of the maximum values and the weights.

1. Set up initial values
   1.1 Clear screen
   1.2 Set maximum marks
   1.3 Set weights
   1.4 Find total weight

Next we will do step 3, since 2 requires no further elaboration.

3. Calculate percentages
   3.1 Find percent for each of the 5 marks (relative to 100 percent)

Notice, it is not necessary at this time to fully define each statement for the program. We are creating an outline to assist in writing the program once we are satisfied with our solution. Now we are ready to print the results.

4. Print results
   4.1 Print heading
   4.2 Print each of 5 marks and related values

The last step is to calculate the total weighted average percent and print it in step 5.

5. Calculate total WA percentage
   5.1 Find total percentage
   5.2 Print total

Now we are ready to write the program shown in figure 3.7.

First, the program assigns maximums for each test in variables M1-M5. This indicates test 1 had a maximum of 30 marks, test 2 of 25, and so on. Next, line 140 allocates the weights for each test. Weight 2 for test 1, weight 1 for test 2, and so on to weight 2 for test 5.

Statements 160 to 200 accept the marks for one student; 210 to 250 calculate the percentage for each test by applying the appropriate weight. Lines 260 to 350 then print the results, including an overall percentage for the student.

Two of the limitations of this program are the need to do a separate calculation for each percentage, although they are similar, and the need for five Prints for the output. Another problem will surface if marks exceeding the maximum for that test are entered. This program will cheerfully accept such a mark with no arguments. These problems will be considered in an improved program in the next chapter.

```
100 REM WEIGHTED AVERAGE
105 PRINT "[clr]"
110 REM MAXIMUM MARK FOR EACH TEST
120 M1=30:M2=25:M3=50:M4=10:M5=75
130 REM WEIGHTING FOR EACH TEST
140 W1=2: W2=1: W3=2: W4=3: W5=2
150 TW=W1+W2+W3+W4+W5
160 PRINT "ENTER FIVE MARKS SEPARATED BY COMMAS"
170 PRINT "FOR EXAMPLE:"
180 PRINT "25,15,40,7,63"
190 PRINT
200 INPUT MA,MB,MC,MD,ME
210 P1=INT((MA/M1*100)*W1/TW)
220 P2=INT((MB/M2*100)*W2/TW)
230 P3=INT((MC/M3*100)*W3/TW)
240 P4=INT((MD/M4*100)*W4/TW)
250 P5=INT((ME/M5*100)*W5/TW)
260 PRINT "[clr]MARK ","MAX ","WEIGHT ","PCTG"
270 PRINT
280 PRINT MA,M1,W1,P1
290 PRINT MB,M2,W2,P2
300 PRINT MC,M3,W3,P3
310 PRINT MD,M4,W4,P4
320 PRINT ME,M5,W5,P5
330 PT=P1+P2+P3+P4+P5
340 PRINT
350 PRINT "TOTAL WEIGHTED AVERAGE PERCENT";PT
360 GOTO 160
```

Figure 3.7 Weighted average program

COMPUTING LOAN PAYMENTS

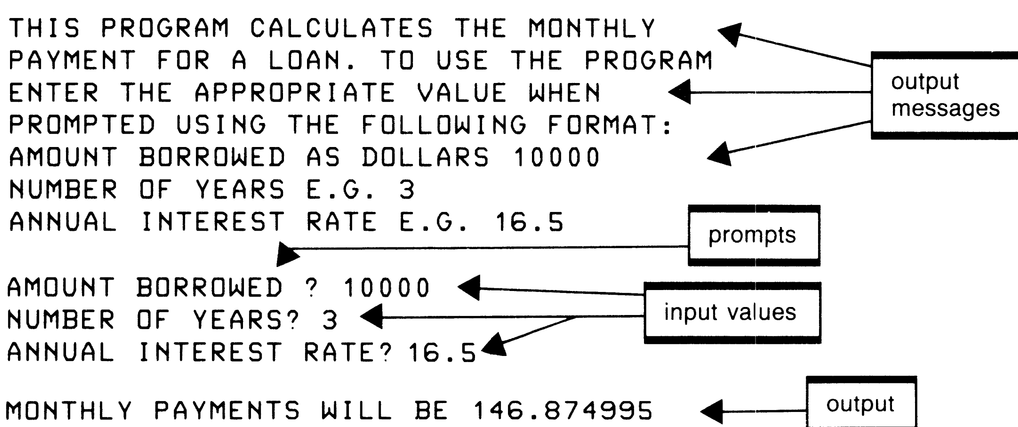
Who among us doesn't need to take out a loan on occasion for a new car, an appliance, or a vacation? With today's high interest rates, a small difference in the percentage can make a large difference in monthly payments. We may also want to try different repayment terms to evaluate our ability to pay each month. Even with our limited knowledge of BASIC at this stage, we can still write a useful program.

First, we need a formula to calculate the payment given the other variables. The periodic rent of annuity formula can be found in business math or accounting books. It is as follows:

$$p = a\left(\frac{i}{1-(1+i)^{-n}}\right)$$

where a is the amount borrowed
i is the interest rate
n is the number of periods
p is the payment

The approach taken in this program is initially to print some instructions about the program's use. Included in these instructions are some sample data to clarify what the program expects from the user. This method is particularly important here since an entry such as interest rate could be entered several different ways: as .165, 16.5, or 16.5%; but only 16.5 would give the correct results. Here is a sample of what we want the input and output to look like:



Since the formula needs the interest as a fraction for the period of the loan, we will need to divide the input by 100, obtaining the fraction, and then divide again by 12 to convert to a monthly rate. Since personal loans are normally taken out for annual periods (1, 2, 3 years), that is how the term will be read, although a 2 1/2-year loan could be entered as 2.5. We will then convert this value into months, which is necessary for the formula.

Now we are prepared to develop the English code.

1. Print instruction messages
2. Input amounts required
   2.1 Input amount borrowed
   2.2 Input number of years
   2.3 Input interest rate
3. Convert to internal amounts
   3.1 Multiply years × 12 giving months
   3.2 Divide interest/100/12 (converts to monthly fraction)
4. Calculate payment
5. Print payment

To reiterate, the principle in problem solving is to define in general terms your solution and then to break down these components until you have completely solved the problem. Now by following each step defined above, the program is written in figure 3.8.

```
100 REM CALCULATING MONTHLY LOAN PAYMENTS
110 PRINT "THIS PROGRAM CALCULATES THE MONTHLY"
120 PRINT "PAYMENT FOR A LOAN. TO USE THE PROGRAM"
130 PRINT "ENTER THE APPROPRIATE VALUE WHEN"
140 PRINT "PROMPTED USING THE FOLLOWING FORMAT:"
150 PRINT "AMOUNT BORROWED AS DOLLARS 10000"
160 PRINT "NUMBER OF YEARS E.G.      3"
170 PRINT "ANNUAL INTEREST RATE E.G.  16.5"
180 PRINT
190 INPUT "AMOUNT BORROWED";A
200 INPUT "NUMBER OF YEARS";N
210 INPUT "ANNUAL INTEREST RATE";I
220 N=N*12        :REM CONVERT TO MONTHS
230 I=(I/100)/12 :REM % PER MONTH
240 P=A*(I/(1-(1+I)↑-N))
250 PRINT
260 PRINT "MONTHLY PAYMENTS WILL BE";P
```

Figure 3.8 Loan payments program

REVIEW QUESTIONS—CHAPTER 3

1. What does calculator or immediate mode mean?
2. How are the four cursor controls used? Explain how to correct the line ?435*18, which should have been a divide (/) instead of multiply (*), when the cursor is on the first position of the next line.
3. Explain how to correct the above command when the 435 should have been 4.35 and the cursor is already on the same line.
4. Describe the purpose of the CLR/HOME key.
5. Discuss two kinds of numbers used on the Commodore-64.
6. Explain the difference between rounding and truncating.
7. What is a number such as 5.8720032E + 12 called? What is the equivalent decimal value?
8. What is a variable? Why is it used in programming? What are the rules for valid variable names?
9. Explain how data may be assigned to a variable.
10. Show how to use the color and reverse keys when creating graphic images with the PRINT statement.
11. Explain the difference between using commas to separate variables in a PRINT statement and using semicolons.
12. How can you clear the screen from within a BASIC program?
13. What is the purpose of the INPUT statement? Describe what happens when a statement like 10 INPUT "RADIUS";R is used in a program.

4 Not So Basic BASIC

The previous chapter introduced some of the elements of BASIC, enough of them to let us write some fairly useful programs. But really to take advantage of the C-64's capability, you must understand statements that permit decision making, looping, data structuring in arrays, and the use of subroutines for program organization and efficiency. These additional features of the language will broaden our horizons considerably and give us the ability to solve a wider range of interesting problems.

DIM

> statement number DIM array name (constant), array name (constant), . . .

The DIM, or Dimension statement, permits us to store multiple data values under the same variable name. For example, in the last chapter we wrote a program to do weighted averages, which used 5 variables M1 to M5 for maximum marks, 5 more W1 to W5 for the weight and MA to ME for the marks. Using DIM, these 15 variables could be reduced to 3, or with a little ingenuity to a single variable.

Variables defined with DIM are called arrays, and may each have a number of elements or positions in which to store data. The variables for weighted average could be described in BASIC as follows:

```
10 DIM M(5),W(5),MA(5)
```

The number 5 in parentheses defines the number of elements in each array. Each element of the array may be referenced by using a subscript on the array name. For instance, position 3 of the array for maximum marks would be referenced in the program by specifying M(3). Arrays are particularly useful since the subscript can also be a variable or an expression.

This feature permits array references such as

```
50 PRINT MA(I)
60 P=MA(N)/M(N)*100
70 H=W(K-1)
```

The arrays we specified in statement 10 can be visualized as shown in figure 4.1. Actually the C-64 allocates 0 as an element, so there are really six elements in each array. For convenience we will usually ignore the zero element, although at times it may be useful.

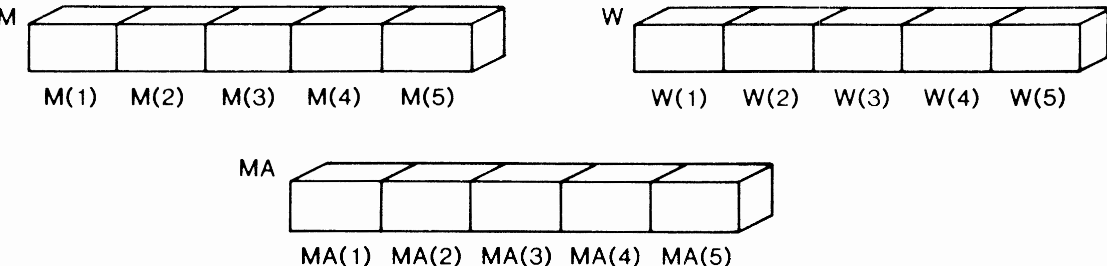


Figure 4.1 Visualizing a one-dimensional array

Storing Values in an Array

If we wished to store ten numbers in an array, an array of ten elements would be defined by a DIM statement. The following program sets up such an array, reads ten numbers into the array, and then displays the contents of the array. The ten numbers are entered in directly from the keyboard, one value at a time.

```
10 DIM K(10)
20 REM LOAD ARRAY WITH NUMBERS
30 I=1
40 INPUT "NUMBER";K(I)
50 I=I+1
60 IF I<=10 THEN 40
70 REM PRINT THE NUMBERS
80 I=1
90 PRINT"NUMBER";I;" = ";K(I)
100 I=I+1
110 IF I<=10 THEN 90
120 PRINT "DONE"
130 END
```

If I is less than or equal to 10 then go to statement 40

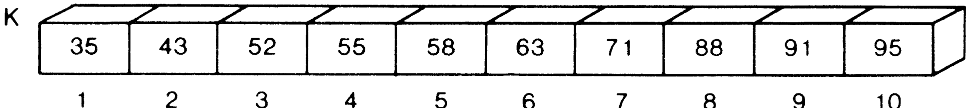
otherwise come here

The IF statement at line 60 is used to determine when 10 numbers have been placed into the array. The second IF statement in line 110 determines when all 10 numbers have been printed. Don't worry if this statement is not quite clear at this point in your reading. The IF will be explained in detail later in this chapter.

When this program is run, a prompt is given for each of the 10 numbers to be entered into the array. These entries would appear on the screen as follows:

```
NUMBER ? 35
NUMBER ? 43
NUMBER ? 52
NUMBER ? 55
NUMBER ? 58
NUMBER ? 63
NUMBER ? 71
NUMBER ? 88
NUMBER ? 91
NUMBER ? 95
```

When the program reaches statement 70, the contents of array K will appear as follows:



Statements 70 to 120 now display these contents to confirm that the numbers are actually in the array. Here is the output.

```
NUMBER 1 = 35
NUMBER 2 = 43
NUMBER 3 = 52
NUMBER 4 = 55
NUMBER 5 = 58
NUMBER 6 = 63
NUMBER 7 = 71
NUMBER 8 = 88
NUMBER 9 = 91
NUMBER 10 = 95
DONE
```

Arrays with one dimension, such as those we have used here, are the most common type in BASIC programs, and we will have occasion to use them frequently in other programs in this and subsequent chapters. But first let's look at arrays that have two or more dimensions.

Multidimensional Arrays

Arrays may be multidimensional, although more than two or three dimensions are rarely useful. Specifications in BASIC, such as:

```
10 DIM A(5,4),B(25,3)
```

are two-dimensional arrays, while

```
10 DIM C(3,5,4),D(2,5,20)
```

are three dimensional.

For example, the first array defined above (A(5,4)) refers to a two-dimensional array with 5 elements by 4 elements. This array can be visualized as a rectangular shape containing five rows and four columns as shown in figure 4.2. Of course, in the C-64's memory the array does not actually exist in this physical shape, but thinking of a two-dimensional array in this way helps us write programs that need this kind of array.

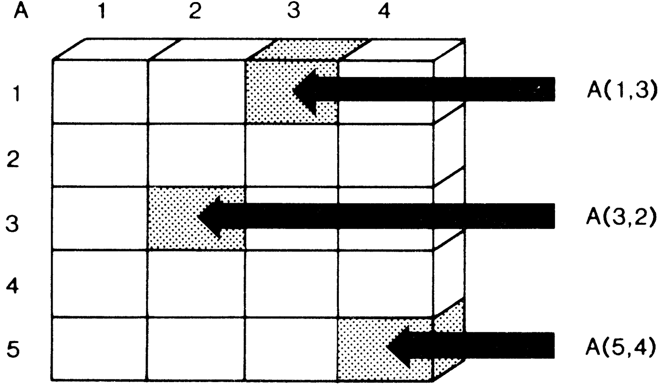
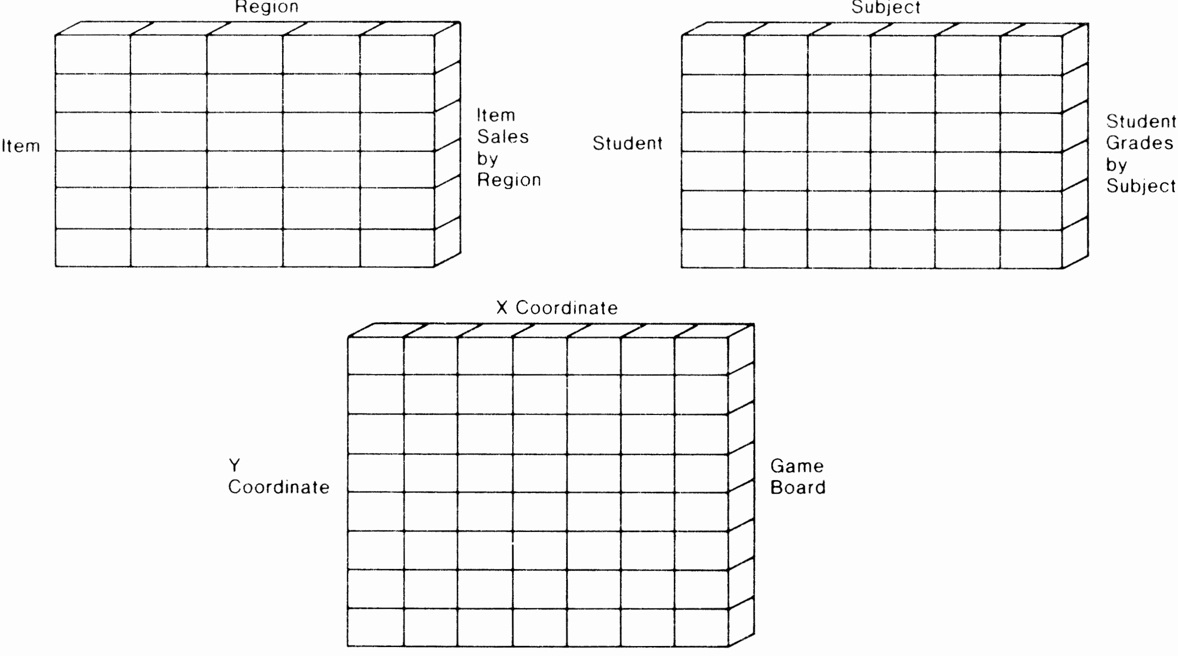


Figure 4.2 A two-dimensional array A(5,4)

Two-dimensional arrays can be used to represent many types of data. Here are just a few examples:



IF—THEN

statement number IF condition { THEN action / GOTO statement number }

The GOTO was an unconditional branch, but when it is used with an IF statement, conditional branching may be used. The IF also permits the programmer to select from one or more operations, depending on the needs of the program. Generally the IF statement examines a condition, and if the condition evaluates true it does the action specified. If the condition is false, the program merely continues at the next statement. Figure 4.3 identifies the operators available for decision making.

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

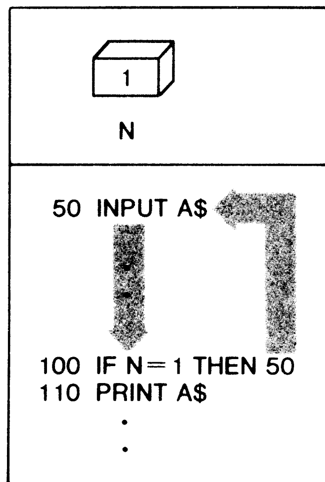
Figure 4.3 Logical operators

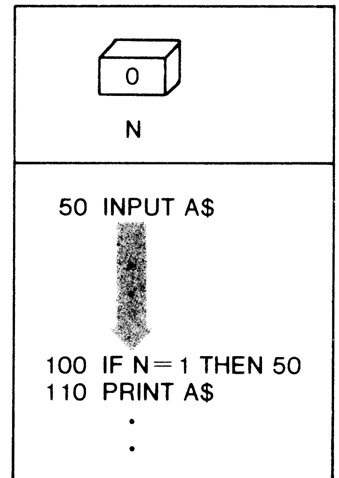
Some possible IF statements are:

```
100 IF N=1 THEN 50
200 IF K>245 GOTO 50
300 IF R$="YES" THEN 5000
400 IF W(I)<=M(K) THEN 100
```

Notice that the THEN or GOTO may be used interchangeably and can cause branching either forward or backward within the program. Either numeric or alphanumeric variables may be compared in the IF statement, although they may not be compared to each other.

In line 100 above, if the value of N is 1, when statement 100 is reached, branching will go to statement 50 where the program will continue. If N were any value other than 1, the program would continue at the next statement following statement 100.





Actions do not have to be GOTOs. They may be almost any other BASIC statement. Note the following examples:

```
100 IF K=L THEN PRINT K,J
200 IF N$="JONES"THEN C=C+1
300 IF Q=1 THEN INPUT "PLEASE ENTER NAME";N$
```

An additional feature of C-64 BASIC is the ability to specify more than one action to be taken when a condition is true. Each action in the IF statement is separated from the other with a colon (:).

```
100 IF N=100 THEN N=N+1:PRINT S:GOTO 10
200 IF M$(J)="JULY"THEN A$=M$(J):T=T+A:K=0
```

Using this feature can reduce a lot of unnecessary branching, thus making your programs easier to follow. Statements written this way also execute faster than if they were individual statements with individual line numbers. A possible disadvantage to the use of multiple statements is that program changes and maintenance can become more difficult.

Note the use of M$(J) in statement 200. This example indicates that a string variable may also be used as an array. The only difference between a numeric and a string array is that the contents of the string array will be string data.

The operators AND and OR (figure 4.4) — called Boolean operators after George Boole, a mathematician — may also be used to combine several conditions in one IF statement. For instance,

if you wanted to branch to statement 20 when a count had reached 100 or a code of 3 was found in an array D, the following statement could be used.

```
200 IF C=100 or D(I)=3 THEN 20
```

The OR evaluates true when either one condition or the other or both are true. AND evaluates true only when both conditions are true.

IF condition 1 {AND / OR} condition 2 THEN action			
Resulting Boolean Operator Value		condition-1	condition-2
AND	OR		
True	True	True	True
False	True	True	False
False	True	False	True
False	False	False	False

Figure 4.4 Boolean operators

FOR—NEXT

```
statement number FOR variable = value1 TO value2 STEP value3
        .
        .
        .
statement number NEXT variable
```

The FOR and NEXT statements are used to provide for looping in a program. Although looping may be accomplished with a combination of arithmetic, IF, and GOTO statements, the FOR—NEXT provides a simpler way of achieving the same thing.

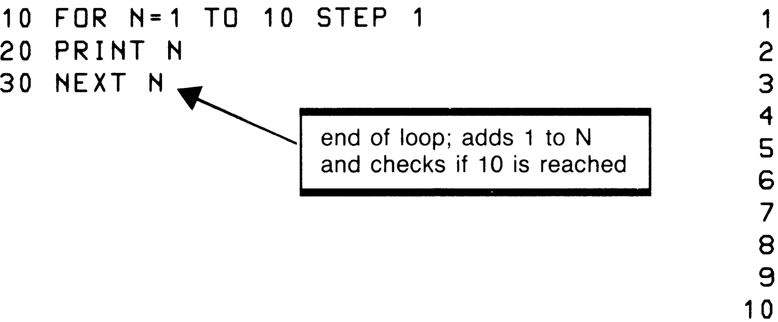
For example, if we wanted to print out the numbers from 1 to 10, we could write the program:

```
10 N=0
20 N=N+1
30 PRINT N
40 IF N<10 GOTO 20
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

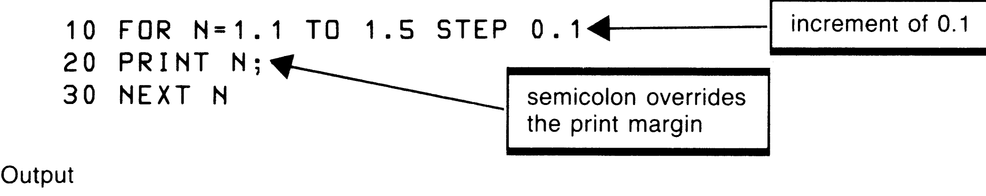
Using the FOR—NEXT, the previous program appears as follows:



The FOR statement defines the variable to be used (N) for the loop, its starting value, and the amount to increment (STEP) N when the end of the loop (NEXT) is reached. Any number of statements may appear inside the loop, including another FOR—NEXT pair. When the increment is 1, as in this case, the STEP may be deleted, giving the same effect.

```
10 FOR N=1 to 10
```

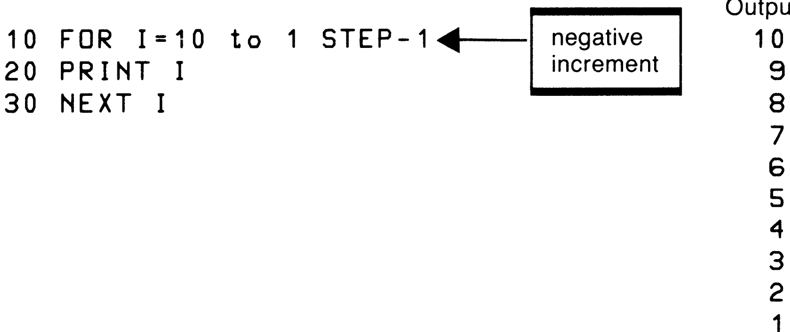
Variables used by the FOR must be numeric but can be fractional if required. The following statements will print the fractional values from 1.1 to 1.5 across one line of the screen.



Output

```
1.1 1.2 1.3 1.4 1.5
```

The FOR may also use negative values for its operation. Often negative amounts may appear in the STEP value although they could be used for starting or ending values as well.



GOSUB—RETURN

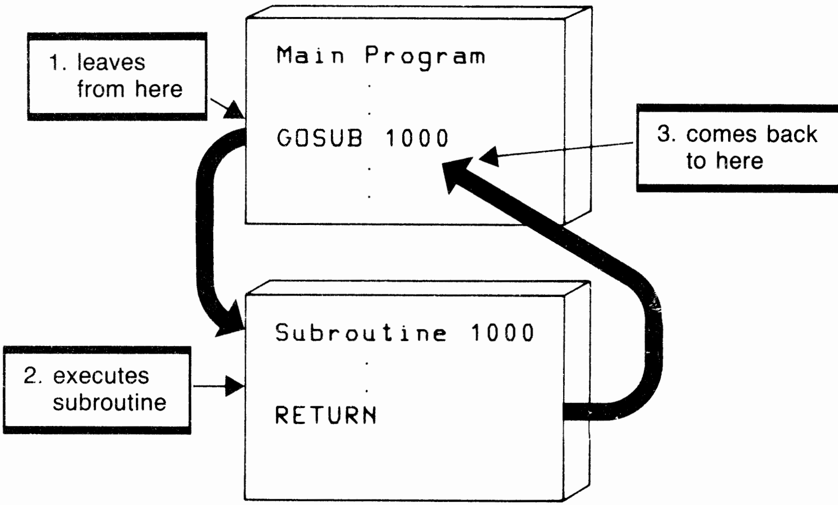
statement number GOSUB statement number

.
.
.

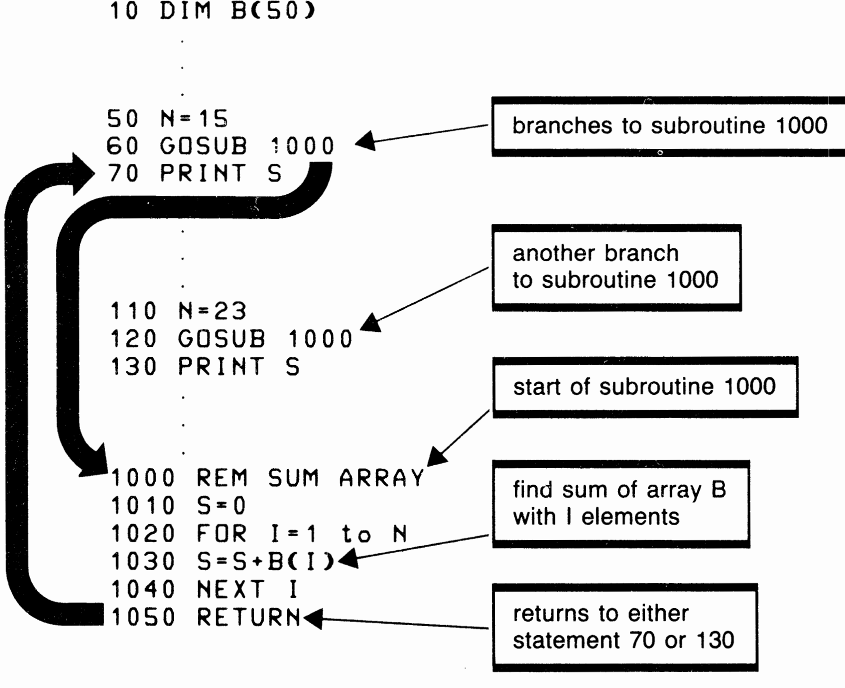
statement number RETURN

GOSUB acts much like the GOTO except the C-64 remembers where it came from and can then RETURN to the statement following the GOSUB in the program. GOSUB means go to a set of BASIC statements written to perform a specific action, with the expectation that when the action is completed, normal program execution will resume at the statement following the GOSUB statement. Such a set of BASIC statements written to perform a specific action with the capability of returning is called a subroutine. For instance, a subroutine may be written to accept the input from a user, to plot a graph on the screen, or to check for errors after a disk operation.

At the end of the subroutine, a RETURN statement tells the computer to revert to the location following the GOSUB in the main program. A nice thing about GOSUB is it may go to a subroutine from many different places in the same program. This is useful since program code common to different parts of the program may be written once as a subroutine but used as often as needed.



Suppose subroutine 1000 is written to find the sum of values previously stored in an array. The variable N defines how many values are in the array at any one time, and the sum is created in S by the subroutine itself.



STOP

statement number STOP

The STOP command may be used to terminate a program when the program has finished its processing. STOP may be used as an independent statement as in

```
200 STOP
```

which terminates after a sequence of program steps have been completed. Or it may be used in a decision

```
150 IF A$="NO" THEN STOP
```

for examining a user's response to such a question as "DO YOU WANT TO CONTINUE," as in the following sequence of statements:

```
140 INPUT"DO YOU WANT TO CONTINUE";A$
150 IF A$="NO" THEN STOP
160 GOTO 10
```

GENERATING RANDOM NUMBERS

Some subroutines, known as functions, are already built into the C-64's ROM and can be directly accessed by the program. The first of these we will use is the RND function, which is needed to generate random numbers. Normally RND produces fractional random numbers such as .974626516 and not integers.

Try these statements on your system:

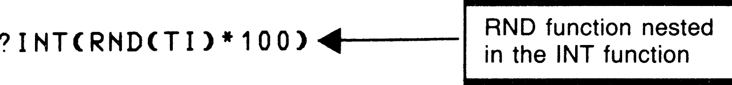
```
?RND(1)
 .974626516

?RND(1)
 .747932106

?RND(TI)
 .435687146
```

The resulting numbers given here will be different from yours since a random number is given each time. The use of TI uses the value of the timer to initiate action for the random number (TI is described fully in chapter 6).

In addition to being fractional, these numbers are not within the specific range required. In the game that follows, we want only numbers in the range of 1 to 100 and these numbers must not be fractional. To limit the range to numbers no greater than 100, we can multiply the random number by 100.

```
?RND(TI)*100
```

If the random number had been .974626516, this expression would transform the number into the value 97.4626516 since multiplying has shifted the decimal point two places to the right. Now to convert to an integer, the INT function comes in handy.



The expression we now have will generate the random number and take only the integer part of it, thus giving us 97 or some other 2-digit integer. At this point, if we try a number of values, we find that we have created values from 0 to 99 rather than 1 to 100, so we simply add 1 to the result, giving the values required for our game. A statement like

```
710 N=INT(RND(TI)*100)+1
```

will be used in the program to generate a random 2-digit number from 1 to 100 and store that number in the variable N.

The number we started with would now become 98 and be the first random number to be stored in N. The next might be .428313721, which would result in the number 43 in variable N.

NUMBER GUESSING GAME

Part of the enjoyment derived from having a C-64 computer is playing the numerous games available. This availability, however, should not discourage you from developing your own games, and an example of how a game program may be written follows. Although the program is quite short, for a game, it does have many of the characteristics of a good game. These qualities are user instructions, interaction with the player, a certain degree of randomness, and the capability to repeat the game if the user desires. Figure 4.5 shows the English code.

Number Guessing Game—Mainline
1. Play game until done
    1.1 Print instructions subroutine
    1.2 Generate number subroutine
    1.3 Accept guess subroutine

Print instructions—subroutine
1. If instructions not wanted then return
2. Display instructions
3. Return

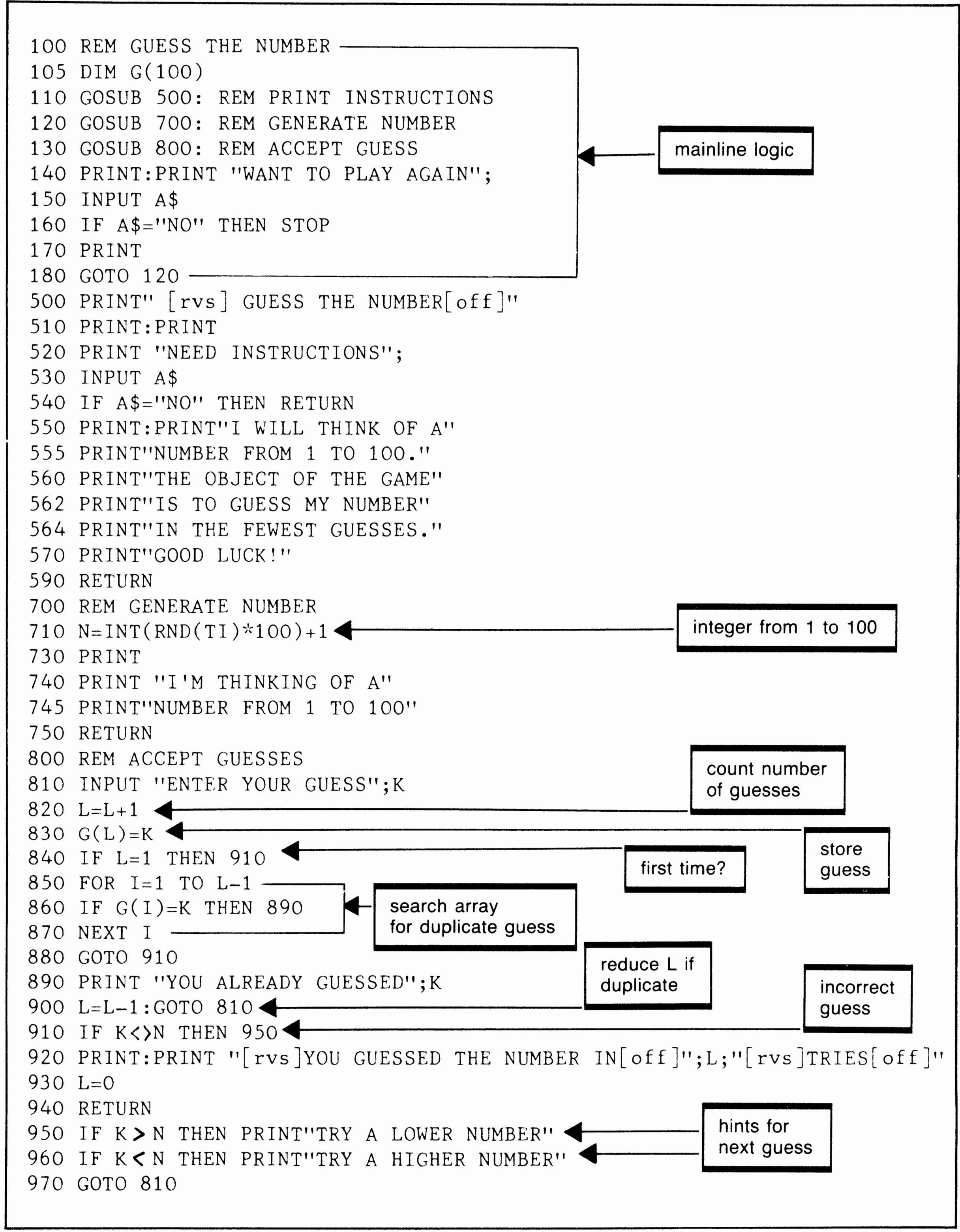
Generate number—subroutine
1. Generate random number
2. Print prompt
3. Return

Accept guess—subroutine
1. Accept guesses until correct answer
    1.1 Accept guess
    1.2 Check answer
    1.3 If wrong give a hint
2. Display number of tries
3. Set number of guesses to 0
4. Return

Figure 4.5 Number guessing game English code

This solution shows how to use subroutines effectively for good program organization. These subroutines Print Player Instructions, Generate the Random Number, and Accept the Player's Guess. The program statements from 100 to 180 control these subroutines and determine if the player wishes another game.

Several new features of BASIC are used in the program in addition to variations on features already discussed up to this point.



```
100 REM GUESS THE NUMBER
105 DIM G(100)
110 GOSUB 500: REM PRINT INSTRUCTIONS
120 GOSUB 700: REM GENERATE NUMBER
130 GOSUB 800: REM ACCEPT GUESS
140 PRINT:PRINT "WANT TO PLAY AGAIN";
150 INPUT A$
160 IF A$="NO" THEN STOP
170 PRINT
180 GOTO 120
500 PRINT" [rvs] GUESS THE NUMBER[off]"
510 PRINT:PRINT
520 PRINT "NEED INSTRUCTIONS";
530 INPUT A$
540 IF A$="NO" THEN RETURN
550 PRINT:PRINT"I WILL THINK OF A"
555 PRINT"NUMBER FROM 1 TO 100."
560 PRINT"THE OBJECT OF THE GAME"
562 PRINT"IS TO GUESS MY NUMBER"
564 PRINT"IN THE FEWEST GUESSES."
570 PRINT"GOOD LUCK!"
590 RETURN
700 REM GENERATE NUMBER
710 N=INT(RND(TI)*100)+1
730 PRINT
740 PRINT "I'M THINKING OF A"
745 PRINT"NUMBER FROM 1 TO 100"
750 RETURN
800 REM ACCEPT GUESSES
810 INPUT "ENTER YOUR GUESS";K
820 L=L+1
830 G(L)=K
840 IF L=1 THEN 910
850 FOR I=1 TO L-1
860 IF G(I)=K THEN 890
870 NEXT I
880 GOTO 910
890 PRINT "YOU ALREADY GUESSED";K
900 L=L-1:GOTO 810
910 IF K<>N THEN 950
920 PRINT:PRINT "[rvs]YOU GUESSED THE NUMBER IN[off]";L;"[rvs]TRIES[off]"
930 L=0
940 RETURN
950 IF K>N THEN PRINT"TRY A LOWER NUMBER"
960 IF K<N THEN PRINT"TRY A HIGHER NUMBER"
970 GOTO 810
```


Figure 4.6 Number guessing game

TIME DELAYS

After you have played the number guessing game a few times, you may wish the computer wasn't so fast. In fact, the speed at which the computer operates can be a psychological disadvantage since the user will feel constant pressure to perform even when there is no need to hurry.

The use of a time-delay program code to slow down the action of the computer while the player takes some action is a useful tool in this program. Create the delay by including a FOR—NEXT loop, which simply takes up some processing time before moving on to the next statement. The larger the value in the loop, the longer the delay. This use of the FOR—NEXT shows the optional use of the NEXT statement without the variable I. The loop

```
580 FOR I=1 TO 3000:NEXT
```

will occupy about three seconds of computer time. Its purpose in this program is to give the player some time to read the instructions before having to guess a number. Although the program could be left without a delay, there is a psychological benefit from not creating undue pressure on the player. Of course in some situations time pressure may be an integral part of the game.

Try these two delays in the previous program and see if they don't improve the operation considerably.

```
580 FOR I=1 TO 3000:NEXT
720 FOR I=1 500:NEXT
```

Statement 580 gives a 3-second delay after the instructions have been displayed. If 3 seconds is too short, then a larger value like 5,000 or more could be tried.

Statement 720 gives the appearance of the computer taking its time to determine a random number. The user will need to wait during this delay, but only for about half a second. This has the effect of making the action more comfortable without actually creating a noticeable delay. Figure 4.7 contains the revised program with time delays included.

```
100 REM GUESS THE NUMBER
105 DIM G(100)
110 GOSUB 500: REM PRINT INSTRUCTIONS
120 GOSUB 700: REM GENERATE NUMBER
130 GOSUB 800: REM ACCEPT GUESS
140 PRINT:PRINT "WANT TO PLAY AGAIN";
150 INPUT A$
160 IF A$="NO" THEN STOP
170 PRINT
180 GOTO 120
500 PRINT"  [rvs]GUESS THE NUMBER[off]"
510 PRINT:PRINT
520 PRINT "NEED INSTRUCTIONS";
530 INPUT A$
540 IF A$="NO" THEN RETURN
550 PRINT:PRINT"I WILL THINK OF A"
555 PRINT"NUMBER FROM 1 TO 100."
560 PRINT"THE OBJECT OF THE GAME"
562 PRINT"IS TO GUESS MY NUMBER"
564 PRINT"IN THE FEWEST GUESSES."
570 PRINT"GOOD LUCK!"
580 FOR I=1 TO 3000:NEXT  <---- 3-second delay
590 RETURN
700 REM GENERATE NUMBER
710 N=INT(RND(TI)*100)+1
720 FOR I=1 TO 500:NEXT  <---- 1/2-second delay
730 PRINT
740 PRINT "I'M THINKING OF A"
745 PRINT"NUMBER FROM 1 TO 100"
750 RETURN
800 REM ACCEPT GUESSES
810 INPUT "ENTER YOUR GUESS";K
820 L=L+1
830 G(L)=K
840 IF L=1 THEN 910
850 FOR I=1 TO L-1
860 IF G(I)=K THEN 890
870 NEXT I
880 GOTO 910
890 PRINT "YOU ALREADY GUESSED";K
900 L=L-1:GOTO 810
910 IF K<>N THEN 950
920 PRINT:PRINT "[rvs]YOU GUESSED THE NUMBER IN[off]";L;"[rvs]TRIES[off]"
930 L=0
940 RETURN
950 IF K > N THEN PRINT"TRY A LOWER NUMBER"
960 IF K< N THEN PRINT"TRY A HIGHER NUMBER"
970 GOTO 810
```

Figure 4.7 Revised number guessing game

IMPROVED WEIGHTED AVERAGE PROGRAM

Now that we know about arrays, let's take another look at the weighted-average program from the previous chapter. Since there were always five marks to be processed with five weights and five maximums, each of these could become an array.

In addition to using arrays, we would like to permit the user to enter marks for another student or terminate the program. This places most of the program logic into one large loop that is continually repeated until all students have been processed. The following English code implements the desired changes.

1. Set up initial values
   1.1 Set maximum marks
   1.2 Set weights
   1.3 Find total weight

2. Do weighted average until no more students
   2.1 Input five marks
      2.1.1 Input mark (i)
      2.1.2 Check for maximum allowed
   2.2 Calculate percents and print
      2.2.1 Find percent for each of the 5 marks (relative to 100%)
      2.2.2 Print each of 5 marks and related values
      2.2.3 Compute total weighted percent
   2.3 Print weighted total
   2.4 Accept another student (Yes/No)?

The arrays are defined as M(5), W(5), and MA(5) respectively. The program in figure 4.8 shows these arrays and the assignment of the maximum and the weight for each test. Now you can make any reference to the arrays by using a FOR loop and a subscript.

A second improvement to the program prints the maximum mark allowed when asking for input. For instance, when the second mark is required, the program prints:

```
MARK 2 (MAX25)?
```

This request is printed from statement 210 where I provides the mark reference number and M(I) gives the maximum for the second mark.

A final improvement, unrelated to arrays, allows the user to enter data for another student or terminate the program, defined by the major loop at step 2 in the English code.

One additional advantage of arrays is the ease with which the program may be changed to accept more or even less data per student. This change is made by simply changing the DIM and each FOR loop. Of course the maximums and weights would also be changed accordingly. You might want to consider generalizing this program so that any number of marks could be entered, depending on the user's needs.

```
100 REM WEIGHTED AVERAGE
110 DIM M(5),W(5),MA(5)
120 REM MAXIMUM MARK FOR EACH TEST
130 M(1)=30:M(2)=25:M(3)=50:M(4)=10:M(5)=75    <---- set data in arrays
140 REM WEIGHTING FOR EACH TEST
150 W(1)=2:W(2)=1:W(3)=2:W(4)=3:W(5)=2    <---- set data in arrays
160 FOR I=1 TO 5
170 TW=TW+W(I)
180 NEXT I
190 PRINT "[clr]"
200 FOR I=1 TO 5
210 PRINT " MARK";I;"( MAX";M(I);")";    <---- prompt
220 INPUT MA(I)
225 IF MA(I)>M(I)THEN PRINT "EXCEEDS MAXIMUM":GOTO 210
230 NEXT I
240 PRINT "[clr]MARK","MAX","WEIGHT","PERCENT"
250 PRINT
260 REM COMPUTE PERCENTS AND TOTAL
270 FOR I=1 TO 5
280 P=INT((MA(I)/M(I)*100)*W(I)/TW)
290 PRINT MA(I),M(I),W(I),P
300 PT=PT+P
310 NEXT I
320 PRINT
330 PRINT "TOTAL WEIGHTED AVERAGE PERCENT";PT
340 PT=0
350 PRINT
360 PRINT
370 INPUT "TYPE (Y) FOR ANOTHER STUDENT";A$
380 IF A$="Y" THEN 190
390 STOP
```

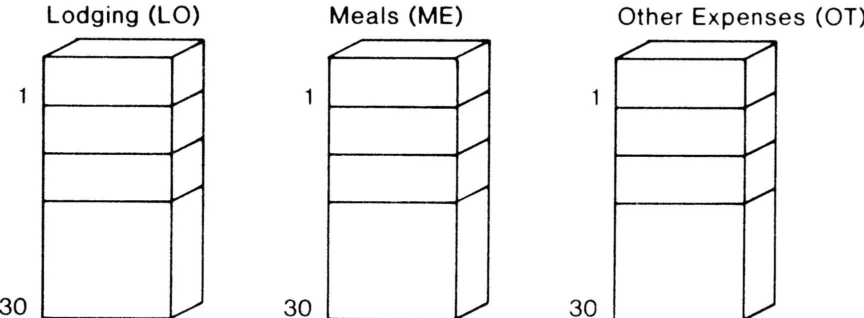
Figure 4.8 Improved weighted average program

ESTIMATING TRIP COSTS

How many times have you planned a trip and spent many hours calculating your anticipated expenses? This program assists in estimating trip costs assuming that you will be driving to and from your destination.

The principle is quite simple. Costs are based on gasoline costs for the entire trip plus other daily expenses, such as lodging and meals. Once the basics of the program are understood, modification for other personal needs you might have is quite easy to provide.

The program provides for up to 30 days' accumulation of data by using three arrays for Lodging (LO), Meals (ME), and Other Expenses (OT). These arrays as shown here define a maximum number of entries, but our program will be designed to use only the required portion of the arrays.



Inputs to this program come in two categories. First we need to know the costs for driving. To keep things simple, these will be limited to gasoline costs but could easily be expanded to include such other entries as oil, tolls, and repairs. To make things more interesting, let's ask the user for the following inputs:

Initial Trip Inputs
1. Number of days on trip 2. Total miles driven 3. Average miles per gallon 4. Average price per gallon

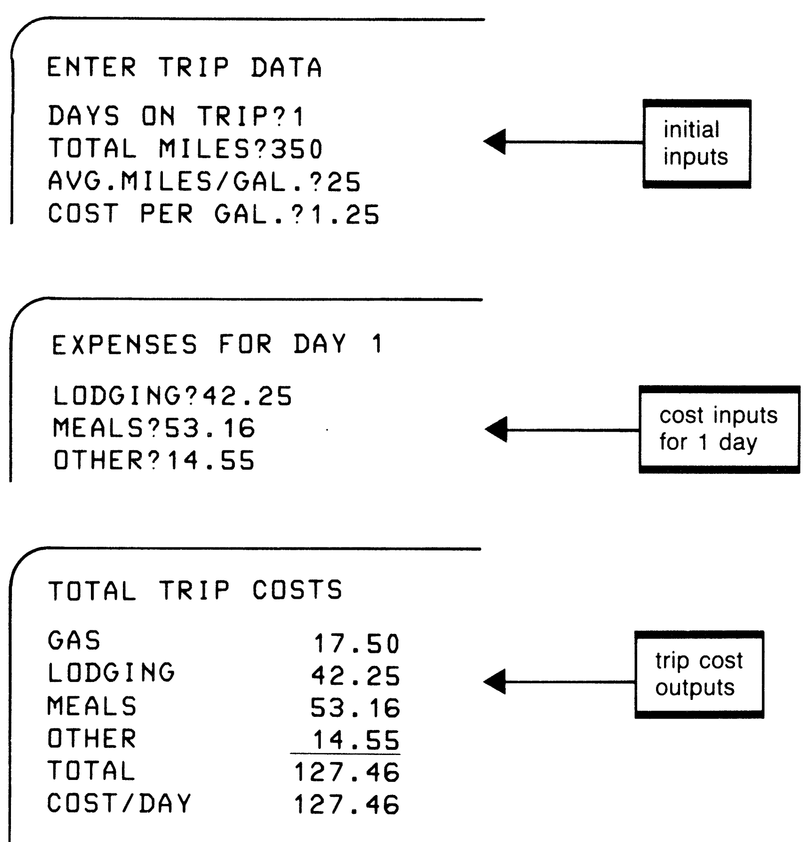
A second group of inputs are the daily expenses. Again there could be many entries but we will keep this area simple too, allowing for additional entries later. Here are the daily inputs:

Daily Inputs
1. Cost of lodging 2. Cost of meals 3. Other expenses

The outputs required from this program would be an itemized list of the trip and daily costs and a total cost for the trip. As an additional output, the average costs per day might be interesting, so let's include it in the output.

Trip Cost Outputs
1. Gasoline costs 2. Lodging 3. Meals 4. Other Expenses 5. Total trip cost 6. Average cost per day

Now that the input and output requirements have been defined in general terms, let's develop a screen layout showing specific prompts, inputs, and outputs. This is easier now since we have at least outlined our needs above.



Now we are ready to develop the English code for the program logic. An overview of the solution suggests this will not be a lengthy program, nor will it be very complex; but it does consist of a lot of input and output operations.

Trip Costs English Code
1. Enter initial trip inputs
2. Enter daily costs
3. Calculate trip costs
4. Display trip costs

Now expand the code:

1. Enter initial trip inputs
   1.1 Accept days on trip
   1.2 Accept total miles driven
   1.3 Accept average miles per gallon
   1.4 Accept average price per gallon
2. Enter daily costs until last day
   2.1 Accept cost of lodging
   2.2 Accept cost of meals
   2.3 Accept other expenses
3. Calculate trip costs
   3.1 Calculate gas cost
   3.2 Sum daily costs until last day
      3.2.1 Accumulate lodging costs
      3.2.2 Accumulate meal costs
      3.2.3 Accumulate other costs
   3.3 Calculate total cost
   3.4 Calculate average cost per day
4. Display trip costs
   4.1 Display costs for gas, lodging, meals, and other
   4.2 Display total cost
   4.3 Display average cost per day

English code lines 2 and 3.2 refer to repetition in the logic needed to control the activities on a daily basis. These lines are represented by FOR loops in the program in figure 4.9.

A slight improvement has been made when coding the program. That is a subroutine at 470 that clears the screen and prints down three lines. This has the effect of keeping the screen clear for each group of activities, moving the cursor down a few lines so the user is looking closer to the center of the screen rather than at the top all the time.

A second improvement is the use of the TAB feature in lines 390 to 450. TAB positions the cursor to the position indicated prior to printing the data. The TAB is discussed further in chapter 5.

```
 100 REM CALCULATE TRIP COSTS
 110 DIM LO(30),ME(30),OT(30)
 120 GOSUB 470
 130 PRINT "[rvs]ENTER TRIP DATA"
 140 PRINT
 150 INPUT "DAYS ON TRIP(MAX 30)";D
 160 INPUT "TOTAL MILES";M
 170 INPUT "AVG. MILES/GAL";A
 180 INPUT "COST PER GALLON";C
 190 FOR I=1 TO D
 200 GOSUB 470
 210 PRINT "[rvs]EXPENSES FOR DAY[off]";I
 220 PRINT
 230 INPUT "LODGING";LO(I)
 240 INPUT "MEALS";ME(I)
 250 INPUT "OTHER";OT(I)
 260 NEXT I
 270 REM CALCULATE COSTS
 280 GC=M/A*C
 290 FOR I=1 TO D
 300 LC=LC+LO(I)
 310 MC=MC+ME(I)
 320 OC=OC+OT(I)
 330 NEXT I
 340 TC=GC+LC+MC+OC
 350 AC=TC/D
 360 GOSUB 470
 370 PRINT "[rvs]TOTAL TRIP COSTS"
 380 PRINT
 390 PRINT "GAS";TAB(12);GC
 400 PRINT "LODGING";TAB(12);LC
 410 PRINT "MEALS";TAB(12);MC
 420 PRINT "OTHER";TAB(12);OC
 430 PRINT TAB(11);"------"
 440 PRINT "TOTAL";TAB(10);TC
 450 PRINT "COST/DAY";TAB(10);AC
 460 STOP
 470 PRINT "[clr]":PRINT:PRINT:PRINT
 480 RETURN
READY.
```

Figure 4.9 Calculating trip costs program

REVIEW QUESTIONS—CHAPTER 4

1. Why are arrays important? In what situations should they be used?
2. Write DIM statements for each of the following:
   a. A list of costs for 20 products
   b. Storage for the names of the months, January to December
   c. A matrix of distances between 10 different cities
3. Explain how data is read into an array using a list of costs as defined in question 2a above.
4. Describe the IF statement and the different logical operators used in it. What alternative paths can be taken when a condition evaluates true or false?
5. Write a FOR—NEXT statement to generate
   a. Integer values from 1 to 25
   b. Even numbers from 2 to 100
   c. A set of real numbers from .001 to .015 in increments of .001
   Print the results of each of the above loops.
6. Describe the benefits of a GOSUB over the GOTO statement.
7. Explain how random numbers may be generated. Write a statement to produce a random number between 1 and 10 inclusive. How would you get a random number between 5 and 15 inclusive?
8. What are some of the characteristics of a good game program?
9. Why are time delays sometimes necessary in a program? How are they created?
10. When designing a relatively complex program, why is it best to first design the formats for input and output? What type of information will be needed to define input and output when only the screen and keyboard are used?
11. What is the purpose of English or Pseudo code? How can it help in program development?

5 More on Input and Output

Many programs for personal and even for business use rely on the C-64's input and output capability. We have already used the INPUT statement for the input of data from the keyboard and the PRINT for screen output. In this chapter we will look at the READ statement, which will let us read data directly recorded within the BASIC program. Second, we will examine some more advanced features of the PRINT statement that will give us greater control over formatting screen output.

READ—DATA

```
statement number READ one or more variables
                  .
                  .
                  .
statement number DATA one or more constants
```

The DATA statement permits the program to store frequently used data within the program and access that data efficiently through the READ statement. A READ has a list of one or more variables separated by commas. Each time the READ is executed in the program, data is selected from the DATA statement in the order in which it appears. For example, the statements

```
10 READ N,S,A$
20 DATA 25,1.75,BOX
```

would cause the value 25 to be read from the DATA statement and assigned to the variable N. S receives the value 1.75 and A$ is given the value BOX. Notice that string data may be given with quotes ("BOX") or without (BOX). When a DATA statement or statements contain more data than the READ accesses, the additional data will be read the next time the program reaches a READ.

CREATING A BAR GRAPH

This program will use DATA statements to provide the values for bar graphs to be displayed on the screen. The program reads a series of five numbers and produces a simple bar graph from them. Each number represents a quantitative measurement that directly affects the length of the bar representing that number.

```
5 C$=[red cyn pur grn blu]"
10 FOR I=1 TO 5
20 READ N
30 FOR K=1 TO N
40 PRINT "[rvs]"MID$(C$,K,1)"[sp][off];
50 NEXT K
60 PRINT: PRINT
70 NEXT I
80 DATA 13,5,20,8,14
```

range of outer loop

range of inner loop

This program uses nested FOR loops, one to control the reading of the data (outside loop) and one to print the graph (inside loop). The outside loop varies I from 1 to 5 to read each of the five numbers from the DATA statement one number at a time. As each number is read in 20, it is stored in N. N is then used in the K loop (statement 30) to control the printing of the bar.

The bar is printed in 40 by using the reverse space within a character string. This is typed as quote, reverse, quote; then the MID$ function (which will be explained in chapter 6), followed by quote, space, reverse off, quote. MID$ is used to select the different color character from C$ for use in each of the bars. The semicolon, at the end of the PRINT statement, causes the reverse characters to continue printing as a solid line without spaces. The K loop controls the number of times it prints. This output from this program is shown in figure 5.1.

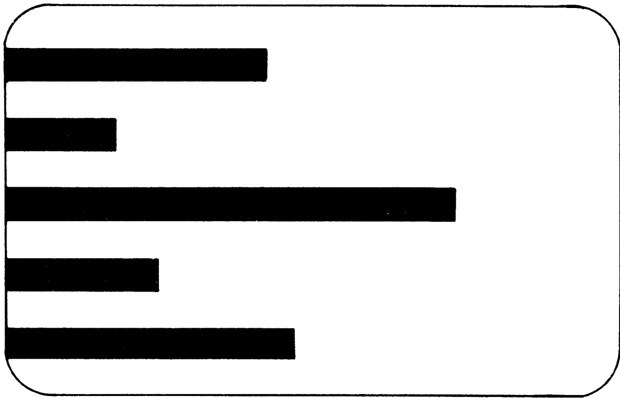


Figure 5.1 Simple bar graph output

One of the problems we sometimes face with the READ statement is knowing when there is no more data to read, which was not the case in the previous program. Suppose the data sometimes consists of 5 numbers but at other times of 8 or 12 numbers. A simple solution is to place a dummy value at the end of the data to denote the end. The program can then test for this value with an IF statement. Ordinarily the dummy value should be something that cannot be confused with real data. Here is another approach to solving the bar graph with the output in figure 5.2.

```
4 POKE 53281,1
5 C$="[blk red cyn pur grn blu yel]"
6 FOR I=1 TO 15
10 READ N
20 IF N=99 THEN STOP
30 FOR K=1 TO N
40 PRINT "[rvs]"MID$(C$,I,1)"[sp][off];
50 NEXT K
60 PRINT: PRINT
65 NEXT I
70 GOTO 10
80 DATA 13,5,20,8,14,17,3,99
```

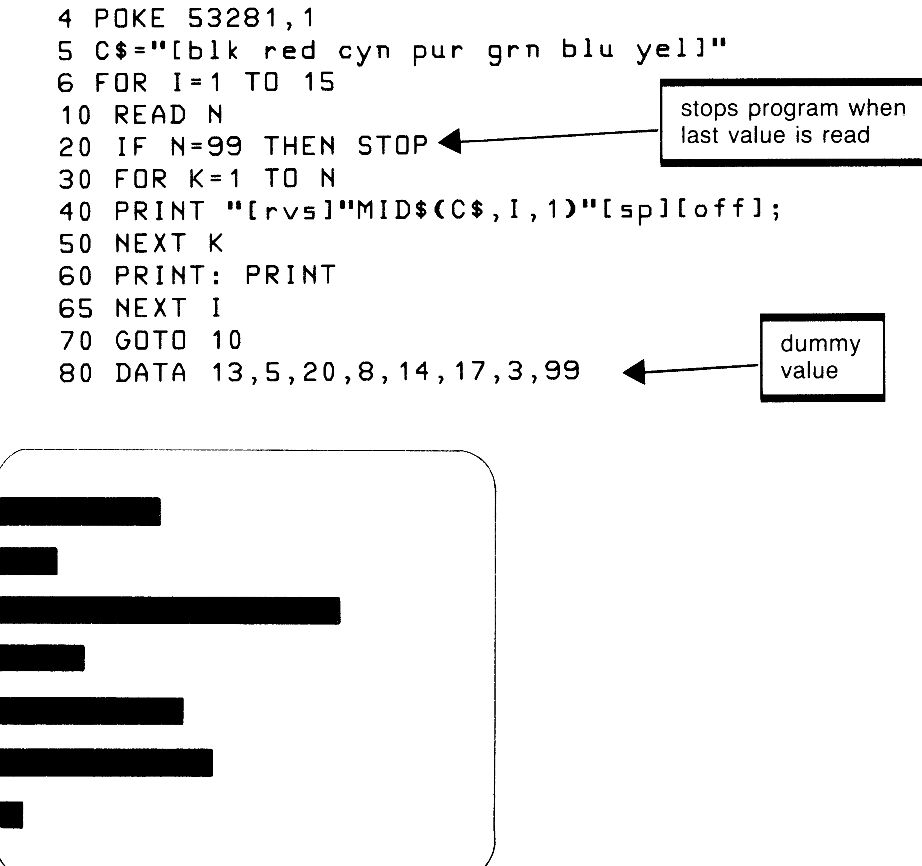
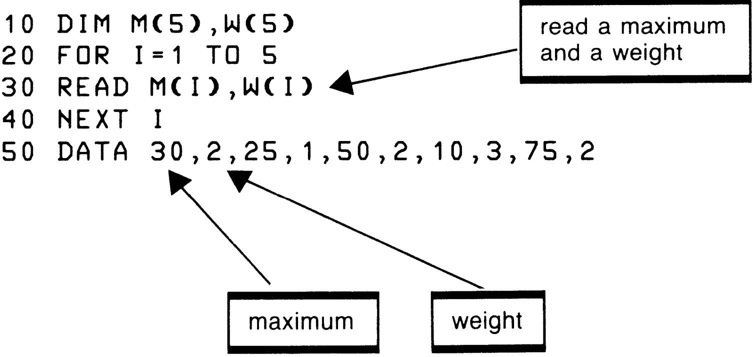


Figure 5.2 Bar graph output with variable data

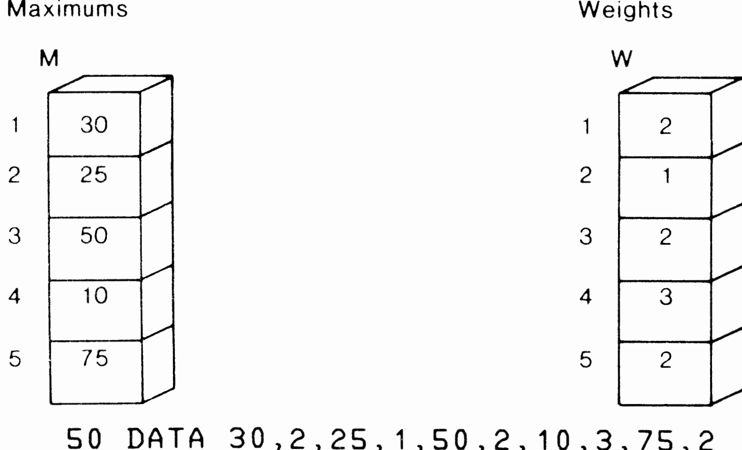
WEIGHTED AVERAGE WITH DATA

For one more example, let's look at the weighted average program again and consider an easier way to assign the maximum marks and weights to the arrays M and W by using DATA statements. The following code reads these values from data statements and assigns them to the array.

```
10 DIM M(5),W(5)
20 FOR I=1 TO 5
30 READ M(I),W(I)
40 NEXT I
50 DATA 30,2,25,1,50,2,10,3,75,2
```



The DATA statements contain a maximum mark followed by its weight. Thus the READ references M(I) and then W(I) to read each of these values, in order, from the DATA statement into the arrays as follows:




```
50 DATA 30,2,25,1,50,2,10,3,75,2
```

RESTORE

> statement number RESTORE

As data is read from DATA statements, the computer keeps track of each value in memory by using an internal data pointer. Although this pointer is not visible to the BASIC programmer, the BASIC interpreter in the ROM makes continuous reference to it.

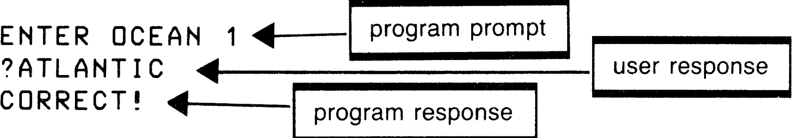
The RESTORE statement can be used in BASIC to set the data pointer back to the first data item in the DATA statements. This feature can be useful if you need to read through the same set of data more than once within the program.

It's not unusual for a Computer Assisted Instruction (CAI) program to give a series of instructions to a student and then ask questions about that same information. In the following program, we will do a simplified version of CAI by teaching the names of the five oceans on earth and then asking the student to respond by typing in the names. Beware! This is not intended to be an example of good CAI—far from it. But it does show how to use the RESTORE statement, which is our current objective.

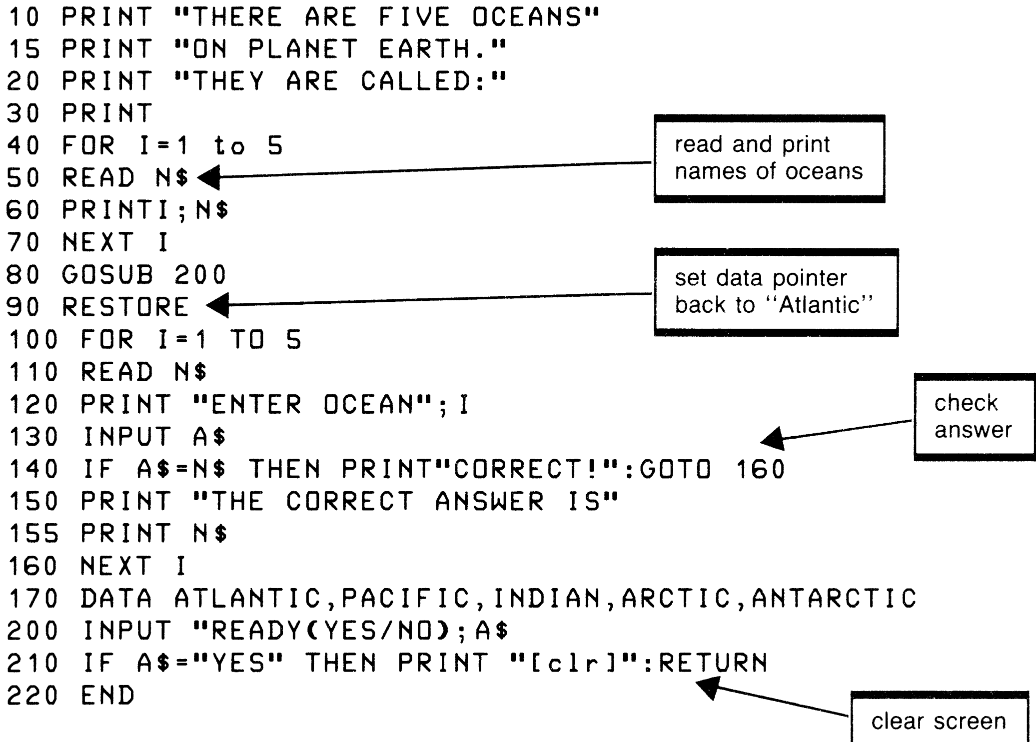
A DATA statement will be used to supply the names of the oceans. These names are first displayed as follows:

```
THERE ARE FIVE OCEANS ON PLANET EARTH.
THEY ARE CALLED:

1 ATLANTIC
2 PACIFIC
3 INDIAN
4 ARCTIC
5 ANTARCTIC

READY(YES/NO)?
```

After the names have been displayed, the program RESTOREs the data pointer so the data may be read again. When the user has had time to review these names and types YES to the prompt, the program proceeds to display the question as follows:



If the user had entered an incorrect answer, the program would then display the correct answer and proceed to the next ocean. Here is the program:




```
10 PRINT "THERE ARE FIVE OCEANS"
15 PRINT "ON PLANET EARTH."
20 PRINT "THEY ARE CALLED:"
30 PRINT
40 FOR I=1 to 5
50 READ N$
60 PRINTI;N$
70 NEXT I
80 GOSUB 200
90 RESTORE
100 FOR I=1 TO 5
110 READ N$
120 PRINT "ENTER OCEAN";I
130 INPUT A$
140 IF A$=N$ THEN PRINT"CORRECT!":GOTO 160
150 PRINT "THE CORRECT ANSWER IS"
155 PRINT N$
160 NEXT I
170 DATA ATLANTIC,PACIFIC,INDIAN,ARCTIC,ANTARCTIC
200 INPUT "READY(YES/NO);A$
210 IF A$="YES" THEN PRINT "[clr]":RETURN
220 END
```

MORE ABOUT PRINT

In chapter 3 we looked at the basic features of the PRINT statement and found the capability to solve a variety of problems relating to screen output. Now we are going to add several additional features to the PRINT that will give us more flexibility in our programming.

Cursor Controls

The cursor controls discussed in chapter 3 may also be used as commands within a string in the PRINT statement. These controls are the CLR, HOME, Cursor Down, Cursor Up, Cursor Right, and Cursor Left keys. These characters must be entered as part of a character string by first pressing the quote ('') and then keying the appropriate cursor character. For example, to clear the screen and position the print to the fourth line, type the statement:

```
100 PRINT "[clr dn dn dn]"
```

The square brackets in this statement indicate the enclosed characters are cursor controls and are entered by pressing the CLR key once, followed by the Cursor Down key three times. The square brackets are not typed; nor are the spaces between clr and dn.

This is how statement 100 would look on your screen:

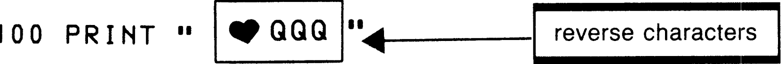
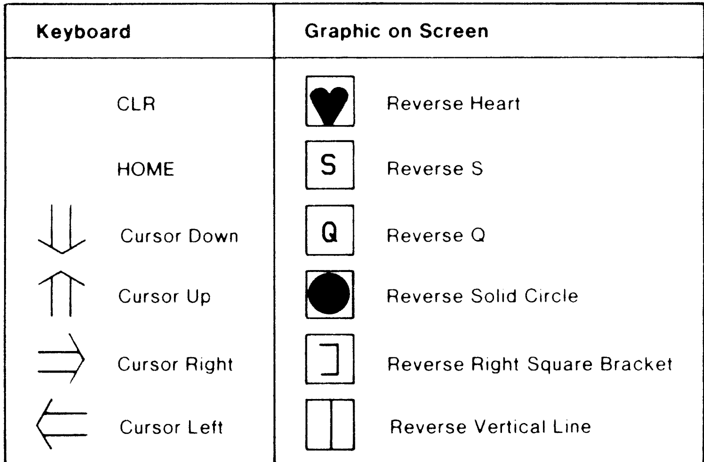


Figure 5.3 shows the cursor controls and the equivalent graphic displayed when they are entered into a string.



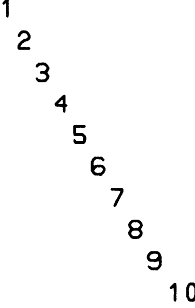
Keyboard	Graphic on Screen
CLR	Reverse Heart
HOME	S Reverse S
Cursor Down	Q Reverse Q
Cursor Up	Reverse Solid Circle
Cursor Right	Reverse Right Square Bracket
Cursor Left	Reverse Vertical Line

Figure 5.3 Cursor controls in a string

To get a feel for the use of cursor controls in the PRINT statement, try the following examples:

```
1 FOR I=1 TO 10
2 PRINT "[dn rt]";I;
3 NEXT I
```

These statements should give the output:



across and down the screen. Try some other variations of the cursor controls and see what results you can achieve.

Now try the code:

```
1 FOR I=1 TO 1000
2 PRINT "[clr dn rt]";I;
3 NEXT I
```

When you run this code, you'll get the numbers 1 to 1,000 displayed rapidly like a counter at the top left corner of the screen. The clear command in the string causes the screen to be cleared after each print operation and the cursor controls bring it back to the previous position to display the next number. Thus each new value is quickly cleared from the screen and replaced by the next, giving a counter effect.

TAB

TAB is a function of BASIC that is used with the PRINT statement to move the cursor to a specified location on the line. The format used is

TAB(n)

where n may be any value from 0 to 255. When TAB is used, the cursor moves to position n + 1 of the line. For example, TAB(12) would move the cursor to position 13.

The TAB function is always used in a PRINT statement prior to the value to be printed. TAB is followed by a semicolon and then the value to be printed at the location specified.

```
10 PRINT "12345678901234567890"
20 PRINT "ABCD";TAB(8);"B"
```

The above use of TAB will create the following output on the screen.

```
12345678901234567890
ABCD B
```

TAB is useful when you want to display outputs at a specific position on the screen, thus leaving a precise amount of space between each field. Multiple TABs may be used in one PRINT statement as follows:

```
20 PRINT "ABC";TAB(8);"B";TAB(13);N;TAB(18);A$
```

Assuming a value of 28 for N and "XYZ" for A$, the following output would be produced.

Columns

```
0         1         2         3
123456789012345678901234567890

ABC B 28 XYZ
```

These values seem to follow the pattern discussed above except for N's value of 28. We would expect it to be in column 14, but instead it is in column 15. The reason is quite simple. All numeric values reserve a position for the sign. In this case the sign is plus and so it does not display but rather leaves column 14 blank. If the value had been −28, then column 14 would be occupied with the minus sign as expected.

SPC

The SPC (space) function is similar to TAB except that SPC defines the number of spaces to leave between the previous output on the line and the one to follow.

```
10 PRINT "12345678901234567890"
20 PRINT "ABCD";SPC(5);"B"
```

The above code gives the results:

```
12345678901234567890
ABCD B
```

A common problem with both TAB and SPC is the space that replaces the sign for positive numeric values. Thus the statement

```
10 PRINT "SUM";SPC(3);S
```

will actually leave four spaces—three for the SPC command and one for the sign if S is a positive value. If S is negative, there will be three spaces and a minus sign before the value.

METRIC CONVERSION PROGRAM

The next program we want to tackle is one to convert from English to metric measure or the reverse. Figure 5.4 shows the conversion factors needed to do the calculations for the appropriate measurement. Values to convert from English to metric are given. Conversion in the opposite direction is achieved by using the inverse of these values.

English	Metric	Factor
Inches (in)	Centimeters (cm)	2.54
Feet (ft)	Meters (m)	.3048
Yards (yd)	Meters (m)	.9144
Miles (mi)	Kilometers (km)	1.6093
Sq. inches (sq.in)	Sq. centimeters (sq.cm)	6.452
Sq. feet (sq.ft)	Sq. meters (sq.m)	.0929
Sq. yards (sq.yd)	Sq. meters (sq.m)	.8361
Sq. miles (sq.yd)	Sq. hectometers (sq.h)	259
Dry quarts (dry qt)	Liters (1)	1.101
Liquid quarts (qt)	Liters (1)	.9463
Liquid gallons (gal)	Decaliters (dl)	.3785
Pecks (pk)	Liters (1)	8.81
Bushels (bu)	Hectoliters (hl)	.3524

Figure 5.4 Metric conversion factors

If we look at this table carefully, we can determine that it consists of three parts: linear, square, and liquid and dry measure. These parts form a basis for the logical organization of our program. Initially the program will read the table information from DATA statements and load it into arrays for the program's use.

So far in this book we have used English code for the purpose of developing program logic. Actually, the use of English or Pseudo code for developing program logic is quite new. Prior to its use, the flowchart was a popular program development tool.

One of the advantages of flowcharting is the visual or graphic aspect that sometimes helps you visualize a solution. Since some people are more verbal oriented and others image oriented, the flowchart might seem to be a good alternative if you are the second type of person. One reason for the move away from flowcharts has been the stress on a technique called structured programming. That, however, is a subject for another book.

Figure 5.5 shows the various symbols used for flowcharting, and figure 5.6 shows the flowchart for the logical solution of this problem.

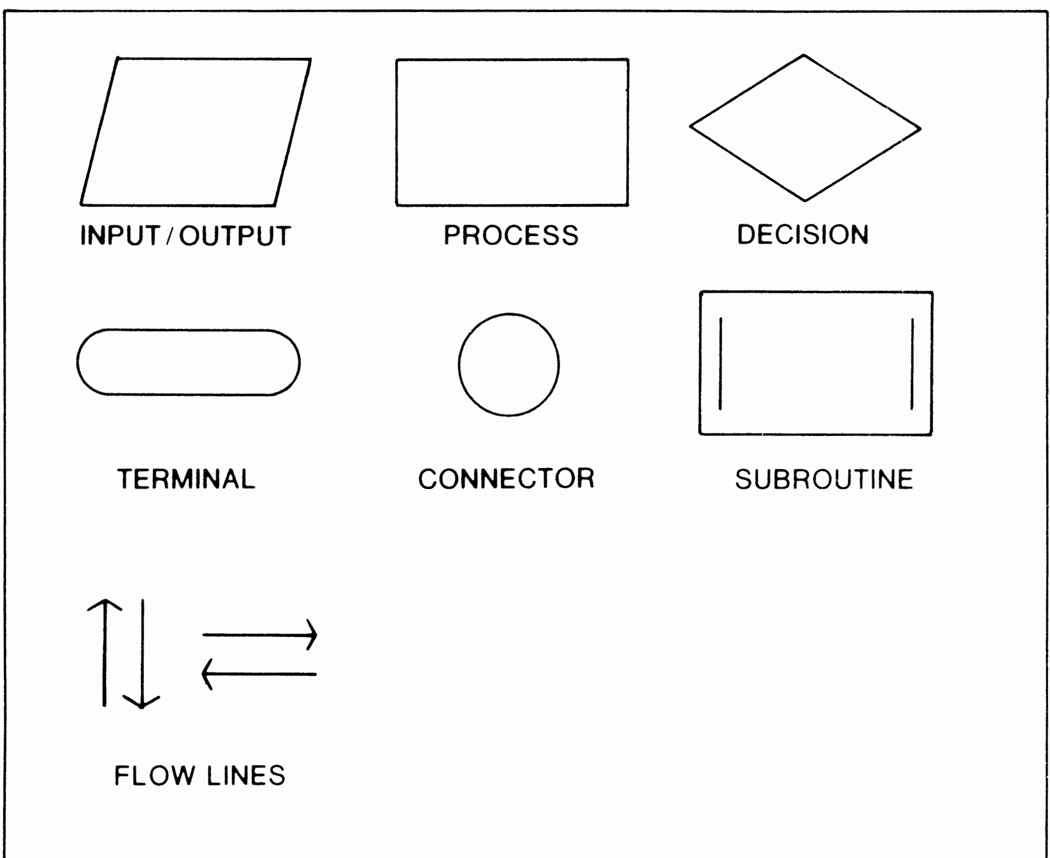


Figure 5.5 Flowchart symbols

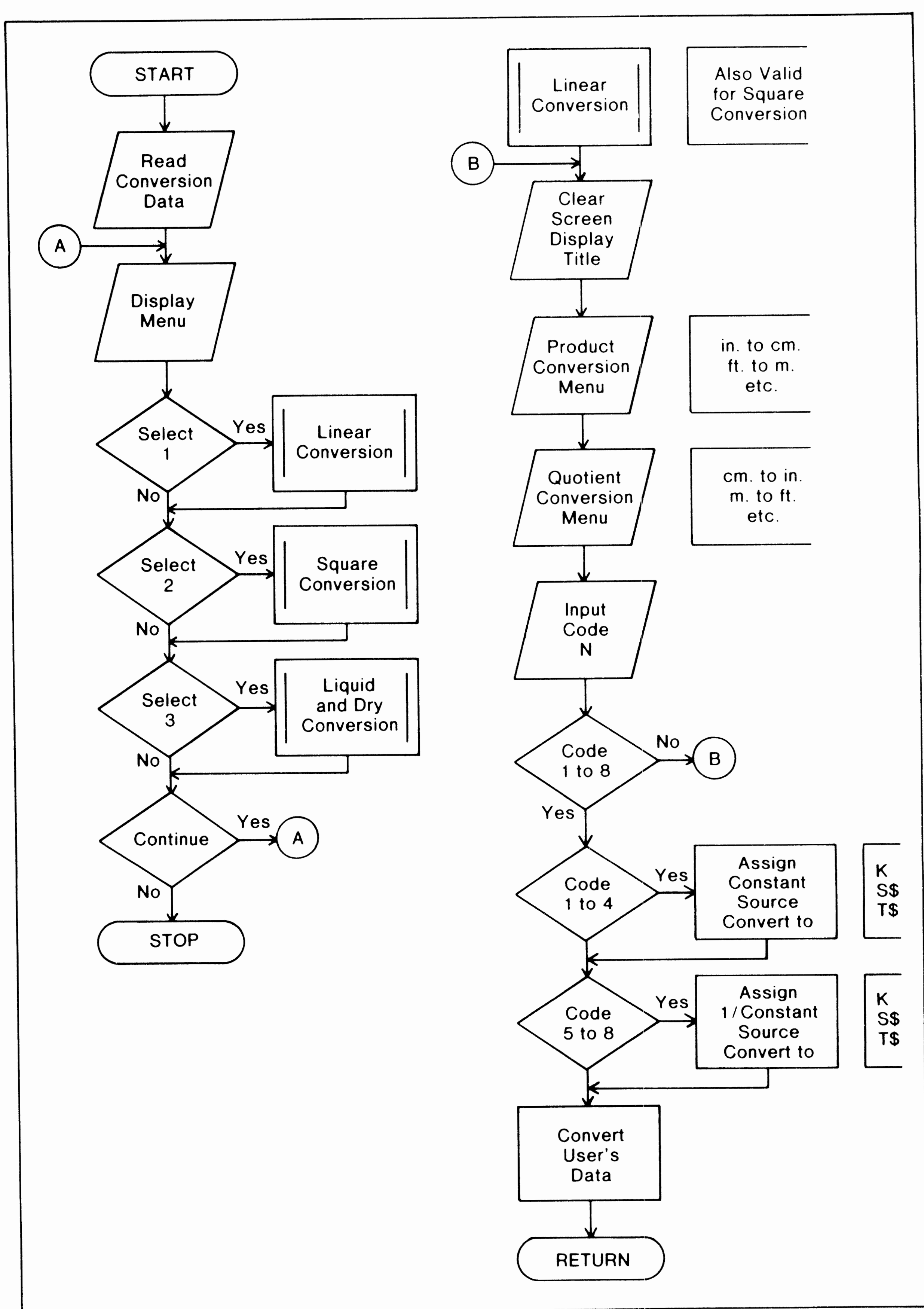


Figure 5.6 Metric conversion flowchart

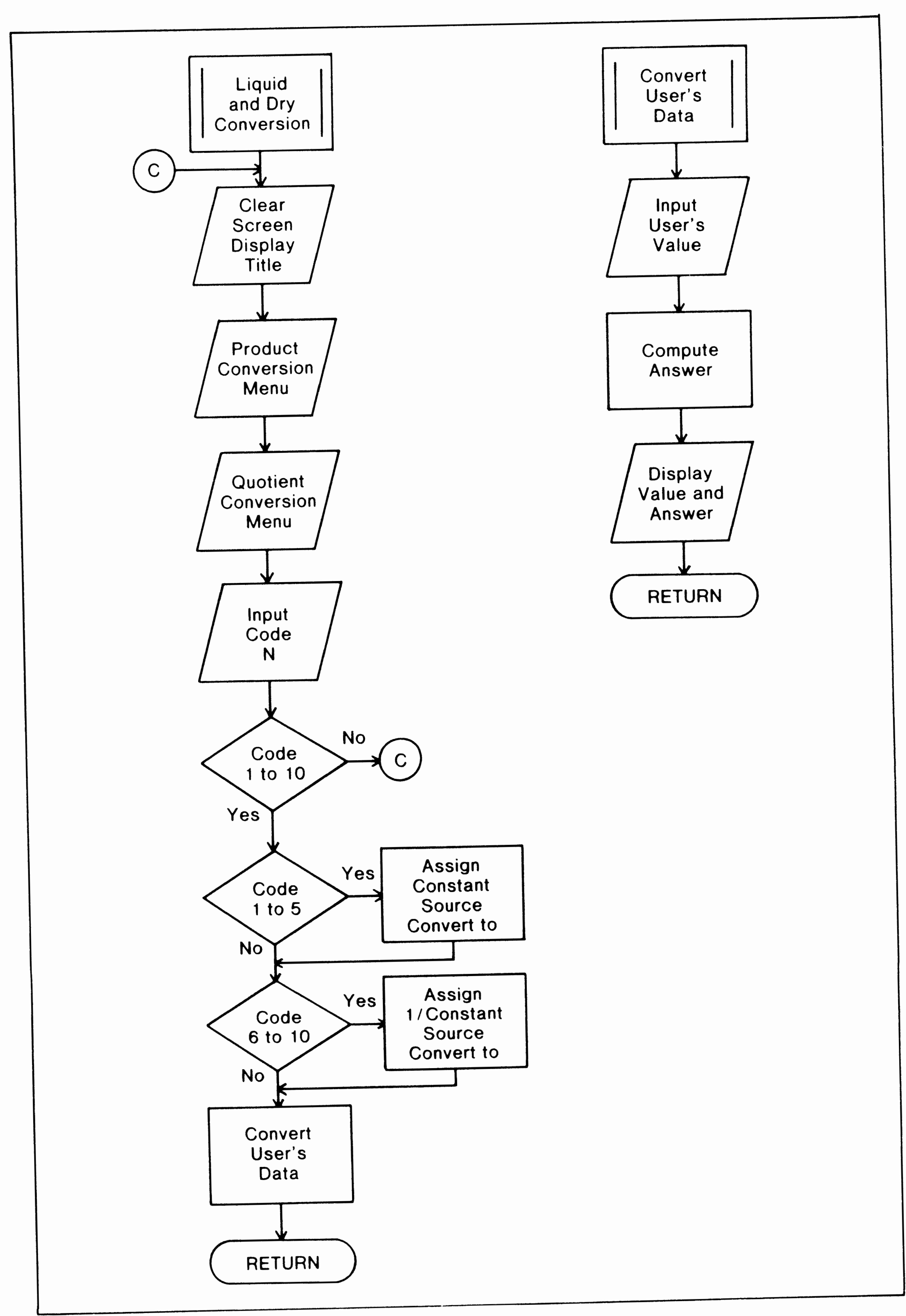


Figure 5.6 (continued)

This program is a natural for arrays and, of course, DATA statements are used heavily. The arrays are organized as shown in figure 5.7. Notice at this time the program makes no distinction between the different categories of measure. Array A$ contains the description of English measurements, B$ the metric descriptions, and C contains each conversion factor for the related English and metric conversion.

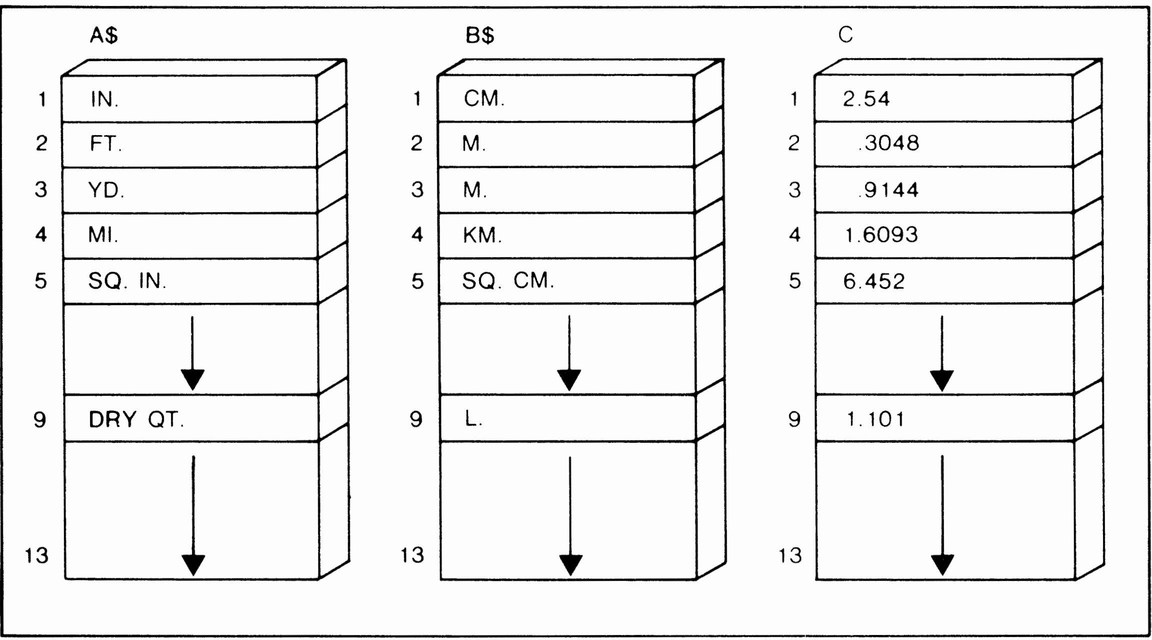


Figure 5.7 Arrays for conversion data

These arrays are first used to produce the menu for selecting the required conversion. For the Linear Conversion Menu, the statements

```
360 FOR I=1 TO 4
370 PRINT I;"-";A$(I);"TO";B$(I)
380 PRINT
390 NEXT I
```

are used to print the sequence:

1 — IN. TO CM.
2 — FT. TO M.
3 — YD. TO M.
4 — MI. TO KM.

The variable I has a twofold purpose here. One, it is used to print the digits 1 to 4 in the menu, and two, it selects the position of arrays A$ and B$ for printing the English and metric descriptions.

The second half of the Linear menu permits conversions from metric to English and uses the code

```
400 FOR I=1 TO 4
410 PRINT I+4;"-";B$(I);"TO";A$(I)
420 PRINT
430 NEXT I
```

These statements print menu items 5 to 8 as follows:

5 — CM. TO IN.
6 — M. TO FT.
7 — M. TO YD.
8 — KM. TO MI.

Again I is used in two ways. This time, to print digits 5 to 8, the expression I + 4 is used in the PRINT statement so when I is 1 the number 5 will print, 2 causes 6 to print, and so on. Subscripts in this case are still 1 to 4, but the array references are now reversed in the PRINT, giving B$ then A$.

This basic approch is used for each menu, except that the starting and ending values of the FOR loop are adjusted to correspond to the location of the names in the arrays. For square conversion, the values are 5 to 8 as indicated in the following code.

```
550 FOR I=5 TO 8
560 PRINT I-4;"-";A$(I);"TO";B$(I)
570 PRINT
580 NEXT I
590 FOR I=5 TO 8
600 PRINT I;"-"; $(I);"TO";A$(I)
610 PRINT
620 NEXT I
```

Liquid and dry conversion uses the values 9 to 13, thus relating to the positions in the arrays for these descriptions.

Once the appropriate menu has been displayed the program's user enters one of the values to select the conversion required. This number is stored in N. From the value of N, three things are accomplished. The variable K is assigned the conversion constant from the array C. S$ is assigned the description of the source of conversion from either array A$ or B$. And T$ is assigned the description we are converting to, from either A$ or B$.

For example, if linear conversion is selected and N was given 6, indicating meters to feet, K will be assigned 1/C(N – 4), which reduces to 1/C(2), thus referring to the value .3048 in position 2 of array C. S$ will be given the value at position 2 of B$ (M.), and T$ is assigned position 2 of A$ (FT.). The following code does this operation.

```
470 IF N<=4 THEN K=C(N):S$=A$(N):T$=B$(N)
480 IF N>4 THEN K=1/C(N-4):S$=B$(N-4):T$=A$(N-4)
```

In the second menu, square conversion, the values of N may also be 1 to 8, but the position of these conversions in the arrays are from 5 to 8. In this case, menu responses of 1 to 4 correspond to positions 5 to 8. By simply increasing N by 4, the program points to the right place. Responses 5 to 8 luckily also point to positions 5 to 8, so no adjustment is needed. Here is the code.

```
660 IF N<=4 THEN K=C(N+4):S$=A$(N+4):T$=B$(N+4)
670 IF N>4 THEN K=1/C(N):S$=B$(N):T$=A$(N)
```

Finally, liquid and dry conversion can have values of 1 to 10 for N. The first five relate to positions 9 to 13 of the arrays. This correspondence is achieved using N + 8 for the subscript. Menu responses 6 to 10 use a subscript of N + 3 to reference array positions 9 to 13.

```
850 IF N<=5 THEN K=C(N+8):S$=A$(N+8):T$=B$(N+8)
860 IF N>5 THEN K=1/C(N+3):S$=B$(N+3):T$=A$(N+3)
```

The complete metric conversion program is shown in figure 5.8.

```
100 REM METRIC CONVERSION
110 DIM A$(13),B$(13),C(13)
120 FOR I=1 TO 13
130 READ A$(I),B$(I),C(I)
140 NEXT I
150 REM DISPLAY MAIN MENU
160 PRINT "[clr 5 dn's]"
170 PRINT TAB(3);"METRIC CONVERSION"
180 PRINT:PRINT
190 PRINT TAB(2);"1 - LINEAR MEASURE"
200 PRINT
210 PRINT TAB(2);"2 - SQUARE MEASURE"
220 PRINT
230 PRINT TAB(2);"3 - LIQUID & DRY"
240 PRINT:PRINT
250 INPUT "ENTER ONE CODE";S
260 IF S=1 THEN GOSUB 320
270 IF S=2 THEN GOSUB 510
280 IF S=3 THEN GOSUB 700
290 INPUT "CONTINUE(YES/NO)";A$
300 IF A$="YES" THEN 160
310 STOP
320 REM LINEAR CONVERSION
330 PRINT "[clr]"
340 PRINT TAB(3);"LINEAR CONVERSION"
350 PRINT:PRINT
360 FOR I=1 TO 4
370 PRINT I;"-";A$(I);" TO ";B$(I)
380 PRINT
390 NEXT I
400 FOR I=1 TO 4
410 PRINT I+4;"-";B$(I);" TO ";A$(I)
420 PRINT
430 NEXT I
440 PRINT:PRINT
450 INPUT "ENTER ONE CODE";N
460 IF N<1 OR N>8 THEN 320
470 IF N<=4 THEN K=C(N):S$=A$(N):T$=B$(N)
480 IF N>4 THEN K=1/C(N-4):S$=B$(N-4):T$=A$(N-4)
490 GOSUB 890
500 RETURN
510 REM SQUARE CONVERSION
520 PRINT "[clr]"
530 PRINT TAB(3);"SQUARE CONVERSION"
540 PRINT:PRINT
550 FOR I=5 TO 8
560 PRINT I-4;"-";A$(I);" TO ";B$(I)
570 PRINT
580 NEXT I
```

Figure 5.8 Metric conversion program

```
590 FOR I=5 TO 8
600 PRINT I;"-";B$(I);" TO ";A$(I)
610 PRINT
620 NEXT I
630 PRINT:PRINT
640 INPUT "ENTER ONE CODE";N
650 IF N<1 OR N>8 THEN 510
660 IF N<=4 THEN K=C(N+4):S$=A$(N+4):T$=B$(N+4)
670 IF N>4 THEN K=1/C(N):S$=B$(N):T$=A$(N)
680 GOSUB 890
690 RETURN
700 REM LIQUID & DRY CONVERSION
710 PRINT "[clr]"
720 PRINT "LIQUID/DRY CONVERSION"
740 FOR I=9 TO 13
750 PRINT I-8;"-";A$(I);" TO ";B$(I)
760 PRINT
770 NEXT I
780 FOR I=9 TO 13
790 PRINT I-3;"-";B$(I);" TO ";A$(I)
800 PRINT
810 NEXT I
820 PRINT
830 INPUT "ENTER ONE CODE";N
840 IF N<1 OR N>10 THEN 700
850 IF N<=5 THEN K=C(N+8):S$=A$(N+8):T$=B$(N+8)
860 IF N>5 THEN K=1/C(N+3):S$=B$(N+3):T$=A$(N+3)
870 GOSUB 890
880 RETURN
890 REM CONVERT DATA
900 PRINT "[clr dn dn dn]ENTER ";S$;
910 INPUT V
920 A=V*K
930 PRINT V;S$;" = ";A;T$
940 RETURN
950 REM CONVERSION DATA
960 DATA IN.,CM.,2.54
970 DATA FT.,M.,.3048
980 DATA YD.,M.,.9144
990 DATA MI.,KM.,1.6093
1000 DATA SQ.IN.,SQ.CM.,6.452
1010 DATA SQ.FT.,SQ.M.,.0929
1020 DATA SQ.YD.,SQ.M.,.8361
1030 DATA SQ.MI.,SQ.H.,259
1040 DATA DRY QT.,L.,1.101
1050 DATA QT.,L.,.9463
1060 DATA GAL.,DL.,.3785
1070 DATA PK.,L.,8.81
1080 DATA BU.,HL.,.3524
```

Figure 5.8 (continued)

SIMPLE PAYROLL PROGRAM

Payroll is a traditional data processing activity that can be implemented successfully on a microcomputer. Although no attempt is made here to write a complete payroll program, some of the principles are demonstrated even in this simple version.

One of the needs of a payroll system is to withhold income tax for each pay period. This is done by using a table that supplies the amount of tax to be deducted for different levels of income. A table of this type for weekly deductions is shown in figure 5.9. The first column represents the maximum gross salary range. Column 2 is the amount of tax to be withheld, and column 3 gives a percentage to be applied to an amount in excess of the minimum for that category.

Gross Amount	Withholding Amount	Percent
0— 46	0	0
46—127	0	15
127—210	12.15	18
210—288	27.09	21
288—369	43.47	24
369—454	62.91	28
454—556	86.71	32
556—999	119.35	37

Figure 5.9 Weekly withholding tax table

To use the table, find the category for a given gross salary and select the tax to be deducted. For instance, a gross of $300 would require a tax of $43.47 plus 24 percent of the remaining $22 (derived from $300 − $288).

A flowchart for the solution is in figure 5.10, the program is in figure 5.11, and a sample output in figure 5.12. The program first loads the tax table into arrays GA (gross amount), WA (withholding amount), and PC (percentage). Notice that the data and array GA contain only the high value for each category. This simplifies the array and yet the lower value is always available in the preceding array element. Since a problem could occur with the first entry, a value of zero is stored in element zero of each array. Remember, arrays begin with element zero in BASIC.

Next, each employee's record is read and displayed on the screen and the user is requested to enter the regular and overtime hours. The program then calculates gross salary (GS) in 330 and looks for the appropriate position in the table in statements 350 to 370. The location is stored in L. Now line 380 calculates the tax. Finally, the gross, tax, and net amounts are displayed.

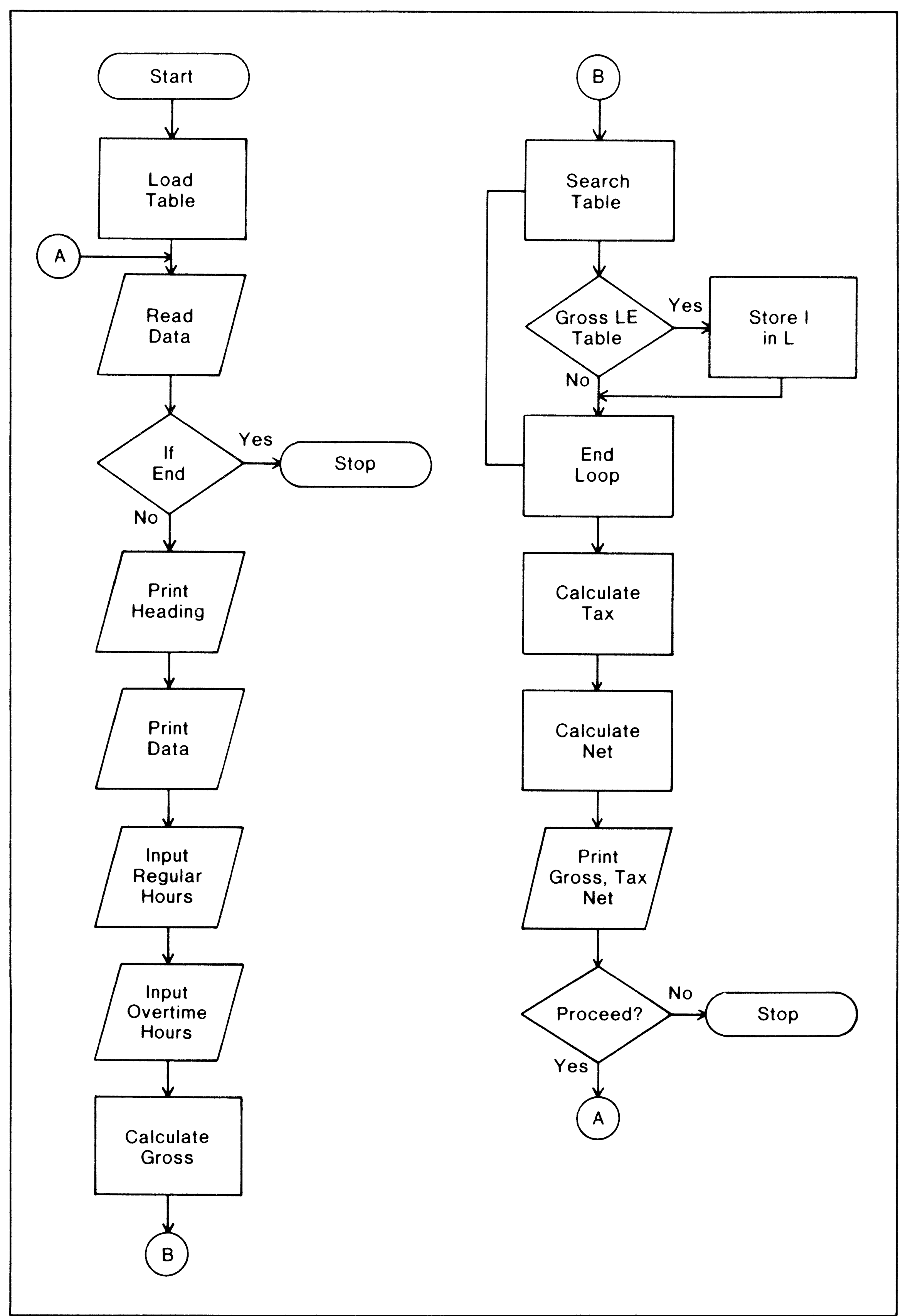


Figure 5.10 Flowchart for simple payroll

```
100 REM SIMPLE PAYROLL
110 DIM GA(8),WA(8),PC(8)
120 REM LOAD TAX TABLES
130 FOR I=0 TO 8
140 READ GA(I),WA(I),PC(I)
150 NEXT I
160 REM TAX TABLE DATA
170 DATA 0,0,0
180 DATA 46,0,0,127,0,.15,210,12.15,.18
190 DATA 288,27.09,.21,369,43.47,.24
200 DATA 454,62.91,.28,556,86.71,.32
210 DATA 999,119.35,.37
220 REM PROCESS PAYROLL
230 PRINT "[clr]"
240 READ D,N$,R
250 IF D=999 THEN STOP
260 PRINT "DEPT    NAME    RATE"
270 PRINT
280 PRINT D;TAB(6);N$;TAB(16);R
290 PRINT
300 INPUT "REG. HRS. WORKED";HW
310 INPUT "OVERTIME HOURS";HO
320 PRINT
330 GS=HW*R+HO*1.5*R
340 REM FIND TAX RATE LOCATION IN TABLE
350 FOR I=1 TO 8
360 IF GS<=GA(I) THEN L=I:I=8
370 NEXT I
380 TAX=WA(L)+((GS-GA(L-1))*PC(L))
390 NS=GS-TAX
400 PRINT "GROSS SALARY";GS
410 PRINT
420 PRINT "TAX DEDUCTED";TAX
430 PRINT
440 PRINT "NET SALARY";NS
450 PRINT:PRINT
460 INPUT "PROCEED(YES/NO)";X$
470 IF X$="YES" THEN 230
480 STOP
490 DATA 10,J. SMITH,8.25
500 DATA 10,K. SIMON,9.15
510 DATA 11,C. ROUSE,8.55
520 DATA 11,P. SHANE,7.90
530 DATA 999,END,9
```

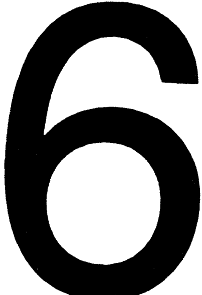
Figure 5.11 Simple payroll program

```
DEPT NAME RATE

10 J. SMITH 8.25

REG. HRS. WORKED?38
OVERTIME HOURS?5

GROSS SALARY 375.375

TX DEDUCTED 64.695

NET SALARY 310.68

PROCEED (YES/NO)?
```

Figure 5.12 Sample payroll screen output

REVIEW QUESTIONS—CHAPTER 5

1. Explain the use of READ and DATA statements. Why use them instead of the INPUT statement?
2. If a program needs to read the same data more than once during processing, give two ways this may be done in BASIC.
3. Why would we want to use cursor control characters in string data or in a PRINT statement?
4. Give an example not used in the chapter of cursor controls used in a string.
5. Explain the use of the TAB and SPC functions.
6. Why do programmers sometimes use flowcharts when developing program logic?
7. What are the advantages and disadvantages of flowcharts compared to English code? Which of these methods do you prefer?
8. Consider the program in this chapter for metric conversion. What other conversions might be useful in a program of this type? How would you revise the program to include additional conversion categories?



Advanced BASIC

Many of the statements in this chapter could be implemented with an appropriate combination of BASIC statements previously discussed. Why then do we bother to use these new statements? The major reason is because in certain circumstances these new commands are easier to use. In other situations the new instructions are faster to execute and more efficient than the old way. So let's get on with a new bag of tricks!

GET

```
statement number GET variable
```

The GET Statement is used to read a single character from the keyboard without the need to press RETURN after the character has been entered (figure 6.1). The GET in a program looks like

```
100 GET A$
```

But since the action is instantaneous, a delay loop is required to wait for user response and to check the variable for a value. This test is done by comparing A$ to a null string. If A$ is still null (or empty), a branch back to the GET is taken.

Whenever a key is pressed, the value typed enters A$, which is no longer null. At this time the program will continue at the statement following 100.

A common use for GET is in response to a question requiring a yes or no as an answer. Try the following example:

```
100 PRINT "DO YOU WANT INSTRUCTIONS (Y/N)?":
110 GET Z$:IF Z$="" THEN 110
120 IF Z$="Y" THEN GOSUB 500
```

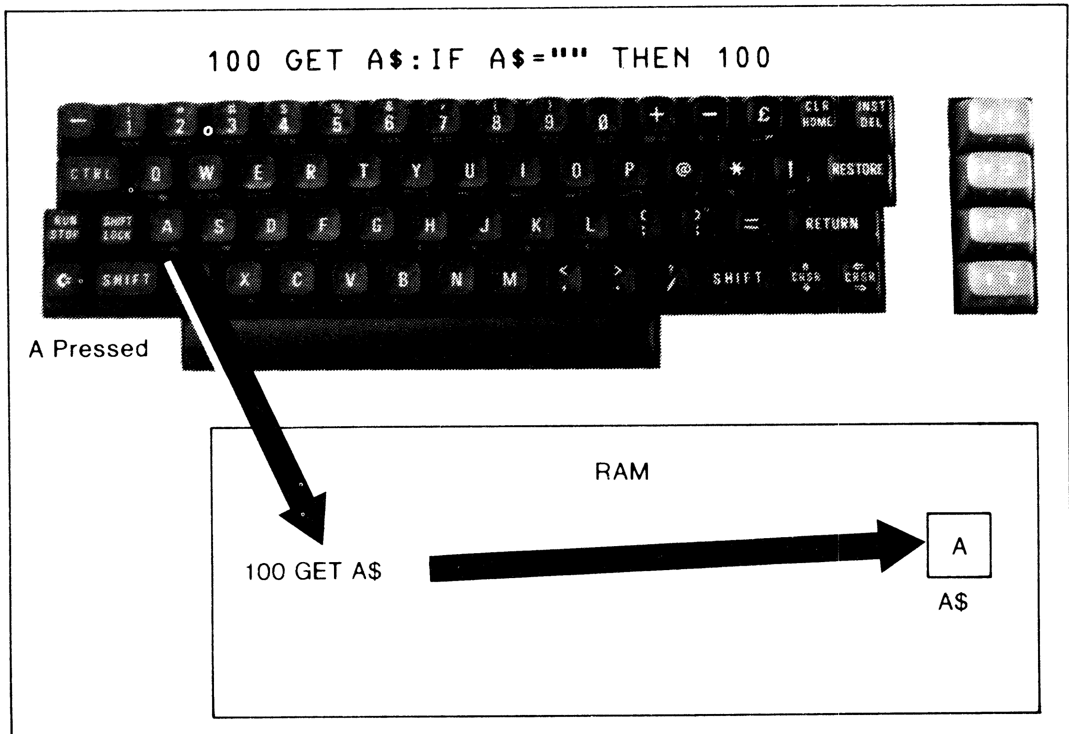


Figure 6.1 GET statement

First the program prints a message indicating the type of response expected, either a Y representing yes or an N for no. Then the GET statement is placed in a waiting loop until the user presses a key. The value of the key pressed goes into Z$, and statement 120 then tests Z$ for a Y response. If the user of our program types the entire word yes, only the first letter will be accepted and the remainder of the word is ignored.

ON—GOSUB

statement number ON {variable / expression} GOSUB sn1,sn2,...

The ON-GOSUB is a single statement that replaces a series of IF—THEN—GOSUBs when the values to be tested are numeric and consecutive. Given a menu

```
1 - RENT

2 - UTILITIES

3 - AUTO

4 - EDUCATION

ENTER ONE CODE?
```

we could test the input response with four IF statements that each branch to the appropriate subroutine for processing. Assuming the program accepted the code with the command

```
100 INPUT N
```

the ON statement will process the response with only one statement.

```
110 ON N GOSUB 1100,1200,1300,1400
```

The ON statement examines the variable (N) presented to it and, depending on its value, branches to one of the line numbers specified (figure 6.2). If N is 1, branching goes to 1100, the first line number. For a value of 2, the program branches to 1200, and so on. Any numeric variable name may be used instead of N and any number of line numbers specified up to a maximum of 255.

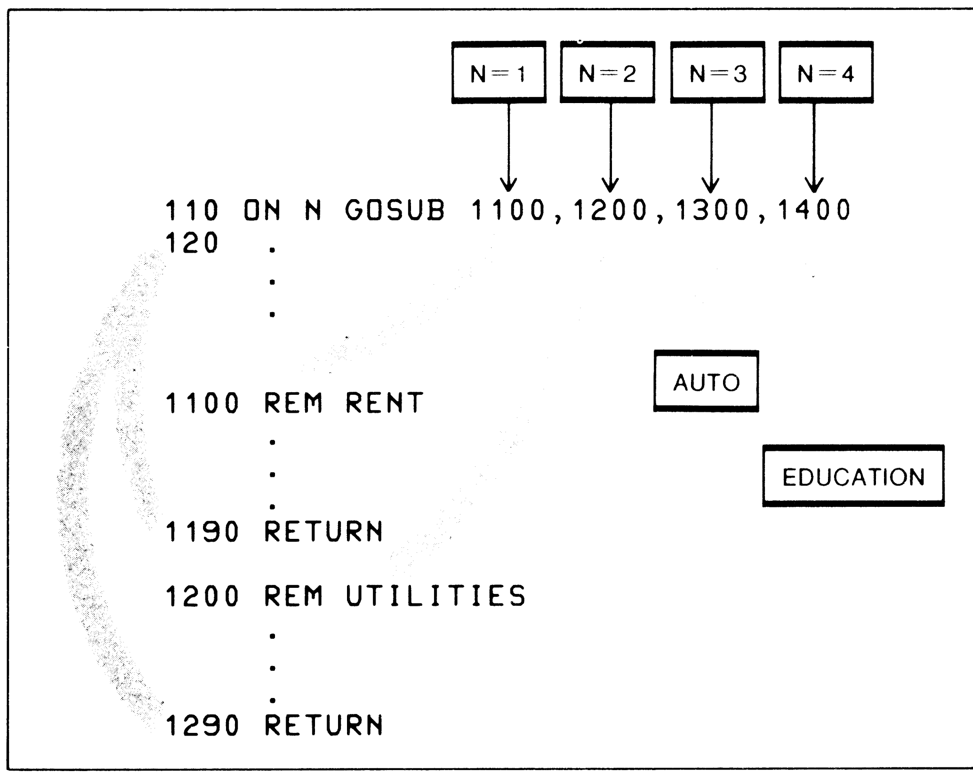
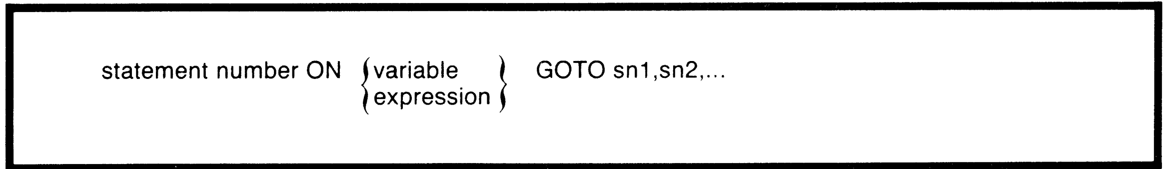


Figure 6.2 Using ON—GOSUB

If the variable contained a value of 0 or if it exceeded the number of entries, for example a value of 5, then the ON continues at the next statement without branching.

Since we used the GOSUB, the program will branch to the appropriate subroutine, such as 1300 for a value of 3 in N. When a RETURN is reached in subroutine 1300, control returns to the ON statement and passes to the next statement following line number 110.

ON—GOTO



The only difference between ON—GOTO and ON—GOSUB is that ON—GOTO does not return from the location to which it has branched. Otherwise the statements function identically.

```
SELECT ONE CATEGORY

1 - CHOLESTEROL

2 - SATURATED FAT

3 - OLEIC ACIDS

4 - LINOLEIC ACIDS
```

The statement to examine the response to this menu might be as follows:

```
40 ON A GOTO 150,200,300,450
```

If the response to the menu was 1 to select CHOLESTEROL, the program would branch to statement 150, SATURATED FAT to 200, OLEIC ACIDS to 300, and LINOLEIC ACIDS to statement 450, as shown in figure 6.3.

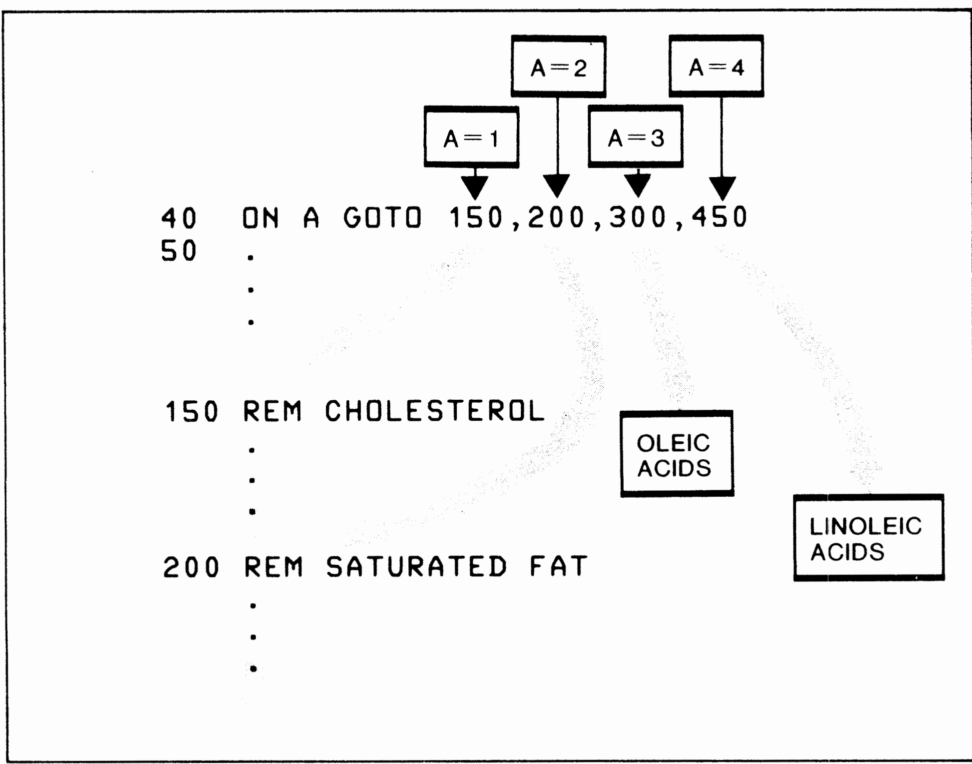


Figure 6.3 Using ON—GOTO

POKE

statement number POKE address, { constant / variable / expression }

The POKE statement places a value directly into a specified memory location. For example, the statement

```
10 POKE 1024,83
```

would place a graphic heart symbol on the upper left-hand corner of the screen. The value 1024 refers to the memory address and may take on values from 0 to 65535 inclusive, while the number 83 is the value being stored at this address. 83 is C-64 machine code for the heart symbol (see appendix E for C-64 codes).

POKE is also useful for setting the C-64 into capital or lowercase mode. Normally the C-64 displays capital letters and graphic characters. But in some cases, such as word processing or instructional programs, the use of capitals and lowercase letters might be required. Pressing the shift and Commodore keys simultaneously from the keyboard sets the C-64 to capital/lowercase just as on a typewriter. Try it!

Pressing shift and the Commodore key returns the C-64 back to normal capitals. This approach is fine from the keyboard, but what if you are writing a program that requires capital/lowercase and don't want the program user to get involved with the need to switch the C-64 to lowercase? In this case a POKE can be used. The statement

```
POKE 53272,23
```

used in the program with an appropriate statement number will switch the C-64 to lowercase. To go back to normal, use

```
POKE 53272,21
```

The same effect can be achieved with the PRINT statement using a CHR$ function.

```
PRINT CHR$(14)      set to lowercase
PRINT CHR$(142)  set to capitals and graphics
```

Another use of POKE is to change the screen and border colors as described in chapter 3.

```
POKE 53280,2:POKE 53281,6
```

would give you a red border with a blue screen. The POKEs are setting a code in the C-64's memory that tells it what colors you want for your screen. To restore your C-64 back to the normal colors use

```
POKE 53280,14:POKE 53281,6
```

The C-64 screen contains 1,000 positions for displaying characters (figure 6.4). These positions are addressed in memory from 1024 to 2023. Any of the screen addresses may be POKEd to place characters directly on the screen for graphics or special effects.

Figure 6.4 also shows a matrix for the color memory. Beginning at address 55296 and extending to 56295 are memory positions, defining the color code for each position of the screen. The colors are numbered from 0 to 7 as follows:

Code	Color	Code	Color
0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light red
3	Cyan	11	Gray 1
4	Purple	12	Gray 2
5	Green	13	Light green
6	Blue	14	Light blue
7	Yellow	15	Gray 3

When a character is POKEd on the screen, a color must also be POKEd into the color memory as shown in figure 6.5. Now try the following program.

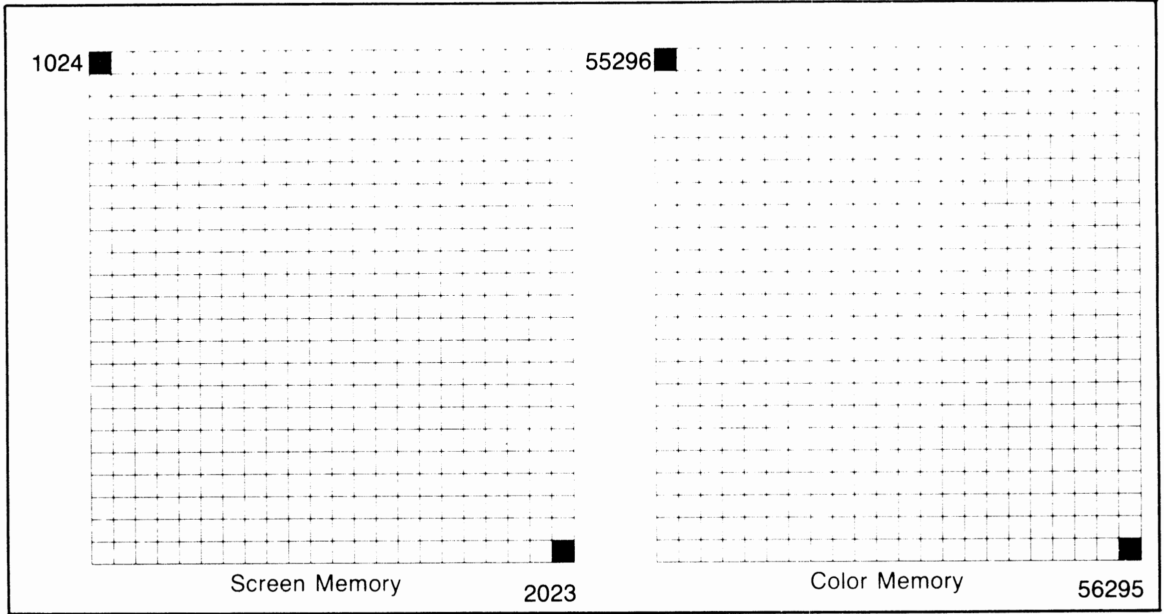


Figure 6.4 Screen and color memory addresses

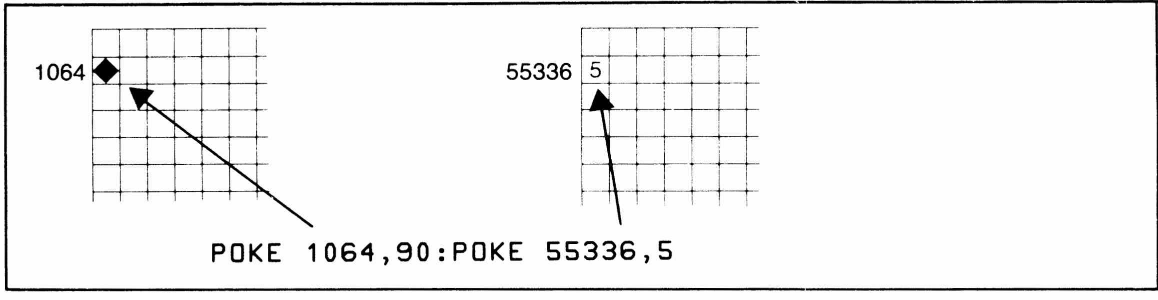
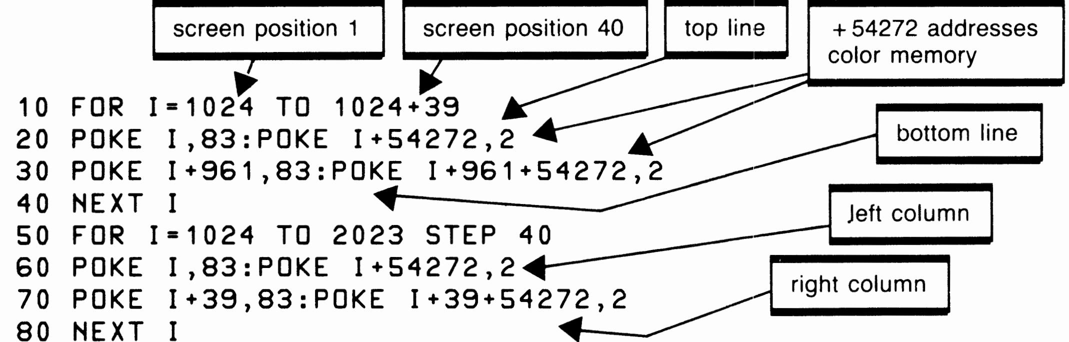


Figure 6.5 Display screen addressing




```
10 FOR I=1024 TO 1024+39
20 POKE I,83:POKE I+54272,2
30 POKE I+961,83:POKE I+961+54272,2
40 NEXT I
50 FOR I=1024 TO 2023 STEP 40
60 POKE I,83:POKE I+54272,2
70 POKE I+39,83:POKE I+39+54272,2
80 NEXT I
```

Now clear the screen and RUN this program. You should get a border of red hearts around the outside edge of the screen. Although this could be done with the PRINT statement, the POKE is faster than printing and permits precise control of the character and color at each screen position, which is particularly useful for animation. We will look at the use of POKEs in much greater detail in the chapter on graphics and animation.

PEEK

statement number PEEK (address)

The PEEK returns the contents of an address in memory. If the results from the previous program were left on the screen so that the border of hearts was still there, the command

```
PRINT PEEK(1024)
```

would display the value 83 (figure 6.6), which is the ASCII code of the heart at memory address 1024. Of course if the heart was no longer there, some other value would be displayed. The memory address could also be a variable, as in

```
PRINT PEEK(I)
```

PEEK may return a value to a variable, such as

```
120 M=PEEK(I)
```

where the result is assigned to a variable such as M. PEEK may be used in an IF statement or any other statement where a numeric value is accepted.

```
240 IF PEEK(J) = 49 THEN PRINT "YES"
```

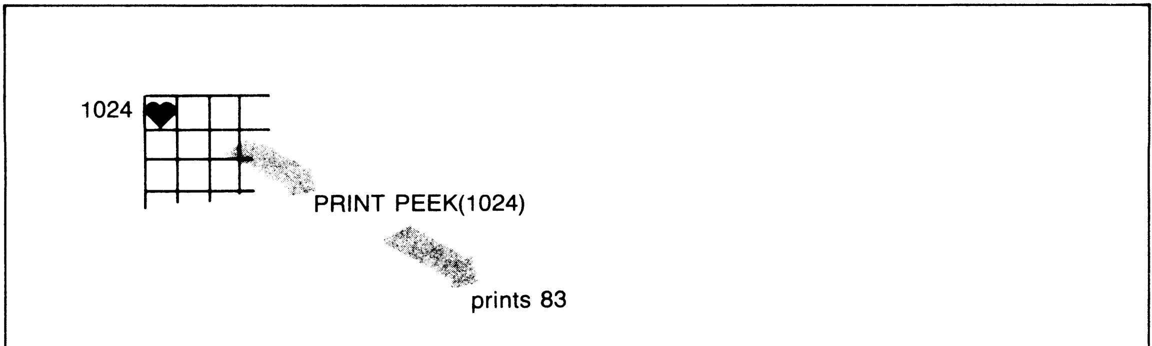
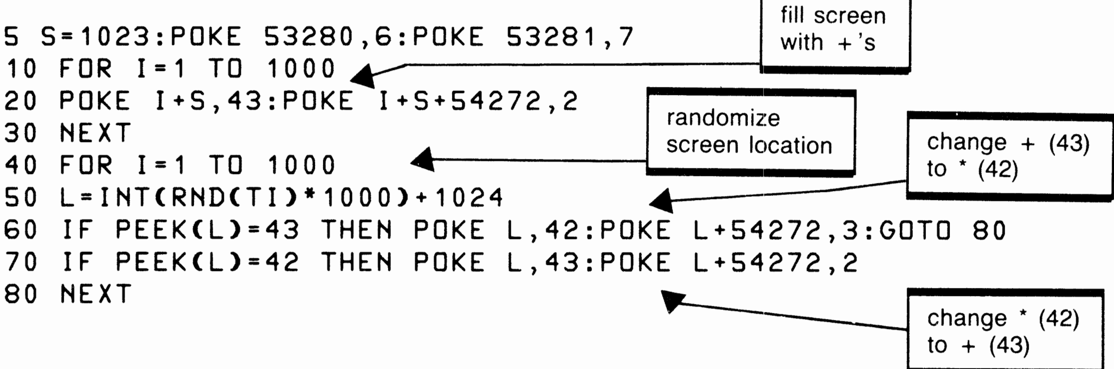


Figure 6.6 PEEKing at screen memory

RANDOM STARS

Here is an interesting program using PEEKs and POKEs. It doesn't do anything practical, but it is fun. The program begins by filling the C-64's screen with red plus signs (ASCII 43). Then it randomly changes + signs to cyan/red * (asterisks), and if it finds an asterisk it changes it back into a cyan/red + sign. Try it!



```
5 S=1023:POKE 53280,6:POKE 53281,7
10 FOR I=1 TO 1000
20 POKE I+S,43:POKE I+S+54272,2
30 NEXT
40 FOR I=1 TO 1000
50 L=INT(RND(TI)*1000)+1024
60 IF PEEK(L)=43 THEN POKE L,42:POKE L+54272,3:GOTO 80
70 IF PEEK(L)=42 THEN POKE L,43:POKE L+54272,2
80 NEXT
```


TI AND TI$ FUNCTIONS

To give you the ability to time operations in your programs, the C-64 has provided a built-in clock that can be accessed using the TI and TI$ functions.

TI or TIme Function

The TI function accesses an electronic counter that begins counting when the C-64 is first turned on. The increments are in 1/60 of a second, called "jiffies." Try entering the immediate mode command

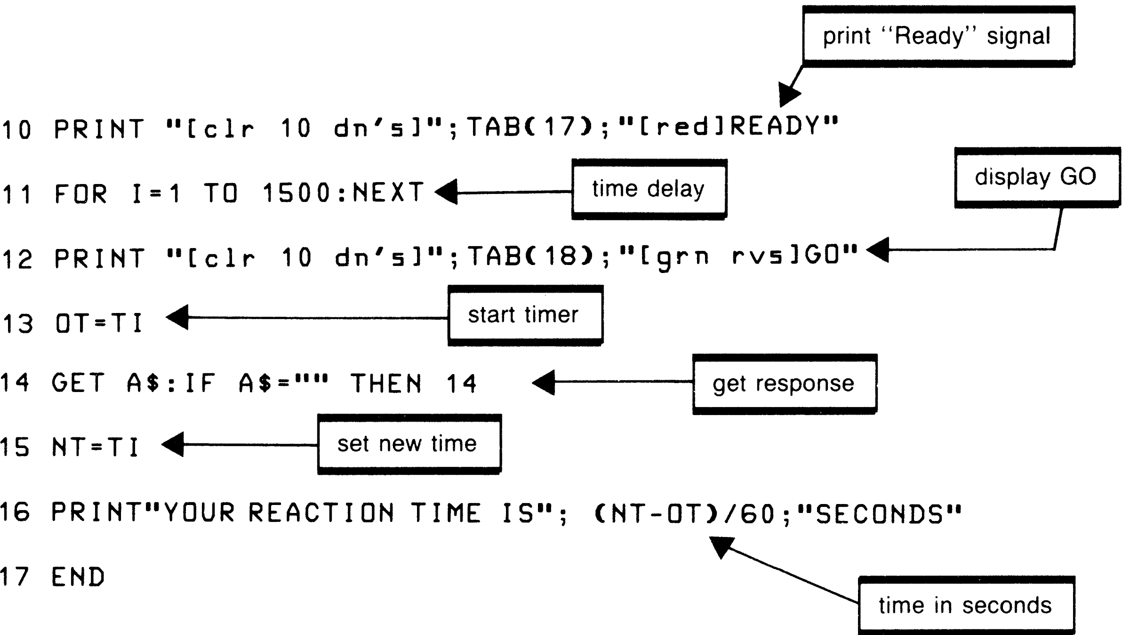
```
?TI
```

and you will get a number something like 67462, meaning it has been 67,462 jiffies since the computer was powered up. If you divide this number by 60 you get 1124.36667, or about 1,124 seconds since the power was turned on. There are:

60 jiffies per second
36,000 jiffies per minute
216,000 jiffies per hour

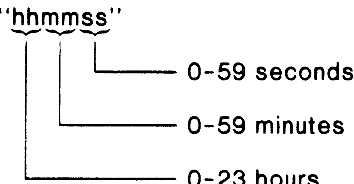
Reaction Timer Program

How fast are your reactions? The following program is a simple reaction timer using the TI function you can use to check your response time. Essentially the program takes the time prior to a GO signal and stores it in variable OT (old time). When you press any key, time is taken again and stored in NT (new time). The reaction time is then the difference between these values divided by 60 to convert the time to seconds.



TI$ or TIme$

The TI$ function returns the time in hours, minutes, and seconds in the form of a six-digit character string. The format of the string is:



Initially TI$ is set to all zeros upon computer power up. You may want it to reflect the actual time of day, and this can be done by assigning the time to TI$. To do this, use immediate mode as follows. Suppose we want to set TI$ to 10:24 a.m. This would be entered a few seconds before the actual time has arrived.

```
TI$="102400"
```

Now wait until exactly 10:24 and press Return. The function now contains the time specified in the above expression. If you are entering time during the afternoon remember this function records 24-hour time and adjust your figures accordingly.

Now try this brief program.

```
10 PRINT TIME$;"[lt lt lt lt lt lt]";
20 GOTO 10
```

DEF FN

statement number DEF FN name (argument) =	{ constant variable expression }

The Define Function statement permits you to define your own functions for use in a BASIC program. Using DEF FN is a particular advantage when a formula needs to be used at several different places in the program. Rather than writing the formula each time it is needed, you simply reference the function described at the beginning of the program.

The format of DEF FN is as follows:

DEF FNname (argument) = formula,

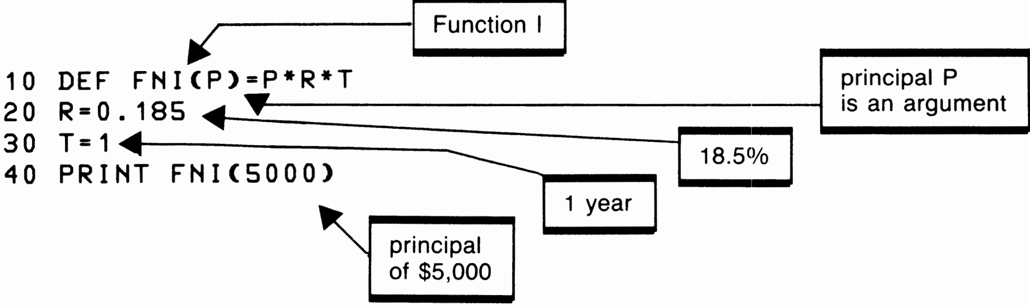
where

—name is the function name consisting of one or two characters according to the rules of variable names.
—argument is a floating-point variable that supplies a value for the function to operate upon.

For example, to define a function that calculates simple interest, the formula i = prt is used, where

i is the interest
p is the principal
r is the rate
t is the time in years

If we wanted to define a function to find the interest for a given investment, the following code could be used.



If a constant rate and time were used (rather unlikely in this case), the function could be defined as

```
10 DEF FNI(P)=P*0.185*1
```

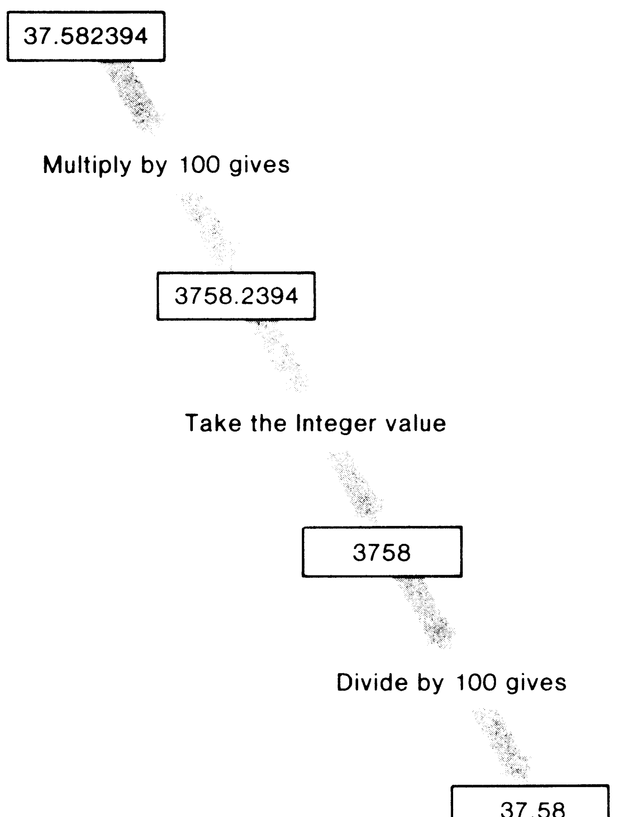
If the rate was constant but the time variable, use

```
10 DEF FNI(P)=T*0.185*T
```

As another example, consider the need to control the maximum number of decimal positions printed. In many situations the result of a calculation can have many decimal positions when only one or two are really necessary. In some BASICs, a Print Using is available for this type of control, but C-64 BASIC doesn't have this feature.

One way of limiting a value to no more than two decimal positions is to multiply the number by 100 and take the integer portion of the result. Then divide this integer by 100 to get back to the original range of the value.

Example:



Here is the program to limit results to no more than two decimal positions.

```
100 DEF FNR(N)=INT(N*100)/100
110 FOR I=1 TO 3
120 READ N
130 PRINT FNR(N)
140 NEXT I
150 DATA 236.45678,1234.567,5.00123456
```

This program gives the output

```
236.45
1234.56
5
```

Several limitations are obvious from this output. One is that this function controls only the maximum number of decimal positions. Therefore in the case of the last value, no decimals print. The other limitation is that the decimal points do not line up as would be required in such reports as those used for accounting applications.

ARITHMETIC FUNCTIONS

```
function-name (argument)
```

Functions are subroutines already built into the ROM to let you do complicated arithmetic operations much more easily than if you had to write your own function or subroutine. Arithmetic functions use a function name and an argument that supplies the value to be acted upon.

For example, the integer function used in a PRINT statement is

```
PRINT INT(26.2987)
```

INT is the function name and 26.2987 is the argument. The function removes the fraction and returns the integer value of 26 to be printed.

Functions that return numeric values may be used any place a number is valid. Therefore they may be used in arithmetic statements, IF statements, FOR statements, subscripts, and even in other functions.

ABS

The ABS function supplies the absolute value of the argument by removing its sign.
Examples:

```
10 N=ABS(256)
20 K=ABS(-256)
30 J=ABS(-23.05)
40 PRINT N;K;J
```

gives the output

```
256 256 23.05
```

ATN

ATN returns the arctangent in radians of the argument. The returned value will be in the range ± 17.

```
?ATN(30)           gives 1.53747533 radians
?180/π*ATN(45)     gives 88.72697 degrees
```

COS

The COS function returns the cosine of the argument. The argument represents a value expressed in radians. To find the cosine of .01 radians

```
?COS(.01)   gives .99995
```

The cosine of 1 radian is

```
?COS(1)     gives .540302306
```

EXP

This function returns the value of e raised to the power of the argument (e^{arg}) where e is the value 2.71828183. The argument must be in the range ±88.0296910.

```
?EXP(0)        gives 1
?EXP(1)        gives 2.71828183
?EXP(10)       gives 22026.4658
```

In the case of the EXP function an argument that exceeds the maximum will give an overflow error, while an argument less than the minimum returns a 0 value.

INT

The INT function returns the integer portion of a real number by truncating the fraction. The result is in real form not integer (percentage). In the case of negative numbers, INT adds −1 to the integer part after truncation.

```
?INT(27.57)       gives 27
?INT(1.05)        gives 1
?INT(0.05)        gives 0
?INT(-27.57)      gives -28
```

LOG

The LOG function returns the natural logarithm or log to the base e, where e is the value 2.71828183. If the argument is 0 or negative, an Illegal Quantity message is generated.

```
?LOG(10)        gives 2.30258509
?LOG(1)         gives 0
?LOG(5000)      gives 8.5171932
```

To find log to the base 10, simply divide the natural logarithm by LOG(10). For example:

```
?LOG(40)/LOG(10)   gives 1.60205999
```

RND

The RND function generates random numbers between 0 and 1. These numbers can be of value in games and simulation programs. The function

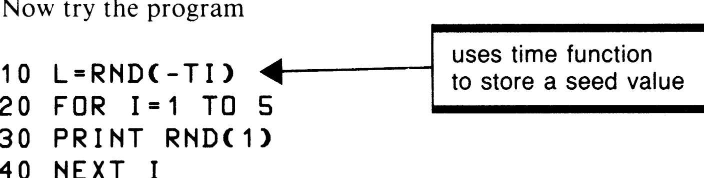
```
RND(argument)       returns a random number
RND(-argument)      stores a new seed number for the generator
```

Storing a new seed begins a new sequence of random numbers.

This step is useful when the C-64 is first turned on to ensure completely random numbers each time a program is run.

Now try the program

```
10 L=RND(-TI)
20 FOR I=1 TO 5
30 PRINT RND(1)
40 NEXT I
```



This program displayed the sequence

```
.574960506
.0214301237
.558740495
.277825403
.0561438672
```

but if you run this program on your C-64, you will get a different sequence of random numbers.

Creating a Specific Set of Random Numbers

Often we need a set of random numbers within a specific range. Take for example the range 0 to 9. This range can be created by multiplying the random number generated by 10 (1 larger than the maximum value required), and then taking the integer of this result.

```
20 N=INT(RND)(1)*10)
```

This statement will generate a number in the range from 0 to 9. A number in the range 1 to 9 is created with the expression

```
20 N=INT(RND(1)*9+1)
```

The +1 adjusts the values 0 to 8 generated by the function to become 1 to 9, the range we wanted.

SGN

The SGN function is used to determine the sign of a number. If the number is

positive a +1 is returned
zero a 0 is returned
negative a −1 is returned

Examples:

`?SGN(25)`	displays 1
`?SGN(0)`	displays 0
`?SGN(-12)`	displays −1

SIN

The SIN function returns the sine of the argument, which is expressed in radians.
Examples:

`?SIN(.01)`	displays 9.99983334E−03
`?SIN(1)`	displays .841470985

SQR

The square root of a positive number is found with the SQR function.
Examples:

```
?SQR(36)        displays  6
?SQR(20.25)     displays  4.5
?SQR(4096)      displays  64
```

TAN

TAN returns the tangent of the argument expressed in radians.
Examples:

```
?TAN(.01)   displays   .0100003333
?TAN(1)     displays  1.55740772
```

Converting Radians to Degrees

If the problem you are solving requires a value in degrees, rather than in radians, the following formula may be used.

degrees = 180 / π *radians

For example,

```
?180/ π *ATN(30)
```

The reverse situation may be true when using functions like SIN, COS, and TAN. To convert from degrees to radians, use

radians = degrees *π / 180

for example,

```
?SIN(60* π /180)
```

STRING FUNCTIONS

Functions in this category are like the arithmetic functions except that character strings are involved in the operation. A string function may require one, two, or even three arguments, depending on the function. When more than one argument is used, each argument is separated from the other with a comma.

ASC

This function returns the ASCII code equivalent to the single character specified in the argument. ASCII codes are numeric values between 0 and 255 as listed in the appendices.

```
?ASC("A")     displays  65
?ASC("1")     displays  49
?ASC("◥")     displays  223
A=ASC("$")    assigns ◥ 36 to variable A
```

One of the ASC's uses is to translate characters that have been entered from the keyboard but are POKEd onto the screen. In order to do this the keyboard characters must be changed to screen ASCII characters, which are in a different ASCII sequence. The following is a subroutine that converts the variable B$, which might have been read with

```
100 GET B$: IF B$="" THEN 100
110 GOSUB 7000
```

into the ASCII variable C, which may then be POKEd onto the screen.

```
7000 REM CONVERT B$ TO ASCII SCREEN CHARACTER C
7010 IF ASC(B$)<96 AND ASC(B$)>63 THEN C=ASC(B$)-64:
     GOTO 7050
7020 IF ASC(B$)<129 THEN C=ASC(B$):GOTO 7050
7030 IF ASC(B$)>159 AND ASC(B$)<192 THEN C=ASC(B$)-64
7040 IF ASC(B$)>191 THEN C=ASC(B$)-128
7050 RETURN
```

If the value A had been entered into B$, this would be the ASCII value 65. Subroutine 7000 identifies this value and subtracts 64 from it in line 7010, creating the POKE value 1. The graphic symbol ◥ is ASCII 223 and is converted to POKE value 95 in line 7040 by subtracting 128 from B$ and storing the result in C.

CHR$

The CHR$ function is the inverse of ASC in that it returns the character equivalent of the single ASCII code specified in the argument. The argument may be a numeric value from 0 to 255.

```
?CHR$("65")        displays  A
?CHR$("49")        displays  1
A=169:?CHR$(A)     displays  ◥
```

One value of the CHR$ function is its ability to display values that cannot be expressed in a PRINT statement. For example, the character (") is a delimiter and cannot be part of a character string in BASIC. It can be displayed with

```
100 PRINT CHR$(34)
```

LEFT$

The LEFT$ function is used to extract the left-most characters from a string. This function requires two arguments:

LEFT$(string, length)
—String is the character string to be used in the operation
—Length is the length of the string to be extracted and must be in the range 0 to 255

Example

A common use for LEFT$ is to select the first letter of a user's response. For example, if a yes answer is expected to a query, we may examine only the first byte of the response for a "Y." This minimizes processing errors due to spelling mistakes, carelessness, or the operator simply not entering the complete word.

```
100 INPUT R$
110 IF LEFT$(R$,1)="Y" then 200
```

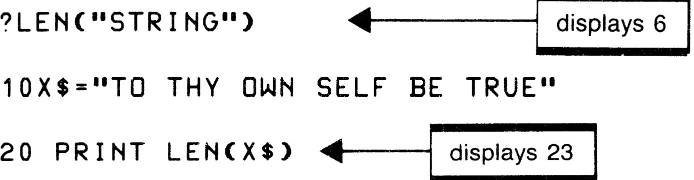
Here is another example:

```
10 T$="[rt rt rt rt rt rt]"
20 INPUT K
30 PRINT LEFT$(T$,K):"WORDS"
```

This code will accept a value K that determines how many cursor rights are used before printing the value WORDS. This technique, or variations of it, can be useful to control the amount of spacing prior to printing a value. Of course other cursor controls or even other values may be substituted to be selected prior to printing.

LEN

The LEN function returns the length of the string argument supplied. For example:



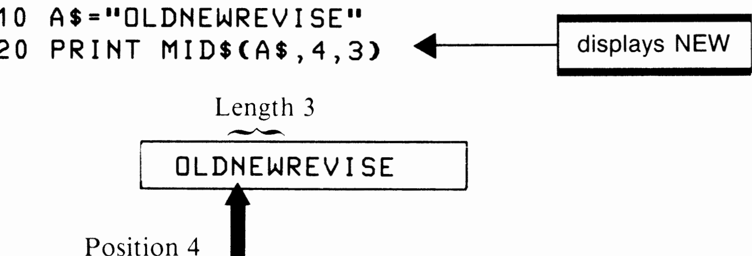
MID$

The MID$ function extracts any specified portion of the character string identified in the argument. The arguments specify the string, the position to begin extraction, and the number of characters to be extracted.

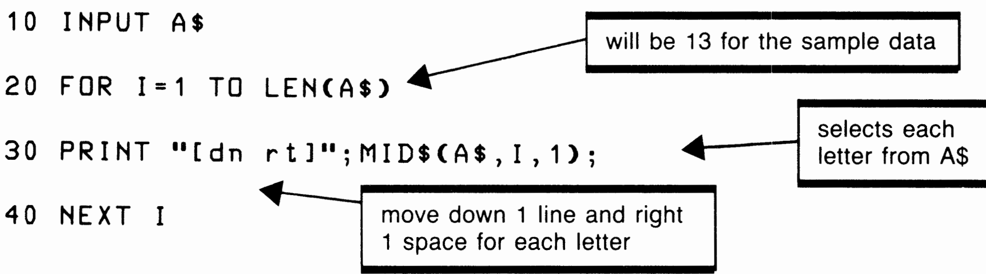
MID$(string, position, length)
—string is the character string to be used in the operation.
—position is the position of the first character to be extracted from the string. This value must be from 1 to 255.
—length is the number of characters to be extracted from 1 to 255.

The original character string is not changed by MID$.

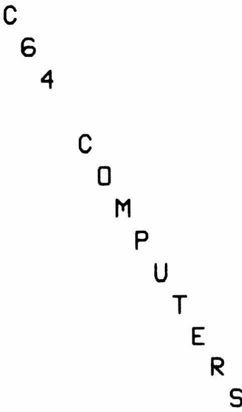
Example:



Suppose we want to input a word and print it diagonally across the screen. Try the following code.



If you enter the input C64 COMPUTERS, the following output will be displayed:



MID$ is useful in applications where the program needs to look through a string for a specific value. For instance, in a CAI application the student might be required to answer a question such as:

```
WHAT ARE TITAN AND DIONE?
```

The response to this might be

```
TITAN AND DIONE ARE MOONS OF SATURN.
```

This response might be processed by the following code, which looks for the word MOONS as part of a correct response.

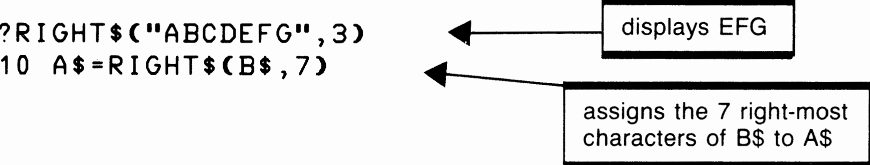
```
100 INPUT X$
110 FOR I=1 TO LEN(X$)-4
120 IF MID$(X$,I,5)="MOONS" THEN 200
130 NEXT I
```

The FOR loop in this code moves through the X$ string one character at a time, looking at groups of five characters for the string "MOONS". If this is found, the program branches to 200, where an affirmative message might be printed or the program could then look for the word "SATURN" to see if the student associated the moons with the planet Saturn. In any event, the program is looking only for the occurrence of these words and in no way analyzes the context in which they have been used.

RIGHT$

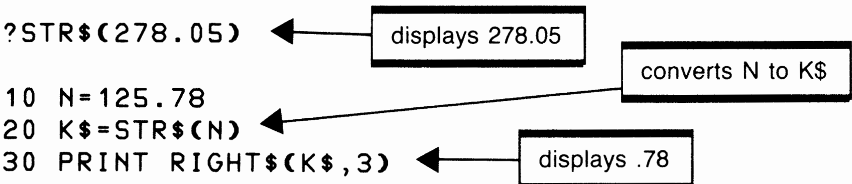
RIGHT$ is similar to the LEFT$ function except that it extracts the right-most characters from the string specified in the argument.

RIGHT$(string, length)
—string is the character string to be used in the operation
—length is the length of the string to be extracted and must be in the range 1 to 255



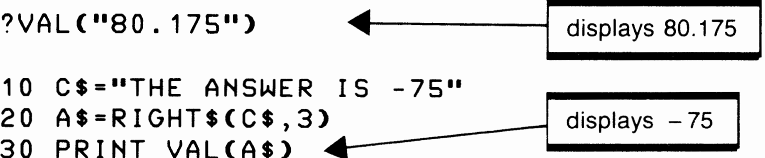
STR$

The STR$ function returns the equivalent string value of the numeric argument. This operation may be useful when a numeric result must be combined with a character string or used in a string operation.



VAL

VAL is the reverse of STR$ since it returns the numeric equivalent of the string argument.



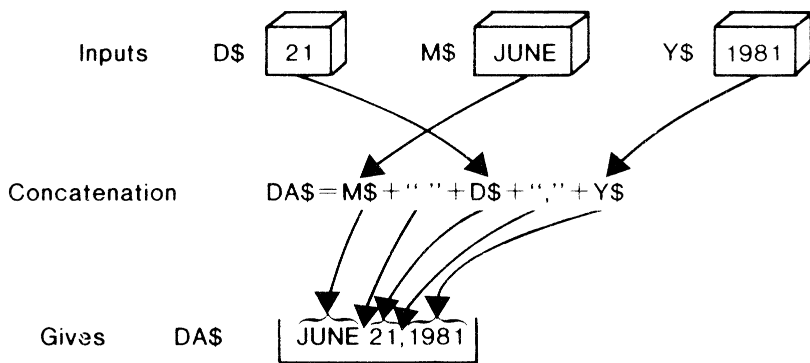
CONCATENATION

Although not a function, concatenation is a useful string operation that combines two or more strings to form a single string of characters. The + sign is used as the concatenation operator but should not be confused with the arithmetic plus sign. Although the same character is used, the results are quite different.

```
?"CONCAT"+"ENATION"
```

displays CONCATENATION

```
10 INPUT "DAY";D$
20 INPUT "MONTH";M$
30 INPUT "YEAR";Y$
40 DA$=M$+" "+D$+","+Y$
50 PRINT DA$
```



An interesting change to this program in statement 50 concatenates the reverse control character to DA$ as follows:

```
50 PRINT "[rvs]"+DA$
```

This simple change causes all of the contents of DA$ (the date) to be printed in reverse characters. Why not try the same idea while using the color codes?

PLOTTING GRAPHS

Drawing graphs on the C-64 presents an interesting challenge for the programmer. Normally if a graph is displayed as its values are calculated, it will come out sideways on the screen and you will need to twist your head sideways to read it. Using a technique developed by Robert Barrett (*Creative Computing*, April 1979), the program that follows displays correctly and is much easier to read. This shift is made by first storing the points of the graph in an array and then displaying them horizontally.

For example, if the function SIN(X) is computed for values of X between 0 and 180, the program would produce the output shown in figure 6.7.

Figure 6.8 is the flowchart for the solution to this problem and figure 6.9 shows the program. Notice that lines 140 to 160 in the program show three different functions. Currently 150 and 160 are Remark statements and so do not affect the program's running. To use another function, it is necessary to change line 140 to a REMARK and remove the REM from the function you want to try.

Function S in line 170 is a scaling function that scales each value of Y to fit within the range of positions available for plotting the graph. The value of X can vary widely, so line 220 computes the position of the Y-axis along the X-axis.

Lines 240 to 280 compute values of Y to determine minimum and maximum for the purpose of scaling the final results. Next, lines 310 to 370 compute the values of Y and store them in the array G at the appropriate intersection of X and Y. Finally, lines 400 to 520 plot the graph and print its coordinates using a combination of POKEs and PRINT statements.

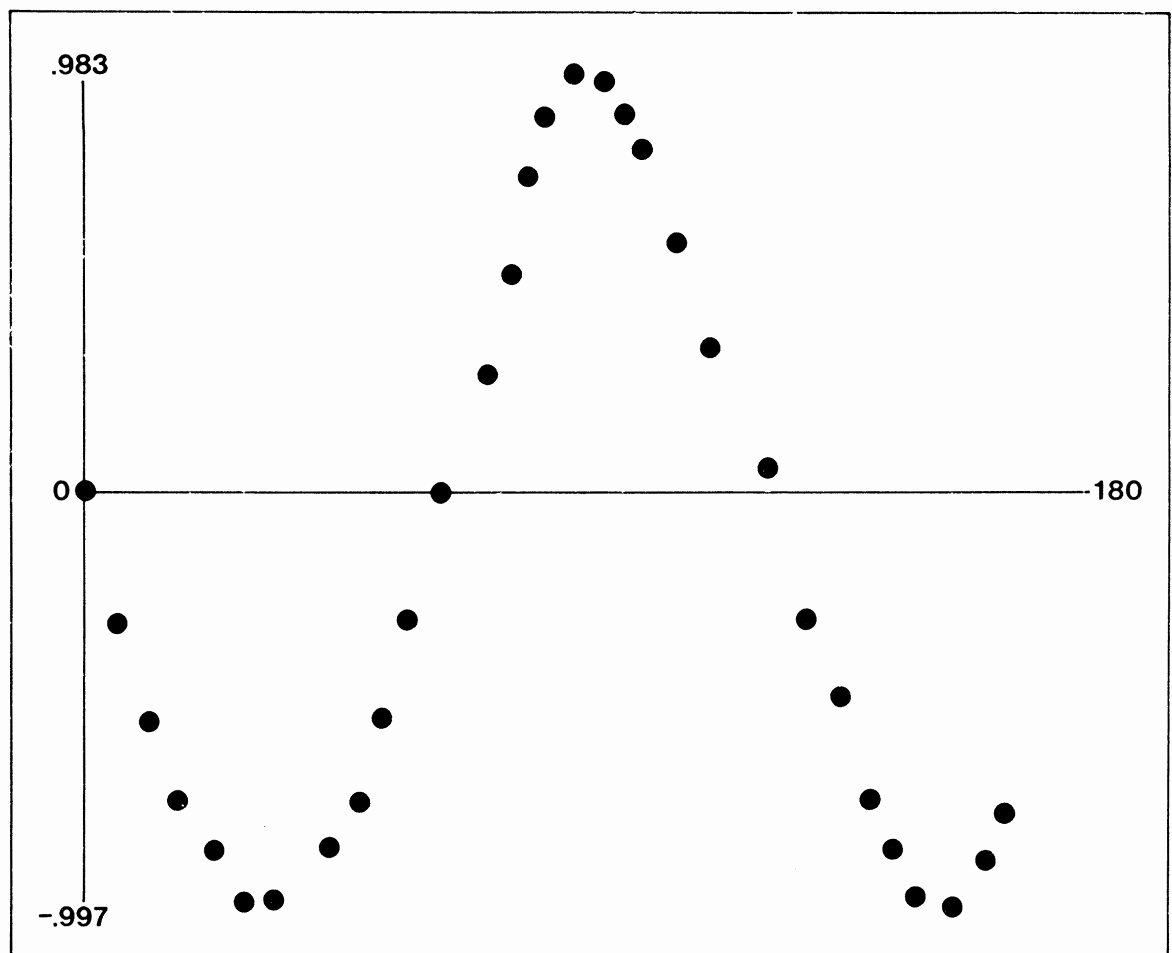


Figure 6.7 Plotting SIN(X) between 0 and 180

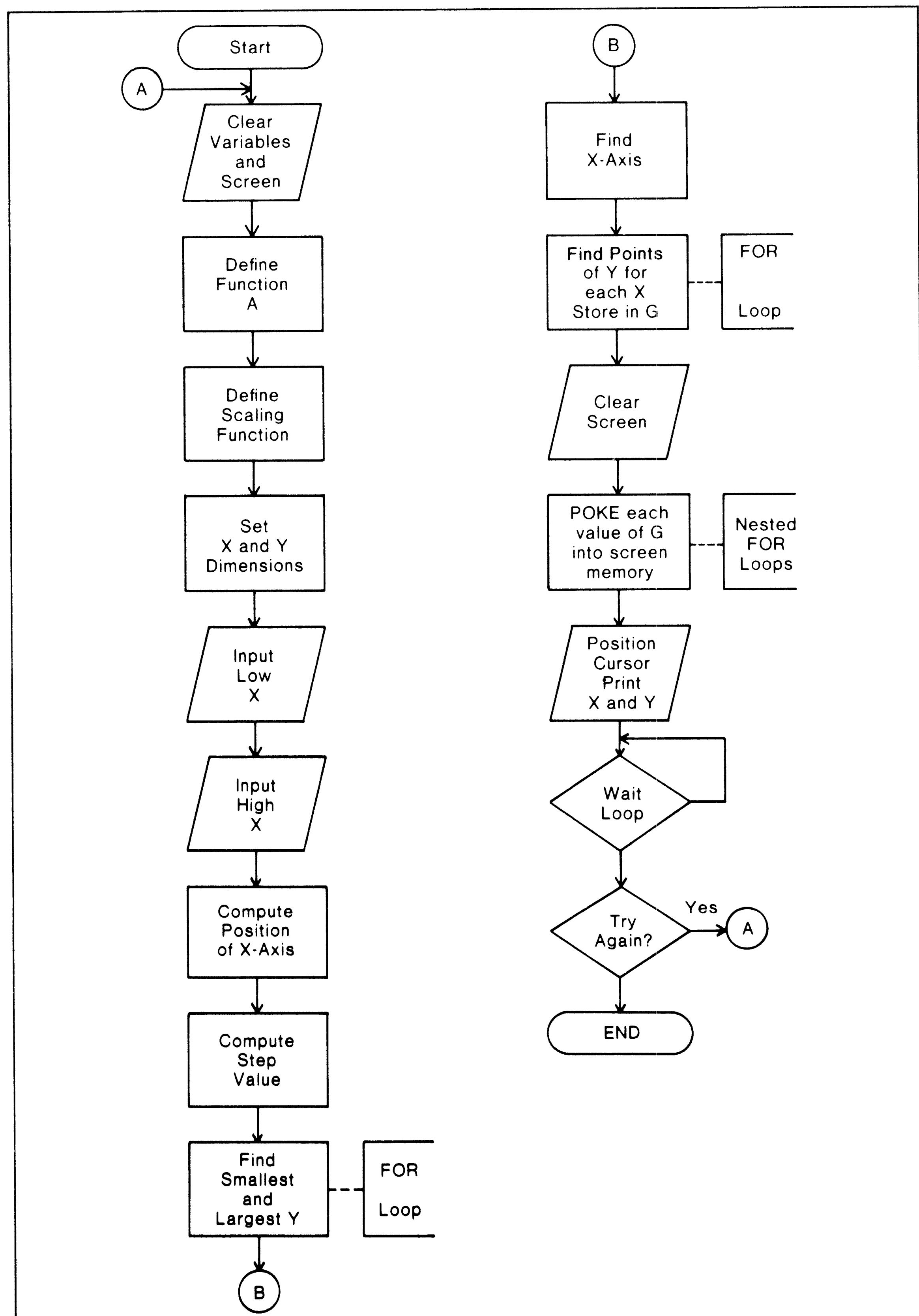
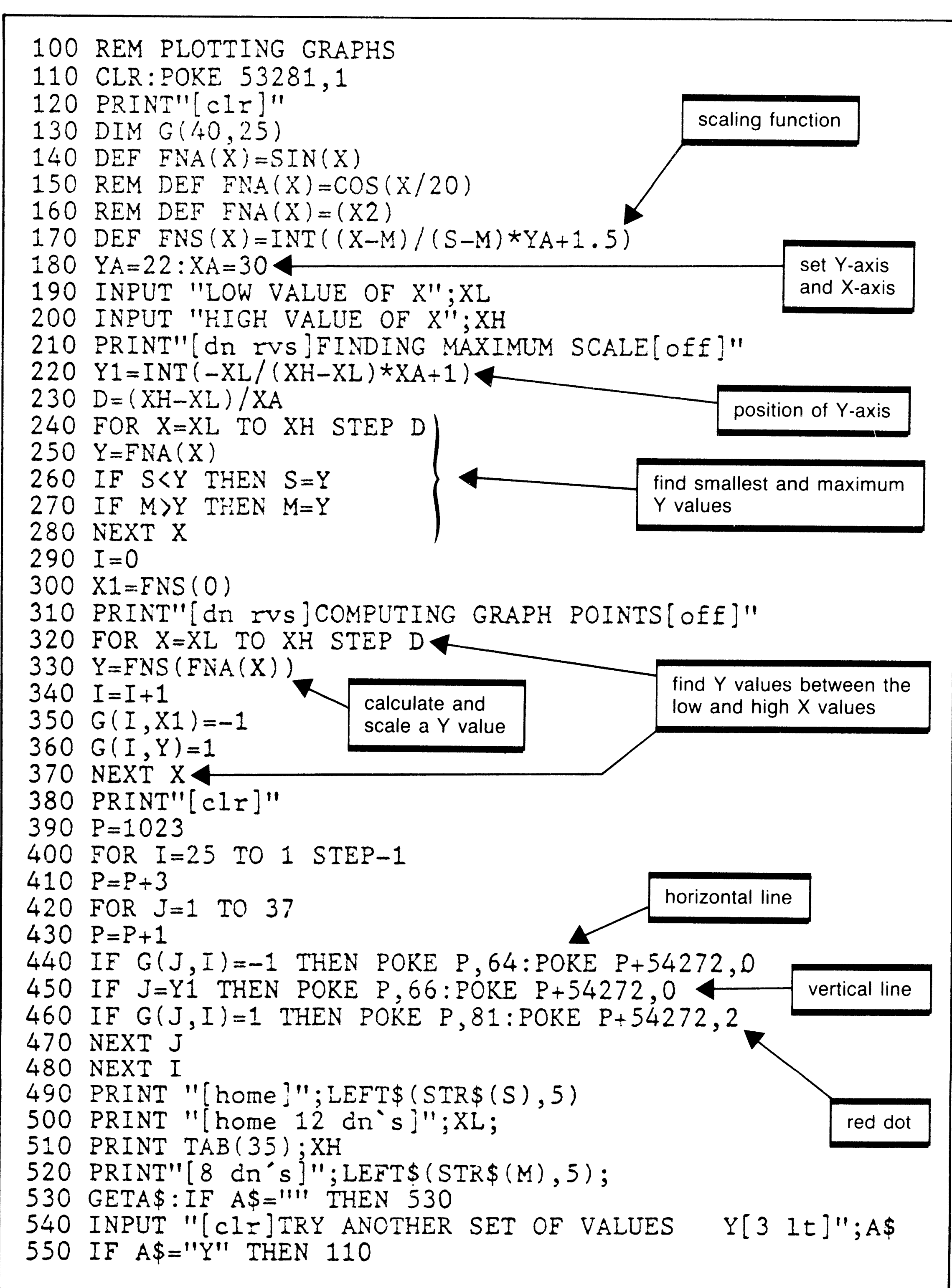


Figure 6.8 Plotting a graph flowchart



```
100 REM PLOTTING GRAPHS
110 CLR:POKE 53281,1
120 PRINT"[clr]"
130 DIM G(40,25)
140 DEF FNA(X)=SIN(X)
150 REM DEF FNA(X)=COS(X/20)
160 REM DEF FNA(X)=(X2)
170 DEF FNS(X)=INT((X-M)/(S-M)*YA+1.5)
180 YA=22:XA=30
190 INPUT "LOW VALUE OF X";XL
200 INPUT "HIGH VALUE OF X";XH
210 PRINT"[dn rvs]FINDING MAXIMUM SCALE[off]"
220 Y1=INT(-XL/(XH-XL)*XA+1)
230 D=(XH-XL)/XA
240 FOR X=XL TO XH STEP D
250 Y=FNA(X)
260 IF S<Y THEN S=Y
270 IF M>Y THEN M=Y
280 NEXT X
290 I=0
300 X1=FNS(0)
310 PRINT"[dn rvs]COMPUTING GRAPH POINTS[off]"
320 FOR X=XL TO XH STEP D
330 Y=FNS(FNA(X))
340 I=I+1
350 G(I,X1)=-1
360 G(I,Y)=1
370 NEXT X
380 PRINT"[clr]"
390 P=1023
400 FOR I=25 TO 1 STEP-1
410 P=P+3
420 FOR J=1 TO 37
430 P=P+1
440 IF G(J,I)=-1 THEN POKE P,64:POKE P+54272,0
450 IF J=Y1 THEN POKE P,66:POKE P+54272,0
460 IF G(J,I)=1 THEN POKE P,81:POKE P+54272,2
470 NEXT J
480 NEXT I
490 PRINT "[home]";LEFT$(STR$(S),5)
500 PRINT "[home 12 dn`s]";XL;
510 PRINT TAB(35);XH
520 PRINT"[8 dn´s]";LEFT$(STR$(M),5);
530 GETA$:IF A$="" THEN 530
540 INPUT "[clr]TRY ANOTHER SET OF VALUES   Y[3 lt]";A$
550 IF A$="Y" THEN 110
```


Figure 6.9 Program to plot graphs

CONTROLLING DECIMAL POSITIONS

Earlier in this chapter we saw how to limit the number of decimal positions to no more than two. However, a number with one or even no decimal positions would not align exactly with numbers containing two or more decimal positions. This means the decimal points will not align in the straight column that may be desirable for some applications, such as accunting or financial reports.

The technique discussed here will limit all decimal numbers to two decimal positions regardless of the number of decimals in the original number. With a slight modification to the program, the results could control output to any number of decimals required. Figure 6.10 shows a variety of decimal numbers and indicates the results required of the program.

Original Number	Desired Results
1	1.00
2.5	2.50
125	125.00
.05	0.05
.125864	0.12
79.16	79.16
-14.5	-14.50
-.458	-0.45

Figure 6.10 Editing to two decimal places

In this example note that every number results in two and only two decimal positions. In some cases, such as the value 1, two decimal positions (.00) are appended to the number, while in other cases (.125864), decimal positions are truncated to result in two decimals.

Figure 6.11 shows the steps followed to take a number and adjust its decimal positions to two. First the number is separated into the integer (left) and fractional (right) parts. The fractional part is then adjusted to two digits, adding a decimal point if necessary and then concatenated back onto the integer part to create the final result.

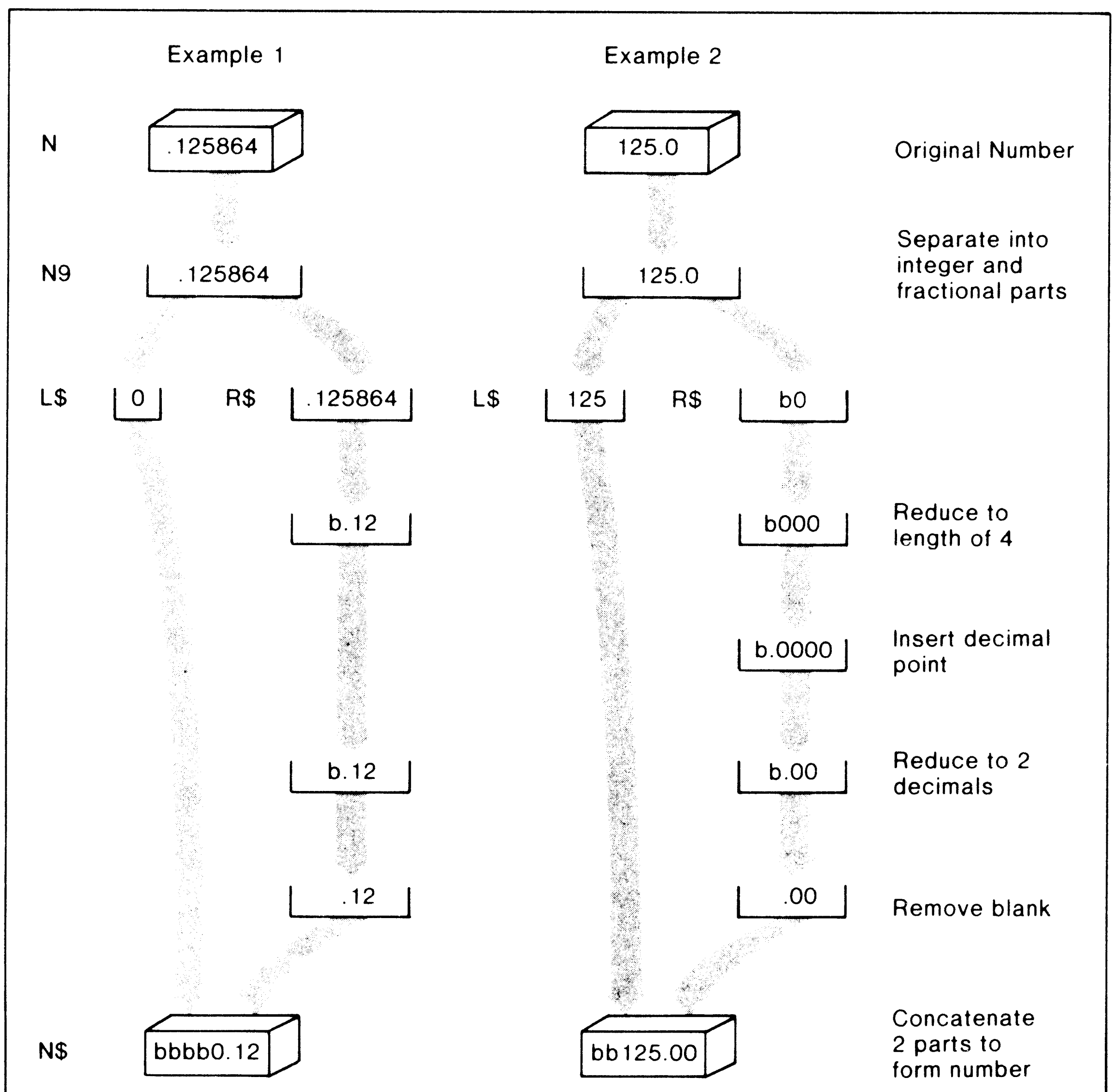


Figure 6.11 Converting to two decimal places

Figure 6.12 shows the flowchart for this problem and figure 6.13 is the complete program. In the programs, statements 200 to 330 are the subroutine while statements 100 to 190 are a driver routine.

Driver Routines

A driver routine is often used to test a part of a program without having the complete program in the computer at the time of testing. For instance, if we were writing a complete home finances program, the decimal subroutine would be required. Using a driver initially lets us try the solution of limiting decimal points for a variety of values without concern for the hundreds of other statements the full-scale program may eventually have. When we are satisfied with the solution for this subroutine, then it may be combined with the full program.

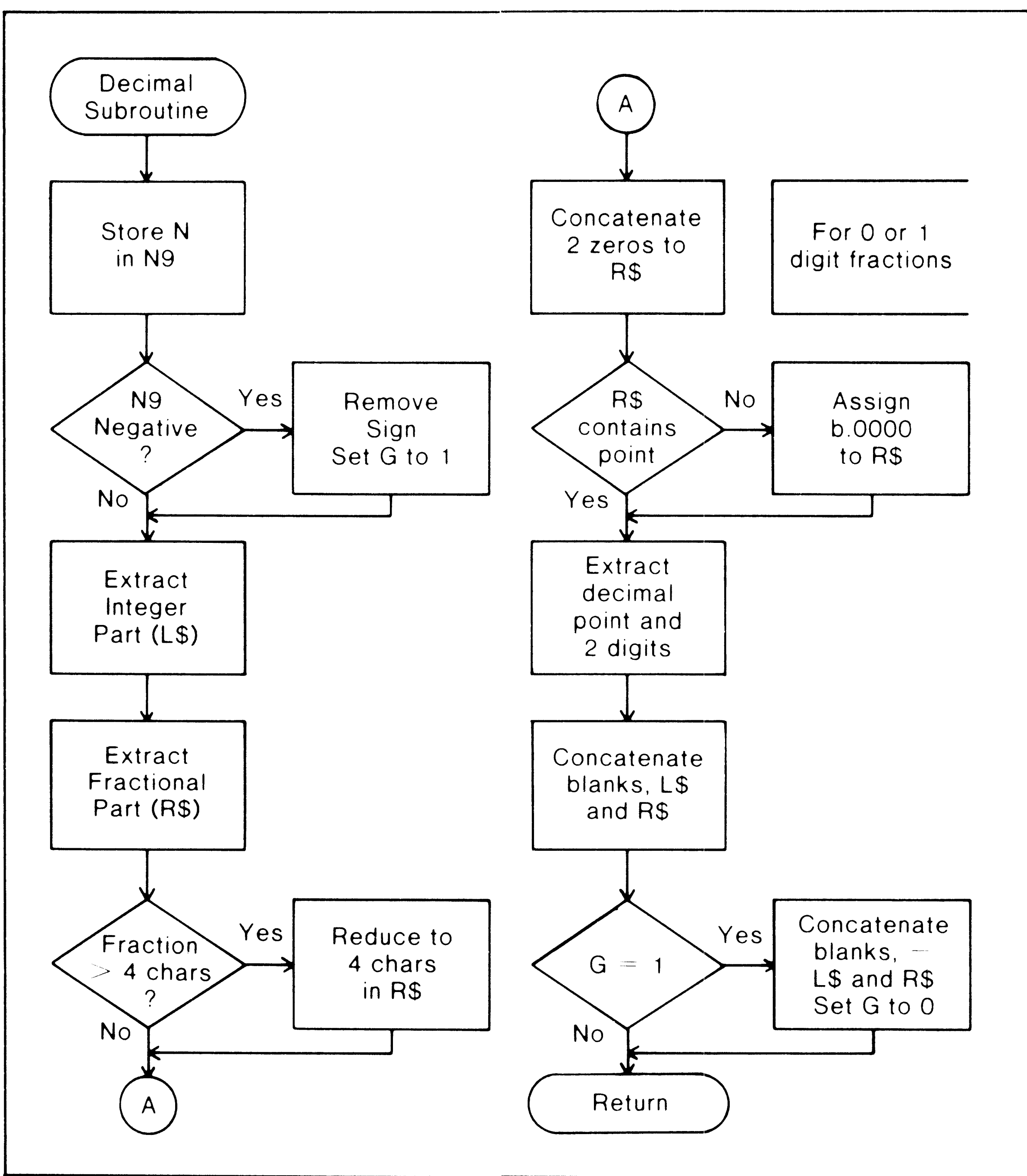


Figure 6.12 Flowchart for converting to two decimals

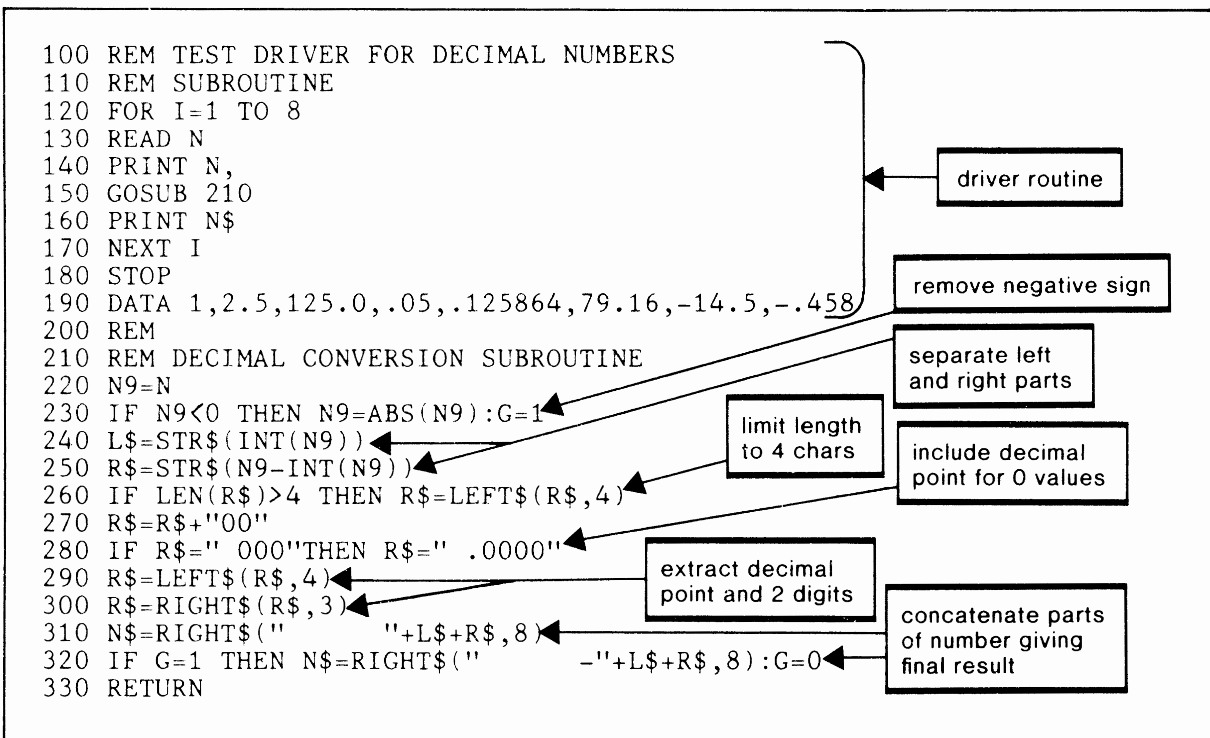


Figure 6.13 Program for converting to two decimals

CAI CHESS AND PROGRAM GENERALIZATION

Computer Assisted Instruction has become an important aspect of education. Although the program discussed here presents some initial instruction in chess, it is generalized so that you may change the data to instruct in any subject that interests you. Figure 6.14 shows the arrangement of the data for this program.

The approach taken here is to develop the program logic in such a way that it is essentially independent of the subject to be presented. Rather than using a statement such as

```
100 PRINT "A RANK IS A HORIZONTAL ROW OF SQUARES"
```

the string is placed in a data statement and then read and printed.

Although this approach at first may seem more involved, it actually means a lot less work in the long run. The reason for this is that only one READ and PRINT is necessary regardless of the amount of data to be read. Therefore the CAI instruction may be as long as needed without lengthening the program itself. Only the data length is increased.

The Data

Each set of data begins with a numeric value (called N in the program) that indicates the number of lines of instruction to follow. This value is 16 in the sample set of data, indicating there are 16 lines of instruction including a blank line. The actual lines of text then follow in subsequent data statements for the program to read into A$.

Another number that tells the program how many questions have been supplied for this text follows the text material. In the example, the value is 3 and will be followed by a set of three questions and their answers. The 3 representing the number of questions is read into the variable NQ.

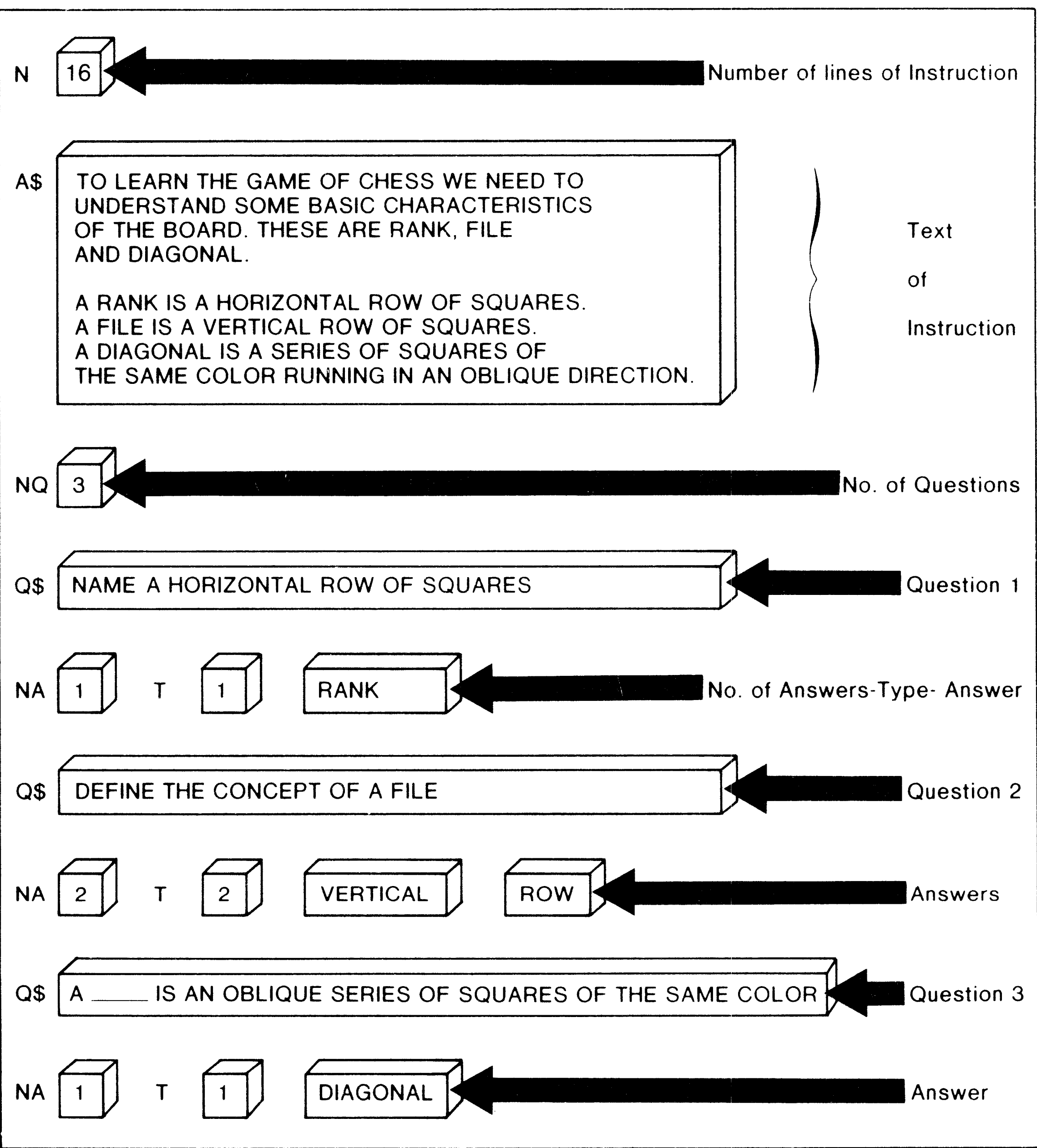


Figure 6.14 One data set for CAI chess

Now we have the first question:

```
NAME A HORIZONTAL ROW OF SQUARES
```

This question is read into Q$ and is followed by a 1, which defines the number of answers (NA). Next is a number that defines the type of answer to follow. There are three different types:

1—Single answer
2—AND type answer
3—OR type answer

The first type requires a single answer that could be either a word or phrase. In the first question the answer is RANK. The program will search the answer given by the user to determine if it contains the word RANK. If it does, the answer is considered to be correct.

A type 2 answer indicates that more than one word is required in the answer and that each word must be present. In the second question:

```
DEFINE THE CONCEPT OF A FILE
```

both words VERTICAL and ROW must be present in the answer for it to be correct. The logic of this evaluation is based on the Boolean "and" truth table.

Type 3 is based on the Boolean "or" and therefore requires that only one of several possible answers be submitted. The question:

```
NAME A PIECE THAT HAS A TWIN
```

has three possible correct answers. ROOK, BISHOP, or KNIGHT are the possibilities. If any one of these answers is given, the response is considered correct. The answer given could be in the form of a single word, such as:

```
BISHOP
```

or it could be a complete sentence. For example:

```
A ROOK IS A PIECE WITH A TWIN.
```

In either case the program would scan the answer for the correct word within the answer string.

The Program

Random Responses Programs used for CAI typically need to respond to both correct and incorrect answers from the user. Of course this can be a simple matter of displaying "correct" or "incorrect," but this tends to become monotonous after a few answers. To add a little variety and spice to your answers, it is possible to use the random function to select one response from a collection of them.

Figure 6.14 shows how a set of four correct answers are set up in array CR$ to be selected for a response to a correct answer. The random function generates a value between 1 and 4, which then acts as a subscript to extract the response from the array.

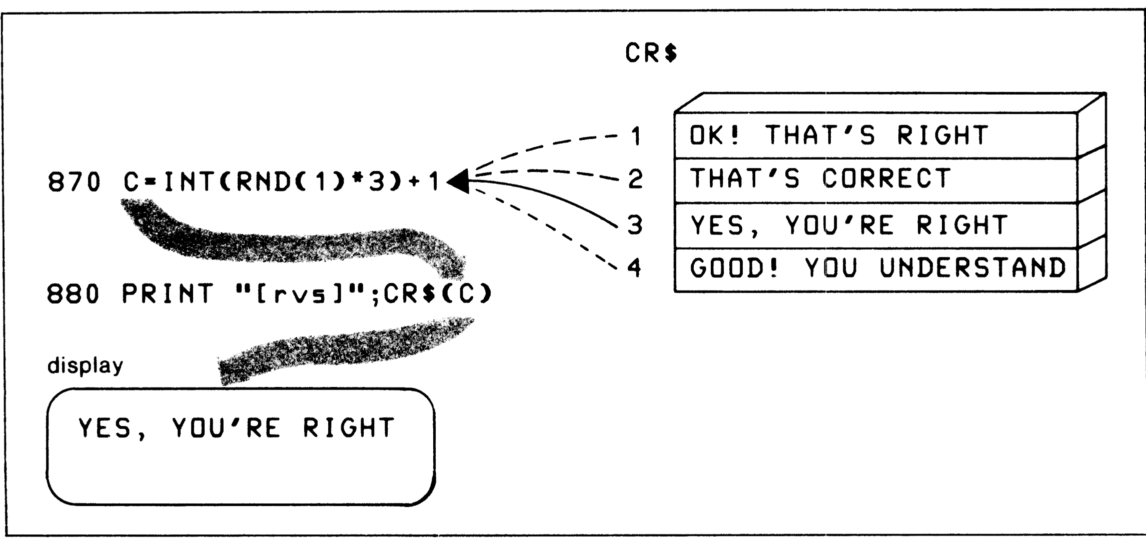


Figure 6.14 Selecting a random correct response

Instring Search Some languages have a statement with the capability of searching a string variable for the presence of a particular word or set of characters. This is called an Instring search. Since C-64 BASIC does not have this feature, it is necessary to write a subroutine that can accomplish the same type of search. The need for this approach becomes apparent when we consider the types of responses we are likely to get from this program's user. Rather than give single-word answers, the user can respond with a sentence, which is a more natural means of communication or, if preferred, with a single word or phrase.

Figure 6.15 shows how to implement a search for the string IN$ within the longer string A$. If IN$ is found, the Found Switch (FS) is set to 1, indicating the presence of IN$ in A$. Otherwise the string is not found and FS remains a 0 value.

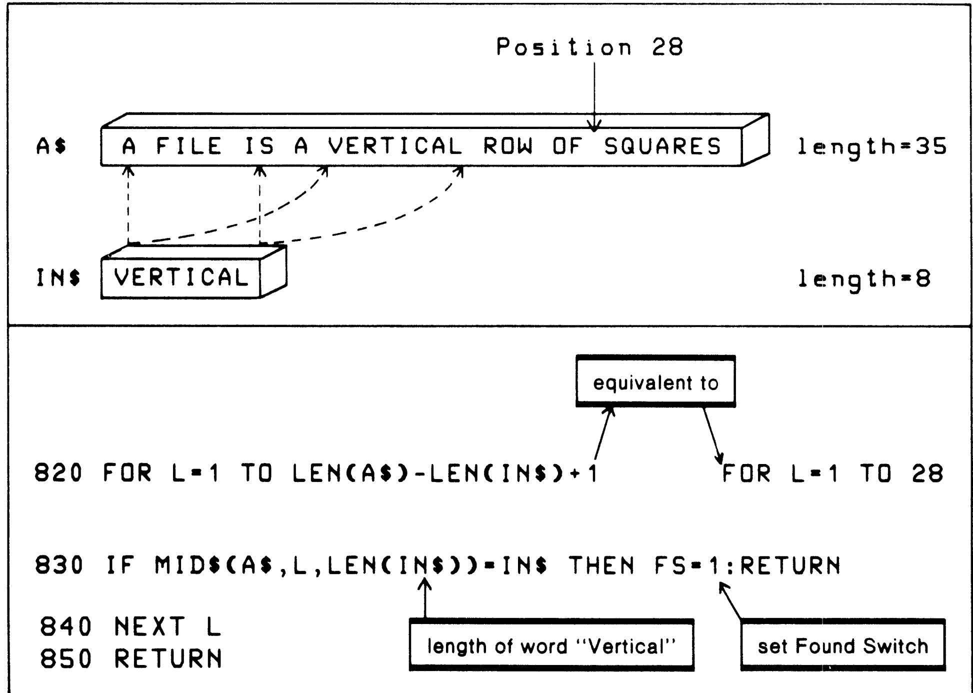


Figure 6.15 Instring search subroutine

Checking AND-Type Responses When an AND-type response is required, it means there are two or more words expected in the answer. The answer string will be examined for the presence of each of these words, and if all of them are found the answer is considered correct. Figure 6.16 demonstrates the solution.

The array A1$ contains each of the words to be searched for in the input string using subroutine 810 for the instring search. For each match found (FS = 1), the search moves on to look at the next word at A1$(M). If the word was not found in the input string (FS = 0), then subroutine 910 is called to produce the incorrect answer response. Finally, if all words are found in the input string, the loop terminates and subroutine 860 produces the correct answer response.

Basically the OR type of response functions in the same way. The primary difference is that only one of the words in array A1$ needs to be found in the answer string.

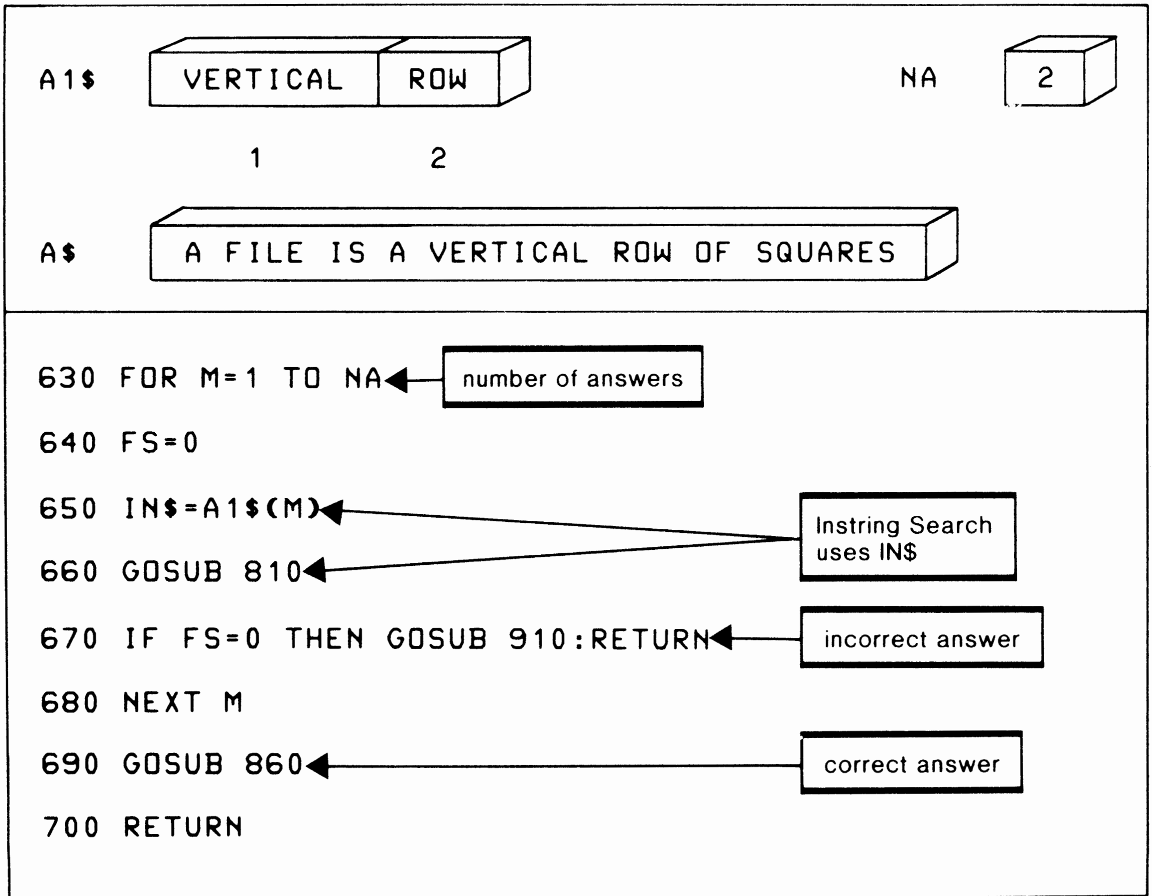
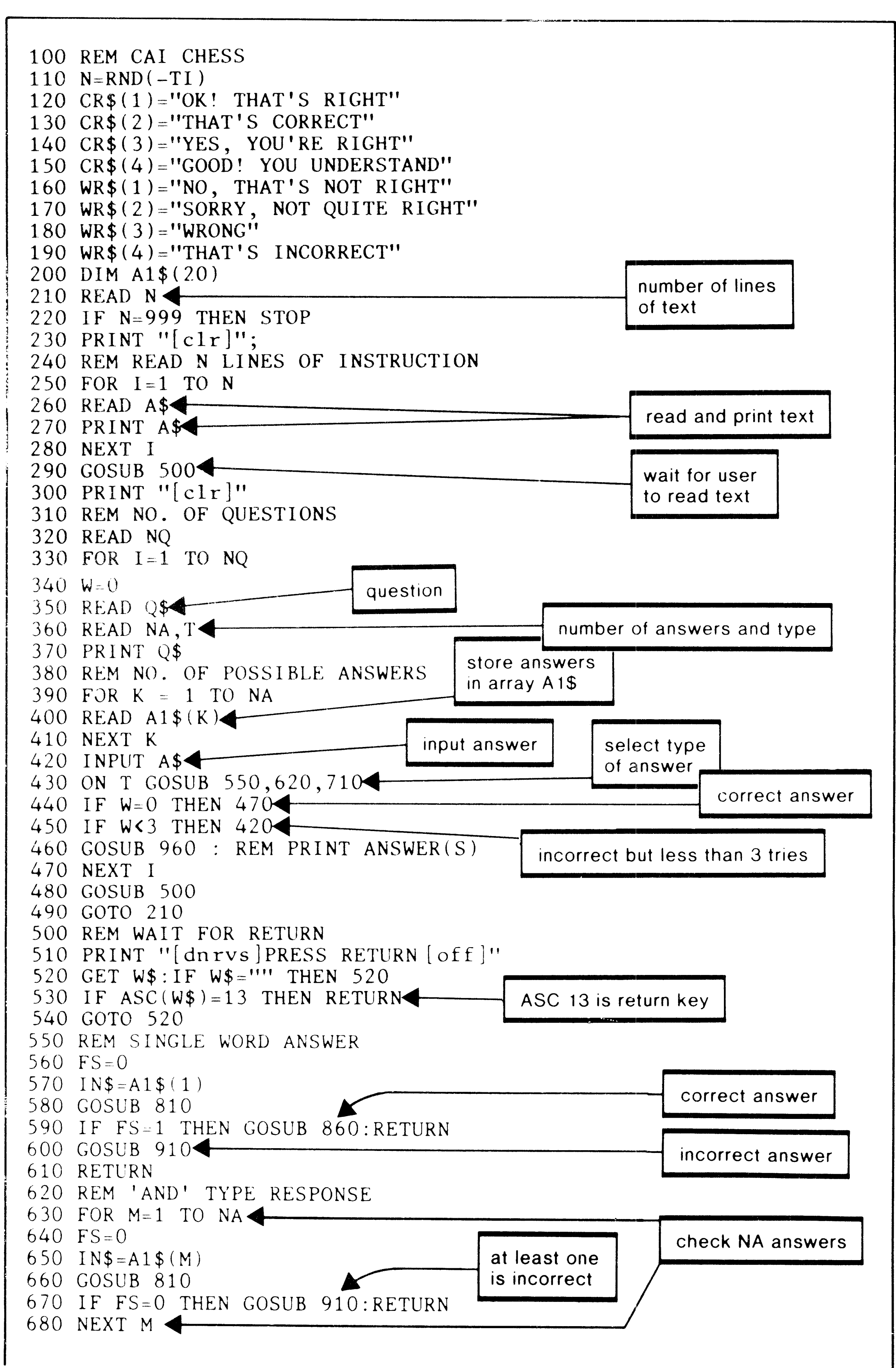


Figure 6.16 Checking AND type responses



```
100 REM CAI CHESS
110 N=RND(-TI)
120 CR$(1)="OK! THAT'S RIGHT"
130 CR$(2)="THAT'S CORRECT"
140 CR$(3)="YES, YOU'RE RIGHT"
150 CR$(4)="GOOD! YOU UNDERSTAND"
160 WR$(1)="NO, THAT'S NOT RIGHT"
170 WR$(2)="SORRY, NOT QUITE RIGHT"
180 WR$(3)="WRONG"
190 WR$(4)="THAT'S INCORRECT"
200 DIM A1$(20)
210 READ N
220 IF N=999 THEN STOP
230 PRINT "[clr]";
240 REM READ N LINES OF INSTRUCTION
250 FOR I=1 TO N
260 READ A$
270 PRINT A$
280 NEXT I
290 GOSUB 500
300 PRINT "[clr]"
310 REM NO. OF QUESTIONS
320 READ NQ
330 FOR I=1 TO NQ
340 W=0
350 READ Q$
360 READ NA,T
370 PRINT Q$
380 REM NO. OF POSSIBLE ANSWERS
390 FOR K = 1 TO NA
400 READ A1$(K)
410 NEXT K
420 INPUT A$
430 ON T GOSUB 550,620,710
440 IF W=0 THEN 470
450 IF W<3 THEN 420
460 GOSUB 960 : REM PRINT ANSWER(S)
470 NEXT I
480 GOSUB 500
490 GOTO 210
500 REM WAIT FOR RETURN
510 PRINT "[dnrvs]PRESS RETURN [off]"
520 GET W$:IF W$="" THEN 520
530 IF ASC(W$)=13 THEN RETURN
540 GOTO 520
550 REM SINGLE WORD ANSWER
560 FS=0
570 IN$=A1$(1)
580 GOSUB 810
590 IF FS=1 THEN GOSUB 860:RETURN
600 GOSUB 910
610 RETURN
620 REM 'AND' TYPE RESPONSE
630 FOR M=1 TO NA
640 FS=0
650 IN$=A1$(M)
660 GOSUB 810
670 IF FS=0 THEN GOSUB 910:RETURN
680 NEXT M
```


Figure 6.17 CAI chess program

```
690 GOSUB 860
700 RETURN
710 REM 'OR' TYPE RESPONSE
720 FOR M=1 TO NA
730 FS=0
740 IN$=A1$(M)
750 GOSUB 810
760 IF FS=1 THEN GOSUB 860:RETURN
770 NEXT M
780 GOSUB 910
790 RETURN
810 REM INSTRING SEARCH
820 FOR L=1 TO LEN(A$)-LEN(IN$)+1
830 IF MID$(A$,L,LEN(IN$))=IN$ THEN FS=1:RETURN
840 NEXT L
850 RETURN
860 REM CORRECT RESPONSE TO QUESTION
870 C=INT(RND(1)*3)+1
880 PRINT"[rvs]";CR$(C)
890 W=0
900 RETURN
910 REM INCORRECT RESPONSE TO QUESTION
920 C=INT(RND(1)*3)+1
930 PRINT"[rvs]";WR$(C)
940 W=W+1
950 RETURN
960 REM 3 ERRORS - PRINT CORRECT ANS
970 PRINT "YOUR ANSWER SHOULD INCLUDE THE TERMS"
980 FOR K=1 TO NA
990 PRINT A1$(K);" ";
1000 NEXT K
1010 PRINT
1020 GOSUB 500
1030 RETURN
1040 DATA 16
1050 DATA "TO LEARN THE GAME OF"
1055 DATA "CHESS WE NEED TO"
1060 DATA "UNDERSTAND THE BASIC"
1065 DATA "CHARACTERISTICS OF"
1070 DATA "THE BOARD WHICH ARE"
1075 DATA "RANK, FILE AND"
1080 DATA "DIAGONAL.
1090 DATA " "
1100 DATA "[rvs]RANK[off] IS A HORIZONTAL"
1105 DATA "ROW OF SQUARES."
1110 DATA "A [rvs]FILE[off] IS A VERTICAL"
1115 DATA "ROW OF SQUARES."
1120 DATA "[rvs]DIAGONAL[off] IS A SERIES"
1125 DATA "OF SQUARES THE SAME"
1130 DATA "COLOR RUNNING IN AN"
1135 DATA "OBLIQUE DIRECTION"
1140 DATA 3
1150 DATA "NAME A HORIZONTAL ROW OF SQUARES"
1160 DATA 1,1,"RANK"
1170 DATA "DEFINE THE CONCEPT OF A FILE"
1180 DATA 2,2,"VERTICAL","ROW"
1190 DATA "A ______ IS AN OBLIQUE SERIES OF SQUARES OF  THE SAME COLOR"
1200 DATA 1,1,"DIAGONAL"
1210 DATA 10
1220 DATA "THE CHESS GAME USES"
1225 DATA "16 BLACK AND 16 WHITE"
```

check NA answers for first correct (lines 720, 770)

one is correct (line 750)

none were correct (line 780)

randomize selection of response (lines 870, 920)

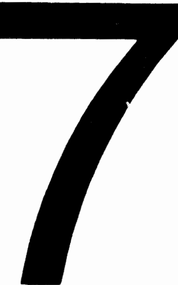
Figure 6.17 (continued)

```
1230 DATA "PIECES WITH THESE"
1240 DATA "NAMES AND NUMBERS"
1250 DATA " "
1260 DATA "             1 - KING"
1270 DATA "             1 - QUEEN"
1280 DATA "             2 - ROOKS"
1290 DATA "             2 - BISHOPS"
1300 DATA "             2 - KNIGHTS"
1310 DATA "             8 - PAWNS"
1320 DATA 6
1330 DATA "HOW MANY BLACK PIECES ARE USED IN CHESS"
1340 DATA 1,1,"16"
1350 DATA "NAME EACH PIECE.(TYPE EACH NAME FOLLOWED BY A SPACE)"
1360 DATA 6,2,"KING","QUEEN","ROOK","BISHOP","KNIGHT","PAWN"
1370 DATA "HOW MANY PIECES ARE WHITE KINGS"
1380 DATA 1,1,"1"
1390 DATA "HOW MANY PIECES ARE BLACK QUEENS"
1400 DATA 1,1,"1"
1410 DATA "NAME A PIECE THAT HAS A TWIN"
1420 DATA 3,3,"ROOK","BISHOP","KNIGHT"
1430 DATA "HOW MANY WHITE PAWNS ARE THERE"
1440 DATA 1,1,"8"      <-- additional material may be inserted here
1450 DATA 999,"END-OF-DATA"
^-- renumber end line
```

Figure 6.17 (continued)

REVIEW QUESTIONS—CHAPTER 6

1. What is the benefit of using a GET statement instead of the INPUT statement? Why is a loop necessary with the GET?
2. What are some of the limitations of the GET compared to the INPUT statement?
3. Consider how you would write a module to get more than one character of input using the GET statement. How would you handle such corrections as using the delete or cursor keys?
4. What are some of the benefits of using the ON group of statements? Discuss the pros and cons of ON GOSUB versus ON GOTO.
5. Describe the purpose of the POKE instruction. What is the only type of data that may be POKEd?
6. Write some BASIC code using POKEs that will draw a (1) horizontal line, (2) vertical line.
7. What does the PEEK statement do? How is it different from POKE?
8. Write a short program to cause a circle to move slowly across the screen from left to right.
9. What is a function? Name some. In general, what is the difference between an arithmetic function and a string function? Give a specific example.
10. Driver routines are sometimes used in programming. What is a driver routine and how is it used?
11. Using the CAI chess program as a model, write a program to give instructions in an interactive mode on a subject you are familiar with.



Interacting with the User of Your Program

Now that you are developing your skill as a C-64 programmer you may soon find that you might frequently write programs that are intended for the use of others. Your programs may be intended for your friends and neighbors, directed to a student audience, or to your colleagues at work, or for use by others in your home or business. It is therefore crucial that you develop programs that interact with the user at a suitable level and with an effectiveness that permits ease and simplicity of operation.

Determining the level of the program's interaction should involve consideration of both the language and technical ability of the program's user and employ prompting and/or dialogue consistent with this level. Effectiveness of operation includes the methods used to communicate with the user. These may include queries, prompting, codes, menus, form filling, and so on. Suitable choices often determine a person's success in using the program and a willingness to use it again.

USER LEVEL

Your program is usually intended for a specific audience. In general, you should assess the age, education, training, intelligence, and motivation of the user. After a general assessment, it is useful to place the user in one of the following three categories: casual, trained, or programming skills.

1. Casual

This is someone who uses a microcomputer infrequently and generally has no training in computers. All first-time users fall into this category, which is typical of most C-64 owners. So do many students and educators.

2. Trained

A trained owner is a person who has been given formal or informal training in the use of computers. This training might be limited to the use of a particular program or be as broad as a computer literacy course.

3. Programming Skills

This is the most sophisticated user. He or she will have done some programming and be familiar with programming terminology and language syntax. Since you have been studying programming in this book, you are in this category.

In addition to these three categories, a user may operate the program on two different levels. James Martin, writer, researcher, and seminar leader on the application of computers for business, defines these levels as active or passive.

An active operator is one who initiates program action by entering commands. This level is typical of games where the user enters values that might control the direction and speed of a ball on the screen.

A passive operator is one who acts on the program's initiative, for instance, when a program asks for the user's name. Many programs will use both active and passive interaction.

USER DIALOGUES

Prompting

In a sense, any information requiring a user response represents prompting. However, our interest here is the occasion when the program displays a statement or question and then waits for the user to type a response. Prompting is appropriate for all levels of users, but the language of the prompt should be directed to the specific user level. In any case, be courteous. A prompt such as

```
PLEASE ENTER NAME?
```

is much better than a curt

```
NAME?
```

This kind of prompt is usually implemented as follows:

```
100 PRINT "PLEASE ENTER NAME";
110 INPUT N$
```

or more directly and efficiently as

```
100 INPUT "PLEASE ENTER NAME";N$
```

With this type of prompt you may get a variety of responses, such as

```
JOHN
JOHN SMITH
SMITH
```

In some programs this doesn't matter, but if it does make a difference the prompt should be more specific.

```
PLEASE ENTER SURNAME
```

Some prompts may have simple alternatives, such as YES or NO, TRUE or FALSE, ADD or SUBTRACT. In these situations, the prompt should indicate which responses are expected.

```
100 PRINT "DO YOU WANT MULTIPLE CHOICE (YES/NO)"
110 INPUT A$
120 IF A$="YES" THEN 200
```

This prompt indicates clearly that a yes or no answer is expected. Some programs also permit the user to respond with just the first letter of the response. This is done by extracting the first letter of A$ using the LEFT$ function and then testing for a "Y" or "N." With this method, single-letter responses as well as the complete word are acceptable.

```
100 PRINT "WOULD YOU LIKE MULTIPLE CHOICE(YES/NO)"
110 INPUT A$
120 IF LEFT$(A$,1)="Y" THEN 200
```

Some prompts use data from previous operations in the program. An example of this is a program that generates drill and practice questions for addition. Prior to the prompt the program generates two values (A and B), which the student then adds mentally or on paper before entering the answer. This prompt might then use both the PRINT and INPUT statements.

```
100 PRINT "[clr dn dn dn dn dn]";
110 PRINT "WHAT IS THE SUM OF";A;" +";B
120 INPUT S
```

Note the use of line 100, which clears the screen and moves the cursor down five lines before printing the prompt. This action avoids any distraction from previous questions that would otherwise remain on the screen.

A more creative solution to this problem might be to print the prompt in the form of a traditional addition question, as follows:

```
100 PRINT "[clr dn dn dn dn dn]";
110 PRINT "[rt rt rt rt]";A
120 PRINT "[rt rt rt]+";B
130 PRINT "[rt rt rt]__"  <------ (__ is a shift C)
140 INPUT "[rt rt rt]";S
```

If A is the value 10 and B is 15, this code displays the following:

```
 10
 15
 __
  ?
```

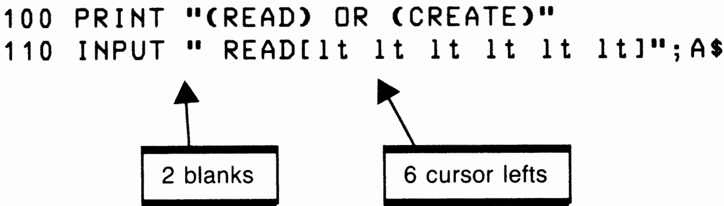
Although this solution takes a lot more work, the results are far superior to previous methods and show the power of the C-64 in implementing an effective solution.

DEFAULT RESPONSES

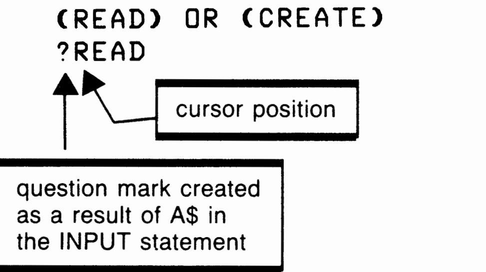
In many applications a choice is given the user, but the response can often be anticipated. A program that normally reads a file but has an option to create one is an example of when a default would be useful. The prompt might be:

```
(READ) OR (CREATE)
?
```

Here the user must type in either READ or CREATE as a response. Since we expect that READ is the most frequent response, it may become the default. This is done by printing the default value with the prompt and then moving the cursor to the left past the default value.



This is what the display shows:



The six cursor lefts are needed to move past the four letters of READ and the two spaces. The question mark appears in the first space and then the cursor automatically moves one position to the right, leaving the second space for readability. Now if the user wants to use the READ default, only the Return key is pressed. If not, CREATE may be typed to select the alternative.

A complete input module might be the following:

```
100 PRINT "(READ) OR (CREATE)"
110 INPUT "  READ[lt lt lt lt lt lt]";A$
120 IF A$="READ" THEN 200
130 IF A$="CREATE" THEN 400
140 PRINT "[clr]ENTER READ OR CREATE"
150 GOTO 100
```

MENUS

A menu provides a list of alternatives from which the user selects one entry by typing a number or letter. The following code gives an example of this approach:

```
100 PRINT "[clr]SELECT ONE SEARCH"
110 PRINT "1-BY AUTHOR"
120 PRINT "2-BY SUBJECT"
130 PRINT "3-BY KEYWORD"
140 PRINT "4-BY TITLE"
150 INPUT A
160 ON A GOTO 1000,2000,3000,4000
170 GOTO 100
```

The codes 1, 2, 3, 4 translate nicely for use in either an ON—GOTO or ON—GOSUB, which greatly simplifies program analysis of the user's response.

A similar approach uses the first letter of each menu item as the Response key. The letter may be highlighted by turning the Reverse on before the character and off after it is displayed.

```
100 PRINT "[clr]SELECT ONE SEARCH"
110 PRINT "[rvs]A[rvs off]UTHOR"
120 PRINT "[rvs]S[rvs off]UBJECT"
130 PRINT "[rvs]K[rvs off]EYWORD"
140 PRINT "[rvs]T[rvs off]ITLE"
150 INPUT A$
160 FOR I=1 TO 4
170 IF A$=MID$("ASKT",I,1) THEN ON I GOSUB
    1000,2000,3000,4000:I=4
180 NEXT I
190 GOTO 100
```

This approach tends to cause the reversed characters to run together on the screen. A simple solution is to separate each line by placing a PRINT at lines 105,115,125,135, and 145 as follows:

```
105 PRINT
115 PRINT
125 PRINT
135 PRINT
145 PRINT
```

Lines 160 to 180 examine the input character against the string "ASKT" in line 170. The response character will be equal to position I of the string. In other words, letter S will be found when I is 2 and the GOSUB will transfer to subroutine 2000. If I = 4 at the end of line 170 forces the end of the FOR loop without the need for additional looping.

Most menus require only a single letter or number response, as seen in the previous examples. When the INPUT statement is used, two keystrokes are required, one for the character and one for the Return key. One advantage of this is the user may change the character any time before Return is pressed by moving the cursor to the left and retyping the character.

In other cases, a quick response is preferred with a minimum of keying. Here the GET statement comes to our rescue. If the previous example replaces line 150 with

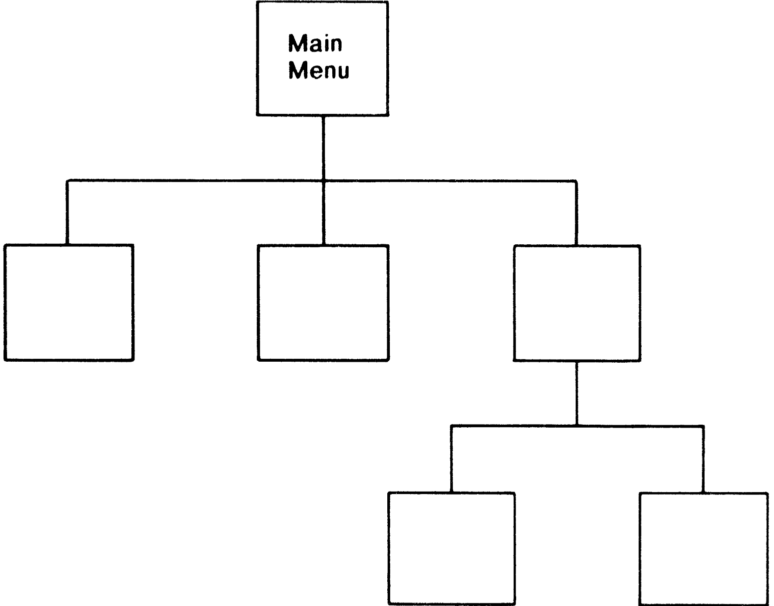
```
150 GET A$:IF A$="" THEN 150
```

the typing of a single letter will be sufficient. This method works only for a single-character response but is unequalled for speed.

MULTILEVEL MENUS

When we are using a screen as the program's medium for communication with the user, there is a limit to the number of items a menu can contain. Usually it is preferable to limit menu items to fewer than 10 if numeric responses are required. With a blank line between each item, this requires at least 19 lines on the screen. However, many applications may require well in excess of 10 items. The procedure for handling many entries is to create a multilevel menu.

The multilevel menu begins with a main menu that branches to several lower-level menus. Lower-level menus may themselves point to even lower levels in a complex application.



These lower-level menus should provide a means of escape in the form of a menu item that points back to the main menu. An example of this application is a budgeting program with a main menu that asks for annual, monthly, or daily budget items. The program then branches to a lower-level menu to itemize particular entries in one of these three categories.

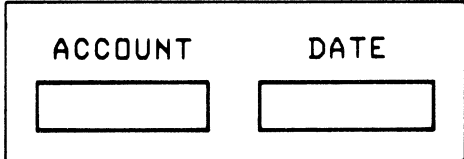
```
100 PRINT "[clr]ENTER A BUDGET CATEGORY"
110 PRINT "[dn rvs]A[rvs off]NNUAL"
120 PRINT "[dn rvs]M[rvs off]ONTHLY"
130 PRINT "[dn rvs]D[rvs off]AILY"
140 GET A$:IF A$="" THEN 140
150 FOR I=1 TO 3
160 IF A$=MID$("AMD",I,1) THEN ON I GOSUB
    200,400,600
170 GOTO 100
200 REM ANNUAL BUDGET ITEMS
210 PRINT "[clr]SELECT AN ANNUAL ITEM"
220 PRINT "[dn]1 - HOUSE INSURANCE"
230 PRINT "[dn]2 - CAR INSURANCE"
240 PRINT "[dn]3 - INCOME TAX"
250 PRINT "[dn]4 - PROPERTY TAX"
260 PRINT "[dn]5 - GOTO MAIN MENU"
270 GET A$:IF A$="" THEN 270
280 N=VAL(A$)
290 IF N=5 THEN RETURN
300 ON N GOSUB 1000,2000,3000,4000
310 GOTO 200
400 REM MONTHLY BUDGET ITEMS
      .
      .
   etc.
```

FORM FILLING

Some application areas, such as accounting, CAI, and testing, can benefit from a technique called form filling. This method permits the program's user to input data in a predefined location on the line. Typically this location is inside a box, as in an accounting ledger, or in a blank area such as a fill-in-the-blank type of test question. The following code shows the use of form filling to enter an account number and date in a precise location within a box. In this code the lowercase "b" represents a blank character.

```
10 PRINT "[clr rvs]bACCOUNTbbbbbbDATEbbb[rvs off]"
20 PRINT "[rvs]b[rvs off]bbbbbbbb[rvs]b[rvs off]bbb
   bbbbbbb[rvs]b[rvs off]"
30 PRINT "[rvs21spaces[rvs off]"
40 INPUT "[home dn rt]";A$
50 INPUT "[home dn 10 rt's]";D$
60 A$=LEFT$(A$,6)
70 D$=LEFT$(D$,9)
```

The above code displays a form on the screen something like this:



When the program asks for input, the question mark appears in the box under the appropriate heading and is followed by the flashing cursor. This type of program presents a unique problem since the INPUT statement accepts all of characters on the line following the question mark. This also includes graphic characters.

One solution is to use the GET statement in a loop, but a far simpler method is to use the LEFT$ function, which selects the number of characters desired from the input string. This of course means you need to know how many characters will be entered.

COMMAND LANGUAGES

Command languages are useful for such applications as word processing, where simple prompting or the use of a menu is either impractical or too unwieldy to enter complex commands.

For instance, a typical command is to change a string from one value to another to correct a spelling error or to change a word. To change the word "error" in the previous sentence to "mistake," a command like

```
C/error/mistake
```

is entered.

Using a command language requires considerably more experience than simply responding to a menu, but it is much faster than using a dialogue. Programming for a command language also tends to be more complex since the program needs to recognize the type of command, often identified by the first character in the command, and the operands that can legitimately accompany that command. Often there is no prompt since this type of application requires an active operator who initiates all action.

The following code shows how the preceding command might be analyzed in BASIC.

```
100 INPUT C$
110 IF LEFT$(C$,1)="C" THEN 500
 .
 .
500 REM DECODE CHANGE COMMAND
505 S1$="": S2$="" :REM EMPTY STRINGS
510 IF MID$(C$,2,1)<>"/"THEN PRINT"COMMAND
    ERROR":GOTO 100
520 L=LEN(C$)
530 FOR I=3 TO L
540 IF MID$(C$,I,1)="/" THEN 570
550 S1$=S1$+MID$(C$,L,1)
560 GOTO 580
570 S2$=RIGHT$(C$,L-I):I=L
580 NEXT I
590 IF LEN(S2$)=0 THEN PRINT "COMMAND ERROR":GOTO 100
```

Statements 510 to 590 ensure the command format is followed by checking for a slash separating the command (C) from the first string and then checking for a second string. Statements 530 to 560 extract the first string by concatenating each character to S1$. When the end of the string is found by 540, the second string is extracted in 570 and stored in S2$.

REVIEW QUESTIONS—CHAPTER 7

1. Why is it important to consider the experience level of persons who will be using your programs?
2. What are the three different levels of computer users? How do these levels correspond to the types of people who will be likely to use your programs?
3. Discuss the types of prompts that may be used for interacting with a user.
4. What is meant by a default? Write an INPUT statement that asks for an INSERT or DELETE response, providing the insert option as the default.
5. Define a menu. How is a menu created on the screen? Describe three possible ways of accepting the user's response to a menu.
6. Describe a multilevel menu. Give an example of this kind of menu other than the example given in the chapter.
7. What is form filling? What are the advantages and disadvantages of this type of user data entry?
8. Explain what is meant by a command language.

8 Graphics, Animation, Sound, and Music

One of the particularly nice features about the C-64 is the availability of the graphic character set. This set consists of 62 characters directly accessible from the keyboard and 4 additional characters that are available with the POKE command. By using the reverse of these characters, a total of 132 characters are available.

GRAPHIC CHARACTER SET

Figure 8.1 shows the graphic character set with the related POKE codes, while figure 8.2 shows the graphic characters organized by type. Refer to the appendices for other graphic codes.

Most graphics are selected by pressing the shift or Commodore key with the appropriate graphic key and can be used in a character string like any other character. For instance, an upper-left corner graphic is selected by pressing the Commodore key and the A key, giving:

By using the corner symbols on keys A, S, Z, and X, the horizontal line (shifted *), and the vertical line (shifted −) a box may be drawn with the following program.

```
10 PRINT "┌──┐"
20 PRINT "│  │"
30 PRINT "│  │"
40 PRINT "└──┘"
```

Using the Reverse key, a solid box may be drawn as follows:

```
10 PRINT "[rvs] ┌──┐"
20 PRINT "[rvs] │  │"
30 PRINT "[rvs] │  │"
40 PRINT "[rvs] └──┘"
```

A solid outline with an empty interior could be produced by turning Reverse on and off.

```
10 PRINT "[rvs] ┌──┐ "
20 PRINT "[rvs] │ [off] [rvs] "
30 PRINT "[rvs] │ [off] [rvs] "
40 PRINT "[rvs] └──┘ "
```

Using the appropriate codes, you can POKE graphics to the screen instead of using the PRINT. Reverse graphic characters are produced by adding 128 to the POKE code. The POKE code for a solid square (reverse space) is 160, which is code 32 + 128. Code 160 may be used as follows to produce a solid box.

```
10 POKE 1104,160:POKE 55376,1
20 POKE 1105,160:POKE 55377,1
30 POKE 1144,160:POKE 55416,1
40 POKE 1145,160:POKE 55417,1
```

This approach may seem more awkward than using PRINT and for this graphic it really is. However, POKE has the advantage of executing faster than PRINT, and since the address and value may both be variables we can take advantage of this characteristic if a pattern is to be duplicated on the screen. But more on this later.

Before we proceed you might want to review the POKE commands for screen characters and color codes from chapter 6.

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	R	r	18	$		36
A	a	1	S	s	19	%		37
B	b	2	T	t	20	&		38
C	c	3	U	u	21	'		39
D	d	4	V	v	22	(		40
E	e	5	W	w	23	)		41
F	f	6	X	x	24	*		42
G	g	7	Y	y	25	+		43
H	h	8	Z	z	26	,		44
I	i	9	[		27	−		45
J	j	10	£		28	.		46
K	k	11	]		29	/		47
L	l	12	↑		30	∅		48
M	m	13	←		31	1		49
N	n	14	SPACE		32	2		50
O	o	15	!		33	3		51
P	p	16	"		34	4		52
Q	q	17	#		35	5		53

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
6		54		O	79			104
7		55		P	80			105
8		56		Q	81			106
9		57		R	82			107
:		58		S	83			108
;		59		T	84			109
<		60		U	85			110
=		61		V	86			111
>		62		W	87			112
?		63		X	88			113
		64		Y	89			114
	A	65		Z	90			115
	B	66			91			116
	C	67			92			117
	D	68			93			118
	E	69			94			119
	F	70			95			120
	G	71	SPACE		96			121
	H	72			97			122
	I	73			98			123
	J	74			99			124
	K	75			100			125
	L	76			101			126
	M	77			102			127
	N	78			103			

Figure 8.1 Graphic character codes. Set 1 is the set normally available on the C-64. To get set 2, first POKE 53272,23 and then type the set 1 character. Reverse characters are the poke codes given above plus 128. To get back to set 1, use POKE 53272,21.

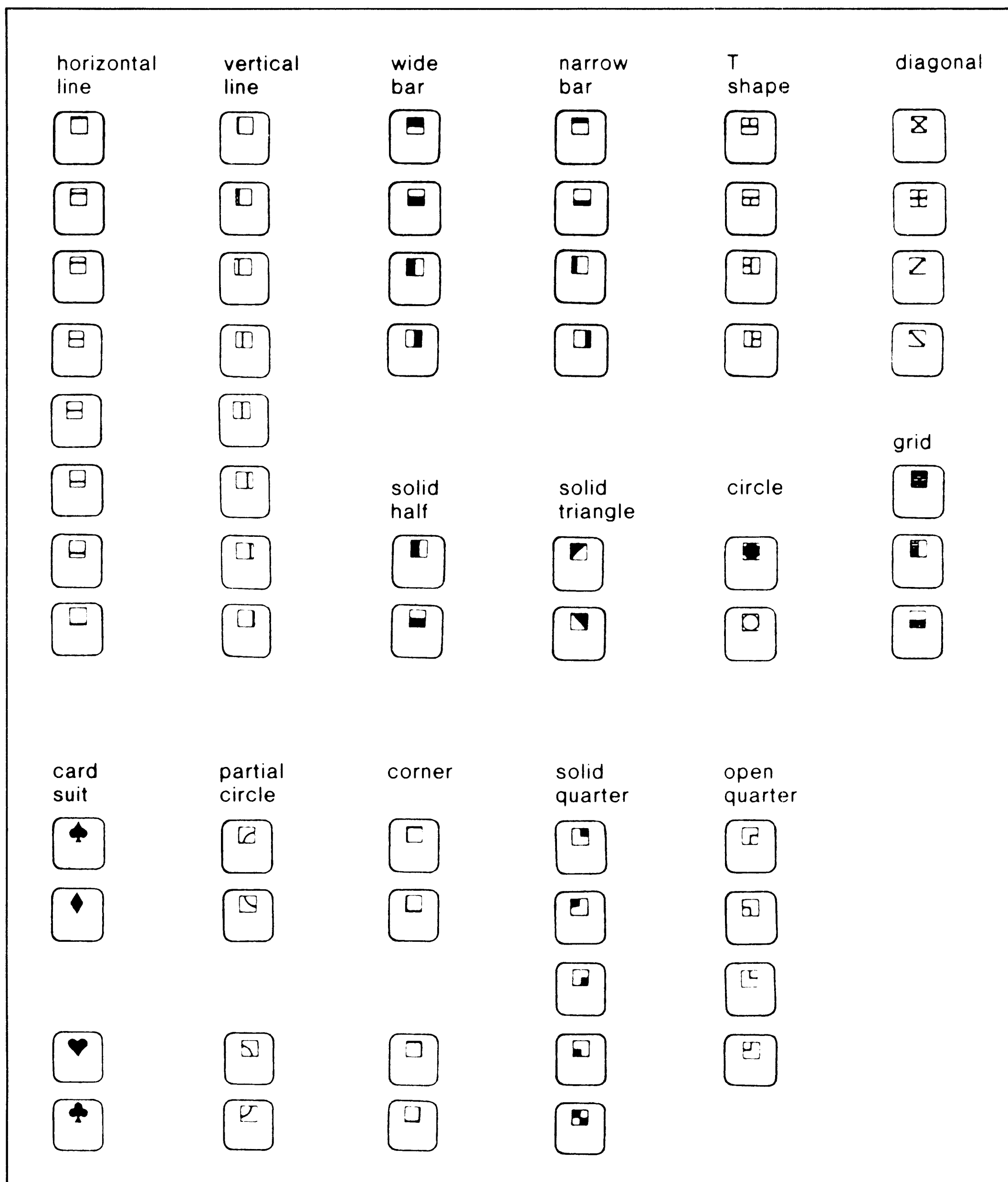


Figure 8.2 Graphic characters by type

REACTION TIMER WITH GRAPHICS

Many game and simulation programs will benefit visually from a type of scoreboard that draws the user's attention to specific information used in the program. For instance, the popular Lunar Lander program uses a scoreboard to display a constantly changing rate of descent, altitude, fuel remaining, and elapsed time.

The Reaction Timer we developed earlier would be more visually appealing if the GO signal were placed in a box to draw attention to it. We could even make the signal green. Then the box could also be used to display the reaction time when a key has been pressed. Figure 8.3 shows a layout of the graphic in the design stage. When graphics become more complex (this one is still quite simple), it often helps to do a layout on graphic paper before trying to code it in the program.

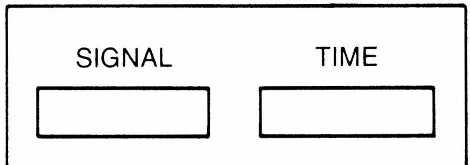


Figure 8.3 Reaction timer graphic layout

Figure 8.4 contains the program for the timer. Once the scoreboard has been drawn, the word "READY" is displayed in red in the box under the title START. This action requires cursor positioning with subroutine 600 and a TAB(5) in line 520. Using both cursor control and TAB permits the program to display a value (READY) inside the box without destroying the lines currently on the screen.

A time delay is then used before the "GO" signal is displayed. Notice that the string containing GO in line 544 contains extra spaces to ensure all the letters of START are cleared from the box.

The remainder of the program is basically unchanged except for line 551. This statement calculates the elapsed time (ET) and reduces the answer to three decimal places so it will fit in the box.

```
100 REM REACTION TIMER
140 PRINT ""
145 PRINT TAB(4)"REACTION  TIMER"
146 PRINT
150 PRINT TAB(3)"┌──────────┬─────────┐"
160 PRINT TAB(3)"| SIGNAL   | TIME    |"
170 PRINT TAB(3)"├──────────┼─────────┤"
180 PRINT TAB(3)"|          |         |"
182 PRINT TAB(3)"|          |         |"
184 PRINT TAB(3)"|          |         |"
190 PRINT TAB(3)"└──────────┴─────────┘"
195 GOSUB 700
200 GOSUB 500
210 GOSUB 540
220 PRINT "TRY AGAIN (Y/N)"
230 INPUT A$
235 IF LEFT$(A$,1)="Y" THEN 100
240 STOP
500 REM DISPLAY READY SIGNAL
510 GOSUB 600
520 PRINTTAB(5)"READY";
530 GOSUB 700
535 RETURN
540 REM DISPLAY GO SIGNAL
542 GOSUB 600
544 PRINTTAB(5)"  G O  ";
546 OT=TI
548 GET A$:IF A$="" THEN 548
550 NT=TI
551 ET=INT((NT-OT)/60*1000)/1000
552 PRINTSPC(3);ET
554 RETURN
600 PRINT "";
610 RETURN
700 REM TIME DELAY
710 FOR I=1TO1500:NEXT
720 RETURN
```

Figure 8.4 Reaction timer program

USING POKE TO PRODUCE A GRAPHIC

The introduction to this chapter mentioned that both PRINT and POKE could be used to produce graphics. POKE generally operates faster than PRINT and also permits the use of variables that will be valuable for the chessboard graphic. Speed is primarily of interest for animation.

Lunar Lander

An all-time graphic favorite is the Lunar Lander, and we will produce it here using POKE statements. First, a graph paper drawing of the LEM is needed, given in figure 8.5.

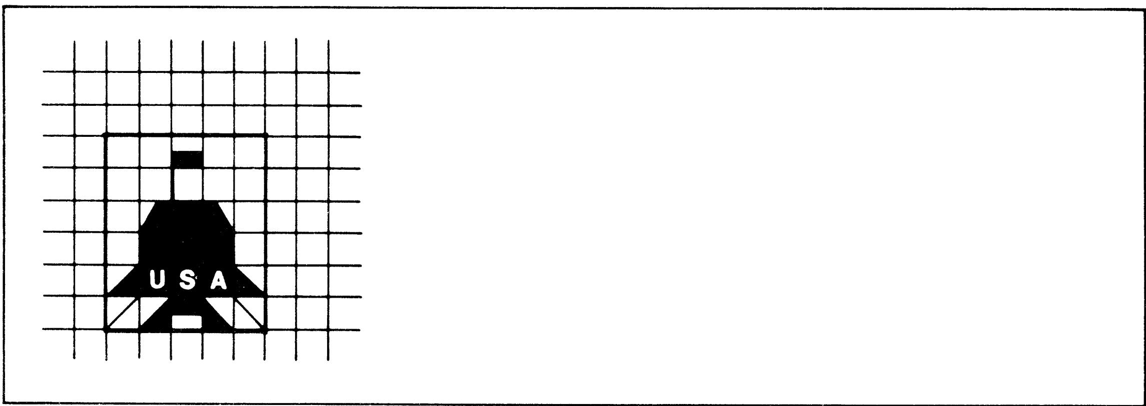


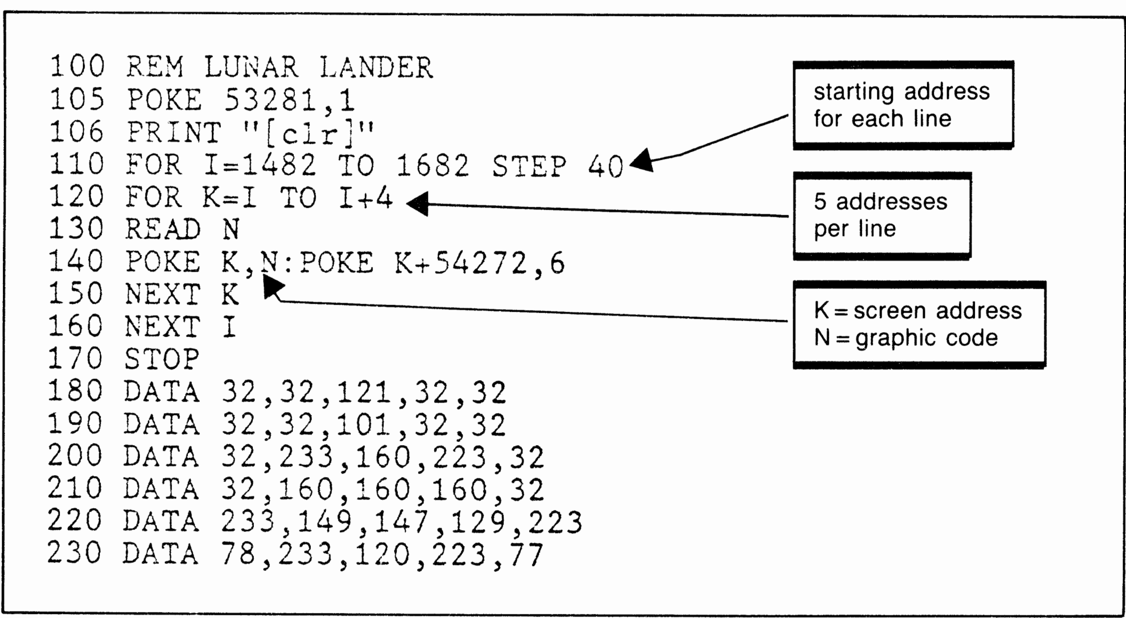
Figure 8.5 Lunar lander working drawing

The next step is to determine the ASCII POKE codes for each of the characters in the graphic. Using figure 8.1 or appendix E, select each code, remembering that some of them are reverse graphic characters (POKE code + 128), and write them down in the order they appear in the design. Now you will have a chart as follows:

32	32	121	32	32
32	32	101	32	32
32	233	160	223	32
32	160	160	160	32
233	149	147	129	223
78	233	120	223	77

We now have a choice. Either a separate POKE can be used for each character, which requires 30 POKEs, or we could read these numbers as data and POKE them using a variable. This is the preferred solution, as shown in figure 8.6. Notice the nested FOR loops that control the reading and POKEing. The starting value (1482) of the outer value controls the screen addresses that are POKEd with the data. The STEP value 40, since the screen has 40 characters per line, advances to the identical column on the next row of the screen. The command POKE K + 54272 with a value of 6 sets the screen color of each character to blue.

For added variety try to change the program to read both a value for the screen character and also a value for the color. This approach allows for different colors, making the lander an even more interesting graphic.



```
100 REM LUNAR LANDER
105 POKE 53281,1
106 PRINT "[clr]"
110 FOR I=1482 TO 1682 STEP 40
120 FOR K=I TO I+4
130 READ N
140 POKE K,N:POKE K+54272,6
150 NEXT K
160 NEXT I
170 STOP
180 DATA 32,32,121,32,32
190 DATA 32,32,101,32,32
200 DATA 32,233,160,223,32
210 DATA 32,160,160,160,32
220 DATA 233,149,147,129,223
230 DATA 78,233,120,223,77
```


Figure 8.6 Lunar lander using POKEs

Chessboard

The problem of displaying a chessboard seems at first quite simple. However, two problems soon overshadow this simplicity. The first occurs when we consider the number of POKE statements needed. If the board is to consist of 3 × 3 squares on the screen, we need 3 × 3 × 8 × 8 POKEs (a board has 8 squares to a side). Obviously, 576 POKEs are too many. But, since each square is either black (blank screen) or white (ASCII 160), then possibly we can compute the value to be POKEd and use variables.

The second problem may not be apparent until our first attempt to display the chess board. If each square on the board is indeed 3 × 3, then a square that is longer than it is wide will display on the screen. The reason for this surprise is that each character on the screen has about a 4:3 height-to-width ratio. This problem may be corrected by making each square on the board four characters wide and three characters high.

The program is in figure 8.7.

```
80 PRINT "[clr]"
90 N=1024
94 FOR S=1 TO 8
95 FOR R=1 TO 3
100 FOR P=1 TO 32 STEP 8
110 FOR P1=1 TO 4
120 POKE N,160:POKE N+54272,2
125 N=N+1
130 NEXT P1
135 N=N+4
140 NEXT P
145 N=N+8
150 NEXT R
160 IF (INT(S/2)*2)<>S THEN N=N+4:GOTO 170
165 N=N-4
170 NEXT
```

Figure 8.7 Chessboard graphic program

ANIMATION

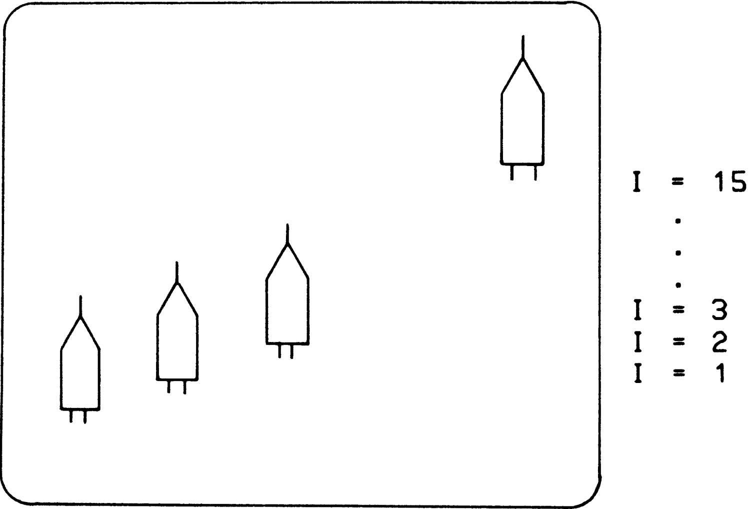
Animation is the process of causing a graphic to move up, down, left, right, or diagonally on the screen. Usually, animating an object on a microcomputer is done with cursor controls in a PRINT statement or by recalculation of a series of POKE addresses. Normally the Hollywood cartoon approach of creating many frames of an object and displaying them rapidly is not used. Avoid this approach, if possible, because of the excessive amount of storage it requires and the substantial amount of time needed to type in all the images.

Rocket 1

Figure 8.8 is a program to create a rocket blasting off. This is one of the easiest animation techniques to master. The program begins with a clear screen and positions the cursor 11 lines from the bottom of the screen by printing the variable A$, which contains cursor down characters.

```
60 A$="dn dn dn dn dn dn dn dn dn dn dn dn dn dn"
95 PRINT LEFT$(A$,15-I);
```

At this position the rocket is displayed using PRINT statements. Next, the FOR loop causes the cursor to be positioned one line higher than for the previous output by selecting 15-I characters from A$. Now when the rocket prints, it is one line higher on the screen than before. Since this happens relatively quickly, motion is produced and the rocket takes off.



One other important feature of the program is the last PRINT statement, which simply prints a blank line of characters. This is necessary to wipe out the previous bottom line of the rocket, which otherwise will stay on the screen as each new rocket image is displayed.

```
60 A$="▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
70 PRINT"▒"
80 FOR I=1TO15
85 PRINT"▒"
95 PRINTLEFT$(A$,15-I);
100 PRINT"     |     "
110 PRINT"    ▒▒▒▒    "
130 PRINT"    ▒  ▒   "
140 PRINT"    ▒  ▒   "
150 PRINT"    ▒  ▒   "
160 PRINT"   ▒    ▒   "
170 PRINT"    ▌ ▌    "
176 PRINT"            "
180 NEXT
```

Figure 8.8 Rocket 1 program

Rocket 2

You can also produce animation by using cursor controls directly in the print string. This is a technique used extensively in the rather amusing program, Toker. Although not as easy to read, the program in figure 8.9 has the advantage of being short. You'll see a second benefit when the program is run. Because of the cursor movement, the motion of the rocket is smooth since multiple PRINTs are unnecessary. The only problem with this method is the difficulty in controlling the speed of the movement, which could be accomplished with a delay loop in the previous program.

```
5 REM ROCKET
10 PRINT"▒":FORI=1TO18:PRINT"▒▒";:NEXT
20 FORI=1TO18:PRINT"⁄▒▒▒▒ ▒▒▒▒▒▒▒**▒▒▒: :▒▒▒▒▒▒▒";:NEXT
```

Figure 8.9 Rocket 2 program

To aid in reading this program, since the graphics are difficult to decipher, here is an interpretation of the characters in statement 20, where CMD refers to the Commodore key.

```
PRINT"CMD G CMD M lt lt dn
      shift £ shift * lt lt dn
      shift) shift arrow lt lt down
      ** lt lt dn
      :: up up up up up lt lt";
```

Egg Timer

The next program combines the PRINT, POKE, and TIME features to produce an animation of an egg timer or hourglass dropping sand for one minute. In addition to dropping the sand, the program displays a countdown of each second as time passes from 60 to 0 seconds.

The program (figure 8.10) begins by storing the screen address in SC, setting a white screen and red border, and then printing the egg timer in subroutine 2500. This routine uses cursor downs and TABs to position the timer on the screen. The image remains on the screen for the duration of the program.

Next, the time counter (T) is set to 60 seconds in line 200 and printed in line 220. After printing the time, the cursor is moved four positions left to prepare for printing the next T value (59). As each value prints, it overprints the previous value, producing a countereffect. Line 210 also stores the current time (TI) in K for future comparison.

The sand is dropped by POKEing a period into five screen positions within the timer. The address is derived by considering the screen to be numbered from 0 to 999 and then adding this value to 1024 to get to the actual address. Statements 230 to 270 produce the dropping effect as follows:

```
100 SC=1024
110 POKE 53280,2:POKE 53281,1
230 FOR I=435 TO 595 STEP 40
245 POKE I+SC,46:POKEI+SC+54272,6
250 GOSUB 3000
260 POKE I+SC,32
270 NEXT I
```

Since each position to receive a grain of sand is directly below the previous one, an increment of 40 is used, which moves to this position. Subroutine 3000 serves two purposes. First it acts as a delay loop so the sand remains on the screen for a reasonable duration before it is cleared in 260 and the next grain is dropped.

Second, subroutine 3000 checks the time to see if one second (60 jiffies) has passed. The passage of a second is detected by comparing the current value of TI to be 60 greater than the previous value, which was stored in K. At this time the second counter T is reduced by one and the new TI value is stored in K. If the value of T has been reduced below 10, then the cursor is moved one position to the right to compensate for the one-digit number. The new time is then printed in line 3050.

```
100 SC=1024
110 POKE 53280,2:POKE 53281,1
120 PRINT"[CLR]"
140 REM    ***PRINT EGG TIMER***
150 GOSUB 2500
190 REM    ***SET TIME 60 SECS***
200 T=60
205 PRINT"[HOME 8 DN'S] TIME"
206 PRINT"[DN] ";
210 K=TI
220 PRINT T;"[4 LT'S]";
225 REM    ***DROP SAND***
230 FOR I=435 TO 595 STEP 40
245 POKE I+SC,46:POKE I+SC+54272,6
250 GOSUB 3000
260 POKE I+SC,32
```

```
270 NEXT I
280 GOSUB 3000
290 GOTO 230
300 PRINT"[CLR 11 DN'S 8 RT'S]";
310 PRINT"[RVS]T I M E ' S   U P !"
320 PRINT"[5 DN'S]"
900 STOP
2100 GET Z$:IF Z$="" THEN 2100
2110 RETURN
2500 PRINT"[5 DN'S]";
2510 PRINTTAB(33);"▒▒▒▒▒▒"
2520 PRINTTAB(33);"▒▒▒▒▒▒"
2530 PRINTTAB(33);"▒▒▒▒▒▒"
2540 PRINTTAB(33);"▒▒▒▒▒▒"
2550 PRINTTAB(33);"\    /"
2560 PRINTTAB(33);" |  | "
2570 PRINTTAB(33);"/    \"
2580 PRINTTAB(33);"|    |"
2590 PRINTTAB(33);"|    |".
2600 PRINTTAB(33);"▒▒▒▒▒▒"
2900 RETURN
2999 REM  ***PRINT NEW TIME EACH SEC***
3000 FOR D=1 TO 25
3010 IF TI< K+60 THEN 3060
3020 T=T-1
3030 IF T< 10 THEN PRINT"[LT]";
3040 K=TI
3050 PRINTT;"[4 LT'S]";
3060 IF T<=0 THEN 300
3070 NEXT D
3080 RETURN
```

Figure 8.10 Egg timer program

Train—Animating Multiple Objects

This short program demonstrates how to animate two separate objects on the screen at the same time. Imagine an old steam engine chugging across the screen from right to left with white smoke puffing from its stack and floating off to the right.

This technique requires a loop to move the train from the right side of the screen to the left. The use of TAB with a variable can control the position where the image of the train is displayed. For example, the statements

```
10 FOR I=38 TO 1 STEP -1
20 PRINT "[home]"
30 PRINT TAB(I)"0"
40 FOR K=1 TO 100:NEXT
50 NEXT I
```

will move the letter 0 from right to left across the screen. This procedure is basically the one used in the train program. Instead of 0, three PRINTS are used to display the image of the train. In addition, the PRINT to the home position includes some cursor downs to position the train lower on the screen.

The next problem to solve is that of showing the smoke. This is achieved by using a second loop inside the one for moving the train, for instance, such a loop as

```
40 FOR J=1 TO 3
42 PRINT TAB(I+J) "S";
44 FOR K=1 TO 40:NEXT
46 PRINT "[1t]";
48 NEXT J
```

to replace statement 40 in the previous program would create the appearance of S moving left to right while 0 moves in the opposite direction.

Figure 8.11 shows the complete program with the appropriate graphics for the blue train with white smoke on a cyan background. Using the method described above, statements 120 to 160 display the train. Statement 170 repositions the cursor in preparation for displaying the puffs of smoke. The J loop at 180 and the print at 190 display the smoke in alternating open and closed circles.

Statement 195 is the delay loop, while 198 moves the cursor to the left and prints blanks to destroy the last puff of smoke before displaying the next one. This process occurs three times before statement 220 causes the I loop to go back and display the train again, one position to the left.

```
80 REM TRAIN
90 POKE 53280,3:POKE 53281,3
100 A$="▣▣▣▣▣▣▣▣▣▣"
110 PRINT"▣";
120 FOR I=28TO1STEP-1
130 PRINT"▣":A$
140 PRINT TAB(I)"   ▣ ■    ┌─┐ "
150 PRINT TAB(I)" ▣◤      ×■ │ "
160 PRINT TAB(I)"<●●_●●  ┴_"
170 PRINT"▣":A$;TAB(I+2);
180 FOR J=1 TO 3
190 PRINTTAB(J);"▣○ ▣";
195 FOR K=1 TO 40:NEXT
198 PRINT "▮▮▮▮   ";
200 NEXT
220 NEXT I
230 PRINT "▣"
```

Figure 8.11 Train program

HIGH RESOLUTION ON THE COMMODORE 64

Have you tried some of the graphic characters on the C-64 in your programs? If so, you will realize that although a remarkable number of shapes and objects can be created with the graphic character set, there are also times when it seems that just the right kind of character for your program is not available. If you have encountered this problem, there is a technique available on the C-64 that will let you control individual pixels to create the shapes you need.

The method we will use here was developed by David Malmberg for *Compute!'s First Book of VIC* and is revised here for use on the Commodore 64. *Compute!* is a popular computer magazine and is an excellent source of articles for the C-64 and other Commodore computers.

To develop our own characters, we first need to understand how the C-64 knows the shape of the characters it currently uses. This is done through the use of a bit pattern the C-64 has in its memory. If you look carefully at a character on the screen, you will notice that it is made up of a pattern of dots. These dots are more obvious on a larger screen and with such letters as X or V.

The dot pattern on the C-64 is composed of a matrix of 8 × 8 dots, or pixels, as shown in figure 8.12. Normally the last or bottom row of dots is not used, so to leave a space between lines on the screen.

```
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

Figure 8.12 Bit pattern for C-64 characters

When any bit in this pattern is set to a value of 1 this in effect turns it on and creates a bright spot on the screen called a pixel. By turning on different combinations of these bits, the C-64 can create different characters. But how does C-64 know which combination to use?

Starting at address 53248 (part of C-64's ROM) is data telling the C-64 how to create the patterns. Each byte of memory starting at this address supplies the pattern for a group of eight bits in the matrix. Therefore a character requires eight bytes to define its bit pattern.

The patterns are stored according to the ASCII code sequence. The @ is stored in the first 8 bytes, which are 53248 to 53255. The next 8 bytes, 53256 to 53263, contain the pattern for the letter A, and so on.

The memory addresses in the C-64 are as follows:

Address	Bit Pattern
53248	Uppercase characters
53760	Graphics characters
54272	Reversed uppercase characters
54784	Reversed graphic characters
55296	Lowercase characters
55808	Uppercase and graphics
56320	Reversed lowercase characters
56832	Reversed uppercase and graphics

Now try running this program to look at the pattern for the letter A.

```
20 POKE 56334,PEEK(56334) AND 254
30 POKE 1,PEEK(1) AND 251
40 FOR I=53256 TO 53263
50 N=N+1:A(N)=PEEK(I)
60 NEXT I
70 POKE 1,PEEK(1) OR 4
80 POKE 56334,PEEK(56334) OR 1
90 FOR I=1 TO 8:PRINT I+53255,A(I):NEXT I
RUN
```

53256	24
53257	60
53258	102
53259	126
53260	102
53261	102
53262	102
53263	0

The first column of numbers is the address we are PEEKing at. The second column is the decimal value of the data which we found (PEEked) at that address. But what do these numbers mean? If you are familiar with binary arithmetic the answer might be obvious. But even if you have never heard of binary, wait, the solution is near.

Figure 8.13 shows the value the C-64 associates with each bit in the pattern. The values 1, 2, 4, 8, 16, 32, 64, and 128 are called a binary sequence. More important, this is how we determine the decimal value for the bit pattern, in this case the pattern for the letter A. In the first row, the characters with a 1 value are under the 8 and 16 columns. If we add 8 and 16 we get 24, which is the value we PEEKed at in address 53256. Each value follows this pattern.

Address	Value	128	64	32	16	8	4	2	1
53256	24	0	0	0	1	1	0	0	0
53257	60	0	0	1	1	1	1	0	0
53258	102	0	1	1	0	0	1	1	0
53259	126	0	1	1	1	1	1	1	0
53260	102	0	1	1	0	0	1	1	0
53261	102	0	1	1	0	0	1	1	0
53262	102	0	1	1	0	0	1	1	0
53263	0	0	0	0	0	0	0	0	0

Figure 8.13 Bit pattern for the letter A

The technique we use is to transfer these bit patterns to RAM memory and then change them as required to create our own characters. The technique developed is shown in the following BASIC code.

```
10 POKE 52,48:POKE 56,48
20 POKE 56334,PEEK(56334) AND 254
30 POKE 1,PEEK(1) AND 251
40 FOR I=0 TO 2047
50 POKE (I+12288) ,PEEK(53248+I)
60 NEXT
70 POKE 1,PEEK(1) OR 4
80 POKE 56334,PEEK(56334) OR 1
90 POKE 53272,(PEEK(53272) AND 240)+12
100 FOR I=0 TO 7
110 READ N:POKE(I+12288),N
120 NEXT I
130 DATA 255,129,129,129,129,129,129,255
140 END
```

Line 10 of this code establishes a new storage location (address 12288) for the bits patterns. Lines 20 and 30 set some internal values so you can access the ASCII pattern stored in the ROM. Lines 40 to 60 transfer the current bit patterns to the new location at 12288, and then lines 100 to 120 read a set of 8 bits to replace the @ with a new bit pattern. The data for the new pattern is supplied with the data statement.

Lines 70 to 90 restore the memory values so we can access the other character sets in ROM.

In this program only one character was changed. After running the program try typing some @s and see what the new shape is. Of course, having made this change we can no longer use an "at" sign but only the new symbol.

Why not try your hand at producing some other new symbols to replace other characters? How about some characters for a foreign alphabet or your own special cryptography?

SPRITE GRAPHICS

Earlier in this chapter we saw how we could produce graphics and animation on the Commodore 64. Although the methods shown can be effective, they were rather difficult to use and did not always produce a smooth operation for animation. By using the C-64's sprite graphic capability these problems can be avoided.

Sprite Design

Sprites are created in much the same way we developed high resolution graphic characters. A grid of 24 dots wide and 21 dots long shown, in figure 8.14, is used to create a sprite. This sprite may then be displayed in a 320 by 200 dot area of the screen.

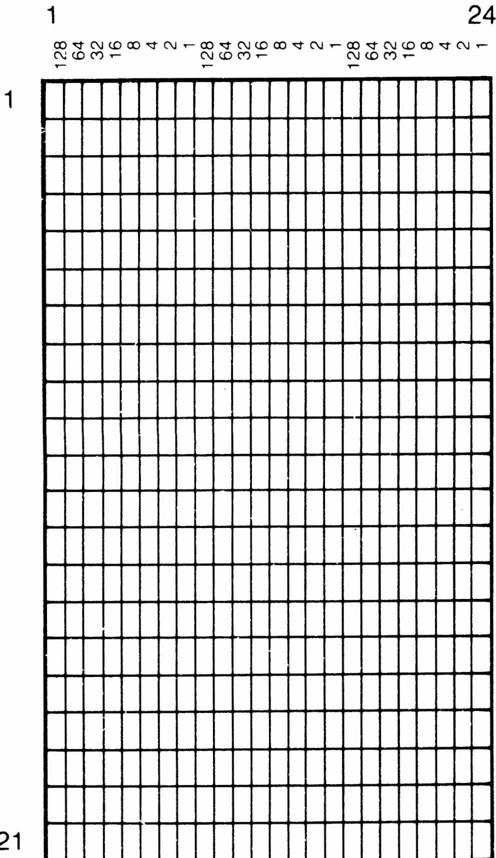


Figure 8.14 Sprite grid

The sprite grid is divided into three groups of columns representing binary values of 1, 2, 4, 8, 16, 32, 64, and 128. Each row will then be represented by 3 values.

To create a sprite, first draw the image on a piece of graph paper and then code 3 values for each row. Since there are 21 rows you will get a total of 63 values.

For example, to create a chopper (helicopter), make a drawing, such as that in figure 8.15. Then compute each square that represents a value of 1 according to the binary value for the position of the square. The numbers to the right of the drawing list these values. These numbers will be used later in data statements in a program that displays and animates the chopper.

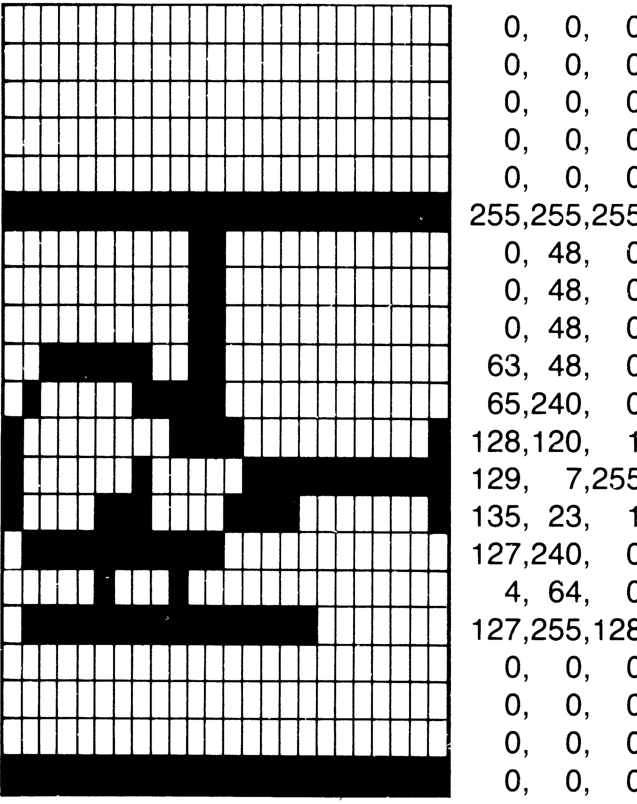


Figure 8.15 Sprite drawing for the chopper

Sprite Registers

A maximum of eight sprites, numbered 0 to 7, may be used in a program. Eight memory locations immediately after the screen memory are used to define where the data for the sprite is stored. These sprite locations are as follows:

2040	2041	2042	2043	2044	2045	2046	2047
Sprite 0	1	2	3	4	5	6	7

If you are using sprite 2, as we will in the sample program, location 2042 will be POKEd with the location of the data that describes the shape of sprite 2. Any combination of the 8 sprites may be used by a program, although of course this would complicate things considerably.

Sprites are displayed with the aid of a special video chip in the C-64. This chip's address begins at memory location 53248 and contains a series of registers for controlling the sprites. These registers are defined as follows:

Register	Description
0	X coordinate sprite 0
1	Y coordinate sprite 0
2-15	X, Y coordinates for sprites 1 to 7
16	Most significant bit (X coordinate)
21	Sprite 1-appear, 0-disappear
23	Expand sprite — Y direction
29	Expand sprite — X direction
39-46	Sprite color (sprites 0 to 7)

A sprite is displayed by setting register 21 to a value of 1, which represents the sprite being used. The register has the following format:

128	64	32	16	8	4	2	1	Decimal values
7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0-disable 1-enable

To enable sprite 2, set the value of register 21 to 4. Here is the program code:

```
20 V=53248
30 POKE V+21,4:POKE 2042,13
```

These two lines get things started. The variable V represents the address of the video chip (53248). Using a variable will make it easier for later reference in the program. Then register 21 is POKEd in line 30 with a value of 4 representing sprite 2. If we were using sprite 3 a value of 8 would be used. If both sprites 2 and 3 were used, a value of 12 would be set into location V + 21. The second POKE says that sprite 2 (its address is 2042) will get its grid of data from memory block 13, which starts at address 832.

The sprite data is now loaded into memory from data statements with the following loop:

```
40 FOR I=0 TO 62:READ D:POKE 832+I,D:NEXT
```

Sprite Movement

Sprites are moved on the screen by changing the values of the X and Y coordinate registers. For sprite 2, registers 4 and 5 control the X and Y coordinates. Values from 0 to 255 may be POKEd into these registers to position the sprite on the screen. Normally this is done by enclosing the POKE inside a FOR loop so that the values may be varied, thus causing movement of the sprite.

For example, to move sprite 2 from the top right of the screen to the bottom left, the following loop could be used.

```
50 FOR K=0 TO 200
60 POKE V+4,255-K:POKE V+5,K
70 NEXT K
```

The value 255 – K, which is POKEd into register 4, gives values from 255 to 55 for the X coordinate, moving the sprite from right to left. The second POKE uses K for register 5's value and will have values of 0 to 200 for the Y coordinate, which moves from top to bottom. Since these values are updated together, the sprite moves along both the X and Y coordinates.

Here is the complete program.

```
10 PRINT "[clr]"
20 V=53248
30 POKE V+21,4:POKE 2042,13
40 FOR I=0 TO 62:READ D:POKE 832+I,D:NEXT
50 FOR K=0 TO 200
60 POKE V+4,255-K:POKE V+5,K
70 NEXT K
100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
110 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
120 DATA 128,120,1,129,7,255,135,23,1,127,240,0
130 DATA 4,64,0,127,255,128,0,0,0,0,0,0,0,0,0,0,0,0
```

Moving a Sprite Horizontally

A sprite may be moved horizontally across the screen by varying only the variable for the X-axis. The Y-axis is held constant. In this case a value of 100 is used for the Y-axis to place the chopper near the center of the screen. In the new program, given here, only line 60 is changed to give the sprite horizontal movement.

```
10 PRINT "[clr]"
20 V=53248
30 POKE V+21,4:POKE 2042,13
40 FOR I=D TO 62:READ D:POKE 832+I,D:NEXT
50 FOR K=0 TO 200
60 POKE V+4,255-K:POKE V+,100
70 NEXT K
100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
110 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
120 DATA 128,120,1,129,7,255,135,23,1,127,240,0
130 DATA 4,64,0,127,255,128,0,0,0,0,0,0,0,0,0,0,0,0
```

Moving a Sprite Vertically

The same program can be easily modified to move the sprite along a vertical axis. To do this, the Y-axis variable is changed while the X-axis is held constant. The change occurs in line 60.

```
10 PRINT "[clr]"
20 V=53248
30 POKE V+21,4:POKE 2042,13
40 FOR I=0 TO 62:READ D:POKE 832+I,D:NEXT
50 FOR K=0 TO 200
60 POKE V+4,150:POKE V+5, K
70 NEXT K
100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
110 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
120 DATA 128,120,1,129,7,255,135,23,1,127,240,0
130 DATA 4,64,0,127,255,128,0,0,0,0,0,0,0,0,0,0,0,0
```

Sprite Size

Registers 23 and 29 are used to expand the size of the sprite to double its original size. Register 23 is POKEd to expand in the Y direction and 29 in the X direction. The value used for expansion is the decimal value that represents a sprite; sprite 2 is represented by the value 4.

The chopper program is shown again here with line 45 inserted to expand the chopper in both the X and Y directions. Expanding in only one direction will create a rather weird shape, but why not try it?

```
10 PRINT "[clr]"
20 V=53248
30 POKE V+21,4:POKE 2042,13
40 FOR I=0 TO 62:READ D:POKE 832+I,D:NEXT
45 POKE V+23,4:POKE V+29,4
50 FOR K=0 TO 200
60 POKE V+4,255-K:POKE V+5,K
70 NEXT K
100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
110 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
120 DATA 128,120,1,129,7,255,135,23,1,127,240,0
130 DATA 4,64,0,127,255,128,0,0,0,0,0,0,0,0,0,0,0,0
```

Sprite Color

Sprites may also be given color properties. The color registers are located in 39 to 46, which are addresses 53287 to 53294. To set the color, POKE the appropriate register with a color code from 0 to 16. Here is the program using statement 5 to set the chopper to a light-green color.

```
5 POKE 53289,13
10 PRINT "[clr]"
20 V=53248
30 POKE V+21,4:POKE 2042,13
40 FOR I=0 TO 62: READ D:POKE 832+I,D:NEXT
50 FOR K=0 TO 200
60 POKE V+4,255-K:POKE V+5, K
70 NEXT K
100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
110 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
120 DATA 128,120,1,129,7,255,135,23,1,127,240,0
130 DATA 4,64,0,127,255,128,0,0,0,0,0,0,0,0,0,0,0,0
```

GENERATING SOUND AND MUSIC

Your Commodore 64 is equipped with a sophisticated music synthesizer that can be programmed to mimic a wide variety of musical instruments. Music may be produced with up to three voices to create three-part harmony, but that's for later.

Sound is produced by POKEing memory locations to set the different characteristics of the sound you want. Five of these settings are needed to produce basic sound. These are Volume, Attack/Decay, Sustain/Release, High Frequency-Low Frequency (actually two settings) and Waveform. Figure 8.16 shows the addresses needed for each of these settings for the three voices.

Item	Voice 1	Voice 2	Voice 3
Volume	54296	54296	54296
Attack/Decay	54277	54284	54291
Sustain/Release	54278	54285	54292
High Frequency	54273	54280	54287
Low Frequency	54272	54279	54286
Waveform	54276	54283	54290

Figure 8.16 POKE addresses for sound settings

Volume

The volume setting determines how loud the sound will be. Values from 0 to 15 may be POKEd into address 54296 to control the sound for all three voices. Value 0 turns the sound off and 15 is the loudest. Normally value 15 is used.

Example:

```
POKE 54296,15
```

Attack/Decay Setting

The attack/decay value determines the two characteristics of the sound known as attack and decay. Attack is the rate at which sound reaches its highest or maximum volume. Decay is the rate at which the sound falls from the highest level to a sustain level.

POKEing values from the following chart into the address for each voice's attack/decay setting will give different rates. For example, a medium attack and a low decay would be represented by values 64 and 2. Adding these together gives 66, which is the value to be POKEd into the address.

Settings may also be combined to use a low and a medium attack with a high and a medium decay. The values for this setting would be 64 + 32 + 8 + 4, giving a value of 108.

Attack/Decay POKE Values

	Voice 1	Voice 2	Voice 3
Memory address	54277	54284	54291

	Attack Value	Decay Value
High	128	8
Medium	64	4
Low	32	2
Lowest	16	1

Example:

```
POKE 54277,108
```

Sustain/Release Setting

The sustain value determines how long a note is prolonged after the initial decay has occurred. Release is the rate at which the note comes to a stop after the duration of the sustain. The combined effect of attack/decay and sustain/release determines the characteristic of the sound and how close it approximates a particular musical instrument.

Like attack/decay, values for sustain and release may be added together to give a combined effect.

Sustain/Release POKE Values

	Voice 1	Voice 2	Voice 3
Memory address	54278	54285	54292

	Sustain Value	Release Value
High	128	8
Medium	64	4
Low	32	2
Lowest	16	1

Example:

```
POKE 54278,68
```

Frequency

The frequency determines what note the C-64 will play. Actually there are two frequency settings for each voice; a high and a low frequency. Both addresses for these settings must be POKEd to give a single note.

Voice Frequency POKE Values

	Voice 1	Voice 2	Voice 3
Memory address High	54273	54280	54287
Low	54272	54279	54286

Notes Fifth octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#
High POKE	34	36	38	40	43	45	48	51	54	57	61	64	68	72
Low POKE	75	85	126	200	52	198	127	97	111	172	126	188	149	169

For example, to set an E, two POKES are necessary for voice 1.

```
POKE 54273,43
POKE 54272,52
```

Waveform Settings

Waveform is another setting that controls the characteristic of a sound. In addition, the waveform setting activates the ASDR features already discussed. There are four types of waveforms: triangle, sawtooth, pulse, and noise. The first three are normally used for music and the fourth type for sound effects.

POKE settings for waveform have two values. The first starts the sound and the second stops it. Normally a delay loop is used between POKES for these values to give the sound its intended duration.

Waveform POKE Values

	Voice 1	Voice 2	Voice 3
Memory address	54276	54283	54290

	Start Value	Stop Value
Triangle	17	16
Sawtooth	33	32
Pulse	65	64
Noise	129	128

A triangle waveform would be started for voice 1 with the command:

```
POKE 542767,17
```

and stopped with the command:

```
POKE 542767,16
```

When the pulse waveform is selected, a pulse rate must also be defined. There is a Hi pulse rate and a Lo pulse rate as follows:

	Voice 1	Voice 2	Voice 3
Hi pulse Value 0 to 15	54275	54282	54289
Lo pulse value 0 to 255	54274	54281	54288

Playing a Single Note

Now to combine all of these settings to play a note on the computer. First, type in the following program.

`10 FOR C=54272 TO 54296:POKE C,0:NEXT`	
`20 POKE 54296,15`	Volume
`30 POKE 54277,9`	Attack/decay
`40 POKE 54278,68`	Sustain/release
`50 POKE 54273,34:POKE 54272,75`	Hi/lo frequency
`60 POKE 54276,17`	Start waveform
`70 POKE I=1 TO 500:NEXT`	Note duration
`80 POKE 54276,16`	Stop waveform

When you run this program you will get a single short note; a C in the fifth octave. Getting a tune is but a short step from here, as shown in the next example.

Take Me Out to the Ballgame

The previous program is revised now to read values from data statements for the frequency values in line 50. Additional lines 45 and 46 read these values and check for the end of data, which is represented by negative ones. Recognize the tune? It's the first two bars of "take me out to the ballgame."

```
10 FOR C=54272 TO 54296:POKE C,0:NEXT
20 POKE 54296,15
30 POKE 54277,9
40 POKE 54278,68
45 READ N1,N2
46 IF N1=-1 THEN POKE 54296,0:STOP
50 POKE 54273,N1:POKE 54272,N1
60 POKE 54276,17
70 FOR I=1 TO 500:NEXT
80 POKE 54276,16
90 GOTO 45
100 DATA 34,75,68,149,57,172,51,97,43,52,51,97,38,126
110 DATA 34,75,68,149,57,172,51,97,43,52,51,97,-1,-1
```

Note Duration

One obvious problem with the above program is that each note is of the same duration. We want to be able to tell the computer when we want a quarter note, half note, whole note, and so on. This is really quite easy to accomplish. An extra value will be added to the data to define the duration according to the following list.

Note Type	Data Value
Eighth	1
Quarter	2
Dotted quarter	3
Half	4
Whole	8

The program is now changed to read an extra value (D) in statement 45 and the delay loop in 70 uses the formula D*100 to control the length of time needed for playing the note. These changes make the tune much easier to appreciate. Here is the program.

```
10 FOR C=54272 TO 54296:POKE C,0:NEXT
20 POKE 54296,15
30 POKE 54277,9
40 POKE 54278,68
45 READ N1,N2,D
46 IF N1=-1 THEN POKE 54296,0:STOP
50 POKE 54273,N1:POKE 54272,N1
60 POKE 54276,17
70 FOR I=1 TO D*100:NEXT
80 POKE 54276,16
90 GOTO 45
100 DATA 34,75,4,68,149,2,57,172,2,51,97,2,43,52,2
110 DATA 51,97,4,38,126,8
120 DATA 34,75,4,68,149,2,57,172,2,51,97,2,43,52,2
130 DATA 51,97,4,-1,-1,-1
```

Sound Characteristics

So far we have used only one type of sound for our musical instrument. Now let's try some others. Any of the values for attack/decay, sustain/release, or waveform can be changed to make a difference to the sound. For instance, try changing the sustain/release in line 40 to the following value:

```
40 POKE 54278,0
```

Now run the program and notice the difference in the sound. Next try setting sustain/release back to 68 and altering the attack/decay with

```
30 POKE 54277,200
```

Again there is a totally different sound. Try your own combination of values. What instruments can you produce?

Now try a new waveform. Use a sawtooth instead of the triangle by changing lines 60 and 80 as follows:

```
60 POKE 54276,33
70 FOR I=1 TO D*100:NEXT
80 POKE 54276,32
```

To try a pulse waveform, pulse rates must also be POKEd into 54275 and 54274 for voice one.

```
60 POKE 54276,65:POKE 54274,15:POKE 54275,15
70 FOR I=1 TO D*100:NEXT
80 POKE 54276,64
```

As you can imagine, the possibilities are almost endless for sound and music on your Commodore 64.

REVIEW QUESTIONS—CHAPTER 8

1. Write some PRINT statements to produce a graphic triangle on the screen.
2. Produce the same triangle in question 1 using POKE statements.
3. Draw a graphic with changing values to simulate a display of a digital watch.
4. Try to change the program above to include stopwatch capabilities.
5. Using sprites, write a program to drop a parachute on the screen.
6. Can you make the above parachute drop from a plane that is in flight across the screen?
7. What addresses need to be POKEd to produce sounds on the C-64? How are these addresses used to produce a steady tone for a one-second duration?
8. How would you produce a beep beep sound?
9. Write a program that will play a short tune when it is RUN.

9 Tape Files Extend Your Reach

What kind of record keeping would you like to do on your C-64? Personal or business accounts? Financial records? Student records? Or maybe you are a collector of books, records, stamps, or even baseball cards. The use of files in the C-64 allows you to store data on tape or disk files that are external to the program. In addition to eliminating or reducing the need for DATA statements, files permit the data to be accessed by any other BASIC program. This is not the case with the DATA statement, which may be accessed only by the program containing it.

Files may be stored and accessed in several ways, depending upon whether tape or disk is used. Sequential filing is possible on either tape or disk and it is the easiest technique to master. Direct access or relative files are available only with disk but represent an important technique to be used for handling large sets of data. In this chapter we will restrict our discussion to sequential tape files.

CONCEPTS

Sequential files may be thought of as DATA statements residing on magnetic tape. Data on tape files are read or written sequentially, starting at the beginning of the file and progressing through each item of data until the end of file is reached. Figure 9.1 shows how data appears on a tape file. The major differences between sequential files and DATA statements are the lack of a statement number and the missing keyword, DATA. In other words, just the data are present on the file.

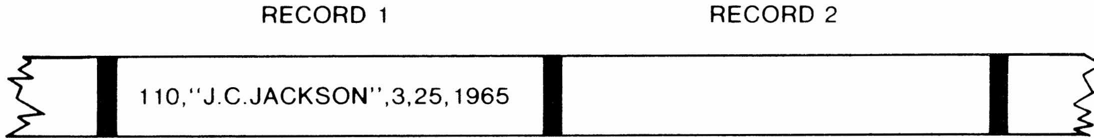


Figure 9.1 A sequential tape file

An important characteristic of tape is its ability to store records sequentially. Since these records do not form part of the computer's memory, except as they are read by the program and depending upon the type of application, files may sometimes exceed the size of the memory capacity.

OPEN AND CLOSE

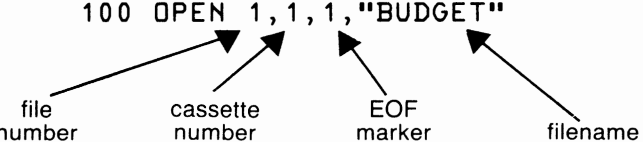
```
statement number OPEN file,cassette,i/o,"filename"

statement number CLOSE FILE
```

Before the program can write data on the tape or read from an existing tape file, you must open the file. Opening makes the file available to our program, defines whether it will be input or output, and gives it a number and a name. The OPEN statement is used for this purpose and has the following format:

10 OPEN a,b,c, "name"
where: a is an integer from 1 to 255 used to identify the file. b is 1 for a cassette file. c specifies read or write as follows: 0—tape is input 1—tape is output with an End-of-File (EOF) marker when it is closed. 2—tape is output with an End-of-Tape (EOT) marker when it is closed. name—is the name used to identify the file.

Let's try a simple example of how this OPEN statement works. If you wanted to create a tape file with data representing your budget, the following OPEN could be used:

```
100 OPEN 1,1,1,"BUDGET"
```



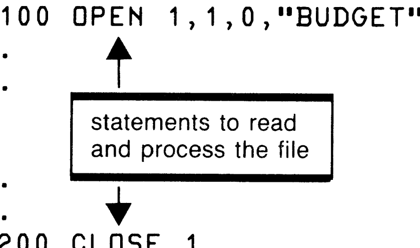
This statement will open file number 1, on cassette number 1, to be written with an EOF marker following the last record on the file. The file on the tape will be identified with the filename BUDGET. When this file is read later, the program can then check the filename, which is recorded at the beginning of the file, to ensure we are reading the correct file.

After the file has been created it must be closed with the CLOSE statement. Closing writes the EOF or EOT marker on output files and also releases the tape for other program operations. For example, an output file could be closed and then opened for input in the same program.

The CLOSE statement simply identifies the file number:

```
200 CLOSE 1
```

A program that reads and processes a tape file will have the following basic structure:



Notice in this example that the third parameter in the OPEN is a zero, which identifies the file as output.

PRINT#

statement number PRINT#file, list of variables

Data are placed on the tape (output) by the PRINT# statement. The number used after the # symbol is the file number assigned to the tape in the OPEN statement. In this case a 1 has been used.

```
100 PRINT#1,A5
```

This statement indicates that the contents of variable A5 is to be written (printed) on tape number 1.

WRITING BUDGET NAMES ON TAPE

Figure 9.2 contains a program that accepts names of budget items from the keyboard and writes them sequentially on the tape.

```
100 REM CREATE NEW FILE
110 PRINT CHR$(147):REM CLEAR SCREEN
120 OPEN 1,1,1,"BUDGET"
130 PRINT "ENTER ITEM NAME (END)"
135 INPUT N$
140 IF N$="END" THEN 170
150 PRINT#1,N$
160 GOTO 130
170 CLOSE 1
```

Figure 9.2 PRINT budget item names on tape

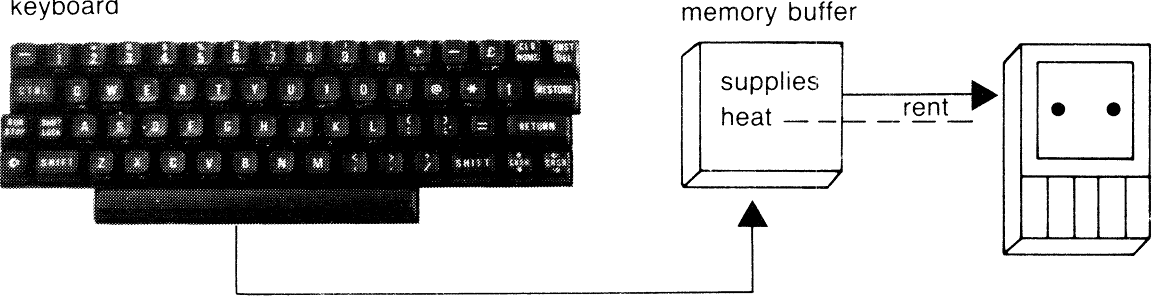
The program begins with the message:

```
PRESS RECORD & PLAY ON TAPE
```

The program then asks the user for an item name or the word END to indicate all of the data has been entered. This is how the screen will appear after a number of entries have been made from the keyboard.

```
ENTER ITEM NAME (END)
?RENT
ENTER ITEM NAME (END)
?TELEPHONE
ENTER ITEM NAME (END)
?LIGHT
ENTER ITEM NAME (END)
?HEAT
ENTER ITEM NAME (END)
?SUPPLIES
ENTER ITEM NAME (END)
?END
```

As these names are being entered, no activity will be apparent on the tape. Eventually, if enough items are entered or END is reached, the tape will then write. What is happening here is that the data is first stored in a memory buffer in the C-64. This buffer can hold a maximum of 191 characters, so only when it has been filled with data or the file is closed will the C-64 then write the data from the buffer to the tape drive. The internal use of a buffer does not usually affect the way you will write tape programs.



The budget item names will appear as follows on the tape:

RENT(cr)TELEPHONE(cr)LIGHT(cr)HEAT(cr)SUPPLIES(cr)

Since each item is being printed individually by statement 150, a carriage return (cr) will follow each one. The carriage return is an ASCII code of 13 and is necessary to separate each item of data. In this example, the carriage return is included automatically since we are printing only one item at a time to the tape.

INPUT#

statement number INPUT#file, list of variables

Data is read from the tape in a manner similar to the keyboard. But instead of using INPUT, the INPUT# statement is used. The statement:

```
100 INPUT#1,NA
```

reads a numeric value from the tape identified as 1 in the OPEN and places the value in the variable NA. Now the need for the carriage return character on the tape is evident. When the number for NA is read, the carriage return signals the end of this item, just as pressing Return on the keyboard does after keying a number.

When data is read from tape, it is necessary to know when all the data has been read. In other words, we need to know when end of file is reached. There are two ways to learn this. The first method requires that you know exactly the number of items to be read. The second uses the C-64's status (ST) code, which signals the program when end of file is reached.

To use the first method, suppose we want to read and display five numbers from a file called NUMBERS. The following program could be used:

```
100 OPEN 1,1,0,"NUMBERS"
110 FOR I=1 TO 5
120 INPUT#1,NA
130 PRINT NA
140 NEXT I
150 CLOSE 1
```

This program reads a tape with five numbers on it, separated by carriage returns as follows:

5.6 (cr) 45 (cr) .05 (cr) 47.8 (cr) 100 (cr)

READING THE BUDGET NAMES TAPE

Figure 9.3 contains a program to read the budget item names created on tape in figure 9.2. This program opens the BUDGET file as input and reads each name, displaying it on the screen. Since this program does not use a fixed number of data items, the ST code is used to detect end of file.

A status of 64 is a signal that end of file has been reached. The ST code is tested in statement 260 prior to the input of a new data item. When the code is equal to 64, then the last item read was the final item on the tape and the program branches to 300, where the file is closed.

A new approach to clearing the screen is also used in this program. Line 230 prints the CHR$ value of 147, which as appendix F shows is the character code equivalent to the CLR key. This approach to using the clear or other functions is useful when a printer that does not print the graphics characters is used with your C-64.

```
175 REM READ BUDGET NAMES
180 PRINT "REWIND TAPE"
190 PRINT "PRESS ANY KEY TO GO ON"
200 GET A$:IF A$ = "" THEN 200
210 REM READ AND DISPLAY FILE
220 OPEN 1,1,0,"BUDGET"
230 PRINT CHR$(147)
240 PRINT "BUDGET ITEM"
250 PRINT
260 IF ST = 64 THEN 300
270 INPUT#1,N$
280 PRINT N$
290 GOTO 260
300 CLOSE 1
```

Figure 9.3 Program to read budget item names

The screen display from this program is as follows:

```
REWIND TAPE
PRESS ANY KEY TO GO ON
PRESS PLAY ON TAPE
```

The screen then clears and displays the following:

```
BUDGET ITEM

RENT
TELEPHONE
LIGHT
HEAT
SUPPLIES
```

HOW TO HANDLE RECORDS WITH MULTIPLE FIELDS

Most computer files require the use of more than one field per record. In a records program, fields such as Name, Occupation, Skills, and Education are necessary for complete information, and as a result each Input or Print operation on tape requires a variable for each of these fields.

We have already seen how each value on tape is followed by a carriage return character. This character separates the data from a following data item. When multiple fields are used, each field in the record must also be separated by a carriage return. Figure 9.4 shows a program to create a budget file that contains both a name of an item purchased and an amount.

```
100 REM CREATE NEW FILE WITH MULTIPLE FIELDS
110 PRINT CHR$(147): REM CLEAR SCREEN
120 OPEN 1,1,1,"BUDGET"
130 PRINT "ENTER NAME (END)"
135 INPUT N$
140 IF N$="END" THEN 180
150 INPUT "ENTER AMOUNT";A
160 PRINT*1,N$;CHR$(13);A
170 GOTO 130
180 CLOSE 1
```

Figure 9.4 Create budget file with name and amount

The carriage return character is included after the name by using the CHR$ function with the ASCII value 13.

```
160 PRINT#1,N$;CHR$(13);A
```

Notice that the fields are separated by semicolons, not commas. Using semicolons keeps the fields tightly packed and saves space on tape.

Another way to create this file would be to use two PRINT# statements:

```
160 PRINT#1,N$
165 PRINT#1,A
```

It is also possible to insert commas between fields as separators:

```
160 PRINT#1,N$;",";A
```

Figure 9.5 shows how to read this file. Since the field separation has been implemented in the create program, using the file as input is now quite natural. As before, ST is used to detect end of file.

```
100 REM READ AND DISPLAY FILE
110 PRINT "REWIND TAPE"
120 PRINT "PRESS ANY KEY TO GO ON"
130 GET A$:IF A$="" THEN 130
140 OPEN 1,1,0,"BUDGET"
150 PRINT CHR$(147)
160 PRINT "NAME","AMOUNT"
170 PRINT
180 IF ST=64 THEN 220
190 INPUT#1,N$,A
200 PRINT N$,A
210 GOTO 180
220 CLOSE 1
```

Figure 9.5 Program to read multiple fields

UPDATING A TAPE FILE

Most files store data that change frequently. This makes files superior to DATA statements. Instead of changing each program that uses the data, only the data file itself needs to be changed. These changes are then reflected as each program accesses the data.

For this exercise we will consider only how to change existing items in the file and not how to add or delete data. As a further limitation, we will limit the file to a maximum of 20 items. Obviously these are serious limitations but for now this approach will help us understand the concepts.

The program will read the budget file into two arrays. The first array, N$, will contain the names and the second, A, will hold the dollar amounts. After the necessary items in the array have been updated, the file will be rewritten from the array onto tape. Figure 9.6 contains the flowchart for this program and figure 9.7 contains the program.

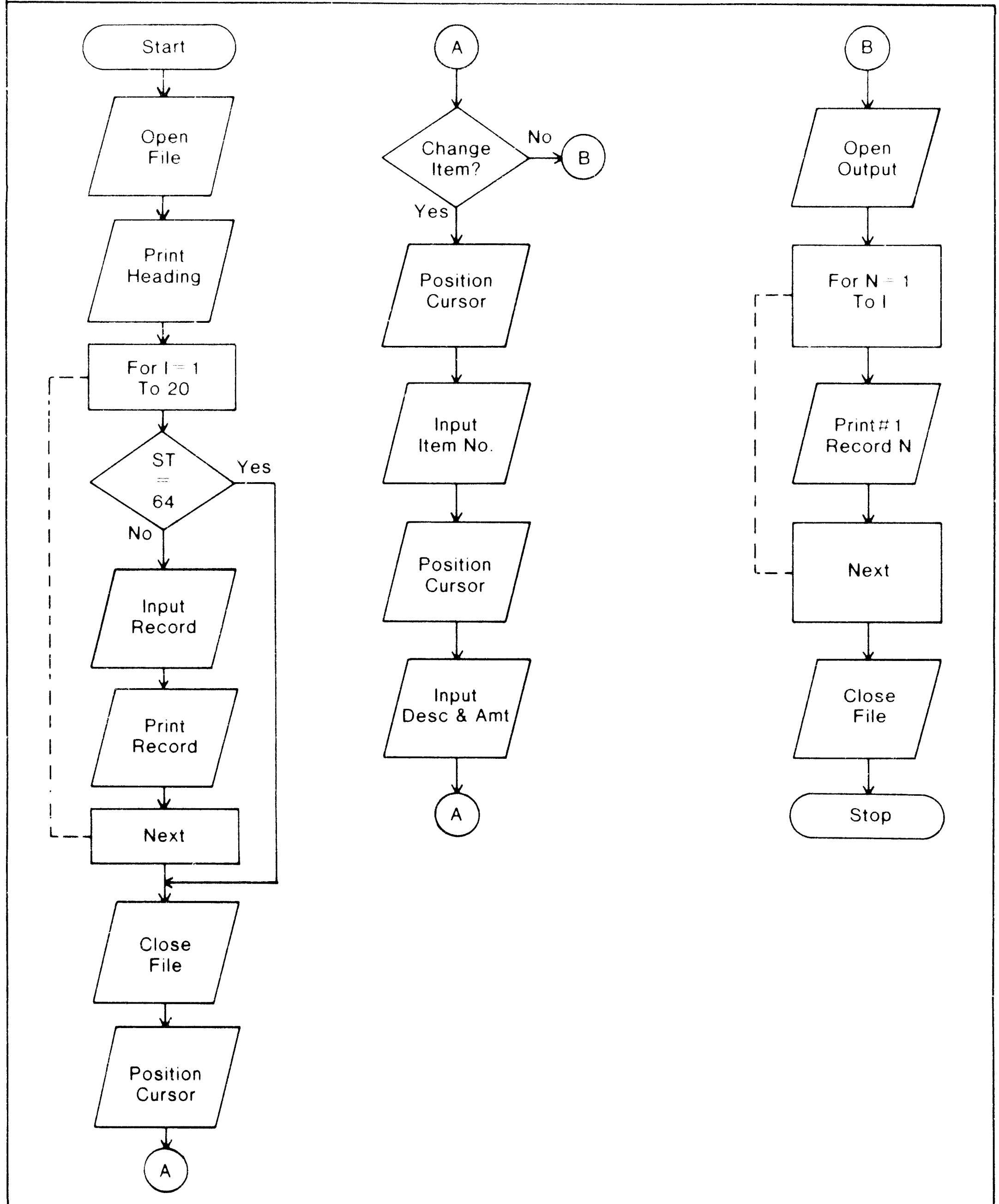
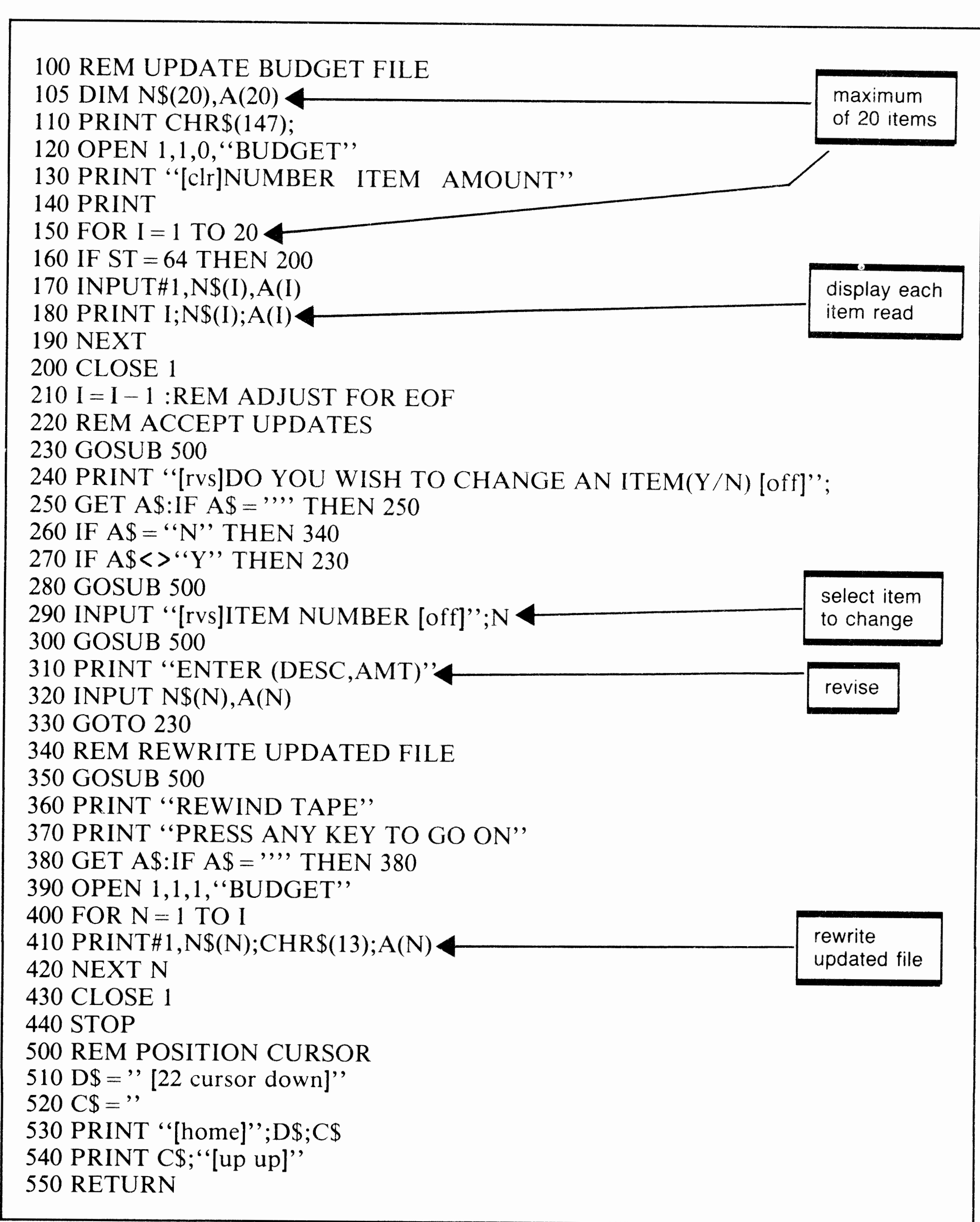


Figure 9.6 Updating a tape file



```
100 REM UPDATE BUDGET FILE
105 DIM N$(20),A(20)
110 PRINT CHR$(147);
120 OPEN 1,1,0,"BUDGET"
130 PRINT "[clr]NUMBER  ITEM  AMOUNT"
140 PRINT
150 FOR I=1 TO 20
160 IF ST=64 THEN 200
170 INPUT#1,N$(I),A(I)
180 PRINT I;N$(I);A(I)
190 NEXT
200 CLOSE 1
210 I=I-1 :REM ADJUST FOR EOF
220 REM ACCEPT UPDATES
230 GOSUB 500
240 PRINT "[rvs]DO YOU WISH TO CHANGE AN ITEM(Y/N) [off]";
250 GET A$:IF A$="" THEN 250
260 IF A$="N" THEN 340
270 IF A$<>"Y" THEN 230
280 GOSUB 500
290 INPUT "[rvs]ITEM NUMBER [off]";N
300 GOSUB 500
310 PRINT "ENTER (DESC,AMT)"
320 INPUT N$(N),A(N)
330 GOTO 230
340 REM REWRITE UPDATED FILE
350 GOSUB 500
360 PRINT "REWIND TAPE"
370 PRINT "PRESS ANY KEY TO GO ON"
380 GET A$:IF A$="" THEN 380
390 OPEN 1,1,1,"BUDGET"
400 FOR N=1 TO I
410 PRINT#1,N$(N);CHR$(13);A(N)
420 NEXT N
430 CLOSE 1
440 STOP
500 REM POSITION CURSOR
510 D$=" [22 cursor down]"
520 C$="
530 PRINT "[home]";D$;C$
540 PRINT C$;"[up up]"
550 RETURN
```


Figure 9.7 Program to update the budget file

THE CASH-FLOW PROBLEM

Who among us doesn't experience a cash-flow problem at times? The simple solution is to have more income than expenses, but who does? When we're faced with the pressures of day-to-day living, our expenses can at times exceed our income and then we are in financial difficulty.

In this section we will develop a program that will let us record our income and expenses, calculate the net gain or loss, and store all the data on tape for further reference and updating. The program will be flexible enough to permit recording on a weekly, monthly, or annual basis or on any other basis that may be suitable.

The first thing we must do in our planning is identify the categories of income and expenses the program will handle. Figure 9.8 shows a list of these as used in the program. If your needs differ, then the list and the program can be readily changed. These categories are included in DATA statements at the end of the program and will be loaded into arrays I$ for income and E$ for expenses for easy reference.

INCOME	EXPENSES	
Salary	Food	Loan
Dividends	Clothing	Education
Interest	Housing	Savings
Rents	Utilities	Travel
Allowances	Insurance	Entertainment
Other	Income tax	Other
	Property Tax	

Figure 9.8 Cash flow income and expense categories

The next consideration in developing the program is the major categories of activity the program will take. Since we are dealing with tape files, it is important to provide for the case where no existing file is present, such as at the start of a new cash-flow period. We also want to provide for reading an existing file.

Whether we're using an existing file or are creating a new one, the program must permit the entry of new data and the review or revision of old. Then we want to be able to check the bottom line. Are we in a gain or loss situation?

Finally, the program should provide for writing new or updated information on tape to be saved for later review.

This planning suggests we should develop a menu at the beginning of the program to display the choices. The menu is shown in figure 9.9.

```
CASH FLOW MENU

1 - NEW FILE

2 - OLD FILE

3 - ENTER/REVIEW DATA

4 - DISPLAY BALANCE

5 - WRITE FILE

6 - END PROGRAM
```

Figure 9.9 Cash-flow program menu

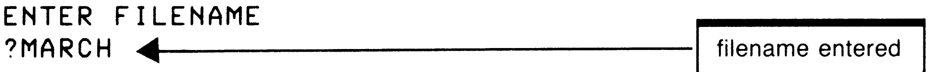
New File

When a new file is to be created, a 1 code would be selected from the menu. This should lead to a prompt that asks for the filename to be used.

The program will record this name to be used later when the file is saved. At this time the program should return to the main menu. Normally option 3 will be selected now since there is no data present. If choice 4 were made, a zero balance would be shown.

Old File

A code of 2 selected from the menu allows an existing file to be read into memory from tape. As for the selection of an old file, a prompt is given asking for the filename.



This entry will cause the program to search the tape for a file with the header MARCH and load it into the arrays storing the income and expenses.

Enter/Review Data

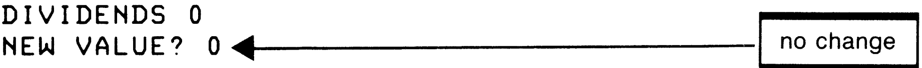
Following either the creation of a new file or the reading of an old one, the menu is displayed. Now a choice of 3 would usually be taken. We want to design choice 3 to permit the user of the program to do a number of things. One is simply to make entries into each category of income and expense. A second is to review the data already there, and a third is to add or subtract to/from current values.

All of this can be done without a great deal of complexity. When a code of 3 is chosen from the menu, the program will display a sequence as follows:

```
INCOME (+ OR -)
 0 FOR NO CHANGE

SALARY 0
NEW VALUE?1500
```

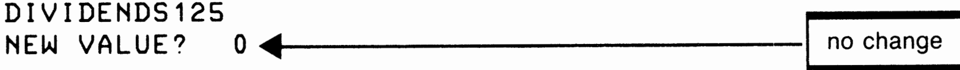
salary entered

The current income value for salary is displayed. It is presently zero, but a new value of $1,500 is entered. Now the program displays the next income entry:



Since we don't receive any dividends, zero is entered. If there had been an old file read and at a previous time we had entered a value of 125 for dividends, then the following display would have appeared:

By entering a negative value, the current amount of 125 is reduced by the amount entered. If dividends had not changed, a zero would be entered and no change would occur.



After all income has been reviewed and entered as required, the program proceeds to the expense categories. Prompts here are similar to the income except for the difference in categories.

```
EXPENSES (+ OR -)
 0 FOR NO CHANGE

FOOD    0
NEW VALUE? 450   <---- food value entered
```

Now the program proceeds through all of the expense categories. When finished, the program will return again to the main menu.

Display Balance

At this time, selecting code 4 will compute the total income and expenses and a net gain or loss. This choice produces another screen of information as follows:

```
INCOME    1500
EXPENSES  1250
NET       250
PRESS ANY KEY TO CONTINUE
```

After one has reviewed the good or bad news, pressing a key will return to the menu. Now changes may be made again (code 3); or if you select code 5 the data may be recorded on tape for future use.

Since this is a mildly complex program, a flowchart is first developed to put our planning into action. The flowchart is shown in figure 9.10. Notice the extensive use of subroutines here. Also of note is the way an ON GOSUB is depicted in the flowchart. This reflects the choices possible from the program's menu. The program follows in figure 9.11.

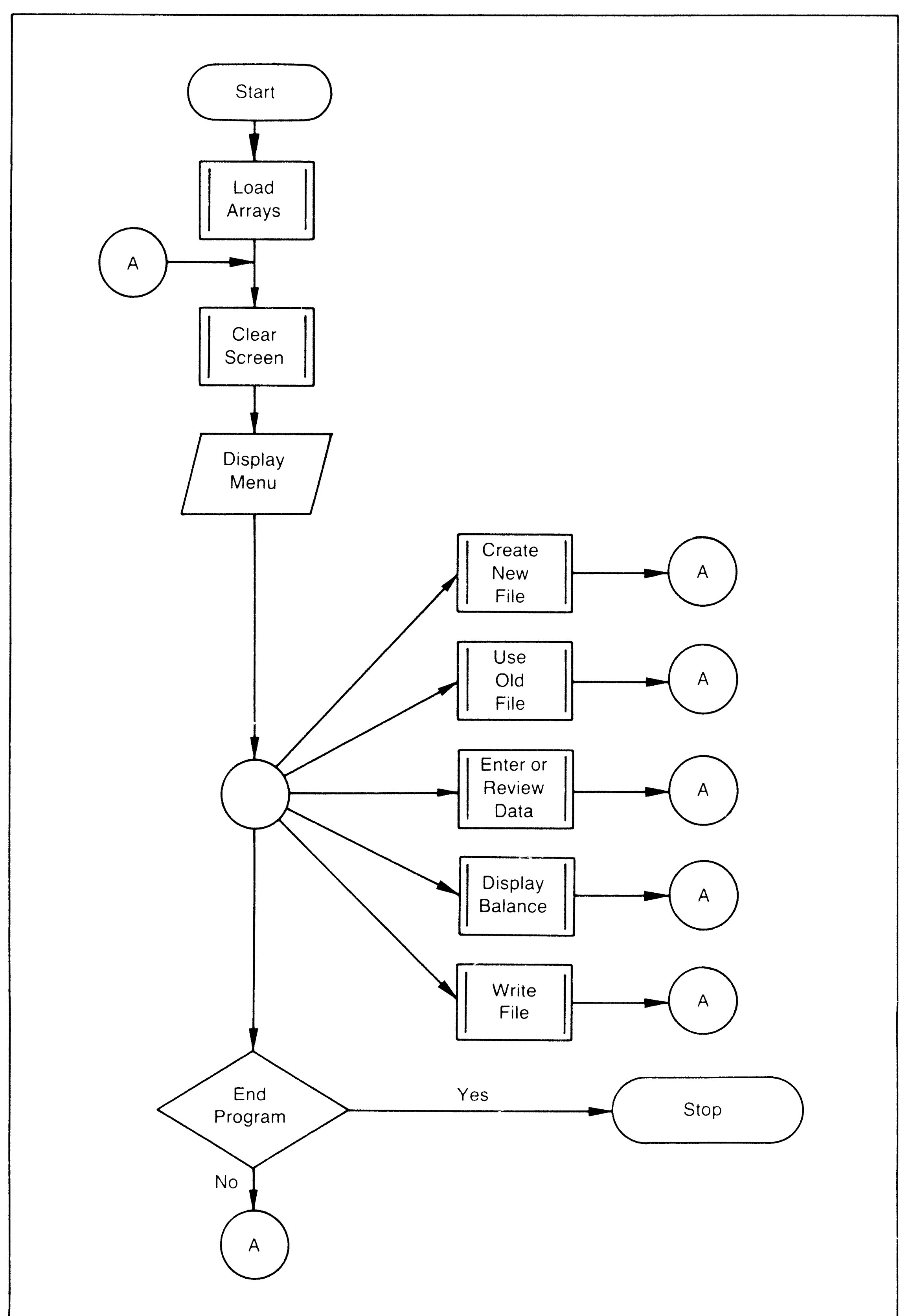


Figure 9.10 Flowchart for cash-flow program

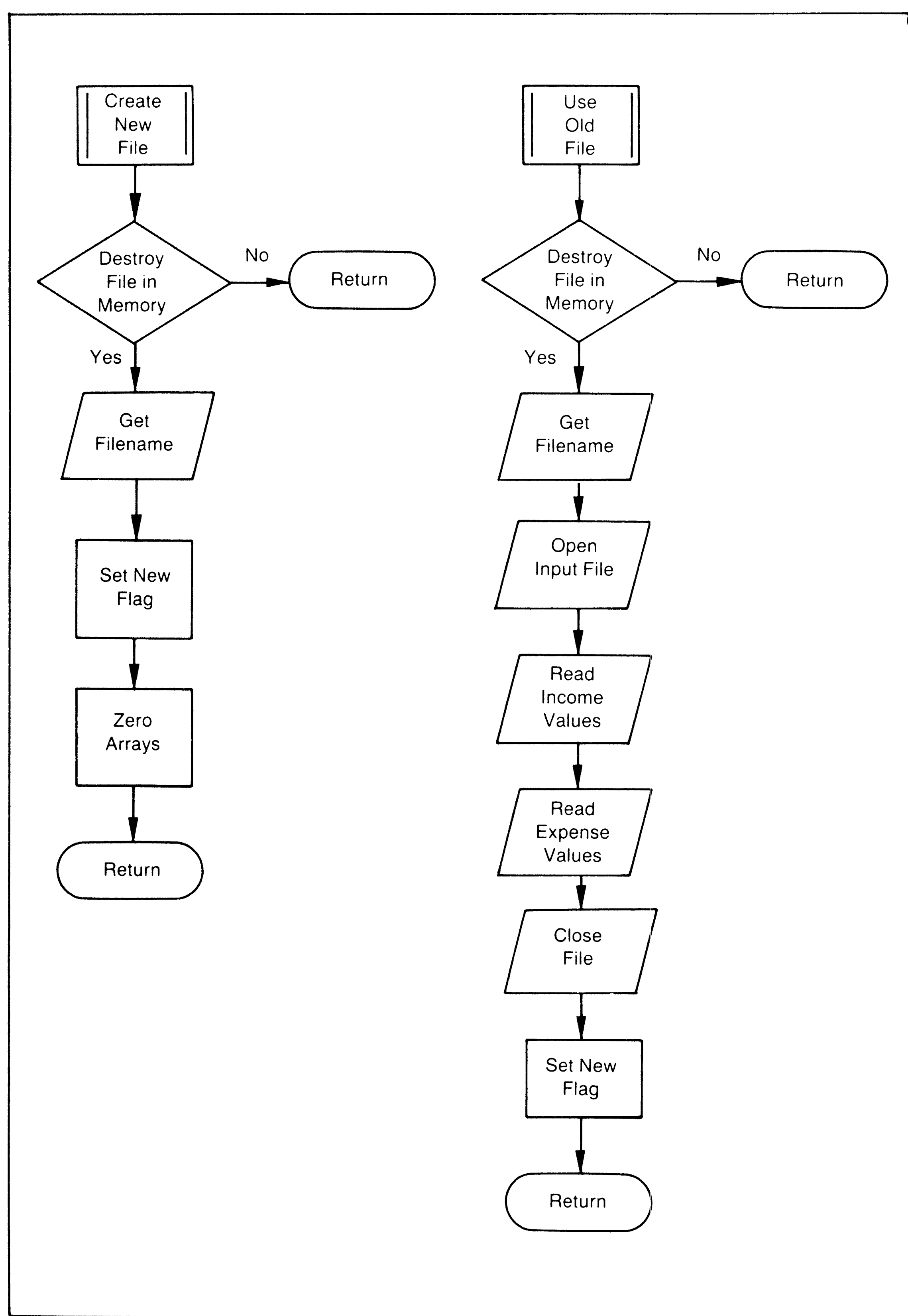


Figure 9.10 (continued)

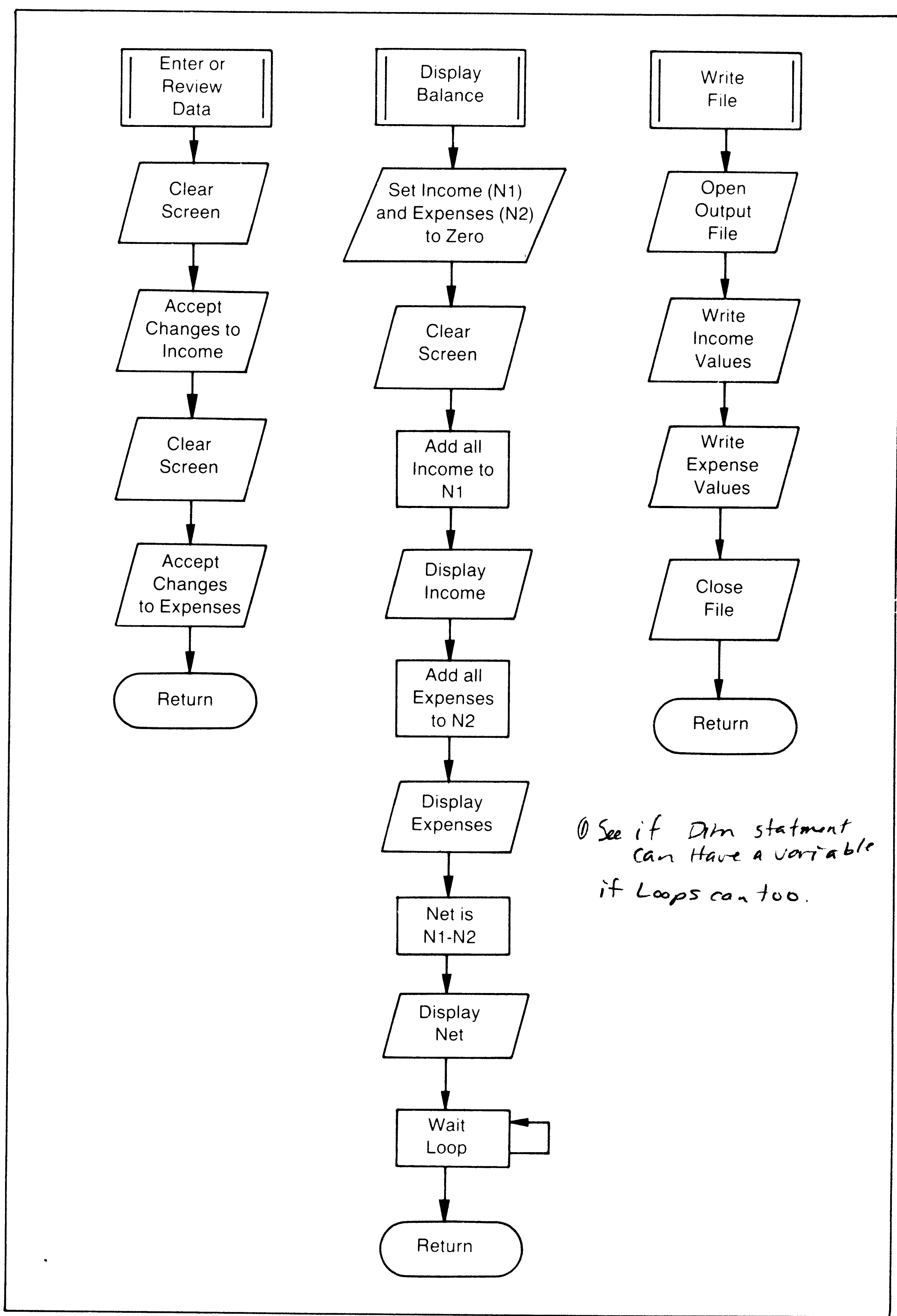


Figure 9.10 (continued)

```
100 REM MONTHLY CASH FLOW
110 DIM I$(6),I(6)
120 DIM E$(13),E(13)
130 GOSUB 1060
140 GOSUB 1030
150 PRINT "CASH FLOW MENU"
160 PRINT:PRINT "1 — NEW FILE"
170 PRINT:PRINT "2 — OLD FILE"
180 PRINT:PRINT "3 — ENTER/REVIEW DATA"
190 PRINT:PRINT "4 — DISPLAY BALANCE"
200 PRINT:PRINT "5 — WRITE FILE"
210 PRINT:PRINT "6 — END PROGRAM"
220 GET A$:IF A$ = "" THEN 220
230 IF A$ = "6" THEN STOP
240 A = VAL(A$)
250 ON A GOSUB 270,340,470,670,830
260 GOTO 140
270 REM CREATE NEW FILE
280 IF NF = 1 THEN GOSUB 980
290 GOSUB 930
300 NF = 1
310 FOR J = 1 TO 6:I(J) = 0:NEXT
320 FOR J = 1 TO 13:E(J) = 0:NEXT
330 RETURN
340 REM OLD FILE
350 IF NF = 1 THEN GOSUB 980
360 GOSUB 930
370 OPEN 1,1,0,(F$)
380 FOR J = 1 TO 6
390 INPUT#1,I(J)
400 NEXT J
410 FOR J = 1 TO 13
420 INPUT#1,E(J)
430 NEXT J
440 CLOSE 1
450 NF = 1
460 RETURN
470 REM ACCEPT DATA
480 GOSUB 1030
490 PRINT "INCOME (+ OR −)"
500 PRINT "0 FOR NO CHANGE"
510 PRINT
520 FOR J = 1 TO 6
530 PRINT I$(J);TAB(13);I(J)
540 INPUT "NEW VALUE";N
550 I(J) = I(J) + N
560 NEXT J
570 GOSUB 1030
580 PRINT "EXPENSES (+ OR −)"
590 PRINT "0 FOR NO CHANGE"
600 PRINT
610 FOR J = 1 TO 13
620 PRINT E$(J);TAB(13);E(J)
```

Figure 9.11 Cash flow program

```
630 INPUT "NEW VALUE";N
640 E(J) = E(J) + N
650 NEXT J
660 RETURN
670 REM DISPLAY BALANCE
680 N1 = 0:N2 = 0
690 GOSUB 1030
700 FOR J = 1 TO 6
710 N1 = N1 + I(J)
720 NEXT J
730 PRINT "INCOME";TAB(13);N1
740 FOR J = 1 TO 13
750 N2 = N2 + E(J)
760 NEXT J
770 PRINT:PRINT "EXPENSES";TAB(13);N2
780 NET = N1 - N2
790 PRINT:PRINT "NET";TAB(13);NET
800 PRINT: PRINT "PRESS ANY KEY TO CONTINUE"
810 GET A$:IFA$ = "" THEN 810
820 RETURN
830 REM WRITE FILE
840 OPEN 1,1,1,(F$)
850 FOR J = 1 TO 6
860 PRINT#1,I(J)
870 NEXT J
880 FOR J = 1 TO 13
890 PRINT#1,E(J)
900 NEXT J
910 CLOSE 1
920 RETURN
930 REM GET FILENAME
940 GOSUB 1030
950 PRINT "ENTER FILENAME"
960 INPUT F$
970 RETURN
980 REM CHECK OLD FILE
990 PRINT "DESTROY CURRENT FILE?"
1000 INPUT R$
1010 IF LEFT$(R$,1) = "Y" THEN RETURN
1020 GOTO 140
1030 REM CLEAR SCREEN
1040 PRINT CHR$(147)
1050 RETURN
1060 REM LOAD ARRAYS
1070 FOR J = 1 TO 6
1080 READ I$(J)
1090 NEXT J
1100 FOR J = 1 TO 13
1110 READ E$(J)
1120 NEXT J
1130 RETURN
```

Figure 9.11 (continued)

```
1140 DATA SALARY,DIVIDENDS,INTEREST,RENTS,ALLOWANCES,OTHER
1150 DATA FOOD,CLOTHING,HOUSING,UTILITIES,INSURANCE
1160 DATA INCOME TAX,PROPERTY TAX,LOAN,EDUCATION
1170 DATA SAVINGS,TRAVEL,ENTERTAINMENT,OTHER
```

Figure 9.11 (continued)

SUMMARY OF TAPE FILES

Tape is a slow but inexpensive means for storing data outside of your programs. As we have seen from the programs in this chapter, tape can be read only sequentially. To get to any record in the file, the program must read all the records prior to the one we really want. For some applications this can be a time-consuming wait, which leads to some frustration on the part of the program user.

This problem may be partially minimized by reading all the file into an array before the program begins. Although this will take time initially, no further reading of the file will be necessary no matter how long it runs. Of course this approach means there must be sufficient space in memory to store all the data, which is not always possible.

The last program in this chapter showed the basic method for updating a file by reading it into memory, making changes in it, and then rewriting it onto the tape. Again, this approach requires that there be sufficient space in memory to store the entire file. If there isn't enough room, one solution might be to divide the file into two or more separate files. But be careful about overwriting one file on another.

REVIEW QUESTIONS—CHAPTER 9

1. Describe how data appears on a tape file.
2. Tape files are called sequential files. What is the advantage of a sequential file? What are the disadvantages?
3. Why is it necessary to use OPEN with tape files? What information does the OPEN specify?
4. What must be done when a program has finished using a tape file?
5. When data is being PRINTed to tape, why is the tape not moving as each item of data is entered?
6. What commands are used to write data on tape and to read from tape?
7. You are writing a program to record subscription number (S), name (N$), and termination date (T$) for each subscription to a publication. Write a single print statement to create a tape record with these three fields.
8. Write a program that permits changes to the termination date on the file in question 7.

10 Disk Files

If tape files extend your reach beyond the capacity of the computer's main memory, this is even more true of the floppy disk. The 1541 floppy disk drive can store about 160,000 bytes of data on one 5 1/4 inch floppy disk.

Figure 10.1 shows how data or programs are written on the disk. Each floppy disk surface is divided into a number of tracks and each track into a number of sectors. The exact number of tracks and sectors is not terribly important to our discussion, but in any case the DOS (Disk Operating System) program in the disk drive takes care of all this technical stuff.

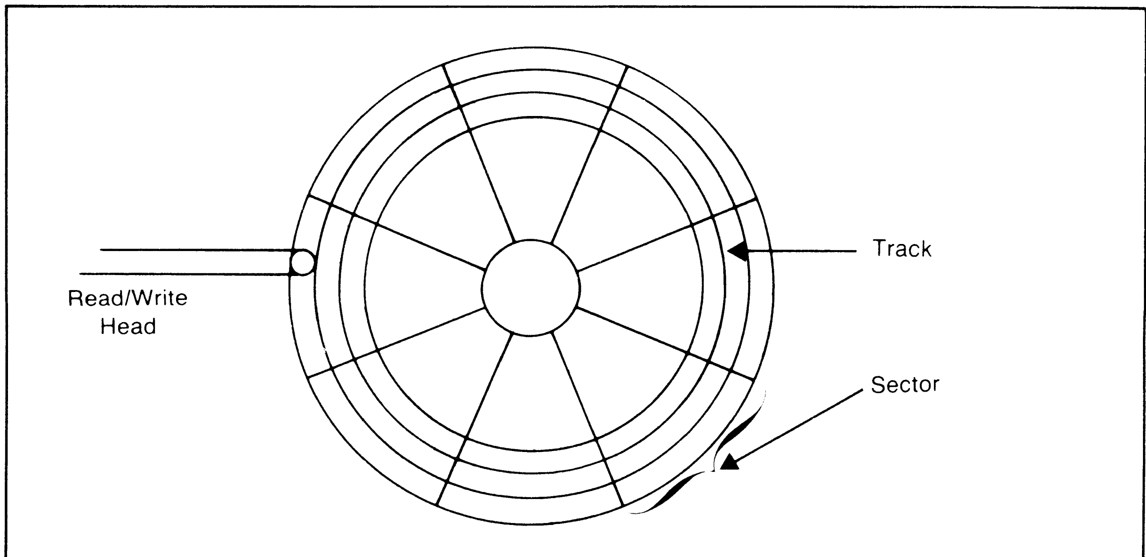


Figure 10.1 Floppy disk

When the floppy disk is inserted in the disk drive, the read/write head comes in contact with the surface of the disk. Under control of the program, data is written to the surface or read from the disk into the program. In addition to data files, programs may also be stored on the disk.

SEQUENTIAL FILES

Using a sequential file on disk is not much different from tape. Although the physical characteristics of disk are different from tape, records are logically organized in the same way. Each record on a sequential file logically follows the preceding record, as shown in figure 10.2. As with tape, a buffer in memory is filled with data before it is written to the disk.

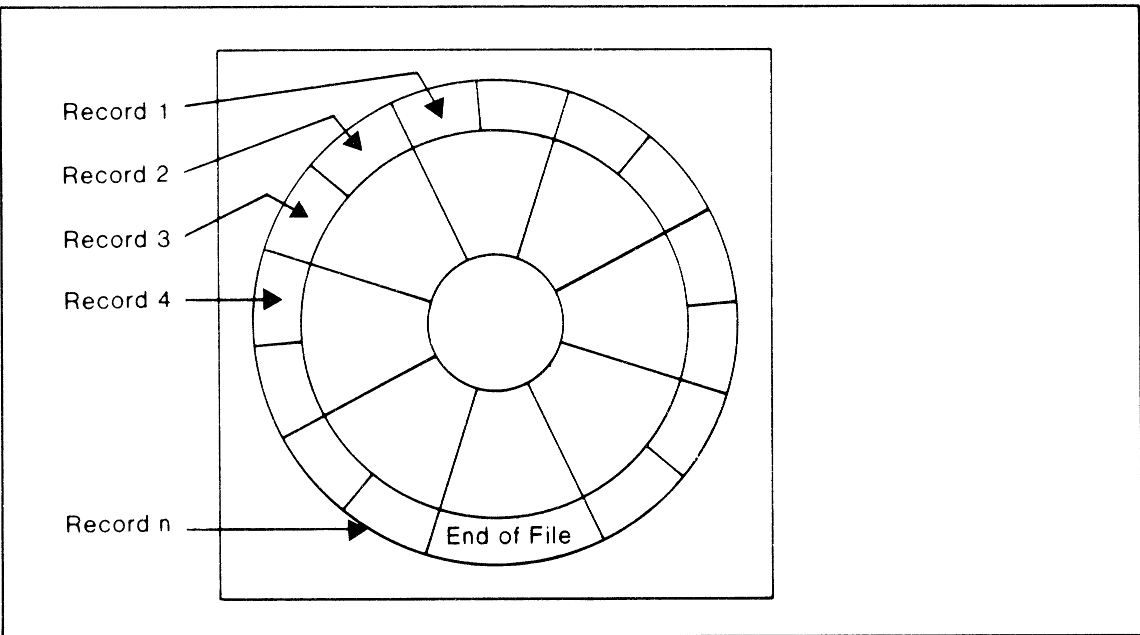


Figure 10.2 A sequential file on floppy disk

When we were using tape records, we discovered that to get to any record all the preceding records needed to be read. This is also true of sequential disk files. In fact, all the limitations for sequential tape apply to sequential disk. Well, that is all except for one: speed. Disk, as you will find, is considerably faster than tape file.

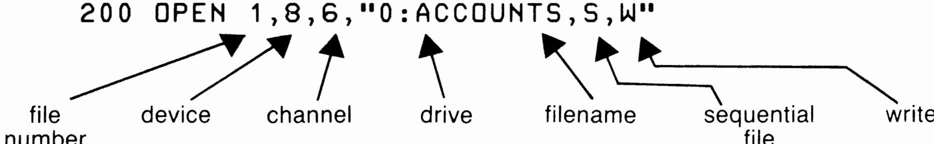
OPEN AND CLOSE

Disk uses its OPEN and CLOSE statements in a fashion similar to tape, which simplifies the use of disk files. The OPEN has a number of options for both sequential and relative files. To keep things simple, only the necessary items for sequential files will be shown now. Additional options are available for use with relative files.

OPEN file#,device#,channel#,string	
where	
file#	refers to the file number you will use in the program to reference this file. Normally a number from 1 to 255 will be used. Most programmers use a single-digit number for a file number.
device#	refers to the device allocated to disk. Normally device 8.
channel#	is the channel number. 1 to 14 may be used for data and 15 for the command channel.
string	defines the file and access method to be used. Its format is drive:"filename",S, {W / R} drive is the disk drive number. Normally 0.
"filename"	is the name that will be used to identify this file. When a file is to be read it will be opened only if the filename matches.
S	refers to a sequential disk file.
{W / R}	the W refers to an output file that will be Written. R refers to an input file that will be Read.

A few examples are in order now to explain how the OPEN may be used.

```
200 OPEN 1,8,6,"0:ACCOUNTS,S,W"
```



In this case a file named ACCOUNTS is opened as filenumber 1 on device 8 using data channel 1. The floppy disk will be in drive 0 and is to be written on as output.

```
150 OPEN 5,8,5,"0:CLASS,S,W"
```

A file called CLASS is opened as number 5. It will be written on as an output file.

```
300 OPEN 8,8,8,"0:RECORDS,S,R"
```

This file is number 8, and is called RECORDS. The file will be read as input using channel 8.

The command channel 15 is used to access error and operational messages from the disk. It may be opened with the statement

```
10 OPEN 15,8,15
```

By comparison, the CLOSE statement is quite simple, as shown in the following general format:

```
CLOSE fn
```

where
fn is the filenumber to be closed.

For example, the statement:

```
500 CLOSE 1
```

will close filenumber 1. No additional entries are needed in the CLOSE statement.

PRINT# AND INPUT#

Once the disk file has been opened, it can be referenced using the PRINT# and INPUT# statements in the same way as for tape files. The filenumber in both the PRINT# and INPUT# refers to the equivalent filenumber used in the OPEN statement. Here are some sample statements:

`100 PRINT#5,A$`	writes variable A$ on file 5
`200 PRINT#3,K`	writes numeric value K on file 3
`300 INPUT#1,X$`	reads string X$ from file 1
`400 INPUT#2,S`	inputs variable S from file 2

WRITING BUDGET NAMES ON DISK

In the tape chapter we wrote a few programs that created and read a sequential tape file for budget data. Now let's look at these same programs written for a sequential disk file. The first program (figure 10.3) writes a series of names of budget items on disk.

Functionally, this program appears the same to the user as the tape program. The program clears the screen and then prompts the user to enter an item or the word END to terminate the input. Each name is then written on the disk file.

```
100 REM CREATE SEQUENTIAL DISK FILE
110 PRINT CHR$(147):REM CLEAR SCREEN
120 OPEN 5,8,5,"0:BUDGET,S,W"
130 PRINT "ENTER ITEM NAME (END)"
135 INPUT N$
140 IF N$="END" THEN 170
150 PRINT#5,N$
160 GOTO 130
170 PRINT#5,"END"
180 CLOSE 5
```

Figure 10.3 Create sequential disk

There are a few subtle differences when we are creating a disk file. First is the use of the OPEN to open the file, give it a name, and assign the disk drive and channel. Once the file is opened, things proceed the same as for tape until the end of the data is reached. Tape files used the status indicator (ST) to test for end of file on the input. But since ST is not available for disk, we must create our own end of file indicator. This program simply writes an extra record on the disk file containing the word "END" to identify the end of file. The data on the disk will look like this:

RENT(cr)TELEPHONE(cr)LIGHT(cr)HEAT(cr)SUPPLIES(cr)END(cr)

When using a special record to indicate end of file, it is important to ensure the choice of word or value is not something that can legitimately appear as part of the data.

Now when this file is read, the program can test for the word "END" and after finding it, it will indicate end of file has been reached.

READING BUDGET NAMES FROM DISK

To read the budget names file, a program similar to the tape input program is required. Again there are some minor differences. One of these changes is how to test for end of file. The program is in figure 10.4.

```
200 REM READ AND DISPLAY BUDGET FILE
210 OPEN 5,8,5,"0:BUDGET,S,R"
220 PRINT CHR$(147)
230 PRINT "BUDGET ITEM"
240 PRINT
250 INPUT#5,N$
260 IF N$="END" THEN 290
270 PRINT N$
280 GOTO 250
290 CLOSE 5
```

Figure 10.4 Read sequential disk file

The end of file is checked in statement 260 immediately after the input is read. If the "END" record was read, then control of the program branches directly to statement 290, where the file is closed and the program terminates.

HOW TO HANDLE MULTIPLE FIELDS ON DISK FILES

Disk files also frequently use records consisting of more than one field. This situation is handled in the same way as tape — by writing a carriage return (CHR$(13)) between each field on the record. Don't forget to separate each field with a semicolon, not a comma.

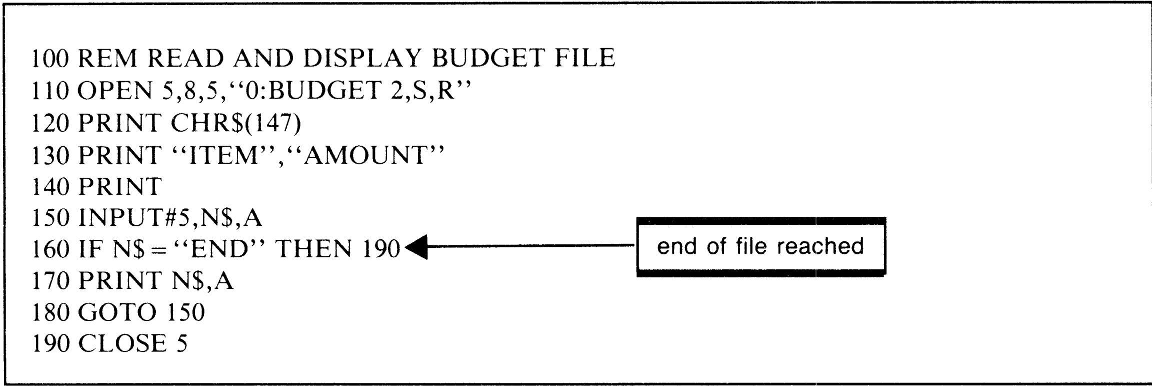
End of file on sequential disk requires a separate record, as we saw in the preceding examples. But since we are recording multiple fields, each field must be given a value. In the following program (figure 10.5), the name field (N$) will be given the value "END" and the amount field (A) will be given a zero value at end of file.

```
100 REM CREATE NEW FILE WITH MULTIPLE FIELDS
110 CR$=CHR$(13)
120 PRINT CHR$(147) :REM CLEAR SCREEN
130 OPEN 5,8,5,"0:BUDGET 2,S,W"
140 INPUT "ENTER ITEM NAME (END)"
145 INPUT N$
150 IF N$="END" THEN 190
160 INPUT "ENTER AMOUNT";A
170 PRINT#5,N$;CR$;A
180 GOTO 140
190 PRINT#5,"END";CR$;0 ◄──── end of file record
200 CLOSE 5
```

Figure 10.5 Create disk file with two fields

Note the use of the variable CR$ for the carriage return. Since carriage-return codes are used more than once in the program, line 110 assigns the value once to CR$, which then may be used as needed. This approach makes it easier to use the return code. It doesn't require defining the code completely each time with the CHR$ function.

Now let's look at a program (figure 10.6) to read this file:



```
100 REM READ AND DISPLAY BUDGET FILE
110 OPEN 5,8,5,"0:BUDGET 2,S,R"
120 PRINT CHR$(147)
130 PRINT "ITEM","AMOUNT"
140 PRINT
150 INPUT#5,N$,A
160 IF N$="END" THEN 190◄──── end of file reached
170 PRINT N$,A
180 GOTO 150
190 CLOSE 5
```

Figure 10.6 Read disk file with two fields

This program is really not much different from the tape program except for end of file handling, which we have already discussed. If at this point you haven't already entered these programs into your computer, why not give them a try and observe first-hand how they work? Then try to rewrite them to include some other fields, such as date, estimated amount, and actual amount. If you try this, it will be necessary either to use a different filename or to first scratch BUDGET 2 from your disk.

DETECTING DISK ERRORS

Occasionally an error will occur when either opening a file or reading or writing to the file. Usually the error, such as trying to write a file that already exists with the same filename, will be the fault of the program. Rarely will the error be a problem with the floppy disk or the disk drive. In any case, when there is an error the program needs to know about it, since to continue on blindly assuming nothing is wrong can be disastrous. In the previous programs, we did make that assumption but this is not advised for serious programs.

How do you know when an error occurs on the disk? Well, try entering the commands

```
10 OPEN 15,8,15
20 INPUT#15,EN$,EM$,ET$,ES$
```

and run the program. This operation will print the disk status string, which should appear as follows:

```
00 , OK, 00 ,00
```

This indicates there was no error on the previous disk operation.

The variables may have any name, but they represent the following

EN$—error number (See appendix)
EM$—error message
ET$—error track on disk
ES$—error sector on disk

Any value other than zero in EN$ would indicate an error of some kind had occurred during the last disk operation.

If a program attempted to open a file for output that already existed, the contents of the variables would be

```
63,FILE EXISTS,00,00
```

and EN$ would be 63. A complete list of error codes is listed in appendix D.

Now let's rewrite the last program that reads and displays the budget file and include code to check the disk status after each I/O operation.

```
100 REM READ AND DISPLAY BUDGET FILE
105 OPEN 15,8,15
110 OPEN 5,8,5,"0:BUDGET 2,S,R"
112 INPUT#15,EN$,EM$,ET$,ES$
115 IF EN$<>"00" THEN PRINT EM$:GOTO 190
120 PRINT CHR$(147)
130 PRINT "ITEM","AMOUNT"
140 PRINT
150 INPUT#5,N$,A
152 INPUT#15,EN$,EM$,ET$,ES$
155 IF EN$<>"00" THEN PRINT EM$:GOTO 190
160 IF N$="END" THEN 190
170 PRINT N$,A
180 GOTO 150
190 CLOSE 5:CLOSE 15
```

Figure 10.7 Check disk status code

At statement 112, 115, 152, and 155 the program now tests EN$ for a zero value that would indicate all is well. However, if EN$ is not equal to zero, then the program will print out the contents of EM$, which will give a description of the error that occurred. The program then branches to the CLOSE statement at 190 and the program is terminated.

If the program had attempted to open file "BUDGET 1" then the error

```
65,FILE NOT FOUND,00,00
```

would have been displayed.

Since the two groups of statements at 112 and 152 are identical, it would be an improvement to include these statements in a subroutine and use a GOSUB to activate them when needed.

MAINTAINING THE CHECKBOOK

A need common to us all is to maintain a checkbook for our deposits and withdrawals against a checking account. To write this program, we need to identify how to do several things. First is the need to create a new file with an opening balance. Next is the need to be able to make additional entries because of deposits and withdrawals against the account. And third is the ability to delete entries at the end of a month or similar time period, for otherwise the file will continue growing larger and larger.

Actually, these provisions are not uncommon for many types of files, for example, an accounts payable or receivable file, an inventory file, or a file of students. Although each application will have different specific needs, some of the basics developed here will not need to change much.

The procedure for updating a sequential disk file (the checkbook) will be the same as for tape. In other words, this program will also read the file into memory using arrays, update the data in the arrays, and then rewrite the updated file back onto the disk. The updated file then replaces the old file, which is scratched from the disk. For this reason it is vitally important that the program be thoroughly tested, since any errors may cause damage to the disk records, which cannot then be retrieved.

REPLACING A CURRENT FILE

How do we tell the disk that we want to replace a file instead of writing a new one? This is done by a code in the OPEN statement. The code is an "@" used as follows:

```
100 OPEN 3,8,3,"@0:CHECKFILE,S,W"
```

You will recognize this open as a disk open for filenumber 3 with a filename of "CHECKFILE." The file is an output file. Notice the use of the @ immediately before the drive number so the string appears as "@0:CHECKFILE,S,W". This use of the @ indicates we are opening an existing file that is to be replaced with new or revised data.

USING VARIABLE FILENAMES

The program we are about to write also has another need. We would like to accept any filename as input to the program. So we are not limited to just one but could access any number of possible files. This is a more realistic approach to disk since one floppy can store a number of different files. This means we need to use a variable for the filename. Here is an example of how a filename might be entered and then used in the OPEN statement:

```
100 INPUT "ENTER FILENAME";F$
105 S$="0:"+F$+",S,W"
110 OPEN 3,8,3,S$
```

This example uses the variable S$ to provide the filename in the OPEN statement. Now statement 110 will open whatever file the user specifies in response to the input query in line 100. But if this is a file to be replaced, we will run into difficulty. It might be possible to ask the user to enter the @, drive number, and filename. This might be acceptable, but it would be much cleaner if the program could take care of this detail. Here's how it could be done:

```
100 INPUT "ENTER FILENAME";F$
105 S$="@0:"+F$+",S,W"
110 OPEN 3,8,3,S$
```

Now line 105 concatenates the @ to the disk number and the user need only enter the name of the file, which is a more natural response.

ENGLISH CODE

Having completed these preliminary discussions, we can now begin developing the program logic. For this program we will once again use English code.

This program makes extensive use of subroutines, which is the approach you should be taking by now in all your programs. As much as possible, strive for generality in all your subroutines so they may be used in other programs as the need arises. Later it will become evident that the solution we have developed here could be made even more general.

Start-Up Menu

1. Create new checkbook file
2. Update checkbook file
3. Stop

Create New Checkbook File Subroutine

1. Accept filename
2. Accept date and opening balance
3. Write entry on new file

Update Checkbook File Subroutine

1. Accept filename
2. Load file into arrays
3. Add, change menu until done
   3.1 If add call add data to checkbook
   3.2 If change call change data on checkbook
4. Rewrite updated file on disk
5. Return

Add Data to Checkbook Subroutine

1. Accept data until done
   1.1 Accept date, desc, with/dep
   1.2 Append to end of table
   1.3 Calculate and display new balance
   1.4 If table full display *warning*
2. Return

Change Data on Checkbook Subroutine

1. Display items until end of table
    1.1 If screen full (20 items) then
        1.1.1 Proceed or change items?
              If change call accept change or delete
    1.2 Proceed to next screen

Accept Changes and Deletions Subroutine

1. If change then call change entry
2. If delete then call delete entries
3. Return

Change Entry

1. Accept entry number
2. Change item
3. Readjust balance
4. Return

Delete Entries

1. Accept entry number
2. Move following items back in array
3. Reset end of table
4. Display screen page with changes
5. Return

The English code at this level is still general. For instance, it doesn't go into detail about how each menu will appear in the program. Neither does it specify exactly what subscripting will be used when referencing array elements. We'll develop this next level of detail in the program code itself.

However, there are a few places in the solution at this stage that may not be at all clear. For example, in the "Change Data on Checkbook Subroutine", reference is made to displaying a screen of data. Also the "Delete Entries" subroutine makes a casual reference to moving items back in the array. Both of these statements need clarification.

DISPLAY SCREEN PAGE

Checkbooks normally have pages with deposits and withdrawals shown and the related balance, date, and description of the entry. In this program we would like to treat the screen display as a page of data rather than one line at a time. To do this we need to display contents of the arrays until one page is shown. Then changes to that page can be accepted.

Figure 10.9 shows how lines from the arrays are displayed to create one page of output. At the end of the page is a prompt asking if there are any changes to be made on this page or if the user wishes to proceed to the next page. If a "C" is entered, then a new prompt asks for the type of change, while a "P" response would cause the next page to be displayed on the screen.

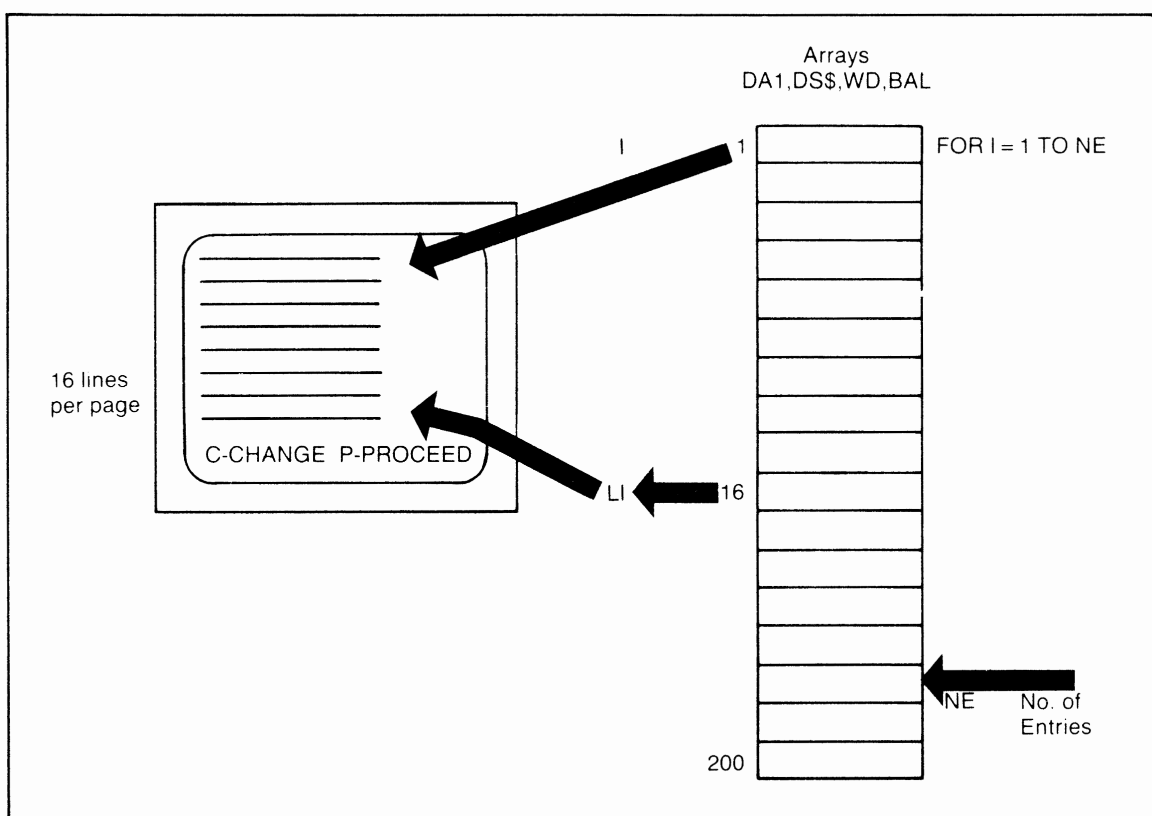


Figure 10.9 Display screen page

The variable LI is used to count the number of lines displayed on the screen. When LI has reached 16, the loop is put on hold temporarily until the user can respond to the prompt. When the command to proceed is given, the program resets LI to zero, restarts the loop, and displays the next 16 lines. At some point there may be fewer than 16 lines left to display. In this case the loop will terminate with fewer than 16 lines on the page.

DELETING TABLE ENTRIES

Another interesting problem to solve is deleting items from the table. This might be done at the end of a month when last month's items are deleted, or if not last month's items then items from several months ago or even a year ago. It is the user's choice.

Assuming entries will be made in date sequence, the program will be designed to delete all entries from a specified position on the page back to the beginning of the checkbook. The user specifies the position by entering the line number of the entry, which has been displayed with the other checkbook fields. Figure 10.10 shows the procedure for deleting table items.

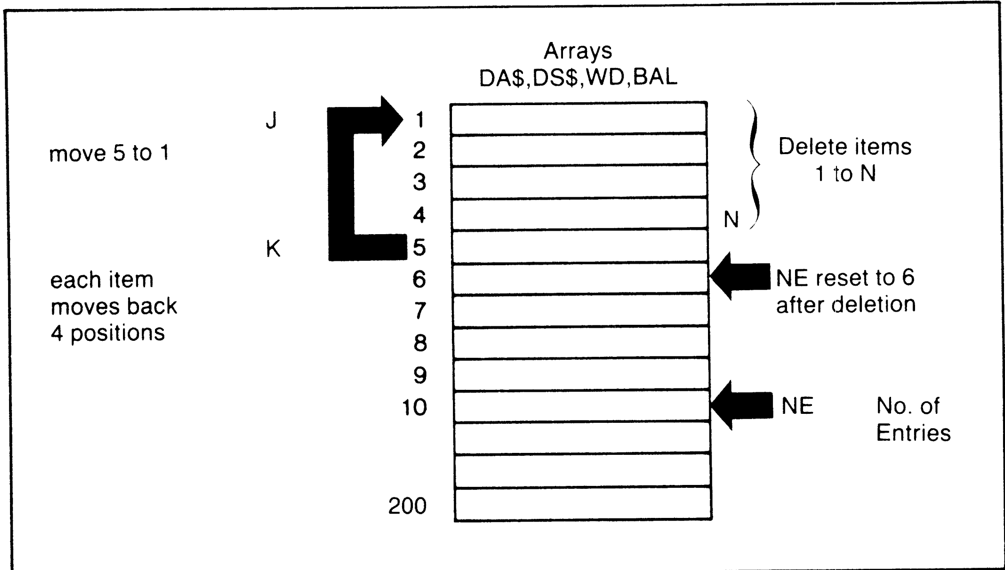


Figure 10.10 Deleting four table items

If the user response indicates that items 1 to 4 are to be deleted, then all items from item 5 to the end of the data (NE) are shifted back four positions in the arrays. In other words, item 5 is moved back to position 1, item 6 to position 2, item 7 to 3, and so on.

When all data up to position NE have been moved, the deletion is complete. Then NE is reduced by 4 (since four items were deleted) and is now at position 6, referring to the current number of entries in the table. One problem with this technique is that the more data the arrays contain, the more time the program will take to do a deletion. Using linked lists, an advanced technique beyond the scope of this book, would improve this time factor considerably.

The complete checkbook program is shown in figure 10.11.

```
80 MA=50
90 OPEN 15,8,15   — not needed
100 DA$(200),DS$(200),WD(200),BAL(200)
110 CR$=CHR$(13)
120 REM MAINTAINING THE CHECKBOOK
130 REM START-UP MENU
140 PRINT "[clr]";TAB(7);"CHECKBOOK"
150 PRINT
160 PRINT "C — CREATE CHECKBOOK"
170 PRINT
180 PRINT "U — UPDATE CHECKBOOK"
190 PRINT
200 PRINT "X — EXIT"
210 GET A$:IF A$="" THEN 210
220 IF A$="X" THEN CLOSE 15:STOP   close 1
230 IF A$="C" THEN GOSUB 260
240 IF A$="U" THEN GOSUB 380
250 GOTO 140
260 REM CREATE NEW CHECKBOOK
270 GOSUB 750
280 PRINT "DATE OF ENTRY(MM/DD)"
285 INPUT D$
290 PRINT "OPENING BALANCE"
295 INPUT BL
```

Figure 10.11 Checkbook update program

```
300 OPEN 3,8,3,"0:"+F$+",S,W"
310 GOSUB 790
320 PRINT#3,1
330 GOSUB 790
340 PRINT#3,D$;CR$;O;CR$;BL
350 GOSUB 790
360 CLOSE 3
370 RETURN
380 REM UPDATE CHECKBOOK FILE
390 REM LEAD CHECK FILE
400 GOSUB 750
410 OPEN 3,8,3,"0:"+F$+",S,R"
420 GOSUB 790
430 PRINT "LOADING FILE ";F$
440 FOR D=1 TO 300:NEXT D
450 INPUT#3,NE
460 FOR I=1 to NE
470 INPUT#3,DA$(I),WD(I),BAL(I)
480 GOSUB 790
490 NEXT I
500 CLOSE 3
510 REM UPDATE MENU
520 PRINT "[clr]";TAB(6);"UPDATE MENU"
530 PRINT
540 PRINT "A — ADD ENTRIES"
550 PRINT
560 PRINT "C — CHANGE ENTRIES"
570 PRINT
580 PRINT "X — EXIT"
590 GET U$:IF U$="" THEN GOTO 590
600 IF U$ = "X" THEN 650
610 IF U$ = "A" THEN GOSUB 840
620 IF U$ = "C" THEN GOSUB 1050
630 GOTO 520
640 REM SAVE UPDATED FILE ON DISK
650 PRINT REWRITING FILE ";F$
660 OPEN 3,8,3,"@0:"F$+",S,W"
670 PRINT#3,NE
680 GOSUB 790
690 FOR I=1 TO NE
700 PRINT#3,DA$(I);CR$;WD(I);CR$;BAL(I)
710 GOSUB 790
720 NEXT I
730 CLOSE 3
740 RETURN
750 REM ACCEPT FILENAME
760 PRINT "[clr]ENTER FILENAME"
770 INPUT F$
780 RETURN
790 REM CHECK DISK STATUS
800 INPUT#15,EN$,EM$,ET$,ES$
805 IF EN$="00"THEN RETURN
```

Figure 10.11 (continued)

```
810 PRINT EM$
820 CLOSE 15
830 STOP
840 REM ADD DATA TO CHECKBOOK
850 PRINT "[clr]";"ENTER NEW DATA"
860 PRINT
870 PRINT "CURRENT BAL = ";BAL(NE)
880 PRINT
890 PRINT "ENTER DATA AS FOLLOWS:"
900 PRINT
910 PRINT "(MM/DD) OR (QUIT)"
915 INPUT D$
920 IF D$ = "QUIT" THEN 1040
930 NE = NE + 1
940 DA$(NE) = LEFT$(D$,5)
960 INPUT "DEPOSIT 0[3 lt's]";D
970 INPUT "WITHDRAWAL 0[3 lt's]";W
980 WD(NE) = D - W
990 BAL(NE) = INT((BAL(NE - 1) + WD(NE))*100 + .5)/100
1000 PRINT
1010 PRINT "NEW BAL = ";BAL(NE)
1020 IF NE = MA THEN PRINT "*WARNING* — TABLE FULL"
1030 GOTO 900
1040 RETURN
1050 REM CHANGE DATA ON CHECKBOOK
1060 PRINT "[clr]NO. DATE WITH/DEP"
1070 PRINT
1075 PRINT "OPENING BALANCE";BAL(1)
1080 FOR I = 1 TO NE
1090 PRINT I;" ";DA$(I);TAB(15);WD(I)
1100 LI = LI + 1
1110 IF LI<16 THEN 1130
1120 GOSUB 1150
1130 NEXT I
1140 I = I - 1:GOSUB 1150
1150 REM ACCEPT CHANGES TO THIS PAGE
1160 IF LI = 0 THEN RETURN
1165 PRINT:PRINT "BALANCE";TAB(14);BAL(I)
1170 PRINT
1180 PRINT "CHANGE OR PROCEED"
1190 GET P$:IF P$ = "" THEN 1190
1200 IF P$ = "C" THEN GOSUB 1230
1210 LI = 0
1220 RETURN
1230 REM ACCEPT CHANGES AND DELETIONS
1240 PRINT"CHANGE DELETE EXIT"
1250 GET P$:IF P$ = "" THEN 1250
1260 IF P$ = "C"THEN GOSUB 1300
1270 IF P$ = "D" THEN GOSUB 1460
1280 IF P$ = "X" THEN RETURN
1290 GOTO 1240
1300 REM CHANGE ENTRY
1310 INPUT"NUMBER OF ENTRY ";N
```

Figure 10.11 (continued)

```
1320 PRINT DA$(N);" ";WD(N)
1330 PRINT
1340 PRINT "ENTER CHANGES AS FOLLOWS:"
1350 INPUT "DATE ";D$
1355 DA$(N)=LEFT$(D$,5)
1370 INPUT "DEPOSIT 0[3 lt's]";D
1380 INPUT "WITHDRAWAL 0[3 lt's]";W
1390 AD=D-W-WD(N):REM ADJUSTMENT TO BALANCE
1400 IF AD=0 THEN 1450
1410 WD(N)=WD(N)+AD
1420 FOR K=N TO NE
1430 BAL(K)=INT((BAL(K)+AD)*100+.5)/100
1440 NEXT K
1450 RETURN
1460 REM DELETE ENTRY
1470 INPUT"DELETE NO. 1 TO ";N
1480 J=1
1490 FOR K=N+1 TO NE
1500 DA$(J)=DA$(K)
1520 WD(J)=WD(K)
1530 BAL(J)=BAL(K)
1540 J=J+1
1550 NEXT K
1560 NE=NE-N
1570 GOSUB 1590
1580 RETURN
1590 REM DISPLAY 1 PAGE
1600 I=16
1610 IF I>NE THEN I=NE
1620 PRINT "NO. DATE WITH/DEP"
1630 PRINT
1640 FOR K=1 TO I
1650 PRINT K;" ";DA$(K);TAB(15);WD(K)
1660 NEXT K
1670 PRINT:PRINT "BALANCE";TAB(14);BAL(I)
1680 RETURN
```

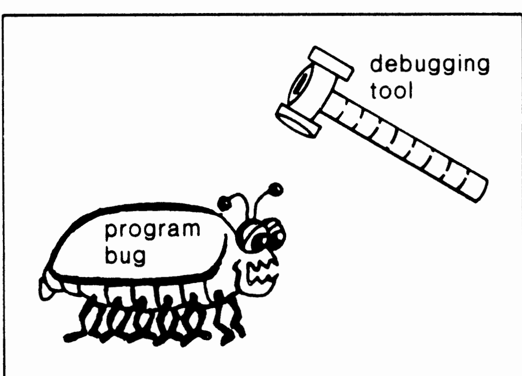
Figure 10.11 (continued)

REVIEW QUESTIONS—CHAPTER 10

1. Describe the organization of a floppy disk.
2. What kinds of files may be used on disk? Discuss the pros and cons of each.
3. Describe OPEN and CLOSE statements for the disk. What defaults may be used in the OPEN?
4. How is end of file handled on a sequential disk file?
5. What is the purpose of the command channel? How is it used for disk applications?
6. The questions that follow refer to the checkbook program in this chapter.
   a. Examine the routine that deletes entries from the checkbook. What happens to the balance when a delete occurs?
   b. Subroutine 1300 accepts changes to entries but requires that all of the data be retyped even if only one component is changed. How could this subroutine be changed so that only changes need to be retyped?
   c. When more than 16 items are in the checkbook, the program scrolls the screen to the next 16 items. How could these begin with a new screen and display from the top?

11 How to Debug Your Programs

Throughout this book we have spent a lot of time writing BASIC programs for the C-64, using many of the features of the BASIC interpreter. In every case the programs we have used were correctly written (I hope!) and would run as expected. But is this assumption realistic? Admittedly, some of the programs did not work the first time and required changes and corrections until they did what was intended. The process of finding and correcting errors is known as debugging the program.



To debug a program successfully requires the development of new skills to supplement your programming skills. By following certain steps, a program may be debugged and made to work in the way you originally intended.

Here are the main things we will consider when debugging a program:

1. Desk checking
2. Syntax errors
3. Test data preparation
4. Immediate mode debugging
5. Tracing program logic

Generally, testing proceeds in this order, but changes to the program at any time may require you to go back to any of the earlier steps. This may be the result of errors that were overlooked previously or because of new errors that were introduced during the debugging process.

DESK CHECKING

Desk checking is probably the easiest of all debugging techniques and yet the most often overlooked. First use one of the program planning techniques we discussed. Either a flowchart or English code will help in the planning stage. It has been proven that programmers who take the time to plan will have substantially fewer errors in their programs.

After carefully planning the solution, write the C-64 BASIC code. Unless the program is very simple, the writing should be done on paper, not on the C-64 since the 25 lines can make reading the program awkward. At this point, take some time to desk check the code by simply reading what you have written. It is surprising how many errors can be detected at this stage without the computer's assistance. Types of errors to look for at this stage are errors in spelling or logic, missing or duplicate statements, and statements out of order.

SYNTAX ERRORS

In a formal sense, *syntax* refers to improper punctuation of statements in the language, whereas another term, *semantics*, refers to the proper understanding of the statement format. This is generally discussed throughout the book, so a detailed presentation will not be given here.

Basically, if you take care to follow the statement formats, your program will be correct as far as BASIC is concerned. If you make a mistake, the C-64 will give you an error message when you attempt to RUN the program. For example, the statement

```
10 FOR I=1,10
```

may sound O.K., but when the program is run the error message

```
?SYNTAX
ERROR IN 10
```

will be displayed on the C-64's screen. The statement's syntax is incorrect because the format of a FOR statement requires the keyword TO instead of a comma between the 1 and 10. Therefore the statement:

```
10 FOR I=1 TO 10
```

is the correct one.

Syntax errors are many and varied. In a number of cases it is difficult to determine what has caused the error. Although there are precise definitions of errors, we are more concerned with finding and correcting them. The following is a list of errors that programmers frequently encounter. Most of them are identified by error messages from the BASIC interpreter.

Type of Error	Example	Explanation
Punctuation	10 N(I K) = A + B	The comma separating subscripts I and K is missing. Two dimension arrays require 2 subscripts separated by a comma.
	10 PRINT A B	Two or more variables in input and output statements must be separated by commas. C-64 BASIC treats this as variable AB rather than an error, so this type of error is easy to miss.
Parentheses	10 N = N + 1)*5 20 A = (K + 1*5 + (J/N)))	Either too few brackets (statement 10) or too many (20) can be the cause of this problem. BASIC flags these as syntax errors. Make sure that all brackets are matching pairs.
Operator	10 N + N = K	The equal operator is on the wrong side of the expression.
	50 J = B(N + L)	The multiply operator is missing between B and the parenthesis. C-64 assumes B to be an array with (N + L) the subscript, so again this is an error easily overlooked.
	70 K = P*/J	In this example two operators are used together; a variable is probably missing.
Mismatch	200 N = "DON CASSEL"	This occurs when an alphanumeric value is assigned to a numeric variable or vice versa. C-64 calls this a TYPE MISMATCH ERROR.
Illegal Verb	10 S = SQRT(256)	An illegal verb is also a class of syntax error. This happens when a keyword is misspelled. SQRT is not legal in C-64 BASIC.

During program execution, other errors, which are not indicated earlier, may be found. These are generally self-explanatory errors, such as dividing by zero, trying to calculate the square root of a negative number, and having arithmetic overflows or subscripts out of range.

For example, in the following case in which a constant is used:

```
10 DIM NO(15)
   .
   .
80 NO(21)=T * K
```

it is evident that the subscript of 21 is the culprit; maybe it should have been 12, or possibly NO should have been dimensioned with more elements. However, if the statement in error had been

```
80 NO(I)=T * K
```

then it would be necessary to trace the values of I to see why it had exceeded the permissible range of the subscript.

TEST DATA PREPARATION

The first rule of test data preparation is that quality and not quantity counts. Beginning programmers often prepare a lot of test data that looks impressive but is often quite meaningless. Each item of test data should have a specific purpose and therefore test for a specific potential problem. Some of the things for which test data should be prepared are:

1. Sequence error—for tape or disk files.
2. Missing data.
3. Positive and negative values.
4. Valid and invalid codes.
5. Numbers within a specific range, including the first and last numbers of the range.
6. Too much or too little data when using tables.
7. Reasonableness. An hourly rate of more than $100 is probably unreasonable.
8. Length. Fields such as phone numbers, account numbers, and social security numbers are of predefined lengths and can be checked.

Not every program will need testing for all these items, and some programs will need testing for other factors. The key is to be aware of possible sources of error and to consider them when testing your programs.

A final consideration is that test data should be supplied to check every statement of code written in the program. Don't assume that the program code will work just because it is obvious or simple. Errors do not happen solely in complex code; they often occur in the most unlikely places.

IMMEDIATE MODE DEBUGGING

One of the advantages of using a microcomputer like the C-64 is the capability of examining the contents of program variables directly on the screen. Earlier in the book we discussed the use of the computer as a sophisticated calculator using immediate mode. This same mode can be used to examine the contents of the program's variables while debugging the program.

Immediate mode debugging is particularly useful if the program terminates prematurely. For instance, you are running a test of the following program:

```
10 DIM A$(100)
20 B$="STRING "
30 FOR I=1 TO 100
40 B$=B$+B$
50 A$(I)=B$
60 NEXT I
```

and the program stops with the message

```
?STRING TOO LONG ERROR IN 40
```

Now from the program code we decide that B$, which is in statement 40, should contain the string "STRING STRING." At least that's what we intended. This can be immediately confirmed or denied by using the immediate mode to display the contents of B$ as follows:

```
?B$
```

The VIC responds with the following display:

```
STRING STRING STRING STRING STRING STRIN
G STRING STRING STRING STRING STRING STR
ING STRING STRING STRING STRING STRING S
TRING STRING STRING STRING STRING STRING
STRING STRING STRING STRING STRING STRI
NG STRING STRING STRING
```

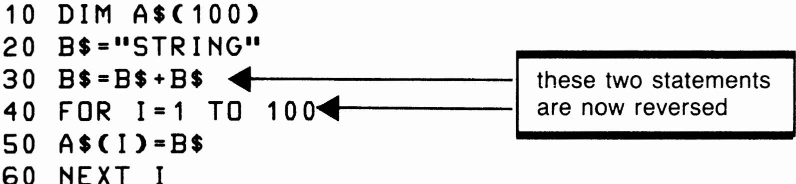
This is obviously not what we wanted. We can also examine I's value:

```
?I
6
```

This tells us that the loop had executed six times when the program terminated. We could try a lot of other things at this point. For instance, we could change the FOR loop to repeat only three times and then check B$'s value. It would contain fewer repetitions but still more than the two required.

By now we should have a good idea of what is wrong. The FOR loop has affected the variable B$ when our expectation was that it would be assigned the string "STRING" concatenated to itself only once. The problem is that statement 40 should precede the loop and not be contained within it.

The program is changed and now appears as follows:




```
10 DIM A$(100)
20 B$="STRING"
30 B$=B$+B$
40 FOR I=1 TO 100
50 A$(I)=B$
60 NEXT I
```

TRACING PROGRAM LOGIC

Tracing is the process of following the program logic step by step to locate an error. This can be done manually for relatively easy problems, or program statements can be used to show the flow of the logic. The key is to know what to expect from your program. If you know what output the program is intended to produce, then when it does not produce this output, a trace can be used to determine why.

Manual Tracing

Trace manually with a printed copy of the program listing (or read it from the screen instead), with a pencil at hand. This method is particularly useful for small programs, simple bugs, or when there is no output to check.

To show how this works, let's write a program to display a chart of Celsius and Fahrenheit temperatures. Actually, the chart may be one we eventually want to print, but it is good debugging technique to display it first. This can save a lot of paper.

The chart is to include Celsius temperatures from 10 to 33 and their equivalent Fahrenheit temperatures as integer values. We expect the output to appear as follows with Celsius in red and Fahrenheit in blue:

```
C   F   C   F   C   F

10 50 11 51 12 53

13 55 14 57 15 59

16 60 17 62 18 64

19 66 20 68 21 69

22 71 23 73 24 75

25 77 26 78 27 80

28 82 29 84 30 86

31 87 32 89 33 91
```

The program written to solve this problem is:

```
 5 POKE 53281,1          } create heading
 10 FOR I=1 TO 3         }
 20 PRINT "   C   F";    }
 30 NEXT I
 40 PRINT:PRINT
 50 FOR I=10 TO 33 STEP 3 <---- start of each line
 60 FOR C=I TO 2
 70 F=INT((C*9)/5+32                                 } C = Celsius
 80 PRINT "[red]"STR$(C);"[blu]"STR$(F);            } F = Fahrenheit
 90 NEXT C
 100 PRINT
 110 NEXT I
```

When you run this program you have

```
C   F   C   F   C   F

10 50
13 55
16 60
19 66
22 71
25 77
28 82
31 87
```

Two problems are evident from this output. First, we are getting only one column of Celsius and Fahrenheit temperatures; second, there are no spaces between the lines of output as indicated in the expected results.

The second problem may be easier to solve at this point. The PRINT statement

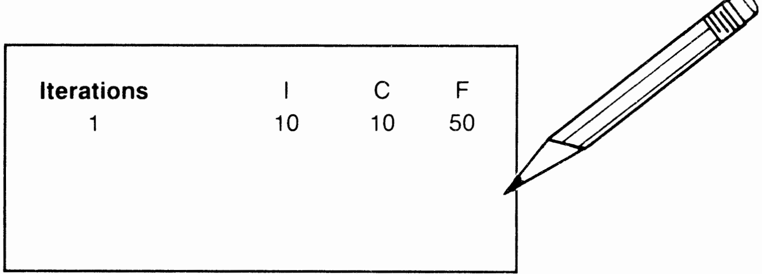
```
100 PRINT
```

starts each new line of output but does not produce a blank line. This can be achieved by using another PRINT statement on line 100 as follows:

```
100 PRINT:PRINT
```

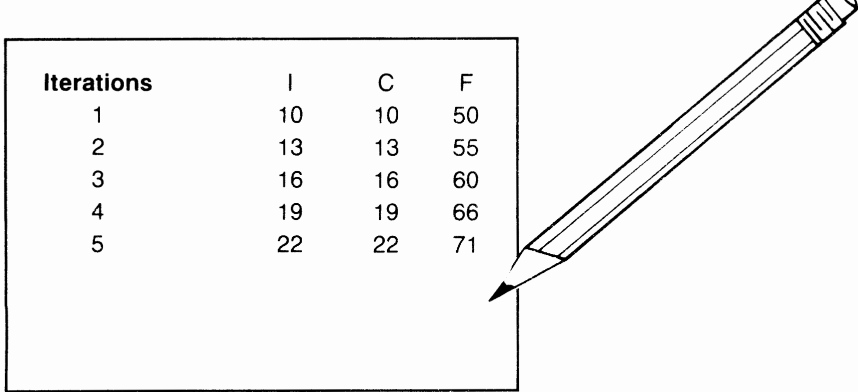
To locate the problem of missing output, we perform a manual trace of the program. To do this, we begin by writing down the names of the variables used in this part of the logic. Since the heading looks O.K., we do not need to trace this part of the program.

The variables that might cause the problem are I, C, and the calculation for Fahrenheit in statement 70. We will trace the FOR loop to the NEXT at 110. The variables are written as follows, with an additional column that counts the number of iterations through the logic. The first time through the program code, the manual trace looks like this:



Iterations	I	C	F
1	10	10	50

Each time that we follow the code visually, we write on the trace the effect that the code had on each variable. But we must be careful to do what the program says, not what we think it says. As we continue, the trace will look like this:



Iterations	I	C	F
1	10	10	50
2	13	13	55
3	16	16	60
4	19	19	66
5	22	22	71

At this point the program continues, but it is not absolutely essential to continue the trace to the bitter end — only long enough to determine what is causing the error in the logic. Reading this trace makes it evident that C is not taking on all the expected values. Since C represents Celsius it should take on each Celsius value. Instead, it takes on only the same values as I. If we check the statement that gives C its value, we might find the error. This is statement

```
60 FOR C=I TO 2
```

From this expression we can see that C starts at I and goes to 2. Well, if I is 10 then the loop is finished the first time, since 10 already exceeds the value 2. The FOR should start at I's value and go on for two

more values to finish the line. In other words, when I is 10, C should create temperatures, 10, 11, and 12. Thus the statement should be:

```
60 FOR C=I TO I+2
```

The program now looks like this with the changes:

```
 5 POKE 53281,1
10 FOR I=1 TO 3
20 PRINT "  C   F";
30 NEXT I
40 PRINT:PRINT
50 FOR I=10 TO 33 STEP 3
60 FOR C=I TO I+2  <------------ revised
70 F=INT((C*9)/5+32)
80 PRINT "[red]"STR$(C); "[blu]"STR$(F);
90 NEXT C
100 PRINT:PRINT  <------------ extra print
110 NEXT I
```

And the correct output is:

```
 C  F  C  F  C  F

10 50 11 51 12 53
13 55 14 57 15 59
16 60 17 62 18 64
19 66 20 68 21 69
22 71 23 73 24 75
25 77 26 78 27 80
28 82 29 84 30 86
31 87 32 89 33 91
```

Automatic Program Tracing

Many programs are either too large or too complex for manual tracing except in the most limited circumstances. In such a case, it is helpful if the program can perform its own tracing. This can be accomplished by placing PRINT statements at temporary locations within the program.

One use of this method is to determine whether the program reaches a specific location. This can be done by placing a PRINT statement, such as

```
PRINT "TAX ROUTINE"
```

at the beginning of the program code to be tested. If the literal TAX ROUTINE displays on the screen, we know that the program does indeed reach this point. It is important to use a message that will not be confused with normal program output. Sometimes reverse characters are useful for tracing. For example:

```
PRINT "[rvs]TAX ROUTINE[rvs off]"
```

When we are certain that the program is executing correctly or that the error has been found, the PRINT statement will be removed, since it has served its purpose.

Another use for this type of statement is to determine the contents of variables as the program executes. This is similar to manual tracing except the program displays it on the screen. A trace statement might be:

```
PRINT IN,N$
```

Each time the program reaches this location, the contents of IN and N$ are displayed. The programmer then examines these values to see if they are what is expected. If not, this gives some indication of the source of the program error.

It is important to realize that this temporary trace output is displayed with other output. With a little experience it is not too difficult to separate visually the normal output from the trace. If this becomes too much of a problem, some programmers temporarily remove the other PRINT statements so the trace is easier to follow.

Often, the above trace methods are combined. For instance:

```
PRINT "ORDER QUANTITY";AC;Q
```

displays both a message and the contents of the variables AC (account) and Q (quantity) when this part of the program is reached.

To demonstrate this technique of debugging, we will write a sales commission program that might be used by a person who is paid by commission for the sales made. In this program, the commission paid is based on a percentage of the dollar amount of sales. The percentages are as follows:

Sales Amount	Commission Percentage
less than $1000	5%
$1000 to $1999.99	7%
$2000 to $2999.99	10%
$3000 or more	12%

Figure 11.1 shows how the program and data might appear after the initial syntax errors have been corrected. Figure 11.2 shows the output that was printed on the first test run.

```
10 REM SALES COMMISSION WITH ERRORS
20 PRINT "[clr]"
30 PRINT " NAME    AMOUNT    COMM"
40 PRINT
50 READ N$,A
60 IF N$="LAST" THEN 210
70 IF A<1000 THEN 130
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
130 GOTO 170
140 C=A*.07
160 C=A*.10
170 PRINT N$;TAB(11);A;TAB(17);C
180 L=L+1
```

```
190 IF L<10 THEN 50
200 PRINT “CONT — PRESS ANY KEY”
210 GET W$:IF W$ = ”” THEN 210
220 GOTO 20
300 DATA “J ABEL”,500
310 DATA “S BELL”,1500
320 DATA “A CAVE”,2300
330 DATA “C CHART”,3100
340 DATA “Q CRUZ”,1000
350 DATA “D DE LANE”,2000
360 DATA “F FLIN”,3000
370 DATA “J FULTON”,5000
380 DATA “A GRAY”,2300
390 DATA “L HOBBS”,1990
400 DATA “W LAKES”,3300
410 DATA “N PARKER”,1999
420 DATA “R MOORE”,4000
430 DATA “F PRATT”,3900
440 DATA “LAST”,0
```

Figure 11.1 Sales commission program with logic errors

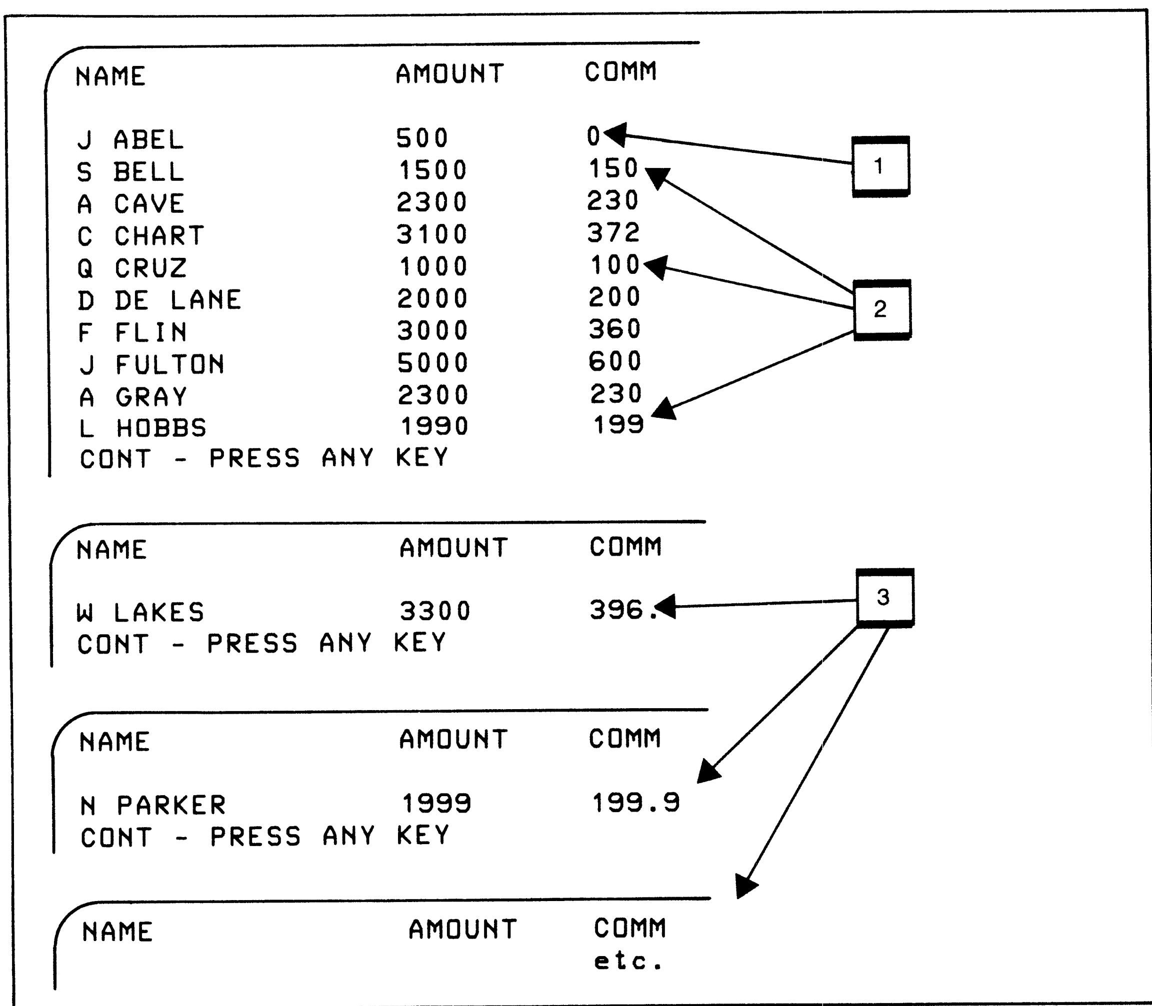


Figure 11.2 Sales commission program output

Examine the output and you'll note several errors. The first, labeled (1), shows that Abel has no commission calculated. This could be due to a forgotten calculation or an error relating to the first data record.

The second error (2) shows Cruz with a $100 commission when it should be $70. This error also occurs for the Bell and Hobbs data, which are computed at 10 percent when the rate should be 7 percent.

Last, (3) indicates that the screen is cleared and a heading displayed for each line after Hobbs. The intent was to display ten lines and then wait for the user to review the lines visually. When the user presses a key to continue, the next 10 lines should display. Instead only one line is shown.

An experienced programmer might be able to find these errors without a trace; however, we will use a trace to discover the source.

Since it seems likely the Cave commission is due to a calculation error, a trace will be placed after the calculation as follows:

```
120 C=A*.05
125 PRINT "LT 1000 COMM";C
```

Problem 2 suggests a difficulty with salespersons in the less-than-$2,000 category, so another trace is placed after the 7 percent calculation.

```
140 C=A*.07
145 PRINT "LT 2000 COMM";C
```

The last error is the problem with headings. Since L is used to count the number of lines, it should be printed each time we print and count a line. This is done with

```
180 L=L+1
185 PRINT "LINE COUNT=";L
```

All these trace statements are included for the next run of the program. The listing is shown in figure 11.3. To save space, the data statements are deleted from this printout. Figure 11.4 shows the output from this test run that we will now consider.

```
10 REM SALES COMMISSION WITH TRACES
20 PRINT "[clr]"
30 PRINT "NAME    AMOUNT    COMM"
40 PRINT
50 READ N$,A
60 IF N$="LAST" THEN 210
70 IF A<1000 THEN 130
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
125 PRINT "LT 1000 COMMISSION";C
130 GOTO 170
140 C=A*.07
145 PRINT "LT 2000 COMMISSION";C
160 C=A*.10
170 PRINT N$;TAB(11);A;TAB(17);C
180 L=L+1
185 PRINT " LINE COUNT=";L
190 IF L<10 THEN 50
200 PRINT "CONT — PRESS ANY KEY"
210 GET W$:IF W$="" THEN 210
220 GOTO 20
```

trace statements

Figure 11.3 Sales commission program with trace statements

When we examine the results of our trace on the screen (figure 11.4), we are led to the necessary corrections for the program. The first error was the commission zero for Abel. If we look at the trace output for Abel, it doesn't exist. This is a major clue for finding the error, which strongly suggests the program never reached statement 125, the trace statement. By the process of logical deduction, we conclude that statement 120 was also not done.

Now the question we should ask ourselves is: What code is responsible for bringing us to statement 120? It is the statement that determines Abel has sales of less than $1,000. This is statement 70. If you examine this statement, you will see that it goes to statement 130, not 120. This is the error. It is corrected as follows:

```
70 IF A<1000 THEN 120
```

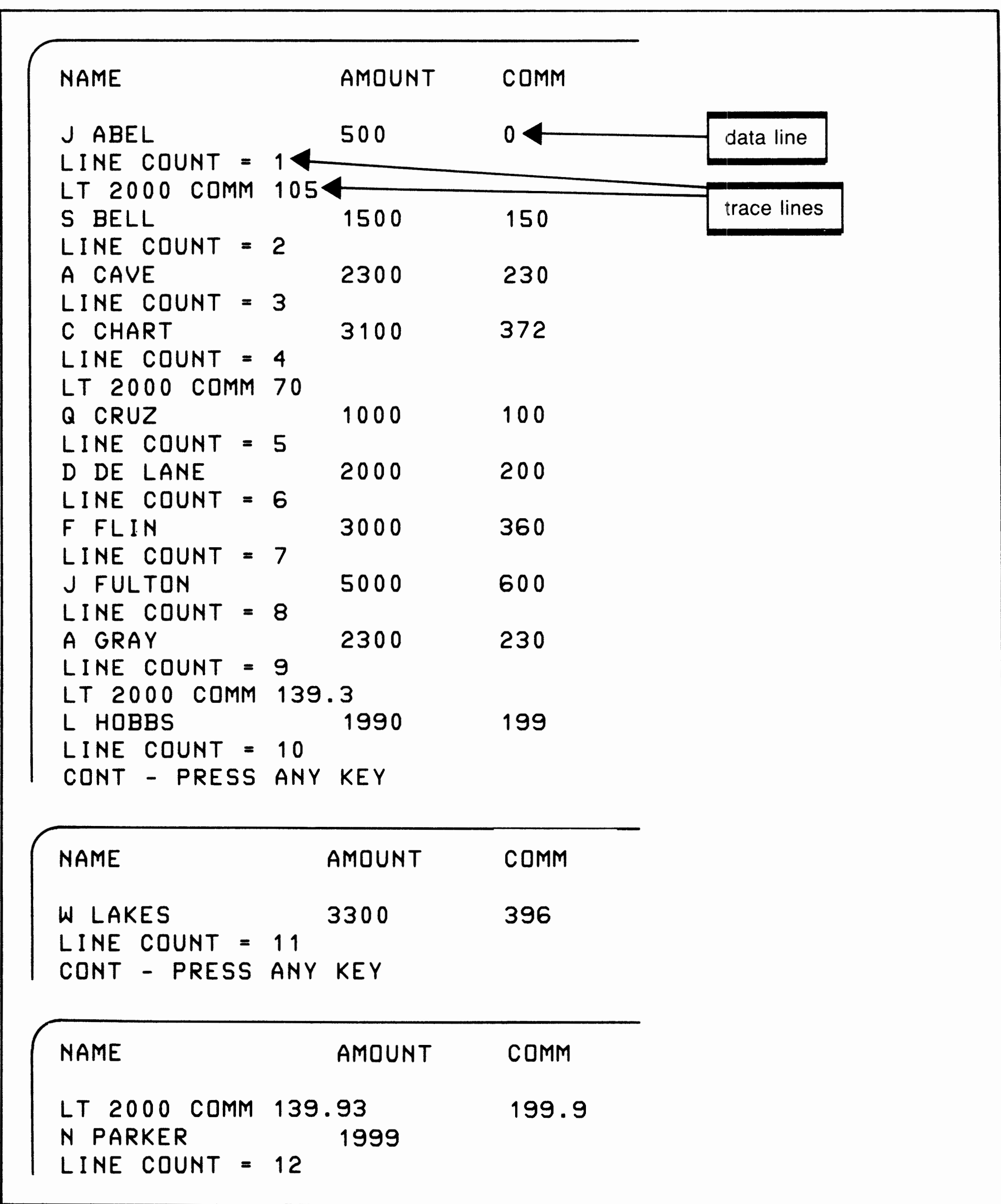


Figure 11.4 Program and trace output

The second error does produce a trace output. It shows that the calculation for Cruz's commission was 70 but that 100 was displayed. Thus the calculation was right but the output was wrong. If we look at this part of the program, we see the 7% is calculated in 140. The next statement is the trace followed by a calculation for 10%. Apparently we forgot the GOTO at 150.

```
150 GOTO 170
```

The last problem was the new screen display after ten lines. The trace shows a line count for each line of output beginning at 1 for the first line but proceeding to 10, then 11, and so on. Since we are allowing only ten lines per screen, we obviously forgot to reset the line counter to zero.

Now we remove the trace statements and include the corrections discussed above. At this stage a final test run is done as shown in figures 11.5 and 11.6. If there is still some concern at this stage of testing about whether all the errors have been found or if they have been properly corrected, the trace statements could be left in the program for one more run.

```
10 REM SALES COMMISSION
20 PRINT "[clr]"
30 PRINT " NAME    AMOUNT    COMM"
40 PRINT
50 READ N$,A
60 IF N$="LAST" THEN 210
70 IF A<1000 THEN 120
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
130 GOTO 170
140 C=A*.07
150 GOTO 170
160 C=A*.10
170 PRINT N$;TAB(11);A;TAB(17);C
180 L=L+1
190 IF L<10 THEN 50
200 PRINT "CONT — PRESS ANY KEY"
210 GET W$:IF W$="" THEN 210
215 L=0
220 GOTO 20
300 DATA "J ABEL",500
310 DATA "S BELL",1500
320 DATA "A CAVE",2300
330 DATA "C CHART",3100
340 DATA "Q CRUZ",1000
350 DATA "D DE LANE",2000
360 DATA "F FLIN",3000
370 DATA "J FULTON",5000
380 DATA "A GRAY",2300
390 DATA "L HOBBS",1990
400 DATA "W LAKES",3300
410 DATA "N PARKER",1999
420 DATA "R MOORE",4000
430 DATA "F PRATT",3900
440 DATA "LAST",0
```

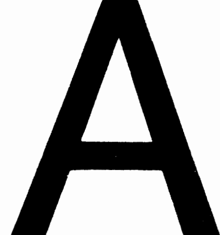
Figure 11.5 Sales commission program—debugged

```
NAME                 AMOUNT     COMM

J ABEL               500        25
S BELL               1500       105
A CAVE               2300       230
C CHART              3100       372
Q CRUZ               1000       70
D DE LANE            2000       200
F FLIN               3000       360
J FULTON             5000       600
A GRAY               2300       230
L HOBBS              1990       139.3
CONT - PRESS ANY KEY

NAME                 AMOUNT     COMM

W LAKES              3300       396
N PARKER             1999       139.93
R MOORE              4000       480
F PRATT              3900       468
```

Figure 11.6 Sales commission program—debugged

REVIEW QUESTIONS—CHAPTER 11

1. What is meant by desk checking and when would it be used?
2. Describe what is meant by syntax errors. What is the usual reason for a syntax error?
3. See how many different kinds of syntax error messages you can create. What is the correction in each case?
4. Give eight considerations for preparing test data.
5. Why is volume of test data not an appropriate concern?
6. Explain how immediate mode operations may be used for debugging a program.
7. What are the two kinds of program tracing? When is it best to use each one?
8. What is a trace statement? How would one be used for debugging a program?
9. How do you know when a program is completely debugged? Can you ever be absolutely certain?



Appendix A
BASIC
Operating Commands

This appendix lists some of the operating commands available in BASIC on the C-64. The majority of these have been discussed throughout the book as they were needed. Additional information can be found by using the index.

CLR

The CLR command sets all numeric variables in a program to zero, all string variables to null (empty string), and frees all array space, and rests memory and stack space.

Example:

```
10 CLR
```

If used at the beginning of the program, all variables are cleared prior to program execution.

LIST [line number] — [line number]

This command lists all or part of a BASIC program. When no line numbers are specified, the entire program is listed. If a single line number is included, only that line of the program is displayed. When two line numbers are included, all statements in that range are listed on the screen.

Example:

LIST	Lists the entire program
LIST 150	Displays line number 150
LIST 100-200	Lists all statements from 100 to 200
LIST-185	Lists all statements up to 185
LIST 560-	Lists all statements from 560 and up

LOAD "filename"[,device]

Normally the LOAD is used to load a program from tape, tape being the default when no device number is specified. Using a device number of 8 will cause the program named to be loaded from disk.

Example:

```
LOAD "CHASE"
```

loads the program CHASE from tape into memory.

```
LOAD "COLLECTION",8
```

loads the program called COLLECTION from the disk drive.

NEW

NEW erases the current program from memory. This command should be used when you begin writing a new program.

Example:

```
NEW
```

RUN [line number]

Used to executive the program that is currently in memory. Including the line number will begin execution at that line in the program.

Example:

```
RUN
RUN 100
```

SAVE "filename"[,device number]

Normally used to save a BASIC program on tape. By including a device number, programs may be saved on disk.

Example:

```
SAVE "PHONE"
```

saves the current BASIC program "PHONE" on tape.

```
SAVE "PHONE",8
```

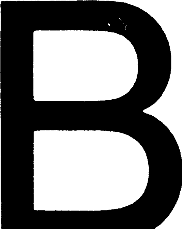
saves the current BASIC program "PHONE" on disk.

VERIFY "filename" [,device number]

Used to verify the correctness of the contents of a program on tape. Normally VERIFY is used after the SAVE command to ensure the program was recorded accurately. After a program has been saved, the tape is rewound and the VERIFY command is entered.

Example:

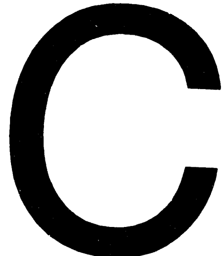
```
VERIFY "PHONE"
```



Appendix B
Reserved Words

Certain words in BASIC are intended to be used only for the purpose for which they were intended and not as variable names. Two-character words can present particular difficulties if they are used as variable names in a program.

ABS	GET	OPEN	SPC
AND	GET#	OR	SQR
ASC	GOSUB	PEEK	ST
ATN	GOTO	POKE	STEP
CHR$	IF	POS	STOP
CLOSE	INPUT	PRINT	STR$
CLR	INT	PRINT#	SYS
CMD	LEFT$	READ	TAB
CONT	LEN	READ#	TAN
COS	LET	REM	THEN
DATA	LIST	RESTORE	TI
DEF	LOAD	RETURN	TI$
DIM	LOG	RIGHT$	TO
END	MID$	RND	USER
EXP	NEW	RUN	VAL
FNP	NEXT	SAVE	VERIFY
FOR	NOT	SGN	WAIT
FRE	ON	SIN	

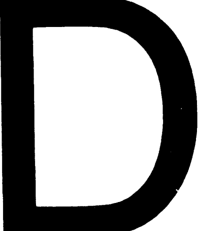


Appendix C
Abbreviations

Early in the book we used the question mark (?) to represent the command PRINT. The BASIC interpreter translated this abbreviation into the equivalent PRINT statement. Most reserved words in BASIC can be represented by a two- or three-letter abbreviation. When the abbreviation is two letters, usually it will be the normal first letter of the reserved word followed by the second letter shifted. For example, the word LIST can be entered by typing L [shift] I. The shifted character "I" will display as a graphic on the C-64.

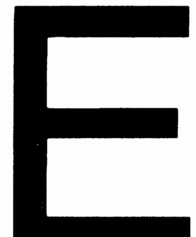
In the following chart, lowercase letters are typed normally while capitals represent a shifted character.

WORD	ABBREV	WORD	ABBREV	WORD	ABBREV	WORD	ABBREV
ABS	aB	GET	gE	OPEN	oP	SPC	sP
AND	aN	GET#	get#	OR	or	SQR	sQ
ASC	aS	GOSUB	goS	PEEK	pE	ST	st
ATN	aT	GOTO	gO	POKE	pO	STEP	stE
CHR$	cH	IF	if	POS	pos	STOP	sT
CLOSE	clO	INPUT	input	PRINT	?	STR$	str$
CLR	cL	INT	int	PRINT#	pR	SYS	sY
CMD	cM	LEFT$	leF	READ	rE	TAB	tA
CONT	cO	LEN	len	READ#	read#	TAN	tan
COS	cos	LET	lE	REM	rem	THEN	tH
DATA	dA	LIST	lI	RESTORE	reS	TI	ti
DEF	dE	LOAD	lO	RETURN	reT	TI$	ti$
DIM	dI	LOG	log	RIGHT$	rI	TO	to
END	eN	MID$	mI	RND	rN	USER	uS
EXP	eX	NEW	new	RUN	rU	VAL	vA
FN	fn	NEXT	nE	SAVE	sA	VERIFY	vE
FOR	fO	NOT	nO	SGN	sG	WAIT	wA
FRE	fR	ON	on	SIN	sI		



Appendix D
Disk
Error Messages

Type	Error Number	Error Message	Track	Sector
Status	00	OK	00	00
	01	FILES SCRATCHED	# Files	00
Read Errors	20	READ ERROR (Block header not found)	T	S
	21	READ ERROR (No sync character)	T	S
	22	READ ERROR (Data block not present)	T	S
	23	READ ERROR (Checksum error in data block)	T	S
	24	READ ERROR (Byte decoding error)	T	S
	27	READ ERROR (Checksum error in header)	T	S
Write Errors	25	WRITE ERROR (Write-verify error)	T	S
	26	WRITE PROTECT ON	T	S
	28	WRITE ERROR (Long data block)	T	S
	29	DISK ID MISMATCH	T	S
Syntax Errors	30	SYNTAX ERROR (General syntax)	00	00
	31	SYNTAX ERROR (Invalid command)	00	00
	32	SYNTAX ERROR (Long line)	00	00
	33	SYNTAX ERROR (Invalid filename)	00	00
	34	SYNTAX ERROR (No file given)	00	00
	39	SYNTAX ERROR (Invalid DOS command)	00	00
	50	SYNTAX ERROR (Record not present)	00	00
	51	SYNTAX ERROR (Overflow in record)	T	S
	52	SYNTAX ERROR (File too large)	T	S
File Errors	60	WRITE FILE OPEN	00	00
	61	FILE NOT OPEN	00	00
	62	FILE NOT FOUND	00	00
	63	FILE EXISTS	00	00
	64	FILE TYPE MISMATCH	00	00
	65	NO BLOCK	T	S
	66	ILLEGAL TRACK AND SECTOR	T	S
	67	ILLEGAL SYSTEM TRACK AND SECTOR	T	S
System Errors	70	NO CHANNEL	00	00
	71	DIR ERROR	00	00
	72	DISK FULL	00	00
	73	DOS MISMATCH	00	00
	74	DRIVE NOT READY	00	00

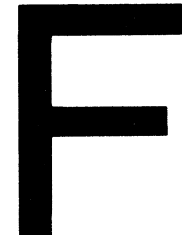


Appendix E
C-64 ASCII and PEEK/POKE Codes

The C-64 uses internal codes to represent all characters available to the programmer. One system of coding is ASCII (American Standard Code for Information Interchange), although Commodore uses a variation of this standard. The ASCII codes are given by the ASC function and used by the CHR$ function.

These codes are useful for producing screen graphics, in which case the PEEK or POKE value is used from the right column of the chart.

ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK
163	99	165	101	184	120	183	119	177	113	214	86
197	69	212	84	185	121	175	111	178	114	219	91
196	68	199	71	181	117	180	116	179	115	206	78
195	67	194	66	182	118	170	106	171	107	205	77
192	64	221	93								
198	70	200	72							166	102
210	82	217	89	161	97	169	105	209	81	220	92
164	100	167	103	162	98	223	95	215	87	168	104
193	65	213	85	207	79	188	124	176	112		
218	90	202	74	204	76	190	126	174	110		
211	83	201	73	208	80	172	108	173	109		
216	88	203	75	186	122	187	123	189	125		
						191	127				



Appendix F
ASCII
and CHR$ Codes

This appendix shows you what characters will appear if you PRINT CHR$ (X), for all possible values of X. It will also show the values obtained by typing PRINT ASC ("x") where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper/lower case, and printing character-based commands (like switch to upper/lower case) that could not be enclosed in quotes.

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0		16	SPACE	32	Ø	48	@	64	P	80
	1	CRSR ↓	17	!	33	1	49	A	65	Q	81
	2	RVS ON	18	"	34	2	50	B	66	R	82
	3	CLR HOME	19	#	35	3	51	C	67	S	83
	4	INST DEL	20	$	36	4	52	D	68	T	84
WHT	5		21	%	37	5	53	E	69	U	85
	6		22	&	38	6	54	F	70	V	86
	7		23	'	39	7	55	G	71	W	87
	8		24	(	40	8	56	H	72	X	88
	9		25	)	41	9	57	I	73	Y	89
	10		26	*	42	:	58	J	74	Z	90
	11		27	+	43	;	59	K	75	[	91
	12	RED	28	,	44	<	60	I	76	£	92
RETURN	13	CRSR	29	—	45	=	61	M	77	]	93
SWITCH TO LOWER CASE	14	GRN	30	.	46	>	62	N	78	↑	94
	15	BLU	31	/	47	?	63	O	79	←	95

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	96	●	113		130	CLR HOME	147		164		181
♠	97		114		131	INST DEL	148		165		182
	98	♥	115		132		149		166		183
	99		111	f1	133		150		167		184
	100		117	f3	134		151		168		185
	101		118	f5	135		152		169		186
	102	○	119	f7	136		153		170		187
	103	♣	120	f2	137		154		171		188
	104		121	f4	138		155		172		189
	105	♦	122	f6	139	PUR	156		173		190
	106		123	f8	140	CRSR	157		174		191
	107		124	SHIFT RETURN	141	YEL	158		175		
	108		125	SWITCH TO UPPER CASE	142	CYN	159		176		
	109	π	126		143	SPACE	160		177		
	110		127	BLK	144		161		178		
	111		128	CRSR	145		162		179		
	112		129	RVS OFF	146		163		180		

Index

Abbreviation 199
ABS 90
Add 23
AND 44
Animation 128-132
Animating Multiple Objects 131
Arithmetic functions 90
Arithmetic statements 23
Arrays 39-42, 70
ASC 93
ASCII 83, 85, 93, 126, 133, 201, 202
Assignments 23
Attack/Decay 140
ATN 90

Backup 11
Bar Charts 60
BASIC 2, 13
Bit 17
Bit pattern 133
Budget program 149, 151, 155, 168
Byte 6, 17

CAI chess 105
Calculator mode 13, 18
Cash flow program 156
Cassette (see tape)
Checkbook program 172
Chessboard graphic 127
CHR$ 94, 151, 202
CLEAR 15
Clear screen 26, 32, 33, 63, 151
Clock (see TI and TI$)
CLOSE 148, 166
CLR 195
Color 6, 22, 33, 60, 84, 126, 139
Color addresses 83-84
Comma 26
Command language 119
Commodore Key 6, 129
Computer Assisted Instruction 62, 105
Concatenation 97
COS 90
CTRL Key 6, 22
Cursor controls 6, 14, 63-64, 128, 129

DATA 59, 105
Debugging 181, 184
Decimal positions, controlling 89, 102
DEF FN 88
Degrees to radians 93
DELETE 15
Defaults 115
Directories 10
DIM 39
Disk 3, 9-11
Disk errors 171, 200
Disk files 165
Divide 24
DOS 3, 165, 200
Driver routines 104

End of file 148
English code 29, 34, 36, 48, 52, 54, 173
Egg timer 130
EXP 91
Exponentiation 24

Fahrenheit-Celsius program 30-32, 186
Filename 148, 167, 172
Files 147, 165
File updating 154
Floating point 19
Flowcharts 67-69, 75, 100, 104, 154, 159
FOR—NEXT 44
Form filling 119
Formatting 10
Frequency 141
Functions 90

GET 79, 117, 202
GOSUB 45
GOTO 28, 44
Graphics 6, 33, 121-124
Graphs 60, 98

Hardware 1
Hierarchy 24
High resolution 132
HOME 7, 15

IF 42
Immediate mode 13, 18, 184
INPUT 27, 55, 62, 114
Input 1, 28
INPUT# 150, 168
INSERT 15
Instring search 108
INT function 32, 47, 90, 91
Integers 17
Integer names 21

K 2
Keyboard 5
Keyboard graphic 121-124

LIST 16, 195
LEFT$ 94
LEN 95
LOAD 8, 9, 196
Load error 8
Loan payments program 36
LOG 91
Logical operators 42
Lower case 83
Lunar lander graphic 126

Memory 2, 17
Menus 116-118, 156
Metric conversion program 66-73
MID$ 95
Multiple fields 152, 169
Multiple statements 22, 43
Multiply 24

NEW 196
Note duration 143
Numbers 16
Number guessing game 48-51

ON GOSUB 80, 117
ON GOTO 81, 116
OPEN 148, 166
OR 44
Output 1

Parentheses 24
Payroll program 74
PEEK 85, 201
Plotting graphs 101
POKE 22, 83-85, 122, 126, 130, 137, 201
PRINT 13, 25, 30, 63, 114
PRINT# 149, 168
Print zones 25
Program 1, 8
Program generalization 105
Prompting 55, 62, 114
Pseudo code (see English code)

Radians to degrees 93
RAM 2
Random numbers 47, 86, 92
Random responses 107
Reaction timer 86, 124
READ 59
Real numbers 17
REM 29
Replacing a file 172
Replacing a program 10
Reserved words 198
RESTORE 62
Restore Key 6, 16
RETURN 6, 45
Return Key 6, 27
Reverse (RVS) 6
RIGHT$ 97
RND function 47, 91
Rocket animation 128-129
ROM 2
Rounding 17
RUN 8, 31, 196
RUN/STOP 6, 15
RVS ON/OFF 6

Sailboat graphic 33
Sales commission program 189-194
SAVE 8-9, 10, 197
Scientific Notation 19
Screen addresses 83, 84
Screen and border colors 23
Screen characters 94, 122
Screen page 174
Scrolling 10
Semicolon 26, 30, 45
Sequential files 147, 166
SGN 92
Simple calculation program 29
SIN 92
Sound 139
SPC 65
Sprite Graphics 135-139
SQR 93
ST 151
STOP 16, 47
STR$ 97
Strings 20, 64
String functions 93
String names 21
Software 1
Subroutines 46
Subtract 24
Sustain/Release 141
Syntax errors 182

TAB 65
TAN 93
Tape 7-9, 147
Test data 184
Time delays 50
TI function 47, 86
TI$ function 87
Tracing 181, 185, 188
Train program 131
Trip costs program 53-56
Truncating 17

Updating (see file updating)
Upper/lower case 83

VAL 97
Variables 20
Variable names 21
VERIFY 197

Waveform 142
Weighted average program 33, 52, 61

Zones 25

wcb
Wm. C. Brown Publishers
Dubuque, Iowa

$16.95 ISBN 0-697-09912-1