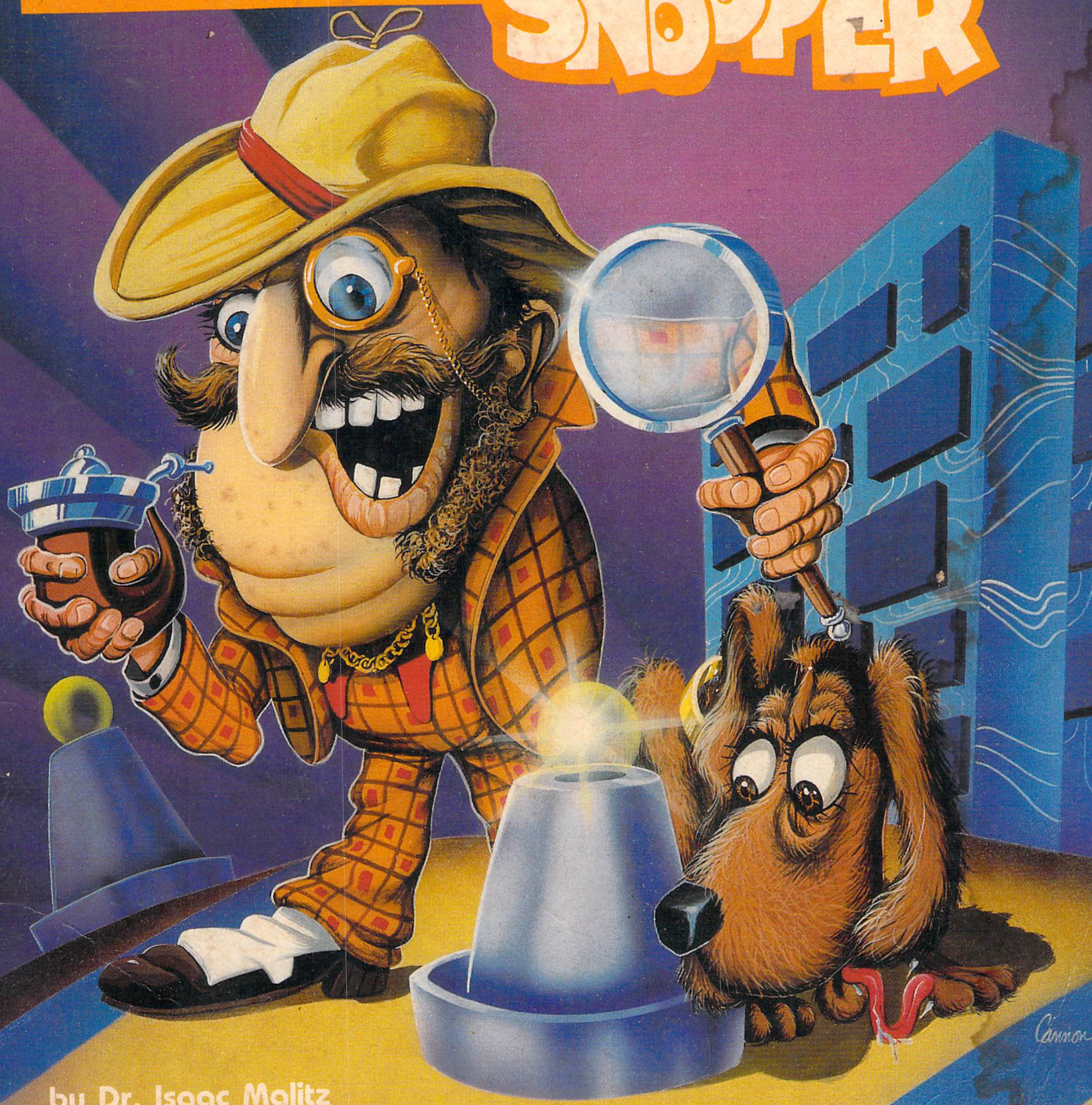
$14.95

THE SUPER COMPUTER SNOOPER



by Dr. Isaac Malitz

FIND OUT WHAT GOES ON INSIDE THE

COMMODORE 64!

The

Super Computer Snooper

Commodore 64

The

Super Computer Snooper

Commodore 64

by

Dr. Isaac Malitz

Illustrated by

Robert Peters

20660 Nordhoff Street
Chatsworth, CA 91311-6152
(818) 709-1202

ISBN 0-88190-356-6

Copyright © 1984 DATAMOST, Inc.
All Rights Reserved

This manual is published and copyrighted by DATAMOST, Inc. Copying, duplicating, selling or otherwise distributing this product is hereby expressly forbidden except by prior written consent of DATAMOST, Inc.

The word Commodore and the Commodore logo are registered trademarks of Commodore Business Machines.

Commodore Business Machines was not in any way involved in the writing or other preparation of this manual, nor were the facts presented here reviewed for accuracy by that company. Use of the term Commodore should not be construed to represent an endorsement, official or otherwise, by Commodore Business Machines.

Printed in U.S.A.

Acknowledgments

I want to express my deep appreciation to the many wonderful people who helped to make this book possible:

For telling me to write this book: Hal Glicksman.

For providing the illustrations and layout, which have so greatly increased the clarity of this book: Bob Peters, Scott Ross, Darlene Nunez.

For turning my manuscript into a real book: Marcia Carrozzo, Editor-in-Chief of DATAMOST, Inc.

For "making it all happen": Dave Gordon, President of DATAMOST, Inc.

Finally, I want to thank my wife and daughter for love, forbearance, and high inspiration.

TABLE OF CONTENTS

I. INTRODUCTION 9
Helpful Hints 11

II. MEMORY 13
How PEEKDEMO Works 16

III. HOW TO ALTER CODES IN MEMORY 19
How POKEDEMO Works 24
How SHOWALL Works 25
How HOTBAN Works 27

IV. PEEKING AND POKING AROUND MEMORY 31
Special Keys 31
How BARCODES Works 42

V. THE STORY OF Z 45
How LIGHTNING Works 49

VI. A PROGRAM TO SCAN MEMORY 53
How SNOOP Works 55

VII. THE MAP OF MEMORY 59

VIII. PROGRAMS 69
How the Program Works 74
How INVIS Works 75

IX. POINTERS 81

X. VARIABLES 89
Kinds of Variables 89
How VARLOOK1 Works 93
How VARLOOK3 Works 99
Summary 100

XI. DISK STORAGE — PART 1 103
How the Program Works 107
How BASEBALL2 Works 109

XII. DISK STORAGE — PART 2 111
How MAPS Works 113

XIII. SOUNDS 115
How SOUNDLAB Works 120

XIV. MACHINE LANGUAGE 125

XV. HARDWARE 135

XVI. CONCLUSION 141

APPENDICES

A. EXTRA TOPICS 145

B. NOTES ON BASIC FOR COMPUTER SNOOPERS 165

C. CHARTS 177

D. PROGRAM LISTINGS 187

GLOSSARY 201

INDEX 205

Chapter I

INTRODUCTION

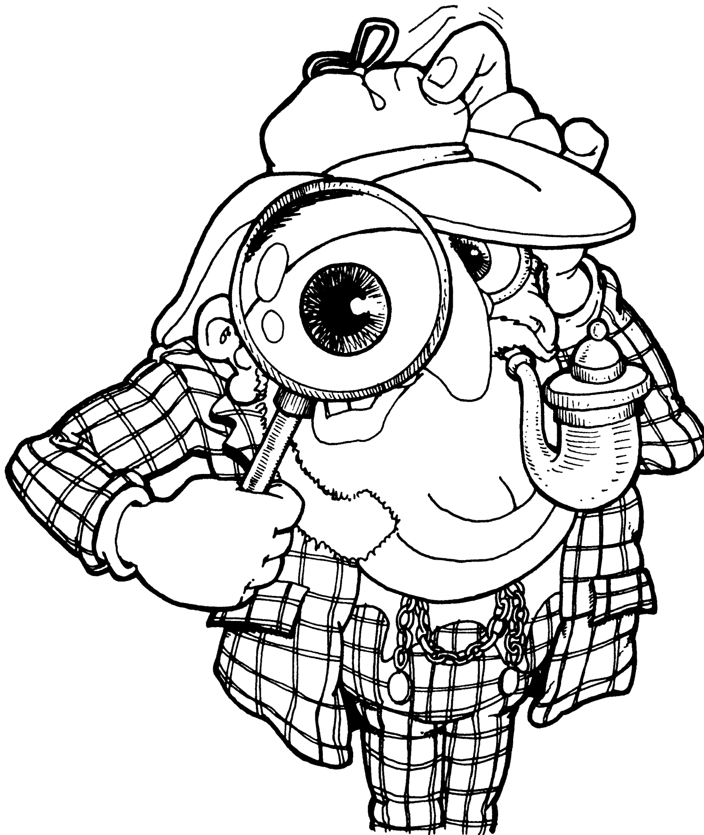
What goes on inside the COMMODORE 64? How does it work?

This book will help you to find out.

You will need a COMMODORE 64 computer and you should know how to write simple programs in BASIC. However, no other special knowledge is needed.

We are going to look at each of the main parts of the COMMODORE 64, and find out how they work together. You will learn about memory, screen, programs and variables, keyboard, disk, graphics, and sound.

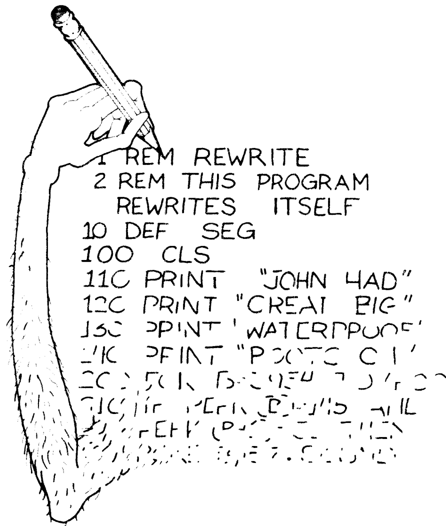
You will perform some interesting experiments:



You will find out how to restore a program that has been accidentally erased.

You will find out how to "listen" to the inner workings of the COMMODORE 64, using an ordinary radio.

You will write a program which re-writes itself.



You will find out how to make the computer display everthing upside-down.

You will create some amazing effects with sound and with graphics.

When you have finished this book,

You will understand better how a computer works — especially the COMMODORE 64.

You will have learned some powerful techniques that you can use when you write your own programs.

You will be prepared to study advanced topics, such as machine language programming or arcade graphics, should you ever want to do this.

The COMMODORE 64 is a mind-boggling machine. Each second, millions of electronic signals pass through it. These signals travel at speeds of nearly one billion feet per second. They criss-cross and interact with dazzling complexity. They are organized to perform tasks that sometimes are almost completely beyond the capability of human beings.

How does the COMMODORE 64 work? What goes on in there?

Let's find out . . .

HELPFUL HINTS

The following tips will help you to enjoy your exploration of the COMMODORE 64

1. We will suggest lots of experiments in this book. We encourage you to try them and to even make up your own experiments. You will have fun, and you will learn a lot about your COMMODORE 64. Don't worry, you won't break anything

2. Some of our experiments will involve the computer's memory. To clear memory afterwards, **turn the computer off, wait ten seconds, and then turn it back on.** Do not use a "warm boot" to clear memory, since a warm boot does not necessarily clear all areas of memory.

Whenever you have finished a session with this book and you want to do something else at the computer, **always clear memory,** as described above. Otherwise your experiments may leave stray information in memory that could interfere with the processing of other programs.

3. If you have a disk drive or cassette recorder, some of our experiments may alter data on disk or tape. Make up a special diskette or tape, to be used only with this book.

4. Nothing in this book will require you to open your computer. We recommend that you not open your computer, except with guidance from an expert.

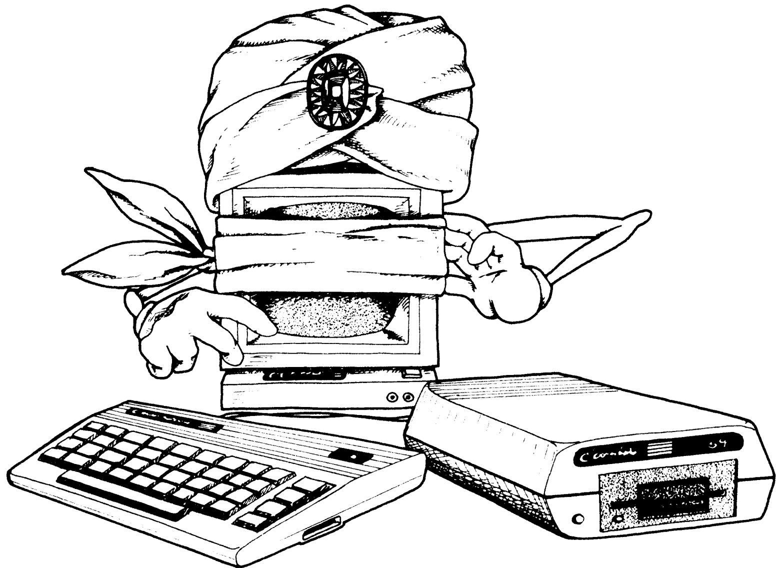
Chapter II

MEMORY

The fastest way to start learning about the COMMODORE 64 is to look inside its *memory.*

Whatever is happening inside the COMMODORE 64, the memory usually knows about it.

> When you are running a program, the program and its variables are in the C64's memory.
>
> The memory knows at all times what line of the program is currently being executed.
>
> The memory knows what is being displayed on the screen.
>
> When the computer is sending or receiving signals from a disk drive, screen, printer, keyboard, joystick, or anything else attached to the computer, memory usually knows about it.



Let's define what is meant by "memory":

Memory is the part of your computer that holds information the computer is currently using. When you are running a program, the program and all of its variables are held in memory. Memory is also used for many other purposes, as we shall see.

Memory is divided into a large number of "storage locations." Each location is like a little box that can store a small amount of information. The locations are numbered 0, 1, 2, 3, and so on. Each location can hold one "byte" of information. A byte is about the same as one character of information. A byte is stored as a value ranging from 0-255. If you look into any location in memory, you will find a value from 0-255.

A COMMODORE 64 normally comes with "64K" of memory. "64K" means 64 thousand bytes, or 64 thousand storage locations. Additionally, the C64 has some special-purpose memory for its internal use, which normally you cannot access. We will find out more about this later. Altogether, a COMMODORE 64 has about 84K of memory of all types.

If you could look into the COMMODORE 64's memory, you could see almost everything that is happening in the computer. You can look into the memory by using the PEEK command. Let's find out how to do this. We are going to write a program in BASIC called PEEKDEMO. This program will show you how to use the PEEK command to look into the computer's memory.

Here is what PEEKDEMO does:

First it clears the screen.

Then it displays the following message in the upper left-hand corner of the screen:

```
BAD-CAT !!
```

Then it displays part of the memory that tracks what is on the screen.

ENTER THIS PROGRAM

```
1 REM PEEKDEMO
2 REM DEMO OF PEEK COMMAND
100 PRINT CHR$(147);
110 PRINT "BAD-CAT !!"
200 PRINT:PRINT
210 FOR I = 1024 TO 1033
220 P = PEEK(I)
230 PRINT P;
240 NEXT I
```

If you have a disk drive or cassette recorder, save this program under the name PEEKDEMO. We will be writing many programs in this book and we suggest that you save them. In that way you will build up a useful collection of programs for computer-snooping. Save all programs under the name listed in the first line of the program.

Now run the program. Here is what you will see on your screen:

```
BAD-CAT !!

2 1 4 45 3 1 20 32 33 33
```

Those codes on the second line are memory's way of describing the upper left-hand corner of the screen. Each of the codes has a meaning. Here is what they mean:

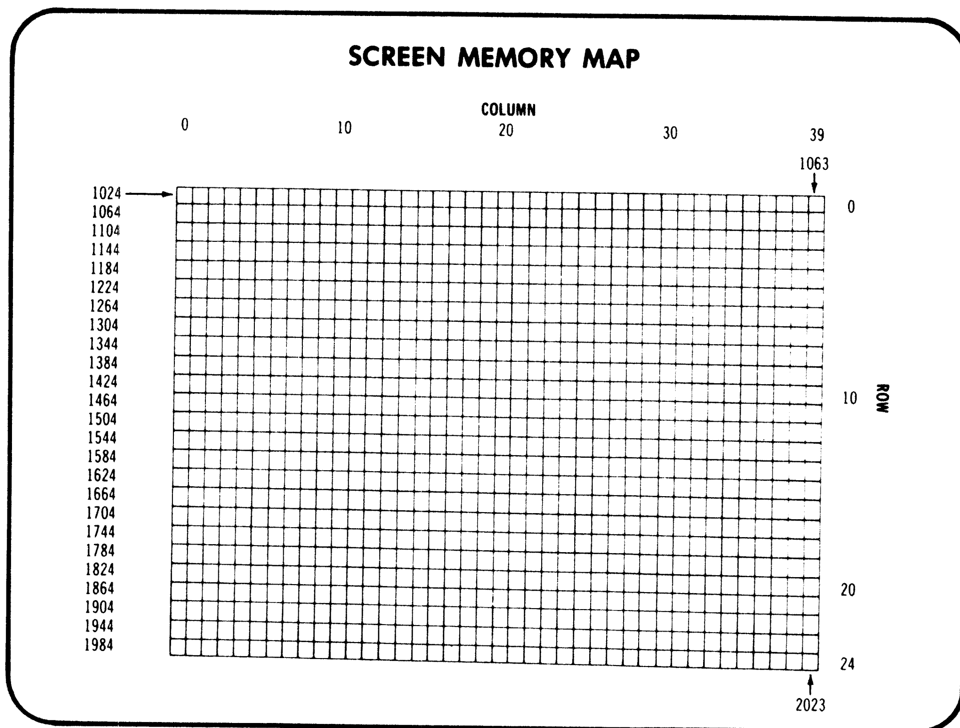
```
2  1  4  45  3  1  20  32  33  33
B  A  D   -  C  A   T       !   !
```

Each of the codes 2, 1, 4, and so on stand for a character that is displayed on the screen. Each code occupies one byte of memory. The value of each code can range from 0 to 255.

These codes are stored in a section of memory whose purpose is to track what characters are on the screen. This section of memory starts at location 1024 and continues for 1000 bytes to location 1123. This section of memory is known as "screen memory." Screen memory consists of 1000 bytes of information. This corresponds exactly to the 1000 positions on your screen (25 rows of 40 characters each).

Wherever there is a character on your screen, there is a corresponding location in memory which holds a code which stands for that character. For instance, when the letter B is in the upper left-hand corner, the code 2 is in memory location 1024. As we shall see shortly, if you alter any of the codes for the screen, the appearance of the screen will change.

Let's go back and look at your program PEEKDEMO, and find out how it works.



How PEEKDEMO Works:

Lines 100 - 110

```
100 PRINT CHR$(147);
110 PRINT "BAD-CAT !!"
```

clear the screen and display BAD-CAT !! in the upper left-hand corner. The command in line 100

```
PRINT CHR$(147);
```

is a programming trick for clearing the screen and positioning the cursor in the upper left-hand corner. This is equivalent to pressing SHIFT-CLR from the keyboard.

The FOR - NEXT loop in lines 200 - 230

```
200 FOR I = 1024 TO 1033
220 P = PEEK(I)
230 PRINT P;
240 NEXT I
```

processes the first 10 bytes starting with byte 1024. Line 220

```
220 P = PEEK(I)
```

finds out the code in one of the bytes of memory. For instance, if I is 1024, then the PEEK command finds out what code is in byte 1024 and places the value in the variable P.

Line 230

```
230 PRINT P;
```

displays the variable P on the screen. You can use the PEEK command to look at any part of memory. The positions in memory are numbered from 0 to 65535. To look at any location in memory, simply designate its location with the PEEK command. For instance, to look at location 11235, use

```
PEEK(11235)
```

To help you find your way around memory, a "memory map" and other useful information will be provided later. We also will write a more powerful version of PEEKDEMO that will help you to look through memory. Some parts of memory involve difficult codes and are complex to analyze, but you will be pleased to realize how much of memory you can understand by the end of this book.

Chapter III

HOW TO ALTER CODES IN MEMORY

In the last chapter, we saw how to use PEEK to look around in memory. We saw that memory is filled with codes. If those codes are altered, amazing things can happen! Let's find out how to alter codes in memory.

To do this, we use the POKE command

RUN THIS PROGRAM:

```
1 REM POKEDEMO1
2 REM DEMO OF POKE COMMAND
100 PRINT CHR$(147);
110 PRINT "BAD-CAT !!"
200 REM LINE 210 CAUSES A 5 SECOND PAUSE
210 FOR X = 1 TO 3500:NEXT X
220 POKE 1024,26
```

Here is what happens when you run POKEDEMO:

The screen is cleared.

In the top left-hand corner you will see

BAD-CAT !!

After about five seconds, the first letter on the screen will change to a Z. The screen will then show

ZAD-CAT !!

How was this done ?

Line 220

```
220 POKE 1024, 26
```

tells the computer to store code 26 in memory location 1024. As we saw in the last section, this memory location indicates what character is in the first position on the screen (i.e., the upper left-hand corner). Code 26 stands for Z. So the character displayed on the screen is a Z.

Line 210

```
210 FOR X = 1 TO 3500:NEXT X
```

is used to cause a five second delay. It is a FOR - NEXT loop which does nothing but count from 1 to 3500. This takes about five seconds.

Let's try some more pokes. To make this easy, we will revise POKEDEMO1 so that you can enter variable poke information.

ENTER THIS PROGRAM:

```
1 REM POKEDEMO2
100 SP$ = "                        "
110 PRINT CHR$(147);
120 PRINT "BAD-CAT !!"
200 PRINT CHR$(19);TAB(200);
210 PRINT SP$:PRINT SP$
220 PRINT CHR$(19);TAB(200);
230 INPUT "LOCATION";L
240 INPUT "CODE";C
250 POKE L,C:GOTO 200
```

Run the program.

The screen will clear. At the top of the screen you will see

```
BAD-CAT !!
```

In the middle of the screen you will see

```
LOCATION?
```

Type 1024 and press RETURN. This tells the computer that we want to POKE something into memory location 1024.

Now a second question will appear:

```
CODE?
```

Type 18 and press RETURN. This tells the computer that you want to POKE the code 18 into the memory location.

Code 18 stands for R, and your screen display will change to

RAD-CAT !!

The computer will now ask you for another LOCATION IN MEMORY and VALUE TO POKE. Here are some values to try:

LOCATION IN MEMORY	CODE TO POKE	EFFECT
1026	20	RAT-CAT !!
1025	0	R@T-CAT !!
1029		CAT gets a heart
	83	
1028	131	Reverse video

A complete chart of screen codes and symbols produced is found in Appendix D.

So far, all of our screen displays have been in black and white. The COMMODORE 64 is also able to display characters in color — 16 colors in all. Each position on the screen can be assigned any one of these 16 colors. How can this be done?

Well, there is a section of memory, called "color memory," which tracks the color of each of the 1000 positions on the screen. Here's how it works:

Color memory starts at location 55296 and continues for 1000 bytes to location 56295. Each of the locations in color memory corresponds to one of the positions on the screen.

When you POKE a value of 0 - 15 into a location in screen memory, you will assign a color to a certain position on the screen. The list of colors is as follows:

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

Let's POKE some values into color memory and see what happens. To do this, we can use POKEDEMO again. Stop POKEDEMO by pressing STOP-RESTORE.

Now RUN the program again. Once again, at the top of your screen, you will see

BAD-CAT !!

Now try these POKEs:

LOCATION IN MEMORY	CODE TO POKE	EFFECT
55296	8	B turns orange
55297	5	A turns green
55298	2	D turns red

Locations 55296, 55297, and 55298 correspond to the first three positions on the screen. The color codes which you POKEd into these locations assigned colors to the first three positions on the screen.

By POKEing the right values into screen memory and color memory, you can put characters anywhere on the screen, and make them any color you want. There is one "catch" however:

> If you POKE a character into a new position on the screen, you will not actually see the character until you also assign it a color in color memory.

In other words, if you want to display a character on the screen in a new position, you must select both a character and a color for that position. Let's try an example to illustrate this rule.

Use the program to POKE these two values:

LOCATION IN MEMORY	CODE TO POKE	EFFECT
2023	83	No visible effect
56295	8	Orange heart at bottom right-hand corner

Here is an explanation of what happened:

> When you POKEd 83 into location 2023, you placed a heart at the bottom right-hand corner of the screen. 83 is the code for a heart. 2023 corresponds to the bottom right-hand corner of the screen.

Unfortunately, the heart was not visible, because you did not yet assign a color to that position on the screen. When you POKEd 8 into location 56295, you assigned the color orange to the bottom right-hand corner of the screen. This "colored" the heart orange and made it visible.

If you would like a more detailed explanation of why the heart was not visible until you did a POKE into color memory, see the TECHNICAL NOTES at the end of the book in Appendix C.

Experiment with some other values. LOCATION can be any number from 1024 to 2023, or 55296 to 56295. CODE can be any value from 0 to 255. Here are a few more sample values to try:

LOCATION IN MEMORY	CODE TO POKE	EFFECT
2023	90	Diamond in lower right-hand corner
1983	90	No visible effect
56235	1	White diamond in lower left-hand corner
1983	129	Reverse diamond in lower left-hand corner

When you want to stop the program, press STOP-RESTORE.

Let's summarize what we have found out about memory and the screen.

The screen consists of 1000 positions — 25 rows, with 40 positions in each row.

There is a portion of memory (locations 1024 - 1123) which tracks what character is in each position on the screen. There is a one-to-one correspondence between the positions on the screen and the bytes in this portion of memory. Each memory location may have any value from 0 to 255.

There is another portion of memory (locations 55296 - 56295) which tracks the color of each position on the screen. There is a one-to-one correspondence between the positions on the screen and the bytes in this portion of memory. Each memory location may have a value of 0 - 15, for a total of 16 possible colors. In Appendix D is a chart which shows what each possible byte value stands for.

In order for a character to appear on the screen, two codes are needed: a character code in screen memory and a color code in color memory. If you just POKE a code into screen memory, this will not put anything up on the screen; you must also POKE a character into color memory.

How POKEDEMO Works:

Line 100

```
100 SP$ = "
```

sets up a variable which consists of 15 spaces, and which will be used to erase information from the screen.

Lines 110 - 120

```
110 PRINT CHR$(147);
120 PRINT "BAD-CAT !!"
```

do the same job as in the previous programs.

Lines 200 - 220

```
200 PRINT CHR$(19);TAB(200);
210 PRINT SP$:PRINT SP$
220 PRINT CHR$(19);TAB(200);
```

clear lines 6 and 7 from the screen, and then position the cursor at the beginning of line 5.

PRINT CHR$(19); positions the cursor at the top of the screen.

TAB(200); moves the cursor down five lines (which is the same as 200 spaces).

PRINT SP$ puts 15 blank spaces on the screen.

Lines 230 - 240 allow you to INPUT your POKE information:

```
230 INPUT "LOCATION";L
240 INPUT "CODE";C
```

Line 250

```
250 POKE L,C:GOTO 200
```

does the POKEing and then transfers back to line 200.

There are a great number of possible combinations of character colors. The following program will show you all of the possibilities.

RUN THIS PROGRAM:

```
1 REM SHOWALL
100 PRINT CHR$(147);
200 FOR I = 0 TO 255
210 POKE 1024+I,I
220 NEXT I
300 FOR C = 0 TO 15
310 FOR I = 55296 TO 55551
320 POKE I,C
330 NEXT I 340 NEXT C
```

How SHOWALL Works:

This program has two parts. The first part

```
100 PRINT CHR$(147);
200 FOR I = 0 TO 255
210 POKE 1024+I,I
220 NEXT I
```

clears the screen and then places all 256 possible characters on the screen, starting in the upper left-hand corner. (The characters will not be visible to you until the second half of the program assigns colors to the characters.)

The second part

```
300 FOR C = 0 TO 15
310 FOR I = 55296 TO 55551
320 POKE I,C
330 NEXT I
340 NEXT C
```

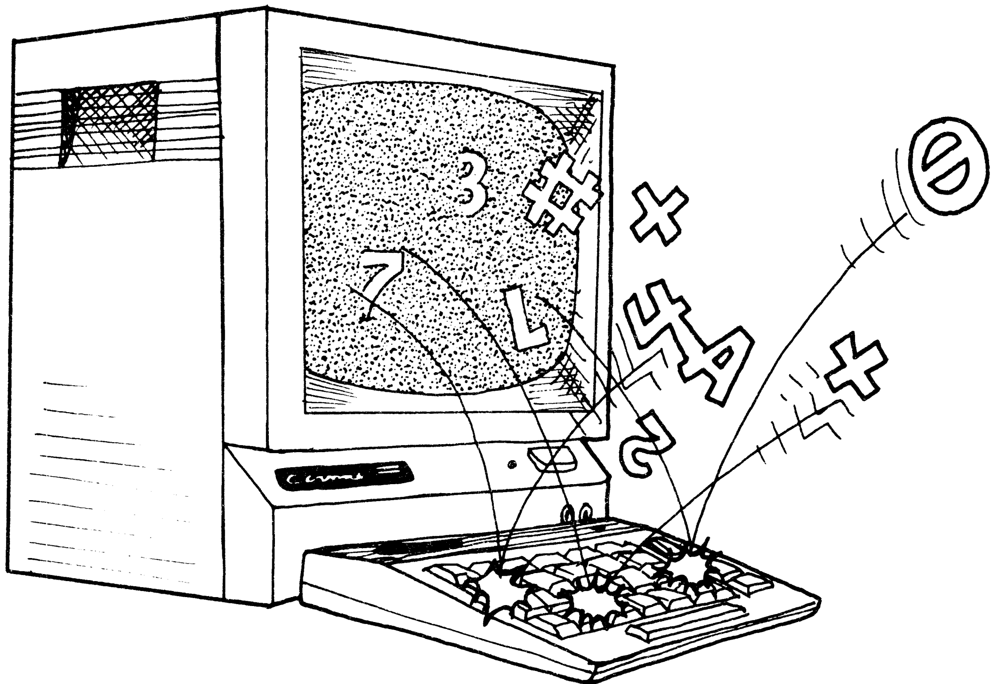
colors the characters in each of the 16 possible colors (0-15). C stands for the color. The FOR - NEXT loop

```
300 FOR C = 0 TO 15
.
.
.
340 NEXT C
```

takes C through the range of possible values, 0 - 15. For each of these values, the 256 characters are "colored" with that value:

```
310 FOR I = 55296 TO 55551
320 POKE I, C
330 NEXT I
```

Actually, there is another complete character set available in the computer, which is not demonstrated in this program. It is discussed in your *Commodore 64 User's Guide,* and we will find out more about it later in this book.



This next program creates a banner from a message that you enter.

RUN THIS PROGRAM:

```
1REM HOTBAN
100 B=1024
110 INPUT "MESSAGE";M$
120 M$=M$ + "  "
130 L=LEN(M$)
200 PRINT CHR$(147);M$
210 FOR I=0 TO 999
220 P=PEEK(B+M)
230 POKE B+I,P
240 M=M+1:IF M=L THEN M=0
250 NEXT I
300 FOR I=55296 TO 56295
```

```
310 R= INT(16*RND(1))
320 POKE I,R
330 NEXT I
400 GOTO 300
```

When you run this program, you will be asked to enter a message. Enter any message, up to 70 characters in length. Then press RETURN. The program will cover the screen with your message, and make it sparkle with many different colors.

How HOTBAN Works:

This program has three parts.

The first part

```
100 B=1024
110 INPUT "MESSAGE";M$
120 M$=M$ + "  "
130 L=LEN(M$)
```

takes care of preliminaries. Line 100 sets variable B to 1024, which is the beginning of screen memory. Line 110 allows you to INPUT a message into variable M$. Line 120 "pads" M$ with a blank space, which will help the message to display more attractively. Line 130 stores the length of the message into variable M$.

The second part of the program

```
200 PRINT CHR$(147);
210 FOR I=0 TO 999
220 P=PEEK(B+M)
230 POKE B+I,P
240 M=M+1:IF M=L THEN M=0
250 NEXT I
```

fills the screen with your message. To begin with, line 200

```
200 PRINT CHR$(147);M$
```

clears the screen and displays the message once in the top left- hand corner. Then the FOR-NEXT loop

```
210 FOR I=0 TO 999
.
.
.
250 NEXT I
```

copies the message onto the rest of the screen. The variable I counts from 0 to 999, which corresponds to the 1000 positions on the screen. For each value of I, a character is "extracted" from your message, and then it is POKEd into one of the locations in screen memory. The "extraction" is done in line 220.

```
220 P=PEEK(B+M)
```

This PEEKs at one of the characters of the message in the top of the screen. M is always less than the length of the message. B is the beginning of screen memory. So

```
P=PEEK(B+M)
```

will always extract one of the characters in the message at the top of the screen. The character which is extracted will depend on the value of M. This starts at 0, increases to the length of the message, drops back to 0 and starts over again. The value of M is controlled by line 230.

The POKEing is done by line 240.

```
240 POKE B+I,P
```

This will POKE the value P into one of the positions in screen memory. The position is determined by I, which starts at 0 and increases to 999.

So in summary, the second part of the program uses the variables M and I to copy the message over the entire screen.

The third part of the program

```
300 FOR I=55296 TO 56295
310 R= INT(16*RND(1))
320 POKE I,R
330 NEXT I
400 GOTO 300
```

makes the screen sparkle with color.

The FOR - NEXT loop in this section traces through the 1000 positions in color memory. At each position, a random number from 0 - 15 is generated.

```
310 R= INT(16*RND(1))
```

This stands for a color code of 0 - 15. The color code is then POKEd into a location in color memory:

```
320 POKE I,R
```

So over and over again, color memory is POKEd with random color codes. This makes your screenful of messages sparkle with many colors. Line 400

```
400 GOTO 300
```

causes the random coloring process to repeat forever. The program will not stop by itself. You must stop it by pressing the STOP key, or by turning off the computer.

Chapter IV

PEEKING AND POKING AROUND MEMORY

We have now explored two regions of memory — screen memory and color memory. There are many other regions of memory where it is interesting to PEEK and POKE. In this section, we will explore a few of them. Not only will this be fun, but also you will get a sense for the variety of ways in which memory is used.

There are several locations in memory which keep track of what keys have been pressed on the keyboard. The following program will show you how two of these memory locations track the keyboard.

SPECIAL KEYS

There are several special keys which are not used very often, and you may not be familiar with everything they can do. Here is a quick summary. For additional information on special keys and other features of the keyboard, see the *Commodore 64 User's Guide* which came with your computer.

Commodore wanted to provide you with a keyboard having a great number of features. However, the number of features would have required several separate keyboards. Commodore solved this problem by providing you with actually several keyboards in one unit. The purpose of some of the special keys is to transform all or part of your regular keyboard into a special keyboard. The following discussion will guide you through the various capabilities of your keyboard. Used normally, the main part of the keyboard produces capital letters. The top row produces numbers.

If you press a SHIFT key (either the left-hand SHIFT key, the right-hand SHIFT key, or the SHIFT LOCK key), this transforms the main part of the keyboard into a "graphics" keyboard. Each key will produce the graphic character shown on the right-hand side of the front part of the key.

> EXPERIMENT: Press a SHIFT key, and while you are holding it down, press the letter Z. A diamond will appear on the screen.

The SHIFT keys transform the top row of the keyboard into a "special symbols" keyboard.

> EXPERIMENT: Press a SHIFT key, and while you are holding it down, press the number 4. A dollar sign ($) will appear on the screen.

The Commodore key (the key in the lower left-hand corner of the keyboard) transforms the keyboard into yet another "graphics" keyboard. When the Commodore key is held down each key will produce the graphic character shown on the left-hand side of the front part of the key.

EXPERIMENT: Press the Commodore key, and while you are holding it, press the letter Z. A "right-angle" symbol will appear on the screen.

The Commodore key transforms the top row of the keyboard into a set of "color switches." The color assigned to each key is marked on the front of the key.

EXPERIMENT: Press the Commodore key, and while you are holding it down, press the number 6. Let go of the Commodore key and press a few keys. All symbols will be colored light green.

The CTRL key transforms the top row of the keyboard into a different set of color switches.

EXPERIMENT: Press the CTRL key, and while you are holding it down, press the number 6. Let go of the CTRL key, and press a few keys. All symbols will be colored dark green.

If you press and hold down a SHIFT key, and then press the Commodore key, this transforms the main part of the keyboard into a "lowercase" keyboard.

EXPERIMENT: Press and hold down a SHIFT key, and then press the Commodore key. Let go of both keys and type a few letters. Everything will appear in lowercase. Now hold down a SHIFT key and type a few letters. Everything will appear in uppercase.

That completes a quick survey of the various keyboards that are packaged into your COMMODORE 64. Now here are some other special keys and key combinations:

Pressing RUN/STOP when a program is running will normally stop the program. (There are some situations where you must press RUN/STOP and RESTORE to stop a program. Also, it is possible to write programs which cannot be stopped, except by turning off the computer.) The combination of RUN/STOP and RESTORE will turn off most special keyboard features and "restore" the computer to normal.

CLR/HOME will move the cursor to the upper left-hand corner of the screen. Pressing CLR/HOME while holding down a SHIFT key will also clear the screen. (However, if there is a program in memory, it will not clear the program or its variables.)

The two CRSR keys allow you to "navigate" the cursor around the screen, right-left and up-down.

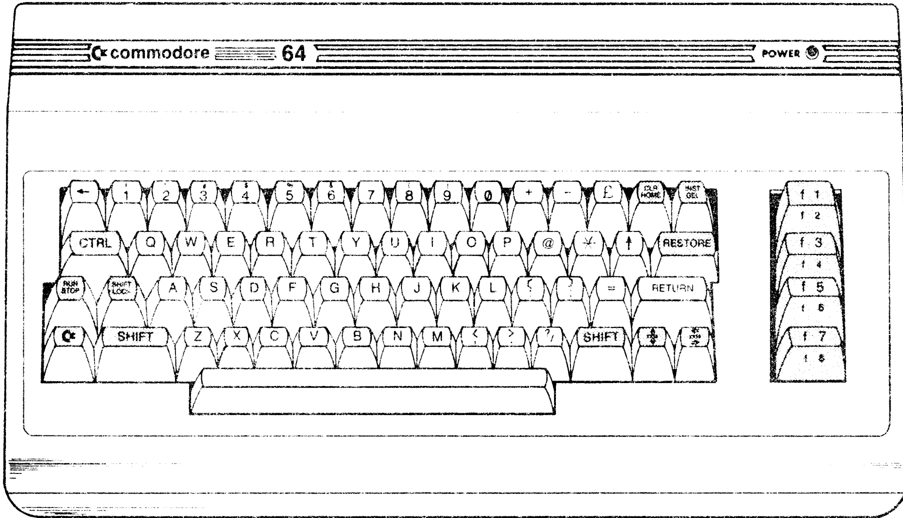
INS/DEL is used for inserts and deletions when you are entering data.

CTRL-9 will cause everything to appear in reverse video.

CTRL-0 will switch the reverse video off.

The function keys F1 - F8 (on the right side of your keyboard) do not have any predefined purpose. These are "wild card" keys which a programmer may define for special use in a program. For example, in a game F1 might mean "Fire missile,"

and F3 might mean "Start new game." In a business program, F1 might mean "Cancel last entry," and F3 might mean "Compute total."



RUN THIS PROGRAM

```
1 REM KBDSTAT
100 PRINT PEEK(197),PEEK(653)
110 GOTO 100
```

You will see the following numbers flash on the screen:

```
64    0
64    0
64    0
```

These are the values currently in bytes 197 and 653 respectively. These two locations track the status of all keys on the keyboard (with the exception of RESTORE, which we will discuss later). When the values of these locations are 64 and 0, this means that no keys are currently being pressed.

Now press Z and hold it down. You will see

```
12    0
12    0
12    0
```

This shows that byte 197 is now 12. And this means that Z is depressed.

Now release the Z key. The display will change back to

```
64    0
64    0
64    0
```

This means that, once again, no keys are currently being pressed.

Now press X and hold it down. You will see

```
23    0
23    0
23    0
```

This means that X is depressed. When you release the X key, the display will change back to

```
64    0
64    0
64    0
```

Experiment with some other keys. You will find that most of them affect only byte 197. For instance:

KEY	BYTE 197	BYTE 65
R	17	0
S	13	0
@	48	0
SPACE BAR	60	0

In fact, there are only three keys that affect byte 653. These are SHIFT, CTRL, and the Commodore key. Let's see what these keys do.

Press and hold down a SHIFT key. You will see

```
64    1
64    1
64    1
```

Release the SHIFT key. You will see

```
64    0
64    0
64    0
```

Press and hold down the Commodore key. You will see

```
64    2
64    2
64    2
```

Release the Commodore key. Now press and hold down CTRL. You will see

```
64    4
64    4
64    4
```

Why are the SHIFT, CTRL, and Commodore keys tracked in a different location than other keys? The answer is that those three keys are meant to be used *in combination with* other keys. For example, you never use a SHIFT key by itself. When you use the SHIFT key, it is always used along with another key. For instance, if you want to produce a $, you press SHIFT and 4 together. The computer must keep track of the fact that you have pressed both SHIFT and 4. So it tracks the two events in separate memory locations.

Let's summarize what we have found out so far about memory and the keyboard.

> There are certain locations in memory which track what keys have been pressed on the keyboard.
>
> Byte 197 tracks the status of all keys on the keyboard, with the exception of SHIFT, CTRL, the Commodore key, and RESTORE. When a key is depressed, a code assigned to that key appears in byte 197. These codes are called "scan codes." A complete table of scan codes is at the end of the book in Appendix D.
>
> Byte 653 tracks the status of SHIFT, CTRL, and the Commodore key. The scan codes for these keys are as follows:

KEY	VALUE
SHIFT	1
Commodore key	2
CTRL	4

Occasionally it is necessary to press SHIFT and CTRL simultaneously, or SHIFT and the Commodore key, or CTRL and the Commodore key. When this happens, how does memory indicate it?

Let's try an experiment. Hold down SHIFT and the Commodore key at the same time. You will see

```
64      3
64      3
64      3
```

The number 3 is the sum of the values for SHIFT and the Commodore key. The computer added the two numbers together!

Try pressing SHIFT and CTRL together. You will see

```
64      5
64      5
64      5
```

which is the sum of the values for SHIFT and CTRL.

Press the Commodore key and CTRL together. You will see

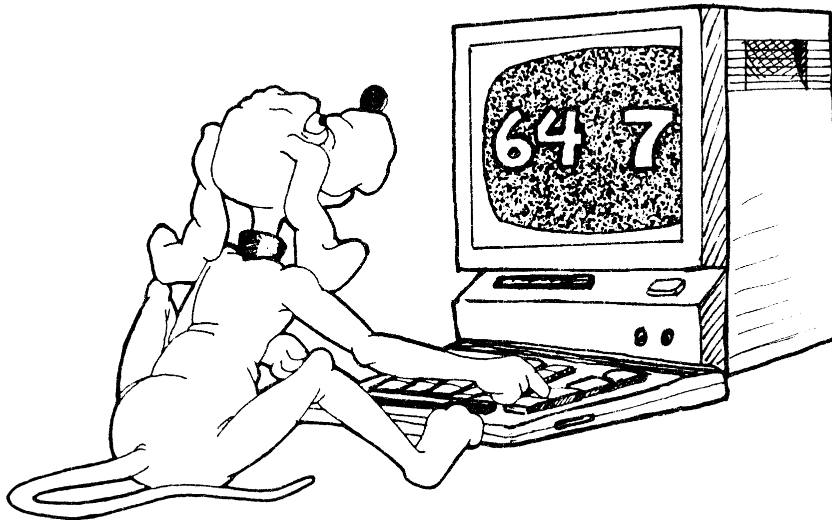
```
64      6
64      6
64      6
```

which is the sum of the values for the Commodore key and CTRL.

What do you think will happen if SHIFT, CTRL, and the Commodore key are all pressed at the same time? Try it!

```
64      7
64      7
64      7
```

The computer adds together the values of all three keys.



The values of the three keys have been cleverly planned so that once you know the value in location 653, you can determine exactly which of the three keys have been pressed.

VALUE IN LOCATION 653	KEYS PRESSED SHIFT	Commodore key	CTRL
0	No	No	No
1	Yes	No	No
2	No	Yes	No
3	Yes	Yes	No
4	No	No	Yes
5	Yes	No	Yes
6	No	Yes	Yes
7	Yes	Yes	Yes

This reveals an interesting fact: With just one byte, it is possible to track several separate pieces of information. In this case, a single byte can track the status of three different keys. We shall have more to say about this later.

Locations 54272 - 54296 are responsible for the sound and music capabilities of your COMMODORE 64. Whenever your computer is producing sounds, this is reflected in the values in this memory area.

When you POKE values into this memory area, the computer will produce sounds. To get an idea of the variety of sounds, let's POKE some random numbers into this area and listen to what happens.

RUN THIS PROGRAM:

```
1 REM ZOUNDS1
100 FOR I=54272 TO 54296
110 R = INT(256 * RND(1))
120 POKE I,R
130 NEXT I
140 GOTO 100
```

You will hear all sorts of strange sounds coming out of your computer. ZOUNDS1 is POKEing random numbers into each of the locations 54272 - 54296, over and over again. The random numbers are generated in line 110

```
110 R = INT(256 * RND(1))
```

This command line generates a random number R between 0 and 255. Then R is POKEd into memory in line 120.

```
120 POKE I,R
```

I varies through the range 54272 - 54296.

When you run ZOUNDS1, you do not hear a solid wall of sound. Instead, you will hear many individual sounds and many moments of silence in between. The reason for this is that not all combinations of numbers produce audible sounds. What can we do to produce more sounds and fewer moments of silence? There are several possible strategies.

> First, we might "tinker" with the program — we could experiment with various small changes until the program "gets better."
>
> Second, we might put a "freeze" feature into the program, so that every time the program produces an interesting sound, we can find out what POKEs were used. This information could help us to design niftier sound programs.
>
> Third, we could try to understand in detail how the computer makes sounds. We could find out what each memory location does to affect the overall sound.

In this chapter, we will use the first two strategies, and then in a later chapter we will get into the third strategy.

First, let's just "tinker" with the program a little. One obvious strategy to try is to modify the random number generator to produce numbers in a more limited range. Or we might limit the range of the FOR - NEXT loop. Here is one experiment which gives better results:

```
1 REM ZOUNDS2
10 POKE 54296,15
100 FOR I=54272 TO 54278
110 R = INT(100 * RND(1))
120 POKE I,R
130 NEXT I
140 GOTO 100
```

This is almost the same program as ZOUNDS1. The random numbers have been limited to the range 0 - 99. The POKE locations have been limited to 54272 - 54278. Also, location 54296 has been permanently set to 15. Location 54296 is the "volume control" for the Synthesizer, and 15 is its maximum volume. Now let's add a "freeze" feature to the program, so that if it produces a sound we like, we can find out how it was done.

```
1 REM ZOUNDS3
10 POKE 54296,15
100 FOR I=54272 TO 54278
110 R = INT(100 * RND(1))
120 POKE I,R
125 PRINT R;
130 NEXT I
135 INPUT X$
140 GOTO 100
```

Your computer is equipped with a clock that helps to schedule and organize its activities. The current time from this timer is tracked in bytes 160 - 162. Here is a program that will enable you to watch the clock run.

RUN THIS PROGRAM:

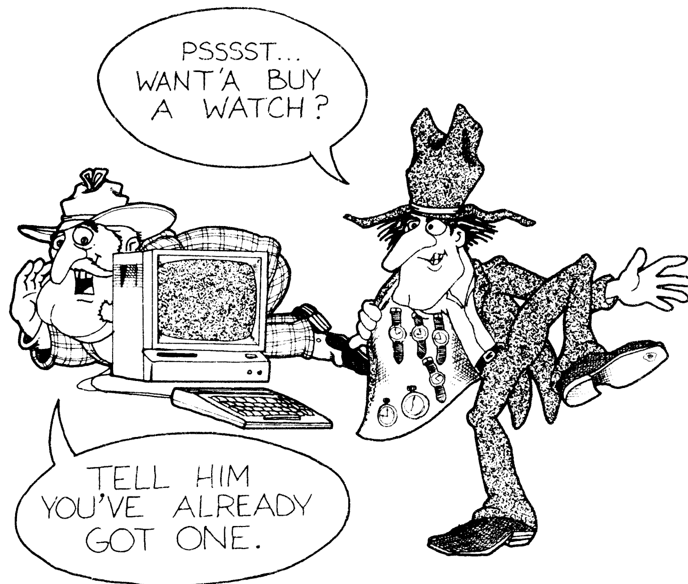
```
1 REM CLOCK
100 A=PEEK(160)
110 B=PEEK(161)
120 C=PEEK(162)
130 D= 65536*A + 256*B + C
140 PRINT A;B;C
150 GOTO 110
```

On your screen you will see a display something like this:

```
0     1     245
0     1     251
0     2     1
```

```
0    2    6
0    2    12
```

Each line is a readout of the current time in the clock. This time is kept in three counters, represented by the three columns of your display. When a counter gets past 255, it rolls back to 0 and the next counter to the left is increased by 1. The clock runs at a speed of approximately 60 "ticks" per second. One 60th of a second is called a "jiffy" in computer jargon. When the computer is turned on, the clock is set to 0 — the value is 0 in locations 160, 161, and 162. So if you read the clock, you can tell how long the computer has been on. The total elapsed time in jiffies is computed as:




```
65536*PEEK(160) + 256*PEEK(161) + PEEK(162)
```

Using the last line from our illustration, the total elapsed time would be

65536 *	0	+	256 * 2	+	12
=	0	+	512	+	12
= 524 jiffies					

This is the same as ZOUNDS2, except that two lines have been added. Line 125

```
125 PRINT R;
```

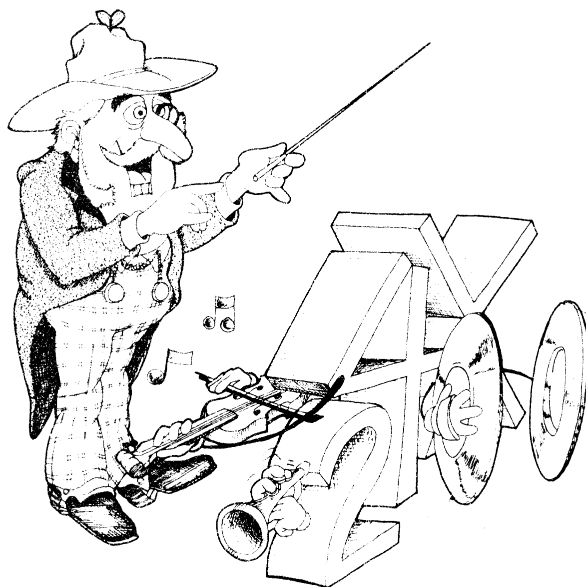
prints the random values that are being POKEd into memory. Line 135

```
135 INPUT X$
```

"freezes" the program each time the FOR - NEXT loop has been completed. This

gives you a chance to decide if you liked the latest sound the program produced, and, if you did, to jot down the POKEs that produced the sound.

In Chapter 13 we will explore the sound capabilities of the COMMODORE 64 in more depth. Even after we have done this, though, you will find a program like ZOUNDS3 useful. The synthesizer is very complex in its capabilities, and a program like ZOUNDS3 helps you to explore those capabilities in ways that you might not have thought of on your own.



Since one second = 60 jiffies, this is equal to a little less than nine seconds.

When you press the letter Z on the keyboard, the computer draws a Z on the screen. In order to do this, the computer has to know how to draw a Z. The letter Z has a certain shape, and the computer must know that shape. How does the computer know the shape of the letter Z?

The answer is that there is a table of values, starting at location 53248, which describes the shapes of the letters of the alphabet and all other symbols used by the computer. This table is sometimes known as a "shape table." Whenever the computer wants to draw a symbol on the screen, it consults the shape table to find out the shape of the character. We are going to find out how the shape table works.

From the point of view of the computer, any symbol is imagined as an 8 by 8 grid of dots, where each dot may be "on" or "off." Here is how a Z looks to the computer.

```
" * * * * * * *
" " " " " * * "
" " " " * * " "
" " " * * " " "
" " * * " " " "
" * * " " " " "
" * * * * * * *
" " " " " " " "
```

An 8 by 8 grid of dots, like the one above. That's what a Z is to the computer. If you examine a Z on your screen, you may be able to see the individual dots that make up the character. The pattern of dots is exactly like the above diagram.

The shape table describes for each character exactly which dots in the grid are on, and which ones are off. It takes 8 bytes in the shape table to describe the entire grid. One byte is required to describe each row in the grid. In the above example, the 8 bytes are as follows:

```
. * * * * * * *      126
. . . . . * * .        6
. . . . * * . .       12
. . . * * . . .       24
. . * * . . . .       48
. * * . . . . .       96
. * * * * * * *      126
. . . . . . . .        0
```

In the next chapter, we will find out how each row is translated into a single one-byte number.

If the information in the shape table could be changed, you could define new symbols that could be produced from the keyboard. For instance, you could modify the shape table so that when the keys

A B C

are pressed on the keyboard, the following three symbols would appear on the screen:

А В Г

or

АБВ

The bad news is that the shape table is "protected" from alteration — if you attempt any POKEs into the region where the shape table is located, the values in the table will remain unchanged. (We will find out more about this later.)

The good news is that you can set up your own shape table in a different part of memory, and then tell the computer to use your table instead of the standard shape table. You can use your own shape tables to define your own symbols. Let's find out how to do this. The following program will set up a shape table which generates symbols known as "bar codes." You have probably seen bar codes in grocery stores — they are a common method for identifying and pricing merchandise.

Our program will set up the new shape table beginning at location 12288. Here is a partial list of the shapes which will be produced by the new shape table.

KEY PRESSED ON THE KEYBOARD	SHAPE WHICH APPEARS ON SCREEN
A	I
B	I
C	II
D	I
E	I I
F	II
G	III
H	I
I	I I
J	I I
K	I II
L	II
M	II I
N	III
O	IIII

RUN THIS PROGRAM:

```
1 REM BARCODES
100 PRINT CHR$(147);"FONE HOME"
200 FOR C=0 TO 63
210 FOR D=0 TO 7
220 POKE 12288 + 8*C + D, C
230 NEXT D
240 NEXT C
300 POKE 53272, 29
```

This program will display the message FONE HOME on the screen, then there will be a pause of about 15 seconds, while the computer is setting up the new shape table. Then the message will be transformed into bar codes.

(The exact appearance of the bar codes will depend on the kind of screen you have. Some screens will blur the bars together slightly.)

Your keyboard has been converted into a "bar code" keyboard. When you press keys, instead of the usual symbols appearing on the screen, you will see bar codes instead. Each key produces a different bar code.

How BARCODES Works:

For the time being, we will just outline the main features of the program. The details will be clarified in the next chapter.

Line 100

```
100 PRINT CHR$(147);"FONE HOME"
```

clears the screen and displays the message FONE HOME at the top of the screen.

Lines 200 - 240

```
200 FOR C=0 TO 63
210 FOR D=0 TO 7
220 POKE 12288 + 8*C + D, C
230 NEXT D
240 NEXT C
```

create the new shape table, starting at location 12288. The table consists of 64 groups of 8 bytes each. Each group of 8 bytes defines one character, as we shall see in the next chapter. The values of the bytes are as follows:

The first 8 bytes (12288 - 12295) are all 0
The next 8 bytes (12296 - 12303) are all 1
The next 8 bytes (12303 - 12311) are all 2

and so on. Each group of 8 bytes has a value of 1 more than the preceding group. Altogether this defines 64 characters.

The variable C counts through the 64 groups of 8 bytes each. Within each group, the variable D counts through the 8 bytes.

Line 300

```
300 POKE 53272, 29
```

tells the computer to use the shape table beginning at location 12288. Depending on the value which is POKEd into location 53272, it is possible to have shape tables at locations other than 12288.

When you would like to get your keyboard back to normal, press RUN/STOP - RESTORE.

Chapter V

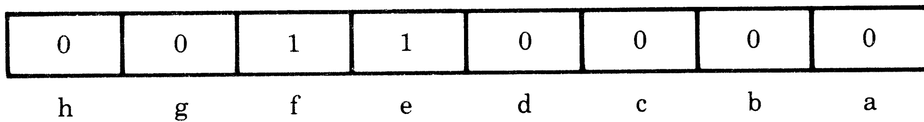
THE STORY OF Z

In the last chapter, we found out that the shape of every symbol is described in a shape table. Each symbol is imagined as an 8 by 8 grid of dots. Each of the 8 rows in the grid is represented by one byte in the shape table. For instance

```
. * * * * * * *        126
. . . . . * * .          6
. . . . * * . .         12
. . . * * . . .         24
. . * * . . . .         48
. * * . . . . .         96
. * * * * * * *        126
. . . . . . . .          0
```

Now let's find out the precise relationship between the numbers in the shape table and the pattern of dots in each row of the grid. (If you are not interested in technical details, it's okay to skim through this discussion.) If you could look at a single byte of memory under a microscope, you would discover that it consists of eight tiny storage compartments. Each compartment holds one "bit" of information. The value of a bit can be either 0 or 1 ("off" or "on"). Here is how a byte value of 48 looks in memory.

When you do a PEEK into memory, the computer calculates a value for the entire byte, based on the value of the individual bits. The formula is as follows. Let's call the eight positions a - h, starting from the right, i.e.,



Each position is assigned a value:

h	128
g	64
f	32
e	16
d	8
c	4
b	2
a	1

To calculate the value for the entire byte, add together the values of the bits that are "on."

Let's try a few examples:

0	0	1	1	0	0	0	0
h	g	f	e	d	c	b	a

The bits which are on are: f, e

The value of f is	32
The value of e is	16
The value for the entire byte is	48

0	0	0	0	1	1	0	0
h	g	f	e	d	c	b	a

The bits which are on are: d, c

The value of d is	8
The value of c is	4
The value for the entire byte is	12

0	1	1	0	1	1	0	1
h	g	f	e	d	c	b	a

The bits which are on are: g, f, d, c, a

The value of g is	64
The value of f is	32
The value of d is	8
The value of c is	4
The value of a is	1
The value for the entire byte is	109

Now we can summarize how the 8 bytes in the shape table describe the shape of a character.

Each character consists of 8 rows of 8 dots each.

For each row, there is a corresponding byte in the shape table. The pattern of bits in that byte describe the points in that row. So,

CHARACTER	BYTE IN SHAPE TABLE	BIT PATTERN IN BYTE
" * * * * * * *	126	01111111
" " " " " * * "	6	00000110
" " " " * * " "	12	00001100
" " " * * " " "	24	00011000
" " * * " " " "	48	00110000
" * * " " " " "	96	01100000
" * * * * * * *	126	01111111
" " " " " " " "	0	00000000

In conclusion, the pattern of bits in the shape table matches the pattern of points in the character.

Here is some jargon about bits with which it is useful to be familiar.

> Bit h — the first one from the left — is known as the "high order bit," since its value is 128.
>
> Bit a — the first one from the right — is known as the "low order bit," since its value is 1.
>
> The eight bits are sometimes referred to as "bit 0," "bit 1," "bit 2," and so on reading from the right. I.e., "bit 0" is the low order bit and "bit 7" is the high order bit. Warning, occasionally this terminology is used in reverse order!

Altogether, there are 256 different possible combinations of bits in a single byte; the value for these various combinations ranges from 0 to 255. A chart of all possible combinations is in Appendix D. By using bits, it is possible to represent any number. This is what is known as "Base Two" or "Binary" notation. It is known as Base Two because, in any number, each digit may have only two values. Our ordinary human notation for numbers is called Base 10, since each digit may have any of ten different values. Here are some examples of numbers in Base Two

BASE TEN	BASE TWO
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

BASE TEN	BASE TWO
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
32	100000
64	1000000
99	1100011
100	1100100
101	1100101
102	1100110
128	10000000
256	100000000
512	1000000000
1024	10000000000

Computers frequently express numbers in binary notation. Engineers have discovered that they can design computers that are faster and less expensive when binary notation is used extensively. The main reason for this is that the most fundamental events in a computer's "life" are binary — switches being open or closed, a flow of current being on or off, a bit of data being present or not present in a certain location.

As we proceed through this book, we shall frequently encounter binary notation. Also, there are certain specific binary numbers which are "favorites" of the computer, and which we shall encounter over and over again:

BASE TEN	BASE TWO
16	10000
32	100000
64	1000000
128	10000000
256	100000000
512	1000000000
1024	10000000000
4096	1000000000000
32768	1000000000000000
65536	10000000000000000

Human beings generally dislike binary notation. All those 0s and 1s are hard on the eyes, and it's very difficult to read large numbers. Here we have a fundamental difference between computers and human beings: Computers generally prefer Base Two, while human beings generally prefer Base Ten.

Now let's have some fun with shape tables. Some of the following discussion will be somewhat technical — in order to play with shape tables, we must get involved with binary notation. If you are not interested in technical details, it's okay to skim through the next two examples.

First, we will modify the shape table from the last chapter, so that when you press the symbol @ on your keyboard, a lightning bolt appears on the screen. The program is the same as BARCODES, except that lines 400 - 500 have been added at the end.

RUN THIS PROGRAM:

```
1 REM LIGHTNING
100 PRINT CHR$(147);"FONE HOME"
200 FOR C=0 TO 63
210 FOR D=0 TO 7
220 POKE 12288 + 8*C + D, C
230 NEXT D
240 NEXT C
300 POKE 53272, 29
400 FOR I=12288 TO 12295
410 READ C
420 POKE I,C
430 NEXT I
440 PRINT "@@@@@@@@"
500 DATA 3,6,12,30,3,6,12,24
```

This program will produce the same screen display as BARCODES, except that at the top of the display you will see eight lightning bolts.

How LIGHTNING Works:

The first part of the program (lines 100 - 300) is the same as BARCODES.

The rest of the program alters the first 8 bytes in the table, and then prints something on the screen.

Lines 400 - 430

```
400 FOR I=12288 TO 12295
410 READ C
420 POKE I,C
430 NEXT I
```

alter the first 8 bytes of the shape table. These 8 bytes describe the shape of the character @. The program sets these 8 bytes to the values in the DATA line at the end of the program:

```
500 DATA 3,6,12,30,3,6,12,24
```

These values describe the shape of a lightning bolt:

GRID FOR LIGHTNING BOLT	BYTE VALUE	ENTRY IN SHAPE TABLE

......**	3	00000011
.....**.	6	00000110
....**..	12	00001100
...****.	30	00011110
......**	3	00000011
.....**.	6	00000110
....**..	12	00001100
...**...	24	00011000

Finally, line 500

```
500 PRINT "@@@@@@@@"
```

tells the computer to print eight @ symbols. But since we have altered the shape table, the computer will display eight lightning bolts instead! If you press the @ key on your keyboard, additional lightning bolts will be displayed.

It's a lot of work entering all of the codes to design your own set of characters. You can sometimes save yourself some work by borrowing from the shape tables in ROM. The following program will show you how to do this.

Before using this program, clear your computer completely. To do this, turn it off, wait ten seconds, and turn it back on.

ENTER THIS PROGRAM:

```
1 REM COPYSHAPES
100 POKE 56334,0:POKE 1,51
200 FOR I=0 TO 511
210 P=PEEK(53248 + I)
220 POKE 12288+I, P
230 NEXT I
300 POKE 1,55:POKE 56334,1
310 POKE 53272,29
```

In its present form, this program is not too interesting. It copies the first 64 characters of the shape table to a new location and tells the computer to use this new shape table. Since the new table is like the old one, you will not see any differences. However, there are some simple modifications of this program which produce very interesting results!!

EXPERIMENT: Change line 210 to

```
210 P=PEEK(53256 +I)
```

and run the program. (The program takes about 15 seconds to run.) Whatever was on the screen will become unrecognizable. Press the letters A, B, and C on your keyboard. On your screen you will see

```
BCD
```

What's going on here? Try pressing some other letters. Each time the screen will display "one-up" from the letter you pressed. If you press P, the screen will display Q; if you press X, the screen will display Y; and so on.

We got this result by modifying line 210 to start at location 53256, instead of 53248. The effect of this is to move the entire shape table by one character. This means that the shape for B is assigned to A, the shape for C is assigned to B, and so on.

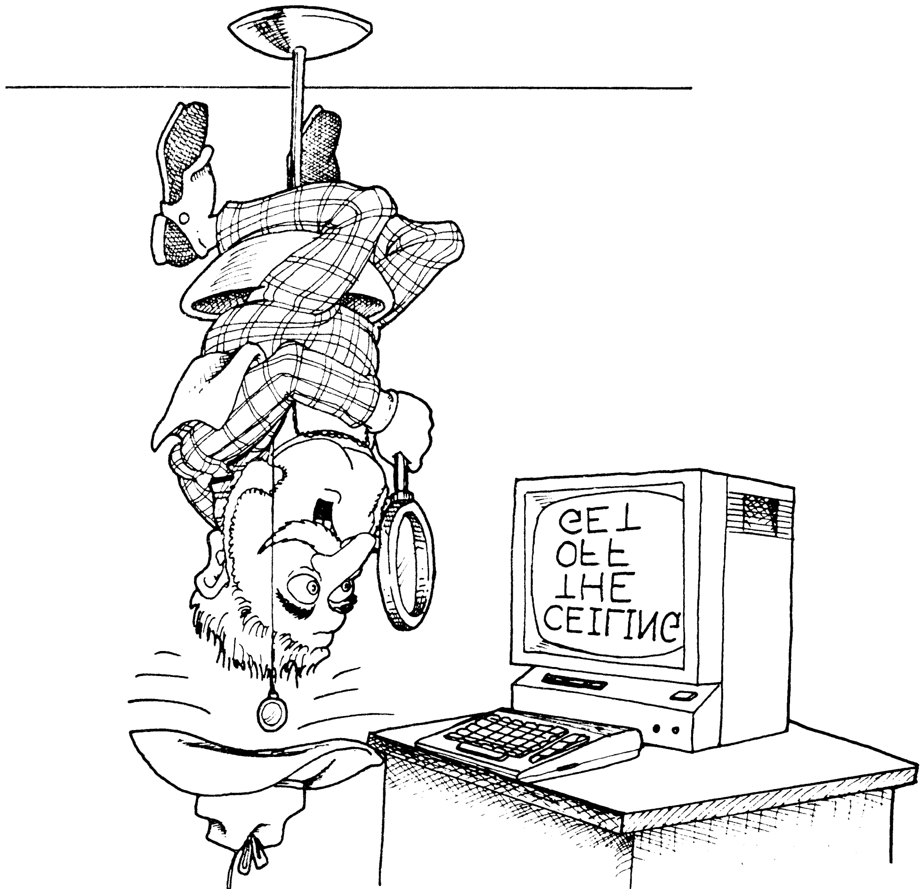
EXPERIMENT: Modify COPYSHAPES to look like this (the only changes are line 205 and line 220):

```
1 REM WOM
100 POKE 56334,0:POKE 1,51
200 FOR I=0 TO 511
205 D=D+2:IF D>16 THEN D=2
210 P=PEEK(53248 + I)
220 POKE 12297+I-D, P
230 NEXT I
300 POKE 1,55:POKE 56334,1
310 POKE 53272,29
```

This program will turn all character shapes upside down. To invert a character shape, it is necessary to get each group of 8 bytes in the shape table into reverse order. For example

The shape codes for Z are: 126, 6, 12, 24, 48, 96, 126, 0.

The shape codes for an upside-down Z are: 0, 126, 96, 48, 24, 12, 6, 126.



Our changes in lines 205 and 220 get each group of 8 bytes into reverse order. Doing this involves a tricky combination of additions and subtractions. If you want to understand exactly how it works, trace the program through for the first few values of I.

There are other slight variations of the COPYSHAPES program which produce astounding effects. For more on this, see Appendix A, EXTRA TOPICS.

Chapter VI

A PROGRAM TO SCAN MEMORY

The memory of your COMMODORE 64 is chock full of interesting information, and thus far we have explored only a small fraction of it. To help you to explore memory further, we are going to write a program called SNOOP, which makes it easy to scan through memory. We will use SNOOP a number of times in the rest of the book.

ENTER THIS PROGRAM:

```
1 REM SNOOP
100 INPUT "START AT";B
200 PRINT CHR$(147);"START= ";B
300 FOR LI=1 TO 10
400 FOR I=B TO B+7
410 P=PEEK(I)
420 IF P<32 OR P>95 THEN P=32
430 PRINT "     ";CHR$(P);
440 NEXT I
450 PRINT
500 FOR I=B TO B+7
510 P=PEEK(I)
520 P$=RIGHT$("     "+STR$(P),4)
530 PRINT P$;
540 NEXT I
550 PRINT
600 B=B+8
610 NEXT LI
620 GOTO 100
```

When you run the program, the following question will appear:

```
START AT?
```

The program is asking at what location in memory you want to position yourself. Type 58488 and press RETURN.

The computer will now display 80 bytes of memory, beginning at location 58488.

```
START= 58488
```

```
       *    *    *    *    C    O
  32   42   42   42   42   32   67   79
   M    M    O    D    O    R    E
  77   79   79   68   79   82   69   32
   6    4         B    A    S    I    C
  59   52   32   66   65   83   73   67
   V    2    *    *    *    *
  32   86   50   32   42    4   42   42
                  6    4    K         R
  13   13   32   54   52   75   32   82
   A    M         S    Y    S    T    E
  65   77   32   83   89   83   84   69
   M                        M
  77   32   32    0   92   72   32  201
 255  170  104  144    1  138   96  170
 170  170  170  170  170  170  170  170
 170  170  170  170  170  170  170  170
```

START AT?

The computer is displaying 80 bytes of memory, beginning with location 58488. SNOOP displays the value in each byte-location. Additionally, whenever the value might possibly be an ASCII code for a character, that character is displayed above the line. ASCII is a standard coding system used for representing letters, numbers, and other symbols in memory. This gives us some clues for understanding the codes in memory. For instance, in our present example you will recognize

```
       *    *    *    *    C    O
  32   42   42   42   42   32   67   79
   M    M    O    D    O    R    E
  77   79   79   68   79   82   69   32
   6    4         B    A    S    I    C
  59   52   32   66   65   83   73   67
        V    2         *    *    *    *
  32   86   50   32   42   42   42   42
                  6    4    K         R
  13   13   32   54   52   75   32   82
   A    M         S    Y    S    T    E
  65   77   32   83   89   83   84   69
   M
  77
```

This is the greeting that appears on the screen when you turn on the computer.

You are probably wondering what the rest of the codes are on the screen. We will find out more about that in the next chapter. Let's display another part of memory. Right now, the computer is asking you for another STARTING BYTE. Type 41642 and press RETURN. On your screen you will see some information like this:

START= 41642

```
D    I    V    I    S    I    O    N
68   73   86   73   83   73   79   78
     B    Y         Z    E    R
32   66   89   32   90   69   82   207
I    L    L    E    G    A    L
73   76   76   69   71   65   76   32
D    I    R    E    C         T    Y
68   73   82   69   67   212  84   89
P    E    M    I    S    M    A
80   69   32   77   73   83   77   65
T    C         S    T    R    I    N
84   67   200  83   84   82   73   78
G         T    O    O    L    O
71   32   84   79   79   32   76   79
N         F    I    L    E         D
78   199  70   73   76   69   32   68
A    T         F    O    R    M    U
65   84   19   70   79   82   77   85
L    A         T    O    O    C
76   65   32   84   79   79   32   67

START AT?
```

This section of memory holds error messages: DIVISION BY ZERO, ILLEGAL DIRECT, TYPE MISMATCH, STRING TOO LONG, and so on. Not everything makes obvious sense. For instance, the last letter seems to be missing from each error message, i.e., DIVISION BY ZER, ILLEGAL DIREC, TYPE MISMATC, STRING TOO LON. What happened to the last letter? Also, there is a phrase which is not listed in the Commodore manuals as a possible error message FILE DAT. Is that an error message which Commodore was planning on using, but decided not to? Or is it an actual error message that can come up, but which wasn't documented in the manuals? What do you think?

As we proceed through this book, you will learn more about the way memory is organized, and the way in which codes are used, and this will help you to interpret the mysteries of the computer's internals.

How SNOOP Works:

Most of SNOOP will be familiar to you from programs we have written earlier. We will just discuss the features that are new or different.

Line 100

```
100 INPUT "START AT";B
```

obtains the location where you want to start, and stores that location in the variable B.

Line 200

```
200 PRINT CHR$(147);"START= ";B
```

clears the screen and displays the starting location at the top of the screen.

The FOR - NEXT loop in lines 300 - 610

```
300 FOR LI = 1 TO 10
.
.
.
610 NEXT LI
```

prints 10 lines of information on the screen. Each line consists of 8 bytes of PEEKs. Each line is displayed in two ways. First, it is displayed translated into ASCII characters wherever possible. ASCII is a standard coding system, used by many computers, which assigns a number to each common symbol. For instance A is assigned the number 65, B is assigned 66, C is assigned 67. When the COMMODORE 64 stores characters in memory, it often uses ASCII codes (but not always!). Lines 400 - 460 translate the codes from memory into ASCII symbols wherever this seems reasonable.

```
400 FOR I=B TO B+7
410 P=PEEK(I)
420 IF P<32 OR P>95 THEN P=32
430 PRINT "     ";CHR$(P);
440 NEXT I
450 PRINT
```

The "reasonable" range for ASCII codes is 32 - 95. Codes in this range stand for ordinary symbols. Codes outside this range generally do not. Line 420 converts all "unreasonable" values to 32, which is the ASCII code for a blank space.

```
420 IF P<32 OR P>95 THEN P=32
```

Line 430 prints the ASCII symbol for P, nicely formatted.

```
430 PRINT "   ";CHR$(P);
```

Only those values which translated into reasonable ASCII characters are displayed. Others are assigned a space.

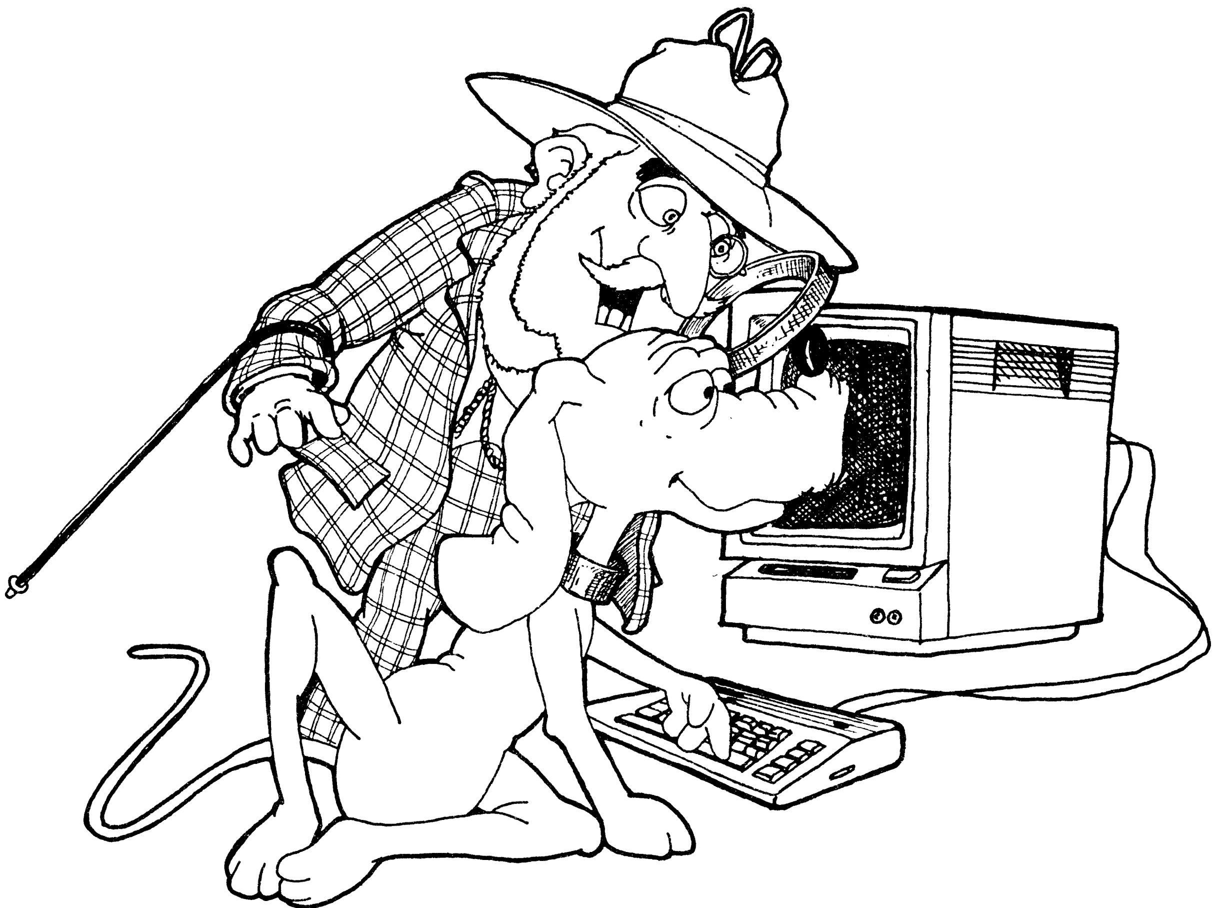
Directly below this, each of the eight values is displayed in numeric form. The following lines of the program do that job.

```
500 FOR I=B TO B+7
510 P=PEEK(I)
520 P$=RIGHT$("    "+STR$(P),4)
530 PRINT P$;
540 NEXT I
550 PRINT
```

Now that you have SNOOP, you can look around memory on your own. Here are some suggestions for regions to explore:

STARTING BYTE	COMMENTS
40960	Helps the computer to process BASIC commands
41120	BASIC command vocabulary
61600	Error messages for disk, tape
2049	Your program, as it is stored in memory

Try looking at some other regions as well. Most of the information will not be easy to interpret (Commodore did not attempt to make memory easy to understand), but you will discover many interesting patterns. In the next chapter, we will find out a great deal more about how to interpret the information in memory.



Chapter VII

THE MAP OF MEMORY

You have now explored a number of interesting parts of memory. Memory is vast, and there is a great deal more that can be explored. To help you find your way around, and to help to understand what you see, we are going to provide you with a map of memory. This map will show you many of the most important regions of memory. In later chapters, we will explore some of these regions in more detail. The memory of your COMMODORE 64 consists of 65,536 memory locations, numbered 0 - 65535. Each location holds one byte of information. The computer uses this memory for many purposes. Here are some of the most important uses.

To track the screen. As we have already seen, the computer uses certain regions of memory to track what is on the screen.

To hold program and variables. When you are running a program in BASIC, the program and its variables are held in memory.

To hold reference information. The computer uses memory to hold reference information that it needs while it is running.

> Information on the shapes of various letters and symbols that can be drawn on the screen.
>
> Lists of error messages that can be displayed on the screen when something goes wrong.
>
> Information on acceptable vocabulary and grammar for BASIC commands.

To hold current-status information. The computer uses portions of memory to track the current status of various activities.

> What line of your BASIC program is currently being executed.
>
> What key or keys have been most recently pressed on the keyboard.
>
> Where the cursor is on the screen.
>
> What time it is, according to the computer's internal clock.
>
> What "peripherals" (disk drive, cassette, printer, ...) are currently communicating with the computer.

To provide a "scratch-pad" for temporary information.

> When the computer encounters a GOSUB in a BASIC program, it needs a

scratch area where it can leave a note to itself where to RETURN to at the end of the subroutine.

When the computer is performing multiplications and divisions, it sometimes needs a scratch area to help it work out the calculations.

When you are storing information on tape or disk, the computer needs an area where it can gather together the information and put it in proper form, before sending it to the cassette or disk unit.

To hold special routines which help the computer to perform its work. Most of these routines are short and perform very specific tasks.

Deciding what position to move the cursor to, each time you press a key on the keyboard.

Clearing the screen when you press SHIFT-CLR.

Keeping track of how far along the computer is in a FOR-NEXT loop.

Displaying the word READY on the screen, when a program is finished.

These routines are held in memory so that they can be called upon quickly. They are written in a high-speed language called "6502 Machine Language." The vocabulary of this language is limited to only the most fundamental activities which the computer can perform. It is possible for a clever programmer to write extremely fast programs in this language. However it's hard work to write programs in machine language, as compared to BASIC. We will find out more about machine language later.

Those are some of the main ways in which memory is used. Now let's survey memory region-by-region, to find out what goes on where. (The locations listed here are the ones which are normal for the computer. It is possible to shift around some of the regions of memory. More on this later.)

Locations 0 - 255: Reference information and current status information. This section of memory is crammed with tables and other data which is needed frequently.

The location in memory where your BASIC program begins.

Information on the regions in memory which are occupied by variables from your BASIC program.

Information on the location of various machine language routines.

What time it is, according to the computer's internal clock. Information on keys most recently pressed on the keyboard.

Locations 256 - 511: Short-term storage. When the computer encounters a GOSUB, it leaves a note to itself in this area, reminding it where to RETURN to at the end of the subroutine. This region also helps to track the status of FOR-NEXT loops.

Locations 512 - 1023: More reference and current status information.

Locations 1024 - 2023: Screen memory. As we already have learned, this area tracks what character is on each position of your screen. This region is exactly 1000 bytes, which corresponds to the 1000 positions on your screen.

Locations 2040 - 2047: Sprite controls. When you are programming with sprites, this area is used to help control the sprite displays.

Locations 2048 - 40959: Your BASIC program and variables. This area is reserved for your use. The text of your program is normally stored in the first part of this region. Immediately after the program, the variables are stored, with the exception of string variables. String variables are stored at the far end of the region. We will find out more about this in the next few chapters.

Locations 40960 - 49151: BASIC Interpreter. When you are running a program in BASIC, the computer breaks up each command into a set of simple tasks. For example, the following command

```
100 PRINT 7*W
```

is broken up into a series of tasks like these:

Transfer the contents of W into a scratch-pad area.

Using the scratch-pad, multiply the value of W by 7 and record the results in a scratch area.

Arrange the results in a nice format.

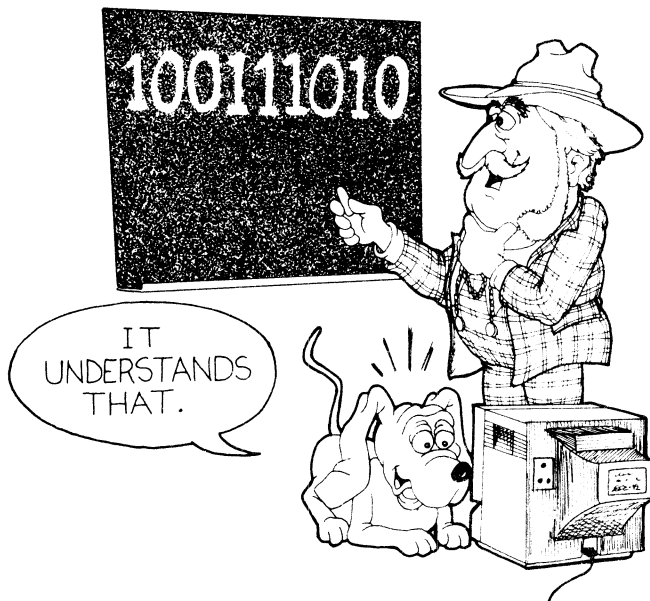
Display it on the screen.

Actually, there are even more tasks than this, to execute a single BASIC command.

Each of the tasks is performed by means of a machine language routine stored in this region, or in another region called the KERNAL, discussed below.

All commands in BASIC are executed in a similar manner. Each command is analyzed by the computer and broken up into a series of simpler tasks. Each of these tasks is then executed by means of a machine language program that is in the computer's memory. The analysis and execution of commands is done by the BASIC Interpreter region, with occasional calls to the KERNAL routines for help with input/output tasks.

BASIC is what is known as a "high-level" language. Its vocabulary is too "high-level" for the computer to understand directly. In order to be able to process a BASIC command, the computer must translate it into a sequence of tasks that the computer can perform. Each of these tasks is the equivalent of a routine in machine language. A single BASIC command may translate into over a hundred machine language instructions! BASIC is much more convenient to write programs in than machine language.



The machine language programs needed to process BASIC are divided between two regions: The BASIC Interpreter region and the region called the KERNAL, described below. The KERNAL is responsible for most of the input/output operations, such as: loading or saving programs to a 1541 disk drive or datassette recorder, detecting which keys were pressed, and plotting characters to the screen.

Locations 49152 - 53247: Additional space for your use. The computer does not assign any special use for this area of memory. You are free to use it as you like.

Locations 53248 - 54271: Screen information. This region holds information relating to screen colors, scrolling, and sprites.

Locations 54272 - 55295: Sound processing. The computer is capable of generating extremely complex sounds. This area of memory is used by the computer to help it generate those sounds. We will explore this region in Chapter 13, SOUNDS.

Locations 55296 - 56319: Color memory for the screen. As we found out earlier, portions of this region track the color of each position on the screen. There are 1000 bytes in this portion, corresponding to the 1000 positions on the screen. The remainder of this region is unassigned.

Locations 56320 - 57343: Tie-ins with peripherals. This area of memory helps the computer to "talk" with the keyboard, disk drive, joystick, modem, printer, or other devices which are attached to the computer. It regulates the speed of transmission of data, it acts as a holding area for incoming and outgoing data, and it keeps things organized so that, for instance, signals from your joystick don't interfere with signals from the keyboard.

Locations 57344 - 65535: The KERNAL. This is a set of machine language routines for performing the most fundamental tasks.

Deciding what position to move the cursor to, each time you press a key on the keyboard.

Clearing the screen, when you press SHIFT-CLR.

Updating the internal clock, in locations 160-162, every 1/60 of a second.

Interrupting your program, when you press STOP or STOP-RESTORE.

Controlling the speed at which data is sent to the printer.

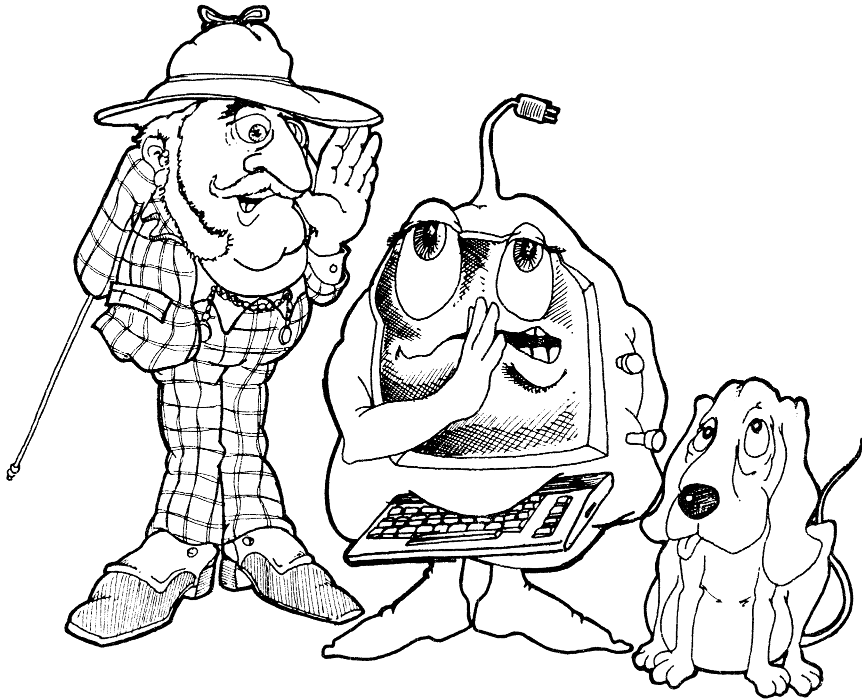
Coordinating the various activities of the computer, so they don't get tangled with each other.

This completes our map of memory. We shall explore some of these regions in more detail in later chapters.

Now that you are becoming more familiar with the computer's memory, you can see that it is not always organized in a convenient manner for human beings. Information is usually expressed in codes, rarely in plain English, and the arrangement of the information in memory can seem rather unnatural. Why, for example, does screen memory begin at location 1024? Why not an easy-to-remember location, such as 1000? And why does the computer express information in codes, instead of plain English?

These are very important questions, because they take us to the FIRST RULE OF COMPUTER SNOOPING:

The computer is an alien form of lower intelligence.



The computer is an intelligent machine; however its level of intelligence is much lower than that of human beings. Also the way in which it thinks is much different from the way in which human beings think. It is an "alien" form of intelligence. When you snoop inside a computer, you should not expect to find things arranged in a manner that is convenient for you. They are arranged in a manner that is convenient for the computer. About the only time that the computer makes things easy for you is when it has to communicate with you.

> When the computer is waiting to receive commands from you, it allows these commands to be entered in a language like BASIC, which is fairly easy for human beings to work with.
>
> However, once you have entered a command, the computer quickly transforms that command to a form that is easy for the computer to work with.
>
> The computer does most of its processing in ways that are alien to human thought. In general, the computer's methods are simple-minded and extremely tedious, by human standards. But they are methods which are convenient for the computer.
>
> When the computer wants to communicate information back to you, it usually goes through a number of operations to convert that information to a form that is convenient for human beings.

The fundamental processes of computers are mainly quite simple, and by the end of the book, you will have become acquainted with most of them. After that, to become an advanced computer snooper is mainly detail work (Warning: a lot of detail work !!)

To conclude this chapter, let's take care of a few "loose ends" in our discussion of memory.

1. A moment ago, we asked why regions of memory do not have locations which are nice, round numbers. For instance, why does screen memory begin at location 1024 rather than, say, 1000?

Part of the answer is that many of the locations *are* nice, round numbers to the computer, although not necessarily to us. To a human being, nice round numbers are: 100, 1000, 10000. These numbers are known as "powers of 10," i.e.,

$$100 = 10^2 = 10 * 10$$
$$1000 = 10^3 = 10 * 10 * 10$$
$$10000 = 10^4 = 10 * 10 * 10 * 10$$

(The notation "10^4" means "four 10s multiplied together", i.e., 10 * 10 * 10 * 10.)

Human beings find powers of 10 easy to remember and work with. Our entire number system is based on powers of 10 — indeed, it is known as "Base Ten."

As was mentioned earlier, most computers are based on powers of 2, rather than powers of 10. Here is a list of some of the powers of 2.

$$4 = 2^2$$
$$8 = 2^3$$
$$16 = 2^4$$
$$32 = 2^5$$
$$64 = 2^6$$
$$128 = 2^7$$
$$256 = 2^8$$
$$512 = 2^9$$
$$1024 = 2^{10}$$
$$2048 = 2^{11}$$
$$4096 = 2^{12}$$
$$8192 = 2^{13}$$
$$16384 = 2^{14}$$
$$32768 = 2^{15}$$
$$65536 = 2^{16}$$

You will see these numbers frequently as you explore your computer. For instance,

There are 8 bits in a byte.

A byte can have 256 possible values.

Screen memory normally begins at location 1024.

BASIC programs are normally stored starting at location 2048.

There are 65536 locations in memory.

Why are powers of 2 so common in your computer? Computer engineers have discovered that by making heavy use of powers of 2 in the design of computers, they can build machines which are faster and cheaper. It would certainly be possible to design a computer where powers of 10 occur frequently, but the computer would probably not be as fast or efficient as power-of-two machines. A detailed explanation would require a lot of mathematics, but the main idea is that modern computers make extensive use of switches. A switch is either on or off, which amounts to two possibilities. That is where the number 2 comes from. Since computers make extensive use of switches, powers of 2 arise naturally. (If someone were to design a computer which depended on a concept different from switches, powers of 2 might not show up frequently.)

2. We mentioned earlier that some of the standard locations in our map can be altered. A couple of examples will illustrate how this is done:

> The standard location for a BASIC program is starting at location 2048. The standard location is recorded in bytes 43-44 of memory. If the value of these bytes is altered, the starting location for BASIC programs will be relocated.
>
> The standard location for screen memory starts at location 1024. The standard location is recorded in byte 53272 of memory. If the value of that byte is altered to an appropriate value, the starting location for screen memory will be relocated.

The information in bytes 43-44, or in byte 53272, is known as "master pointer" information. A master pointer "points" to the location in memory of something important. It is like a road sign that tells the computer the location of something important. You will find out more about master pointers in the remainder of this book. Major regions of memory can be relocated by altering master pointers. (Warning, this can be tricky. Sometimes there are several master pointers that depend on one another. If you alter one without altering the others, you'll really screw things up!)

3. If you enter a program and then turn off the computer, the computer will "forget" your program. When you turn the computer back on, the program will no longer be in the computer's memory. However, there is some "knowledge" which the computer never seems to lose. For instance,the computer always remembers what greeting to put on the screen when you turn it on. It remembers how to execute BASIC commands. It remembers a collection of Error Messages which it can display when you make a mistake.

These facts show that there are two kinds of memory in the computer.

The first kind of memory goes blank whenever the computer is turned off. This kind of memory is called "volatile memory," "Read-Write memory," or "Random Access Memory" (RAM). We will refer to it as RAM, since this is the most common term for it. The COMMODORE 64 has 64K of RAM. (That's how it got the name "COMMODORE 64".) When the computer is off, RAM contains no information. When the computer is on, information may be recorded in RAM. Also, while the computer is

on, information that is in RAM may altered or erased. When the computer is turned off, everything that was in RAM is lost.

The second kind of memory is called "Read Only Memory" (ROM). ROM has information permanently frozen into it. This information can be "read," but it cannot be altered or erased. That is why it is called "Read Only Memory." The information in ROM is always there, even when the computer is off. The COMMODORE 64 is equipped with three important ROM units:

> Character ROM: This holds the shape tables for all symbols that can be drawn by the computer. There is a total of 4K of Character ROM.
>
> BASIC Interpreter ROM: This holds the machine language routines for processing BASIC commands. BASIC Interpreter ROM is normally assigned to locations 40960 -49151 in memory. There is a total of 8K of BASIC Interpreter ROM.
>
> KERNAL ROM: This holds the machine language routines for the KERNAL. KERNAL ROM is normally assigned to locations 57344 - 65535 in memory. There is a total of 8K of KERNAL ROM.

So altogether, the computer has 84K of memory units:

RAM	64K
Character ROM	4K
BASIC Interpreter ROM	8K
KERNAL ROM	8K

4. (Warning: This part of our discussion gets rather intricate. If you are not interested in technical details, it's okay to skim this part.)

We have just seen that the COMMODORE 64 is equipped with 84K of memory units. Unfortunately, because of an engineering limitation of one of the components of the computer, the computer cannot use all of these units at one time! The most it can fully use at any one time is 64K. In other words, the COMMODORE 64 is slightly overloaded with memory units.

You can bet that the engineers at Commodore spent plenty of late nights in their laboratories, trying to figure out what to do about this problem. Here is an outline of the solution they came up with.

> They drew a distinction between "foreground memory" and "background memory." The computer was set up to have 64K of foreground memory. The locations in foreground memory are numbered from 0 to 65535.
>
> They set up the computer so that 64K of the memory units would occupy foreground memory and 20K would occupy background memory. Whatever occupied foreground memory would be completely accessible and usable. Whatever occupied background memory would be inaccessible or accessible only in a limited way.

Some electronic "switches" were built into the computer, to make it possible to switch portions of memory from foreground to background, or background to foreground. These switches would operate at electronic speeds. They would be activated by POKEs. This meant that something could be moved from foreground to background, or vice versa, in tiny fractions of a second. And this would make it possible for a clever programmer to use all or almost all of the 84K, by cleverly moving portions of memory from foreground to background, and vice versa, very quickly.

The normal arrangement for foreground and background memory would be as follows.

FOREGROUND		BACKGROUND
0 - 40959	RAM	
40960 - 49151	BASIC Interpreter ROM	Locations 40960 - 49151 from RAM
49152 - 53247	RAM	
53248 - 57343	RAM	Character ROM
57344 - 65535	KERNAL ROM	Locations 57344 - 65535 from RAM

You can calculate from this chart that 64K is in the foreground and a total of 20K is in the background. Relegated to the background are a total of 16K of RAM and the Character ROM.

There are two ways in which background memory can be used. First, by making the correct POKEs, it is possible to switch a portion of background memory into foreground. That portion of memory then becomes fully usable. This is what is done with Character ROM. Normally, Character ROM is in the background. However, the computer frequently will switch it into the foreground for a split second, to obtain information on character shapes. The computer then switches it into the background again.

Second, the engineers decided that if a program attempts to record information on some ROM memory that is in the foreground, the computer will record this information in the RAM memory that is "behind" (in the background). Background RAM cannot be PEEKed at (until it is brought into the foreground), but it can be POKEd to.

The techniques for switching foreground and background are tricky, and they will not be covered in detail in this book. One of the problems is that if you are using BASIC and switch either the BASIC Interpreter ROM or the KERNAL ROM to background, the computer may be unable to execute your BASIC commands. If you are curious about how to play with foreground and background memory, study the program COPYSHAPE, in Appendix E, which shows how to switch Character ROM into foreground. Also, read Appendix A to learn more about advanced topics in snooping.

Chapter VIII

PROGRAMS

When you run a program in BASIC, a copy of that program is held in the computer's memory. However, the version in memory looks quite different from what you originally typed into the computer. In this chapter, we are going to find out what a program looks like in memory, and why it looks that way.

Clear your computer's memory completely. To do this, turn the computer off, wait ten seconds, and then turn it back on. This is to assure that nothing that you were doing previously will interfere with our experiments in this chapter.

Now type the program SNOOP into the computer (or, if you have a copy of it on cassette or disk, LOAD it).

RUN the program. When it asks you where to START AT, type 2048 and press RETURN. You will recall from our MAP in the last chapter that location 2048 is the normal starting point of the area reserved for BASIC programs and variables. On your screen you will see the following display:

```
START = 2048

                                       S
  0   13    8    1    0  143   32   83
  N    O    O    P
 78   79   79   80    0   32    8  100
                  "    S    T    A    R
  0  133   32   34   83   84   65   82
  T         A    T    "    ;    B
 84   32   65   84   34   59   66    0
  9                                  (
 57    8  200    0  153   32  199   40
  1    4    7    )    ;    "    S    T
 49   52   55   41   59   34   83   84
  A    R    T    =         "    ;    B
 65   82   84   61   32   34   59   66
       I         ,                   L
  0   73    8   44    1  129   32   76
  I         1                   1    0
 73  178   49   32  164   32   49   48
       Y                             I
  0   89    8  144    1  129   32   73
```

This is the first part of your program SNOOP as it is held in memory. Parts of the program are easy to recognize: the REM statement at the beginning, the variable

names, the words that are displayed on the screen. The other features of your program, such as the line numbers and the command names, are not so easy to recognize. Let's go over part of this screen byte by byte, and see how it relates to what's in the program SNOOP.

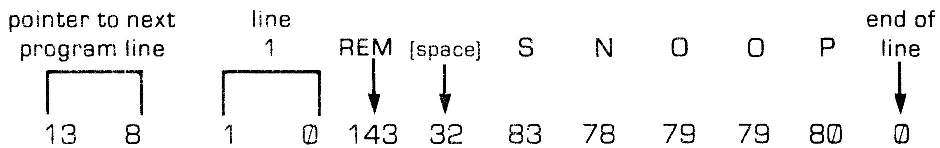
The first line in SNOOP is

```
1 REM SNOOP
```

In memory this shows up as

```
13  8  1  0  143  32  83  78  79  79  80  0
```

The following diagram and comments will explain what each of the codes means:



The first two codes

```
13    8
```

are a "pointer" to the next line in the program. A "pointer" tells you the location of something important. This pointer tells where the next line of the program begins. This is calculated as follows: The location of the beginning of the next program line is

```
13  +  256 * 8
```

which is equal to 2061 (i.e., the next line of the program begins at location 2061). The purpose of pointers like these is to help the computer find information quickly. We will find out more about pointers later.

The next two codes

```
1     0
```

give the line number. It is calculated as follows

```
    1 + 256 * 0
=   1 + 0
=   1
```

So the line number is 1.

The code

```
143
```

stands for REM. All of the command words in BASIC are abbreviated in memory as numbers from 128 to 255. Here are a few examples of the codes for BASIC command words:

FOR	129
NEXT	130
INPUT	133
REM	143
POKE	151
PRINT	160

A more complete chart is given at the end of this chapter.

The code

    32

is the ASCII code for a space. This indicates the space between REM and SNOOP in the program line.

The next five codes

    483   78   79   79   80

are the ASCII codes for the letters S N O O P.

The final code

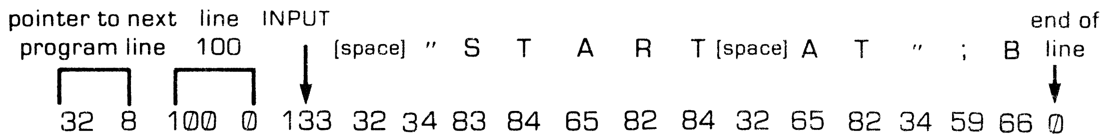
    0

marks the end of the program line.

Let's figure out the next line of SNOOP.

```
100 INPUT "START AT";B
```

In memory this is represented by



The comments will explain what each of the codes means.

The first two bytes

    32   8

are a pointer to the program line after this one. The location it points to is

```
  32 + 8 * 256
= 32 + 2048
= 2080.
```

i.e., the next program line begins at memory location 2080.

The next two bytes

```
100    0
```

give the line number. It is calculated as

```
  100 + 0 * 256
= 100 + 0
= 100.
```

i.e., the line number is 100.

The code

```
133
```

stands for INPUT. Whenever an INPUT command occurs in a BASIC program, it is abbreviated in memory as 133.

The two codes

```
32    34
```

stand for the blank space and quotation mark between INPUT and START AT.

The group of codes

```
83 84 65 82 84 32 65 84
```

are the ASCII codes for

```
START AT
```

The two codes

```
34 59
```

stand for the right quotation mark and the semicolon.

The code

```
66
```

is the ASCII code for the the letter B. This is how the variable B is represented in memory.

The final code

```
0
```

marks the end of the program line.

If you would like to learn more about how BASIC programs are stored in memory, use SNOOP to look at examples of BASIC commands as they are stored in memory. Also, we suggest that you experiment with using POKE commands to alter a program that is in memory. By trying out various POKEs you can find out in detail what the purpose is of every code in memory. As you are exploring, there are a couple of "fine points" that you should be aware of:

> Normally, the BASIC program area starts at location 2048. (In the next chapter, we will find out how that starting point can be relocated.)The first byte of this area is always set to 0. Your program always begins at the second byte, i.e., location 2049. If the first two bytes of a program line are
>
> 0 0
>
> then this really isn't a program line at all. Instead, it is a signal that the end of the program has been reached. This is how the computer marks the end of a program. If you are examining memory, and arrive at a program line which ends with 0 0, you are done; you have reached the end.

We are now going to illustrate the power of the POKE with a couple of examples of programs that are transformed by a few POKEs.

First, we will try out an example of a program that rewrites itself.

ENTER THIS PROGRAM EXACTLY AS WRITTEN:

```
1 REM REWRITE
100 PRINT "JOHN HAD"
110 PRINT "GREAT BIG"
120 PRINT "WATERPROOF"
130 PRINT "BOOTS ON"
200 FOR I = 2048 TO 2150
210 IF PEEK(I) = 153 THEN POKE I,143
220 NEXT I
```

Make sure that you have entered this program exactly as it appears in this book. Do not leave out the REM statement or revise any of the program lines from how they are shown in this book, or the program may not work.

When you run this program, it will do two things:

> First, it will display a message on the screen.
>
> Second, it will rewrite lines 110 - 140. After you have run the program, we will run it again, and you will discover that it has changed.

RUN THE PROGRAM.

On your screen you will see a message:

```
JOHN HAD
GREAT BIG
WATERPROOF
BOOTS ON
```

The program has also rewritten itself. To prove this, RUN THE PROGRAM AGAIN.

This time, no message will appear on the screen. What has happened?

To find out has happened, LIST the program. It will look like this:

```
1 REM REWRITE
100 REM "JOHN HAD"
110 REM "GREAT BIG"
120 REM "WATERPROOF"
130 REM "BOOTS ON"
200 FOR I = 2048 TO 2150
210 IF PEEK(I) = 153 THEN POKE I,143
220 NEXT I
```

The four PRINT commands that were in lines 100 - 130 have been changed to REM statements. That is why when you ran the program the second time, nothing appeared on the screen. REWRITE has rewritten itself!

How the Program Works:

Lines 200 - 220 scan the area where the program is, and look for any occurrences of the code 153. This is the computer's code for the PRINT command. Whenever it finds one of these codes, line 210

```
210 IF PEEK(I) = 153 THEN POKE I,143
```

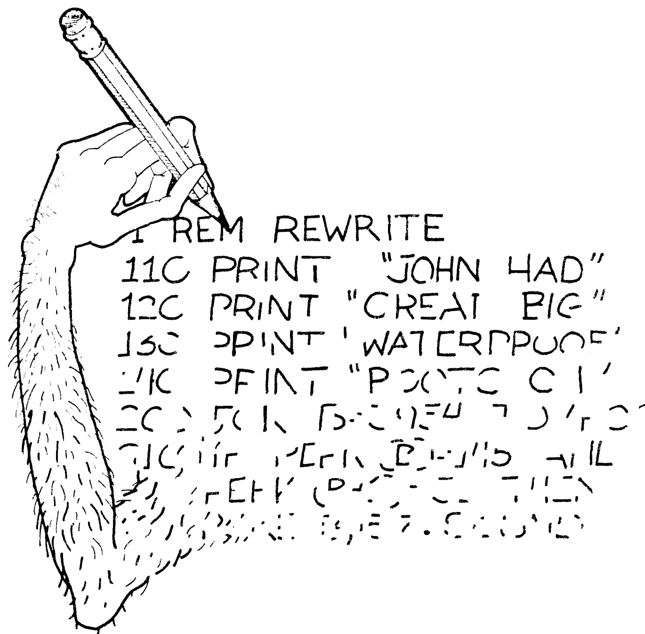
changes it to 143, which is the code for REM. The FOR-NEXT loop ensures that all PRINTs are changed to REMs.

This next program uses POKEs to make itself "invisible."

ENTER THIS PROGRAM EXACTLY AS WRITTEN:

```
1 REM INVIS
100 PRINT "HI MOM!"
110 PRINT "HERE I AM!!"
200 POKE 2050,3
```

When you run this program, it will display a message on the screen. It will also perform some "magic" on itself so that it cannot be listed, except for the first line.



Run the program. On your screen you will see

```
HI MOM!
HERE I AM!!
```

Now LIST the program. All that will appear is

```
1 REM INVIS
```

Where is the rest of the program? Well, it was not erased from memory. To prove this, type RUN and press RETURN. Once again the message

```
HI MOM!
HERE I AM!!
```

will appear. This proves that the program is still operating. You just can't see it anymore!

How INVIS Works:

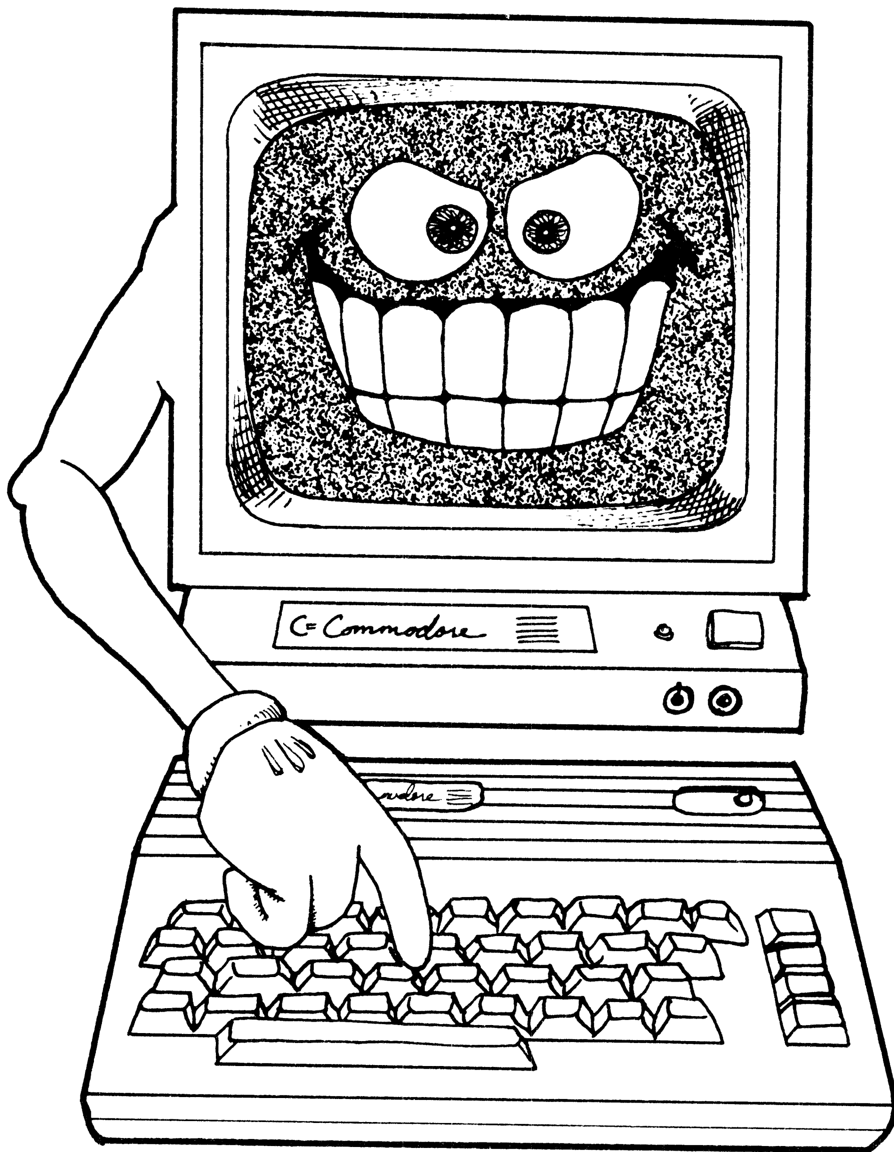
The trick to the program is in line 200

```
200 POKE 2050,3
```

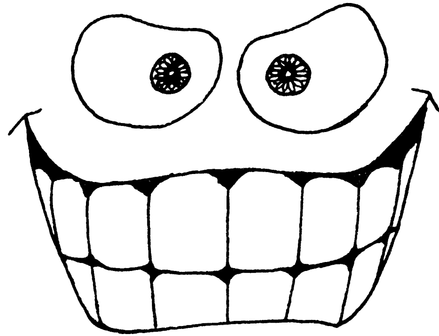
This command POKEs a new value into location 2050, which is the second byte of your program. As we discovered earlier, the first two bytes of the program contain a "pointer," which tells the computer where the next line of the program begins. The computer needs this information when you want to LIST the program. The pointer information helps the computer to locate each line of your program in memory.

The POKE in line 200 altered some of this pointer information. The effect of this was to confuse the computer as to where the second line of the program is. We altered the pointer so that when the computer looked for the second line of the program, it was directed to an area of memory that has nothing but 0s. When the computer saw those 0s, it believed that it had arrived at the end of the program. So the computer quit LISTing the program after the first line.

Now, your original program has not been erased at all. Each of the commands is still in memory, just as it was. And because of this, the computer can still execute the program. It happens that the computer does not need much pointer information to execute a program. So, although the altered pointer affects the ability of the computer to LIST the program, it does not affect its ability to execute the program.



If you have a disk drive or cassette recorder, you can still SAVE, and reLOAD the program. When you reLOAD it, you will find that the entire program will LIST again. This is because the computer will automatically set the pointers in proper order each time the program is LOADed. However, if you run the program again, it will once again become invisible.



To conclude this chapter, we present a chart which shows many of the codes used for abbreviating BASIC commands and functions. In the next chapter, we will find out more about pointers, and why the computer likes to use them.

BASIC KEYWORDS

CODE	BASIC KEYWORD
128	END
129	FOR
130	NEXT
131	DATA
132	INPUT#
133	INPUT
134	DIM
135	READ
136	LET
137	GOTO
138	RUN
139	IF
140	RESTORE
141	GOSUB
142	RETURN
143	REM
144	STOP
145	ON
146	WAIT
147	LOAD
148	SAVE
149	VERIFY
150	DEF
151	POKE
152	PRINT#
153	PRINT
154	CONT
155	LIST
156	CLR
157	CMD
158	SYS
159	OPEN
160	CLOSE
161	GET
162	NEW
163	TAB(
164	TO
165	FN
166	SPC(
167	THEN
168	NOT
169	STEP
170	+
171	-
172	.
173	/
174	^

CODE	BASIC KEYWORD
175	AND
176	OR
177	>
178	=
179	<
180	SGN
181	INT
182	ABS
183	USR
184	FRE
185	POS
186	SQR
187	RND
188	LOG
189	EXP
190	COS
191	SIN
192	TAN
193	ATN
194	PEEK
195	LEN
196	STR$
196	VAL
198	ASC
199	CHR$
200	LEFT$
201	RIGHT$
202	MID$

Chapter IX

POINTERS

In the last chapter, we found out that when a program is held in memory, it contains many "pointers." A "pointer" is a piece of information which tells the computer where something is located. At the beginning of each line of the program, there is a pointer which tells the computer where in memory the next program line begins.

This is only one of many situations where the computer uses pointers. Memory is filled with thousands of pointers. In this chapter, we will find out why there are so many pointers, and what they are used for. We will see that there are two main reasons why the computer uses pointers:

First, pointers help the computer to find information quickly.

Second, pointers help the computer to arrange information more efficiently.

To begin, let's find out what the pointers in program text are used for.

Each line of the program begins with a pointer. That pointer tells the computer where the next program line begins. How is this information useful to the computer? Well, when a program is running, and the computer encounters a GOTO command, the pointers help it find the line of the program that it needs. Instead of scanning the entire program, it merely follows the chain of pointers within the program. Each time it arrives at a new line in the program, it looks at just the Line Number. If that's not the line number which is needed, the computer follows the pointer to the next line of the program. Thanks to pointers, the computer can find the beginning of each line very quickly — it does not have to get bogged down "reading" the entire text of each line. This helps the computer find the line number it needs very quickly.

Let's find out what else pointers are used for. The following program will enable you to look at several very important pointers in memory. Clear your computer completely by turning it off, waiting 10 seconds, and then turning it back on. Then

RUN THIS PROGRAM:

```
1 REM POINTERS
100 P=PEEK(43) + 256*PEEK(44)
110 PRINT "BASIC PGM STARTS AT";P
200 P=PEEK(45) + 256*PEEK(46)
210 PRINT "VARIABLES START AT";P
300 P=PEEK(55) + 256*PEEK(56)
310 PRINT "BASIC REGION ENDS AT"; P
```

On your screen, you will see some information like this:

```
BASIC PGM STARTS AT 2048
VARIABLES START AT 2232
BASIC REGION ENDS AT 40960
```

The computer is telling you three important locations in memory.

The first line on the screen tells you where the region for BASIC programs begins. This is already familiar to you. Normally this region begins at location 2048.

The second line tells you where the region for variables begins. It always begins just after the end of the text of your BASIC program. So the exact location will depend on the length of the program currently in memory. In our illustration, it begins at location 2232. The third line tells you where the region for BASIC ends. In this illustration, the end is location 40960. So, the computer has reserved the region 2048 - 40960 for BASIC program and variables. The computer got this information from three pointers in memory.

The pointer in locations 43 - 44 tells where the region for BASIC programs begins.

The pointer in locations 45 - 46 tells where the region for BASIC variables begins.

The pointer in locations 55 - 56 tells where the region for BASIC ends.

In each case, the pointer is stored in two bytes of memory. The value of each pointer is calculated as follows:

VALUE OF FIRST BYTE + (256 * VALUE OF SECOND BYTE)

So, for instance, if

VALUE OF THE FIRST BYTE = 184
VALUE OF THE SECOND BYTE = 8

then

VALUE OF THE POINTER = 184 + (256 * 8)
= 184 + 2048
= 2232

This formula is typical of how most pointer values are calculated. The formula seems tedious to calculate, but actually it is a very easy kind of formula for the computer to calculate. The mathematical reasons for this are beyond the scope of this book, but the main idea is that 256 in Base Two is

100000000

and this is an easy number for the computer to multiply with. It's easy for roughly the same reason that it's easy for a human being to multiply a number by one thousand. (What's easy for a computer is not necessarily easy for humans, and vice versa!)

If you would like to learn more about why computers find pointer values like this easy to calculate, see the reading list in Chapter 16, CONCLUSION.

The pointers we have just looked at are useful to the computer for two reasons.

First, they help the computer find important regions of memory quickly. For instance, if the computer needs to scan the variables in memory, it can look up the the pointer in locations 45 - 46 to find out where the region for variables begins.

Second, the pointers make it very easy for the computer to rearrange memory. Suppose that you add some lines to your program. This means that the region for variables must be shifted farther out. The computer can shift that region merely by changing the value of the pointer in locations 45 - 46.

We call pointers like these "master pointers." Master pointers tell the computer where important areas of memory are located.

There is a master pointer which tells the computer where screen memory is. If the value of that pointer is changed, screen memory is shifted to a different location.

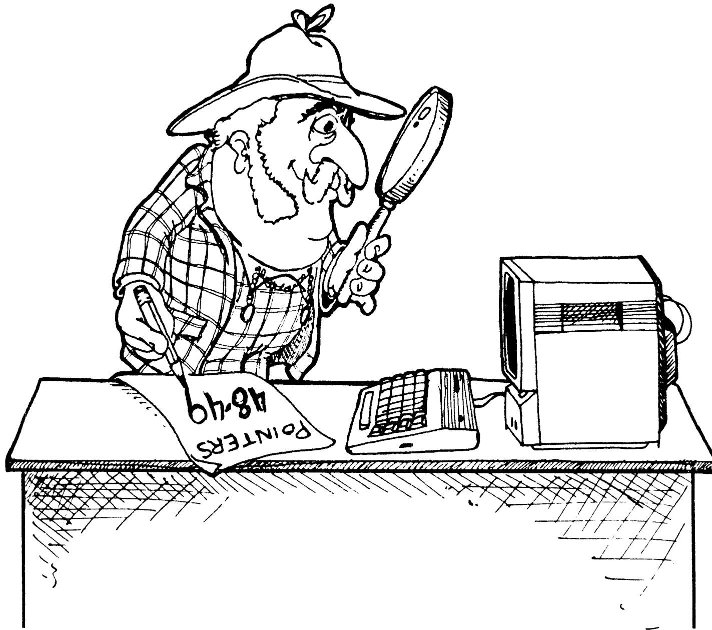
There is a master pointer which tells the computer where the Shape Table is. If the value of that pointer is changed, the computer will use another region of memory for its shape table.

There are two master pointers which tell the computer where there is additional space for variables.

There are a large number of master pointers which tell the computer where various machine language routines are located in memory.

When you turn on the computer, the master pointers are set to standard values. But it is possible to modify them. For instance, if someone writes a program for the COMMODORE 64 which does not use BASIC, the program could alter some of the master pointers for BASIC and thus obtain more space to use in memory.

Pointers are easy for the computer to work with, but human beings generally find them hard to get used to. To help you get adjusted, it is useful to look at some simple examples. The following examples use systems of pointers that are similar to what you find in the computer, but they are presented in a form that is a little more pleasant.



Here is a sample of data that uses pointers:

<BEG>THIS IS THE STORY <051><BEG>I DREAMT OF <082> OF LITTLE RED RIDING HOOD <094>FLYING <113>THREE TURKEYS <128>TO ALASKA <144>AND BB WOLF<END>IN A HELICOPTER<END>

The data contains two different messages. The first is:

> THIS IS THE STORY OF LITTLE RED RIDING HOOD THREE TURKEYS AND BB WOLF

The second is

> I DREAMT OF FLYING TO ALASKA IN A HELICOPTER

In one chunk of data, there are two different messages weaved together by means of pointers. Let's find out how this was done.

The data is 163 characters long altogether. Along with the actual messages, there are some markers that tell you how the information is organized.

> The <BEG> markers.
>
> The <END> markers.
>
> The pointers — markers consisting of brackets with a number in between.

The <BEG> markers mark the beginning of a message. The <END> markers mark the end of a message. The pointers indicate that a message leaves off in a certain place and picks up at a later location. For instance, in the first line, we see

<BEG>THIS IS THE STORY <Ø51>

The pointer <051> says that this message leaves off here and continues at position 51. Position 51 is at the beginning of the second line

OF LITTLE RED RIDING HOOD <Ø94>

This piece of the message ends with another pointer, <094>. This means that the message leaves off here and continues at position 94, which is

THREE TURKEYS <128>

That piece of the message ends with a pointer, <128>. This means that the message leaves off here and continues at position 128, which is

AND BB WOLF<END>

This piece of the message ends with <END>, which signifies the end of the message.

So, what we have here is one message which has been snipped into four separate parts. The parts are connected together by means of pointers.

THIS IS THE STORY

points to

OF LITTLE RED RIDING HOOD

which points to

THREE TURKEYS

which points to

AND BB WOLF

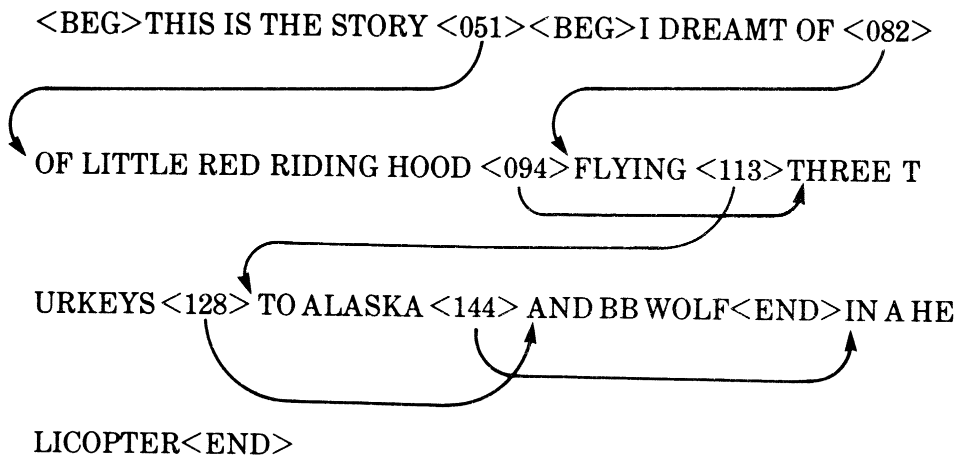
The second message

I DREAMT OF FLYING TO ALASKA IN A HELICOPTER

has been snipped into four pieces, which are connected by means of points. The four pieces are

I DREAMT OF
FLYING
TO ALASKA
IN A HELICOPTER

These four pieces are connected by means of pointers. The following diagram summarizes how the pointers connect everything together.



What we have here, then, are two messages, each of which have been snipped into little pieces and then weaved together. Pointers are used to keep the right pieces of each message connected. The <BEG> and <END> markers are used to show the start and end of each message.

One of the advantages of pointers is that they help you use space efficiently. Suppose you have a message with 10,000 characters, and you do not have a single free area in memory big enough for the entire message. However, you do have 10 smaller areas in various parts of memory that together provide enough space. Well, you can snip the message into 10 smaller pieces, store each piece in a different part of memory, and use pointers to connect the 10 pieces together. Another advantage of pointers is in dealing with changes in a message. Let's suppose that we want to remove the phrase

THREE TURKEYS

from the Little Red Riding Hood message. Here's how to do it:

<BEG>THIS IS THE STORY <051><BEG>I DREAMT OF <082> OF LITTLE RED RIDING HOOD <128>FLYING <113>THREE TURKEYS <128>TO ALASKA <144>AND BB WOLF<END>IN A HELICOPTER<END>

All we did was change the pointer in the second line, after the word HOOD. The pointer used to be <094>, which pointed to the phrase

THREE TURKEYS

We changed it to <128>, so that it now points to

AND BB WOLF

In other words, we adjusted a pointer so that it points past the phrase we wanted to omit. Now when you trace the pointers, you will get the following message

THIS IS THE STORY OF LITTLE RED RIDING HOOD AND BB WOLF

The phrase THREE TURKEYS is still in the data, but it has been deleted from the message, since the pointers skip past it.

You can also use pointers to quickly insert new data in a message. See if you can figure out what happens in this example:

<BEG>THIS IS THE STORY <051><BEG>I DREAMT OF <082> OF LITTLE RED RIDING HOOD <164>FLYING <113>THREE TURKEYS <128>TO ALASKA <144>AND BB WOLF<END>IN A HELICOPTER<END>A NEAT OLD GRANDMA <128>

The Little Red Riding Hood message has now become

THIS IS THE STORY OF LITTLE RED RIDING HOOD A NEAT OLD GRANDMA AND BB WOLF

Here's how we got this result:

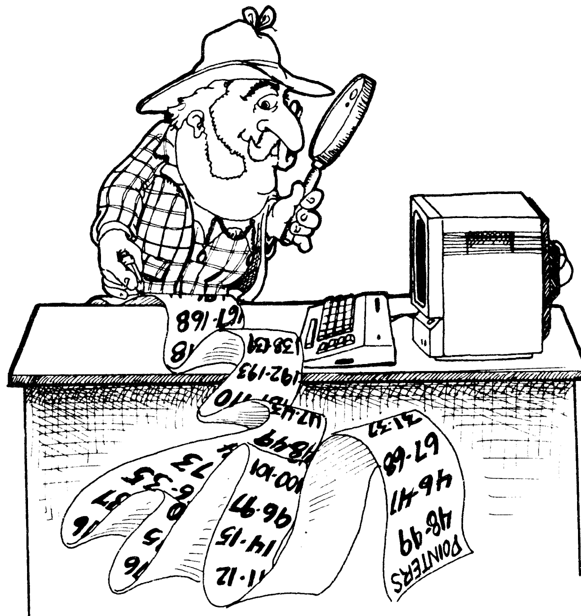
At the end of the data, we added the phrase A NEAT OLD GRANDMA.

We adjusted the pointer after HOOD to <164>, which points at the beginning of the new phrase A NEAT OLD GRANDMA.

We set the pointer at the end of the new phrase to <128>, which points to the beginning of the phrase AND BB WOLF.

The convenience of this is that we did not have to rewrite the old part of the message. This is a great advantage when you are dealing with long messages.

The pointers and markers that you will find in the COMMODORE 64 are similar to the ones in these examples. However, the markers are not usually as easy to recognize and interpret as it was here. That's because what is easy for the computer is not necessarily easy for human beings.



Chapter X

VARIABLES

In an earlier chapter we saw what a program looks like in memory. Programs use variables. In this chapter we will find out what variables look like in memory.

KINDS OF VARIABLES

Your COMMODORE 64 allows you to use several kinds of variables for various purposes.

Floating point variables are probably most familiar to you. Examples of floating point variables are:

A, Q, A7, WZ, I

A floating point variable may hold any kind of number — positive or negative, whole number or decimal. Examples of allowable values are:

5, 0, 9999888, -197.325, 59.32, 177324E+13

The last number

177324E+13

means 177324 with thirteen 0s tacked on to the end, i.e.,

1773240000000000000

Notation like this is known as "exponential notation." Exponential notation is a convenient way of writing very large or very small numbers. For instance,

1.7332E-10

means

.00000000017332

To find out more about notation like this, see your *Commodore 64 User's Guide.*

Integer variables may have only integers (whole numbers) as values. Integer variables always end with a %. Examples of integer variables are:

A%, Q%, A7%, WZ%, I%

Examples of allowable values are:

5, 0, 17993, -1533

String variables may hold strings of symbols of any kind. String variables always end with a $. Examples of string variables are:

A$, Q$, A7$, WZ$, I$

Examples of allowable values are

"MRFL@X", "123998", "Z198&().XAP", "COMBAT BOOTHS"

An **array** is a group of variables which is defined in one fell swoop. An array is set up by means of a DIM command. For instance,

DIM QJ$(15)

tells the computer to set up 16 different variables, whose names are:

QJ$(0), QJ$(1), QJ$(2), QJ$(3), ... , QJ$(14), QJ$(15)

Each of these 16 variables is a separate string variable. Each one may have its own value.

You may use the DIM command to define arrays of floating point variables, arrays of integer variables, or arrays of string variables.

For additional information on variables, see your *Commodore 64 User's Guide.*

The following program will allow you to INPUT a number into a variable called AB%. Then the program will display how the variable looks in memory.

ENTER THIS PROGRAM:

```
1REM VARLOOK1
100 INPUT "VALUE FOR AB%";AB%
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT:GOTO 100
```

When you run this program, it will ask you to enter a number. That number will be stored in the variable AB%. Then the program will display the part of memory in which AB% is stored.

RUN THE PROGRAM

The program will ask you VALUE FOR AB% ?. Type 259 and press RETURN. The following information will appear on the screen:

```
193 194  1  3  0  0  0
```

This is what AB% looks like in memory.

The following diagram will explain what each of the codes means:

The first two codes

```
193 194
```

are the name of the variable. 193 stands for A, and 194 stands for B. Here is how the codes 193 and 194 are derived:

The ASCII code for A is	65
	+ 128
	193
The ASCII code for B is	66
	+ 128
	194

The computer adds 128 to the ASCII codes to signify that this is an integer variable. An integer variable is a variable that ends with a %. An integer variable is allowed to have only integer values, such as 1, 2, 3, -1, -2, -3, or 0.

The next two codes

```
1  3
```

represent the value of the variable. If the variable is positive, the value is calculated as

(256 * FIRST CODE) + (SECOND CODE)

In this example, the value works out to

(256 * 1) + 3
= 256 + 3
= 259

The last three codes

```
0 0 0
```

are unused.

Let's try a few more examples. Try entering each of these numbers:

```
 260
 519
1097
   0
```

You will get the following information on the screen:

```
VALUE  FOR   AB%?  260
193    194   1     4     0     0     0

VALUE  FOR   AB%?  519
193    194   2     7     0     0     0

VALUE  FOR   AB%?  1097
193    194   4     73    0     0     0

VALUE  FOR   AB%?  0
193    194   0     0     0     0     0
```

In each case the variable AB% looks the same, except for the third and fourth bytes, which show the value in the variable.

Now let's find out what AB% looks like when it is holding a negative value. Try the following examples:

```
-1
-2
-3
```

You will get the following information on the screen:

```
VALUE  FOR   AB%?  -1
193    194   255   255   0     0     0

VALUE  FOR   AB%?  -2
193    194   255   254   0     0     0

VALUE  FOR   AB%?  -3
193    194   255   253   0     0     0
```

If you try a few more examples using negative numbers, the pattern will be clear. This method of representing negative numbers seems strange, but it is quite common with computers. Computers can calculate very efficiently with negative numbers when using this representation. If you would like to learn more about it, it is known as "Two's complement notation," and it is discussed in most books on machine language programming or beginning computer science. See the suggested reading in Chapter 16, CONCLUSION.

How VARLOOK1 Works:

Line 100

```
100 INPUT "VALUE FOR AB%";AB%
```

allows the user to INPUT a number, which is then stored in the variable AB%. The % signifies that the variable is an integer variable.

Line 200

```
200 B=PEEK(45) + 256*PEEK(46)
```

calculates the beginning of the region where variables are stored. Locations 45 - 46 hold the pointer which points to the beginning of this region. B is the value of the pointer.

Variables (with the exception of array variables) are stored in this region in the order in which they first appear in the program. AB% is the first variable to appear in our program. So it will occur first in the region.

Each variable uses seven bytes in memory. Lines 300 - 320

```
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
```

display the values in the first seven bytes of the region. This will be the information on the variable AB%.

Line 400

```
400 PRINT:PRINT:GOTO 100
```

skips two lines on the screen and takes you back to line 100, to INPUT another value.

Different variable types are stored in memory in different ways. We have seen how integer variables are stored. Now let's find out how floating point variables are

stored. A floating point variable is simply the ordinary kind of variable used to store numbers of all kinds, including decimals. Examples of floating point variables are:

A, B, I, Z, A7, Q9, PQ, AB.

To see how floating point variables are stored in memory, we can use VARLOOK1, with just two small changes in the program. We will change the program so that it uses the variable AB instead of AB%. AB is a floating point variable, since it does not have a % at the end.

CHANGE VARLOOK1 TO LOOK LIKE THIS:

```
1REM VARLOOK2
100 INPUT "VALUE FOR AB";AB
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT:GOTO 100
```

This version is the same as the old version, except that we removed the % in line 100.

Now run the program. When it asks you for a number, type 0 and press RETURN. On your screen you will see the following codes:

```
VALUE FOR AB?
0 65 66 0 0 0 0 0
```

This is similar to what you saw earlier with AB%. Here are the differences:

The first two codes name the variable as before, except that the value of 128 is no longer added. If the value of 128 is not added, this tells the computer that this is a floating point variable.

65 66

means floating point, and

193 194

means integer.

The remaining five codes

0 0 0 0 0

represent the value of the variable.

Let's see how some other values are stored in AB. Try each of these values:

```
1
2
3
4
8
16
 .1
 .2
```

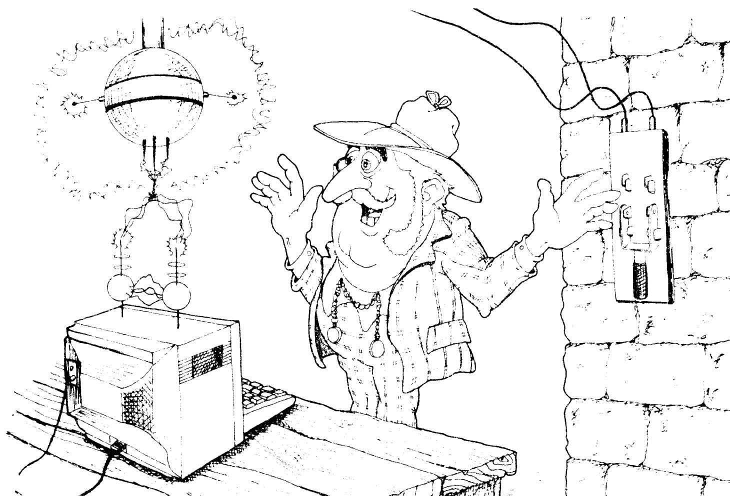
You will get the following information on your screen:

```
VALUE  FOR   AB?   1
65     66    129   0     0     0     0
VALUE  FOR   AB?   2
65     66    130   0     0     0     0
VALUE  FOR   AB?   3
65     66    130   64    0     0     0
VALUE  FOR   AB?   4
65     66    131   0     0     0     0
VALUE  FOR   AB?   8
65     66    132   0     0     0     0
VALUE  FOR   AB?   16
65     66    133   0     0     0     0
VALUE  FOR   AB?   .1
65     66    125   76    204   204   205
VALUE  FOR   AB?   .2
65     66    126   76    204   204   205
```

In each case, the last five codes on a line represent the value that is in AB. As you can see, the pattern is strange! The rules which are used for representing values are too complicated to be discussed in this book.

This brings us to the SECOND RULE OF COMPUTER SNOOPING:

You can learn a lot about how your computer works by running experiments.



This rule is important for several reasons.

First, you will discover, if you haven't already, that good technical documentation is hard to find (for the COMMODORE 64 or almost any other computer). It's difficult to find documentation that is easy to understand. It's difficult to find documentation that explains all technical details. Also, you will discover that documentation can be inaccurate or incomplete. However, it's often easy to figure out what you need to know by running some experiments.

Second, it reminds us that computers have a low order of intelligence. They tend to do their work in tedious, simple-minded ways. This makes it possible to understand almost everything they do, if you have enough patience.

Third, experimenting is fun! And, as long as you do not open your computer, your experiments can cause no harm.

Now, returning to the problem at hand, here are some suggested numbers to try out, to figure out how numbers are represented in floating point variables. Try out these numbers in the order listed:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
-1, -2, -4, -8, -16, -32, -64
-1, -2, -3, -4, -5, -6, -7, -8
1, .5, .25, .125, .0625, .03125
-1, -.5, -.25, -.125, -.0625, -.03125
1, 10, 100, 1000, 10000, 100000, 1000000, 10000000

Another way to approach the problem is to set up a variable in memory, find its location, POKE some values into it, and then see what happens. If you would like to do this, see the program VARPOKE in Appendix A, EXTRA TOPICS.

Now let's find out what a string variable looks like. A string variable is simply any variable that ends with a $; it holds any kind of information, numeric or otherwise.

The COMMODORE 64 uses a fairly complicated scheme to organize string variables in memory. This scheme enables the COMMODORE 64 to accomodate strings of various lengths, anywhere from 0 to 255 characters. The method involves splitting the variable into two parts: the SYSTEM BLOCK and the DATA BLOCK.

> The SYSTEM BLOCK holds the name of the variable, the number of characters of data in the string, and a pointer to the DATA BLOCK. The SYSTEM BLOCK is held in the same part of memory as floating point variables.
>
> The DATA BLOCK holds the string data in the upper region of BASIC memory. (This is normally location 40960.)

The following program will allow you to see both parts of string variable AB$.

ENTER THIS PROGRAM:

```
1REM VARLOOK3
100 INPUT "VALUE FOR AB$";AB$
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT
500 C=PEEK(B+2)
510 B1= PEEK(B+3) + 256*PEEK(B+4)
520 PRINT "DATA BLOCK STARTS AT";B1
600 FOR I+B1 TO B1+C-1
610 PRINT PEEK(I);
620 NEXT I
700 PRINT:PRINT:PRINT:GOTO 100
```

RUN the program. When it asks you

```
VALUE FOR AB$
```

type ABCDEF and press RETURN. Information like this will appear on the screen.

```
VALUE FOR AB$? ABCDE
65 194 6 250 159 0  0

DATA BLOCK STARTS AT 40954
65 66 67 68 69 70
```

The first line of numbers is the SYSTEM BLOCK for AB$. The first two codes

    65 194

give the name of the variable. This time, 128 has been added to the second code only. This is the computer's way of indicating that this is a string variable, rather than an integer variable or a floating point variable.

The next code

    6

is the number of characters in the variable. This says that AB$ is six characters long.

The next two numbers

    250 159

are a pointer to the start of the DATA BLOCK. In this illustration, the DATA BLOCK starts at

```
  250 + 256*159
= 250 + 40704
= 40954
```

The last two bytes in the SYSTEM BLOCK

    0 0

are not used.

The last line

    65 66 67 68 69 70

are the values in the DATA BLOCK (locations 40954 - 40959 in this example). These are the ASCII codes for the data in the variable.

Let's try entering new data into AB$. Right now the computer is asking

    VALUE FOR AB$?

type GHI and press RETURN. Information like this will appear on the screen.

```
VALUE  FOR    AB$?   GHI
65     194    6      247    159    0    0

DATA   BLOCK  STARTSAT      40951
71     72     73
```

Notice that this time the DATA BLOCK begins in a lower location in memory than before. The first time, our DATA BLOCK started at location 40954. This time, the DATA BLOCK begins at 40951. If you INPUT data into AB$ again, you will find the DATA BLOCK starting at a lower location again. The computer always tries to store DATA BLOCKs in the highest unused region for BASIC. It "starts at the top and works its way down." At the end of this chapter, we will summarize how variables are stored in memory.

How VARLOOK3 Works:

The first part of the program is similar to VARLOOK1 and VARLOOK2:

```
100 INPUT "VALUE FOR AB$";AB$
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT
```

It allows the user to INPUT some data into AB$. The program then finds the SYSTEM BLOCK for AB$. To do this, it goes through the same calculations as finding the information for an integer variable or floating point variable.

In lines 500 and 510 the program extracts some information from the SYSTEM BLOCK. Line 500

```
500 C=PEEK(B+2)
```

finds the number of characters in the DATA BLOCK.

Line 510

```
510 B1= PEEK(B+3) + 256* PEEK(B+4)
```

it examines the pointer to the DATA BLOCK, and stores the starting location of the DATA BLOCK in the variable B1. In line 520, it PRINTs that location.

```
520 PRINT "DATA BLOCK STARTS AT";B1
```

Lines 600 - 620

```
600 FOR I=B1 TO B1+C-1
610 PRINT PEEK(I);
620 NEXT I
```

display the contents of the DATA BLOCK.

Line 700

```
700 PRINT:PRINT:PRINT:GOTO 100
```

skips a few lines on the screen and then takes you back to line 100.

SUMMARY

There is a region of memory which is reserved for storing variables.

The low end of the region begins just after the end of the program currently in memory. The exact location is given by the pointer in locations 45 - 46.

The high end of the region is given by the pointer in locations 55 - 56. Normally the high end of the region starts at location 40960.

Integer variables, floating point variables, and SYSTEM BLOCKs for string variables are stored in the low end of the region. For each variable, seven bytes of information are used. The meaning of the information depends on the kind of variable.

String variable DATA BLOCKs are stored at the high end of the region. Each DATA BLOCK may have from 1 to 255 bytes, depending on the number of characters in each string variable.

As the computer processes integer variables, floating point variables, and string variables, more and more of the low end of the region is used. Also, as the computer processes string variables, more and more of the high end of the region is used. As the low end and high end are used more and more, they grow toward each other.

If the low end and high end of the variable region "collide," the computer will attempt to rearrange data as neatly as possible, in order to reclaim some usable space. The following program will demonstrate what happens.

RUN THIS PROGRAM:

```
1 REM VARWHERE
100 DIM A$(255)
110 FOR I=TO 255
120 P$= P$+CHR$(I)
130 NEXT I
200 R1=INT(256*RND(1))
210 R2=100 +INT(150*RND(1))
220 A$(R1)=LEFT$(P$,R2)
230 PRINT PEEK(51) +256*PEEK(52)
240 GOTO 200
```

This is a "do-nothing" program which randomly adds and removes data from the various elements of the array A$(). The program tends to add more data than it removes, and gradually A$() will accumulate more and more data. Eventually, memory capacitiy will be exceeded, and you will get an OUT OF MEMORY error.

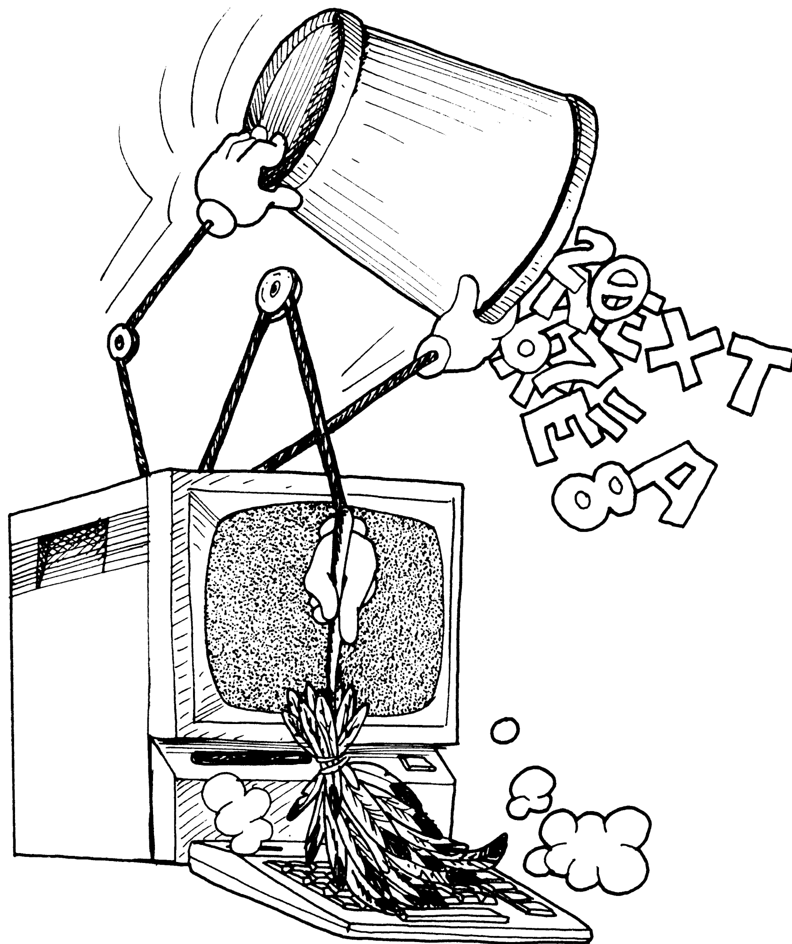
While the program is running, it will give you frequent readouts on the usage of variable memory. Line 230

```
230 PRINT PEEK(51) +256*PEEK(52)
```

displays the value of a pointer which shows the lowest location in memory in use for DATA BLOCKs for string variables. As the program is running, you will see the number go down into the low 3000s, as more and more memory has been consumed. Then the program will seem to stop for a few seconds. During this time, the computer is rearranging data in memory, to free up usable space. After this "housecleaning" is completed, the program will continue. You will then see that higher areas of memory are again being used.

Eventually, the program will get down into the 3000s again. The program will seem to stop again, while another housecleaning takes place. The program will resume, and again you will see that higher areas of memory are being used.

Several more times, you will see the computer get into the 3000s, do a housecleaning, and then resume in higher areas of memory. But as the program progresses, A$() accumulates more and more data. Housecleaning does less and less good. Memory is becoming overcrowded. Finally, you will get the OUT OF MEMORY message. The computer cannot find any more room for variable data.



Chapter XI

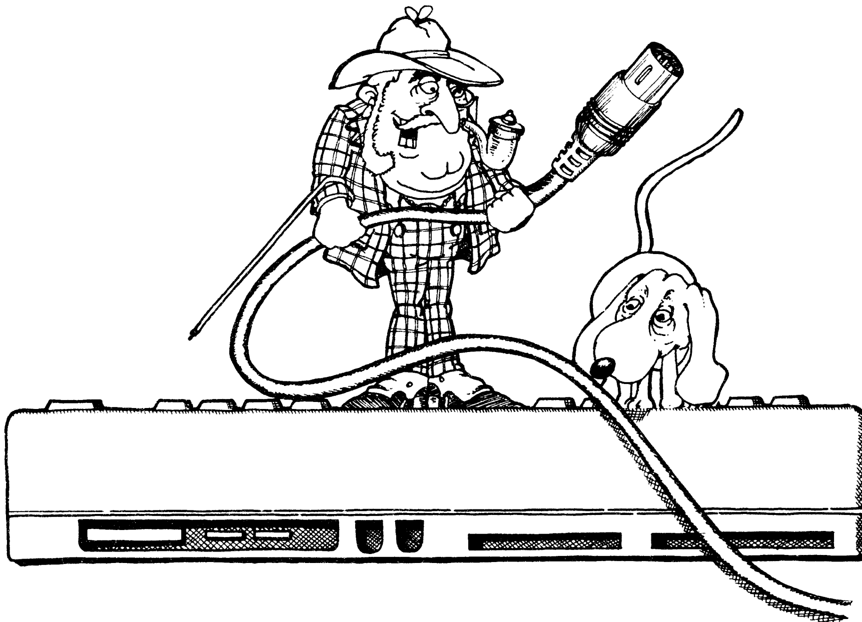
DISK STORAGE — PART 1

In this chapter and the next, we will investigate how disk drives work. Even if you do not own a disk drive right now, these chapters will probably be useful to you for two reasons:

> Disk drives are being used more and more these days, and the cost of a disk drive continues to drop. So there is a good chance that eventually you will use a disk drive with your COMMODORE 64,
>
> Even if you never get a disk drive, the chances are very great that eventually you will use some other computer which does have a disk drive. Most disk drives work in about the same way, regardless of computer. So the information that you obtain here will probably be useful later.

A disk drive is a lot like a cassette tape recorder. In fact, a disk drive and cassette recorder are so similar that you can actually attach a cassette recorder to your computer and use it much like a disk drive.

Commodore's datassette recorder is a simpler mechanism than a disk drive. It is also more familiar to us than a disk drive. For this reason, we will begin by discussing how a datassette recorder works with a computer. Then we will turn our attention to disk drives. In the back and on the side of your computer, there are a number of plugs. You can attach various devices to these plugs — printer, screen, modem, joystick, or datassette recorder. The computer can send and receive information from devices which are attached to these plugs.



If you plug Commodore's datassette recorder into your computer, the computer can send electronic signals out through the plug and into the recorder. The recorder will then record each signal onto its tape. Each signal that is recorded on the tape is equal to one bit of information. For instance, if the computer wanted to store the word WAX in the recorder, here is what would happen:

The computer would transmit 24 electronic signals to the cassette recorder:

0 1 0 1 0 1 1 1 0 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0

The meaning of these signals is as follows:

```
!       W       ! !       A       ! !       X       !
0 1 0 1 0 1 1 1 0 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0
```

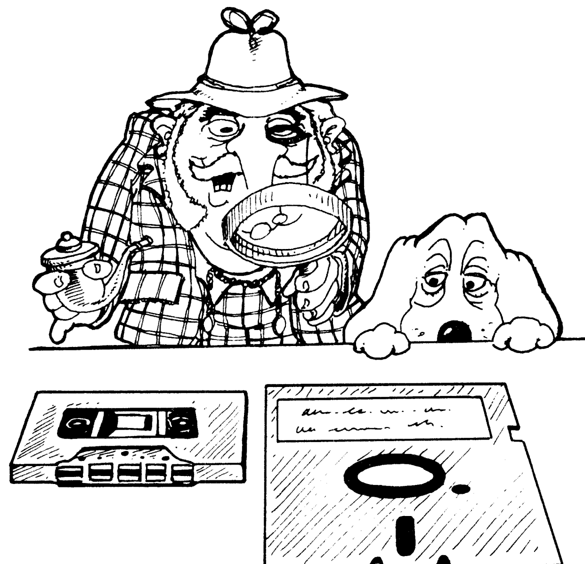
The first eight signals are ASCII for W, the next eight signals are ASCII for A and the last eight are ASCII for X.

These 24 signals are recorded on the tape in the datassette recorder. If you played the cassette back through a speaker, you would actually hear 24 sound impulses — the 1s have a certain kind of sound and the 0s have a certain kind of sound. (The overall sound effect is an uneven "rat-ta-tat".)

In addition to these 24 signals, the computer might send some other signals as well. For instance, it might send some signals to turn the cassette motor on or off, or to mark the start or end of a stream of data.

When you want to retrieve information from a cassette recorder and get it back into the computer, the reverse occurs:

The recorder "reads" a portion of the tape and generates a stream of electronic signals. Each signal stands for a "1" bit or a "0" bit.



These signals go back into the computer.

That is basically how a datassette recorder works with your computer.

Cassette recorders are simple and inexpensive. Unfortunately, they can be annoyingly slow and awkward to use. Suppose that you have a cassette tape with 20 programs on it, and you need the 18th program on the tape. In order to get at that program, you have to wind the tape all the way past the first 17 programs. If you then needed the first program on the tape, you would have to wind it all the way back to the beginning.

You do not have these problems with a disk drive. A disk drive uses the same basic storage concepts as a cassette recorder, but it is cleverly engineered to be much faster and more convenient. The most important features of a disk drive are as follows:

1. A disk drive has a read/write head similar to a cassette recorder. This read/write head is able to read or record electrical impulses on a magnetic oxide surface similar to a cassette tape.

2. Instead of storing and reading information from a tape, a disk drive uses a disk. The surface of a disk is the same kind of material as on a tape. In fact, you can think of a disk as a short, fat tape, whose ends have been joined together.

3. Disk drives use high-precision engineering that allows a large number of electronic signals to be stored in a compact area. This makes it possible to get at a desired piece of information more quickly than with a cassette recorder. No piece of information is very far away.

4. A disk drive rotates the disk at high speed. This enables any particular location on the disk to be accessed more quickly by the read-write head.

5. A disk is divided into "tracks" similar to the tracks on a recording tape. The tracks are numbered 1, 2, 3, ... Within each track are a number of "sectors" of 256 bytes each. (Normally, the first two bytes of each sector are reserved for the computer's internal use and cannot be used by you.) Within each track, the sectors are numbered 0, 1, 2 ... 6. On each disk there is a special area which maps out where information is stored in each sector. With the help of this map, the computer can locate information very quickly.

The technical details behind these concepts are quite complex. The reason for this complexity is

P E R F O R M A N C E

By using advanced techniques for compacting information, organizing it, and mapping it, computer engineers have been able to make disk drives that are much faster and more convenient than cassette recorders.

Now let's find out what information looks like when it is stored on disk.

It is possible to store many different kinds of information on a single disk. On the same disk you can store programs, data files (such as a list of favorite baseball players, or a list of checks which you wrote last year), and text files (such as a letter to your uncle). In order to keep different kinds of information from getting scrambled together, the computer asks you for a file name each time you store information on a disk. On an ordinary floppy diskette, you can have over 100 separate files. Each file may be as short or as long as you like, up to the storage limits of the disk. The computer organizes all files on the disk so that they do not get tangled together.

We are going to create a simple example of a typical data file, and then we will find out what it looks like on disk. The name of the file will be CHAMPS, and it will hold a list of favorite baseball players. The file will hold the following information:

1 RUTH
2 MANTLE
3 KOUFAX

It is a list of three favorite baseball players, ranked from 1 to 3. Of course, this file could be much longer, but to keep the example simple, we will limit the file to three entries. Each entry then will consist of

RANK (1-3) PLAYER'S NAME

The following program will create the file:

RUN THIS PROGRAM:

```
1 REM BASEBALL1
100 OPEN 2,8,2,"@:CHAMPS,S,W"
200 FOR I=1 TO 3
210 READ A$
```

```
220 PRINT#2,I;A$
230 NEXT I
300 CLOSE 2
400 DATA "RUTH","MANTLE","KOUFAX"
```

When you run the program, you will hear the disk drive whirr while the file is being created. The program will create a file called CHAMPS which holds the rank and name of three baseball players.

How the Program Works:

Line 100

```
100 OPEN 2,8,2,"@:CHAMPS,S,W"
```

tells the computer to start up a new file, whose name will be CHAMPS. The number 2 means that in the remainder of the program, we will refer to CHAMPS as file number 2. The number 8 is the computer's shorthand for "disk drive." The number 8 tells the computer that we want to start this file on our disk drive, rather than, say, a cassette recorder.The number 2 refers to the channel which communicates with the disk drive. The @ sign tells the computer that if there is already a file named CHAMPS on the disk, then replace it with this new file that we are about to create. The letter S at the end means that this is a Sequential file — more about that later. The letter W means that we want to Write (record) information in the file.

Lines 200 - 230

```
200 FOR I = 1 TO 3
210 READ A$
220 PRINT#2,I;A$
230 NEXT I
```

store the three records in the file. The READ command tells the computer to READ an item from the DATA list in line 400

```
400 DATA "RUTH","MANTLE","KOUFAX"
```

and to store it in the variable A$.

Line 220

```
220 PRINT#2,I;A$
```

is the command that actually stores information in the file. PRINT#2 tells the computer to store data in file number 2 (which is CHAMPS). I;A$ are the data to be stored.

Line 300

```
300 CLOSE 2
```

tells the computer we are finished writing to the CHAMPS file. Now let's see what the CHAMPS file looks like. To do this, we will write a program called BASEBALL2 which allows us to "peek" at individual bytes in a file. Unfortunately we cannot use the PEEK command to look into files, but there are other commands that allow us to get the same result.

ENTER THIS PROGRAM:

```
1 REM BASEBALL2
100 OPEN 2,8,2,"CHAMPS,S,R"
200 GET#2,A$
210 PRINT ASC(A$),A$
220 IF ST=0 THEN 200
300 CLOSE 2
```

RUN the program. The computer will display the data in CHAMPS, one byte at a time, in two columns. The left-hand column will show the ASCII code for each character. The right-hand column will display the character itself.

The information fills more than one screen. For your convenience, here is a complete display of what will appear on your screen:

```
32
49    1
32
82    R
85    U
84    T
72    H
13

32
50    2
32
77    M
65    A
78    N
84    T
76    L
69    E
13

32
51    3
32
75    K
79    O
85    U
70    F
65    A
88    X
3
```

Most of the codes are obvious in their meaning. We have highlighted those codes which are not.

Each record consists of a number and a name. The code

```
13
```

is used to separate each record from the next. There is a separator like this between

```
1 RUTH
```

and

```
2 MANTLE
```

and

```
3 KOUFAX
```

32 is the ASCII code for a space.

How does the computer know when the end of a file has been reached? It gets a signal from the disk drive. The disk drive is an "intelligent" device — it contains a powerful "microprocessor" chip and considerable RAM memory. It is able to sense when the end of a file has been reached. When it senses the end of a file, it sends a signal to the computer. This signal is stored in a special variable called ST (for STatus). Normally ST is 0. But when the end of a file is reached, ST is set to 64.

Data files can be used to store a wide variety of information. The information is divided into units, with each unit separated by the code 13. When the end of a file is reached, the disk drive sends a signal to the computer, which shows up in the variable ST.

How BASEBALL2 Works:

Line 100

```
100 OPEN 2,8,2,"CHAMPS,S,R"
```

tells the computer that we want to read information in the file CHAMPS. This command is similar to the OPEN command in BASEBALL1. The crucial difference is the R at the end of the command. This stands for Read. This informs the computer that we only want to read information that is already in the file; we do not want to alter or erase any data.

Line 200

```
200 GET#A$
```

is used to "get" individual bytes from the file. Each time the command is executed, it gets one byte and stores it in the variable A$. The GET command retrieves bytes consecutively. The computer automatically keeps track of where the GET command is in a file.

Line 210

```
210 PRINT ASC(A$), A$
```

prints the ASCII code for A$ and A$ itself.

Line 220

```
220 IF ST=0 THEN 200
```

checks the STatus variable, to see if the end of the file has been reached. If the end of file has not been reached, ST will be 0, and the computer will go back to the GET command in line 200. If the end of file has been reached, then ST will be set to 64. Once this happens, the computer will go from line 220 to line 300.

```
300 CLOSE 2
```

This command tells the computer that we are finished reading file 2 (which is CHAMPS).

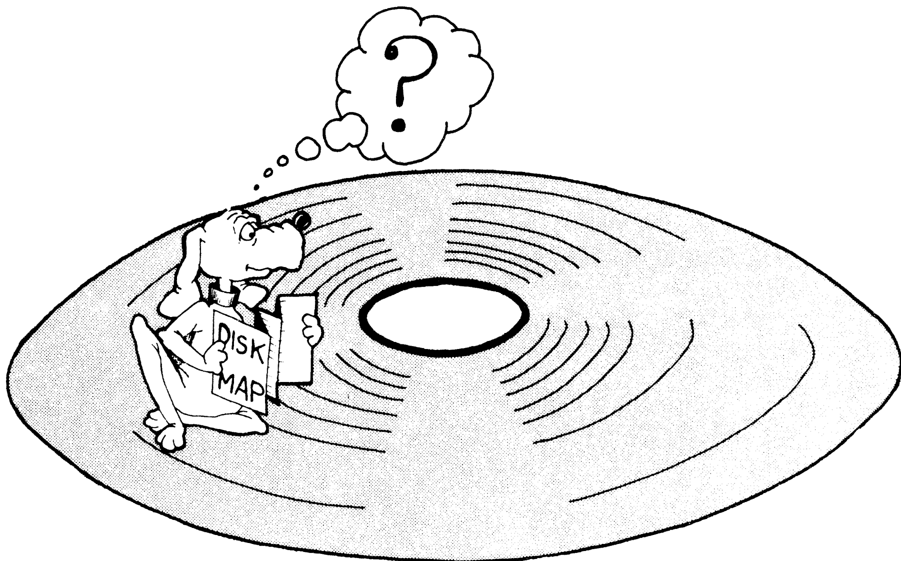
Chapter XII

DISK STORAGE — PART 2

In the last chapter, we found out about the main concepts in the operation of your disk. We also saw how information looks when it is stored in a file.

In this chapter, we are going learn about how the computer keeps track of where various information is on the disk. A floppy diskette can hold up to 144 files and a total of almost 175,000 bytes of data. Yet when you need a piece of information from the disk, the computer can usually find the information almost instantly! How does it do that?

The answer is that on each disk, the computer maintains a pair of "maps," which show where everything is on the disk. When you tell the computer to retrieve information from a file, the computer consults one of these maps to find out exactly where the file is located. The manner in which this is done is very complicated. We are going to survey just the main ideas that are involved, and then take a look at the "maps." They will not look like any maps you've seen before!



As you found out in the last chapter, a diskette is divided into 35 tracks, and each track is divided into sectors or blocks of 256 bytes each. There are 17 to 21 sectors per track. There are 683 sectors altogether. Track 18 holds the two maps of the disk.

One of the maps is called the BLOCK AVAILABILITY MAP (BAM). The BAM shows exactly which sectors (blocks) on the disk are used and which are unused. Every time a file is added to the disk, every time a file is expanded, and every time old files are cleaned off the disk, the BAM is updated. BAM is 140 bytes long. For

each sector on the disk, there is a bit in BAM which is assigned to that sector. If the sector is unused, the corresponding bit in BAM is set to 1. When the sector is used, the computer sets the corresponding bit in BAM to 0. By inspecting the BAM, the computer can tell exactly which sectors have been used and which are unused.

The other map is called the DIRECTORY. The DIRECTORY is a list of files on disk, with some important information about each file. For each entry in the directory, the computer records the name of the file, the file type (Program file, Sequential file, User file, and Relative file), and the track and sector number where the file begins. When you tell the computer you want to use a file, the computer looks in the DIRECTORY to find out where the file begins on the disk. This saves the computer the trouble of searching through the entire disk to find the file you requested, which is why the computer can find files on your disk so quickly.

Once the computer finds the beginning of a file, it can find the remainder of the file quickly. This is because each sector in the file includes a 2-byte pointer, which tells the computer where the next sector of the file is located. The various sectors of a file may not be located side-by-side on the disk, but the pointers in each sector enable the computer to jump from one sector to the next quickly.

The precise layout of the BAM and the DIRECTORY is quite complex, and we will not discuss them in detail here. If you are interested in their exact layout, the details are covered in the Commodore disk drive manual.

The following program will allow you to take a look at both maps, so that you can get a sense for what they are like.

RUN THIS PROGRAM

```
1 REM MAPS
100 OPEN 2,8,2,"$,S,R"
200 GET#2,A$
210 IF A$ = "" THEN PRINT N,0: GOTO 230
220 PRINT N, ASC(A$)
230 N=N+1
240 C=C+1:IF C=20 THEN C=0:INPUT X$
250 IF ST=0 THEN 200
300 CLOSE 2
```

This program will display the contents of a file called $. This is a special file, automatically set up by the computer when you initialize a diskette. This file contains the BAM and the DIRECTORY.

This program will display the contents of $, 20 bytes at a time. Each time you want to see 20 more bytes, press RETURN. Two columns of information will be displayed — the byte number and the value of the byte in that location.

BAM will show up as bytes 2 - 141. You will notice that the pattern of byte values is irregular. This is because there are more sectors in some tracks than others, and because there are more bits in BAM than there are sectors on disk.

The DIRECTORY will begin at approximately byte 257. You will see ASCII codes for the names of files stored on the disk.

How MAPS Works:

The program is similar to BASEBALL2. The main difference is that it processes a file called $, rather than CHAMPS. Also, the information from the file is handled a little differently.

Line 100

```
100 OPEN 2,8,2,"$,S,R"
```

tells the computer we want to read the file called $. This is similar to the OPEN command in BASEBALL2.

Line 200

```
200 GET#2,A$
```

extracts one byte of information from the file. As we shall see, the variable N keeps track of how many bytes have been extracted.

Line 210

```
210 IF A$ = "" THEN PRINT N,0: GOTO 230
```

handles the situation where ASC(A$) will not properly substitute a zero for a "null character."

Line 220 displays the value of N, and the ASCII code of A$.

```
220 PRINT N, ASC(A$)
```

Lines 230 - 240 keep count of the number of characters processed. Every time C gets to 20, the screen display is frozen until RETURN is pressed.

```
230 N=N+1
240 C=C+1:IF C=20 THEN C=0:INPUT X$
```

Lines 250 and 300

```
250 IF ST=0 THEN 200
300 CLOSE 2
```

take care of finishing the program when the end of the file has been reached.

Now, a few concluding remarks about your disk drive.

The COMMODORE 64 disk drive is a very complex device. As was mentioned earlier, it is equipped with its own "microprocessor" and memory. Essentially it has a small computer built right into it and accepts several different sets of commands. Some of these commands are easy to learn (LOAD, SAVE), while others are intended only for advanced programmers. If you want to snoop further into the disk drive, here are some tips:

> Study the user's manual which comes with the disk drive. The manual is stingy with examples and explanations, but you can extract almost everything you need to know by reading it closely.
>
> Plan on writing a lot of experimental programs to familiarize yourself with the disk drive. Practice will fill in the gaps left by the disk drive user's manual. Bear in mind that there are four file types that are allowed by the disk drive. "Program files" are for programs. "Sequential files" are the simplest kind of data files. The file in the last chapter was a sequential file. "Relative files" are more advanced, and "User files" are the most advanced.

Chapter XIII

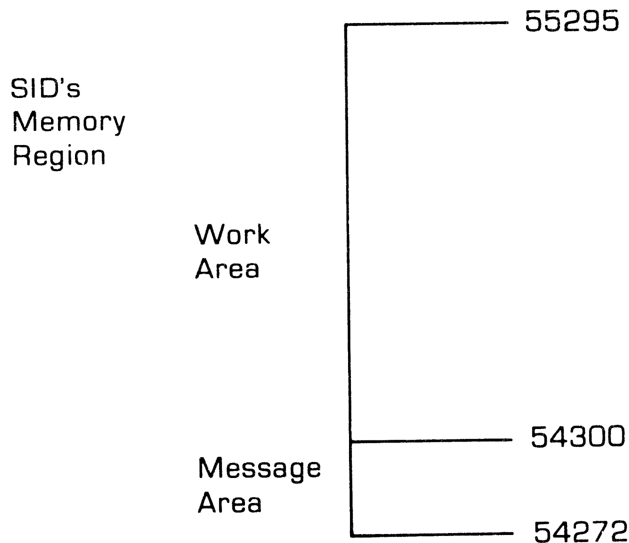
SOUNDS

Earlier in this book, we experimented with the sound-making abilities of the COMMODORE 64. In this chapter, we will find out more about how your computer makes sounds.

As you already know, the COMMODORE 64 is a fabulous sound-making machine. It has many of the capabilities of a good electronic organ or sound synthesizer. It can imitate the the sounds of a piano, a clarinet, a guitar, and many other musical instruments. It can produce fabulous electronic arcade effects and weird "outer space" sounds. The total range of sounds is so vast that many possibilities have not yet even been explored. There are certainly many kinds of sounds and musical effects still waiting to be discovered.

Sounds are made possible by a special device inside your computer called the Sound Interface Device, or SID for short. SID is actually a tiny computer in its own right. It is a *special purpose* computer. Its sole job is to produce sounds. Here are the essentials of how it works:

> SID is "tied in" to a certain region of memory, locations 54272 - 55295. This region is reserved for SID.
>
> The first 29 bytes of this region are reserved as a "message area" for SID. When the computer wants SID to do something, it leaves appropriate information in the message area. SID is constantly watching the message area. When a message comes in, SID does whatever the message says for it to do.



The remaining 995 bytes of the region are reserved as a "work area" for SID's use only.

SID does not actually produce sounds, only streams of electronic signals. When those signals are fed into a TV set or stereo system, they are then converted to sound.

You can control SID by POKEing information into the message area for SID, locations 54272 - 54300. When you POKE information into the message area, SID will "read" the message, and it will do whatever the message tells it to do.

Now, some bad news: SID is not a "friendly" device to communicate with. In order to communicate with it, you must POKE codes into the various positions in the message area. Each position has its own meaning and contributes in its own way to the overall sound. To thoroughly understand SID, you would have to understand the function of each byte in the message area and the way it responds to various POKEs. Also, you would have to understand the ways the various positions contribute to the overall sound. This is sometimes difficult to predict — some combinations produce subtle and surprising effects. Even the best musicians are unable to predict all effects.

So, you cannot expect to understand everything about SID all at once. To help you get started, we will present a map which shows many of the features of the message area. Then we will write a program that will help you to experiment with various POKEs into the message area. This will acquaint you with many of the main features of SID and will provide you with a foundation for your own investigations of SID.

The following map will show you the main features of the message area. (A few of the most advanced features are omitted.) As you study this map, do not expect to understand everything the first time. We are trying to describe sounds with words, which is difficult. Later in the chapter, you will have a chance to experiment with SID and discover with your own ears how it works.

Locations 54272 - 54278: Controls for voice 1. SID is able to produce three completely separate sounds at one time. Each of these sounds is known as a "voice." Each voice has its own set of controls. Voice 1 is controlled with the information in locations 54272 - 54278. We will now explain the meaning of each of those locations.

Locations 54272 - 54273: Frequency (pitch). The values in this location determine the "pitch" of the sound — how high or low it is. A high pitch is like a piccolo or the chirp of a bird. A low pitch is like a tuba or the growl of a bear.

The pitch value is defined as

(VALUE OF BYTE 54272) + (256 * VALUE OF BYTE 54273)

The higher this value, the higher the pitch.

Locations 54274 - 54275: Pulse waveform width. This is used to adjust the sound quality of "pulse waveform," described below.

Location 54276: Sound quality. Sounds can have various qualities — they can be pure, mellow, buzzy, raspy, noisy, and so on. SID can produce four important basic sound qualities. All four of these qualities have precise technical characteristics which we will not discuss here. The four qualities are:

TRIANGLE WAVE: Mellow, flute-like

SAWTOOTH WAVE: Bright, brassy

PULSE WAVE: Can have a variety of qualities, some of them surprising. There are many kinds of pulse waves, varying from "thin" to "fat." This can be controlled in locations 54274 - 54275.

WHITE NOISE: Sounds like radio static

The following chart gives the POKEs for these four qualities:

QUALITY	START	STOP
TRIANGLE WAVE	17	16
SAWTOOTH WAVE	33	32
PULSE WAVE	65	64
WHITE NOISE	129	128

So, if you wanted to make a sound with a pulse wave, you would start the sound by POKEing 65, and you would stop it by POKEing 64. This will be illustrated later in the chapter.

Locations 54277 - 54278: ADSR. The volume of a sound changes as you are listening to it. For instance, the sound of a bell begins with an intense ping, then the volume drops quickly to a lower level, holding at nearly that level for a long time before it dies away very slowly. The sound of a stick breaking begins more gradually, climaxes with an intense S N A P, and then disappears quickly. Many sounds get much of their distinctive character from the subtle ways in which their volume rises and falls.
Scientists measure the rise and fall of volume by means of four concepts:

ATTACK: How suddenly the sound begins

DECAY: How quickly the sound falls to a middle-range volume

SUSTAIN: The level of the mid-range volume

RELEASE: How slowly it dies away

Attack, Decay, Sustain, and Release are known as the ADSR for a sound or the "envelope" for the sound. The following is a sample ADSR chart for a bell:

ATTACK, DECAY, SUSTAIN, and RELEASE each may have a value of 0 - 15. Here is how the values of locations 54277 and 54278 are set:

VALUE OF LOCATION 54272 = (16 * ATTACK) + (DECAY)

VALUE OF LOCATION 54273 = (16 * SUSTAIN) + (RELEASE)

You will have a chance to experiment with ADSR later in the chapter.

This completes the list of controls for Voice 1.

Locations 54279 - 54285: Voice 2. Similar to voice 1.

Locations 54286 - 54292: Voice 3. Similar to voice 1.

Locations 54293 - 54295: Filter. This is similar to the "Treble" and "Bass" tone controls on a stereo system.

Location 54296: Filter, Volume. This location is used to control both the filter and the overall volume of sound produced by SID. The filter control may have a value of 0 - 7. It works along with the "Treble" and "Bass" controls, to affect the proportion of "highs" and "lows" that you hear in a sound. The volume control regulates the overall volume of sound produced by SID. It may have a value of 0 - 15; 0 means total silence, 15 is the highest volume. Here is how the value of location 54296 is set:

VALUE OF LOCATION 54296 = (16 * FILTER) + (VOLUME)

Locations 54297 - 54298: Connections with other devices. It is possible to connect joysticks and other devices to SID. For instance, you could tie in a joystick so that whenever you move the joystick forward, SID makes a higher sound, and whenever you move the joystick backwards, SID makes a lower sound. It is possible to connect up to two devices to SID. The signals from these devices will show up in locations 54297 and 54298.

Location 54299 - 54300: Readouts on Voice 3. These two locations describe in mathematical terms what voice 3 is doing. A clever programmer can "feed back" this information into other parts of the message area, and produce very complex effects.

The following chart summarizes the POKE values which are possible for various locations in the SID message area:

LOCATION	POSSIBLE SETTINGS	COMMENTS
54272	0 - 255	Locations 54272 - 54273 control
54273	0 - 255	frequency (pitch) for Voice 1.
54274	0 - 255	Locations 54274 - 54275 control pulse
54275	0 - 15	waveform for Voice 1. Used only when
		pulse wave has been selected.
54276	16, 17	Selects sound quality for Voice 1.

LOCATION	POSSIBLE SETTINGS	COMMENTS
	32, 33 64, 65 128, 129	Options are: TRIANGLE WAVE, SAWTOOTH WAVE, PULSE WAVE, and WHITE NOISE.
54277	0 - 255	Locations 54277 - 54278 select the ADSR ("envelope") for Voice 1. Location 54277 selects Attack and Decay.
54278	0 - 255	Selects Sustain and Release for Voice 1.
54279 - 54285		Controls for Voice 2
54286 - 54292		Controls for Voice 3
54293	0 - 7	Locations 54293 - 54295 control the filter.
54294	0 - 255	
54295	0 - 255	
54296	0 - 255	Controls filter, overall volume for SID.
54297	0 - 255	Locations 54297 - 54298 allow joysticks, game paddles, other devices to be tied in with SID. The signals received from these devices are registered here.
54298	0 - 255	
54299 - 54300		Locations 54299 - 54300 give readouts on what is occurring with Voice 3. This information is intended for programmers who want to produce "feedback" effects with SID. You may PEEK at the information in these locations, but POKEing was not intended.

To really understand the meaning of the various controls for SID, you need to experiment with the many combinations of POKEs that are possible with SID. To get you started, we are going to write a program which helps you to experiment with some of the settings for Voice 1 only. This will show you a great deal about what SID can do. And you can use the program as a basis for further experiments on your own.

ENTER THIS PROGRAM:

```
1 REM SOUNDLAB
100 POKE 54296,15
110 B=54272
200 LF=0:HF=250
210 LP=0:HP=15
220 QQ=16
230 A=0:D=1
240 S=0:R=9
300 T1=1:T2=500
```

```
600 AD= 16*A + D
610 SR= 16*S + R
700 POKE B,LF:POKE B+1,HF
710 POKE B+2,LP:POKE B+3,HP
720 POKE B+5,AD:POKE B+6,SR
800 POKE B+4,QQ+1
810 FOR Z=1 TO T1:NEXT Z
820 POKE B+4,QQ
830 FOR Z+1 TO T2:NEXT Z
900 GOTO 700
```

This program POKEs various settings into the message area for Voice 1. Run the program. You will hear a little bell ringing.

Let's take a look at how the program works. Then we will experiment with SID by modifying the program in various ways.

How SOUNDLAB Works:

SOUNDLAB is a "skeleton" program which POKEs values into the message area for Voice 1. The easiest way to explain what it does is to list the meanings of the variables in the program.

B	The beginning of the message area for SID
LF, HF:	These set the frequency (pitch) of the sound.
LP, HP:	If the sound quality is PULSE, LP and HP control the width of the pulse.
QQ:	This sets the sound quality. The possible settings are:

16	TRIANGLE WAVE
32	SAWTOOTH WAVE
64	PULSE WAVE
128	WHITE NOISE

A,D,S,R	These set Attack, Decay, Sustain, and Release
T1	When a sound is turned on, this controls how long it stays on.
T2	SOUNDLAB plays a note over and over. T2 controls how much time there is between notes.

Now let's see find out what the program does.

Line 100

```
100 POKE 54296,15
```

sets the volume control to the highest possible setting, which is 15.

Lines 110 - 300

```
110 B=54272
200 LF=0:HF=250
210 LP=0:HP=15
220 QQ=16
230 A=0:D=1
240 S=0:R=9
300 T1=1:T2=500
```

set the variables listed above to appropriate values.

Lines 600 and 610 perform some calculations that are needed before information can be POKEd.

```
600 AD= 16*A + D
610 SR= 16*S + R
```

The next group of lines POKE information into the appropriate locations.

```
700 POKE B,LF:POKE B+1,HF
710 POKE B+2,LP:POKE B+3,HP
720 POKE B+5,AD:POKE B+6,SR
```

Lines 800 - 830 start and stop the tone.

```
800 POKE B+4,QQ+1
810 FOR Z=1 TO T1:NEXT Z
820 POKE B+4,QQ
830 FOR Z=1 TO T2:NEXT Z
```

Line 800 starts the tone. This is done by POKEing one more than the value of QQ into the appropriate location. Line 810 is a time delay loop. It determines how long the tone will sound before it is turned off.

Line 820 turns off the sound. This is done by POKEing the QQ into the same location where QQ was. Line 830 is another time delay loop.

Line 900

```
900 GOTO 700
```

enables the program to repeat the tone again and again.

Now let's experiment with the program, to learn more about SID. First we will learn about frequencies.

Add the following line to the program:

```
840 HF = HF-10
```

This will decrease the frequency (pitch) of the sound each time it is repeated. Run the program. You will hear the bell tone get lower and lower. When HF decreases below 0, you will get an error message.

To find out about sound qualities, change line 220 to

```
220 INPUT QQ
```

You will be able to INPUT various numbers into QQ when the program starts. The allowable values are:

```
16     (SAWTOOTH)
32     (TRIANGLE)
64     (PULSE)
128    (WHITE NOISE)
```

Try INPUTing each of these values, and notice the different kinds of sounds produced. The value of 128 is especially interesting; it produces a horrendous "banging" sound.

Now let's play with Attack, Decay, Sustain, and Release. Change line 220 back to

```
220 QQ=16
```

Change line 200 to

```
200 LF=0:HF=100
```

This will lower the pitch of all sounds to a more convenient level.

Add the following line.

```
245 INPUT A,D,S,R
```

This will allow you to INPUT values for Attack, Decay, Sustain, and Release.

Run the program and try these values for A, D, S, and R:

A	D	S	R	Comments
0	1	0	9	Bell
0	1	0	0	Clink or blip
0	15	0	0	Short toot
15	15	0	0	Almost a tap
3	7	7	7	Plink

ADSR settings can produce colorful effects when used with the WHITE NOISE waveform (QQ=128). Let's try a few examples.

Change line 220 to

```
220 QQ=128
```

This will tell SID to produce WHITE NOISE. Run the program and then INPUT the following values:

A	D	S	R	Comments
0	1	0	9	Horrendous banging
15	1	1	1	Footstep
15	1	1	10	21 gun salute

Let's find out about pulse waveforms. Pulse waves can be "fat" or "thin" and can be controlled by varying the value of the variable HP.

Make the following changes in your program:

> Remove line 245. This was the line that allowed you to INPUT various values for A, D, S, and R.
>
> Change line 220 to
>
> ```
> 220 QQ=64
> ```
>
> This tells SID to use the pulse waveform.
>
> Add this line:
>
> ```
> 840 HP= HP - 1
> ```
>
> This will gradually reduce HP from 15 to 0.

Now run the program. You will hear the quality of the sound change as HP is reduced from 15 to 0. When HP gets below 0, the program will stop with an error message.

Finally, you should experiment with the timing loops controlled by T1 and T2. By varying these loops, you can create very interesting "interference patterns" between sounds. To get acquainted with these effects, rerun any of the experiments in this chapter, but change line 300 to

```
300 T1 =1:T2=1
```

or change it to

```
300/ T1 =100:T2=1
```

If you would like to find out more about SID, here are some suggestions:

Experiment some more with SOUNDLAB. This will help you to discover with your own ears how individual features of SID work.

After you are familiar with the individual features of SID, you need to find out how to combine features and produce interesting musical effects. A good place to start is the *Commodore 64 User's Guide,* which has some nice examples of sound programs.

For a detailed understanding of SID, see the *Commodore 64 Programmer's Reference Guide.* This is hard reading, so don't expect to understand everything the first time!

To become really creative with SID, find out about the subject of "acoustics" — the science of sound. It will help you to understand the "nuts and bolts" of sound effects and musical qualities, and will stimulate your own creativity.

Chapter XIV

MACHINE LANGUAGE

Several times we have mentioned 6510 Machine Language. All of the routines of the KERNAL are written in this language. Also, when you are running a program in BASIC, the computer actually translates each command into 6510 machine language before executing it. The computer does this because machine language is the "native language" of the computer. The computer cannot perform any task unless it has been expressed in the commands of machine language.

In this chapter we will find out what machine language programs are like.

Machine language consists of the simplest, most fundamental commands that can be given to a computer. Some sample commands are:

> Transfer a byte of information from one storage location to another.
>
> Add two numbers.
>
> Multiply two numbers.
>
> Compare two values; if they are the same, set a certain storage location to 1; otherwise set it to 0.
>
> Compare the individual bits in two different bytes; whenever the bits don't match, set the appropriate bit in the first byte to 0.
>
> Go to a certain place in the machine language program, if a certain value is zero.

Many of the commands that you enjoy in BASIC do not have counterparts in machine language; a few examples are:

```
PRINT
X$ = "EVERYTHING IS FINE"
SAVE
INPUT
IF ... THEN
FOR ... NEXT
```

It is possible to use a group of machine language instructions that will accomplish the same results as any of these BASIC commands, but it is tedious work. A machine language program to do the same job as this command

```
SAVE "SNOOP",8
```

would require hundreds of instructions!

Why would anyone use machine language? There are three main reasons:

The COMMODORE 64 is constructed from certain fundamental components which only understand machine language. So the engineers who developed the COMMODORE 64 had no choice but to do some of the initial programming in machine language.

Machine language programs run much faster than BASIC programs, often 10 to 100 times faster.

You can do things in machine language which are impossible in BASIC. For instance, many communications programs must be written in machine language in order to work reliably (Communications programs often require data to be transmitted and received at precise speeds. BASIC programs execute at slightly "uneven" speeds, and this makes BASIC unsuitable for some communications programs. In machine language, the execution speeds can be controlled precisely.)

Here is what a machine language program looks like:

10100000000000001010100100100000100110010000000000000100
10011001000000000000010110011001000000000000011010011001
00000000000001111100100011010000111100011010000000000000
10101001000001111001100100000000000001001100100010101001
00001111100110010000000000000100110010001010100100010011
10011001000000000000010011001000101010010000100010011001
00000000000001001100100010100000000000001010100100000001
10011001000000001101100011001000100110010000000011011000
11001000100110010000000011011000110010001001100100000000
1101100001100000

Each 1 or 0 is one bit in the program. The job of this particular program is to clear the screen, and then display the word GOSH in the upper left-hand corner. We are going to analyze this program piece by piece to see how it works. Our aim in doing this is not really to learn machine language — that would require an entire book in itself. Instead, we just want to get a sense for what the language is like. This will give us insight into how the computer really "thinks."

The program consists of 31 commands. There is no punctuation in the program to indicate where one command ends and the next one begins. The computer figures out the beginning and end of each command from context. For instance, the program begins with the following eight bits:

```
10100000
```

From this the computer knows that the first command is an LDY- Immediate command, which is two bytes (16 bits) in length. So it knows that the second command begins with the 17th bit in the program. When the computer examines the second

command, it will figure out how many bits long that command is, and then it will know where the third command begins. And so on. To make the program easier to read, we are going to break it up into the 31 commands of which it is composed.

```
[  1 ]    1010000000000000
[  2 ]    1010100100100000
[  3 ]    100110010000000000000100
[  4 ]    100110010000000000000101
[  5 ]    100110010000000000000110
[  6 ]    100110010000000000000111
[  7 ]    11001000
[  8 ]    1101000011110001
[  9 ]    1010000000000000
[ 10 ]    1010100100000111
[ 11 ]    100110010000000000000100
[ 12 ]    11001000
[ 13 ]    1010100100001111
[ 14 ]    100110010000000000000100
[ 15 ]    11001000
[ 16 ]    1010100100010011
[ 17 ]    100110010000000000000100
[ 18 ]    11001000
[ 19 ]    1010100100001000
[ 20 ]    100110010000000000000100
[ 21 ]    11001000
[ 22 ]    1010000000000000
[ 23 ]    1010100100000001
[ 24 ]    100110010000000011011000
[ 25 ]    11001000
[ 26 ]    100110010000000011011000
[ 27 ]    11001000
[ 28 ]    100110010000000011011000
[ 29 ]    11001000
[ 30 ]    100110010000000011011000
[ 31 ]    01100000
```

We have numbered these commands from [1] to [31]. Now let's look at some of these commands individually. (If you are not interested in all of the technical detail, it's okay to just skim through this discussion.)

Command [1]

```
1010000000000000
```

is known as an LDY-Immediate command. It tells the computer to "load" eight bits into a storage location known as "register Y." Here is how to read the command:

The first eight bits

```
10100000
```

mean that this is an LDY-Immediate command. The next eight bits

00000000

are the bits to be moved into Register Y.

So, in summary, the command says to move the following eight bits

00000000

into register Y.

A *register* is a special kind of memory that is used for tasks that must be performed at the highest possible speed. Register-memory is engineered in a different way than regular memory. It is expensive and operates at a higher speed than ordinary memory. A register holds eight bits. There are just seven registers altogether. Registers cannot be accessed from BASIC by using PEEK and POKE commands.

Most machine-language operations require that the data involved be moved into registers to be operated upon. So, if you want to add two numbers, one of the numbers must first be placed in a register, and the result of the addition is left in a register. This entails a lot of shuffling around of information inside the computer. However, engineers have found this to be a very efficient way of designing computers overall — the advantages outweigh the inconveniences.

Command [2]

1010100100100000

is known as an LDA-Immediate command. This is similar to the previous command. It tells us to load eight bits into a storage location known as "register A." Here is how to read the command:

The first eight bits

10101001

mean that this is an LDA-Immediate command. The next eight bits

00100000

are the bits to be moved into register A.

So the command says to move the following eight bits

00100000

into Register A.

Command [3]

1001100100000000000000100

is known as a STA-Absolute,Y command and is similar to a POKE command. It stores the contents of register A into a certain location in memory. Here is how it works:

The first eight bits

10011001

tell the computer that this is a STA-Absolute,Y command.

The remaining 16 bits

0000000000000100

are used by the computer to determine the location where the contents of register A will be stored. To determine that location, the computer calculates a value based on those 16 bits, and then adds the value currently in register Y.

In summary, STA-Absolute,Y is like a POKE command. In this case, the command does the same work as the following POKE comand

POKE 1024,32

Commands [4], [5], and [6] are also STA-Absolute,Y commands. Notice that they all begin with the eight bits

10011001

This tells the computer that they are STA-Absolute,Y commands. All machine language commands are identified by the first eight bits.

We will come back and find out more about commands [4], [5], and [6] later. Right now, let's look at a few other commands in the program.

Command [7]

11001000

is known as an INY command. This command tells the computer to "increment" the value of register Y by 1. For example,

OLD VALUE IN REGISTER Y	NEW VALUE IN REGISTER Y
00000000	00000001
00000001	00000010
00000010	00000011
11111110	11111111

If the old value in register Y is 11111111, INY will change it to 00000000.

OLD VALUE IN REGISTER Y	NEW VALUE IN REGISTER Y
11111111	00000000

Command [8]

1101000011110001

is known as a BNE command. This tells the computer that if the value in register Y is not equal to 00000000, it should back up 15 bytes in the program. This takes the computer back to Command [3]. BNE is kind of a special version of an IF ... THEN command in BASIC. BNE says that if a certain value is not 00000000, then go to a certain location in the program. Here is how to read the command:

The first eight bits

11010000

tell the computer that this is a BNE command.

The next eight bits

11110001

tell the computer how far to back up in the program if the value of register Y is not 00000000.

Command [31]

01100000

is known as an RTS command. This tells the computer that the program is done and to return to whatever the computer was previously doing before this program began.

You have now seen all the different kinds of commands that go into the program. The entire program is just a combination of LDY, LDA, STA, INY, BNE, and RTS commands. Let's review what these commands do:

LDY and LDA load information into registers.

STA stores some information from a register into a location in memory.

INY increments that value in a certain register.

BNE tells the computer to go back to an earlier instruction if the value in a certain register is not 00000000.

RTS finishes the program.

All of these commands perform very simple tasks. The entire program consists of loading information into registers, transferring information from registers to certain locations in memory, incrementing values in registers, and a sort of IF ... THEN loop. Since the individual commands are so simple, couldn't you write a program in BASIC that would do the same work? The answer is Yes. Here is a program in BASIC which is almost identical, line for line:

	MACHINE LANGUAGE		BASIC
[1]	1010000000000000	1	Y=0
[2]	1010100100100000	2	A=32
[3]	100110010000000000000100	3	POKE 1024+Y, A
[4]	100110010000000000000101	4	POKE 1280+Y, A
[5]	100110010000000000000110	5	POKE 1536+Y, A
[6]	100110010000000000000111	6	POKE 1792+Y, A
[7]	11001000	7	Y=Y+1:IF Y=256 THEN Y+0
[8]	1101000011110001	8	IF Y<>0 THEN 3
[9]	1010000000000000	9	Y=0
[10]	1010100100000111	10	A=7
[11]	100110010000000000000100	11	POKE 1024+Y, A
[12]	11001000	12	Y=Y+1:IF Y=256 THEN Y=0
[13]	1010100100001111	13	A=15
[14]	100110010000000000000100	14	POKE 1024+Y, A
[15]	11001000	15	Y=Y+1:IF Y=256 THEN Y=0
[16]	1010100100010011	16	A=19
[17]	100110010000000000000100	17	POKE 1024+Y, A
[18]	11001000	18	Y=1:IF Y=256 THEN Y=0
[19]	1010100100001000	19	A=8
[20]	100110010000000000000100	20	POKE 1024+Y, A
[21]	11001000	21	Y=Y+1:IF Y=256 THEN Y=0
[22]	1010000000000000	22	Y=0
[23]	1010100100000001	23	A=1
[24]	100110010000000011011000	24	POKE 55296+Y, A
[25]	11001000	25	Y=Y'+1:IF Y=256 THEN Y+0
[26]	100110010000000011011000	26	POKE 55296+Y, A
[27]	11001000	27	Y=Y+1:IF Y=256 THEN Y=0
[28]	100110010000000011011000	28	POKE 55296+Y, A
[29]	11001000	29	Y=Y+1:IF Y=256 THEN Y=0
[30]	100110010000000011011000	30	POKE 55296+Y, A
[31]	01100000	31	END

As you read through the BASIC version, think of the variable Y as standing for Register Y, and think of the variable A as standing for Register A.

Now that we have seen in detail what machine language is like, let's summarize some important points about it.

Machine language is frustrating to read. The lack of punctuation and the exclusive use of 1s and 0s are especially troublesome.

The individual commands perform very simple tasks. It is fairly easy to understand what each individual command does. The only real difficulty is in understanding the details of how various numbers are expressed (i.e., how the bits 0000000000000100 stand for location 1024 in memory). We have not discussed those difficulties in detail, but they are fairly easy to master.

It takes a lot of commands to perform even a simple task. In our sample program, it took seven commands just to clear the screen.

Machine language is really the only language that the computer can understand. If you give it a command in BASIC, the computer will translate the command into a series of machine language commands before executing your original command.

Your computer can process machine language commands very quickly, often 10 to 100 times faster than equivalent programs in BASIC. Machine language is awkward for human beings, but easy for the computer.

You can do things with machine language that cannot be done at all with BASIC.

Memory is filled with hundreds of machine language routines to perform various fundamental tasks. Although any single routine performs a small task, the computer system can build more complex routines from simpler ones.

If you were to analyze how the computer performs a complex task, you would see that it usually breaks it down into a series of simpler tasks. For instance, consider the task of displaying a program on the screen when you enter the command LIST. This requires the computer to trace through the entire program, line by line, character by character, and to translate each code into the appropriate code for displaying on the screen. It requires deciding when to begin a new line of display on the screen. It also requires the ability to handle special situations. (What to do if the program is too big to fit on the screen, what to do if something is encountered in the program which does not make any sense.) The computer carries out this job by breaking it down into some simpler tasks. Here are some of the simpler tasks:

Find the next byte of the program in memory.

Translate a byte from the program into one or more characters that can be displayed on the screen.

Display a certain character at a certain position on the tscreen.

Check to see if a certain position on the screen is the last position on the screen.

If the screen is filled, scroll the screen up one line.

Advance to the next unused position on the screen.

These are not all of the tasks required for LISTing a program. Some important tasks have not been mentioned at all. Also, some of the tasks shown above can be broken down into a set of still simpler tasks. But this example makes it clear that if a computer knows how to perform a large number of simple tasks, it will then be able to combine these tasks to accomplish more significant work.

THE THIRD RULE OF COMPUTER SNOOPING:

Your computer system is built up in layers.

The way the COMMODORE 64 was created was to start with some electrical components that can perform various simple commands. These components were assembled in the same "box." Machine language programs were then developed to "teach" the computer to perform a large number of simple but important tasks. Once this was accomplished, the next step was to develop some additional machine language programs that combine the simpler programs, in order to accomplish more complex tasks. Here are some examples of what is found in each layer:

LOWEST LAYER: Machine language commands such as LDY-Immediate, LDA-Immediate, STA-Absolute,Y, INY, BNE, and RTS.

NEXT LAYER: Simple machine laguage routines such as:

> A machine language routine to clear the screen.
>
> A machine language routine to display the word READY on the screen.
>
> A machine language routine to make the cursor flash at the proper speed.

HIGHER LAYERS: Machine language routines that combine simpler routines to perform more complex tasks:

> A routine to display the appropriate symbols on the screen, in the right places, as you are typing.
>
> A routine to restore the computer back to normal when you press STOP-RESTORE.

EVEN HIGHER LAYERS: Machine language routines which translate your BASIC commands into machine language routines.

Chapter XV
HARDWARE

In each chapter of the book so far, we have investigated some important aspect of your COMMODORE 64. We have found out about the screen, various parts of memory, programs, variables, sound, and machine language.

In this chapter we will take a different approach. Instead of concentrating on just one aspect, we will take an overview of the entire computer system. To do this, we will survey all of the main components of the COMMODORE 64, and discuss how they work together.

The main parts of the COMMODORE 64 are

An MOS 6510 Microprocessor. This is the device that does most of the thinking for the computer. It processes all machine language commands. It organizes and controls almost all activities of the computer.

Memory. As we have seen, memory is used for many things. It holds reference information and machine language programs. It is used to track most of the activities that go inside the computer. Parts of it are reserved for use as a scratch pad.

Add-ons (Peripherals). This includes all the devices that can be attached to the COMMODORE 64 to make it more convenient and useful. Examples are: Screen, printer, disk drive, modem, and joystick.

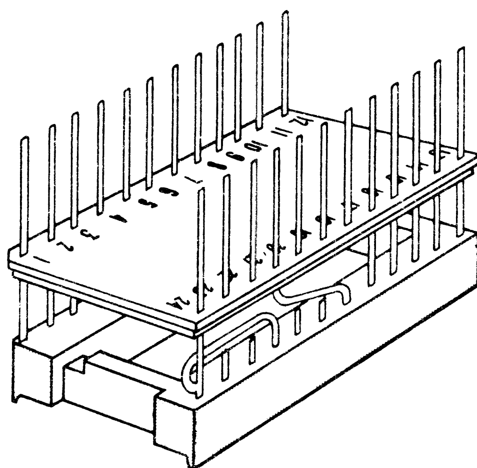
Communications lines. These lines enable the different parts to send information to each other and to work together.

Special-purpose devices. The COMMODORE 64 is equipped with several devices which specialize in certain kinds of tasks. One example is SID, which specializes in making sound and music. Another example is the VIC II microprocessor, which specializes in screen displays.

Let's find out about each of these parts in more detail.

MOS 6510 microprocessor. The MOS 6510 microprocessor is what does most of the "thinking" for the computer. It is actually a complete computer in miniature. From the outside it looks like a little black box with 40 wire "teeth" on it. (The technical term for them is "pins.")

Inside that black box is an enormous amount of complex electronic circuitry.



This circuitry is similar to what used to be found in big computers. Because of some major engineering advances in recent years, it has become possible to miniaturize all of that circuitry and pack it into a tiny area. Don't be fooled by the smallness of the 6510 — it is a very powerful computer. Some of its capabilities are as follows:

> It can be programmed in 6510 machine language. This is the language which has been mentioned several times in the book. You saw an example of it in the previous chapter.
>
> It can process machine language commands at speeds approaching one million commands per second.
>
> Although the 6510 has very little memory of its own (only about a dozen bytes), you may attach to it over 64,000 bytes of memory.
>
> The 6510 is a "stripped down" computer — it comes without a keyboard, screen, disk drive, printer, or any other "peripherals." However it allows peripherals to be attached to it. It is possible to attach dozens of peripherals to the 6510 at one time.

Obviously, the 6510 does not look like the usual computer that you see in a computer store. The main differences between a 6510 and a regular microcomputer are:

> A 6510 is much smaller and cheaper. (A 6510 can be bought for less than $3.)
>
> A 6510 comes with very little memory — only about a dozen bytes.
>
> A 6510 comes without any peripherals such as a disk drive, screen, or keyboard.

While it is possible to "hook up" various devices to a 6510, it's hard work. Let's make a comparison: If you want to hook up a printer to your COMMODORE 64,

all you have to do is go to a computer store and buy a printer that is advertised as being compatible with the COMMODORE 64. Then you go home and plug it into the back of your computer. If you wanted to hook up a printer to a 6510, here are some of the problems you would be up against:

> A 6510 does not know how to communicate to a printer. In order to make a printer print, you need to send it various streams of signals, transmitted at a certain precise speed. You also have to be able to receive and understand various signals that the printer may send to you, such as the signal for "I am out of paper." A 6510 does not know how to do any of this. In order for a 6510 to communicate properly with a printer, you would have to write a series of machine language programs that "teach" the 6510 how to communicate.
>
> A 6510 does not know how to handle problems that can occur when a printer is operating. For instance, the 6510 does not know what to do if the printer is sending signals that it is busy, or if it is getting no response from the printer. Programs must be written that tell the 6510 how to detect problem conditions and how to respond to them.

Memory. There are several different kinds of memory in your COMMODORE 64. First, there is the small amount of memory that is in the 6510. This memory is divided into units called *registers*. Each register holds either 16 or 8 bits. Registers are used by the 6510 as little scratch pads to hold information which it currently needs. This memory is not accessible through BASIC.

Second, there is 64K of "read-write memory," or RAM as it is more commonly called. RAM is defined as memory where information may be stored, altered, or erased. When you turn off the computer, all information in RAM is lost. Inside your computer, RAM takes the form of eight little black boxes called "RAM chips." Each RAM chip holds 8K.

Third, there are three important units of "read-only memory" (ROM). ROM is memory which has information permanently frozen into it. This information may be read, but it may not be altered or erased. The three ROM units are:

> Character ROM (4K): This holds the shape tables for all symbols that can be drawn by the computer.
>
> BASIC Interpreter ROM (8K): This holds the machine language routines for processing BASIC commands.
>
> KERNAL ROM (8K): This holds the machine language routines for the KERNAL. The KERNAL is responsible for the some of the most fundamental tasks of the computer, such as making the cursor blink.

The work done by these units was discussed in Chapter 7, THE MAP OF MEMORY.

Fourth, most peripherals, and most "special-purpose" units (such as SID), have at least a little of their own memory reserved for their own use.

Finally, it is useful to remember that disk drives are also a kind of memory. They are used mainly to store large quantities of information for a long period of time. Also, they can be used as a spillover area when other memory gets filled. For instance, if you write a program which is too large to fit into memory, you could split it into two parts, and store each part as a separate program on the disk. When you run the program, just one of the two parts can be kept in memory; when the other part is needed, it can be loaded from the disk.

Communications lines. In order for the 6510 to communicate with memory and the other parts of the computer, some communications lines are needed. There are two ways that the 6510 is linked up to other devices.

> In some cases there are special lines directly between the 6510 and other devices. For instance, if you are storing a program on a cassette recorder, there is a direct line between the cassette recorder and the 6510, which tells the 6510 when it is okay to begin sending data over to the cassette.
>
> In other cases, there are communications lines which run through the computer that many devices can "tap into." A communications line like this is known as a "bus." A bus is like a road on which many different electrical signals can travel. Many devices may be tapped into the same bus. All messages sent on a bus are accompanied by additional signals which "identify them." That way, each device knows which signals are for it and which to ignore.

The "bus" concept makes it possible for a very large number of devices to be attached to the computer. Each device is tapped into the bus and is assigned a code which identifies it. Any signals for that device are identified with its code.

Add-ons (Peripherals). Some of the peripherals you are already familiar with:

Keyboard
Screen
Cassette recorder
Disk drive
Printer
Joystick
Modem

Each of these peripherals is able to communicate with the 6510 through the buses. The communications protocols can become quite complex. For instance, if you are SAVEing a program on disk, the computer must

> Verify that the disk drive is on.
>
> Tell the disk drive to get ready to receive some information.
>
> Send the data to the disk drive.
>
> Check to make sure that the data has been received by the disk drive.

Actually, even more steps than this are required. For those steps to be carried out, a number of signals must be sent and received by the 6510.

Special devices. If the 6510 had to do everything all by itself, the speed and performance of your computer would be hampered. So the designers of the COMMODORE 64 equipped it with a number of special-purpose devices. Each device specializes in certain kinds of tasks. Here are the most important special devices.

> 6581 Sound Interface Device (SID): You have already learned a lot about SID. SID is a special-purpose microprocessor whose sole job is to create sounds and music.
>
> 6566/6567 Video Interface Controller (VIC-II): VIC-II is a special-purpose microprocessor which is responsible for everything that happens on the screen. Whenever the 6510 wants something to happen on the screen, the 6510 sends a message to the VIC-II, and VIC-II takes care of the rest. A few examples of what VIC-II can do:
>
>> Display the letter Q in yellow in the upper left-hand corner of the screen.
>>
>> Make the border of the screen red.
>>
>> Move sprites around the screen.
>>
>> Make the screen scroll.
>
> 6526 Complex Interface Adapter (CIA): CIA is "switchboard central" for signals that are traveling between the computer and its peripherals. It makes sure that none of the signals interfere with each other or get lost. There are so many signals traveling through your computer that two CIAs are needed to keep all of the signals under control.

Now we can summarize, in just a few paragraphs, how the COMMODORE 64 works:

The part of the computer that "runs the whole show" is the 6510. The 6510 is a tiny computer that can process commands written in 6510 machine language.

Your computer's memory always has a large number of machine language programs for all the elementary tasks that are required to run a computer. If you take a snapshot of the 6510 at any time, you will discover that it is always executing one of these programs.

The 6510 can control other devices by sending signals to them through communications lines. Some of these devices are there for your benefit — e.g., the screen, the keyboard, the printer, and so on. Other devices are there to perform special jobs. For instance, SID is in charge of sound, and VIC II is in charge of the screen. The CIAs act as "switchboards" to keep under control all of the signals that are traveling around the computer.

The 6510 can also receive signals from other devices through the communications lines. Depending on what signals are received, the 6510 decides what it is going to do next.

In summary, the 6510 is the primary "thinking organ" in the computer. Its behavior is made more intelligent by means of the many machine language programs which reside in memory. These programs tell the 6510 how to perform the many different tasks that it must perform. The 6510 gets help from special-purpose devices such as SID, VIC-II, and the CIAs and can tie in with many other devices through communications lines.

Chapter XVI

CONCLUSION

You have reached the end of our introductory tour of the internals of the COMMODORE 64. We would now like to draw some final conclusions, and make some suggestions for how you can best continue learning about the COMMODORE 64.

So far, we have presented three rules of Computer Snooping:

1. The computer is an alien form of lower intelligence.

2. You can learn a lot about how your computer works by running experiments.

3. Your computer system is built up in layers.

To these we add a final rule:

4. To understand your computer, find out about

Codes
Pointers
Languages
and Maps

that pertain to your computer.

By **codes** we mean ASCII, binary, the codes for variable types, the codes that represent the various commands in BASIC, the codes for representing the value of floating-point variables, and so on. As you have discovered, the computer uses codes extensively to put information in a form that is convenient for its own use. These codes often look horrendous the first time you see them, but we have seen that it often is possible to make sense of them pretty quickly. Of course, it helps a lot to have good reference sources to explain the codes, and we will suggest some shortly.

By **pointers** we mean the computer's way of designating various locations in memory. Your computer uses pointers extensively. Pointers are often tedious to trace, but the main concept of pointers can be grasped fairly quickly. When you deal with pointers, make sure you understand the exact rules that the computer uses to determine the meaning of a pointer.

The key **language** for the COMMODORE 64 is 6510 machine language. A knowledge of that language is very useful for snooping. There are also some minor languages that are useful to know, for instance the "language" of codes that the COMMODORE 64 uses for communicating with various peripherals.

Maps tell you where things are inside the computer. In this book we have given you some maps — a general map of memory and a detailed map of the portion of memory that controls the screen. There are other maps that are useful, such as a map of certain reference tables in the early parts of memory.

If you keep in mind codes, pointers, languages, and maps, you will progress more quickly as a computer snooper.

Now for some books that can help you in the future:

1. *Commodore 64 User's Guide.* Besides presenting BASIC and elementary operational matters, this manual presents useful information relating to codes and maps. Also, there are many excellent sample programs throughout the manual that will help you to explore the capabilities of your computer. The information that you will want is scattered throughout the manual. However, the manual is fairly easy to read and is worth going through for whatever useful information it can provide.

2. *Commodore 64 Programmer's Reference Manual.* This is crammed with useful information for snoopers — a large amount of detailed information on codes, pointers, languages, and maps. The bad news is that the manual is addressed to professional engineers and is tough reading. But even if you can understand only a small percentage of the material, a great deal can be learned.

3. *VIC-1541 Single Drive Floppy Disk User's Manual.* This little book is crammed with useful information on the disk drive. Like the *Commodore 64 Programmer's Reference Manual,* it is mostly difficult reading.

4. *COMPUTE! Magazine.* Since the COMMODORE 64 came out on the market, this magazine has been a gathering place for articles by expert computer snoopers. With the background that you have from this book, you will be able to get a lot out of those articles. (Do not be concerned if you don't understand everything in those articles — almost nobody understands everything in those articles!) Try to get your hands on some of the back issues. And be sure to read some of the articles by Jim Butterfield — he is one of the all-time great COMMODORE 64 snoopers.

5. *The Commodore 64 Revealed* by Nick Hampshire. An advanced, well-organized discussion of how the COMMODORE 64 works.

In addition to reading, we encourage you to continue with your own direct investigations of the COMMODORE 64. To help you along, we have added a long chapter of EXTRA TOPICS in Appendix A. Most of these topics are presented in an open-ended manner, so that you can use them as a starting point for your own adventures.

Have fun!

WHO ORDERED THE LARGE WITH EXTRA TOPICS ?

APPENDIX A. EXTRA TOPICS

1. A PROGRAM TO SEARCH THROUGH MEMORY

Suppose you want to find something in memory, but you don't know where it is. For instance, you might want to find all of the error messages that are in memory. One way to do this, of course, would be to use SNOOP and look at each part of memory. This would take a very long time! The following program provides an easier way. It allows you to INPUT a string into a variable S$, and then the program will print all of the locations in memory which match the string.

RUN THIS PROGRAM:

```
1 REM SEARCHER
100 PRINT CHR$(147);
110 INPUT "SEARCH FOR ";S$
200 S1=ASC(S$)
210 LS=LEN(S$)
300 FOR L=0 TO 65535
310 IF PEEK(L)=S1 THEN GOSUB 900
320 NEXT L
330 END
900 T$=""
910 FOR I=1 TO LS
920 P=PEEK(L+I-1)
930 T$=T$+CHR$(P)
940 NEXT I
950 IF S$=T$ THEN PRINT L;
960 RETURN
```

The computer will ask you

```
SEARCH FOR?
```

Let's have it search for all occurrences of the word ERROR.

Type the word ERROR and press RETURN.

The computer will scan through memory from beginning to end, and whenever it finds the word ERROR, it will display the location on the screen. After the program is finished, you can use SNOOP to go back and look at each of the locations in memory.

Searching through memory requires creative thinking. For instance, in this example, we might not pick up all error messages by looking for the word ERROR.

Some error messages may use the abbreviation ERR. Also some error messages use neither of these — they may use the word ILLEGAL instead.

For some searches, you will want to modify this program. If you wanted to search only locations 40000 - 50000, you could change line 300 to

```
300 FOR L=40000 to 50000
```

You might also modify the program to actually print out some of the information that it finds in memory, in addition to the locations of information.

Remember that your program and its variables are held in memory, and this will affect what is "discovered" during the search. When the program scans through the area where the contents of S$ are stored, it will pick up a "match."

One other tricky matter to keep in mind: the computer uses several systems of codes to store information in memory. This particular version of the program is designed to locate information that is in ASCII. If you are looking for information which may be expressed in other codes, you will need to adapt this program.

How SEARCHER Works:

The first few lines of the program set up several important variables:

S$ is the string you are searching for.
S1 is the ASCII value of the first character in S$. If S$ = "ZYGOTE", then S1 is 90, which is the ASCII code for Z.

LS is the length of S$.

The program proceeds to search through memory, byte by byte. The variable L keeps track of what location we are on.

The key line in the program is line 310

```
310 IF PEEK(L)=S1 THEN GOSUB 900
```

This command PEEKs into each byte, to see if it has the value S1. If it does, then this might be the beginning of an occurrence of S$. In that case, the command tells the computer to carry out the subroutine at line 900. That subroutine checks to see if this is an actual occurrence of S$.

2. "NEW" DOES NOT ACTUALLY CLEAR MEMORY.

If you run a program, wait until it's finished, and then type NEW, this seems to clear the program and its variables from memory. If you type LIST, nothing will appear. Also, if you try to PRINT any of the variables from the program, all of their values will have been erased.

Actually, most of the program and variables are still in memory. The main thing NEW does is "cut off" some pointers to the program and the variables. Once these pointers are cut off, the computer no longer "believes" that your program or variables still exist. Most of the residue is still there. We are going to run a program, NEW it, and then look at the residue of the variables.

RUN THIS PROGRAM:

```
1 REM LOTSA
100 DIM A$(255)
200 FOR I=65 TO 69
210 R=255*RND(1)
220 A$(R)=A$(R)+CHR$(I)
230 NEXT I
240 C=C+1:PRINT C:IF C<5000 THEN 200
```

This program builds up a group of variable strings, which consist of random combinations of the letters A, B, C, D, and E.

When the program is finished, type NEW and press RETURN to "clear" the program and variables.

Now load in SNOOP and run it. When it asks you START AT?, type 20000 and press RETURN. You will see a display something like this:

```
START= 20000

          A  B  D  E  E  B  B  A
         65 66 68 69 69 66 66 65
          B  B  A  C  C  C  A  E
         66 66 65 67 67 67 65 69
          D  A  C  D  D  B  B  B
         68 65 67 68 68 66 66 66
          A  A  B  E  E  D  E  C
         65 65 66 69 69 68 69 67
          C  B  E  D  E  D  B  B
         67 66 69 68 69 68 66 66
          B  B  A  E  C  B  E  A
         66 66 65 69 67 66 69 65
          D  B  A  E  D  C  A  A
         68 66 65 69 68 67 65 65
          C  A  D  D  C  E  B  B
         67 65 68 68 67 69 66 66
          E  A  D  E  D  E  E  C
         69 65 68 69 68 69 69 67
          C  C  E  D  A  D  B  A
         67 67 69 68 66 68 66 65

START AT?
```

You are looking at a portion of memory where some of the variable data was stored in the previous program.

3. HOW TO UN-NEW A PROGRAM

We just saw that NEW does not really clean variable data out of memory. The same rule holds for the program itself. When you NEW a program, all of the text for the program actually remains in memory. All that has been eliminated is pointer information. If you POKE the pointer information back in, the program will reappear! The following example will demonstrate how this is done.

Before entering this program, clear your computer completely by turning it off, waiting ten seconds, and turning it back on.

ENTER THIS PROGRAM EXACTLY AS WRITTEN

```
1 REM UNNEW
100 PRINT "HERE IS HOW TO UN-NEW ME:"
110 PRINT
120 PRINT "POKE 2049,";PEEK(2049)
130 PRINT "POKE 2050,";POKE(2050)
```

When you RUN the program, it will display instructions for recovering the program after it has been NEWed. Run the program. You will get a display something like this:

```
HERE IS HOW TO UN-NEW ME:

POKE 2049, 13
POKE 2050, 8
```

(The numbers on your screen may be different from the ones in our illustration.)

The computer has given you instructions for recovering this program after it has been NEWed. Let's NEW the program and then try to recover it.

Type NEW and press RETURN. This will "clear" the program.

Type LIST and press RETURN. Nothing will appear. The program is "gone."

Now let's recover the program. Enter the two POKE commands which appeared on your screen a moment earlier. In our illustration these commands were:

```
POKE 2049, 13
POKE 2050, 8
```

Use the POKE commands which appeared on your screen.

Now type LIST and press RETURN. The program has appeared again. Congratulations! You have un-NEWed a program!

Here's how it worked:

The two values that you POKEd are the first two bytes in the program. They are a pointer. They point to the second line in the program.

When you NEW a program, the computer changes both of these bytes to 0. This signals to the computer that there is no program at all. Actually, the entire text of the program is still right where it was in memory. When you change the 0s back to their original values, your program is "restored."

To un-NEW a program, you have to know what values to change those two 0s back to. In this case it was easy — the program was designed to tell you what the values were supposed to be. In "real life" cases, you will probably have to do some guesswork. The correct value for the pointer will depend on the length of the first line of the program. The longer that line, the higher the value of the pointer at the beginning of the program. If you want to become an expert at un-NEWing programs, play around with a few examples, and you will soon get the hang of it.

WARNING: There are pitfalls in the procedure we have described. This is because there are other pointers relating to your program which may have been altered after you entered the NEW command. Our procedure should enable you to LIST the program, and you will probably be able to run it at least once. But you may get unexpected effects if you use the program more than that. If the program is important to you, you should reenter it.

4. GETTING USED TO NUMBER SYSTEMS AND CODES

Part of becoming a good computer snooper is getting used to base two (binary) and ASCII. The following programs will help you to become familiar and comfortable.

This first program will help you to relate binary and ASCII to base ten.

RUN THIS PROGRAM:

```
1 REM CODES1
100 PRINT CHR$(147);
110 INPUT "ENTER A NUMBER (0-255)";N
120 PRINT "BASE TEN IS: ";N
200 PRINT:PRINT "BINARY IS:   ";
210 V=N
220 T=128
230 FOR I=1 TO 8
240 IF V < T THEN PRINT "0";:GOTO 260
250 PRINT "1";: PRINT "1";: = -V-T
260 T=T/2:NEXT I
290 PRINT
300 PRINT "ASCII SYMBOL IS: ";CHR$(N)
```

The computer will ask you to

```
ENTER A NUMBER (0-255)
```

Type 90 and press RETURN. On your screen you will see

```
BASE TEN IS: 90
BINARY IS:  01011010
ASCII SYMBOL IS: Z
```

The computer is showing you the "counterparts" of 90 in binary and ASCII. Try some other values and keep working with this program until you get used to the relationship between these systems. *WARNING:* There will be a few values which will cause strange behavior on the screen. This is because some ASCII codes, such as do not generate symbols, but are "screen control" codes. For instance, 147 is the screen control code for clearing the screen. PRINT CHR$(147) does not print a symbol, but clears the screen.

The next program deals with the same systems, but in a different way.

RUN THIS PROGRAM:

```
1 REM CODES2
100 PRINT CHR$(147);
110 INPUT "ENTER A CHARACTER ";A$
115 N=ASC(A$)
120 PRINT "BASE TEN IS: ";N
200 PRINT:PRINT "BINARY IS:  ";
210 V=N
220 T=128
230 FOR I=1 TO 8
240 IF V < T THEN PRINT "0";:GOTO 260
250 PRINT "1";:V=V-T
260 T=T/2:NEXT I
290 PRINT
300 PRINT "ASCII SYMBOL IS: ";CHR$(N)
```

When you run this program, it will ask you to enter a character.

Type Z and press RETURN. On your screen you will see

```
ENTER A CHARACTER? Z
BASE TEN IS: 90
BINARY IS:  01011010
ASCII SYMBOL IS: Z
```

The ASCII value (in base ten) of Z is 90. The last three lines show three equivalent forms for the same value.

One other number system you will eventually need for computer snooping is base 16, also known as "hexadecimal" or "hex." Hex is like base ten except that, instead of using just 10 symbols for counting, it uses 16 symbols. Those symbols are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E

The following chart compares a few numbers as expressed in decimal and in hexadecimal.

BASE TEN	HEXADECIMAL
0	0
1	1
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	12
19	14
31	1F
32	20
64	40
128	80
254	FE
255	FF
256	100
65535	FFFF

Hexadecimal notation is often a very convenient method for describing what is in the memory of the computer and for discussing machine language. You will often encounter hexadecimal when you read technical books and articles about computers.

This next program will give you some practice in comparing base ten and hexadecimal.

```
1 REM HEX
100 T$="0123456789ABCDEF"
110 DIM T$(15)
120 FOR I=1 TO 16
130 T$(I-1)=MID$(T$,I,1)
140 NEXT I
200 INPUT "NUMBER (0-255)";N
210 L=INT(T/N)
220 R=N-16*L
300 PRINT "BASE TEN IS: ";
310 PRINT "HEX IS: "; T$(L);T$(R)
```

This program will simply ask you to enter numbers in base ten, and then it will display the hexadecimal equivalent. By the way, lines 100 - 140 simply set up an array T$() where T$(0)="0", T$(1)="1", and so on up to T$(14) ="E" and T$(15)="F".

If you would like to learn more about hexadecimal notation, it is discussed in the *Commodore 64 Programmer's Reference Guide,* as well as most middle-level technical books on the COMMODORE-64.

5. SEE AND HEAR BINARY

All information that the computer handles is expressed in bits — just 1s and 0s. Human beings rarely like to see information in this form, because it is so hard to read and interpret. However, there are times when you want to see information in binary. The following program shows a method for converting any set of bytes to binary. It allows you to INPUT a message into a variable M$. Then it prints the binary version of M$. The program also includes some sound effects, so that you can "hear" binary. (This is similar to what data "sounds" like when it is sent and received by a disk drive or cassette recorder.)

RUN THIS PROGRAM:

```
1 REM INBIN
10 SD=54296:POKE SD,0
20 FOR I=1 TO 10:POKE SD-I,67:NEXT I
100 PRINT CHR$(147);
110 INPUT "MESSAGE ";M$
120 PRINT
200 FOR P=1 TO LEN(M$)
210 V=ASC(MID$(M$,P,1))
300 T=128
310 FOR I=1 TO 8
320 IF V<T THEN PRINT "0";:POKE 54287,40:GOTO 400
330 PRINT "1";:POKE 54287,70:V=V-T
400 GOSUB 900
410 T=T/2
420 NEXT I
500 PRINT " ";
510 POKE 54287,255:GOSUB 900
520 NEXT P
530 END
900 POKE SD,15
910 FOR Z=TO 100:NEXT Z
920 POKE SD,0:RETURN
```

How INBIN Works:

Lines 10 - 20

```
10 SD=296:POKE SD,0
20 FOR I=TO 10:POKE SD-I,67:NEXT I
```

are a standard way of setting up the computer to make a simple sound. This method is discussed in Appendix B, NOTES ON BASIC FOR COMPUTER SNOOPERS. If you like, you can just think of it as a "black box." Once these lines have been placed at the beginning of a program, you can start a tone with

```
POKE SD,15
```

and you can stop the tone with

```
POKE SD,0
```

You can vary the pitch of the tone by POKEing various values to location 54287. The higher the value, the higher the pitch.

Lines 100 - 110

```
100 PRINT CHR$(147);
110 INPUT "MESSAGE ";M$
```

clear the screen, and allow you to INPUT a message into M$.

When the computer asks you to enter a message, type PZAZ and press RETURN. On your screen you will see

```
MESSAGE? PZAZ
01010000 01011010 00100001 01011010
```

The line of 1s and 0s is the binary version of PZAZ. Since each character in the message translates into 8 bits, a space is added after each 8 bits for readability. As the bits are being displayed on the screen, you will hear a series of tones — a high tone for each 1, a low tone for each 0, and a very high tone between each group of 8 bits.

Lines 200 - 520

```
200 FOR P=1 TO LEN(M$)
.
.
.
520 NEXT P
```

is a big FOR - NEXT loop. Each time the computer passes through the loop, it translates one of the characters of M$ into binary.

Line 210

```
210 V=ASC(MID$(M$,P,1))
```

extracts one of the characters from M$ and translates it into its ASCII code. For example, suppose M$ is HI MOM! and P is 2. Then V would be set to 73, which is ASCII for the letter I.

Lines 300 - 330 translate the code in V into binary. Let's trace through how it works, using V = 73 as an example:

V	I	T	Bit	New value for V
73	1	128	0	73
73	2	64	1	9
9	3	32	0	9
9	4	16	0	9
9	5	8	1	1
1	6	4	0	1
1	7	2	0	1
1	8	1	1	0

If you read the 8 bits off the Bit column from top to bottom, you will get 01001001 which is the binary equivalent of 73 (base ten).

Lines 300 - 330 whittle down V to 0 in eight steps. In the first step we try to subtract 128 from V, then 64, then 32, then 16, 8, 4, 2, and 1 successively. We perform a subtraction only when V is at least as big as the number we want to subtract. Depending on whether a subtraction can be performed or not, the program displays a 1 or a 0 on the screen.

The sound effects work like this: Whenever a 1 bit is to be displayed, we have the command

```
POKE 54287,70
```

This sets up SID to produce a high pitch.

Whenever a 0 bit is to be displayed, we have the command

```
POKE 54287,40
```

This sets up SID to produce a lower pitch.

After every 8 bits the following command is executed:

```
POKE 54287,255
```

This sets up SID to produce a very high pitch (you may not even hear it).

The pitches are actually turned on and off by the subroutine starting at line 900

```
900 POKE SD,15
910 FOR Z=1 TO 100:NEXT Z
920 POKE SD,0:RETURN
```

Line 900 turns on the sound, line 910 is a time delay, and line 920 turns the sound off and RETURNs.

6. HOW GRAPHICS DISPLAYS ARE TRACKED IN MEMORY

Earlier in the book we found out how memory tracks what is on the screen when you are displaying text. Your computer can also display graphics and animation. In this section, we will find out a little about what is in memory when this is being done.

The COMMODORE 64 can do graphics displays in several ways. We are going to look at the method which gives the programmer the most precise control over the screen. This method is known as "Standard Bit Map Mode."

The main idea of Standard Bit Map Mode is simple: Your screen is a collection of tiny positions (dots) — 320 across and 200 down, for a total of 64,000 positions. Each position is known as a "pixel" in computer jargon. Each pixel may be any of 16 different colors. Standard Bit Map Mode allows you to control each pixel on the screen independently. This enables you to draw anything you want on the screen, pixel-by-pixel (dot-by- dot). You can draw circles, triangles, curves, spirals, a street map, even a portrait of the family dog. Here is how you do it:

> When you use Standard Bit Map Mode, the computer provides two areas in memory for controlling the screen. These areas are called "screen memory" and "color memory."
>
> Screen memory is 64,000 bits long (i.e., 8000 bytes). Each bit in screen memory stands for one pixel on the screen. If a bit is set to 1, this turns the corresponding pixel on. If a bit is set to 0, then the pixel is off.
>
> Color memory divides the screen into 1000 regions. For each region on the screen, there is one byte in color memory. That byte is used to define the color of pixels that are on in that region, and the color of pixels that are off.

The following program will help you understand the details of how Standard Bit Map Mode works. The program will fill up the screen with on pixels, one pixel at a time. By comparing the program with what happens on the screen, you will find out exactly how Standard Bit Map Mode works.

RUN THIS PROGRAM:

```
1 REM STANBIM
100 A(0)=1
110 FOR I= 1 TO 7
120 A(I)=A(I-1) + 2↑I
130 NEXT I
200 POKE 53265,59:POKE 53272,29
300 FOR I=1024 TO 2047
310 POKE I,33
320 NEXT I
400 FOR I=8192 TO 16191
410 POKE I,0
420 NEXT I
500 FOR I=8192 TO 16191
510 FOR J=0 TO 7
520 POKE I,A(J)
530 FOR Z=1 TO 50:NEXT Z
540 NEXT J
550 NEXT I
```

When you run the program, here is what will happen:

First, the screen will fill up with red and white garbage. During this time, color memory is being set up. It is being set up so that all on pixels will be red, and all off pixels will be white.

Second, the whole screen will gradually become white. During this time, all of the bits in screen memory are being set to 0 (off).

Third, the screen will slowly fill up with red pixels, one pixel at a time. During this time, all of the bits in screen memory are slowly being set to 1 (on).

How STANBIM Works:

Lines 100 - 130

```
100 A(0)= 1
110 FOR I= TO 7
120 A(I)=A(I+1)+2↑I
130 NEXT I
```

set up an array which we will need later on in the program. The values in the array are: 1, 3, 7, 15, 31, 63, 127, 255.

Line 200

```
200 POKE 53265,59:POKE 53272,29
```

tells the computer we will be using Standard Bit Map Mode, and that screen memory will begin at location 8192.

Lines 300 - 320

```
300 FOR I=24 TO 2047
310 POKE I,33
320 NEXT I
```

fill in the appropriate values in color memory. When you are using Standard Bit Map Mode, color memory always is at locations 1024 - 2047.

Lines 400 - 420

```
400 FOR I=92 TO 16191
410 POKE I,0
420 NEXT I
```

set all bits in screen memory to 0. This will make the screen completely white.

Lines 500 - 550

```
500 FOR I=8192 TO 16191
510 FOR J=0 TO 7
520 POKE I,A(J)
530 FOR Z=1 TO 50:NEXT Z
540 NEXT J
550 NEXT I
```

turn on each of the pixels on the screen, one-by-one. Each time line 520 is executed

```
520 POKE I,A(J)
```

one more bit in screen memory is turned on, and this turns on one more pixel on the screen.

Line 530

```
530 FOR Z=1 TO 50:NEXT Z
```

is a time delay which slows up the program enough so that you can watch the pixels turn on, one-by-one.

There are several other graphics modes which we will not discuss in this book. They are:

> Multi-Color Bit Map Mode: This is like Standard Bit Map Mode, except that your capabilities with color are increased, and your ability to draw in detail is somewhat diminished.

Sprite Mode: This enables you to set up little images (trees, fairies, monsters, ducks) which can easily be moved around the screen. Sprite mode is intended mainly for animated graphics.

To learn more about these modes and other graphics techniques, see the suggested reading list in Chapter 16, CONCLUSION.

7. POKING INTO A VARIABLE

Earlier in the book, we used VARLOOK2 to see what a floating point variable looks like in memory when it holds various values. Now we are going to do the reverse; we will create a variable AB, find its location in memory, POKE some values into it, and see what happens.

RUN THIS PROGRAM:

```
1 REM VARPOKE
110 AB=0
100 B= PEEK(45) + 256*PEEK(46) + 2
200 PRINT "AB = ";AB
210 FOR I=B TO B+4
220 PRINT PEEK(I);
230 NEXT I
240 PRINT:PRINT
300 INPUT "POSITION TO POKE (0-4) ";P
310 INPUT "VALUE TO POKE (0-255) ";V
320 POKE B+P,V
330 PRINT:GOTO 200
```

The program will ask you for numbers to POKE into positions 0, 1, 2, 3, and 4. These five positions control what value the variable AB has. Here are a few sets of numbers to try:

		POSITION			
0	1	2	3	4	VALUE OF AB
0	0	0	0	0	0
128	0	0	0	0	.5
128	128	0	0	0	−.5
128	128	128	0	0	−.501953125
128	128	128	128	0	−.501960754
128	128	128	128	128	−.501960784
1	0	0	0	0	2.93873588 E-39
1	1	0	0	0	2.96169475 E-39
1	1	1	0	0	2.96178444 E-39
1	1	1	1	0	2.96178479 E-39
1	1	1	1	1	2.96178479 E-39
1	1	1	1	4	2.96178479 E-39
1	1	1	1	5	2.9617848 E-39

8. RUNNING MACHINE LANGUAGE PROGRAMS

Earlier in the book we looked at an example of a machine language program. It is possible to enter that program into the computer and run it. Let's find out how to do this. The strategy for running a machine language program is as follows:

> As you saw earlier, a machine language program is just a long sequence of bits.
>
> To run a machine language program, the first step is to get all of those bits into memory. To do this, we use the POKE command, which stores data in memory 8 bits (1 byte) at a time. We must be careful to use an area of memory which is not being used for anything else.
>
> Once the machine language program is in memory, it is ready to run. To run it, we use a special command in BASIC called SYS.

The following program takes care of getting our machine language program into memory and starting it:

```
1 REM C64ML
100 FOR L= 40000 TO 40064
110 READ V
120 POKE L,V
130 NEXT L
200 SYS 40000
300 PRINT "SUCCESS"
310 END
800 DATA 160,0,169,32
810 DATA 153,0,4
820 DATA 153,0,5
830 DATA 153,0,6
840 DATA 153,0,7
850 DATA 200,208,241
860 DATA 160,0
870 DATA 169,7,153,0,4,200
880 DATA 169,15,153,0,4,200
890 DATA 169,19,153,0,4,200
900 DATA 169, 8,153,0,4,200
910 DATA 160,0,169,7
920 DATA 153,0,216,200
930 DATA 153,0,216,200
940 DATA 153,0,216,200
950 DATA 153,0,216
960 DATA 96
```

Lines 800 - 960 are the actual machine language program. The numbers in the DATA statements are the values of the individual bytes of the machine language program. For instance, the first two bytes of your machine language program were:

```
10100000  00000000  (base two)
```

These translate into

```
160       0         (base ten)
```

And those are the first two values occurring in the DATA statements (see line 800).

To POKE this data into memory, we use the FOR - NEXT loop in lines 100 - 130:

```
100 FOR L=40000 TO 40064
110 READ V
120 POKE L,V
130 NEXT L
```

This loop READs the numbers in the DATA statements, one-by-one, and then POKEs them into memory starting at location 40000. This is an area of memory which is not usually used for anything else, so it is a "safe" place to put a machine language program.

After this loop is finished, the machine language program is ready to run. To start the machine language program, a SYS command is used:

```
200 SYS 40000
```

This command tells the computer "Run the machine language program which begins at location 40000."

The computer will then run the machine language program. In this example, the program clears the screen and displays the word GOSH in the upper left-hand corner.

When the computer has finished running the machine language program, it will automatically come back to your BASIC program and pick up where it left off. In this case, it will come back to line 300

```
300 PRINT "SUCCESS"
310 END
```

If you try running C64ML, you will be amazed at how fast it runs. It finishes in less than a second, and almost all of that time was actually "set-up" time (the loop in lines 100 - 130).

The method we have described for running machine language programs is tedious, but it is "tried and true." Frankly, there is no really easy way to write machine language programs. That's the reason higher level languages like COBOL and BASIC were invented. However, there is a method which is somewhat easier, and that is to use "assembly" language. Assembly language is a programming language

which essentially has the same set of commands as machine language; however the commands are expressed in numbers and letters rather than 1s and 0s. Assembly language looks like this:

```
5000 LDY #0
5002 LDA #32
5004 STA 1024,Y
.
.
```

Assembly language eliminates about 50% of the headaches of programming in machine language. To find out more about assembly language, a good way to start is to visit a computer store that specializes in software for the COMMODORE 64, and ask to see assemblers for the COMMODORE 64.

9. STOMPING THE OPERATING SYSTEM

There are several reasons why you might want to know how to stomp the operating system and cripple your computer system:

You have aggressive or destructive tendencies!

You want to find out about things you should not do, so you will not do them.

You want to write programs that have powerful security features in them. (One kind of security strategy is to design a program so that if someone misuses it, the program will disable the computer system.)

Whatever your motives, here is an example of a program that stomps the operating system.

RUN THIS PROGRAM:

```
1 REM STOMP
100 PRINT "GOODBYE ... ":PRINT
110 PRINT L
120 POKE L,0
130 L=L+1:GOTO 110
```

GOODBYE ... will appear on the screen. Then you will see numbers displayed on the screen as the program "stomps" on low memory byte-by-byte. When the counting stops, the operating system is dead. Your keyboard will not respond to anything that you type, not even STOP - RESTORE.

Don't worry, you didn't damage anything permanently. All this program does is wipe out some crucial data in memory. To bring your computer system "back to life," turn it off, wait ten seconds, and turn it back on.

How STOMP Works:

The lowest part of memory holds important reference data, and it provides a number of tiny "work areas" for your computer system.

STOMP replaces a portion of the lowest part of memory with 0s. The result is that the computer loses some of the information that it needs to do its work. That is why the counting stops and the keyboard freezes.

10. LISTENING TO YOUR COMMODORE 64 WITH A RADIO

The COMMODORE 64 generates a great deal of radio frequency energy while it is running. The energy can be picked up and heard with an ordinary radio. The sounds that you get will depend on what tasks the computer is performing. Let's listen to what a few programs sound like over the radio.

You will need an FM radio. Place it close by the computer and tune it to 105 MHz. Then

RUN THIS PROGRAM:

```
1 REM RADIO1
2 REM PLAYS RANDOM TONES FOR RADIO
100 A = 100 * RND(1)
110 B = 10 * RND(1)
120 PRINT A,B
200 FOR I = 1 TO A
210 FOR J = 1 TO B
220 NEXT J
230 NEXT I
300 GOTO 100
```

You should hear a series of random tones over the radio, at the rate of one to four different tones per second. To get the best reception on your radio, experiment with various positions near the computer. Also, try other frequencies on the FM or AM band. You probably will find several frequencies which give you reception, with some frequencies better than others. You may get good results in the 105 - 108 MHz range on the FM band.

The kind of sound which is produced is determined by a number of factors:

The execution speed of various commands or sections of the program.

The components of the computer which are involved in executing the program. There are a large number of electrical components in your computer, and different components throw off different kinds of electrical energy.

The frequency at which your radio is tuned. Different radio frequencies pick up different portions of the radio frequencies generated by the computer.

It would be very difficult to analyze in detail how and why the computer produces the particular sounds that it does. However, it is possible to discover many interesting sound qualities by experimenting with various small programs. With experience, you can produce some very pure and interesting sounds, by means of the right programs.

Here are a couple of interesting programs to try:

RUN THIS PROGRAM:

```
1 REM RADIO2
100 INPUT A
110 FOR I=1 TO A:NEXT I
120 GOTO 100
```

This program allows you to hear what a FOR - NEXT loop sounds like. When the program asks you to enter a number, try one of these:

1000
3000
5000
7000
9000

The number that you enter will be the number of repetitions of the FOR - NEXT loop. You will hear different pitches at various times during the loop.

RUN THIS PROGRAM:

```
1 REM RADIO3
100 FOR I = 1 TO 255:A$ = A$ + "Z":NEXT I
110 A$ = "":GOTO 100
```

This builds up a string of Zs, one byte at a time, in the variable A$. When A$ reaches maximum length (255 characters), it is cut back down to zero-length, and the process repeats.

When you listen to this program, you will notice that the tone produced gets lower as the string gets longer.

APPENDIX B. NOTES ON BASIC FOR COMPUTER SNOOPERS

The following notes will summarize some features of BASIC that are especially useful for snooping inside your computer.

1. FUNDAMENTAL TOOLS

The two most important tools for snooping are PEEK and POKE. PEEK allows you to find out the contents of any byte in memory. POKE allows you to alter the contents of any byte in memory (as long as it is not ROM memory).

To PEEK at a location in memory, simply indicate the location you want to PEEK at. For instance,

```
PRINT PEEK(19449)
```

will display the value in location 19449.

To POKE a value into a location, indicate the location and the value you want to store in that location. For instance,

```
POKE 1039, 197
```

will store the value 197 in location 1039.

There are a few areas of memory where PEEKs and POKEs may not work as expected because of special uses of those portions of memory. These are the areas where there is ROM memory, or where there is both "foreground" and "background" memory. These were discussed in Chapter 7, THE MAP OF MEMORY.

Also, bear in mind that some areas of memory are being updated frequently by the computer. So, if you POKE something into one of these areas, the computer may replace your POKE with new values almost immediately.

2. SCREEN DISPLAYS

To analyze what you find in the computer, you need some ways to display information neatly. The following tools are useful for that purpose.

To PRINT several pieces of information on the same line, use PRINT with commas or semicolons. For example, the following program

```
100 A=3:B=79:Q$="LULU"
110 PRINT A,B,Q$
120 PRINT A;B;Q$
```

will give you the following display on your screen

```
3             79            LULU
3  79  LULU
```

The following command will clear your screen and position the cursor in the upper left-hand corner:

```
PRINT CHR$(147);
```

147 is the code for clearing the screen. It does the same work as the SHIFT-CLR combination from your keyboard.

The following command will position the cursor in the upper left-hand corner, without erasing the screen:

```
PRINT CHR$(19);
```

19 is the code for "home." It does the same work as the HOME key on your keyboard.

To indent from the left-hand side of the screen, use the TAB command.

The following example will illustrate these commands

```
100 PRINT CHR$(147);
110 PRINT:PRINT:PRINT"WILLY"
120 PRINT CHR$(19);
130 PRINT TAB(18);"FRED"
```

This program will print the word FRED at the center of the top line of the screen, and WILLY at the beginning of the third line of the screen. Here is what each line of the program does:

Line 100

```
100 PRINT CHR$(147);
```

clears the screen and positions the cursor in the top left-hand corner.

Line 110

```
110 PRINT:PRINT:PRINT"WILLY"
```

prints the word WILLY on the third line.

Line 120

```
120 PRINT CHR$(19);
```

positions the cursor in the top left-hand corner of the screen, without erasing anything.

Line 130

```
130 PRINT TAB(18);"FRED"
```

moves the cursor 18 positions to the right, and prints the word FRED.

The above techniques enable you to display information any way you want on the first few lines of the screen. If you want to display information at other locations on the screen, the following routine can be useful.

```
1 REM ANYWHERE
10 RW$=CHR$(19);
20 FOR I=1 TO 24:RW$=RW$+CHR$(17):NEXT I
100 INPUT R,C
110 PRINT LEFT$(RW$,R);TAB(C);
120 PRINT "X";
```

When you run the program, it will ask you to INPUT values for the variables R and C. The program will then display an X at row R, column C on your screen. This program demonstrates how you can display information at any position you want on the screen.

Lines 10 - 20

```
10 RW$=CHR$(19);
20 FOR I=1 TO 24:RW$=RW$+CHR$(17):NEXT I
```

set up the crucial variable that makes the program work. RW$ consists of the code 19 followed by a series of code 17s. 19 is the code for "home," and 17 is the code for "one line down." By PRINTing portions of this variable, you can move the cursor a certain number of lines down from the top of the screen, to the row which you want.

Line 110 is the line that positions the cursor in the row and column which you want.

```
110 PRINT LEFT$(RW$,R);TAB(C);
```

It prints a certain portion of RW$, just enough to move the cursor down to the row you want. The TAB(C); moves the cursor to the column on the screen you want.

One other way you can position information on the screen, is by making POKEs to screen memory and color memory, as was discussed in Chapter 3.

3. MAKING SOUNDS

When you are analyzing complex information, the use of sound effects can be helpful. The following sample program will show you a simple way to create sound with a minimum of POKEs to SID.

```
10 SD=54296:POKE SD,0
20 FOR I=1 TO 10:POKE SD-I,67:NEXT I
100 INPUT
110 POKE SD,15
120 INPUT
130 POKE SD,0
140 GOTO 100
```

When you run the program, at first you will hear nothing. Press RETURN once and you will hear a tone. Press RETURN again, and the tone will turn off. Press RETURN again and the tone will turn on again. And so on.

Lines 10 and 20

```
10 SD=54296:POKE SD,0
20 FOR I=1 TO 10:POKE SD-I,67:NEXT I
```

set up the program to make sounds. Line 20 fills up the message area for Voice 3 with 67s. This will make a nice sound. Line 10 POKEs a 0 into the "volume control" for SID. This turns the sound off.

The sound is turned on by POKEing a 15 into the volume control. This is done in line 110

```
110 POKE SD,15
```

The volume is turned off again in line 130

```
130 POKE SD,0
```

The above sample program will create a sound with a constant pitch. If you want to vary the pitch, make various POKEs to location 54287, which helps to control the frequency for Voice 3.

4. MATHEMATICAL CALCULATIONS

If you need to round off a number, use INT. For example:

```
100 A = 1235.79
110 PRINT INT(A)
```

This program would print

```
1235
```

If you need the remainder from a division, the following sample program will show you how to obtain this.

```
100 A=80:X=7
110 Q=INT(A/X)
120 R=A - Q*X
130 PRINT Q,R
```

In this example the computer would print the following numbers:

```
11   3
```

If you divide 80 by 7, the quotient is 11 and the remainder is 3.

If you need random numbers, use the RND function. The following way of using it is usually satisfactory. Let's suppose you need a random integer in the range 0 - 99.

```
100 R = INT(100 * RND(1))
```

RND(1) will be a number between 0 and 1. So INT(100 * RND(1)) will be in the range 0 - 99.

Each time RND(1) is used in a program, it generates a new random number. Each time you run a program, the same sequence of random numbers is produced. (The numbers are not produced by chance. Rather they are generated by a special mathematical formula that generates a series of numbers which has the appearance of being random.) If this is not satisfactory, you need to find out about "seeding" the random number generator. This is discussed in your *Commodore 64 User's Guide.*

5. CONVERTING BETWEEN CODES, FORMATS, NUMBER SYSTEMS

The following functions will help you to get back and forth between various codes, formats, and number systems.

The ASC function gives the ASCII value of a character or symbol. For example,

ASC("Z") is 90

The CHR$ function goes in the opposite direction. It converts a number into the ASCII symbol it stands for. For example,

CHR$(90) is Z

Sometimes numbers are represented in string variables or need to be represented in string variables.

To create a string variable version of a number, use STR$. For example,

```
100 N = 237
110 N$ = STR$(N)
120 PRINT N$
```

The output would be 237.

The VAL function goes in the opposite direction.

```
100 N$ = 976
110 N = VAL(N$)
120 PRINT N
```

The output would be 976.

6. RUNNING MACHINE LANGUAGE PROGRAMS

Before you can run a machine language program, you must learn several new concepts. To help you get started, an example is given in Appendix A of how to run a simple machine language program from BASIC.

The only special commands in BASIC that you need for machine language programming are

```
SYS
USR
```

Examples of the use of SYS are in Appendix A. To find out more about SYS and USR, see the *Commodore 64 Programmer's Reference Guide.*

7. MANAGING VARIABLES

If you want to clear all variables from the program, use the CLR command. This will not erase your program, only its variables.

To find out how much free space is available, use the FRE function.

```
PRINT FRE(0)
```

will print the number of free bytes.

9. OTHER COMMANDS TO BE AWARE OF

You may know about these commands already, but to make sure, we will review them briefly.

LEN calculates the length of a string variable.

```
100 V$ = "MAMBO"
110 PRINT LEN(V$)
```

would display the value 5.

MID$ allows you to pick out a substring of a variable string.

```
100 V$ = "ABCDEFGHIJK"
110 PRINT MID$(V$,3,5)
```

would display CDEFG.

To join strings together, use +.

```
100 A$="YOO":B="HOO"
110 C$=A$ + B$
120 PRINT C$
```

would display YOOHOO.

DIM allows you to define a large set of variables in one fell swoop.

```
100 DIM QU(173)
```

creates 174 variables. Their names are: QU(0), QU(1), QU(2), QU(3), and so on up to QU(172), QU(173).

TECHNICAL NOTES

Ch. 2 — Memory. Throughout this book, the concept of memory has been refined several times.

In its most general sense, memory includes all capabilities in the computer for storing data. Types of memory would include: registers, RAM and ROM, disk and cassette storage, and special purpose memory devices (e.g., the RAM and ROM memory which resides in the disk drive unit).

Throughout most of the book, we were concerned primarily with that memory which is accessible via the PEEK and POKE commands. The PEEK command and the POKE command are each able to address a total of 64K bytes of memory at any time. There are some limitations on the abilities of PEEK and POKE. Also, there are some techniques which can extend the range of PEEK and POKE beyond 64K bytes.

PEEK and POKE can access only RAM and ROM memory. Later in the book, we discuss other kinds of memory, especially disk memory and register memory.

Ch. 3 — Invisible characters on the screen. If a certain area of the screen is blank, and you POKE a character into that area, you will not see anything happen on the screen. This is because the computer has assigned the same color to the character as the background of the screen. The character is "there" on the screen, but it is invisible, because it has the same color as the background.

When you POKE an appropriate color code into color memory, this assigns a contrasting color to the character, and it then becomes visible.

Ch. 4 — Shape tables. The shape table starts at location 53248, and extends for 4096 bytes to location 57343. Since each character requires 8 bytes in the table, there are a total of 512 characters represented in the table. The first 64 characters in the table represent the characters which are produced by the keyboard in its normal mode. The remaining characters in the table are for lower case, graphics mode, and reversed characters. In Ch. 4 and Ch. 5, we are only concerned with the first 64 characters (256 bytes) in the shape table. However, any of our sample programs may be modified in a straghtforward manner to encompass a complete 512 character shape table.

Ch. 5 — Binary notation is hard to read. To reduce the difficulties of reading binary notation, computer people usually replace it with "hexadecimal" ("hex") notation. In hex notation, each group of 4 binary digits is represented by 0, 1, 2,..., 9, A, B, C, D, E, F.

BINARY	HEX
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

For instance, the binary number 11010110 is represented as D6 in hex. Hex notation is much easier to read than binary notation. Hex notation is used widely in technical articles and books on computers.

Ch. 7 — 65,536 = 64K. The size of a computer's memory is usually measured in units of "K-bytes". "1K" is defined as 1024 bytes. So 64K is equal to

64 * 1024

which is 65,536 bytes.

"K" is sometimes loosely used to mean 1000 bytes. Strictly speaking however, it means 1024 bytes (1024 is 2 to the tenth power).

Ch. 7 — The computer is an alien form of lower intelligence. Some people would object to speaking of a computer as having intelligence.

When we speak of the computer as being a form of lower intelligence, we mean that the computer is capable of some kinds of behavior that are ordinarily thought of as requiring some intelligence. For instance, the computer is capable of responding correctly to sets of instructions in artificial languages such as BASIC. The computer is also capable of making generally appropriate responses to problem situations. For instance, if you enter the command

```
PRIMT 21 * 19
```

the computer will give you an error message, which will indicate that it cannot understand your command.

Some people would say of these examples that while they show that a computer is capable of complex and useful behavior, they do not show that a computer has any kind of "intelligence." This brings up a difficult issue of what intelligence is.

One view of intelligence equates intelligence with the ability to perform tasks which ordinarily require human intelligence. Under this view, a computer would qualify as having some intelligence, although not as much as human beings. Computers can perform such tasks as carry out commands in an artificial language, or play a competent game of chess. We think of these tasks as ordinarily requiring some human intelligence. So a computer qualifies as being intelligent to some degree.

Another view of intelligence is that it involves more than just the ability to perform tasks such as carrying out commands in an artificial language, or playing chess. Under this view, it is also important how the computer "perceives" or "understands" the situation that it is in. Importance is placed on not just what the computer does, but also how it does it, or perhaps on what goes on "inside" the computer. Under this view, the computer might be imagined as just a chunk of machinery which "goes through the motions" of acting intelligent, but actually is not capable of any intelligence of its own.

Both of these views can be elaborated into a number of different versions. All that we have done here is to sketch two views that are possible. It will not be necessary in this book to elaborate these issues further or to attempt to resolve them.

Our beliefs about the "intelligence" of the COMMODORE 64 may be summarized as follows:

> The COMMODORE 64 is capable of perfcrming some tasks which ordinarily would require considerable human intelligence and effort.
>
> The computer often uses methods for carrying out tasks which seem primitive, by human standards.

We also believe that the computer is an "alien" form of intelligence, in that its methods for performing tasks are often radically different from human processes, and often incongenial to human understanding.

Ch. 14 — 6510 machine language. The COMMODORE 64 is based on the MOS 6510 microprocessor, which is a slight variant of the popular 6502 microprocessor. The machine language for the 6510 microprocessor and the 6502 microprocessor is identical. Because of this, it is sometimes said that the COMMODORE 64 is programmable in "6502 machine language."

Ch. 14 — Assembly language. Writing programs in machine language is hard work. A somewhat easier way is to write programs in "assembly language" instead. Assembly language is a programming language which essentially has the same set of commands as machine language; however, the commands are expressed in numbers and letters rather than 1s and 0s. See Part 8 of Appendix A — EXTRA TOPICS.

APPENDIX C.

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
00000000	0	0	00	
00000001	1	1	01	
00000010	2	2	02	
00000011	3	3	03	
00000100	4	4	04	
00000101	5	5	05	
00000110	6	6	06	
00000111	7	7	07	
00001000	10	8	08	
00001001	11	9	09	
00001010	12	10	0A	
00001011	13	11	0B	
00001100	14	12	0C	
00001101	15	13	0D	
00001110	16	14	0E	
00001111	17	15	0F	
00010000	20	16	10	
00010001	21	17	11	
00010010	22	18	12	
00010011	23	19	13	
00010100	24	20	14	
00010101	25	21	15	
00010110	26	22	16	
00010111	27	23	17	
00011000	30	24	18	
00011001	31	25	19	
00011010	32	26	1A	
00011011	33	27	1B	
00011100	34	28	1C	
00011101	35	29	1D	
00011110	36	30	1E	
00011111	37	31	1F	
00100000	40	32	20	Space
00100001	41	33	21	!
00100010	42	34	22	"
00100011	43	35	23	#
00100100	44	36	24	$
00100101	45	37	25	%

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
00100110	46	38	26	&
00100111	47	39	27	'
00101000	50	40	28	(
00101001	51	41	29	)
00101010	52	42	2A	*
00101011	53	43	2B	+
00101100	54	44	2C	,
00101101	55	45	2D	-
00101110	56	46	2E	.
00101111	57	47	2F	/
00110000	60	48	30	0
00110001	61	49	31	1
00110010	62	50	32	2
00110011	63	51	33	3
00110100	64	52	34	4
00110101	65	53	35	5
00110110	66	54	36	6
00110111	67	55	37	7
00111000	70	56	38	8
00111001	71	57	39	9
00111010	72	58	3A	:
00111011	73	59	3B	;
00111100	74	60	3C	<
00111101	75	61	3D	=
00111110	76	62	3E	>
00111111	77	63	3F	?
01000000	100	64	40	@
01000001	101	65	41	A
01000010	102	66	42	B
01000011	103	67	43	C
01000100	104	68	44	D
01000101	105	69	45	E
01000110	106	70	46	F
01000111	107	71	47	G
01001000	110	72	48	H
01001001	111	73	49	I
01001010	112	74	4A	J
01001011	113	75	4B	K
01001100	114	76	4C	L
01001101	115	77	4D	M
01001110	116	78	4E	N
01001111	117	79	4F	O
01010000	120	80	50	P
01010001	121	81	51	Q

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
01010010	122	82	52	R
01010011	123	83	53	S
01010100	124	84	54	T
01010101	125	85	55	U
01010110	126	86	56	V
01010111	127	87	57	W
01011000	130	88	58	X
01011001	131	89	59	Y
01011010	132	90	5A	Z
01011011	133	91	5B	[
01011100	134	92	5C	£
01011101	135	93	5D	]
01011110	136	94	5E	↑
01011111	137	95	5F	←
01100000	140	96	60	
01100001	141	97	61	♠
01100010	142	98	62	
01100011	143	99	63	
01100100	144	100	64	
01100101	145	101	65	
01100110	146	102	66	
01100111	147	103	67	
01101000	150	104	68	
01101001	151	105	69	
01101010	152	106	6A	
01101011	153	107	6B	
01101100	154	108	6C	
01101101	155	109	6D	
01101110	156	110	6E	
01101111	157	111	6F	
01110000	160	112	70	
01110001	161	113	71	●
01110010	162	114	72	
01110011	163	115	73	♥
01110100	164	116	74	
01110101	165	117	75	
01110110	166	118	76	☒
01110111	167	119	77	
01111000	170	120	78	♣
01111001	171	121	79	
01111010	172	122	7A	♦
01111011	173	123	7B	⊞
01111100	174	124	7C	
01111101	175	125	7D	

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
01111110	176	126	7E	
01111111	177	127	7F	
10000000	200	128	80	
10000001	201	129	81	
10000010	202	130	82	
10000011	203	131	83	
10000100	204	132	84	
10000101	205	133	85	
10000110	206	134	86	
10000111	207	135	87	
10001000	210	136	88	
10001001	211	137	89	
10001010	212	138	8A	
10001011	213	139	8B	
10001100	214	140	8C	
10001101	215	141	8D	
10001110	216	142	8E	
10001111	217	143	8F	
10010000	220	144	90	
10010001	221	145	91	
10010010	222	146	92	
10010011	223	147	93	
10010100	224	148	94	
10010101	225	149	95	
10010110	226	150	96	
10010111	227	151	97	
10011000	230	152	98	
10011001	231	153	99	
10011010	232	154	9A	
10011011	233	155	9B	
10011100	234	156	9C	
10011101	235	157	9D	
10011110	236	158	9E	
10011111	237	159	9F	
10100000	240	160	A0	
10100001	241	161	A1	
10100010	242	162	A2	
10100011	243	163	A3	
10100100	244	164	A4	
10100101	245	165	A5	
10100110	246	166	A6	
10100111	247	167	A7	
10101000	250	168	A8	
10101001	251	169	A9	

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
10101010	252	170	AA	
10101011	253	171	AB	
10101100	254	172	AC	
10101101	255	173	AD	
10101110	256	174	AE	
10101111	257	175	AF	
10110000	260	176	B0	
10110001	261	177	B1	
10110010	262	178	B2	
10110011	263	179	B3	
10110100	264	180	B4	
10110101	265	181	B5	
10110110	266	182	B6	
10110111	267	183	B7	
10111000	270	184	B8	
10111001	271	185	B9	
10111010	272	186	BA	
10111011	273	187	BB	
10111100	274	188	BC	
10111101	275	189	BD	
10111110	276	190	BE	
10111111	277	191	BF	
11000000	300	192	C0	
11000001	301	193	C1	
11000010	302	194	C2	
11000011	303	195	C3	
11000100	304	196	C4	
11000101	305	197	C5	
11000110	306	198	C6	
11000111	307	199	C7	
11001000	310	200	C8	
11001001	311	201	C9	
11001010	312	202	CA	
11001011	313	203	CB	
11001100	314	204	CC	
11001101	315	205	CD	
11001110	316	206	CE	
11001111	317	207	CF	
11010000	320	208	D0	
11010001	321	209	D1	
11010010	322	210	D2	
11010011	323	211	D3	
11010100	324	212	D4	
11010101	325	213	D5	

CODES 192-223 SAME AS 96-127

CODES 224-254 SAME AS 160-190

CODE 255 SAME AS 126

ASCII Codes

BASE TWO (BINARY)	BASE EIGHT (OCTAL)	BASE TEN (DECIMAL)	BASE SIXTEEN (HEXADECIMAL)	PRINTED ASCII
11010110	326	214	D6	
11010111	327	215	D7	
11011000	330	216	D8	
11011001	331	217	D9	
11011010	332	218	DA	
11011011	333	219	DB	
11011100	334	220	DC	
11011101	335	221	DD	
11011110	336	222	DE	
11011111	337	223	DF	
11100000	340	224	E0	
11100001	341	225	E1	
11100010	342	226	E2	
11100011	343	227	E3	
11100100	344	228	E4	
11100101	345	229	E5	
11100110	346	230	E6	
11100111	347	231	E7	
11101000	350	232	E8	
11101001	351	233	E9	
11101010	352	234	EA	
11101011	353	235	EB	
11101100	354	236	EC	
11101101	355	237	ED	
11101110	356	238	EE	
11101111	357	239	EF	
11110000	360	240	F0	
11110001	361	241	F1	
11110010	362	242	F2	
11110011	363	243	F3	
11110100	364	244	F4	
11110101	365	245	F5	
11110110	366	246	F6	
11110111	367	247	F7	
11111000	370	248	F8	
11111001	371	249	F9	
11111010	372	250	FA	
11111011	373	251	FB	
11111100	374	252	FC	
11111101	375	253	FD	
11111110	376	254	FE	
11111111	377	255	FF	

KEYBOARD SCAN CODES

SCAN CODE	KEY
0	INST/DEL
1	RETURN
2	CRSR L/R
3	F7
4	F1
5	FE
6	F5
7	CRSR U/D
8	3
9	W
10	A
11	4
12	Z
13	S
14	E
15	
16	5
17	R
18	D
19	6
20	C
21	F
22	L
23	X
24	7
25	Y
26	G
27	8
28	B
29	H
30	U
31	V
32	9
33	I
34	J
35	0
36	M
37	K
38	O
39	N
40	+
41	P
42	L
43	-
44	.
45	:
46	@

SCAN CODE	KEY
47	,
48	PD
49	*
50	;
51	CLR/HOME
52	
53	=
54	UP ARROW
55	/
56	1
57	LEFT ARROW
58	
59	2
60	SPACE BAR
61	
62	Q
63	RUN/STOP

COLOR CODES

CODE	COLOR
0	BLACK
1	WHITE
2	RED
3	CYAN
4	PURPLE
5	GREEN
6	BLUE
7	YELLOW
8	ORANGE
9	BROWN
10	LIGHT RED
11	GRAY 1
12	GRAY 2
13	LIGHT GREEN
14	LIGHT BLUE
15	GRAY 3

APPENDIX D.

```
1 REM PEEKDEMO
2 REM DEMO OF PEEK COMMAND
100 PRINT CHR$(147);
110 PRINT "BAD-CAT !!"
200 PRINT:PRINT
210 FOR I = 1024 TO 1033
220 P = PEEK(I)
230 PRINT P;
240 NEXT I

READY.
```

```
1 REM POKEDEMO1
2 REM DEMO OF POKE COMMAND
100 PRINT CHR$(147);
110 PRINT "BAD-CAT !!"
200 REM LINE 210 CAUSES A 5 SECOND PAUSE
210 FOR X = 1 TO 3500:NEXT X
300 POKE 1024,26

READY.
```

```
1 REM POKEDEMO2
100 SP$="                         "
110 PRINT CHR$(147);
120 PRINT "BAD-CAT !!"
200 PRINT CHR$(19);TAB(200);
210 PRINT SP$:PRINT SP$
220 PRINT CHR$(19);TAB(200);
230 INPUT "LOCATION";L
240 INPUT "CODE";C
250 POKE L,C:GOTO 200

READY.
```

```
1 REM SHOWALL
100 PRINT CHR$(147);
200 FOR I = 0 TO 255
210 POKE 1024+I,I
```

```
220 NEXT I
300 FOR C = 0 TO 15
310 FOR I = 55296 TO 55551
320 POKE I,C
330 NEXT I
340 NEXT C

READY.
```

```
1 REM HOTBAN
100 B=1024
110 INPUT "MESSAGE";M$
120 M$=M$ + " "
130 L=LEN(M$)
200 PRINT CHR$(147);M$
210 FOR I= 0 TO 999
220 P=PEEK(B+M)
230 POKE B+I,P
240 M=M+1:IF M=L THEN M=0
250 NEXT I
300 FOR I=55296 TO 56295
310 R=INT(16*RND(1))
320 POKE I,R
330 NEXT I
400 GOTO 300

READY.
```

```
1 REM KBDSTAT
100 PRINT PEEK(197),PEEK(653)
110 GOTO 100

READY.
```

```
1 REM ZOUNDS1
100 FOR I=54272 TO 54296
110 R = INT(256 * RND(1))
120 POKE I,R
130 NEXT I
140 GOTO 100

READY.
```

```
1 REM ZOUNDS2
10 POKE 54296,15
100 FOR I=54272 TO 54278
110 R = INT(100 * RND(1))
120 POKE I,R
130 NEXT I
140 GOTO 100

READY.
```

```
1 REM ZOUNDS3
10 POKE 54296,15
100 FOR I=54272 TO 54278
110 R = INT(100 * RND(1))
120 POKE I,R
125 PRINT R;
130 NEXT I
135 INPUT X$
140 GOTO 100

READY.
```

```
1 REM CLOCK
100 A=PEEK(160)
110 B=PEEK(161)
120 C=PEEK(162)
130 PRINT A;B;C
140 GOTO 100↑
150 GOTO 100↑

READY.
```

```
1 REM BARCODES
100 PRINT CHR$(147);"FONE HOME"
200 FOR C=0 TO 63
210 FOR D=0 TO 7
220 POKE 12288 + 8*C +D, C
230 NEXT D
240 NEXT C
300 POKE 53272, 29

READY.
```

```
1 REM LIGHTNING
100 PRINT:PRINT"FONE HOME"
200 FOR C=0 TO 63
210 FOR D=0 TO 7
220 POKE 12288+8*C+D,C
230 NEXT D
240 NEXT C
300 POKE 53272,29
400 FOR I = 12288 TO 12295
410 READ C
420 POKE I,C
430 NEXT I
440 PRINT "@@@@@@@@@"
500 DATA  3,6,12,30,3,6,12,24

READY.
```

```
1 REM COPYSHAPES
100 POKE 56334,0:POKE 1,51
200 FOR I=0 TO 511
210 P=PEEK(53248 + I)
220 POKE 12288+I, P
230 NEXT I
300 POKE 1,55:POKE 56334,1
310 POKE 53272,29

READY.
```

```
1 REM WOM
100 POKE 56334,0:POKE 1,51
200 FOR I=0 TO 511
205 D=D+2:IF D>16 THEN D=2
210 P=PEEK(53248 + I)
220 POKE 12297+I-D, P
230 NEXT I
300 POKE 1,55:POKE 56334,1
310 POKE 53272,29

READY.
```

```
1 REM SNOOP
100 INPUT "START AT";B
200 PRINT CHR$(147);"START=";B
```

```
300 FOR LI=1 TO 10
400 FOR I=B TO B+7
410 P=PEEK(I)
420 IF P<32 OR P>95 THEN P=32
430 PRINT "   ";CHR$(P);
440 NEXT I
450 PRINT
500 FOR I=B TO B+7
510 P=PEEK(I)
520 P$=RIGHT$("  "+STR$(P),4)
530 PRINT P$;
540 NEXT I
550 PRINT
600 B=B+8
610 NEXT LI
620 GOTO 100

READY.
```

```
1 REM REWRITE
100 PRINT "JOHN HAD"
110 PRINT "GREAT BIG"
120 PRINT "WATERPROOF"
130 PRINT "BOOTS ON"
200 FOR I = 2048 TO 2150
210 IF PEEK(I) = 153 THEN POKE I,143
220 NEXT I

READY.
```

```
1 REM INVIS
100 PRINT "HI MOM!"
110 PRINT "HERE I AM!!"
200 POKE 2050,3

READY.
```

```
1 REM POINTERS
100 P=PEEK(43) + 256*PEEK(44)
110 PRINT "BASIC PGM STARTS AT";P
200 P=PEEK(45) + 256*PEEK(46)
```

```
210 PRINT "VARIABLES START AT";P
300 P=PEEK(55) + 256*PEEK(56)
310 PRINT "BASIC REGION ENDS AT";P

READY.
```

```
1 REM VARLOOK1
100 INPUT "VALUE FOR AB%";AB%
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT:GOTO 100

READY.
```

```
1 REM VARLOOK2
100 INPUT "VALUE FOR AB";AB
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT:GOTO 100

READY.
```

```
1 REM VARLOOK3
100 INPUT "VALUE FOR AB$";AB$
200 B=PEEK(45) + 256*PEEK(46)
300 FOR I=B TO B+6
310 PRINT PEEK(I);
320 NEXT I
400 PRINT:PRINT
500 C=PEEK(B+2)
510 B1=PEEK(B+3) + 256*PEEK(B+4)
520 PRINT "DATA BLOCK STARTS AT";B1
600 FOR I=B1 TO B1+C-1
610 PRINT PEEK(I);
620 NEXT I
700 PRINT:PRINT:PRINT:GOTO 100

READY.
```

```
1 REM VARWHERE
100 DIM A$(255)
110 FOR I=1 TO 255
120 P$=P$+CHR$(I)
130 NEXT I
200 R1=INT(256*RND(1))
210 R2=100 + INT(150*RND(1))
220 A$(R1)=LEFT$(P$,R2)
230 PRINT PEEK(51) + 256*PEEK(52)
240 GOTO 200

READY.
```

```
1 REM BASEBALL1
100 OPEN 2,8,2,"@:CHAMPS,S,W"
200 FOR I=1 TO 3
210 READ A$
220 PRINT#2,I;A$
230 NEXT I
300 CLOSE 2
400 DATA "RUTH","MANTLE","KOUFAX"

READY.
```

```
1 REM BASEBALL2
100 OPEN 2,8,2,"CHAMPS,S,R"
200 GET#2,A$
210 PRINT ASC(A$),A$
220 IF ST=0 THEN 200
300 CLOSE 2

READY.
```

```
1 REM MAPS
100 OPEN 2,8,2,"$,S,R"
200 GET#2,A$
210 IF A$ = "" THEN PRINT N,0:GOTO 230
220 PRINT N, ASC(A$)
230 N=N+1
240 C=C+1:IF C=20 THEN C=0:INPUT X$
250 IF ST=0 THEN 200
300 CLOSE 2

READY.
```

```
1 REM SOUNDLAB
100 POKE 54296,15
110 B=54272
200 LF=0:HF=250
210 LP=0:HP=15
220 QQ=16
230 A=0:D=1
240 S=0:R=9
300 T1=1:T2=500
600 AD=16*A + D
610 SR=16*S + R
700 POKE B,LF:POKE B+1,HF
710 POKE B+2,LP:POKE B+3,HP
720 POKE B+5,AD:POKE B+6,SR
800 POKE B+4,QQ+1
810 FOR Z=1 TO T1:NEXT Z
820 POKE B+4,QQ
830 FOR Z=1 TO T2:NEXT Z
900 GOTO 700

READY.
```

```
1 REM SEARCHER
100 PRINT CHR$(147);
110 INPUT "SEARCH FOR";S$
200 S1=ASC(S$)
210 LS=LEN(S$)
300 FOR L=0 TO 65535
310 IF PEEK(L)=S1 THEN GOSUB 900
320 NEXT L
330 END
900 T$=""
910 FOR I=1 TO LS
920 P=PEEK(L+I-1)
930 T$=T$+CHR$(P)
940 NEXT I
950 IF S$=T$ THEN PRINT L;
960 RETURN

READY.
```

```
1 REM LOTSA
100 DIM A$(255)
200 FOR I=65 TO 69
```

```
210 R=255*RND(1)
220 A$(R)=A$(R)+CHR$(I)
230 NEXT I
240 C=C+1:PRINT C:IF C<5000 THEN 200

READY.
```

```
1 REM UNNEW
100 PRINT"HERE IS HOW TO UN-NEW ME:"
110 PRINT
120 PRINT "POKE 2049,";PEEK(2049)
130 PRINT "POKE 2050,";PEEK(2050)

READY.
```

```
1 REM CODES1
100 PRINT CHR$(147);
110 INPUT "ENTER A NUMBER (0-255)";N
120 PRINT "BASE TEN IS: ";N
200 PRINT:PRINT "BINARY IS:    ";
210 V=N
220 T=128
230 FOR I=1 TO 8
240 IF V < T THEN PRINT "0";:GOTO 260
250 PRINT "1";:V=V-T
260 T=T/2:NEXT I
290 PRINT
300 PRINT "ASCII SYMBOL IS: ";CHR$(N)
400 PRINT:PRINT:GOTO 110

READY.
```

```
1 REM CODES2
100 PRINT CHR$(147);
110 INPUT "ENTER A CHARACTER ";A$
115 N=ASC(A$)
120 PRINT "BASE TEN IS: ";N
200 PRINT:PRINT "BINARY IS:    ";
210 V=N
220 T=128
230 FOR I=1 TO 8
```

```
240 IF V < T THEN PRINT "0";:GOTO 260
250 PRINT "1";:V=V-T
260 T=T/2:NEXT I
290 PRINT
300 PRINT "ASCII SYMBOL IS: ";CHR$(N)
400 PRINT:PRINT:GOTO 110

READY.
```

```
1 REM HEX
100 T$="0123456789ABCDEF"
110 DIM T$(15)
120 FOR I=1 TO 16
130 T$(I-1)=MID$(T$,I,1)
140 NEXT I
200 INPUT "NUMBER (0-255)";N
210 L=INT(N/16)
220 R=N-16*L
300 PRINT "BASE TEN IS: ";N
310 PRINT "HEX IS: ";T$(L);T$(R)
400 PRINT:GOTO 200

READY.
```

```
1 REM INBIN
10 SD=54296:POKE SD,0
20 FOR I=1 TO 10:POKE SD-I,67:NEXT I
100 PRINT CHR$(147);
110 INPUT "MESSAGE ";M$
120 PRINT
200 FOR P=1 TO LEN(M$)
210 V=ASC(MID$(M$,P,1))
300 T=128
310 FOR I=1 TO 8
320 IF V<T THEN PRINT "0";:POKE 54287,40:GOTO 400
330 PRINT "1";:POKE 54287,70:V=V-T
400 GOSUB 900
410 T=T/2
420 NEXT I
500 PRINT " ";
510 POKE 54287,255
520 NEXT P
530 END
900 POKE SD,15
```

```
910 FOR Z=1 TO 100:NEXT Z
920 POKE SD,0:RETURN

READY.
```

```
1 REM STANBIM
100 A(0)=1
110 FOR I=1 TO 7
120 A(I)=A(I-1) + 2↑I
130 NEXT I
200 POKE 53265,59:POKE 53272,29
300 FOR I=1024 TO 2047
320 POKE I,33
330 NEXT I
400 FOR I=8192 TO 16191
410 POKE I,0
420 NEXT I
500 FOR I=8192 TO 16191
510 FOR J=0 TO 7
520 POKE I,A(J)
530 FOR Z=1 TO 50:NEXT Z
540 NEXT J
550 NEXT I

READY.
```

```
1 REM VARPOKE
100 AB=0
110 B=PEEK(45) + 256*PEEK(46) + 2
200 PRINT "AB = ";AB
210 FOR I=B TO B+4
220 PRINT PEEK(I);
230 NEXT I
240 PRINT:PRINT
300 INPUT "POSITION TO POKE (0-4) ";P
310 INPUT "VALUE TO POKE (0-255) ";V
320 POKE B+P,V
330 PRINT:GOTO 200

READY.
```

```
1 REM C64ML
100 FOR L=40000 TO 40064
110 READ V
```

```
120 POKE L,V
130 NEXT L
200 SYS 40000
300 PRINT "SUCESS"
400 END
800 DATA 160,0,169,32
810 DATA 153,0,4
820 DATA 153,0,5
830 DATA 153,0,6
840 DATA 153,0,7
850 DATA 200,208,241
860 DATA 160,0
870 DATA 169,7,153,0,4,200
880 DATA 169,15,153,0,4,200
890 DATA 169,19,153,0,4,200
900 DATA 169, 8,153,0,4,200
910 DATA 160,0,169,7
920 DATA 153,0,216,200
930 DATA 153,0,216,200
940 DATA 153,0,216,200
950 DATA 153,0,216
960 DATA 96

READY.
```

```
1 REM STOMP
100 PRINT "GOODBYE ...":PRINT
110 PRINT L
120 POKE L,0
130 L=L+1:GOTO 110

READY.
```

```
1 REM RADIO1
2 REM PLAYS RANDOM NOTES FOR RADIO
100 A = 100 * RND(1)
110 B = 10 * RND(1)
120 PRINT A,B
200 FOR I = 1 TO A
210 FOR J = 1 TO B
220 NEXT J
230 NEXT I
300 GOTO 100

READY.
```

```
1 REM RADIO2
100 INPUT A
110 FOR I=1 TO A:NEXT I
120 GOTO 100

READY.
```

```
1 REM RADIO3
100 FOR I=1 TO 255:A$=A$ + "Z":NEXT I
110 A$="":GOTO 100

READY.
```

GLOSSARY

ADDRESSABLE MEMORY: ROM and RAM memory which may be accessed with the PEEK command. Contrast this with other parts of the computer which may be also called memory: registers, disk memory, special purpose memory chips in the disk drive, etc.

ASCII: A code system which assigns a meaning to each of the byte values 0 - 127. For instance, Q is assigned to 81, and ! is assigned to 33. ASCII stands for American Standard Code for Information Interchange.

ASSEMBLY LANGUAGE: A version of machine language which is more convenient for human beings to work with than machine language. The vocabulary of commands is the same, but the 1s and 0s of machine language are replaced by more meaningful symbols.

BAM (BLOCK AVAILABILITY MAP): A "map" on each disk which shows which sectors (blocks) are used, and which are unused.

BASE: Refers to a number system — e.g., Base Two (Binary), Base Eight (Octal), Base Ten (Decimal), or Base Sixteen (Hexadecimal or Hex).

BINARY: Base Two. In Base Two, the numbers from zero to eight are: 0, 1, 10, 11, 100, 101, 110, 111, 1000 .

BIT: The smallest unit of information processed by the computer. A bit may have only two possible values, usually expressed as 1 or 0, or on or off. A set of 8 bits is called a byte.

BOOT: The process that the computer must go through when it is started, before it is ready to respond to any of your commands. The process includes setting up certain reference tables in memory, clearing the screen, and displaying the initial greeting.

BUFFER: A temporary holding area for information before it reaches its final destination. For instance, when information is sent to the disk, the computer will sometimes accumulate a quantity of the information in a memory buffer, until a large enough amount has been obtained. And then all of the information in the buffer will be stored on the disk at the same time.

BUS: A communications line which runs through the computer. All major components are attached to the bus. The components may send signals to each other through the bus.

BYTE: A group of 8 bits. A byte may range in value from 0 to 255 (in binary: 00000000 to 11111111).

CHIP: Highly miniaturized electronic circuitry, compressed onto a tiny wafer of silicon (or similar material). Chips are usually packaged in a black protective housing, and the entire unit is often referred to as a chip. Examples of chips are the MOS 6510 microprocessor, and SID (the chip that makes sound and music).

COMMAND INTERPRETER (COMMAND PROCESSOR): A program or set of routines that translates a human command into machine language commands which the computer can understand. For instance, when you issue the command `PRINT 7 * (A + Q(3))`, the computer cannot directly understand that command. However a command interpreter translates that command into machine language which can be understood by the computer.

COMPILER: A program which translates a program in a high-level language, such as BASIC or Pascal, into machine language.

CPU (CENTRAL PROCESSING UNIT): The 6510 microprocessor, which is the primary "thinking organ" of your computer.

CRT: The screen or monitor on which the computer displays information. CRT stands for Cathode Ray Tube.

DIRECTORY: The table of contents for a disk — the list of files on a disk.

DISK: A disk is used for long-term storage of programs and data files. The surface of a disk is composed of the same material as is used in a cassette tape. The circular shape of a disk increases reliability and performance, as compared to a cassette tape. A flexible plastic disk is known as a diskette or floppy disk. An inflexible, high-storage-capacity metal disk is known as a hard disk.

DISK CONTROLLER: The Commodore disk drive has built into it a microprocessor, memory chips, and disk operating system software, which help it to store, retrieve, and manage data on disks. The term disk controller refers to the collection of chips and operating system software built into the disk drive.

DUMP: A printout or screen display of the exact contents of an entire file.

FIELD: Short sections of stored data. In the case of a mailing list file, examples of fields would be: Name, Address, City, State, and Zip. In the case of an inventory file, examples of fields might include: part number, description, quantity-on-hand, cost, and selling price.

FILE: A collection of data which is stored on the disk under one name. The name of each file is stored in the disk directory.

MEGABYTE: One million bytes, or 1000 K bytes.

HARD DISK: See DISK.

HEX (HEXADECIMAL): Base 16. In Hex, the numbers from zero through 20 are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14.

INTERPRETER: A program or set of routines that enables the computer to process commands in a high-level language such as BASIC. For each command in the high-level language, the interpreter translates the command into machine language, tells the computer to execute it, and then goes on to the next high-level command. Contrast this with a compiler, which translates an entire high-level program into machine language, all at one time, then executes the machine language version of the program.

K: Strictly speaking, K means exactly 1024 bytes (1024 is 2 to the tenth power). But K is loosely used to mean 1000 bytes. So for instance, 32K would mean 32 thousand bytes.

KERNAL: This is a set of machine language routines which perform fundamental tasks, such as controlling the position of the cursor, updating the internal clock (locations 160 - 162), and interrupting your program when you press STOP-RESTORE.

MACHINE LANGUAGE: An extremely primitive language of 1s and 0s which is the only language which the computer can really understand. Before the computer can process commands in other languages (such as BASIC), the commands must first be translated into machine language.

MEMORY: Usually refers to addressable memory — the memory which may be accessed with the PEEK and POKE commands. Altogether, there is 64K - 84K of this kind of memory, depending on exactly how you define the word addressable. Sometimes memory is used more broadly to include all components of the computer system which can remember information. This would include, for instance, the disk drives, and the registers in the 6510 microprocessor.

MICROPROCESSOR: A computer on a chip. A microprocessor is an entire computer, stripped to its bare essentials, and compressed onto a single chip. A microprocessor typically has very little memory, and no peripherals such as a screen or keyboard.

MICROSECOND: One millionth of a second. The fastest machine language commands are executed in somewhat less than a microsecond.

MILLISECOND: One thousandth of a second.

NANOSECOND: One billionth of a second. 1000 nanoseconds equals one microsecond.

OCTAL: Base 8. In Octal, the numbers from 0 to 20 are: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23, 24.

OPERATING SYSTEM: A set of routines or programs which guide a computer through fundamental activities, such as displaying information on the screen, starting and stopping the motors in the disk drives, and displaying error messages. The KERNAL and BASIC are major parts of the COMMODORE 64 operating system.

PERIPHERAL: An add-on to your computer system, such as a screen, disk drive, printer, joystick, and so on.

POINTER: A piece of information in memory or on disk, which designates another location in memory or on disk. By means of pointers, information may cross-reference other pieces of information.

RAM (RANDOM ACCESS MEMORY): High-speed electronic memory, used chiefly to hold programs and routines currently being executed, and to act as a temporary storage area for small amounts of information. When the computer is turned off, all of the information in RAM is lost.

RECORD: In the case of an inventory file, a record would be the set of information on one item in inventory. The entire file would consist of a large number of records with the same information-format. In the case of a mailing list file, a record would be the name and address information on one person. The entire file would consist of a large number of records like this.

REGISTER: A small memory area on a microprocessor. The storage capacity of each register in the 6510 microprocessor is one byte (eight bits). Most microprocessors have no more than one or two dozen registers. Registers are used by a microprocessor primarily as tiny scratch pads, to help them in doing their work.

ROM (READ ONLY MEMORY): Like RAM, except that information is permanently "burned in" to it. This information cannot be altered, and when the computer is turned off, the information remains in the ROM. The COMMODORE 64 is equipped with ROM which holds routines for the operating system and for executing BASIC.

ROUTINE: A series of commands, written in a programming language (machine language, BASIC, FORTRAN) which performs some meaningful task.

SPECIAL KEYS: These are the SHIFT, CTRL, Commodore, and other special purpose keys which increase the power and versatility of your keyboard.

STORAGE: Usually refers to disk storage or cassette storage. Sometimes is used to mean other kinds of memory as well.

VARIABLE TYPES: In Commodore BASIC, there are three variable types: floating point (holds integer and decimal values); integer (holds integer values only); and string (used for all kinds of information).

INDEX

A
add-ons 135, 138
arrays 90
ASCII 56
assembly language 160

B
base 16 151
base ten 47, 48, 65
base two 47
BASIC 62
binary 47, 48
bit 47
  high order 47
  low order 47
Block Availability Map (BAM) 111
byte 14

C
codes 141
commas 165, 166
Complex Interface Adapter (CIA) 139
communications lines 135, 138

D
data block 97
datassette recorder 103, 104
directory 112
disk drive 103, 105

H
hexadecimal 151

I
inverse characters 51

J
jiffy 39

K
KERNAL 62, 63
keyboard graphics 31
keys 31
  commodore 31
  function 32

L
language 141

M

machine language 60, 63, 125
master pointer 66
math calculations 168, 169
memory 13, 135, 137
  background 67
  clear 11
  color 21
  foreground 67
  map 17, 59, 142
  screen 15
microprocessor, MOS 6510 .... 135, 136
Multi-Color Bit Map Mode 157

N

negative numbers 93

P

peripherals 135, 138
pixel 155
pointers 66, 70, 81, 141

R

radio frequency 162
RAM 66, 137
random numbers 37, 169
register 128, 137
ROM 67
  BASIC interpreter 137
  character 137
  KERNAL 137

S

scratch pad 59
semicolons 165, 166
shape table 40, 41
SID 115, 139
sound 115
special purpose devices 135
Standard Bit Map Mode 155
storage locations 14
system block 97

V

variables 89
  floating point 89, 93, 94
  integer 89, 90, 93
  string 90, 97
Video Interface Controller (VIC-II) 139

W

white noise 117, 122

THE SUPER COMPUTER SNOOPER

COMMODORE 64

THERE'S SOMETHING GOING ON!

You're already writing simple programs in BASIC, but you want to know more. You've typed endless characters into the Commodore 64, but you're not sure what happens to them once they leave the keyboard. What goes on inside the C-64? How does a computer actually work?

In this fascinating book, you'll learn how the Commodore 64 "thinks." *THE SUPER COMPUTER SNOOPER* traces the path of a character from the keyboard to areas of memory, to the disk, and onto the screen and printer. You'll find out how to restore a program that has been erased accidentally, how to "listen" to the inner workings of the C-64, how to identify deleted or hidden files on a disk, and how to write a program which re-writes itself.

By the time you've finished this book, you'll be fully prepared to study such advanced topics as machine language programming and arcade graphics. You'll not only understand how a computer works, but you will have learned powerful techniques that you can use in your own programs.

There's something going on inside the Commodore 64. *THE SUPER COMPUTER SNOOPER* clears away the mystery and reveals how the C-64 works!

ISBN 0-88190-356-6

20660 Nordhoff Street, Chatsworth, CA 91311-6152
(818) 709-1202