Dr. C. Wacko Presents

ATARI BASIC & THE WHIZ-BANG MIRACLE MACHINE

David L. Heller & John F. Johnson

Dr. C. Wacko Presents Atari BASIC and The Whiz-Bang Miracle Machine

David L. Heller and John F. Johnson

ADDISON-WESLEY PUBLISHING COMPANY, INC.
Reading, Massachusetts • Menlo Park, California • Don Mills, Ontario
Wokingham, England • Amsterdam • Sydney • Singapore • Tokyo
Mexico City • Bogota • Santiago • San Juan

Cover and Book Design-Teapot Graphics/John Johnson

Many thanks to Atari, Inc. for the art used in Appendix C, ATASCII Codes. Used with permission of Atari, Inc.

Library of Congress Cataloging in Publication Data

Heller, David L.
Dr. C. Wacko presents Atari BASIC and the whiz bang miracle machine.

Includes index.
1. Atari computer--Programming. 2. Basic (Computer program language) I. Johnson, John, 1944-
II. Title. III. Title: Doctor C. Wacko presents Atari BASIC and the whiz bang miracle machine.
QA76.8.A82H45 1984 001.64'2 84-21687
ISBN 0-201-11491-7

Printed from camera ready boards supplied by Teapot Graphics, Santa Cruz, California.

ISBN 0-201-11491-7

ABCDEFGHIJ-HA-8987654
First printing, September 1984

Contents

DR. WACKO PRESENTS ATARI BASIC

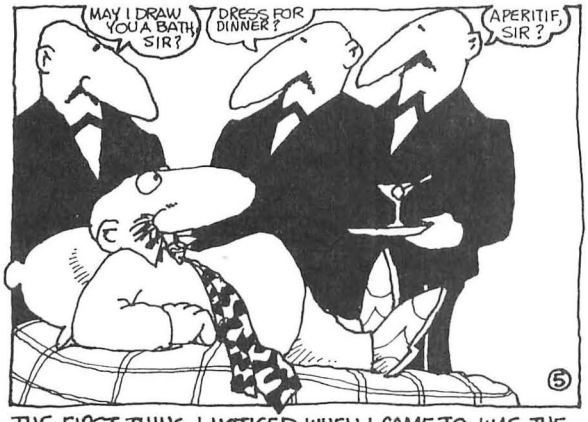
Preramble To The Convolution

IT ALL STARTED MANY YEARS AGO WHILE I WAS STEAMING OFF THE COAST OF MADAGASCAR WITH THE NEPALESE NAVY.

A SUDDEN SQUALL HAD COME UP, POUNDING THE FLEET LIKE MATCH BOXES. IT DAMPENED MY ENTHUSIASM SOMEWHAT...

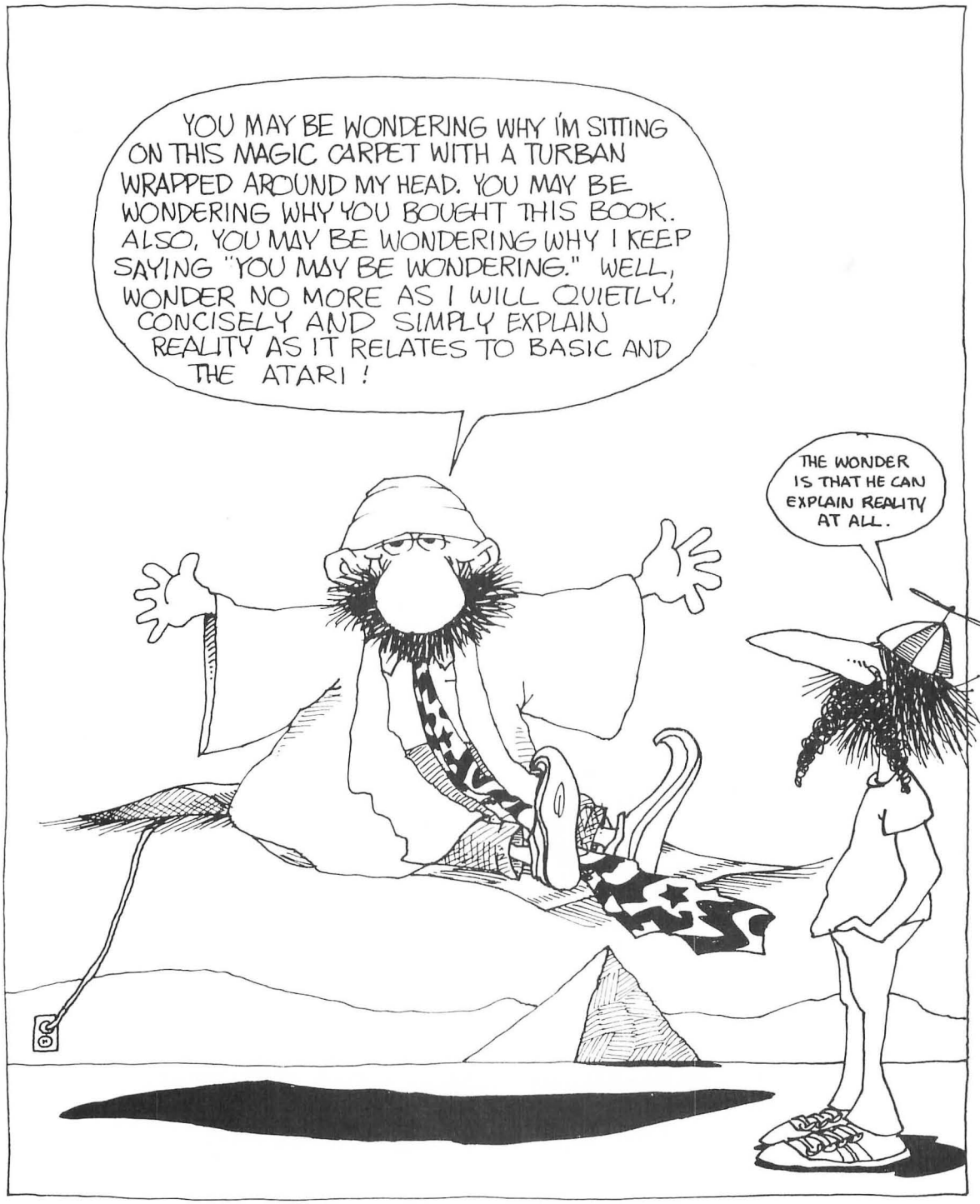
WE TURNED INTO THE WIND AND HEADED FOR A SHELTERED LAGOON NESTLED BETWEEN THE EDGE OF THE DESERT AND THE RAGING SEAS

STIFELING HOT, DRY AIR BLOWN FROM THE DESERT COMPOUNDED BY A DESPERATE THIRST MADE ME THINK I WAS HALLUCINATING.



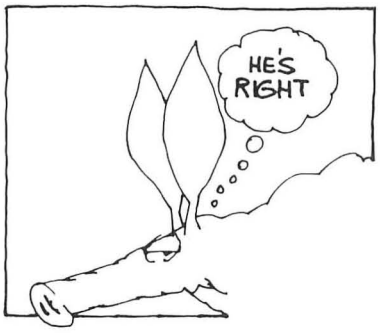
THE FIRST THING I NOTICED WHEN I CAME TO WAS THE INSIDE OF A RIPPLING PURPLE TENT SMELLING OF INCENSE. I WAS SURROUNDED BY A BAND OF WEIRD LOOKING CHARACTERS - WEIRD BECAUSE THEY ALL APPEARED UPSIDE DOWN AS THEY BENT OVER ME. I STRUGGLED TO A SITTING POSITION, AND MUCH TO MY SURPRISE, THEY STILL LOOKED WEIRD.

AND SO BEGAN MY ADVENTURE THROUGH THE DESERT IN SEARCH OF THE WHIZ-BANG MIRACLE MACHINE AND IT'S UNIVERSAL ABILITY FOR BASIC COMMUNICATION. **WE DID INDEED FIND THE MIRACLE MACHINE, AN ATARI COMPUTER! NOW WE INVITE YOU TO JOIN US FOR THE AMAZING ADVENTURE OF DISCOVERY THROUGH THE DESERT OF KNOWLEDGE. SO SADDLE CLYDE AND TURN THE PAGE.**


YOU MAY BE WONDERING WHY I'M SITTING ON THIS MAGIC CARPET WITH A TURBAN WRAPPED AROUND MY HEAD. YOU MAY BE WONDERING WHY YOU BOUGHT THIS BOOK. ALSO, YOU MAY BE WONDERING WHY I KEEP SAYING "YOU MAY BE WONDERING." WELL, WONDER NO MORE AS I WILL QUIETLY, CONCISELY AND SIMPLY EXPLAIN REALITY AS IT RELATES TO BASIC AND THE ATARI !
THE WONDER IS THAT HE CAN EXPLAIN REALITY AT ALL.

So, You Want To Talk To Your Atari?

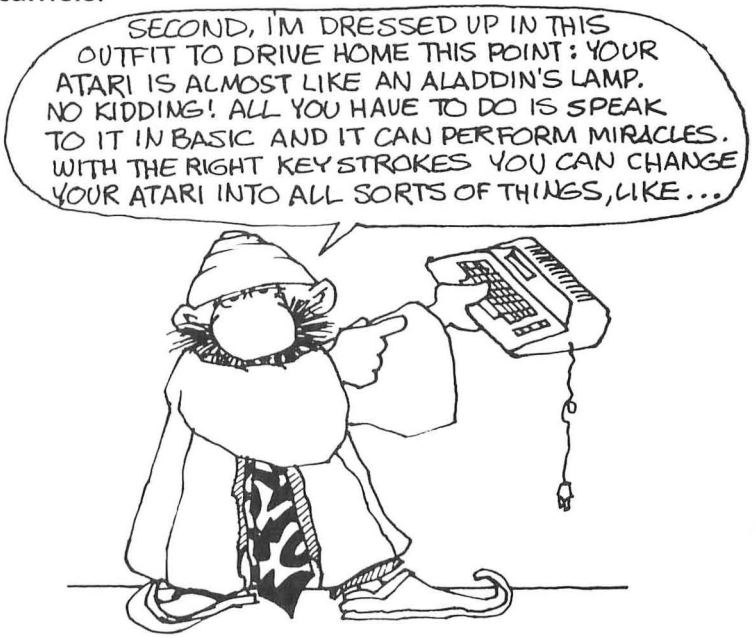
Holy whiz bang! If you want to talk to your computer, this is it! The most fabulous book on Atari BASIC ever written! But a word of caution: Take care as you flip through these pages. The characters residing herein are absolutely wacko. Don't catch their wackiness! Friends, neighbors, relatives, aardvarks, and other terrestrial beings might not understand.



You've been warned, and you're still reading? Well, brave soul, since you've taken the plunge, you might as well find out what's in store for you.

This Book Was Expressly Written (it took two days)

First, this book was expressly written for *all* Atari 400, 800, and XL computer owners equipped with Atari's exotic BASIC programming language and their camels.



- A fancy typewriter (or word processor)
- A file cabinet (or data base)
- An accountant (your very own number cruncher)
- A game machine (fun)
- A machine that teaches you about itself (conceited)
- A spelling helper (my favorite)
- An artist's palette (messy)
- A paper weight (functional!)

You'll learn how to change your computer into all sorts of weird things, plus lots of other miraculous stuff as you meander through this book.

Tripping Through This Book (Ooops!)

This book is so well organized that there is absolutely, positively, *no way* you can get lost during your journey. (Dr. Livingston, I presume?)

Limber Up Your Fingers: Your trip begins as you exercise your fingers and get acquainted with the keyboard. Once you've savored Atari's many keyboard delights, you march bravely on to the rigors of Atari Boot Camp.

Basic BASIC Training: Here, I'll introduce you to some BASIC fundamentals and reveal some of my more subtle BASIC tricks. You'll be astounded by many new and exciting things, like PRINT statements, variables, and line numbers. But have no fear; Dr. Wacko will be right behind you all the way.

On, To the Great White Expanse: When you leave Basic BASIC Training (either by graduating or going AWOL), you'll slip into the central section of this book, otherwise known as the Great White Expanse. Here, as you wander through the desert, bold examples of BASIC

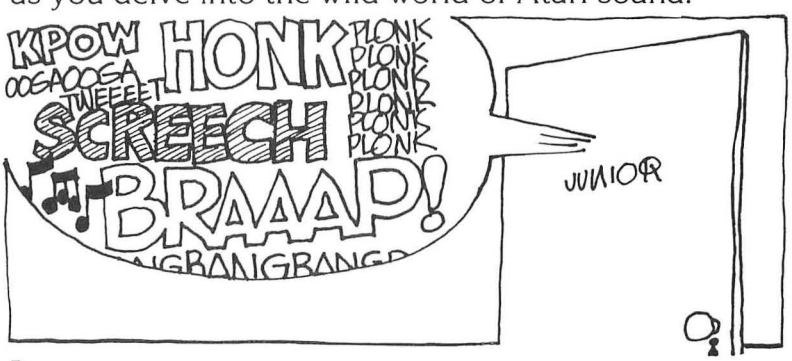
usage pop off each page like lush oases to dazzle your eyes. (Water! Water!)

Build Your BASIC Vocabulary: Having trouble expressing yourself? In this section the Wackos and I help you build your BASIC vocabulary in a logical, step-by-step way. Short, fun-filled, and utterly ridiculous programming examples show you what each BASIC command does, how to use it, and how to combine it with other BASIC words so you can really start conversing with your computer.

Obnoxious Colors, Weird Shapes, and Bright Lights: After leaving the stark white desert, you enter the vibrant and colorful world of Atari graphics. Here you'll see and do things with your knowledge of the BASIC language that you never dreamed possible. (Well, almost never.)

You'll learn my favorite, and until now, carefully guarded graphics tricks. After you've wandered through only a few pages of this section, you'll astound your friends, neighbors, and yourself as strange shapes, obnoxious colors (pugnacious purple), and bright lights dance on your screen, all under your control.

ZOUNDS!: Piercing sounds tingle your eardrums, eerie flute-like tones float melodiously from your computer room, and the neighborhood cats howl as you breeze through this section. So close the door, put cotton in your ears, and get set for some real audio excitement as you delve into the wild world of Atari sound!



Sounds good, doesn't it? But, that's not all. There's more!

Weaving The Perfect Flying Carpet-The Art of Programming: Here's where it all comes together. Now you'll be prepared to fully utilize your BASIC knowledge, talent, and creativity to design some of the weirdest, most useful, and most exciting programs in the universe. Never before in history (well, almost never) has there been a book that teaches you *the fine art of programming.* The *art* of programming is all here in this one-of-a-kind section.

You'll start with an idea or need, and develop programs using my patented "Wacko Modular Approach." I'm with you each step of the way, right down to inflicting your finished product on your unsuspecting friends.

You will learn much, much more than BASIC programming. You'll learn to "see" every detail of *any* problem clearly; you'll discover and use the *creative* (Wacko) side of your brain, and you'll learn that programming is just like making anchovy burritos. (Honest!)

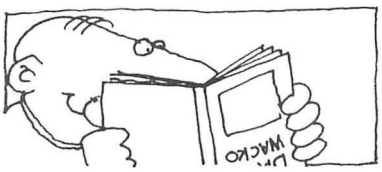
Building-Blocks and Lots of Schlocks: I wrap up this wondrous book with creative building-block projects that show you how to build your own: Mini Word Processor, Secret Text Coder/Decoder, Infamous Model Of The Universe, and *much, much more!*

Are you ready to go? OK, just hop aboard your camel and we'll start our adventure!

What's this camel stuff, anyhow? Good question, and it will all become crystal clear (as mud) after you meet the weird gang that will accost you during the rest of this great saga.

OK gang, let's get this caravan rolling. Do your stuff!

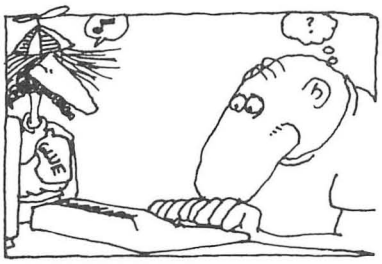
EZ Book Instructions



1. Open this mystical book.

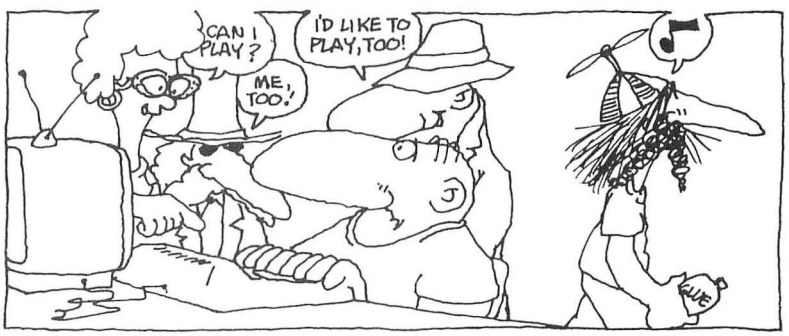
2. Place next to Atari computer.

3. Turn on your fingers.



4. Place your fingers on the keyboard.

5. Grab the closest magic carpet, put on your turban, fez, burnoose, or beanie and enjoy your voyage through Dr. C. Wacko's exciting world of Atari BASIC programming!



DR. WACKO PRESENTS ATARI BASIC

DR. WACKO

EVICTED FROM IOWA IN 1936, INVENTED WIND-UP COMPUTER IN 1947. SWITCHED TO BASIC IN AN ATTEMPT TO TALK TO HIS SON, JUNIOR. DISCOVERED BASIC WAS USEFUL IN PLACES LIKE NEPAL, MADAGASCAR AND SILICON VALLEY.

CAPTAIN ACTION

RESIDENT FIXIT & SWASHBUCKLING HERO OF GAME FREAKS, HE LEFT REALITY YEARS AGO TO WORK FOR WACKO. NEVER RETURNED, EITHER. SOMETIMES KNOWN TO TEASE JUNIOR.

MRS. PETUNIA WACKO

WACKO'S CHARMING AND ATTRACTIVE WIFE. AN ADVANCED MAINFRAME COMPUTER PROGRAMMER AND DR. WACKO'S MOST VOCAL CRITIC. SHE IS ALSO THE ONLY ONE IN THE GROUP WITH ANY PRACTICAL KNOWLEDGE.

SNIDLEY SEERSUCKER

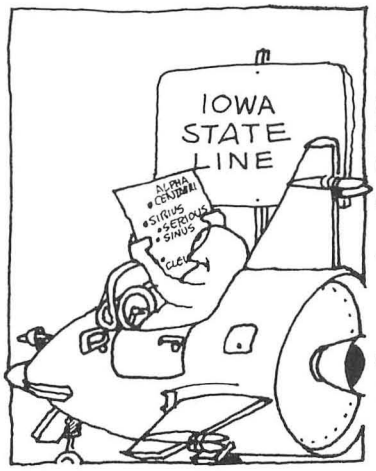
LOVES TO FIND MISTAKES IN DR. WACKO'S PROGRAMS. HE'S WACKO'S 2ND. COUSIN TWICE REMOVED (TO THE COUNTY JAIL) AND OCCASIONALLY HELPS WACKO WITH COMPUTER GRAPHICS.

JUNIOR

LOVINGLY CALLED CEMENTHEAD BY HIS MANY FRIENDS, HE'S BEING GROOMED TO TAKE OVER THE BUSINESS WHEN HIS FOLKS RETIRE TO THIER CABIN IN CLEVELAND. HE COLLECTS TEST PATTERNS.

MS. PEEKY

FAMOUS FOR PEEKING AT THE LAST PAGE FIRST, SHE'S OUT HERE TO EXPERIENCE ADVENTURE, EXCITEMENT AND (HOPEFULLY) TRUE ROMANCE.

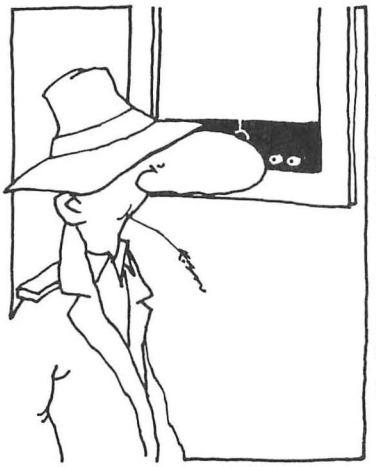


GROVER

AN ILLEGAL ALIEN WHO GOT LOST ON THE WAY TO ALPHA CENTAURI BECAUSE OF A BENT WARP DRIVE ON HIS '59 CADDY SPACE SHIP. IT CAN'T FLY OVER 200 FEET BECAUSE CAPTIAN ACTION FIXED IT.

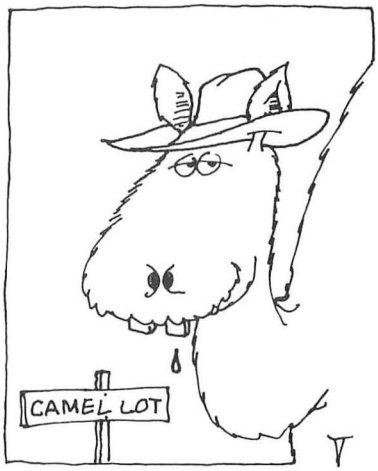
BERNICE BERNOOSE & LAWRENCE OF NEWARK

KICKED OFF OF THE CLEVELAND BALLET, BERNICE HAS BEEN KICKING AROUND THE SAHARA HOT SPOTS PURSUED BY LAWRENCE, WHO'S LOOKING FOR SILICON CHIPS IN THE SAND.



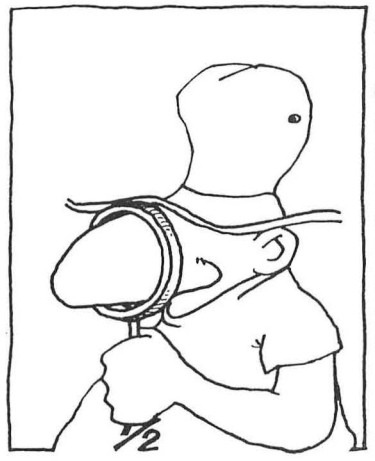
SMOKEY PEEK

THE GROUP'S EXPERT AT WINDOW SHOPPING FOR PEEK LOCATIONS. A FRIEND OF LAWRENCE OF NEWARK, HE'S SEEN THE CALVIN COOLIDGE STORY 16 TIMES.



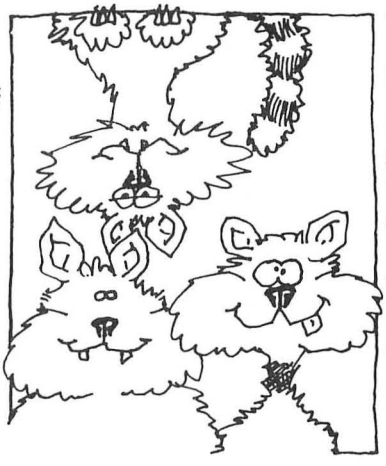
CLYDE

CHIEF TRANSPORT FOR EVERYBODY BUT JUNIOR, HE OCCASIONALLY SPITS OUT A FEW NUMBERS FOR WACKO



SLOW POKE

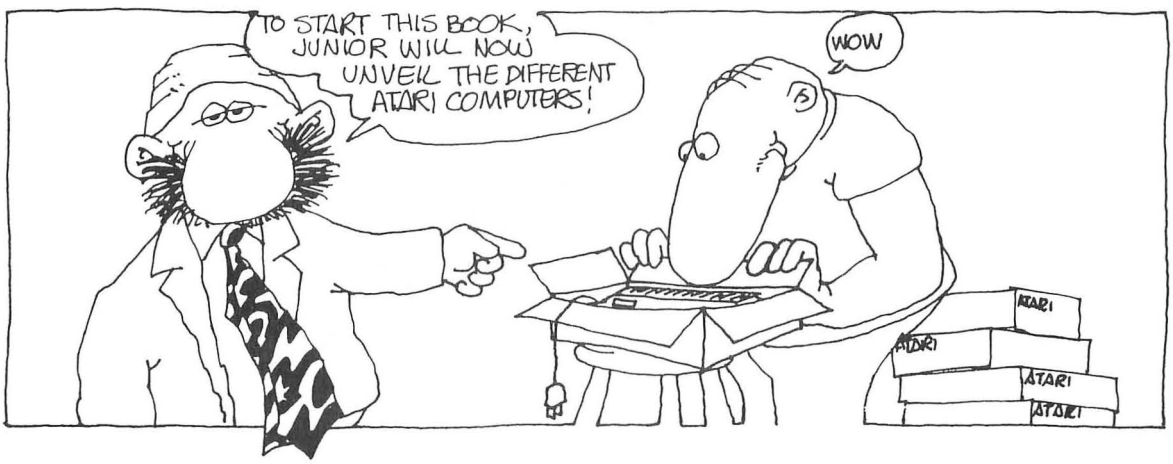
AFTER 13 YEARS OF CORRESPONDENCE SCHOOL, HE'S THE POKE EXPERT HERE... MAINLY BECAUSE HE'S USUALLY POKING AROUND SOMEBODY ELSE'S BUSINESS.



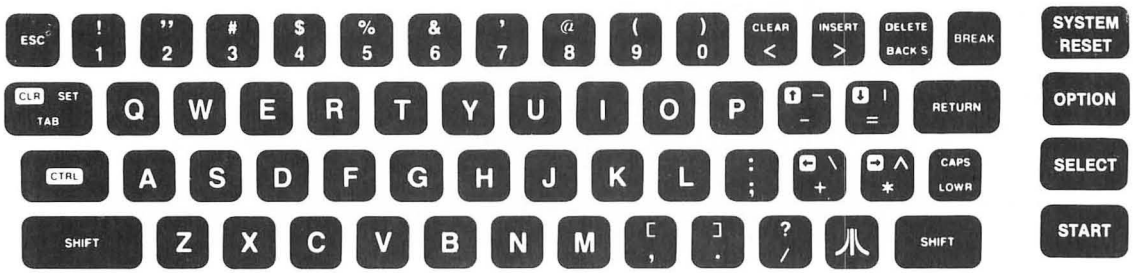
THE WACKO CATS

KEYS, PADDLES & JOYSTICK LOVE TO WALK ON THE COMPUTER AND MUNCH CONTROLS WHEN NO ONE IS LOOKING. TOTALLY UNHOUSEBROKEN.

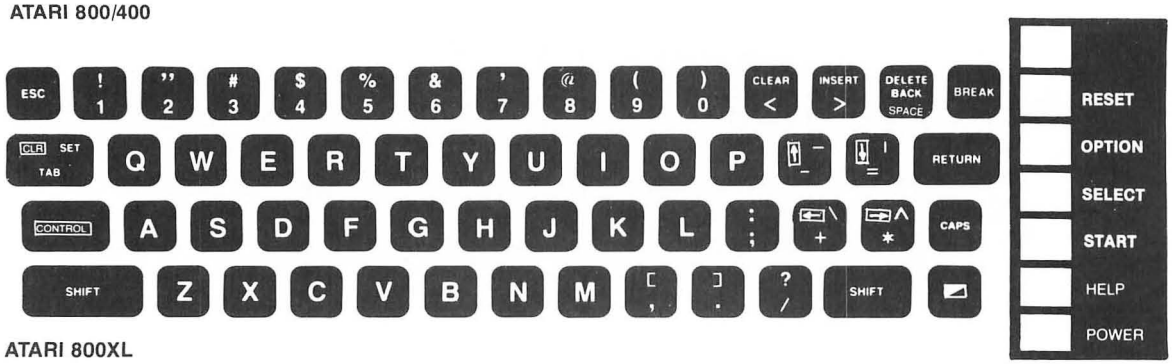
1
Limber Up Your Fingers



Here they are, the keyboards you've all been waiting for. On the top, I present the fabulous Atari 800/400 keyboard, and on the bottom, the new improved super-deluxe XL version. You should be sitting in front of one of the two now. If you aren't, you took the wrong flight!

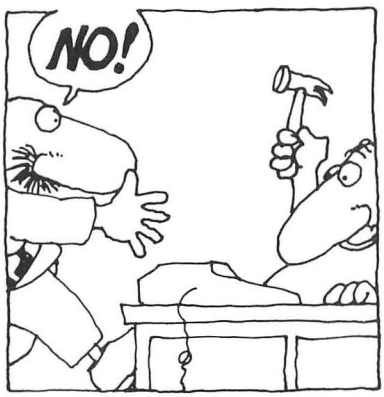


ATARI 800/400



ATARI 800XL

Get To Know Your Keyboard



Your keyboard is your best friend. It provides a real handy way of communicating with your computer. You talk to it by banging on the keys. Simple, isn't it?

Try It Without BASIC

If you are in front of an 800, 400, or 1200XL, make sure that the Atari BASIC language cartridge is *not* stuck in the cartridge slot. If it is, take it out! (400/800 owners: Remember to CLOSE THE CARTRIDGE DOOR!) Now, turn on your computer!

If you've got a 600XL, 800XL, or 1450XL, Atari BASIC is built into your computer! To acquaint yourself with your keyboard you'll have to type in and RUN the short program that follows. When the word "READY" appears on your screen, just type each line of the program *exactly as shown*.


```
10 GRAPHICS 0
20 OPEN #1,4,0,"K:"
30 GET #1,A
40 PRINT CHR$(A);
50 GOTO 30
```

Press the RETURN key at the end of each line. When you've finished, type the word "RUN", press the RETURN key and you're ready to continue.

All set? Ready to limber up your fingers? Ok, let's go for it.

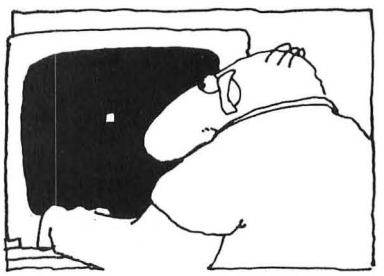
Press some buttons and keys. Don't be timid, the worst that can happen is that you'll dent your fingertips. What have you got to lose?

Clean Up That Mess!

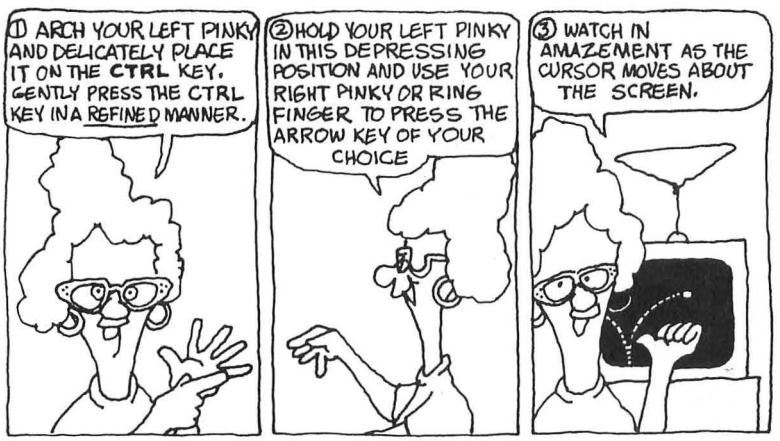
Boy, are you sloppy! Look at the mess you left on the screen. It looks a little like Junior's bedroom. Don't panic! It's real easy to clean up. Hold down the SHIFT key, press the CLEAR key, and your screen is as clean as a whistle. (How clean is a whistle, anyhow?)

Some Cursory Cursor Movement

Now that everything is straight-arrow and shipshape it's time for some cursory cursor movement. To move that white square (known as a "cursor" by real computer wackos), hold down the control [CTRL] key and press one of the four arrows at the right of the keyboard.



I know, it's obvious that the arrow printed on each of the four keys shows which way the cursor moves when you press it. But moving the cursor the Wacko way requires good manners and a sense of proper etiquette. Here's Ms. Peeky to show you what I mean.



Not only does this method work, but if anyone sneaks into your computer room, you'll look like a real pro programmer! (Or a real weirdo.)

Bore yourself by moving the cursor all around the screen. Stop before you start crying in frustration and

locate the CAPS/LOWR key on the right side of your keyboard.

Did you find it? OK, press it once and type "the first thing that comes to your mind."

A Case of Uppercase Shift

Your Atari must be getting old — it's acting like an old-fashioned typewriter. All those words you typed appear on your screen in lowercase!

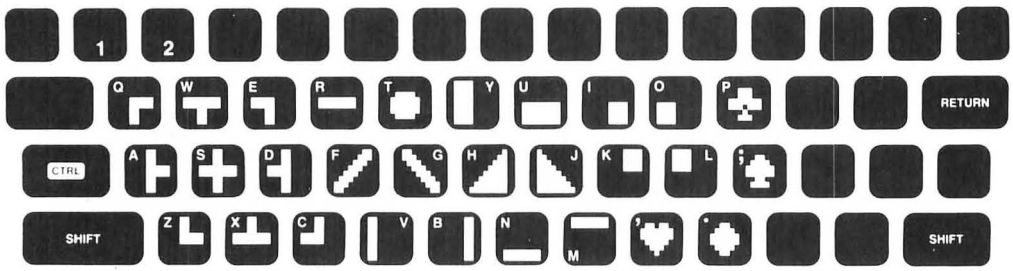
Just to prove my point, hold down one of the SHIFT keys and press any other key. Voila! An upper-case letter or symbol! See, I told you. It's acting just like a typewriter.

Lock Up Your Keyboard!

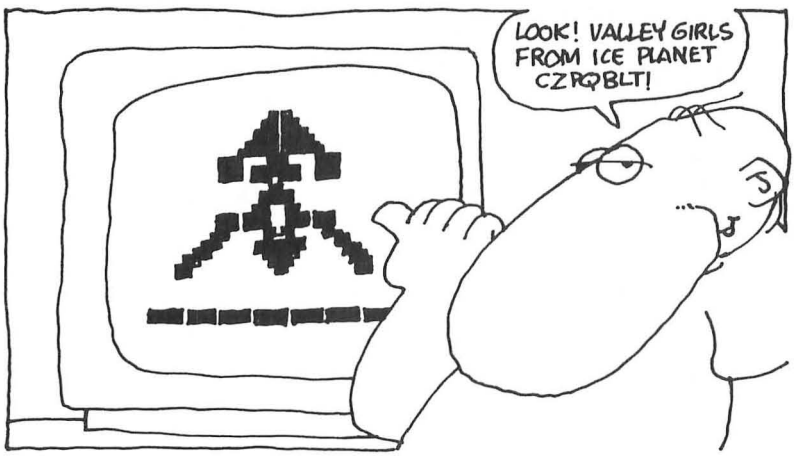
To lock up your keyboard, in upper-case that is, before it starts believing that it's a typewriter, just hold down SHIFT and press the CAPS/LOWR key.

The Control Key: Weird Shapes and Odd Graphics

Take a look at this keyboard picture. Pretty strange-looking, isn't it? You can make all those weird and strange graphic shapes appear on your screen with a little help from the CTRL key.



Just hold down the CTRL key, press any key, and a completely new character will magically appear on your screen. You can use these shapes to draw your own strange pictures, with or without Atari BASIC on line!



For example, hold down CTRL and press the comma key (,), the period (.), the semicolon (;) and the "P" key. Now you're ready for a game of Gin Rummy, or Junior's favorite, Fifty-Two Pick Up!

RETURN To The Keyboard...Please!

I know that Junior's having fun playing cards, even if he isn't playing with a full deck. But before you both get totally engrossed, please RETURN to the task at hand by pressing the RETURN key!

The RETURN key is one of the most important keys on the keyboard. Press it now, and the cursor moves down one space and whizzes over to the left margin. Just like pressing the carriage return on a typewriter. But when you are using Atari BASIC, pressing RETURN wakes up your computer and lets it know that you are telling it something.

When you finish your tour of the keyboard and move on to Basic BASIC Training, you'll learn a lot more about the wondrous RETURN key and its many mystical powers.

TABu-aroonio (CLR-SET-TAB)

The key that's marked CLR-SET-TAB works just like the TAB key on an old fashioned typewriter, it moves the cursor a set number of spaces to the right. To witness this wondrous apparition, first press RETURN to move your cursor to the left side of the screen. Now press the TAB key. The cursor moves 10 spaces to the right. Press the TAB key again, and the cursor moves another 10 spaces.

Oy Vey! It seems to have a mind of its own. But it's easy to take control and make that cursor stop where you want it to. Just follow along with Wacko:

1. Press RETURN to position the cursor at the left of the screen.
2. With the space bar, move the cursor 3 spaces to the right.
3. Hold down SHIFT and press the CLR-SET-TAB key.

   You've just set a TAB stop at the cursor's position!

4. Now, press RETURN, then press the TAB key again and watch that little rascal stop just where it's supposed to!

To clear the TAB stop under the cursor, hold down CTRL, and press the CLR-SET-TAB key.

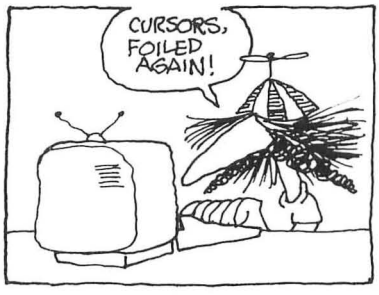
- TAB: Moves the cursor.
- SHIFT + CLR-SET-TAB: Sets a TAB stop at the cursor's position.
- CTRL + CLR-SET-TAB: Clears the TAB stop under the cursor.

Now that you've got that all sorted out, it's time to correct a few mistakes and do a little editing.

Editing...Looking Good

I never make any typing or spelling mistakes when I'm writing letters or entering programs. But if you ever do, your Atari lets you make instant corrections on the screen, using its world-renowned and easy to use screen-editing functions.

Send Your Cursor Home



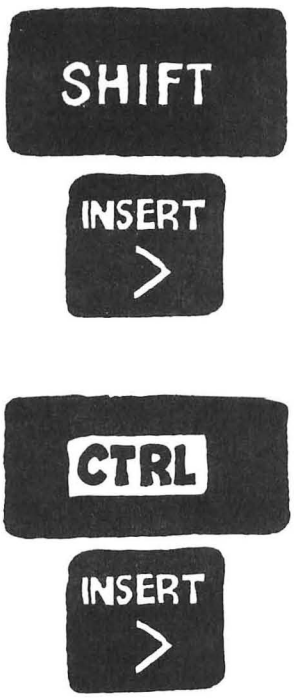
You already know how to clear your screen. Remember the old SHIFT + CLEAR trick? You can also use the CTRL key in combination with the CLEAR key to accomplish the same results. Both of these great methods erase the screen and send your cursor home

to the upper left corner of your screen.

Your knowledge of cursory cursor movement is immense, and you are well prepared to start fixing up those pesky screen-entry mistakes.

Ready to have some fun? OK, let's play a game of "Simple Wacko Says," as you learn all about screen editing.

Simple Wacko says, "Type the following sentence *exactly* as it appears."



I here by volunteer for BASIC Trainng.

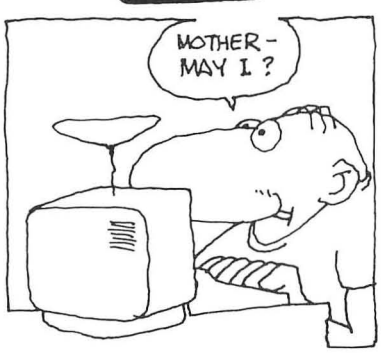
Inserting a line
SHIFT + INSERT

Simple Wacko says, "Place your cursor over the letter **I** at the beginning of the sentence."

Simple Wacko says, "Hold down SHIFT and press the INSERT key." You've just pushed the whole sentence down one line! This is great for creating space for new program lines or text. But watch out! Stuff written on the bottom line of your screen is pushed down and disappears behind your TV's channel knob. It's not gone forever, though!

Insert A Weird Character
CTRL + INSERT

Simple Wacko says, "Place your cursor over the second **n** in the word **trainng**. There yet? OK, Simple Wacko says, "Hold down CTRL and press the INSERT key." You've just created a space between the two **n**'s. To correct my horatious* spelling, just type the lower-case letter **i** and you're all set.



Did you do it? Gotcha! Simple Wacko didn't say, "Type the lower-case letter."

* Horatious: horrible and atrocious mixed together; Wacko/Webster Space Duck Dictionary.

You don't care? You don't want to play any more? OK, be that way. Well, even if you are fed up with that silly game, you've just done some fancy insertion, and your sentence is really starting to shape up. Here's how it should look now:

I here by volunteer for BASIC Training.

Delete That Character
CTRL + DELETE/BACK S

Ooops! There's a space between the words "here" and "by" that shouldn't be there. It should read "hereby".

To set things right, place the cursor between the words "here" and "by," hold down CTRL, and press the DELETE/BACK S key.

You did it! Your sentence is perfectimundo! Here's how it looks now:

I hereby volunteer for BASIC Training.

You do? Weren't you told *never* to volunteer! OK, in a minute you can flip the page and join the rest of the troops.

Delete The Entire Line
SHIFT + DELETE S

If you've changed your mind, and really don't want to go, you can delete your pledge with a few simple strokes on your keyboard. (Which is more than I could do when I signed up for the Nepalese Navy.) Just move the cursor over the letter **I** at the beginning of the sentence, hold down SHIFT and press the DELETE S key. The entire line disappears!

Very Special Computer Keys

Stop, before you go marching off to Basic BASIC

Training. Let's take a look at some very special computer keys.

The Great ESCape

The escape [ESC] key prints the characters that represent the various control-key functions. Here's an example of how the Great ESCape works. Press the ESC key once, then hold down CTRL and press CLEAR. Instead of clearing your screen, the symbol for "CLEAR" appears on your screen. Amazing, isn't it? Here are a few more escape combinations you can try:

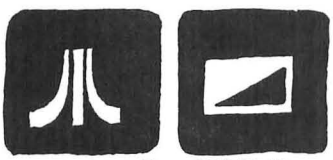
ESC, then CTRL + INSERT
ESC, then CTRL + DELETE/BACK S
ESC, then CTRL + TAB

Later, I'll show you how to use this great escape act to spice up your BASIC programs. (Promise!)

The BREAKout

You'll use the BREAK key to interrupt your computer while it's thinking about your BASIC programming.

The Perverse Inverse Character Key



Now, last but not least, the infamous inverse character key. If you've got a 400 or 800 Atari computer, look for a key with the Atari symbol on it at the lower right of your keyboard. If you're in front of an XL model computer, find the key with a diagonal white/black design on it.

Did you find it? If you did, you're in for a big surprise. What are you waiting for? Press it!

Now, type something. All the characters are printed on your screen in inverse video!

When you get tired of staring at perverse inverse characters, press it again to return to normal print.

You did it! Your fingers are limbered up and your adventure through BASIC programming is about to begin! So buckle your seat belts, hold on tight, and get set to march into Basic BASIC Training.

2
Basic BASIC Training

BASIC training was a snap for me. I breezed through the short course in a few days, and just look where it got me!

If *you've* already graduated from boot camp, you may want to breeze through this chapter, pick up the highlights, and move bravely on to the Great White Expanse.

But just so you don't bop ahead of the game, here's what I'll be covering during Basic BASIC Training:

- The Immediate Mode
- PRINT
- Lower, Higher, and Higher Than Lower Arithmetic
- Line Numbers
- Statements
- Variables

Sign up for BASIC training, and you'll be in good company. Look at that enthusiastic group at the top of the page. Don't they inspire and instill a sense of confidence? If they can make it, so can you! As you march along with these bright-eyed recruits, you'll learn to use many of the tools of the BASIC programming trade. And once the basics are out of the way, you'll be prepared to reach into your creative self and produce some masterful programs.

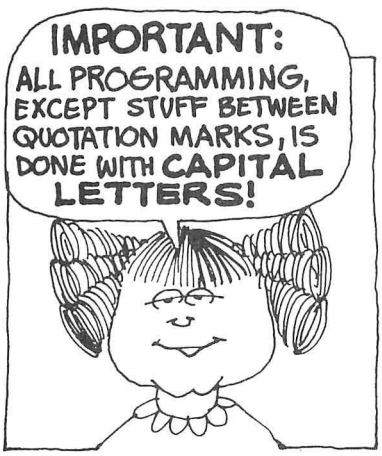
After graduation, you'll have a working knowledge of the most useful (and useless) programming tools, and as you progress through BASIC training I'll help you use these basic tools, plus your creativity, to design *any* program (no matter how wacko) to suit almost any job, task, or whim.

So, now that you and your Atari are present and accounted-for, let's dive right in. It's time to start having some fun with that computer squatting in front of you. If your Atari is equipped with built-in BASIC, activate it by turning your computer off, then on again. If

you're working with a BASIC cartridge, insert it into the cartridge slot and we'll get started!

Attenhut! Programming is simple. Just follow my incomprehensible instructions, use your imagination and brain power, and you're on your way! Here, I'll show you. But first, put the plug in!

A Wacko word of caution: Your Atari is very literal minded. It will go bzzrk and print a numbered Error Message on your screen if you don't enter a program *exactly* as written or try to perform an incomprehensible programming feat. Before you flip out, flip back to Appendix A to see what each numbered Error Message means.



Now, Just type in the following words, *exactly as written*, then press the RETURN key when you see "[RETURN]".

GRAPHICS 1:POSITION 0,8:PRINT #6;
"Why Am I Doing This?"[RETURN]

Simple. Because you've just seen one of your Atari's amazing graphics tricks, entered your first program, *and* gotten some instant and weird feedback. You made your computer do something dumb instead of just sitting there like a sleek piece of furniture! But that's not all. You entered a program in what is called the *immediate mode*.

The Immediate Mode: Life in The Fast Lane

The *immediate mode* of program entry is just like living life in the fast lane. After you type the words and press RETURN to "tell" your computer what to do, you instantly see the results of your programming. Learning to give orders to your Atari is what Basic BASIC Training is all about. And, when you issue orders in the immediate mode, your Atari carries them out immediately. Immediately after you press RETURN, that is.

SYSTEM RESET

When you get sick of staring at a questioning computer screen, you can shut it up by pressing the SYSTEM RESET key.

After you press SYSTEM RESET (don't worry, you won't scramble your Atari's brain) the screen turns blue and the word "READY" appears in the upper left-hand corner. Your computer is talking back to you. It's saying, "I'm READY, Sarge, to accept more of your silly commands."

Don't ignore this insubordination! Now's a good time to get really angry. Yell at the computer. Or, if you're like me, yell at Junior. After you've got it out of your system, you're READY to start pushing *it* around!

PRINT (?)

In Atari BASIC, your computer responds to a number of simple, one-word commands. One way it does this is by printing words on its screen. To make it say what you want it to, use the PRINT command. If you want to get sneaky, just use a question mark (?) instead of the word PRINT. Your computer will understand. Trust me!

Type in the word PRINT and press RETURN.

Uh, oh! There's that word "READY" again! But don't fly off the handle. If you look at your screen closely, you'll see that the computer *did* print something. It printed a blank line!

Not impressed? Neither am I. So, let's get some real utility out of the PRINT command. Let's make it print something really important. Ready? OK, type these words (including the quotation marks) and press RETURN:

PRINT "Stop banging on my keyboard!!"

Geez, it seems you can never make it happy. Now it's talking back, and being insubordinate! But, I'm sure you get the idea.

To make your computer print something on the screen just type the word PRINT, enclose your words of wisdom within quotation marks, and press RETURN.

Anything you put between quotation marks after a PRINT command is printed on the screen, including numbers and symbols! Type this short message, and press RETURN to see the result.

PRINT "E = MC^2, sometimes."

LPRINT If You Have a Printer

If you've got a printer, and would like to see your words PRINTed on paper, replace the PRINT command with LPRINT in the Einsteinian example above. Don't use LPRINT without a printer or when your printer is turned off. If you do, you'll see ERROR message #138 ("Your printer's not hooked up!") displayed on your screen.

Your Computer Isn't Very Bright

There's one thing you should know before you start bossing your computer around. Your computer isn't very bright. It only understands commands that are entered perfectly, one-hundred-percent correctomundo. For example, if you entered PRIR when you meant to enter the command PRINT, your Atari would get confused, burp, and tell you it doesn't understand PRIR by displaying the word "ERROR" followed by the misspelled word. Don't get upset by this. Just remember that it is, after all, only a computer.

LOWER ARITHMETIC

Don't worry, Ms. Peeky. Math has very little to do with BASIC programming. Actually, all you need to know is simple arithmetic, the kind you learned in grade school. In fact, your computer can help you. It's just a super-calculator that's really easy to use.

When you see: "[RETURN]" just press the RETURN key.

Addition

Try typing these simple equations:

PRINT 5 + 5 [RETURN]

PRINT 4 + 6 + 7 [RETURN]

See how easy it is? And you don't even have to use an equal sign! You also don't put quotation marks around the numbers. If you do, your computer will print them instead of performing the math. Wrap quotes around the numbers to see what I mean.

Subtraction

Check out these easy examples:

PRINT 10 – 5 [RETURN]

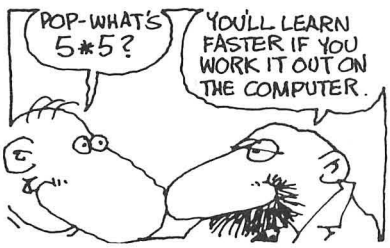
PRINT 20 – 10 – 5 [RETURN]

If you want to get real fancy, try combining addition and subtraction functions like this:

PRINT 20 + 5 – 10 [RETURN]

HIGHER THAN LOWER ARITHMETIC

Now I'm going to move on to the two biggies, multiplication and division.



Multiplication

An asterisk [*] is used as the symbol for multiplication by your computer. (The "*" key is located next to the "+" key on the right side of your keyboard.)

PRINT 5*5 [RETURN]

Division

A slash [/] is used as the symbol for division by your computer. (The "/" key is located next to the "." key on the bottom right of your keyboard.)

PRINT 25/5 [RETURN]

LOFTY CONCEPTS OF LOWER AND HIGHER ARITHMETIC

Lofty Concept 1: A Set of Logical Rules. When you combine multiplication, division, addition, and subtraction in a single statement, things get a little hairy (even if you're as bald as I am). But they do follow a specific set of logical rules.

Here's an example. Type it in, press RETURN, and check out the answer. It should be 9.25.

PRINT 5-2+5*5/4

Don't run away, Ms. Peeky. It's no big deal! Here's Petunia, she'll explain it to you.

If you enter: **PRINT 5 + 7 + 8**

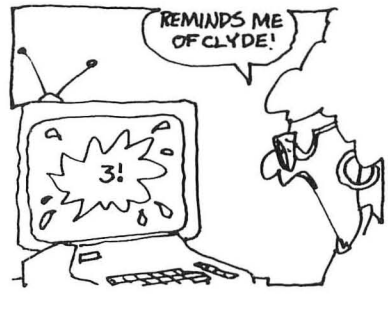
Your computer first adds 5 + 7 = 12, then it adds 12 to 8 to arrive at the answer: 20.

Here's another example. But, this time I've mixed in subtraction as well as addition.

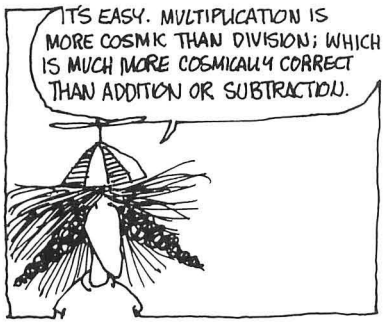
If you enter: **PRINT – 5 + 9 – 3 + 2**

Your computer first adds – 5 + 9 = 4, then it adds 4 – 3 = 1, and finally adds 1 + 2 to arrive at the answer: 3.

When it's all finished calculating, it spits the answer out and displays it on your screen.



I'll summarize this simple operating order, then show you some more examples.



The Cosmic Order of Things

1. For addition and subtraction only, the computer calculates left to right across the screen.

2. For multiplication and division only, the computer performs the multiplication first, then the division, regardless of the order.

3. When combining addition, subtraction, multiplication, and division, the computer performs the multiplication first, then the division, and last but not least, the addition and subtraction from left to right across the screen.

Let's work through the arithmetic problem that scared Ms. Peeky, and you'll see how the computer follows these three operating rules to arrive at 9.25 as the answer.

Here's the problem again:

```
PRINT 5 - 2 + 5 * 5/4
```

Following the proper order, the computer first performs the multiplication:

```
5 * 5 = 25
```

Next, moving right along, the computer does the division:

```
25/4 = 6.25
```

Last, but again, not least, the computer does the addition and subtraction from left to right and arrives at its answer:

```
5 - 2 + 6.25 = 9.25
```

See, *no problem* at all! Your computer was simply

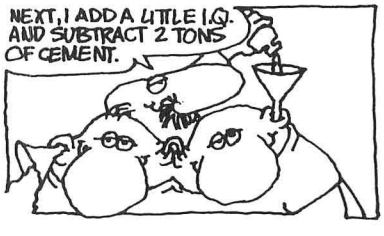
following orders. And even if you switch the order of things around like this:

PRINT 5*5/4+5–2

The answer will still be — you guessed it — 9.25!

Lofty Concept 2: Operations in Parentheses are Done First. You can use the parentheses to change the order of things, and get your own way, so to speak.

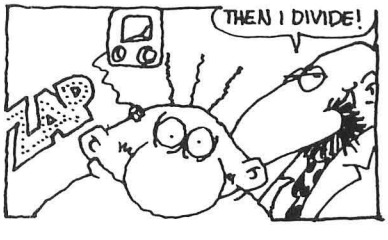
Suppose, in the above example, you really want to mess things up. You want the division performed last, *after* the multiplication, addition, and subtraction. Well, it's simple. Just use parentheses, like this:



PRINT (5–2+5*5)/4

Here's how your computer will operate on this problem.

First it performs the multiplication inside the parentheses:



5*5=25

Next it does the addition and subtraction within the parentheses:

5–2=3

Then it adds the two results within the parentheses together:

3+25=28

And, now that it's finished operating within the parentheses, it hops out and performs the division:

28/4=7

The answer is 7! The same numbers were used in both examples. But in the first case the answer was 9.25 and

in the last example the answer was 7, and all because of the parentheses!

Here's another example that uses the same numbers and parentheses. But this time, amazingly enough, the answer is different than the first two examples.

```
PRINT (5 – 2 + 5)* 5/4
```

When you type this problem into your computer and press RETURN, the answer is 10. Step through this example yourself to see why.

You can breathe easier. We are finished with the arithmetic. Now you know enough to be an expert BASIC programmeroonie. Ms. Peeky, you should be awarded a medal for your perseverance.

See, I told you it was easy! Here's a short summary list to help you remember these simple arithmetic concepts:

- Multiplication first
- Division second
- Addition and subtraction third; left to right
- Operations in parentheses are done first

Leaving the Fast Lane: Bye, Bye, Immediate Mode

Up to this point, you've been living in the fast lane, entering stuff into your computer and experiencing instant results. But living recklessly has its drawbacks. Here's the short program example you first typed in:

```
GRAPHICS 1:POSITION 0,8:PRINT#6;"WHY AM
I DOING THIS?"[RETURN]
```

After you typed this example and pressed RETURN you saw the message written in bold letters across your screen. But if you cleared your screen (by pressing the SYSTEM RESET button) and wanted to repeat the message, you had to *type all* that code again!

The Infamous Programming Mode: Line Numbers and the Perfect 10 & 20

You'll all be happy to hear that there is another way of entering a program: Use *line numbers* in the infamous *programming mode*.

I'll show you what I mean. Here's another nifty example. But this time there is a subtle difference. There are two lines, and each line has a *line number* in front of it. The perfect 10 and the perfect 20! See them?

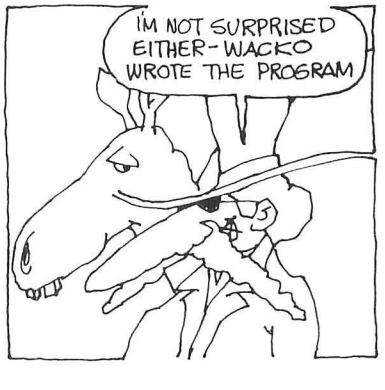
```
10 FOR X=1 TO 20:SOUND 1,X,10,8:NEXT X
20 GOTO 10
```

Turn up the volume on your TV and get set to hear the strange warblings of the nearly extinct Wacko Bird.

Type this program, just as it's shown. Press RETURN at the end of each line, and don't forget the perfect 10 and 20.

Did you do it? What happened?

Nothing appeared to happen, and I'm not surprised.



But appearances can be deceiving, Snidely. Fooled you! Something very important *did* happen after you pressed RETURN. *The Wacko Bird program was stored in your computer's memory!* And your Atari was trying to tell you about it when it said it was "READY."

RUN, RUN, RUN Around

Now that you know the Wacko Bird is hiding somewhere inside your computer, let's find it, drag it out, and get it to warble, so to chirp. To get a program to run around and act weird, all you have to do (you guessed it!) is type the RUN command and press RETURN. Simple, isn't it?

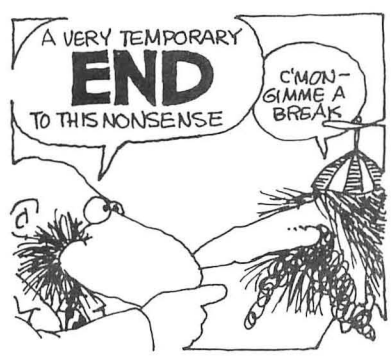
What are you waiting for! Go for it! Type RUN and press RETURN.

Now your Atari is doing its own thing and telling you that it's having fun by singing a little tune! There's nothing like having a happy computer to bang on.

BREAK It up Gang!

If the neighbors are complaining about the noise, or if your cats are getting real interested in your computer, you can stop the Wacko Bird's song in mid-note by pressing the BREAK key.

After you press the BREAK key (you won't break your Atari's heart), the message, STOPPED AT LINE 10, appears. You've stopped your computer right in the middle of its "I'm making lots of noise" routine!



Put an END to this Nonsense

Do you still hear a solitary, persistent, and repugnant tone? Even after you've pressed BREAK?

To put an end to this endless cacophonous tone, just type the word "END" and press RETURN.

END is a command that's often used at the end of a program (where else?). In this case, you've used it in the immediate mode to help Captain Action concentrate on *his* immediate mode, finding the Bug Byte Beer.

CONTinue Where You Left Off!

If you feel that you've hurt your Atari's feelings by stopping its happy warbling, there are two things you can do to make it happy again.

You can either type CONT and press RETURN to make your Wacko Bird CONTinue where it left off. Or (if you want to get snooty) you can type the word "RUN" and press RETURN to reRUN your program.

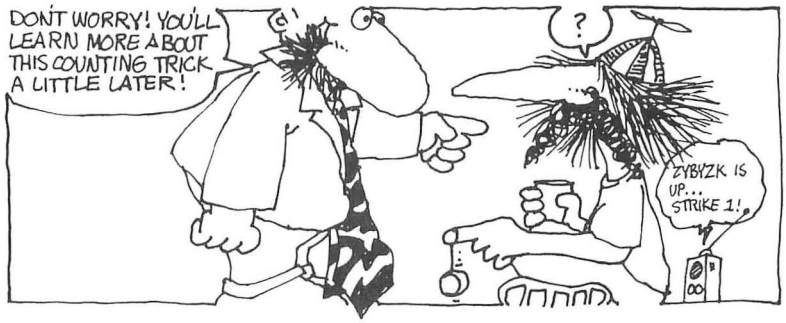
By adding line numbers to a program, you've told your computer to store it in its memory. Then, when you type the word "RUN" and press RETURN, it's off and RUNning, acting strange! The best part of using line numbers is that *you don't have to retype the program when you want to replay it!*

Take a Closer Look at That Program

Get out your magnifying glass and really examine that program.

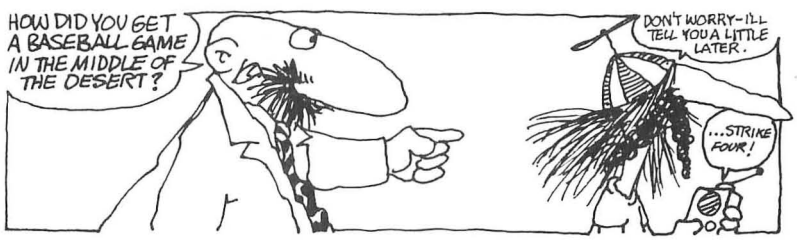
You'll see that line 10 of this simple program is made up of three *statements*. Each statement is separated by a colon (:).

The first statement is: **FOR X = 1 TO 20**. This tells your Atari to sequentially count from 1 to 20.



The second statement is: **SOUND 1,X,10,8**. This tells the computer to sing its warbling song.

The third, and last statement in line 10 is: **NEXT X**, which tells your Atari when to count the next number.



Here's how line 10 of this program reads in English: "Hey, computer, count from 1 to 20, and change the note in the SOUND command accordingly. When you're finished counting, go to the next line."

Line 20 contains one simple statement: **20 GOTO 10**. This tells your computer to *go to* line 10, where the counting and warbling begin again.

Old Lines for New

Let's get obnoxious and break the Wacko Bird program up by using four line numbers, one line number for each statement. But before you can break up this happy program you'll have to get rid of it — either by replacing it or totally annihilating it!

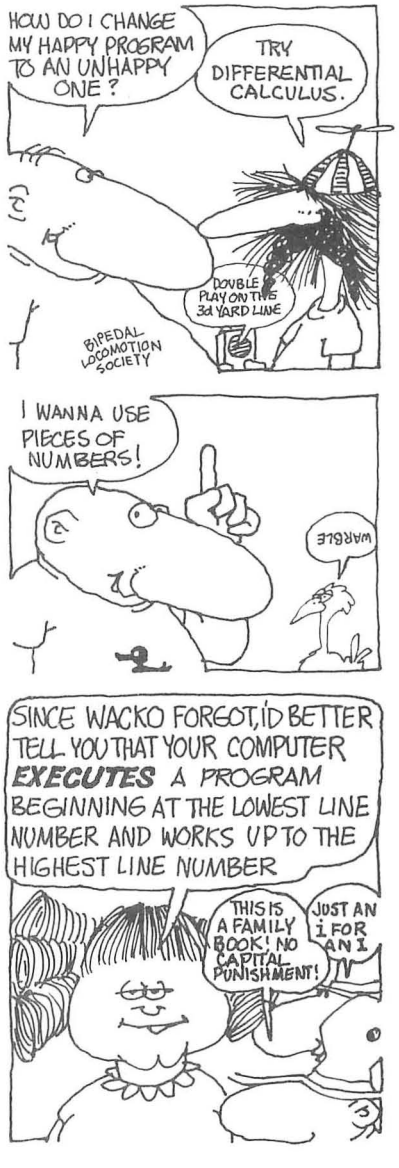
Junior's method of changing a line, *replacing the old line with a new one,* works just fine. So, to change a line of programming, all you've got to do is type its replacement and press RETURN. Your new line will replace the old one. By the way, you can use any whole number as a line number.

For example, you can assign line numbers 1, 2, 3 and 4 to your program. But expert programmers like me number programs by tens; 10, 20, 30, 40. This leaves me plenty of room between each line to squeeze all my creative second thoughts into the program sometime later.

Now that you know all about line numbering, let's revise line 10 of the Wacko Bird program. Here's the original program again, in all its fine-feathered glory:

```
10 FOR X = 1 TO 20:SOUND 1,X,10,8:NEXT X
20 GOTO 10
```

Now, just type this new program line and press RETURN:



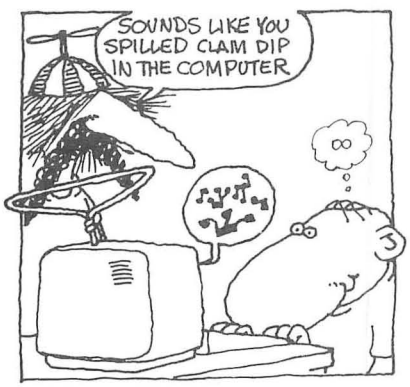
10 FOR X = 1 TO 20

Did you do it? If you did, that long line 10 with all those wild and crazy statements has just been replaced with a real shorty.

Now, let's assign line numbers to each statement in the original line 10, and a new line number to the one statement in line 20.

Here's how your new, improved, and spiffier program looks when you're finished (Don't forget to press RETURN after you type each line of programming!):

```
10 FOR X = 1 TO 20
20 SOUND 0,X,10,8
30 NEXT X
40 GOTO 10
```



This new Warbling Bird sings *exactly* the same tune as the one from the two-line program you started with; even though it now has four program lines! If you don't believe me, check it out by typing RUN and pressing RETURN.

Do you remember how to turn it off? You do? Well, before Captain Action gets violent, *turn it off!!!*

CLEAR up this Mess

By now your screen is probably full of program lines, STOP signs, and other nonsense. Pretty confusing, isn't it? Well, before your screen becomes a jumbled mess, let's clean it up! Just hold down the SHIFT key, press the CLEAR key, and your screen clears instantly. Go ahead and try it.

Type LIST or L.

Don't panic! Your four-line program is just hiding inside your Atari's memory. To get it to appear on the screen, type LIST and press RETURN. Go ahead, type LIST and press RETURN already!

Wheew, there's your program again. As a matter of fact, any time you want to see your program *listed* on the screen, simply type LIST or its abbreviated form, L., and press RETURN.

If you want to list a program on your printer, type:

LIST "P:" [RETURN]

If you want to list one specific line of your program, (just line 20, for example) simply type the command LIST and the line number you want listed, then press RETURN. Like this:

LIST 20 [RETURN]

To list line 20 to your printer, type:

LIST "P:",20 [RETURN]

That's all there is to it! "But," you may ask (I might not answer), "how do I list a specific number of program lines?" It's simple! First type the word LIST followed by the lowest line number you want listed, then type a comma, and finally the highest line number you want listed. A picture is worth a megabyte of words. So, here's how you list *only* lines 20 and 30 of the Warbling Bird program:

LIST 20,30 [RETURN]

Listing lines 20 and 30 to your printer is a snap!

LIST "P:",20,30 [RETURN]

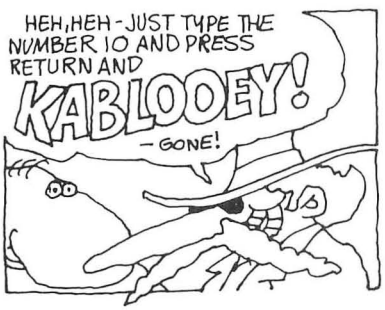
CTRL + 1 = A Screeching Halt!

Later you'll be working with longer listings. And, as the listing scrolls down your screen, you might want to stop it dead in its tracks to look at a line of code in closer detail, do a little editing, or bail out of the computer room. Here's how to make your program listing come to a screeching halt!

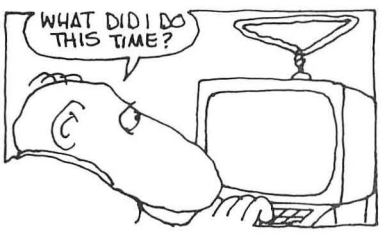
After you type **LIST [RETURN]**, Put your left ring finger on the CTRL key and hover your left middle finger above the 1 key. If you're double-jointed, use any finger you want. To stop the program as it scrolls down your screen, simply press the 1 key. To send it on its way, press the 1 key again. When the program stops, press the BREAK key and lean forward and stare glassy-eyed at your masterpiece.

Total Line Annihilation

I know that you're quick on the trigger. So, now that you've got your program back, let's get vicious and annihilate it, line by line! (Snidely would approve.) To totally annihilate a line of code, simply type its line number and press RETURN. POOFF, it's gone!



If you've followed Snidely's dastardly instructions, line 10 of your four-line program has disappeared, erased completely from your Atari's memory! You don't believe me? LIST the program again; I'll wait while you go through the motions.



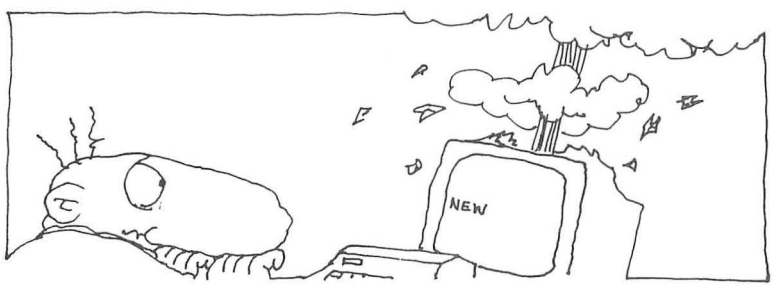
See, line 10 has been ZAPPED. It's vanished. Now, you've really done it!

If you really want to be devious, just type 20 and press RETURN, then 30, and finally 40. When you're through you will have completely annihilated the Warbling Bird.

What's NEW?

If you think that's not nice, just watch while I reveal Snidely's secret weapon: The NEW command. Typing the word "NEW" and pressing RETURN completely and irrevocably erradicates your entire program. Gadzooks!

CAUTION: The NEW command is extremely powerful. Use with extreme care. It not only destroys the Warbling Bird, but completely erases your computer's brain! It's almost as shocking as turning off your computer or opening the cartridge door.



You're Becoming a Real Smarty!

From this point on, I won't always remind you to press "RETURN" after entering each program line or immediate command. You know all about that now!

Now that you know about line numbers, the PRINT command, Higher and Lower Math, Wacko Birds and lots of other stuff, it's time to make your computer go bzzzrk.

Warning: The Three-line Limit

But first, I have to let you know one very important limit to your programs. *You can only enter a maximum of three screen-lines of information after a line number.* When you are about to reach the end of your Three-line Limit the computer gets worried and sounds its buzzer. Type this program to *hear* what happens:

```
10 PRINT "I LIKE TO RAMBLE ON AND ON,
   WHILE I PROGRAM. SOMETIMES THIS GETS
   ME INTO VERY SERIOUS TROUBLE...OOPS!"
20 PRINT "BETTER START A NEW LINE!"
```

Now that you understand all about the Three-line Limit, obliterate this program with the NEW command, and we'll go on to more sensible stuff!

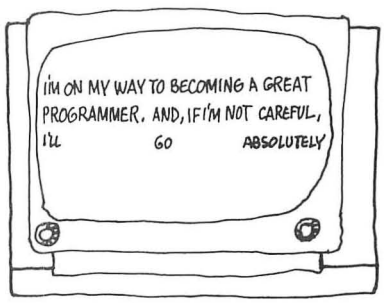
Colons, Commas, Semicolons, and the Remarkable REM

Here's a short program that I'd like you to type in and run:

Absolutely Wacko

```
10 PRINT "I'M ON MY WAY TO BECOMING A
   GREAT":PRINT "PROGRAMMER. AND, IF I'M
   NOT CAREFUL,"
20 PRINT "I'LL","GO","ABSOLUTELY"
30 REM:PRINT "WACKO";
40 REM:GOTO 30
```

After you run this program your screen will look like this:



Pretty weird, isn't it?

Two lines of text are printed on your screen, then (and here's the weird part) the words "I'LL", "GO", and "ABSOLUTELY" are each separated by 10 blank spaces.

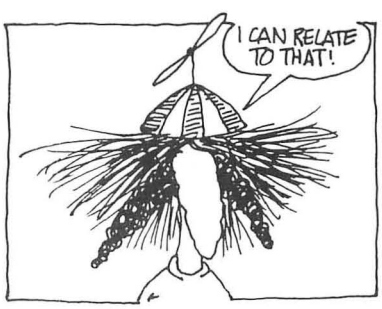
What's going on here? Allow me to elucidate.

The COLON (:)

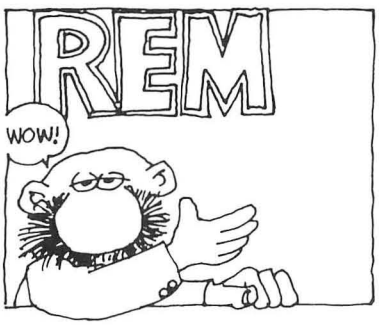
Two PRINT statements are squeezed onto line 10. Each statement is separated by a colon. This accounts for the *two* lines of text you saw on your screen. *Using a colon lets you put more than one instruction in a program line.* I used this nifty space-saving method earlier when I showed you my famous Warbling Bird program. But beware of the Three-line Limit!

The COMMA (,)

Careful placement of the *comma* lets you print your output in columns ten spaces apart. That's why the words "I'LL", "GO", and "ABSOLUTELY" are so spaced out.



Press the SHIFT and CLEAR keys to clean up the mess on your screen, LIST the program, and I'll show you what all this REM stuff is about.



REM (.)

I use a REM statement (or its abbreviated form, a period) at the beginning of a program instruction or a bunch of words when:

1. I don't want that line of code to be executed.
2. I want to leave a message to myself or other wackos about my program.
3. I want to make my program listing easy to read and understand.

I'm a pacifist and have used REM statements in lines 30 and 40 to prevent these two lines from being executed. If you're curious, and just have to see what happens when these REM statements are removed, I'll tell you how to go about it. But please give me a few seconds to leave the room before you run the program!

Here goes! Follow these instructions precisely, and you will cause mayhem and disorder.

1. Hold down the CTRL key and use the up, down, left, and right arrow keys to march your cursor up the screen.
2. Halt when your cursor is positioned directly above the *R* of REM in line 30.
3. Continue to hold down the CTRL key and press the DELETE key four times...until the word "REM" and the colon are completely obliterated.
4. Press RETURN.
5. Repeat this diabolical process again to remove the REM in line 40.
6. Run the program after I run out of the room! EEEEYOW!

WACKO, WACKO, WACKO! That's what you'll see when you run the program after you've removed the REM statements. I can't stand it! Please, I beg you, press the BREAK key and CLEAR the screen before I return to the room.

Whew! Thanks. I'm wacked-out enough as it is.

Semiwacko, or the Semicolon (;) Did It!

Do you see that semicolon at the very end of line 30? You do? Well, that's what caused the screen to fill up with WACKO's!

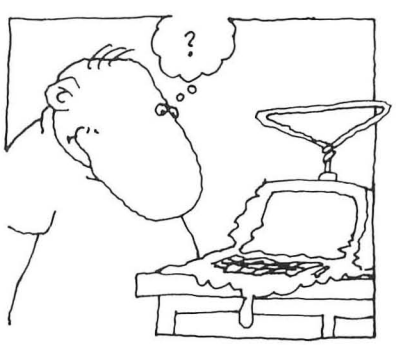
Remove that pesky semicolon. Hold down the CTRL key and move the cursor on top of the semicolon, then press the DELETE key once. Thanks. Now RUN the program again. See, the WACKO's aren't going across the screen any more, just down the screen. Oh, well.

Here's one semipractical use of the semicolon. Press BREAK to stop the program, and clean out your computer's brain with the NEW command. Now enter and RUN this program.

```
10 PRINT "2 CAMELS PLUS 1 MAGIC LAMP = ";
20 PRINT "3 SHEKELS...SOMETIMES"
```

Since you are, by now, semiwacko, you'll want to use lots of semicolons to print stuff from different lines next to each other.

If you've been overtly alert and paying attention (and paying some bribes on the side) most of the mystery of that computer in front of you has probably evaporated.



Let's see, you've entered your first computer programs in the immediate mode, learned about the PRINT command, crawled through Lower and Higher, and Higher Than Lower Arithmetic, made a few statements, entered some programs, watched them acting silly, and banged on the keyboard a lot.

It's Time for a Party!

If you are still standing while sitting and reading this sentence (what?), you've got it made—or you've got

heartburn! You've reached the pinnacle of Basic BASIC Training. All that drill, cleaning sand out of your boots, listening to bird calls, and eating C-rations is over.

But before you get carried away (literally), I'm going to introduce you to two important programming concepts. Then you'll be ready to hop on your camel, charge off to the Great White Expanse, and change that magical computer of yours into all sorts of weird and wondrous things.

Important Concept #1: NUMERIC VARIABLES

Variables are, you guessed it, things that vary or change.

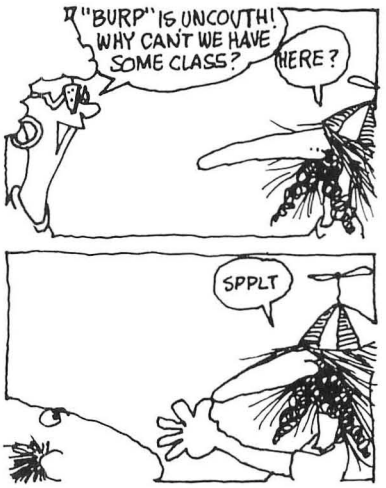
Some examples of things that *do* vary are:

- My waist size after I eat too much falafel
- Petunia's hair styles
- Captain Action's beer consumption
- The price of a good camel
- Ms. Peeky's boyfriends
- Time

Captain Action uses his computer to keep track of his Bug Byte Beer consumption. To make things simple, he chose the shortened *variable name,* BURP, to represent the number of cans of beer he has consumed.

BURP is not as outrageous as it sounds. You can assign any name or abbreviation you want to a variable.

One Wacko word of caution though: Don't use two words with a space between them as your variable name. **TWO WORDS** will just confuse your computer. Write it like this, **TWOWORDS**, and you'll have no problem.



You're right, Ms. Peeky. If Captain Action was a real gentleman, he might have assigned the variable name CANS, BEER, or...BLAZZT!

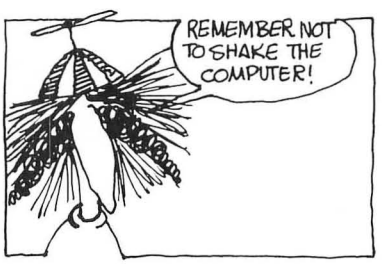
LET's BURP along with Captain Action

Turn on your fingertips again, and we'll BURP along with Captain Action. Captain Action consumed 99 cans of Bug Byte Beer today, and LET his computer know about his gluttony by typing:

```
LET BURP = 99
```

Now because the variable BURP has been assigned a value of 99, his (and your) Atari is filled with 99 cans of beer.

You Don't Have to Use LET



Your Atari is really smart. You don't have to use LET to let it know that a value is assigned to a variable. You can write it like this: **BURP = 99**, and your Atari will know what you're talking about. But, in general, it's a good

idea to LET your Atari know officially you're assigning a value to a variable.

Now before your computer gets tipsy, tell it to:

```
PRINT BURP
```

It will respond by (politely!) belching the number 99.

Hmm, before things get too messy, let's get out of the immediate mode and write a program using the variable BURP.

Burp

```
10 BURP = 99
20 PRINT "CAPTAIN ACTION DRANK "; BURP;
   " CANS OF BEER!"
```

Your screen will look like this after your run the BURP program:

Since BURP is equal to 99, 99 is printed on your screen when line 20 is executed. Replace the 99 in line 10 with your favorite number, and RUN the program again...hic!

Semicolons Are Surrounding BURP!

Do you see the semicolons surrounding the variable BURP? The first semicolon prints "99" after the word "DRANK", and the second semicolon insures that the word "CANS" is printed on your screen directly after the "99." *Always surround a variable with semicolons when it's embedded in a PRINT statement.* Did you notice the blank space I left after the word "DRANK" and before the word "CANS"? They make room for all that beer!

BURP is a NUMERIC Variable

BURP is a *numeric variable* because you assigned a *number* to it. Here are some examples of numeric variables:

1. My waist size after eating to much falafel: INCHES = 52
2. The number of Petunia's hair-style changes in an average week: STYLES = 12
3. The price of a good camel: SHEKELS = 120
4. Junior's grades: GRADES = 0

Ms. Peeky's got a point. All those variable examples do deal with numbers. In the BURP program, BURP equals the *number* 99.

Important Concept #2: STRING VARIABLES, or, the Verbal Variable

In Atari BASIC there's one other type of variable, called a *string variable*, that's just perfect for invariably verbal people like Ms. Peeky.

What's a String?

A *string* is no more than a stringing together of letters, characters, or spaces. Each character, word, space, sentence, and paragraph on this page is a string.

Using string variables is lots of fun! They let you get really verbose and obnoxious, like me! Also, string variables are easy to use. Even the Wacko Cats (Keys, Paddles, and Joystick) like to play with string.

But First, Tell Your Atari

To tell your Atari that you are using a string variable, rather than a numeric variable, just put a dollar sign ($) after the variable's name and surround the string with quotation marks like this:

```
A$ = "BELCH!"
```

In this example the *string variable name is A$, the string is BELCH!*

Your variable name doesn't have to be "A", any single letter, or short word followed by a dollar sign will work just fine. Here's what string variables can look like:

```
NAME$     = "STRING"
INCHES$   = "FIFTY-TWO"
TIME$     = "2 P.M."
NAME$     = "Dr. C. Wacko"
GRADES$   = "0"
A$        = "BELCH!"
```

Numbers, when they're surrounded by quotation marks, become strings. Miraculous!

Making Room for a "BELCH" with a DIMension Statement

Before you can use a silly-looking string variable, you

first have to make room for it inside your computer's memory. To do this, you use a DIMension statement.

Since the string "BELCH!" contains six characters, (including the exclamation point!) DIMension A$ like this:

```
DIM A$(6)
```

If you want to play it safe, you can set aside (DIMension) more room than you need. But if you're a C.M.C.A (Computer Memory Conservation Addict) like me, make room for the *exact* number of characters and spaces that make up your string. Easy, isn't it?

Let's write a short program using a DIM statement and the string "BELCH!." Stand back...here it comes.

```
10 DIM A$(6)
20 A$ = "BELCH!"
30 PRINT "CAPTAIN ACTION DRANK ";A$;
    " CANS OF BEER"
```

Here's another example:

```
10 DIM A$(6), B$(14)
20 A$ = "BELCH!"
30 B$ = "Captain Action"
40 PRINT B$;" drinks too much beer...";A$
```

Now, let's get real tipsy and use numerical and string variables in one stupendous and horrific program.

Horriffic Program

```
10 DIM A$(6), B$(14)
20 A$ = "BELCH!"
30 B$ = "Captain Action"
40 BURP = 99
50 PRINT B$;" drank ";BURP;" cans of beer...";A$
```

Now Type CLR to Clear up This Mess!

Numeric and string variables are two of my favorite programming concepts. I'll be showing off while showing you how to use these two phenomenal concepts throughout the rest of this phenomenal book. But for now, I'd like to show you what happens if you don't clear out your computer's memory after you've assigned strings and numbers to variables.

First, enter and RUN the HORRIFIC PROGRAM. READY? That's what your computer should say after you've followed my instructions.

Now, enter the following in the immediate mode:

PRINT A$

Oopps! Your Atari just belched again! Now, try this in the immediate mode:

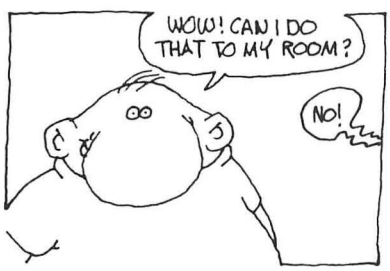
PRINT B$

My best friend's name! And, this:

PRINT BURP

Unfortunately, once you assign a repugnant name or number to a variable, your computer remembers it forever—almost.

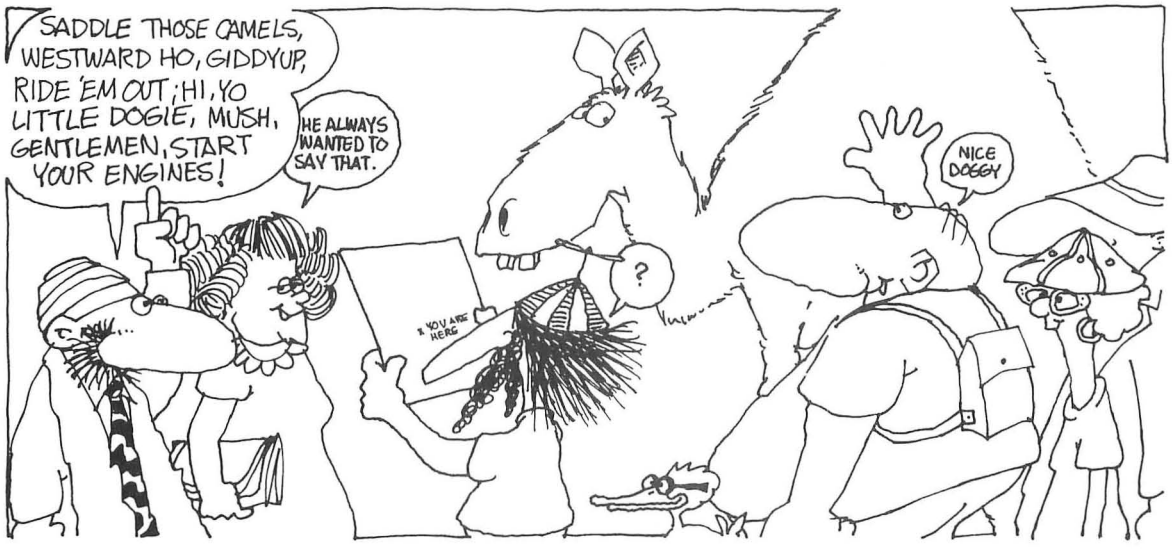
To clear up this mess, and clean up your computer's act, just type:



CLR [RETURN]

Now, your computer's variable tables are as empty as the desert. But finally, it' s time to leave boot camp and push on to the Great White Expanse. So put on your sunglasses, hop on your camel, and we're off!!

3
Entering the Great White Expanse



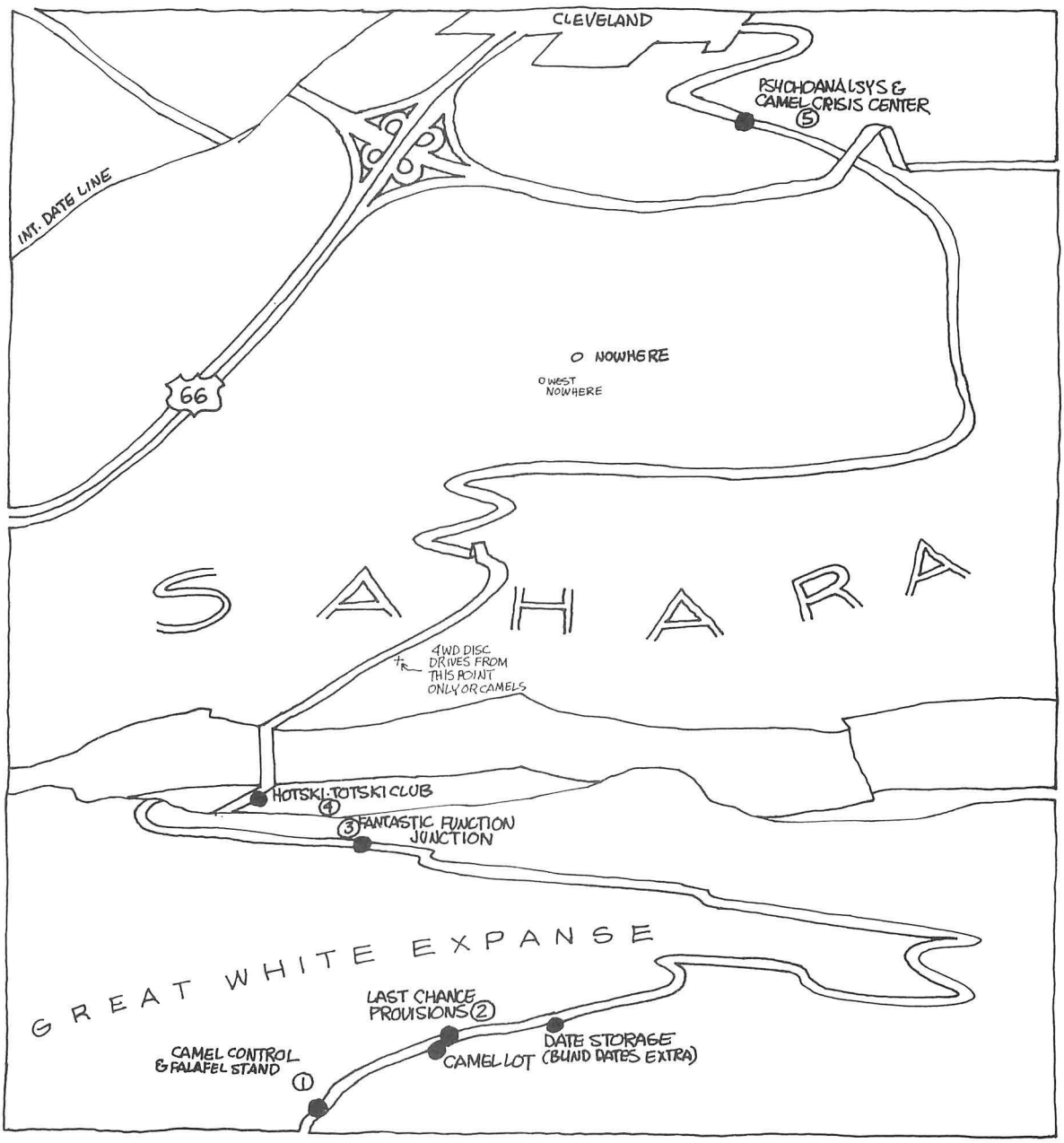
Nice to see that you made it through Basic BASIC Training! You made it without going AWOL, your hair's starting to grow back, and you have a working knowledge of some useful BASIC programming tools. Now, it's time to join the caravan and enter the Great White Expanse.

You'll build your BASIC vocabulary with each step through these pages. Every page is like a small oasis, presenting short, and if you're hungry, easily digestible programming treats.

As you journey through the Great White Expanse you'll discover exciting BASIC words, with simple examples that show you how to use each one to talk to your computer and get your programs under control.

Before you go charging off, let's take a quick look at this map.

The Great White Expanse



Oasis 1: Control That Camel. When the caravan stops at this oasis, you'll have a snack and learn all the BASIC words and commands that will get your program under control. You wouldn't want it to run off without you, would you?

Oasis 2: Provision Your Caravan and Store Your Dates. This oasis is loaded with more tasty delights, just ripe for the picking. Here, I'll show you how to store your treats inside your computer so you can continue across the Great White Expanse with a full stomach.

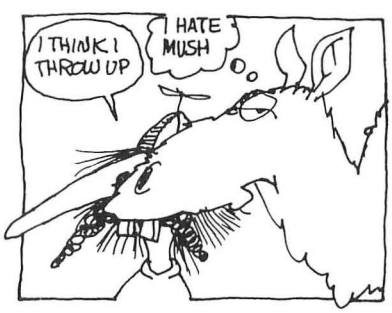
Oasis 3: Fantastic Function at the Oasis. Its time for a little revelry. At this oasis, you'll learn three BASIC functions, SGN, INT, and RND, that are guaranteed to add a touch of professionalism and wackiness to your programs.

Oasis 4: Don't String Me Along, Including Strings Revisited, ATASCII Codes, and the Hotski-Totski Chart. The sun will really get to you by the time you arrive at this oasis. When we arrive, we'll take a closer look at all the fun things you can do with strings. Then, I'll introduce you to the frivolous Hotski-Totski Charts!

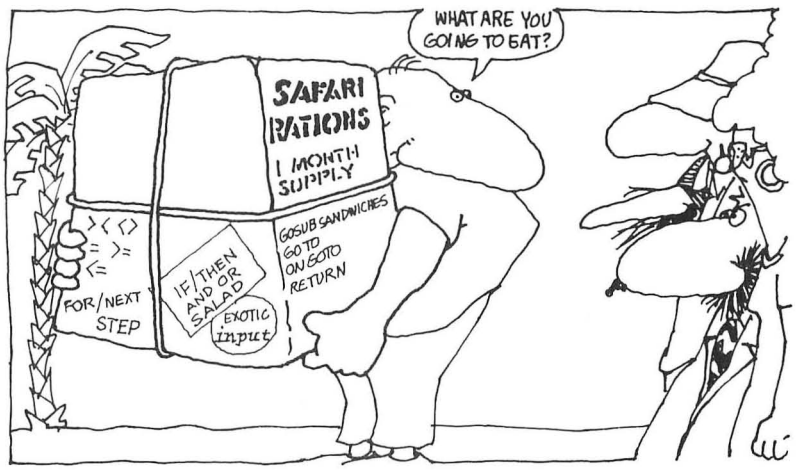
Oasis 5: Chat with Your Fellow Travelers and Psychoanalyze Your Computer. Your journey is almost over. And as the sun sets over the Sahara, it's time to swap camel jokes, talk to your programs, disk drive, and cassette recorder, then peek inside your Atari's brain and poke around.

After you've gathered a gigantic BASIC vocabulary and charted each oasis, you'll fly off (on your magic carpet, of course!) to the mystical worlds of Graphics, Sound, and Creative Programming where you will reinforce and expand your new knowledge. And here's the best part: If you want a quick review, you can always revisit any oasis to sip a refreshing drink from its well of knowledge.

Have you packed your sunglasses? Ready to join the caravan through the Great White Expanse? Then put on your muklucks, and we'll get going! Mush, Clyde!



Oasis 1: Eat Some Falafels and Control Your Camel



Welcome to your first oasis. You must be a little bushed and hungry. So, pull up a cushion, make yourself comfortable, eat a few falafels, and learn how to control that camel of yours.

Here beneath the swaying palms, I'll show you how to combine your Basic BASIC Training skills with some new BASIC words to increase your programming prowess and really take control of your programs.

The Querulous INPUT

You've probably heard the expression "garbage in, garbage out." Well, the fabulous INPUT statement lets you make this expression become a reality. When you use INPUT, your computer asks a weird question and waits politely for you to enter an even stranger answer, and press RETURN. Simply put, *an INPUT statement lets you (you guessed it) put information into your computer.*

Here, I'll show you how INPUT works in a short, but fattening, program:

Falafel Calorie Counter

```
10 PRINT "How many falafels did you eat";
15 .
20 INPUT FALAFELS
25 .
30 CALORIES = FALAFELS*200
40 PRINT: PRINT CALORIES;" Calories! Ms.
   Peeky wouldn't ";
50 PRINT "approve!"
```

The Falafel Counter program was one of Ms. Peeky's inventions. She originally designed it to embarrass everyone at the oasis so she could sneak extra falafels on the sly. Still, she did a great programming job! Besides demonstrating the INPUT statement, she:

1. Provided a nomad-friendly user's guide. In line 10, Ms. Peeky was thoughtful enough to tell us what information to type into the program. If she hadn't made her program nomad-friendly, all you'd see on your screen would be a lonely question mark.

Ms. Peeky also added a semicolon (;) at the end of the PRINT statement so the question mark (generated by the INPUT statement) appears on your screen right after she asks you how many falafels you ate.

Take out the semicolon, run the program again, and watch what happens to that querulous question mark. Wheew! Ms. Peeky also helped when....

2. She made her program easy to read. In lines 15 and 25, Ms. Peeky used a period (.) instead of REM to make the INPUT statement on line 20 stand out.

She also went to a lot of extra trouble when she chose "FALAFELS" and "CALORIES" as her variable names. Her choices make the program easier to understand. She could have used abbreviated forms, like *F* for

"FALAFEL," and *C* for *"CALORIES."* You may also notice that....

3. Ms. Peeky was overwhelmed in line 30. She made the numeric variable "CALORIES" equal to "FALAFELS*200", and then imbedded "CALORIES" in the PRINT statement on line 40.

Using INPUT with a String Variable

When you INPUT a string variable you've got to DIMension it first.

Here's one of Paddle's favorite programs:

No Fishing at the Oasis!

```
10 DIM FISH$(30)
20 PRINT "Hey cat! What kind of fish is that"
25 .
30 INPUT FISH$
35 .
40 PRINT :PRINT "This isn't a fish. It's a ";FISH$
45 .
50 PRINT :PRINT "Can't you read?"
60 PRINT "It says, NO ";FISH$;"ING!!"
```

In line 10, I DIMensioned FISH$ to accept up to thirty characters (DIM FISH$(30)) because I couldn't think of any sea creatures with names longer than that. Can you? If you can, reDIMension FISH$ (something smells fishy here!).

When the Oasis Warden asks you what kind of fish you've caught, have some fun and enter a strange name, like "LOBSTER", "WACKO", or "MS. PEEKY"!

The Easy to Use GOTO

Even if Clyde's not going anywhere, your program can, with the easy-to-use, and utterly simple, GOTO statement. Try this:

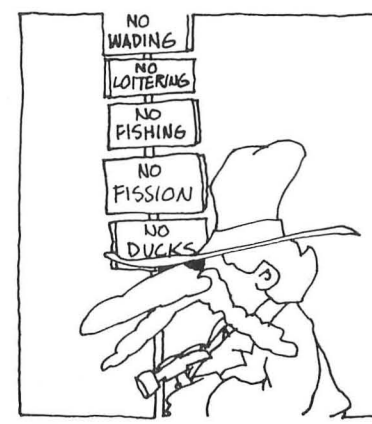
```
10 PRINT " Buy An Edsel! ";
20 GOTO 10
```

Besides filling up your screen with advertisements for defunct products, the versatile GOTO statement can spiffy up programs like the No Fishing gem you just played with.

Every time you want to use the No Fishing program you'll have to type the word RUN and press RETURN. What a waste of time! It's also hard on the fingernails. Well, fear not, GOTO to the rescue! Add the following new line 70 to the No Fishing program and all your nail-splitting problems will be solved:

```
70 GOTO 20
```

When you've finished razzing the Oasis Warden, you'll notice that the program goes to line 20 and you can go fishing again!



Here's another short example:

Another Short Example

```
10 ? "How many shekels did your camel cost"
20 INPUT SHEKELS
30 ? :? SHEKELS; " smackers! Great deal!"
40 .
50 ? :GOTO 10
60 REM — A quick reminder: ? is the same as
     PRINT
```

See how much time the GOTO statement saves you? But that's not all! You can also use GOTO to count your shekels.

Counting Shekels

```
 5 .We start with 1 shekel. (Initialize the variable)
10 S = 1
15 Now we print our balance
20 PRINT S
25 Shekels equals shekels Plus 1
30 S = S + 1
35 Back to line 20 to begin again
40 GOTO 20
```

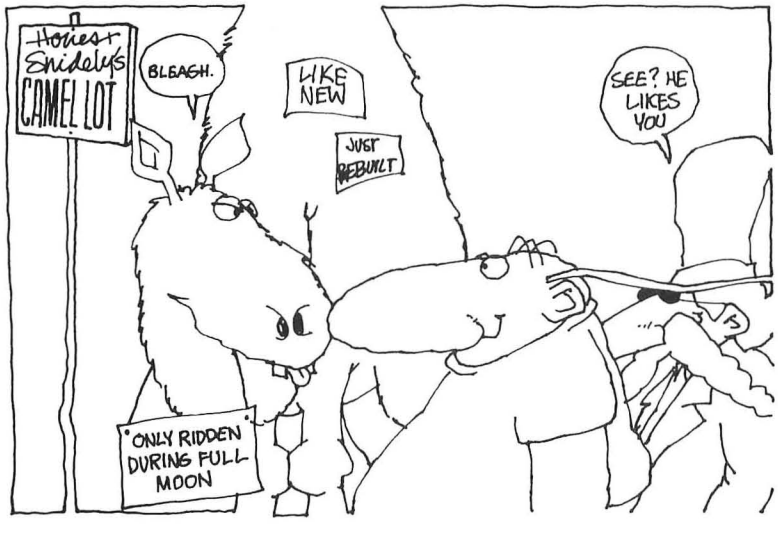
(To keep things manageable, I've used *S* to represent "Shekels".)

Accumulating the Maximum

I play the Game of Life to *accumulate* the maximum number of positive camels! Unfortunately, Snidely Seersucker doesn't play by the same rules. His goal is to sell the maximum number of used camels, and accumulate the maximum number of shekels! Snidely's dastardly accumulations are made possible with a little help from the GOTO statement.

Regardless of whether you take your camel with one hump or two, the best way to experience Snidely's accumulating experience is to walk on to his Used Camelot and start buying a few camels.

So, first enter Snidely's Used Camelot, lose all your money, get in debt, and I'll counsel you when you return.



Honest Snidely's Used Camelot

```
 10 SHEK = 5000:CAM = 1:TOT = 0:SPEND = 0
 20 PRINT "***SNIDELY'S FRIENDLY USED
    CAMELS***"
 30 PRINT :PRINT "WHAT DO YOU OFFER FOR
    CAMEL #";CAM;
 40 INPUT SPEND
 50 PRINT :PRINT "YOU NOW, (SNICKER) OWN"
    ; CAM; "  USED CAMEL(S)"
 55 .
 60 CAM = CAM + 1:SHEK = SHEK - SPEND:
    TOT = TOT + SPEND
 65 .
 70 PRINT "YOU'VE GOT ";SHEK; " SHEKELS
    LEFT"
 80 PRINT "THAT'S $";TOT; "  WISELY SPENT.
    (SNARK!)"
 90 PRINT
100 GOTO 30
```

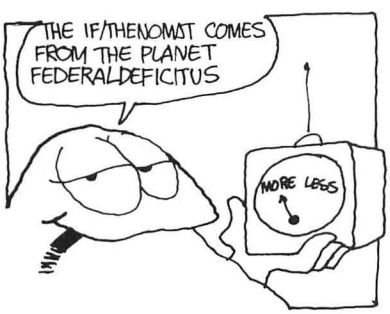
A Tour through Camelot

SHEK:	The number of shekels you have (or don't have!).
CAM:	The number of each camel, and the number of camels purchased.
TOT:	The *accumulated* total
SPEND:	The amount you foolishly spend on each camel
SNARK:	A dastardly exclamation.
SNICKER:	Snidely's way of saying hello.

In line 10, all the variables are "initialized" (set to their beginning values). So, you started with 5000 shekels, and if you weren't careful and paid too much for each used camel, you quickly discovered that you were in debt to Snidely. One of his nastier tricks.

Line 60 is where Snidely keeps track of how many camels he's sold, **CAM = CAM + 1;** keeps tabs on how much money you spend and have left, **SHEK = SHEK – SPEND** and accumulates all his ill-gained wealth, **TOT = TOT + SPEND.**

The Dynamic, Decision-Making IF/THEN



It's a good thing that we brought the brilliant IF/THEN statement along with us on our trek across the Great White Expanse. IF/THEN is really the brains of this motley outfit. It makes most of our decisions, and we'd be lost without it.

The IF/THEN statement compares two things and divines the truth. It has a simplistic, but effective,

philosophy. To the IF/THEN, something is either true or false. But before you use it, you need to know how to indicate comparisons.

>	Means *greater than*
<	Means *less than*
=	Means *equal to*
> =	Means *greater than or equal to*
< =	Means *less than or equal to*
< >	Means *not equal to*

How the IF/THEN makes Comparisons

- **IF A > B THEN** Means *If A is greater than B*
- **IF A < B THEN** Means *If A is less than B*
- **IF A = B THEN** Means *If A is equal to B*
- **IF A > = B THEN** Means *If A is greater than or equal to B*
- **IF A< = B THEN** Means *If A is less than or equal to B*
- **IF A < > B THEN** Means *If A is not equal to B*
- **IF A = X AND B = X THEN** Means *If both A and B are equal to X*
- **IF A = X AND B = Y THEN** Means *If A equals X and B equals Y*
- **IF A = X OR B = Y THEN** Means *If A equals X or B equals Y*

What Happens THEN?

After the IF/THEN statement makes a comparison and divines that it is true, your computer does whatever you put after THEN. If the comparison is false, your computer moves on to the next program line.

Take out the following magic lamp and rub it a few times; the mystery of the IF/THEN statement will be revealed.

ASK THE GENIE

```
10 PRINT "How many wishes do you want";
20 INPUT WISHES
25 .
30 IF WISHES >3 THEN GOTO 50
35 .
40 PRINT "Your wish is my command.":
   GOTO 10
45 .
50 PRINT "Aren't you being a little
   greedy?":GOTO 10
```

IF the comparison in line 30 is true (you got greedy and asked for more than three wishes), **THEN** the program goes to line 50, and the Genie tells you off!

IF the comparison in line 30 is not true (you asked for three or less wishes), **THEN** the program skips on to line 40, and your wish is the Genie's command.

The Use 'm All Program

Ask the Genie shows how the IF/THEN statement makes a greater than (>) comparison. The Use 'm All program uses *virtually* all the IF/THEN comparisons possible. Note that the INPUT statement on line 20 of this program lets you enter and compare *two* numbers. You can enter your choice of numbers in either of two ways:

1. **Enter the first number, press RETURN, then enter the second number and press RETURN again.**
2. **Enter the first number, then enter a comma; enter the second number and press RETURN.**

This short program will surprise you with its brilliance, regardless of which method you use to enter your two numbers. So type it in, enter some numbers, and watch the amazing results.

Use 'm All

```
10 PRINT "Enter ANY two (2) numbers"
20 INPUT N1,N2
30 IF N1 = N2 THEN GOTO 100
40 IF N1 < > N2 THEN GOTO 200
50 IF N1 > N2 THEN GOTO 300
60 IF N1< N2 THEN GOTO 400
70 IF N1 = 50 OR N2 = 100 THEN GOTO 500
80 IF N1< 100 AND N2 >100 THEN
   GOTO 600
90 PRINT :GOTO 10
92 .
95 .
97 .
100 PRINT "N1 EQUALS N2":GOTO 40
200 PRINT "N1 is NOT EQUAL to N2":
    GOTO 50
300 PRINT "N1 is GREATER than N2":
    GOTO 60
400 PRINT "N1 is LESS than N2":GOTO 70
500 PRINT "Either N1 equals 50 OR N2 equals
    100":GOTO 80
600 PRINT "N1 is LESS than 100 AND N2 is
    GREATER than 100":PRINT :GOTO 10
```

Here's what your brilliant computer says when you enter the numbers 50 and 500:

```
N1 is NOT EQUAL to N2
N1 is LESS than N2
Either N1 equals 50 or N2 equals 100
N1 is LESS than 100 and N2 is GREATER than
100
```

Pretty smart, isn't it?

IF/THEN String Comparisons

There's some exciting news at the casbah! The IF/THEN statement can make more than just numerical comparisons. It can be used to compare two or more strings! Later, when we revisit strings, you'll learn lots of weird ways to get tangled up. But for now, here's a neat example that illustrates one common usage of IF/THEN string comparisons.

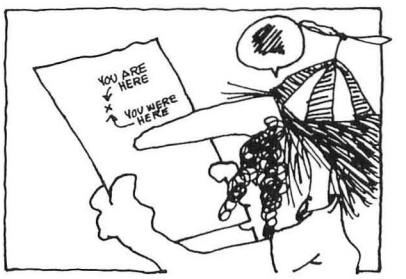
Ryde Clyde

```
10 DIM NAME$(20),A$(1)
20 PRINT "What's your first name";
30 INPUT NAME$
40 PRINT NAME$; ", would you like to ryde
   Clyde";
50 INPUT A$
55 .
60 IF A$ = "Y" OR A$ = "y" THEN GOTO 80
65 .
70 PRINT "I don't blame you!":PRINT:GOTO 20
80 PRINT "Outrageously courageous, ";
   NAME$;"!":PRINT:GOTO 20
```

Here's how this outrageous program works. Line 50 waits for and accepts your INPUT. Line 60 is where all the decision-making occurs. If you enter a string that begins with either a capital *Y* or a lowercase *y* the program assumes you say, "YES" and zips outrageously to line 80. But if you enter any other letter, the program *assumes* that you mean "NO," and bops to line 70 where you are praised for your common sense!

The IF/THEN statement's decision making and logical branching abilities are key programming elements. As you continue your adventure and begin developing your own brilliant programs, the IF/THEN statement will become one of your most valuable tools — almost as valuable as a date-picker.

Going around in Circles with the Dizzy FOR/NEXT Loop



It's easy to get lost out here in the desert. Caravans have been known to go around in circles, repeating the same course for days, before getting back on track again.

Getting dizzy is very unhealthy for caravans. But going around in circles can actually improve and enhance your programs! When you need to repeat part of your program a specific number of times, the dizzy FOR/NEXT loop is just what the shiek ordered.

The FOR/NEXT loop has many camel-boggling applications. But since a picture is worth a thousand shekels, here's a very short, and eye-popping, example. Some of the elements contained in this short eye-popper might be new to you. Don't panic! Take a chance and type it in anyhow. The results are well worth the risk.

Eye-Popper

```
10 FOR X = 1 TO 255
20 POKE 710,X
30 NEXT X
```

Don't be squeamish! Open your eyes wide and run this short program.

Wasn't that fabulous? Your screen looked like the aora-boringallofus, and your eyes are probably bloodshot!

To set your screen back to blue, just enter the following in the immediate mode: **POKE 710,148**

Thanks to the FOR/NEXT loop, you've just sampled all the brilliant colors your Atari can deliver. Later, I'll show you how to astound your camels with more

psychedelic color wizardry. But now it's time to show you how the FOR/NEXT loop made it all possible.

The Eye-Popper program is real easy to understand. The dizzy FOR/NEXT routine begins on line 10 with the statement **FOR X = 1 TO 255**. This simply means that the value of X equals 1 when the program starts, and will equal 255 when FOR/NEXT is finished looping around.

All that snazzy color-changing takes place in line 20. The value of X in **710,X**, is changed from 1 to 255, the current value of X, by the FOR/NEXT loop that surrounds it. Line 30, **NEXT X**, tells the program to go back to line 10 and add 1 to the value of X.

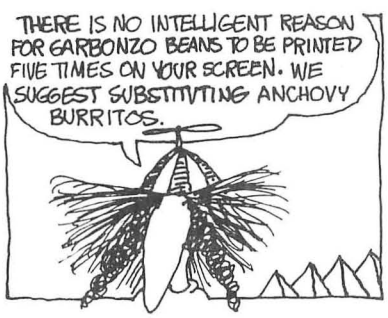
A FOR/NEXT loop can be thought of as a garbanzo bean sandwich — a handful of programming statement(s) squashed between a FOR and NEXT statement.

Here's another short example that will help you get the picture — or heartburn.

Garbanzo Beans

```
10 FOR A = 1 TO 5
20 PRINT "Garbanzo beans"
30 NEXT A
```

See! The garbanzo beans *are* squashed between a FOR and NEXT statement! When you run this tasty program, the words "Garbanzo beans" are printed on your screen five times.



If you really like garbanzo beans, just change the number 5 in line 10 to a larger number and run the program again.

Now, to get real sneaky, add this line and run your program.

```
25 PRINT A
```

Wow! The value of A increases by 1 each time the program takes a bite of the sandwich.

I've used the variable A instead of X in this example. You can use any variable you like. You've just got to be consistent. If you use A after the FOR statement, use it again after the NEXT.

Stay in STEP with More FOR/NEXT Loop Madness

The STEP statement is used to make a FOR/NEXT loop count in increments other than 1.

Here's a high-step'n example that illustrates this wondrous delight:

High Step'n

```
10 FOR X = 0 TO 20 STEP 2
20 PRINT X
30 NEXT X
```

Run this spiffy number and you'll see that the program is now counting by twos!

To really grasp this concept, make it count by fours by changing the number that follows STEP to a 4.

You can also use STEP to count *down* by subtracting instead of adding to X. Here's an example that counts down from 20 to 1 in one-step increments:

Countdown

```
10 FOR X = 20 TO 1 STEP —1
20 PRINT X
30 NEXT X
```

Experiment with these concepts a little until you have a good feeling for the way STEP works in a FOR/NEXT loop. Remember, the number following STEP can be either positive or negative. It can even be a decimal, like 0.5!

Now Just Wait a Minute!

Now that you're familiar with the FOR/NEXT loop and STEP, here's a common application I call Wait. It's one that you'll use often in your programs.

If things are happening too fast in your program — the display doesn't stay on the screen long enough, a sound is too short, or you want a color to remain on the screen for a fixed period — it's time to use my infamous Wait routine.

Wait

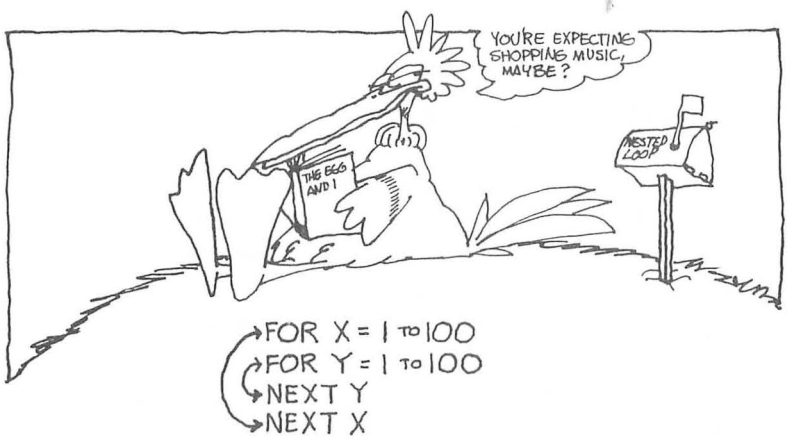
```
10 FOR X = 1 TO 10
20 PRINT,X
25 .
30 FOR WAIT = 1 TO 100
40 NEXT WAIT
45 .
50 NEXT X
```

When you run the Wait program you'll notice a pause before the computer prints each number on your screen. That's because the Wait routine is cleverly inserted at lines 30 and 40 of the program. If you want to shorten the delay, just decrease the value of the number following TO. To lengthen the interval, increase the value.

Nested FOR/NEXT Loops

I'll bet you didn't realize when you typed in the Wait

program that you were entering a "nested loop." A nested loop is simply one or more FOR/NEXT loops nested inside other FOR/NEXT loops.

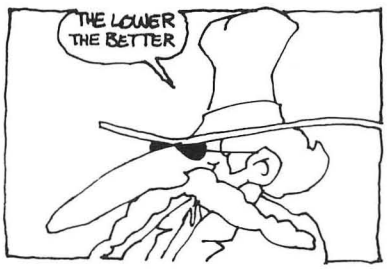


In the Wait program, the WAIT routine on lines 30 and 40 is a nested loop because it's inside another FOR/NEXT loop.

This next program, Clyde's Lament, is one of my favorites. It gets lonely out here in the desert, and a little music is always welcome. So, put on your dancing shoes and RUN this short program. It shows how nested loops are structured, and I guarantee it will make you smile.

Clyde's Lament

```
10 FOR A = 1 TO 10
20 FOR B = 5 TO 0 STEP —1
30 FOR WAIT = 1 TO 20
35 REM
40 SOUND 0,B,4,15
45 REM
50 NEXT WAIT
60 NEXT B
70 NEXT A
```

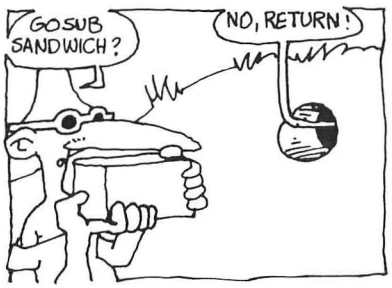


Line 10 of Clyde's Lament is set to play ten times. If you want to shorten Clyde's song, change the number 10 to a smaller number.

Line 20 changes the tone in the SOUND statement in line 40. (You'll learn all about sound in a little while.)

Line 30 is the first part of the WAIT routine. Changing the 20 to any other number will vary the length of Clyde's song. (The shorter the better.)

The Subterranean Subroutine, Starring GOSUB and RETURN



Lurking in the depths of the oasis is the dark and mysterious subterranean subroutine. A subroutine is a chunk of programming that your program uses one or more times to do something special.

When I get hungry I send out for a date-anchovy pizza from the oasis pizzeria. My special treat is delivered in a flash by the local merchant. When your program wants a special treat, it sends its order to the subroutine and, voila, instant delivery!

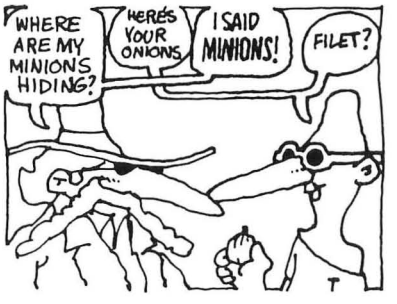
I'll show you a devastating example in just a second. But before you send an order to a subroutine, you should know that GOSUB is used to send the order to the subroutine and RETURN zips your computer back to the main part of your program after the subroutine has filled the order.

It Remembers from whence It Came!

GOSUB works just like GOTO. It sends your program racing to the line number placed after it. And, with a little help from its partner, RETURN, the program remembers where it came from and returns after it has finished executing the subroutine.

Snidely, always the perfect villain, has devised this subterranean example:

Where Are My Minions?




```
10 . The main part of the program:
15 .
20  PRINT "SNIDELY: Where are my minions
    hiding?":GOSUB 60
30 PRINT "SNIDELY: Snark!"
40 END
45 .
50 . The Subterranean Subroutine:
55 .
60  PRINT "LURKING IN THE DEPTHS OF
    YOUR PROGRAM.":RETURN
```

The GOSUB in line 20 tells the program to race down to the subroutine in line 60. After Snidely receives his answer in line 60, the RETURN sends the program back to line 30, where Snidely makes a snide remark.

The END in line 40 is used to separate the main part of the program from the subroutine below it.

Snidely's dastardly program shows how the GOSUB works with RETURN to get to and return from a subroutine. But remember, subroutines come in real handy when you've written a program containing short chunks that it uses repeatedly. Later in this book, when I help you weave the perfect flying carpet, you'll experience more subroutine madness.

The Smart and Speedy ON GOTO

The ON GOTO statement is smart, speedy, and patient. It waits for numeric input, then races to the cor-

rect line number. Here's what a typical ON GOTO statement looks like:

```
ON N GOTO 100,200,300,400
```

In this example, if the value of N is 1, your program races to line 100, the first line number after the command GOTO. If N equals 3, line 300, the third number after GOTO is executed. (Off with its head!)

Enter and run this short program to see how smart the ON GOTO really is.

Smarty Harem Pants

```
10 PRINT :PRINT "Enter a number from 1
   to 5";
20 INPUT N
25 .
30 ON N GOTO 100,200,300,400,500
40 .
100 PRINT "It's a 1! (or a 0 or a number greater
    than 5).":GOTO 10
150 .
200 PRINT "You entered a 2.":GOTO 10
250 .
300 PRINT "You entered a 3.":GOTO 10
350 .
400 PRINT "You entered a 4.":GOTO 10
450 .
500 PRINT "You entered a 5.":GOTO 10
```

See how smart ON GOTO is? The program goes, in order, to the correct line number. Just for fun, scramble line 30 like I've done below, and run SMARTY HAREM PANTS again.

```
30 ON N GOTO 200,300,500,100,400
```

Now, when you enter the number 1, the program goes to line 200! (The first line number after GOTO.)

If you enter a zero or a number greater than 5, the program goes to the first line number on ON GOTO's list. But don't enter a negative number or a number greater than 255. If you do, your computer will get heartburn and spit out an error message. Try it, if you have a strong constitution.

I use the smart and speedy ON GOTO in programs to give the user a choice of options. What's Up includes some nifty programming techniques (in line 10) to clear the screen and turn off the cursor.

This stupendous program also contains a WAIT subroutine in line 1000. Since the program uses a WAIT three times, the subroutine saves a lot of typing and makes this program super efficient!

What's Up

```
  5 . Line 10 clears the screen and turns off the
      cursor.
 10 PRINT CHR$(125):POKE 752,1
 15 .
 20 PRINT "1. What's Clyde's favorite word?"
 30 PRINT "2. What does Snidely say?"
 40 PRINT "3. What is Junior's I.Q.?"
 50 PRINT :PRINT "Enter a number from 1
      to 3";
 55 .
 60 INPUT N
 65 .
 70 ON N GOTO 100,200,300
 75 .
100 PRINT "Grunt!":GOSUB 1000:GOTO 10
120 .
200 PRINT "Snark!":GOSUB 1000:GOTO 10
220 .
```

```
300 PRINT "—35":GOSUB 1000:GOTO 10
320 .
340 . Here's the subroutine!
350 .
1000 FOR WAIT = 1 TO 300:NEXT WAIT:RETURN
```

ON GOSUB

ON GOSUB is just about the same as ON GOTO. A typical ON GOSUB statement looks like this:

```
10 ON N GOSUB 1000,2000,3500
```

In this example, a response of 1 sends the program racing to a subroutine on line 1000. If the response is 3, the program goes to the subroutine in line 3500. After the subroutine has finished doing its thing, the program returns to the next statement after the GOSUB.

In this example, if N has a value less than 1 or greater than 3, the program will omit any subroutine and go on to the next line. Negative numbers, or numbers greater than 255, make the computer display an error message (oops!).

Oasis 2: Provision Your Caravan and Store Your Dates

Congratulations! You just left Oasis 1, one of the longest stopovers in this entire book! You've taken a giant step, written some programs, and learned how to take control of them. This knowledge puts you on top of the sand pile. And, as Captain Action once said, it's all down dune from now on. Wow!

I've been emptying sand out of my shoes waiting for you. Junior's been emptying sand out of his head. So, welcome to your second oasis. First, with a little help from a couple of really wacked-out date nuts, READ and DATA, I'll show you how and where to store all the supplies you've been lugging around. After breezing through a few pages you'll be twirling information in and out of your programs like a Sahara sand storm!

Next, after you've scattered some sand and really made a mess of things, my beautiful and talented wife, Petunia, will make an appearance to help us *all* get organized with her special storage system, *Arrays!*

This bit of desert paradise is chock full of easy-to-understand programming examples that, with a little

imagination, can be expanded into real blockbusters; not to mention something useful. So settle in, and check out the scenery.

Getting Organized with DATA

In many programs there's a real danger of tripping over numbers, strings, and figs at every stop of the way. What's a nomad to do? Just enter Plaudit to Wacko, RUN it, and I'll show you how it solves your storage problems.

Plaudit to Wacko

```
 10 READ X
 20 IF X = - 1 THEN GOTO 50
 30 PRINT X
 40 GOTO 10
 50 PRINT "WHO DO YOU APPRECIATE?
    WACKO, WACKO, WACKO!":END
100 DATA 2,4,6,8, - 1
```

After RUNning this strange program and viewing the humble results, you probably asked yourself, "What's being stored, where is it stored, and what's in store for me next?" I'll answer all these questions in order. After all, this chapter is about getting organized!

What's Being Stored and Where Is It?

The provisions you take with you on your programming adventure include *numbers and strings.* That's what you need to store! And to make your journey a snap, Atari BASIC lets you store them after a DATA statement!

"Wow" Is in Store for You Next

Storing numbers or strings after a DATA statement is just like placing words in a book. For example the word "Wow" has been waiting patiently for you to come along and *read* it. And what's amazing is that you can go back and read it again, and again, and again, and... just like stored information you'll find after a DATA statement, as you'll soon see!

In Plaudit to Wacko, I've stored the numbers 2,4,6,8 and – 1 after the DATA statement in line 100. Here's line 100 again so you can take a closer look at it:

```
100 DATA 2,4,6,8, - 1
```

Numbers or strings are always placed after a DATA statement according to these four simple rules:

1. A comma *always* separates each chunk of data.
2. There is no comma before the first chunk or after the last chunk of data. So, in this example, there's no comma before the number *2* or after the *– 1.*
3. You can store as many numbers or strings on a line of data as you want, as long as you don't exceed the three-line limit we mentioned in Basic BASIC Training.
4. You can position a line, or lines, of data *anywhere* in your program, and the program will still work. Sound impossible? Just for fun, in Plaudit to Wacko, change line 100 to read line 5, and rerun the program. It still works!

Go Bonkers and READ It!

Eeeeyargh! You know that numbers and strings can be stored after a DATA statement. But just like the words in this book, if you don't *read* them, they won't drive you bonkers. If you're fearless, and want your program

to read the stuff you place after DATA statements, you'll have to use the logically literate READ statement!

READ is always used with a variable, like this:

```
10 READ X
```

Here I've used the variable X, but you can use any variable you'd like. When your program reads string data, use a string variable, like A$.

READing with an Invisible Finger

Junior always uses his finger to read. Likewise, READ has a mysterious, invisible finger (called a "pointer") that points to each number or string after the DATA statement.

The pointer moves from left to right (just like your eyes scanning a book) and sequentially reads *each* number or string. As the pointer "looks" at each number or string, the variable following the READ statement is assigned that value.

In Plaudit to Wacko, the invisible pointer first looks at the number 2. The program prints *2* on your screen because X equals 2 when the PRINT statement in line 30 is executed. Next, it points to the number 4, and *4* is printed on your screen. Next, well, I'm sure you get the point!

Just to make sure that you've got it, enter and run this stripped down version of the Plaudit to Wacko program.

Stripped Down Plaudit

```
10 READ X
20 PRINT X
30 GOTO 10
40 DATA 2,4,6,8
```

Ooops, a boo boo! I'll bet you thought your camel was running on high octane and you were home free at the oasis, until the number 8 appeared on your screen, followed by "ERROR 6 AT LINE 10." Uh Oh!

Not to worry. Everybody goofs (except Dr. Wacko). Your computer is just telling you that the pointer ran out of data. After reading the number 8, it fell off the right edge of the program. Plop!

FLAG it Down!

The original program didn't plop because I cleverly placed the number −1 at the end of my data, right after the number 8.

I used −1 as a *flag* to tell the pointer it had reached the last chunk of data. After the pointer read −1, the IF/THEN statement in line 20 told the program to go to line 50, print some nonsense, and come to a screeching halt − END.

Any number (or string) can be used as a *flag*. To see what I mean, modify Stripped Down Plaudit like so:

1. Change line 40 to read: **40 DATA 2,4,6,8,0**
2. Add this line: **15 IF X = 0 THEN END**

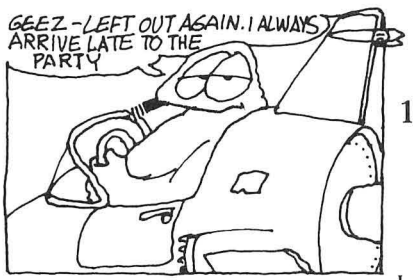
This time I used zero (0) as the *flag*. When you run your modified program, the IF/THEN statement in line 15 stops the program when it is flagged down by the 0.

Strings of Dates and Data

The DATA statement comes in real handy when you want to store a list of names, dates, pomegranates, or brilliant ideas. Here's a motley program full of motley wackos to demonstrate the DATA statement's string handling versatility.

Motley Crew

```
10 DIM NAME$(20)
20 PRINT "Presenting our motley crew:"
30 READ NAME$
40 PRINT NAME$
50 IF NAME$ = "SNIDELY" THEN END
60 GOTO 30
100 DATA DR. WACKO,MRS. PETUNIA
    WACKO,JUNIOR,CAPTAIN ACTION,MS.
    PEEKY,CLYDE,SMOKEY PEEK,SLOW
    POKE,SNIDELY
```



Instead of numbers, this motley program is filled with wackos! We're all crammed into line 100, right after the DATA statement, and each of us is separated by a comma. I've listed my name first, of course, and naturally Snidely Seersucker (that dastardly villain) is last. "SNIDELY" is the *flag* that indicates the end of the list. And because he wanted to see his name printed on the silver screen, I placed the IF/THEN statement in line 50, *after* the PRINT statement. Clever, no?

Juggling More Than One!

You know how to make your program point to and READ one number or string at a time. No problem! Now comes the exciting part. Stand back as I show you how to turn your computer into a data juggling virtuoso! Take a look at this READ statement — if you dare:

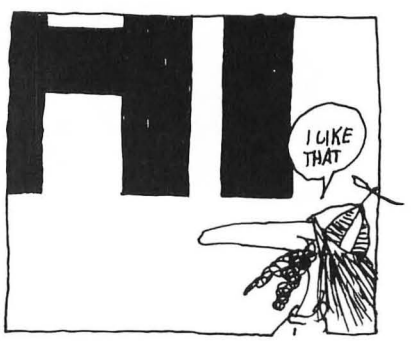
```
10 READ X,Y
```

This strange looking statement contains two, count 'em, two variables separated by a comma! Right again, your computer can juggle more than one variable at a time!

Here, I'll show you. Just enter my infamous Hi! program. When you're all set, clear the screen by pressing the SHIFT and CLEAR keys, then RUN it! I'll explain all, after you've witnessed your Atari's juggling prowess.

Hi!

```
10 READ X,Y
15 IF Y = -1 THEN END
20 POSITION X,Y:PRINT "X"
30 GOTO 10
40 DATA 10,2,10,3,10,4,10,5,10,6
50 DATA 11,4,12,4,13,4
60 DATA 14,2,14,3,14,4,14,5,14,6
70 DATA 16,2,16,3,16,4,16,5,16,6
80 DATA 19,2,19,3,19,4,19,6
90 DATA -1,-1
```



Did you see that? A gigantic "HI!" This nifty program actually juggles two numbers to perform this mystical feat!

First, the pointer reads the value for X, then it switches hands and reads the value for Y. In the Hi! program, starting at line 40, it first reads 10, assigns this value to X, then 2, and assigns this value to Y. Then it reads the second 10, assigns it to X, and finally 3, assigning it to Y. It continues this juggling act until it arrives at the −1 in line 90.

The program reads each set of values, uses the values in the POSITION statement in line 20, and prints an *X* on your screen. It takes a set of numbers to position stuff on your screen, but more about the amazing POSITION statement in Chapter 4.

RESTORE's Also in Store for You!

Out here in the desert, one day seems to blend into the next. Junior has come up with a crude method of keeping track of each day of the week so he knows when to watch his favorite TV shows. At the end of each day's march he notches his skateboard; one notch for Monday, two for Tuesday, and so on. But even my brilliant son gets confused after a while. To help him translate his skateboard notches into days of the week, I have devised this simple but effective program, Notcheroonio. The great thing about Notcheroonio is that it introduces even more exciting DATA tricks!

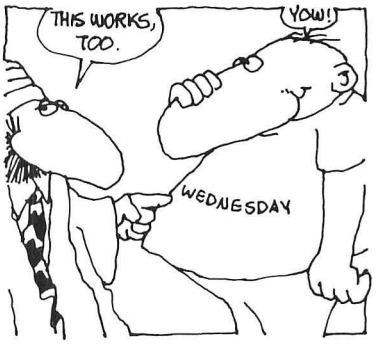
Here's Notcheroonio. Before I explain all the new stuff that's in it, enter it, RUN it, and start having some fun!

Notcheroonio

```
 10 DIM DAY$(10)
 20 RESTORE
 30 PRINT "ENTER A NOTCH BETWEEN 1
    AND 7";
 40 INPUT N
 50 READ A,DAY$
 60 IF N = A THEN PRINT DAY$:GOTO 20
 70 IF N > 7 OR N < 1 THEN PRINT "OOPS! ";
    :GOTO 20
 80 GOTO 50
100 DATA 1,MONDAY,2,TUESDAY,3,
    WEDNESDAY,4,THURSDAY,5,FRIDAY,6,
    SATURDAY,7,SUNDAY
```

I'll bet the first weird thing you noticed about this program was the RESTORE in line 20. RESTORE resets the pointer back to the beginning of a DATA statement.

In line 60, when the number you've entered (N) is the same as the number just read by the pointer (A), the pointer comes to a screeching halt. For example, if you enter the number 3, the pointer reads up to and including "3,WEDNESDAY." Then the computer prints "WEDNESDAY" (DAY$) on your screen, and the program returns to line 20. In line 20 the RESTORE statement returns the pointer to the beginning of the DATA statement so you can try again.



Some programs you may develop might include sets of DATA on different lines. To reset a specific line of data just place the appropriate line number after RESTORE, like this: **RESTORE 100**. In this case, the program will RESTORE only the data beginning on line 100.

Now for the other weird thing about Notcheroonio. Yes, it is possible, as you have just seen, to read string and numeric data at the same time. Just remember to DIMension the string variable and set things up like this:

```
50 READ A,DAY$
```

You've also got to make sure that you alternate numeric and string data, as I've done in line 100. Remember, your program can't read a string into a numeric variable.

Ms. Peeky's Black Book

The other day Ms. Peeky lost her little black book, and I found it. She'd listed all her favorite men in it and included comments on each. What an embarrassing revelation!

Drat! Caught in a Sand TRAP!

There are only two new concepts in her little black book, and they both occur in line 60. Before you peek

at Ms. Peeky's book, here's line 60, complete with new concepts:

```
60 TRAP 80:IF DESC$(1,3) = NAME$(1,3) THEN
   PRINT DESC$:GOTO 30
```

The TRAP statement is used to trap any error that might occur in line 60. (See Appendix A for a complete list of error messages.) If an error occurs, the program moves on to the line that has the same number as the number following TRAP. In this example (**TRAP 80**) the program will go to line 80. This is really great! Now, when an error occurs, your program won't stop, and your screen won't display an error message! It just goes on about its business. Here's a simple example that illustrates the point:

```
10 PRINT "ENTER A NUMBER";
20 TRAP 10:INPUT N
30 PRINT N:GOTO 10
```

In this program, if you goof and enter a letter instead of a number, the TRAP statement in line 20 sends the program back to line 10 so you can try again. No mess, no fuss, and no error messages. Neat, isn't it?

Now, look at those two weird-looking string variables: **DESC$(1,3)** and **NAME$(1,3)**.

DESC$** and **NAME$ by themselves aren't unique at all. But by adding **(1,3)** after each one, they've become Selective Strings!

The two numbers inside the parentheses select a portion of the string variable, in this case, the first three letters. In Ms. Peeky's little black book, this neat trick is used to compare *only* the first three letters of each of her boyfriend's names.

Here's a short example that shows how this concept works:

```
10 DIM A$(20)
20 A$ ="Caravan"
30 PRINT A$(1,3)
```

After you run this program the word "Car" will appear on your screen! That's because the program selects *only* the first through the third letters of "Caravan." To print the word "van", just change line 30 to read: **30 PRINT A$(5,7)** and the fifth through the seventh letters of "Caravan" will print on your screen!

Now that we're out of the sand trap and understand those weird string variables, it's time to peek at Ms. Peeky's Little Black Book. Just enter her friend's first names to see what happens. It's absolutely shocking!

Little Black Book

```
10 . Ms. Peeky's Little Black Book
20 DIM NAME$(10),DISC$(50)
30 RESTORE :PRINT "ENTER HIS FIRST NAME"
40 INPUT NAME$
50 READ DISC$
60 TRAP 80:IF DISC$(1,3)=NAME$(1,3) THEN
   PRINT DISC$:GOTO 30
70 GOTO 50
80 PRINT NAME$;" IS NOT IN MY LITTLE BLACK
   BOOK!":GOTO 30
82 .
85 . Here comes the DATA!
87 .
100 DATA MARVIN MAINSTREAM (A real macho
    guy!),WEIRD HAROLD(Weird me out. Really!)
110 DATA SLOW POKE (A real sweetheart
    xxxx),GRUESOME GEORGE (Good in a clinch.)
120 DATA SNEAKY PEEK (Eeeuk!),CAPTAIN
    ACTION (Faar out!!),WACKO (Gag me with a
    ladle!)
```

Think of Ms. Peeky's Black Book as a rudimentary file cabinet called a *data base*. The program stores information in the DATA statements, points to it, READs it, then displays it on your screen. The best part of this program is its ability to selectively search and retrieve information. By modifing it, you can tailor this short program to perform all sorts of data-storage tasks. Here are some ideas:

1. Store and retrieve names, addresses, and phone numbers.
2. File those mouth-watering clam dip recipes.
3. Store information about your favorite lacrosse team.
4. Organize your record collection.
5. List all your creditors and the amounts you owe them.
6. Keep track of the tamale sauce and anchovy burritos.
7. Store and retrieve the birthdays of all your worst enemies, so you remember not to send them cards.

Use a FOR/NEXT Loop to READ Data

Before I run off and make way for my charming wife, Petunia, I'd like to share one final DATA trick with you.

You can incorporate a FOR/NEXT loop within your data-reading program to let you select the *specific* chunks of data you'd like to read. Here's a final example that illustrates this FOR/NEXT reading trick.

FOR/NEXT DATA Trick

```
10 FOR X = 1 TO 5
20 READ A
30 PRINT A;" ";
```

```
40 NEXT X
50 END
60 DATA 1,2,3,4,5,6,7,8,9,10
```

When you RUN this program, you'll see that it READS only the first five numbers and prints them on your screen. Change the FOR/NEXT loop in line 10 to **FOR X = 1 TO 10**, and the program will present *all* the data on this line!

Now that you know all about DATA statements, it's time for me to go and feed Clyde. Anyway, here comes Petunia to tell you all about *arrays!* Ciao, bambino, see you later!

Presenting Mrs. Petunia Array Wacko!

Actually, arrays are easier to understand than my husband. You just have to be organized, like me. I did most of the packing for this motley caravan and became an expert at storing stuff in wicker baskets. I then labelled each basket so I'd know what was in it.

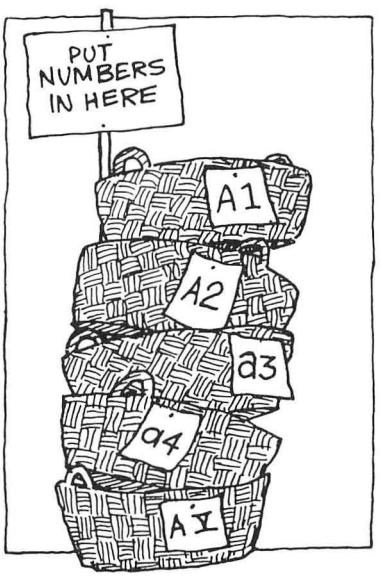
Array names are just like variable names, only different. Variable names can identify a basket full of information. For example, in the expression X = 100, "X"

is the label for a basket that contains the number 100. Variable baskets, however, have one important limitation. They can only hold one piece of information at a time.

But here's the exciting part! Arrays let you store a whole bunch of numbers in individually marked wicker baskets, *without* having to use a whole bunch of different variable names! How is this possible? Simple, I'll show you how it's done. First you have to know how many numbers you want to store. Then you've got to DIMension the array to tell your computer to set aside individual baskets for each of those numbers. For example, if you want to store five numbers, dimension your array like this:

```
DIM A(5)
```

This means that you are reserving five wicker baskets, labelled A(1), A(2), A(3), A(4), and A(5) to hold each of your five numbers.



You don't have to use the letter A; you can use any variable name as the first part of the label, then put a number within parentheses right after the variable name. After you've labelled all your baskets, you can store numbers in each one and know where each number is stored.

I think it's about time to show you how all this works in a short, but organized example named after my brilliant son, Junior.

Basket Case

```
10 DIM A(5)
15 . 1. Set up the array.
20 FOR X = 1 TO 5
30 A(X) = X: . Each value of "X" is assigned to
     the array here.
40 NEXT X
```

```
45 . 2. Put numbers into the array.
50 FOR X = 1 TO 5
60 PRINT "Put a number in basket ";X;
70 INPUT N
80 A(X) = N
90 NEXT X
```

Let's just belly-dance our way through this program.

The FOR/NEXT loop in lines 20 through 40 sets up the array. Line 30 labels each of the five empty baskets A(1) through A(5) respectively. The loop in lines 50 through 90 lets you place one number in each wicker basket.

When you run this program, you'll be asked to put a number in each basket. Just so we're on the same light beam, put 10 in basket 1, 20 in basket 2, and any numbers you want in the remaining three baskets. When your screen says, "READY," you can look at the contents of each basket by using a PRINT statement in the immediate mode, like this:

```
PRINT A(1) [RETURN]
```

See! A 10 is stored in basket A(1)! Check out basket A(2). I'll bet you find a number 20 in it!

Put Stuff into Arrays from DATA

Basket Case shows how to use an INPUT statement to put stuff into an array. Here's a short program that takes numbers stored after a DATA statement and READs them into an array.

DATA Array

```
10 DIM A(5)
15 . 1. Set up the array.
```

```
20 FOR X = 1 TO 5
30 A(X) = X
40 NEXT X
45 . 2. Read numbers into the array.
50 N = 0
60 READ A
70 IF A = - 1 THEN STOP
80 N = N + 1
90 A(N) = A
100 GOTO 60
110 DATA 10,20,30,1,2, - 1
```

This program is quite similar to Basket Case. But instead of using an INPUT statement, the program takes numbers stored after the DATA statement on line 110 and READs them into each basket.

Stay in Shape!

Junior is a junk-food junky! I've tried and tried to get him to eat the right foods, but he just doesn't listen. Finally, out of sheer exasperation, I put him on a 500-calorie-per-meal diet. To keep track of the number of calories he munches each meal, I developed a short calorie-counting program that shows off a great array application.

Here's a list of some of Junior's favorite foods, along with calories per serving for each.

Anchovy Burritos:	280 Calories each
Twinkle Cakes:	340 Calories a look
Guacamole Juice:	90 Calories per slurp
Clam Dip:	70 Calories a dip
Greaso Burgers:	470 Calories per bun
Quicko TV Dinner:	400 Calories a tray
Pizza a la Hollandaise Sauce:	900 Calories a sniff

Now you can see how serious the situation is! If he eats just one Anchovy Burrito, looks at a single Twinkle Cake, slurps some Guacamole Juice, and sniffs a Pizza a la Hollandaise Sauce, he's in big trouble!

Just to show you how effective arrays are in curbing excessive appetites, enter Calorie Counter and RUN it. It is a long listing. Take your time and type it in carefully. It'll be well worth the effort, and you'll probably burn off a few calories in the process!

Calorie Counter

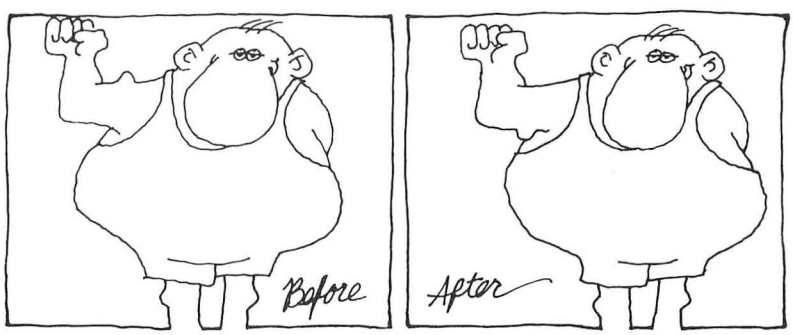
```
 10 DIM C(21):T = 0
 15 . 1) Empty the baskets.
 20 FOR X = 1 TO 21:C(X) = 0:NEXT X
 25 . 2) Clear the screen.
 30 PRINT CHR$(125)
 40 PRINT "HOW MANY JUNKY MEALS DID
    YOU DEVOUR"
 45 . 3) Input Number of meals.
 50 INPUT N
 60 PRINT :PRINT "ENTER
    CALORIES/MEAL—":PRINT
 70 FOR X = 1 TO N:PRINT "MEAL #";X;":";
 75 . 4) Put calories in each basket.
 80 INPUT M
 90 C(X) = M
 95 . 5) Keep track of total calories.
100 T = T + C(X)
110 NEXT X
120 PRINT
130 PRINT T;" calories in ";N;" meals."
135 . 6) 'A' equals average calories per meal.
140 A = T/N
150 PRINT A; " calories per meal."
160 IF A > 500 THEN PRINT "Grody! Grease me
    out!":GOTO 180
170 PRINT "Junior, you're doing great!"
```

```
180 PRINT :PRINT "Press START to eat again"
185 . 7) Fancy press START routine.
190 IF PEEK(53279)< >6 THEN GOTO 190
195 . 8) Clear keyboard and start again.
200 POKE 764,255:CLR :GOTO 10
```

Here's what my screen looked like after I entered Junior's first day (three meals) of dieting.

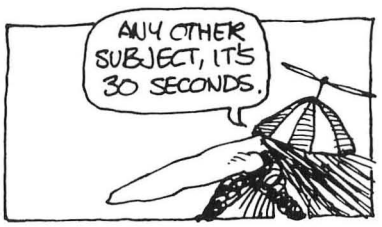
```
HOW MANY JUNKY MEALS DID YOU
DEVOUR?3

ENTER CALORIES/MEAL

MEAL#1:?620
MEAL#2:?470
MEAL#3:?490

1580 calories in 3 meals
526.666666 calories per meal.
Grody! Grease me out!

Press START to eat again
```

Junior went wild and overdid it a bit. When I wasn't looking, he must have scarfed down a bunch of anchovy burritos topped off with a generous helping of Twinkle Cakes! But things have improved. Just look at these before-and-after pictures.



The results of this calorie-counting program were made possible, in large part, by the amazing array. Although this program is long, it's really easy to understand. To help you, I put REM statements, numbered 1 through 8, in the program. Here's a broader (no pun intended) explanation of Calorie Counter.

First, empty the baskets. It's always good practice to clear out each basket before you start stuffing anything into it. Line 20 places zeros in each array location. I DIMensioned twenty-one locations to hold a maximum of three-week's gorging. That's about the limit of Junior's attention span.



Second, clear the screen. Line 30 does just that. Dr. Wacko will explain how this works later.

Third, input the number of meals Junior devoured. In line 50 the number of meals, N, is INPUTed. The program also uses this number to label each basket with the FOR/NEXT loop that begins in line 70.

Fourth, put calories in each basket. All the action takes place in line 90.

In the FOR/NEXT loop and INPUT routine beginning on line 70 and ending on line 110, you enter the calories per meal into the array.

Here's a close look at these important program lines. The FOR/NEXT loop begins on line 70, and is set to the number of meals (N) you've provided in line 50. Each meal number (X) is printed on the screen by the second statement in line 70.

The INPUT statement in line 80 lets you enter total calories for each meal. Then, in line 90, the calories are put into array C(X).

Line 100 keeps a running total of calories. Finally, in line 110, the loop goes back to line 70 to continue the process. When the program has finished looping around, it moves on to line 120. Wheew!

Fifth, keep track of the total calories. Line 100 is a simple counting routine, just like the one Snidely used to count his shekels on page 59.

Sixth, A equals the average calories per meal. In line 140, A equals the total calories divided by the total number of meals.

Seventh, try a fancy press START routine. The IF/THEN statement on line 190 waits until you press the START key. When you do, the program drops down to line 200. Don't worry about the PEEK you see in line 190. You'll learn more about PEEKs soon, when you analyze your computer.

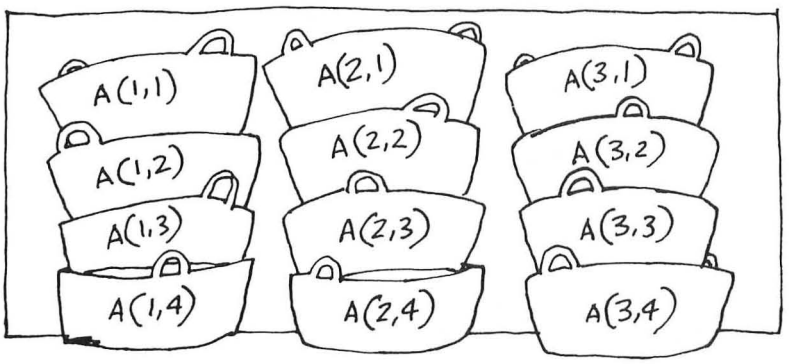
Finally, clear the keyboard and start again. In line 200, **POKE 764,255** clears out the keyboard so the last key you press doesn't interfere with the first INPUT on line 50. Delete this amazing POKE, run the program a few times, and I'm sure you'll see what I mean. The CLR in line 200 is used to clear out your computer's memory so you can start again.

Now that Junior's eating habits are under control, play with the Calorie Counter program. Modify it to your heart's content. When you really understand how it works, you'll be ready to move on to my next specialty.

The Marvelous Matrix

That calorie-counting program was great, but it can only be used to keep track of one hedonist at a time. To keep tabs on all the Wackos, I had to resort to a fancier storage system, called a *matrix.*

A matrix is just a fancy array that lets you organize stuff in columns and rows. Each basket in each column and row is individually labelled and looks just like this weird drawing.



The great thing about a matrix is that it lets you store information in a really organized fashion. For example, you could store the calories gorged by Junior in the baskets in column 1, Dr. Wacko's prodigious intake in column 2, and Ms. Peeky's petite pickings in column 3.

Setting Up A Matrix

Setting up a matrix is almost like setting up a simple array. First you DIMension the matrix like this:

```
DIM A(3,4)
```

This means that you are reserving three columns, and that each column consists of a stack of four baskets. If you want to set up a matrix with ten columns having five baskets in each column, just DIMension the matrix like this:

```
DIM A(10,5)
```

After you DIMension the matrix it's always a good idea to empty all the baskets. Here's how its done:

Empty The Baskets

```
10 DIM A(3,4)
20 FOR X = 1 TO 3
30 FOR Y = 1 TO 4
40 A(X,Y) = 0
50 NEXT Y
60 NEXT X
```

This short routine places zeros in each basket.

After you've emptied each basket, it's time to start filling each one with information. The best ways to do this are to use an INPUT statement, or to use DATA and READ statements to read numbers into each basket. You can also put stuff into a matrix the hard way by filling up each basket one by one.

This short program shows how easy it is to put data into a matrix. It sets up a matrix with two columns and two rows.

Matrix Stuffer

```
10 DIM A(2,2)
20 FOR X = 1 TO 2:FOR Y = 1 TO
   2:A(X,Y) = 0:NEXT Y:NEXT X
30 PRINT "Put a number into each basket"
40 FOR X = 1 TO 2:FOR Y = 1 TO 2
50 PRINT X;",";Y;" ";
60 INPUT NUMBER
70 A(X,Y) = NUMBER
80 NEXT Y:NEXT X
```

All the stuffin' takes place in the FOR/NEXT loop in lines 40 through 80. Here's what my screen looked like when I put numbers into each basket:

Put a number into each basket
1,1 ?20
1,2 ?40
2,1 ?60
2,2 ?80

Run the program and put numbers into each location. Then, when your screen says "READY", look into each basket in the immediate mode like this:

PRINT A(1,1) [RETURN]

If you put 20 in basket (1,1) 20 will be printed on your screen!

Now that you know how to DIMension a matrix and put numbers into it, I'll show you how to take information out of the matrix and do something useful with it.

The next program, Wacko Quotient, really puts matrices to the test. In fact, I developed it to test up to five wackos and average each of their scores.

I asked each member of this weird group to rate themselves on a scale of 1 to 10 by filling out this simple questionnaire.

SIMPLE QUESTIONNAIRE
FOR SIMPLE WACKOS

RATE YOUR WACKO QUOTIENT ON A SCALE OF 1 TO 10. ENTER YOUR RATING IN THE SPACE PROVIDED

1. OUT TO LUNCHNESS _____
2. BRAIN POWER _____
3. SLUGGISHNESS _____
4. SEX APPEAL _____
5. TAP DANCING ABILITY _____

After I collected all this meaningful information, I entered it into my Wacko Quotient program. Here are Captain Action's and Junior's results. Captain Action is "Wacko 1," Junior is "Wacko 2."

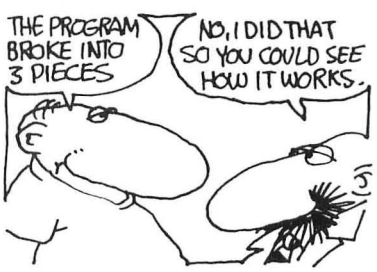
```
How many Wackos?2
Remember, there are five questions!

Wacko 1: Answer to question 1?10
Wacko 1: Answer to question 2?5
Wacko 1: Answer to question 3?8
Wacko 1: Answer to question 4?10
Wacko 1: Answer to question 5?10

Wacko 2: Answer to question 1?10
Wacko 2: Answer to question 2?1
Wacko 2: Answer to question 3?10
Wacko 2: Answer  to question 4?1
Wacko 2: Answer to question 5?1

Wacko 1's total is:43
Average wackiness is 8.6

Wacko 2's total is:23
Average wackiness is 4.6
```

Here's the great program that made it all possible!

Wacko Quotient

```
10 DIM A(5,5)
20 . Clean up your act...
30 FOR X = 1 TO 5:FOR Y = 1 TO
   5:A(X,Y) = 0:NEXT Y:NEXT X
40 PRINT CHR$(125):PRINT :PRINT "How many
   Wackos";
50 INPUT W
60 . Only accept up to 5 wackos
70 IF W > 5 OR W < 1 THEN GOTO 40
80 PRINT "Remember, there are 5 questions"
90 Q = 5
92 .
94 .
96 .
```

```
100 . Puttin' answers into the array
110 PRINT
120 FOR X = 1 TO W:FOR Y = 1 TO Q
130 PRINT "Wacko ";X;": Answer to question ";Y;
140 INPUT ANSWER
150 . Only accept answers from 1 to 10
160 IF ANSWER< 1 OR ANSWER > 10 THEN
    PRINT "Ooops!":GOTO 130
170 A(X,Y) = ANSWER
180 IF Y = Q THEN PRINT
190 NEXT Y:NEXT X:GOTO 260
192 .
194 .
196 .
200 . Takin' stuff out of the array
210 PRINT
220 FOR X = 1 TO W:FOR Y = 1 TO Q
230 PRINT "A(";X;",";Y;") = ";A(X,Y)
240 IF Y = Q THEN PRINT
250 NEXT Y:NEXT X
260 T = 0
270 FOR X = 1 TO W:FOR Y = 1 TO Q
280 T = T + A(X,Y)
290 IF Y = Q THEN PRINT "Wacko ";X;"'s total
    is:";T:A = T/Q:PRINT "Average wackiness is
    ";A:PRINT :T = 0
300 NEXT Y:NEXT X
```



The first section, lines 10 through 90, clears out the array by putting zeros into all the baskets, then lets you enter up to five wackos.

The second section, lines 100 through 190, lets you fill the array with information from the questionnaire. It also limits the range of numbers you can enter to those between 1 and 10.

The final section, lines 200 through 290, takes information out of the array, totals it, averages it, and prints it out on your screen.

As you can see, a matrix is ideal when it comes to rating a bunch of wackos, but it's also perfect for anything that can be charted in columns and rows.

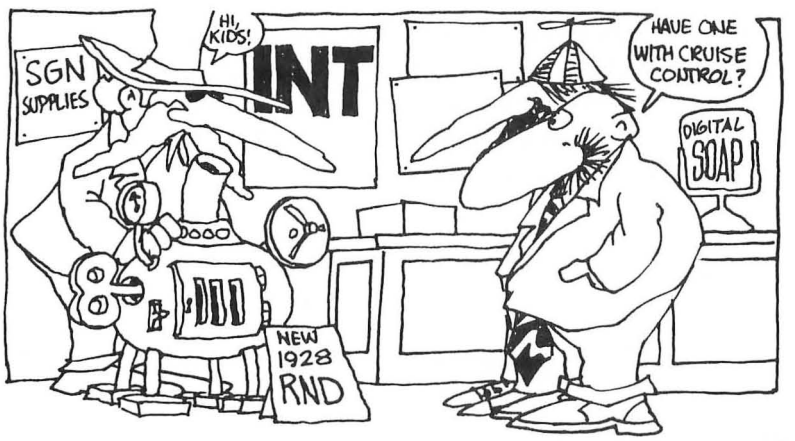
The best way to set up a matrix is to first get out the ol' pencil and paper, draw a chart containing columns and rows, then write labels across the top and down the side. Once you've finished, simply transfer your masterpiece to your matrix program.

Here's the chart I drew before I programmed Wacko Quotient:

	WACKO 1	WACKO 2	WACKO 3	WACKO 4	WACKO 5
ANSWER 1					
ANSWER 2					
ANSWER 3					
ANSWER 4					
ANSWER 5					
TOTALS					
AVERAGE					

Well, that's it for now. I've got to go and retest Junior. Any way, here comes my lovable and cute husband to invite you to a fantastic function at the oasis. Bye!

Oasis 3: Three Fantastic Functions

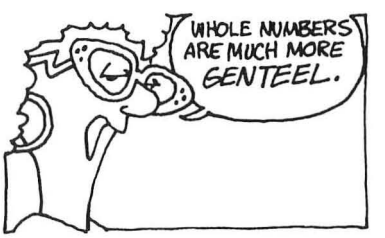


During your short stay at this oasis you'll meet three fantastic *functions*: INT, RND, and SGN, that are guaranteed to add pizazz (or pizza) to your programs.

So, join in on the fun, and I'll introduce you to the first important *function*.

The Round-Down INT

INT, which is an abbreviation for the word "integer," rounds off answers to math problems. By using INT you can round fractional answers *down* to integers or whole numbers.



The best way to see what INT does is to put it to work and watch it in action. Try these simple math examples in the immediate mode, and you'll discover how easy INT is to use.

First, here's a division example.

PRINT 55/20 [RETURN]

The answer is

2.75

Now suppose you don't want to fool around with parts of camels, but you do want a whole number as your answer. Not to worry! To round *down* the answer to the nearest whole number, just use INT.

PRINT INT(55/20) [RETURN]

The answer, amazingly enough, is 2 because of the fantastic INT function. In plain English, the statement **PRINT INT(55/20)** means divide 55 by 20, round down the answer to the nearest whole number, and print it.

Here's a multiplication example.

PRINT 2.75*18 [RETURN]

The answer is

49.5

To round this *down* to the nearest whole number use INT, like this.

PRINT INT(2.75*18) [RETURN]

Now, the answer is

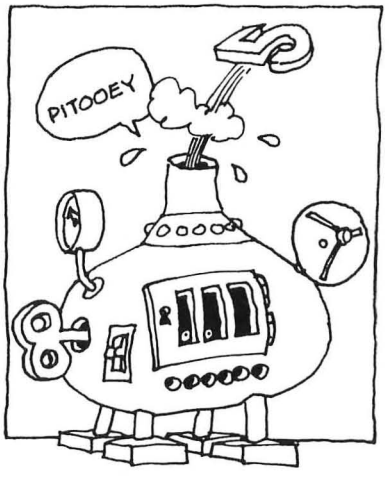
49

Amazing!

To use INT, just type in INT, a beginning parenthesis, the math, and a closing parenthesis. Then press RETURN.

Always keep in mind that INT rounds *down* the answer. So, if the math inside the parentheses equals 2.9999, INT will round the answer *down* to 2.

The Random RND



Your Atari computer has a built-in random-number generator. Allah be praised! By using RND and rubbing gently, you can make your computer spit out *random* numbers that you can use in all sorts of inventive programs. For example, you can use random numbers in games of chance whenever you want to roll dice, select cards, or simulate the spin of a roulette wheel.

Random numbers also liven up adventure games by varying the computer's response to a player's choice. Every time the player "opens" a door or decides to "walk" down a corridor, he or she is guaranteed a surprise from a new and randomly selected experience. Just like in real life.

The rollicking RND makes quiz games more rambunctious by offering your players a variety of randomly selected questions to answer.

Although programs that use the RND function are unpredictable and exciting, generating random numbers is predictable and easy.

Here's a short program that generates whole numbers between 1 and 10. Enter it, run it, and, when you've seen how it works, I'll explain all. Trust me!

Random-Number Generator

```
10 N = INT(RND(0)* 10) + 1
20 PRINT N
30 GOTO 10
```

It works pretty well, doesn't it? After you ran this program, a string of random numbers flowed down your screen. And it was all made possible by the random RND in line 10.

Here's a close-up view of the entire line so you can see how it all fits together.

```
10 N = INT(RND(0)* 10) + 1
```

I've placed an INT function before RND to generate whole numbers. Remove the INT, run the program again, and you'll get the picture. Or indigestion.

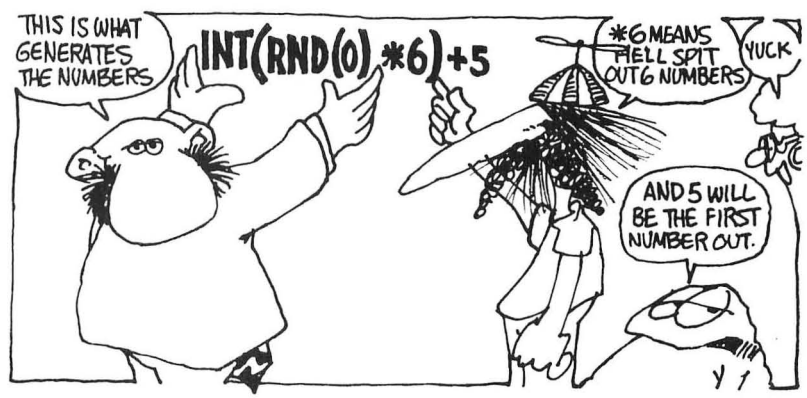
First, Start Spitting

The **+ 1** at the end of the line tells RND where to start. This program spits out numbers (just like Clyde) starting with 1.

How Many Numbers?

The ***10** tells the random generator how many numbers to spit out. Here we get ten numbers. Replace the 10 with a 5, and the program spits out five numbers between 1 and 5.

Will The Real Random Generator Please Stand up!



RND(0) is the generator. A zero in parentheses always follows RND. Here are two more short examples that show the random generator in action.

RND Example 1

```
 0 . This generator starts at 5 and spits out six
     numbers: 5,6,7,8,9, and 10.
10 N = INT(RND(0)*6) + 5
20 PRINT N
30 GOTO 10
```

RND Example 2

```
 0 . This generator starts at 1 and spits out five
     numbers: 1,2,3,4, and 5.
10 N = INT(RND(0)*5) + 1
20 PRINT N
30 GOTO 10
```

I now present The Standard Guess My Number Program. This unoriginal program is almost identical to the dull programs found in every other BASIC programming book! Almost, but not quite. This program not only illustrates a cute use of the random RND, but it has a few wacky touches of my own to spice it up. So here it is!

The Standard

```
10 . The Standard 'Guess My Number' Program
20 T = 0:N = INT(RND(0)*100) + 1
30 PRINT "I'm thinking of a number between 1
     and 100. "
40 PRINT "So, what's my number...huh";
50 INPUT GUESS
55 T = T + 1
60 IF GUESS = N THEN PRINT "Wow, gee,
     cowabonga! You got it in ";T; " tries!":PRINT
     :GOTO 20
```

```
70 IF GUESS < N THEN PRINT "Your guess is
   too low. Try again.":GOTO 50
80 IF GUESS > N THEN PRINT "Your guess is
   too high. Try again.":GOTO 50
```

When you look at it, this standard program is pretty standard. It generates a number between 1 and 100 in line 10, uses a counter (T = T + 1) to keep track of the total number of guesses, and has a bunch of IF/THEN statements that check for a correct guess. No problem!

A Random-Problem Generator

If you want to generate a lot of problems, a random-problem generator is just the ticket!

To help Junior improve his math skills, I developed this short Multiplication Practice program. It uses two generators to create baffling random problems.

Two numbers for each problem are generated in lines 30 and 40. Just change the number 10 to 1000 and really make them tough!

Multiplication Practice

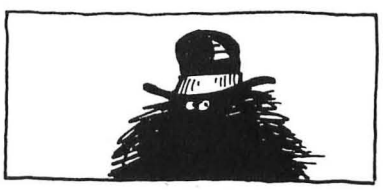
```
10 . Junior's Multiplication Practice
20 C = 0:W = 0
25 . Beware! Problems generated below:
30 A = INT(RND(0)* 10) + 1
40 B = INT(RND(0)* 10) + 1
45 .
50 PRINT "What's ";A;" times ";B;
60 INPUT ANSWER
70 IF C* 10 = 100 THEN PRINT :PRINT
   "* * * WOW! 100%. My son the
   genius!! * * *":PRINT :GOTO 20
```

```
80 IF C + W = 10 THEN PRINT :PRINT "* You got
   ";C;" right and missed ";W;":";C* 10;"%":
   PRINT :GOTO 20
90 IF ANSWER = A*B THEN PRINT "Right on
   Junior! Here's another one.":PRINT:
   C = C + 1:GOTO 30
100 IF ANSWER< >A*B THEN PRINT "Ooops!
   Maybe you read it wrong. Here it is
   again!":W = W + 1:GOTO 50
```

The principles used in this multiplication program can easily be expanded into one gigantic program that generates a random addition, subtraction, multiplication, and division quiz. No kidding!

Enter and SGN In, Mystery Guest

Presenting last, but not least, a little-known and less-understood friend of Lawrence of Newark, the mysterious SGN.



Don't panic! SGN has nothing whatsoever to do with trigonometry. It is not sine/cosine stuff.

When you use SGN, as I have below, it sets the variable *A* equal to -1, 0 or 1 to correspond with the value inside the parentheses (either negative, zero, or positive). Here's what it looks like.

```
A = SGN(N)
```

And now, presenting a very short program that demystifies SGN, but tells you nothing at all about Lawrence of Newark.

SGN In

```
10 PRINT :PRINT "Enter a number";
20 INPUT N
25 .
30 A = SGN(N)
35 .
40 IF A = - 1 THEN PRINT "It's a negative
   number!":GOTO 10
50 IF A = 1 THEN PRINT "It's a positive
   number!":GOTO 10
60 IF A = 0 THEN PRINT "It's Junior's grade-
   point average.":GOTO 10
```

Don't run away! Just run this program and enter some numbers. You might try a positive number like 100 first, then enter a negative number like – 5, and finally a 0.

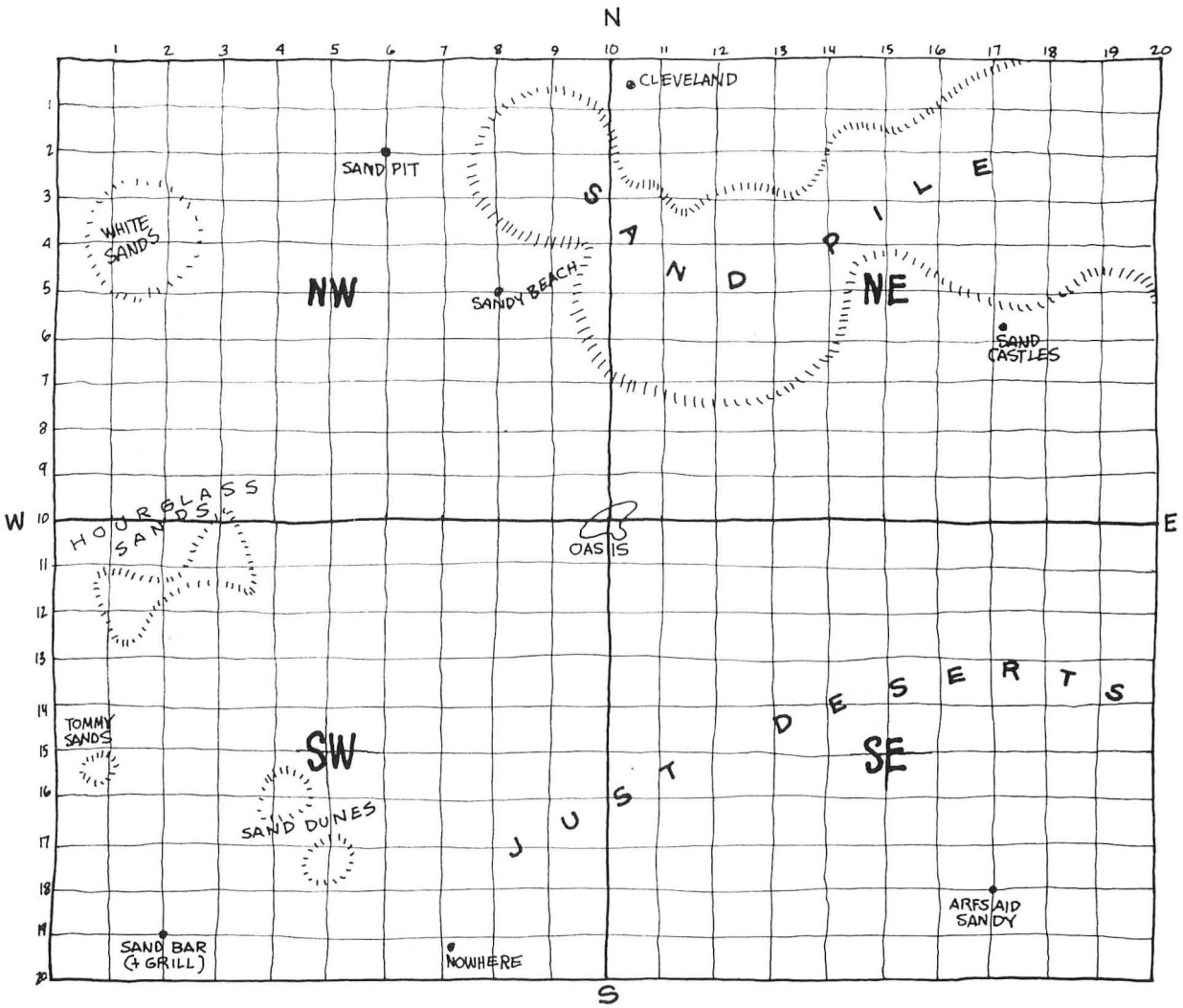
SGN has some pretty esoteric uses in BASIC programming. It can control the movement of screen images in BASIC arcade games, and lots of other stuff. (Check out my other book: "Dr. C. Wacko's Miracle Guide To Designing And Programming Your Own Atari Computer Arcade Games!") I just thought I'd show you SGN now so you can add it to your bag of programming tricks. By the way, who is Lawrence of Newark, and where is he?

Lawrence of Newark Is Lost in the Desert!

Lawrence is lost in the desert, and you can help him find his way back to the oasis! So, get set to hone up your navigation skills by playing a few games of Rescue Lawrence.

I've placed this bonus game here, at the end of this oasis, because it uses all the programming elements

you've learned so far. It really puts the RND function to good use. And besides, it's time to have some fun!



Here's a map of the desert. The oasis is in the center, and you and Lawrence are lost somewhere out in the sand dunes. You've got ten days (turns) to find the oasis. You can travel up to five miles each day in any direction(s) you choose. To help you, your compass displays the direction you are in relation to the oasis. What makes the game challenging is that you don't know how far you are from the oasis. Uh oh!

Here's the program listing. Type it in, RUN it, then try not to get lost! After you've enjoyed the game I'll review some of this program's highlights and lowlights.

Rescue Lawrence

```
 10 T = 1
 15 . Generate your initial location
 20 X = INT(RND(0)*20) + 1
 30 Y = INT(RND(0)*20) + 1
 35 .
 37 . Make sure that you don't start at the oasis
 40 IF X = 10 AND Y = 10 THEN GOTO 20
 45 .
 50 PRINT :PRINT "Day ";T;":You are ";
 60 IF T = 11 THEN PRINT "Ooops! Out of
    water!":GOTO 10
 65 . You've arrived at the oasis!
 70 IF X = 10 AND Y = 10 THEN PRINT "Welcome
    to the oasis!":GOTO 10
 75 . Print your position relative to the oasis
 80 IF Y > 10 THEN PRINT "South ";
 90 IF Y < 10 THEN PRINT "North ";
100 IF X < 10 THEN PRINT "West ";
110 IF X > 10 THEN PRINT "East ";
115 .
120 PRINT "of the oasis!"
130 IF T = 10 THEN PRINT "***You're out of
    water! Try again***":GOTO 10
140 PRINT :PRINT "How many miles North
    today";
150 INPUT N:GOSUB 300
160 IF N > 0 THEN GOTO 190
170 PRINT "How many miles South today";
180 INPUT S:GOSUB 300
190 PRINT "How many miles East today";
200 INPUT E:GOSUB 300
210 IF E > 0 THEN GOTO 240
```

```
220 PRINT "How many miles West today";
230 INPUT W:GOSUB 300
240 T = T + 1
245 . Update your position relative to the oasis
250 IF X > 10 THEN X = X—W
260 IF X < 10 THEN X = X + E
270 IF Y > 10 THEN Y = Y—N
280 IF Y< 10 THEN Y = Y + S
290 GOTO 50
295 .
297 . This subroutine checks for an input greater
    than 5
300 IF N > 5 OR S > 5 OR E > 5 OR W > 5 THEN
    PRINT "You can't travel more than 5
    miles":GOTO 140
310 RETURN
```

Here's what my screen looked like after I played a couple of rounds of Rescue Lawrence:

```
Day 1:You are South East of the oasis!

How many miles North today?5
How many miles East today?0
How many miles West today?3

Day 2:You are East of the oasis!

How many miles North today?0
How many miles South today?0
How many miles East today?0
How many miles West today?3

Day 3:You are welcome to the oasis!
```

I made it in two moves. Wheew! Here's how this fabulous game works.

The program randomly generates your initial location

in lines 20 and 30. The map is a 20-column by 20-row grid, and the random generators define your position in terms of a row and a column. A position of 3,4, for example, starts you off in the upper left, or North West quadrant of the map. (You're 3 across and 4 down, on the left side of the map.)

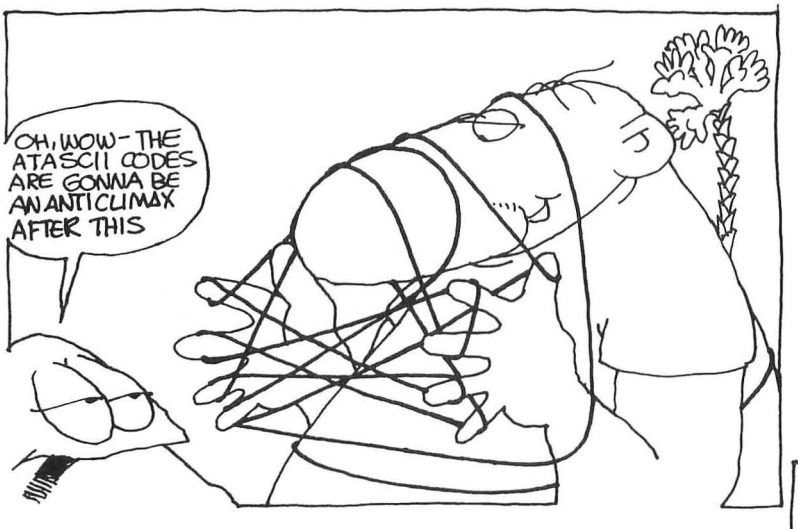
The IF/THEN statement in line 40 makes sure that you don't start off *at* the oasis (10 across and 10 down).

Lines 80 through 90 print your position relative to the oasis. Let's say you start at 3 across and 4 down, (X = 3 and Y = 4). Because Y is less then 10, the program displays "North," and because X is also less than 10, the word "West" is displayed. The combined display reads like this, **Day 1: You are North West of the oasis!**

In lines 250 through 280 the program adds or subtracts your mileage inputs to or from your current position to update your position relative to the oasis. New, updated values of X and Y are then sent back up to line 50 to determine whether you begin another turn, run out of water, or arrive at the oasis.

As you move on to the graphics and sound chapters, you'll discover all sorts of ways to spruce up this simple program. But in the meantime, call in your friends, relatives, or camel and challenge them to a game of Rescue Lawrence!

Oasis 4: Don't String Me Along Including Strings Revisited plus ATASCII Codes and the Hotski-Totski Chart



"What's so advanced about string?" That's a question I hear daily from Keys, Joystick, and Paddles, the cats we took along on our trip. To them, string is just string. But to wackos like me, advanced string-handling is one of the niftiest treats Atari BASIC has to offer.

During your visit at this oasis, I'll show you some very special string-handling tricks that you can use immediately to make your programs sizzle with excitement!

The Selective String

When we stopped at Oasis 2, I introduced you to some selective string variables. They helped Ms. Peeky choose an escort from her Little Black Book.

Remember? Well if you don't, or if you skipped ahead a bit, here they are again.

By entering a string variable and then following it with two numbers in parentheses like this, **A$(1,3)**, you can select a portion of the string.

The following short example shows how this concept works.

```
10 DIM A$(20)
20 A$ = "CAREFREE"
30 PRINT A$(1,3)
40 PRINT A$(1,4)
50 PRINT A$(5,8)
60 PRINT A$(2,2)
```

After you run this program your screen will look like this.

```
CAR
CARE
FREE
A
```

In line 30, the statement **A$(1,3)** selects the first through the third characters; line 40 prints the first through the fourth characters; line 50 prints the fifth through the eighth characters; and finally, line 60 singles out just the second character for printing.

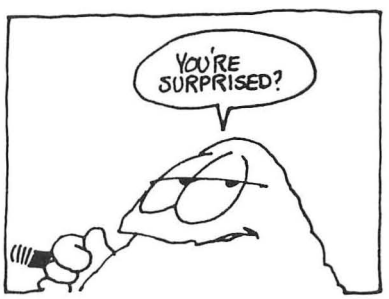
This concept is really important when you're designing programs that do lots of word manipulation. And, when you put numeric variables inside the parentheses these selective strings become super-powerful!

Presenting my super-powerful Random Nonsense Generator! It shows you how to use selective strings with numeric variables.

Random Nonsense Generator

```
10 DIM A$(60)
20 X = INT(RND(0)*50) + 1
30 Y = INT(RND(0)*50) + 1
40 IF X > Y THEN GOTO 20
50 IF Y—X > 7 THEN GOTO 20
60 A$ = "CRUNCH SNORF WOW! ZONKERS
   GLOP ZING DING BAT BLRRP"
70 PRINT A$(X,Y);
80 GOTO 20
```

Finally the truth can be revealed. I used this nonsensical program to write this book! It generates random words by continually changing the values of X and Y in the selective string on line 70. With each new set of random numbers, your screen displays a different letter or group of letters.



The IF/THEN statement in line 40 makes sure that the value of Y is always greater than the value of X. Otherwise, an error would occur since the selective string reads from left to right, and *Y always has to have a higher value than X.*

The IF/THEN statement in line 50 limits the size of each "word" to seven characters. Make this number larger if you want to show off some fancy vocabulary.

Now that you know how it's done, you can write the Great American Novel or a letter to Captain Action.

If you look closely at the RND statements in lines 20 and 30, you'll see that they spit out numbers between 1 and 50. There are 50 characters — a space counts as a character — in line 60. But, amazingly enough, I didn't have to count each character and space to find the number of characters in that line. I used the LEN statement!

The LENgth of Your String

When you want to write nonsensical programs, knowing how long your string is comes in real handy. If you know how long a string is, you can also slice it up to perform some pretty fancy string-manipulation tricks.

To learn how I learned how long my string was in the infamous Random Nonsense Generator, try this little program:

```
10 DIM A$(100)
20 A$ = "CRUNCH SNORF WOW! ZONKERS
   GLOP ZING DING BAT BLRRP"
30 PRINT LEN(A$)
```

After you run this program, the answer 50 appears on your screen. That's the length (including the spaces between the words), of string A$.

To show you how LEN works, I've modified this program to accept your wacked-out inputs.

```
10 DIM A$(100)
20 PRINT "Enter some nonsense"
30 INPUT A$
40 PRINT LEN(A$):GOTO 20
```

Now, for some wild and crazy applications.

Limit the Length of an Input

When you're designing a computerized questionnaire, it's sometimes (but not always) nice to limit the length of the data you enter. Here's an example that limits the length of a string input to 10 characters.

Limiter

```
10 DIM A$(100)
20 PRINT :PRINT "What's your favorite sport,
   sport."
30 PRINT "(10 characters or less, please.)"
40 INPUT A$
45 .
50 IF LEN(A$) > 10 THEN PRINT A$;"? That's
   ";LEN(A$);" characters!":GOTO 20
55 .
60 PRINT "What do you mean by that?":
   GOTO 20
```

The LEN in line 50, in combination with the IF/THEN statement, does all the work by checking the length of A$. Here's a sample run of Limiter so you can preview the typical results of this zany program.

```
What's your favorite sport, sport.
(10 characters or less, please.)
?CAMEL RACING
CAMEL RACING? That's 12 characters!

What's your favorite sport, sport.
(10 characters or less, please.)
?CMLRCNG
What do you mean by that?
```

Besides using it to check the length of an input, I use LEN to look for specific words within a string. This technique will come in real handy when you start working with disk-directory names. Just sweep this bit of programming knowledge to the back of your mind (or under the Persian rug) and pull it out when you start talking to your disk drive.

The next example, written by Captain Action, "looks" at the last three characters of a string. If the last three

characters are equal to **"MAN"**, **ALL RIGHT!** prints on your screen.

Be Cool

```
10 DIM A$(100)
20 INPUT A$
25 .
30 IF A$(LEN(A$)-2,LEN(A$))="MAN" THEN
   PRINT "ALL RIGHT!":GOTO 20
35 .
40 PRINT "NOT SO COOL.":GOTO 20
```

Here are two runs of Be Cool:


```
?YOU'RE A REAL COOL DUDE, MAN
ALL RIGHT!

?YOU'RE A REAL GARBANZO BEAN
NOT SO COOL.
```

Captain Action really likes to end his sentences with "man," so he went to a lot of effort to produce his Be Cool program. The statement in line 30 may look a little weird, but it's real easy to understand. Here's the important part.

```
IF A$(LEN(A$)-2,LEN(A$))="MAN" THEN
```

If this line of BASIC programming were written in English it might read, "If A$(from the length of A$ minus two characters) to the (length of A$) equals *MAN*, then...do something!"

The program begins counting back on the last character, the **N**. From there it goes back two places to arrive at the first letter of the word **MAN**. If you change the *−2* to a *−3* the program will isolate four characters; **MAN** plus the space that precedes it. If you change the *−2* to *−1* only **AN** is isolated.

The best way to understand Captain Action's masterpiece is to mess around with his progam. Just change the —*2* to another number, and MAN to a word of your own choosing, and you're on your way to becomming a LEN expert!

ESREVER NI YLERITNE NET-TIRW EREW KOOB TAERG SIHT FI TAHW

What? What if this great book were written entirely in reverse! Well, it was, until Junior got tired of standing on his head, and I got tired of talking to his feet.

And here's the short program that started all the confusion. It's Reverseroonio (It used to be Oinooreserver), and it demonstrates just how much fun you can have with a simple LEN statement and a selective string.

Reverseroonio

```
10 DIM A$(200)
20 ? "Type in a word or sentence. Please don't
   exceed three lines of text."
30 INPUT A$
40 FOR X = LEN(A$) TO 1 STEP – 1
50 PRINT A$(X,X);
60 NEXT X
```

Reverseroonio counts backwards, from the last string character to the first. All this topsy-turvy counting takes place in line 40, then the result prints on your screen in line 50. Short, sweet, and zany, isn't it?

Camel Latin

The people out here in the desert speak a strange and

mysterious dialect, called Camel Latin. It was a mystery to me, until Junior told me how it works.

They take the first letter of each word, put it at the end of the word, then add the letters *AY*. Instead of saying "Hello" they say "Ellohay" — almost like Hawaiian. In Camel Latin the word *desert* translates into *esertday*. Wheew!

I wondered how Junior was able to do all this fancy translating. Then one day I spotted a short program that he had hidden underneath a skateboard at the back of his closet.

This program, called Easy Latin, can only convert one word at a time from English to Camel Latin. So, after you enjoy it, I'll dazzle you with my own creation, Ardhay Atinlay. It can translate entire sentences. Wow!

Both of these programs show even more ways to use the LEN statement in combination with a selective string. So, avehay unfay!

Easy Latin

```
10 DIM A$(30)
20 PRINT "Enter one word then press RETURN."
30 INPUT A$
40 FOR X = 2 TO LEN(A$):PRINT A$(X,X);
50 NEXT X
60 PRINT A$(1,1);"AY"
70 PRINT :GOTO 20
```

Ere'shay owhay histay emgay orksway. In line 40, the program counts from the second letter of the word you've entered, **X = 2**, through the last letter of the word, **LEN(A$)**, and prints each letter, one at a time.

When the FOR/NEXT loop gets tired of looping, the program drops down to line 60, where it adds the first

letter, **A$(1,1)**. Then, for the final touch of nonsense, it adds the letters **AY** to the end of each word.

All this was great. But I wanted a program that could translate entire sentences to Camel Latin. So, here's Ardhay Atinlay!

One word of caution! Ardhay Atinlay has lots of problems when it comes to translating the one-letter-words *I* and *A*. That's because it works by taking the first letter of a word, placing this letter at the end of the word, and then adding the letters *AY*. When your computer tries to do this with one-letter words, its brain gets scrambled and the program comes to a screeching halt! So, be nice to this program by not entering any one-letter words. After all, your Atari is only a computer.

Ardhay Atinlay

```
10 DIM A$(128):S=2
15 ? "Type in a word or sentence. Please don't
   exceed three lines of text."
20 INPUT A$
30 FOR X=1 TO LEN(A$)
40 IF A$(X,X)=" " THEN PRINT
   A$(S,X-1);A$(S-1,S-1);"AY";" ";:S=X+2
50 IF X=LEN(A$) THEN PRINT
   A$(S,X);A$(S-1,S-1);"AY"
60 NEXT X
70 ? :? :? "THAT'S ALL FOLKS!"
```

Isn't that great! Now you can talk to camels and other assorted desert wanderers.

Ardhay Atinlay is really a lot like the Easy Latin program. The FOR/NEXT loop (lines 30 through 60) counts from the first letter of the sentence through, and including, the last letter. All the translating takes

place in line 40. Here's a close-up shot of this amazing programming line:

```
40 IF A$(X,X) = " " THEN PRINT A$(S,X - 1);
   A$(S - 1,S - 1);"AY";" ";:S = X + 2
```

Pretty long, isn't it? But don't panic; here's what it all means. "If **A$(X,X)** equals a blank space, then print the word represented by **A$** from its second character to its last character (which is a blank space) minus one. Then print the first character of this word and add the letters *AY*. Finally, move **S** (*S*tart) to the second character of the next word by making the value of **S** equal to **X + 2**. Comprende? (If not, read it again!)

The Last Word

Everything is hunky-dory until the program reaches the last word in the sentence. Because the program's always looking ahead for a blank space then retreating to rearrange the previous word, when it gets to the last word in the sentence it looks ahead, finds nothing and goes brzzzrk!

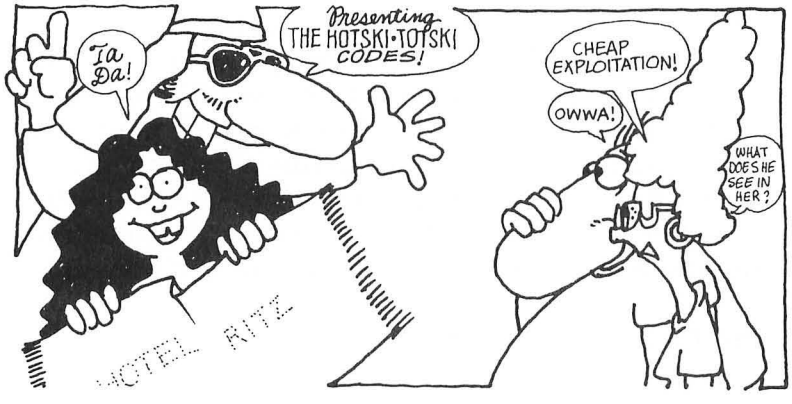
I added line 50 to remedy this little glitch. Here it is.

```
50 IF X = LEN(A$) THEN PRINT A$(S,X);
   A$(S - 1,S - 1);"AY"
```

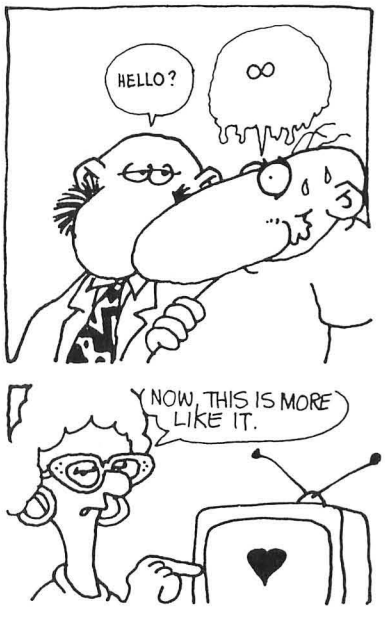
In line 50, when X equals the last letter of the sentence, the program prints the last word (minus its first letter), followed by the first letter of the last word, and finally the letters *AY*.

Understanding this program may require a little thought. The best way to really understand it is to step through each small substatement until you've got it down cold. Then, before you freeze, warm up a bit by moving on to the Hotski-Totski codes!

The Hotski-Totski Codes! Also (Sometimes) Known As The ATASCII Codes



Don't flip out! Just flip to Appendix C at the back of this book and take a look at the ATASCII Code chart. See it! Three columns filled with numbers, shapes, and typing instructions.



The first column lists numbers from 0 to 255. Next to each number is a character, and next to each character is a typing instruction.

If you want a character to appear on your screen, just follow the simple instructions next to each character's picture. For example, if you want a heart shape (number 0) to appear on your screen, hold down the CTRL key and type a comma (,).

That was easy, wasn't it? Now, on to the decimal code numbers. Most of the numbers on the chart represent letters or characters that are printed on your screen. For example, 65 represents the uppercase letter *A*, and 66 represents *B*. So far so good.

Now, here comes the sneaky part. Some of these

numbers represent control functions. A control function isn't printed on your screen. It does something, like clear the screen, ring the buzzer, or move the cursor.

Presenting CHR$

Here's a new BASIC statement called CHR$ that will help you see and hear what I'm talking about. Enter this line in the immediate mode:

```
PRINT CHR$(65) [RETURN]
```

The letter *A* appears on your screen! Replace the number inside the parentheses with 66 and the letter *B* appears. Just look at the decimal code column in the Hotski-Totski chart and you'll see how it all works!

CHR$ converts all those numbers to their associated characters, and it can also put the control characters through their paces.

Here are the "control" numbers and the functions they represent:

Control Numbers	Control Function
28	Cursor moves up
29	Cursor moves down
30	Cursor moves left
31	Cursor moves right
125	Clears the screen
126	Backspace and delete one character
127	TAB
155	The end of a line (RETURN)
156	Total line annihilation

157	Insert an entire line
158	TAB function
159	TAB function
253	Ring the buzzer (One Ringey-Dingey)
254	Delete character in front of cursor
255	Insert a character

Here are some short examples that let you see and hear how CHR$ makes my favorite control characters come to life.

```
 0 . Number 125: Clear that screen
10 PRINT "CLEAR UP THIS MESS!"
20 FOR WAIT = 1 TO 500:NEXT WAIT
30 PRINT CHR$(125);

 0 . Number 253: Listen to the buzzer
10 PRINT CHR$(253);

 0 . Number 126: Delete one Character at a time
10 PRINT "You REALLY wipe me out!";
20 FOR WAIT = 1 TO 500:NEXT WAIT
30 FOR X = 1 TO 23
40 PRINT CHR$(126);
50 FOR WAIT = 1 TO 30:NEXT WAIT
60 NEXT X

 0 . Number 156: Total line annihilation
10 PRINT "You REALLY, REALLY wipe me out!";
20 FOR WAIT = 1 TO 500:NEXT WAIT
30 PRINT CHR$(156)

 0 . Number 127: TABaroonio
10 FOR X = 1 TO 10
20 PRINT CHR$(127);
30 FOR WAIT = 1 TO 200:NEXT WAIT
40 NEXT X
```

```
  0 . Numbers 28, 29, 30, and 31: Move that
    cursor
 10 C = 0
 20 PRINT CHR$(125): . Clear the screen
 30 FOR X = 1 TO 38
 40 PRINT CHR$(28 + C);
 50 FOR WAIT = 1 TO 30:NEXT WAIT
 60 NEXT X
 70 IF C = 3 THEN PRINT CHR$(253);:END
 80 C = C + 1:GOTO 20
```

This last example not only moves the cursor, but really shows how to put CHR$ to work in your programs.

I use CHR$ a lot, and you'll really get an eyeful (of CHR$'s that is) when we design a full-fledged word processor later in our journey. By the way, if you return to the Ardhay Atinlay Program, and look at line 40, you'll see that you can replace the two quotation marks with CHR$(32), the space bar!

Finally, here's a real short routine that displays all the characters and codes on your screen.

```
  0 . Display them all!
 10 FOR X = 1 TO 255
 20 PRINT CHR$(X);
 30 NEXT X
```

Wow!

Ask the ASC

The ASC has all the answers. Just place a character or control function in quotation marks, surround it in parentheses, and the amazing ASC tells you its decimal code number. Now you won't have to look at the Hotsi-Totski chart to get a character's number, the ASC does all the looking for you!

Here's how it's used.

```
PRINT ASC("A") [RETURN]
```

When you type this in, and press the RETURN key, the number *65* appears on your screen. Put any letter, or character, inside quotes, wrap parentheses around it, and watch the results. Amazing! It's just like the Hotski-Totski chart!

"But," you may ask (I might not answer), "how do I find the decimal code number for a control function?"

ESC to the rescue!

Finding the decimal code number for a control function is a snap. All you need is the ESC key. Here's how to find the number for the function that clears the screen.

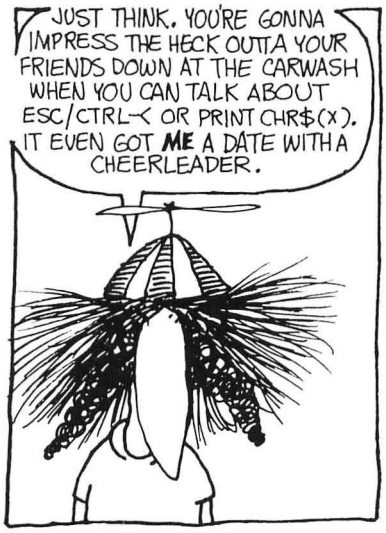
First, type in the PRINT and ASC statements, an opening parenthesis, and a quotation mark, like this:

```
PRINT ASC("
```

Next, press the ESC key. Now, press the CTRL key and the < key. A bent arrow, just like the symbol in the box next to decimal code 125 in the chart, will appear on your screen. Finish off the statement with another quotation mark and an ending parenthesis. Here's what your input should look like:

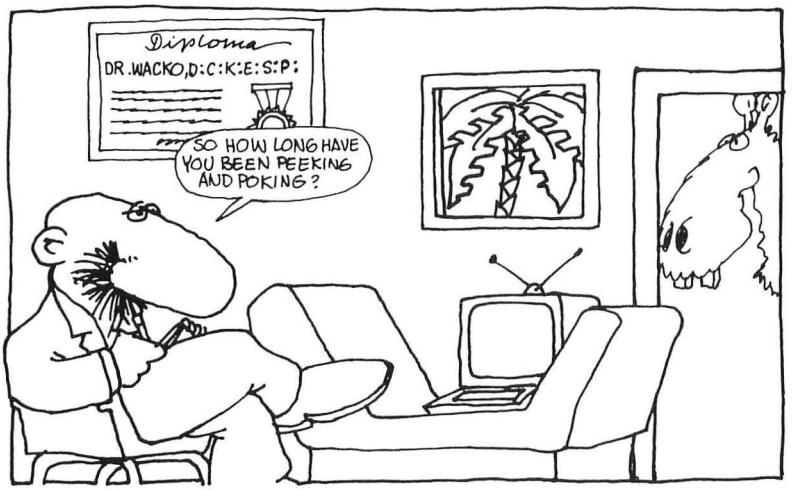
```
PRINT ASC("■")
```

The abbreviated instructions for entering control characters are listed in the Hotski-Totski chart next to each character's picture. Here are the abbreviated instructions for entering the clear-screen control function: "ESC/CTRL-< or ESC/SHIFT-< "



So, now you know most of my string-manipulation tricks. Now, when you're conjuring up a wordy program, you can use these tricks to dazzle (and confuse) your admirers!

Oasis 5: Chat with Your Fellow Travelers and Psychoanalyze Your Computer



All that programming you've learned during your journey through the desert lets you perform some pretty snazzy computer magic. But if your programs can't talk to and control the peripheral devices of your system, like the printer, screen, keyboard, disk drive, cassette recorder, or the interface, you're missing out on lots of pizazz.

This is the last oasis you'll visit before entering the dazzling worlds of graphics and sound. It might very well be the most important. In the last part of this book we'll be designing a word processor and secret text coder/decoder. To run these programs, your computer has to chit-chat with other parts of your system. So if you want to join in the coffee klatch, spend a little time at this oasis and I'll show you how your computer raps with its fellow travelers.

This oasis has another treat in store for you. After you've seen how to make your computer talk to its

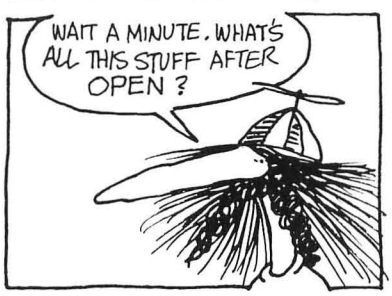
peripherals, I'll show you how to look inside your computer's brain, scramble it up, and produce some pretty weird effects. So grab a cushion, sit down in front of your Atari, munch a fig, and we'll get started.

Garbage In, Garbage Out

Your Atari talks to its *peripheral devices* (the printer, screen, keyboard, disk drive, cassette recorder, interface unit, or anything else that talks back, like Clyde) by using an abbreviated name for each device. Here's a list of their names.

Device	Abbreviated Name
Printer	P:
Screen	S:
Screen editor	E:
Keyboard	K:
Disk drive	D:
Cassette recorder	C:
RS-232 Interface	R:
Junior	CEMENT HEAD:

There are eight input and output channels, numbered 0 through 7, that your Atari sets aside for all this chit-chat. But because it reserves channels 0, 6, and 7 for its own use, you can only use channels 1 through 5 to tell it to talk to these devices.



OPEN up a channel. Before you can tell your Atari to chat with one of the devices, you've got to OPEN up a channel for communication. You do this with the OPEN statement, like this.

```
OPEN #1,4,0,"K:"
```

Captain Action's got a good point. What is all that stuff behind the OPEN?

Choose a channel. First, the **#1** means that channel 1 has been opened. You can use any number between 1 and 5.

Boss the device around. The next number is the order number. It lets you tell your computer and the device what they're supposed to be doing.

In this example, **4** means that you want to *read* the keyboard's output. Petunia's drawn this simple chart (called "Petunia's Chart" naturally) that explains all, bless her heart.

PETUNIA'S CHART

DEVICE	ORDER NUMBER	ORDERS
[P:] Printer	8	Write to the printer.
[S:] Screen	8	Clear the screen and write to it.
	12	Clear the screen, write to it, and read from it.
[E:] Screen Editor	8	Screen output.
	12	Keyboard input & screen output.
	13	Screen input & output.
[K:] Keyboard	4	Read the keyboard.
[D:] *Disk File	4	Read from a disk file.
	6	Read the disk directory.
	8	Write a new file.
	9	Write to and update an existing file.
	12	Read and write to a file and update it.

CONTINUED...

[C:]	4	Read from a cassette tape.
Cassette Tape	8	Write to a cassette tape.

* Disk file: The abbreviation for disk file, D:, is always followed by a filename and an optional extension like this: "D:FILENAME.EXT". To talk to a disk drive other than drive 1, place the drive number after the "D" like this: "D2:FILENAME.EXT". In this example your computer will be talking to disk drive 2.

Repeat Junior's grade point average. The third number after OPEN is *always* a 0! It doesn't do anything but take up space. But you've always got toput it there! Crazy, isn't it?

The decisive device. The abbreviation of the device you want your computer to talk to is placed after the last comma, and wrapped in quotation marks like this, **"P:"**

All those commas! You've got to include *all* those commas between each statement following OPEN to open a channel for communication. Wheew!

In a minute I'll show you how to use OPEN. But before I get ahead of myself and get carried away, here are some typical uses of OPEN and their translations:

OPEN #1,8,0,"P:"
Translation: Open Channel 1. Write to the printer.

OPEN #3,12,0,"S:"
Translation: Open Channel 3. Clear screen, write, and read.

OPEN #2,4,0,"K:"
Translation: Read the keyboard.

CLOSE The Channel

It's easy (but nasty) to pull the plug and disconnect your computer from one of its devices. Still, you should close each channel after it's been used. So, if you're ready to close a channel, just use the CLOSE statement like this.

```
CLOSE #1
```

This example closes channel #1. But of course, you can close any channel you want. Just replace the number after the "#" with a channel number from 1 to 5.

Now that you know how to OPEN and CLOSE channels, it's time to show you how to put all those devices to work.

GET and PUT (That Famous Vaudeville Team)

The GET and PUT statements work together like Tom and Jerry. After you OPEN a channel, you can use GET to *get* information from a device, and PUT to *put* it into your program. Or, if you want to be sneaky, your can use GET to *get* information from your program, and PUT to *put* it into, or on, a device!

The Keyboard and Your Screen

Sometimes it's nice to be able to GET information from the keyboard and display it on your screen while your program is running around doing its thing. This technique can be used as the basis of a simple word processor, where you're able to *get* keyboard information printed on your screen *while the program is running.*

To do this, you've just got to open a channel, and turn on the K: device. Here's a short keyboard program that prints stuff on your screen while the program is running around and acting silly!

```
 0 . Keyboard
10 OPEN #1,4,0,"K:"
20 GET #1,A
30 PRINT CHR$(A);
40 GOTO 20
```

The GET statement in line 20 *gets* each character's ATASCII code from the keyboard as you type it in. Each character's ATASCII code is placed in variable A. Then, in line 30, the program prints the character on your screen.

Line 40, sends the program back up to line 20 so it can GET the next character.

When you're finished playing with this program, type **CLOSE #1 [RETURN]** in the immediate mode to turn off the keyboard.

This short example is the heart of the word processor that we'll develop together later.

The keyboard program has one drawback. It doesn't clear your screen, and it doesn't let you take full advantage of Atari's built-in editing functions. Here a short

program that uses both the keyboard and screen editor. It also demonstrates how that great team, GET and PUT, work together.


```
 0 . Keyboard and Screen Editor
10  OPEN #1,4,0,"K:":OPEN #2,12,0,"E:"
20 GET #1,A
30 PUT #2,A
40 GOTO 20
```

In line 20, this program GETs each character's ATASCII code from device #1 (the keyboard) and PUTs it into device #2 (the screen editor) in line 30. **PUT**, in this program, replaces the **PRINT CHR$(A)** statement I used earlier in the keyboard program.

Line 40 sends the program back to line 20 to GET the next character from the keyboard.

When you're finished being astounded by this program, turn off channels #1 and #2 by using CLOSE statements in the immediate mode.

Your Prose in Print

The next program is really exciting. It opens channels to the keyboard, the screen editor, and the printer. It then lets you send all the great prose you've written on the screen to your printer!

Here's the program. Type it in, RUN it, and start writing.

To print the contents of your screen to your printer:

1. Press the RETURN key after the text you want printed.

2. Type in an asterisk (*), and your printer will come to life (if it's turned on).

```
 0 . Keyboard, Screen Editor and Printer
10 DIM B(500):X = 1
20 OPEN #1,4,0,"K:":OPEN #2,12,0,"E:"
30 GET #1,A
40 PUT #2,A
50 IF A = 42 THEN OPEN #3,8,0,"P:":PUT #3,
   155:FOR P = 1 TO X—1:PUT #3,B(P):NEXT
   P:CLOSE #3:X = 1:GOTO 30
60 B(X) = A:X = X + 1:GOTO 30
```

In line 30, this short program first GETs each character's ATASCII code from the keyboard. Then, in line 40, it PUTs each character on the screen. Finally, it goes down to line 60, where it places each character's ATASCII code into array B(X). Far out!

The Printing Takes Place in Line 50

When you type in the asterisk (*) you set the value of A to 42, (the ATASCII value of '*') so line 50 goes into action.

First, the program opens a channel to the printer. Then the program prints ATASCII code 155, which positions your printer so it will start printing on the left side of your paper.

The FOR/NEXT loop PUTs each character (except the asterisk, because the loop goes from **P** to **X—1**) to the printer.

When the loop runs out of steam, the program CLOSEs the channel to the printer, resets the value of X to 1, and returns back to line 30 to get the next keyboard input. Nifty!

In line 10, I only DIMensioned B to store 500 characters. Increase this number if you'd like to print out longer bits of prose.

Store Your Lore on Disk or Cassette

Now that you've printed your undying prose, you may want to store it on a diskette or cassette. There are lots of ways to store information so you can retrieve it later. I'll show you a few of my favorite tricks and introduce you to some wacked-out concepts. later you'll be able to use this knowledge to design your own secret coder/decoder and word processor. Hot stuff!

Helpful hint: Before you move on, take a minute to flip back to Appendix B to learn how to name files properly.

List The Grist with a Set of Nifty Programs

We are going to discuss two programs. The first, Store Numbers, PUTs numbers on a disk file named WACKO.LST. The second, Retrieve Numbers, GETs the numbers from the disk file WACKO.LST and prints them on your screen.

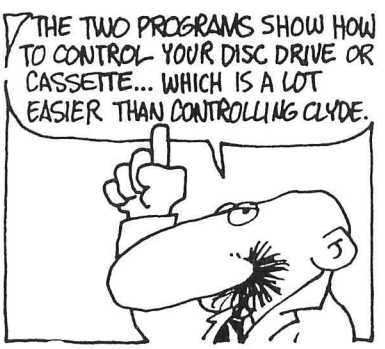
To store and retrieve information from a cassette recorder, just change the device symbol that appears after the OPEN statement in both programs to: **C:**. Don't use a filename.

Look at the two programs now. I've used the filename WACKO.LST in both programs. If you're working with a disk drive, you can use any filename you'd like, as long as its first part (before the period) contains eight or less characters, and its extension (after the period) has three or less characters. But remember, both programs complement each other and each look for the *same* file names.

Yes, these little programs are short, sweet, and jucy. Here's how to use them.

First, RUN Store Numbers. It will open a file named WACKO.LST on your disk and store the numbers 1 through 9.

When the word READY appears on your screen, type **NEW [RETURN]** to clear out memory, then enter and RUN Retrieve Numbers. The program will enter the numbers 1 through 9 into your computer and display them on your screen. Amazing!



Store Numbers

```
10 OPEN #1,8,0,"D:WACKO.LST"
20 READ A
30 TRAP 60:PUT #1,A
40 GOTO 20
50 DATA 1,2,3,4,5,6,7,8,9
60 CLOSE #1
```

Retrieve Numbers

```
10 OPEN #1,4,0,"D:WACKO.LST"
20 TRAP 50:GET #1,A
30 PRINT A;
40 GOTO 20
50 CLOSE #1
```

The Nitty Gritty: Inside Store and Retrieve

First, let's look at the Store Numbers program. The order number, 8, in the OPEN statement tells the computer and disk drive that you want to write a new file.

Line 20 READs the numbers that appear after the DATA statement in line 50. Line 30 PUTs each number into device #1 (the disk drive). Then, from line 40, the program goes back to line 20 to READ the next number.

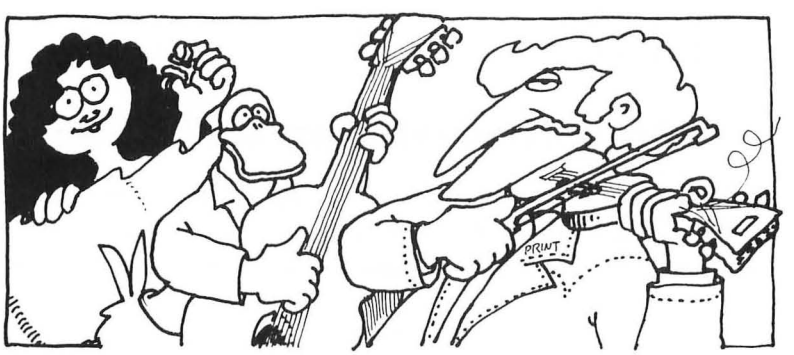
The TRAP statement sends the program down to line 60 to close channel #1 when the *pointer* runs out of data to READ.

Now, a closer examination of the Retrieve Numbers program. The order number, 4, in the OPEN statement tells the computer and disk drive that you want to read a file.

In line 20, the program GETs each number from device #1 (the disk drive). Then, in line 30, the program prints each number on the screen. Line 40 sends the program back to line 20 to GET the next number.

The TRAP statement in line 20 sends the program down to line 50 to close channel #1 when there are no more numbers on the disk file — closed for the day, just like Mort's Falafel Stand and Deli when they run out of falafels.

Lingering Strings, Featuring, Your Old Tent Mates: INPUT and PRINT



The two commands, INPUT and PRINT, come in real handy when you're storing and retrieving strings. You can use them with an OPEN statement by just adding the channel number. Here's what I mean.

```
INPUT #1,A$
```
or
```
PRINT #1,A$
```

The best way to see how these two old friends futz around is to watch them in action. So here's another set of programs, called Store String and Retrieve String, that show one way to (you guessed it) store and retrieve strings.

Make sure that your disk drive is turned on and ready to go. Now, RUN Store String first, erase your computer's memory with the NEW command, then enter and RUN Retrieve String.

Store String

```
10 DIM A$(100)
20 OPEN #1,8,0,"D:STRING.LST"
30 READ A$
40 TRAP 60:PRINT #1,A$
50 GOTO 30
60 DATA HI GANG. WACKO HERE!
70 CLOSE #1
```

Retrieve String

```
10 DIM A$(100)
20 OPEN #1,4,0,"D:STRING.LST"
30 TRAP 60:INPUT #1,A$
40 PRINT A$;
50 GOTO 30
60 CLOSE #1
```

Unravelling Strings

The only difference between the Store String and Store Numbers programs occurs in line 40 where the program READs the string from line 30 and then PRINTs it on to device #1 (the disk drive).

In the Retrieve String program the INPUT statement on line 30 accepts the input of string A$ from device #1 (the disk drive). Line 40 prints the string to your screen.

Ask a Question without a Question Mark?

You can use INPUT with an OPENed device to ask questions *without* having a question mark appear on your screen. And here's how it's done.

No Question Mark?

```
10 OPEN #1,4,0,"K:":OPEN #2,12,0,"E:"
20 PRINT "ENTER A NUMBER, PLEASE ";
30 INPUT #2,A
40 PRINT A*25
50 CLOSE #1:CLOSE #2
```

The INPUT statement in line 30 acts just like the INPUT you're familiar with. It waits for your input! But because it is waiting for an input to a specific device, no question mark appears on the screen!

You can use this trick to spruce up the look of some of your programs, or baffle your aardvarks.

Deciphering the Disk Directory

Before moving on to one of my favorite topics, dissecting your computer's mind, here's a last example of divisive device communications — reading the disk directory. The *disk directory* contains the names of all the programs stored on the disk, shows how much space — measured in sectors — each program occupies and how much space remains on the disk.

Directory

```
 10 DIM NA$(100)
 20 GRAPHICS 0:POKE 752,1:POSITION 3,1:?
    "Press START to print disk directory":?
 30 IF PEEK(53279)< >6 THEN GOTO 30
 32 .
 35 . Here's where the action is:
 37 .
 40 POKE 752,1:CLOSE #2:OPEN
    #2,6,0,"D:*.*"
 50 TRAP 80:INPUT #2,NA$
 60 PRINT NA$
 70 GOTO 50
 75 .
 80 CLOSE #2:POSITION 6,22:? "Press START
    To Continue"
 90 IF PEEK(53279)< >6 THEN GOTO 90
100 ? CHR$(125):POKE 752,0:GOTO 20
```

All the action takes place in lines 40 through 70. In line 40 the order number, 6, tells the computer to read the disk directory. Then line 50 accepts the listing of names from the disk as string input NA$.

Line 60 prints the file names on your screen, and then in line 70 the program goes back to line 50 for more. After

the program has received all the directory file names, and the pointer sends a message that the file has ended, the TRAP on line 50 sends the program to line 80 to CLOSE the channel. Wheew! Now we can proceed to psychoanalyze your computer.

Psychoanalyze Your Computer

Throughout the course of this book you may have noticed a couple of weird-looking statements hidden in the recesses of some of the programming examples. They looked like this.

POKE 752,1
or
PEEK(53279)

"What," you may have asked yourself, "is Dr. Wacko doing?"

Elementary, my dear desert wanderer. I was peeking inside the computer with the PEEK statement, and poking around with the POKE statement.

The PEEK statement lets you look inside your computer's brain. When you find out what's in it, you can use the POKE statement to scramble it up!

There are lots of hidden recesses inside your computer's memory, called *locations* or *addresses,* that do special things, or hold certain types of information. Sneaky Peek's listed a bunch of his favorite locations back in Appendix D, but they won't do you much good unless you know how to get in there and really mess them up.

So get out your scalpel and we'll cut through all the rhetoric and peek inside your computer's brain.

Here's how to use a PEEK statement to see what's going on inside a location.

PRINT PEEK(53279) [RETURN]

When you execute this short routine in the immediate mode, the number 7 appears on your screen. That's what's inside location 53279. Let's try another one:

PRINT PEEK(82) [RETURN]

This time *2* appears on your screen. You guessed it. The number 2 is stored in location 82.

It's easy! The number inside the parentheses, after the PEEK statement, is the location you're peeking into.

Now, lets POKE around inside location 82, and put our own number inside:

POKE 82,10 [RETURN]

Uh oh, Something happened! The cursor moved ten spaces to the right, and the entire left margin is out of whack! To set things straight again, just

POKE 82,2 [RETURN]

Wheew, back to normal! That was a close call. But it did demonstrate the power of the POKE.

You can POKE a location with any whole number between 0 and 255 by placing it after the comma, behind the location number.

Now that you know how to PEEK inside your computer's brain and POKE around, I'll show you how to use some of my favorites.

If you don't want the cursor to appear on your screen, you can get rid of it by POKEing 752 with 1. Try this.

POKE 752,1 [RETURN]

To get the cursor to return, **POKE 752,0**. Simple, isn't it?

Controlling the Control Keys

Now for another of my favorites. Manipulating the OPTION, SELECT, and START keys.

Wouldn't if be loverly if your programs let the user press one of the special function keys to make something happen? For example, you might want the user to press the START key, to start the program, or press the SELECT key to select a level of difficulty. No problem. **PEEK(53279)** to the rescue!

Here's a handy chart that shows what number is placed in location 53279 after you press each of these keys.

PEEK(53279)

Value	Key(s) Pressed
0	OPTION, SELECT and START
1	OPTION and SELECT
2	OPTION and START
3	OPTION
4	SELECT and START
5	SELECT
6	START

But don't take my word for it! Just enter and run the following program, press the special function keys, and watch the results.

```
10 PRINT PEEK(53279):GOTO 10
```

You can use a short program like this to see what goes on inside locations that relate to the keyboard. Just for fun, replace the number in parentheses after **PEEK** with **764**, run the program again, and start banging on the keyboard. You'll see lots of different numbers on your screen!

Now that you've had some fun, here's how to put PEEK(53279) to work in your programs.

```
10 POKE 752,1
20 PRINT "Press START to begin"
30 IF PEEK(53279)< >6 THEN GOTO 30
40 PRINT "You did it this time, buster!"
```

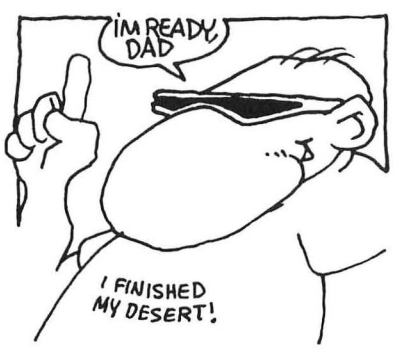
Wow! Did you see that? The program first turns off the cursor, then waits in line 30 until you press the START key! Change line 30 to read:

```
30 IF PEEK(53279)< >5 THEN GOTO 30
```

Now the program won't move on until you press the SELECT key. Amazing.

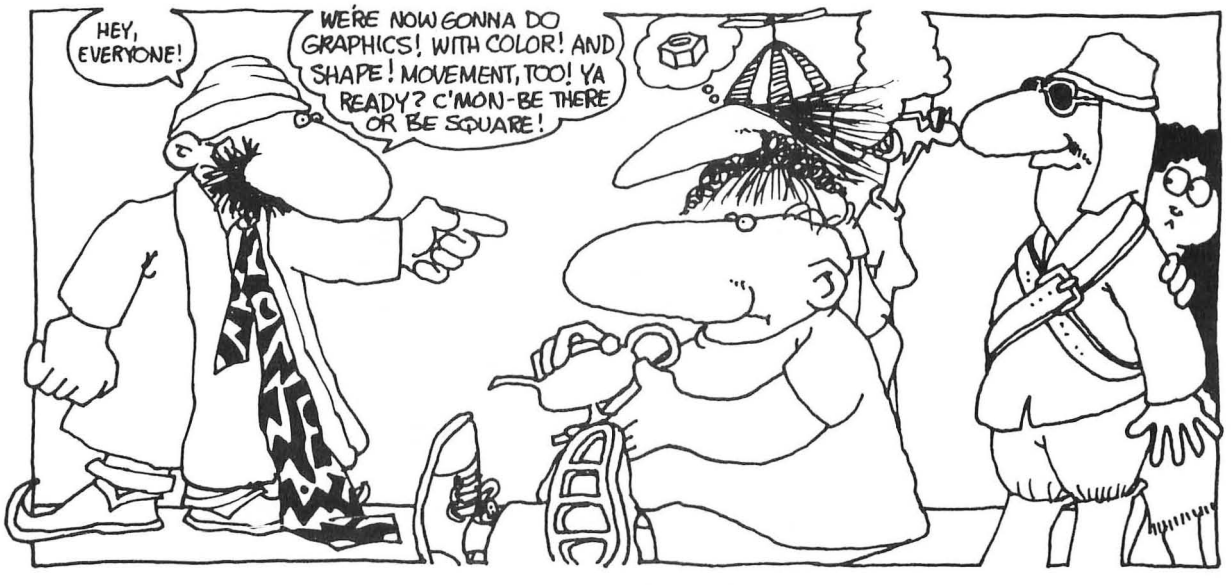
There are lots more handy PEEKs and POKEs waiting for you back in Appendix D: Smokey Peek's Pokes & Peeks. Wander back there now and have some fun.

You Made It!



Congratulations! You've made it through the desert, and now it's time for total pixel control! So flip the page, leave your sunglasses on, and we'll explore the dazzling world of Atari graphics!

4
Graphics Power: An End to The Desert Blues



Up to this point you've been sitting in front of a blue screen filled with white printing, and I'll bet that you're about to hang it up and return to reality. Things can get awfully dull without color or interesting shapes.

I know. Three months of traveling around in the desert was enough to dull my senses and make me almost color-blind! White, white, WHITE! I couldn't stand it any more, and neither could my charming wife, Petunia; Snidely Seersucker; Ms. Peeky; or Captain Action. (Clyde and my brilliant son, Junior, loved it!) I almost had a mutiny on my hands!

Your Atari computer has the graphics power to dazzle and mesmerize you with vibrant colors, weird shapes, and strange movable objects. So, don't mutiny! I'm about to show you how easy it is to take charge of Atari's computing power and end those desert blues.

Get set for a real treat. In this amazing chapter you'll learn how to doodle in the sand, create your own colorful Camel Racetrack, and recreate my infamous Model of the Universe, plus lots of other fascinating stuff. Wow!

But seeing is believing. Leave your sunglasses on, turn on your creative self, and we'll get started.

I'd like you to enter, run, and have some fun with the short program below so you can get a taste of what's in store for you as you wander through this chapter. The program is called Doodle. It lets you create all sorts of colorful shapes on your screen. Many of the commands and statements in Doodle will be new to you. Many of them will be old hat. Don't worry! When you've finished this chapter, Doodle will seem as easy as riding a camel or dancing at the oasis.

Doodle

```
 10 GRAPHICS 3:T = 1
 20 C = INT(RND(0)*3) + 1
 30 GOSUB 90
 40 COLOR C
 50 PLOT X,Y
 60 GOSUB 90
 70 DRAWTO X,Y
 80 GOTO 20
 90 PRINT T;") ENTER A NUMBER FROM 0 TO
    39";:INPUT X
100 PRINT " ENTER A NUMBER FROM 0 TO
    19";:INPUT Y
110 PRINT CHR$(125):T = T + 1
120 RETURN
```

I'll bet you noticed a few oddball words as you typed in Doodle — words like GRAPHICS, COLOR, PLOT, and DRAWTO. As I said before, don't sweat it! (Not easy out

here in the desert.) You'll learn *all* in a little while. Now it's time to run the program and have some fun.

The first thing you'll see after you run Doodle is

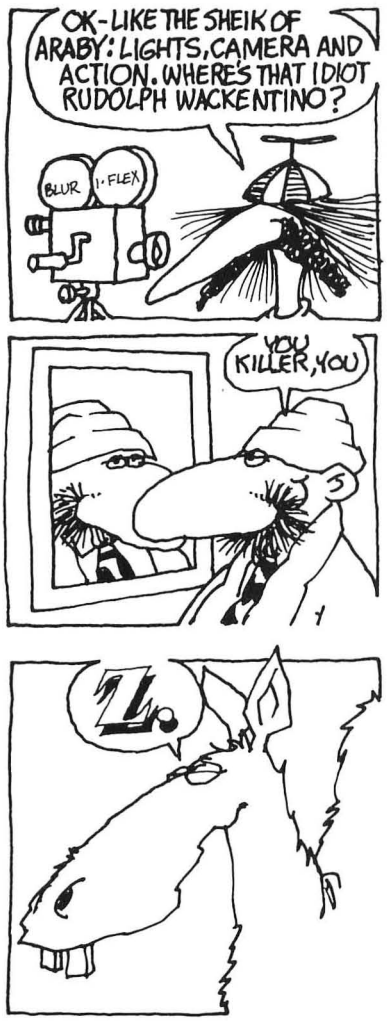
```
10
20 ENTER A NUMBER FROM 0 TO 39?
30
```

Enter a number, press RETURN, and you'll be asked for another number between 0 and 19. Just follow the instructions, keep on entering numbers, and a multicolored drawing will appear on your screen.

To help you get started, here are the numbers I entered to draw a pyramid.

Entry	My Inputs
1)	18 5
2)	18 15
3)	18 15
4)	28 15
6)	28 15
7)	18 5

Now that you've expressed your creative drawing talent, and sampled a bit of your Atari's colorful capability, it's time to push on to the nitty, gritty, and sometimes glamorous world of Atari graphics.



The Sheik of Araby — Lights, Camera, Action. Rudolph Wackentino?

I know I look handsome, sheik (sic), and suave in this getup. But I'll bet you're wondering why I'm dressed up this way? Well, to be perfectly honest, I did it to attact your attention! Ever since boot camp, Petunia and the other recruits refused to talk to me, and I had to find some way to get their attention.

Now that I've gotten your attention — well almost everyone's attention — it's time to tell you the secrets of my success on the silver screen.

Your Atari's Screen

I perform some of my most dazzling feats right here on your Atari's screen, not at the local casbah. And so can you! But before you can astound the geni with your genius you'll have to select the proper scenery, then do some set design and construction.

I always like to ham it up on the Atari. This great computer offers so many different stage settings (called graphics modes) that I can always find the right one for *any* performance I've conceived — the weirder, the better! Each graphics mode lets me express my brilliant talent in a different way. I use some graphics modes to display the text titles at the beginning of the show, and others to wow the audience with my artistry.

Setting The Stage: Atari's Graphics Modes

Constructing a stage set and backdrop for your desert fantasy takes a lot of backbreaking work, so to make things easier, you first need to get acquainted with the Atari computer's world-famous graphics modes.

The first pearl of knowledge I'm going to impart is this: To select a particular graphics mode, use the BASIC command **GRAPHICS** (or its abbreviated form **GR.**). For example, this short, direct command selects and displays graphics mode 1:

GRAPHICS 1 [RETURN]

Give it a try!

Since a picture is worth a caravan of words, check out your screen after you type in and run the following program. It shows you all the "stage settings" your Atari computer offers. Your Atari computer may have as many as *twelve* graphics modes. To keep things simple, let's only discuss and use graphics modes 0 through 8. Not that I'm lazy, Allah forbid! It's just that graphic modes 9 through 11 use lots of precious memory, memory that can be used to make camels and other assorted desert beasties race around the Camel Racetrack that you'll be designing soon.

Dr. Wacko's World-Renowned Selecting-the-Stage Program

```
10 FOR X = 0 TO 8
20 IF X = 0 THEN GRAPHICS X:POKE
   752,1:POSITION 13,11:PRINT
   "GRAPHICS ";X:X = 1
30 FOR A = 1 TO 100:NEXT A
40 IF PEEK(53279)< >6 THEN GOTO 40
50 GRAPHICS X:POKE 752,1:PRINT :PRINT
   CHR$(127);CHR$(127);"GRAPHICS "
   ;X:POSITION 5,5:PRINT #6;"GRAPHICS "
   ;X
60 FOR A = 1 TO 100:NEXT A
70 IF PEEK(53279)< >6 THEN GOTO 70
80 NEXT X:GOTO 10
```

After your run this program, just press START to view each of the graphics modes.

Vive La Difference!

As you flip through each of the nine graphics modes presented in this program you'll notice that the screens differ in several, important ways.

Text Modes. The first three screens you'll view (graphics modes 0,1, and 2) are known out here in the desert as the text modes, and are normally used (you guessed it again!) to display text.

Pixel Modes. The remaining six screens (graphics modes 3 through 8) display colored squares in place of text. Each colored square is called a "pixel."

The Thin Blue Strip

A thin blue strip runs mirage-like across the bottom of graphics modes 1 through 8. (In graphics mode 8 the blue strip is hard to see because I'm nearsighted, and the entire screen is the same blue color as the strip. But it's there. Trust me.)

This blue strip is called a "text window." It's used to display standard letters, numbers, and symbols.

You can use the text window to display information and messages, like I've done in the next strange example. Enter and run the following weird program:

```
10 GRAPHICS 2
20 PRINT "Earth calling Dr. Wacko..."
```

To get rid of the text window, add the integer 16 to the graphics designation number, and you'll create a full screen display.

To change graphics mode 2 from a split-screen to a full-screen display, use the command **GRAPHICS 18** (2 + 16 = 18). Try this two-line program:

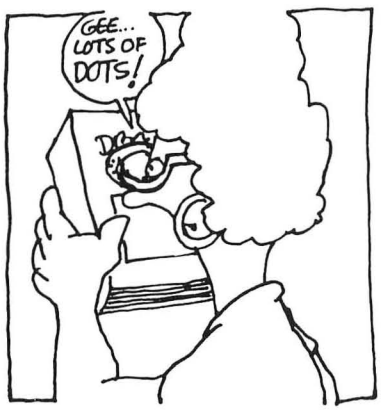
```
10 GR. 18
20 GOTO 20
```

You've eliminated that pesky blue strip, and the screen's grown! Do you remember the Selecting-the-Stage program? (How soon we forget!) You can cleverly modify the program by deleting line 50 and replacing it with:

```
50 GRAPHICS X + 16:POKE 712,INT
   (RND(0)* 100) + 10
```

Now you can press START and flip through each graphics mode without having to watch that obnoxious thin blue strip run across the bottom of your screen. I've identified each full-screen graphics mode by taking its fingerprints and assigning it a different random color to help you spot each screen as it flashes by.

The Wacko Unified Hole Theory — Columns, Rows, and Coordinates

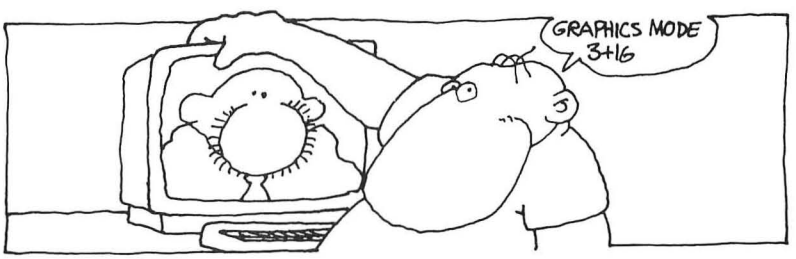


Take a magnifying glass (or squint a lot) and look at the color cover of this book.

Right, Ms. Peeky. And graphics are displayed on your computer's screen in the exactly the same way: each graphics mode contains hundreds of "holes" (pixels) waiting to be filled in with color. Just like the well at the oasis waiting to be filled with water.

Resolution

Graphics mode 0 is the screen you've used when you've typed in all the programs I've introduced to you. The screen size for graphics mode 0 is 40 columns across by 24 rows down (40X24). This means that it can accept up to 40 characters on each line and display up to 20 lines filled with characters.



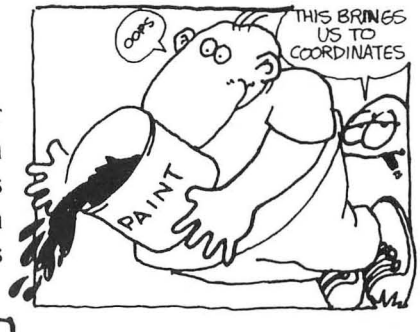
Interestingly enough, the screen size for graphics mode 3 + 16 is also 40X24, but it dosn't display characters, it's used to draw on! Even though one mode displays text and the other displays pixels, *both of these*

graphics screens have the same resolution (number of holes per screen).

Tucked away in appendix E is a handy chart that will help you learn about the different sizes and shapes of *all* the graphics modes.

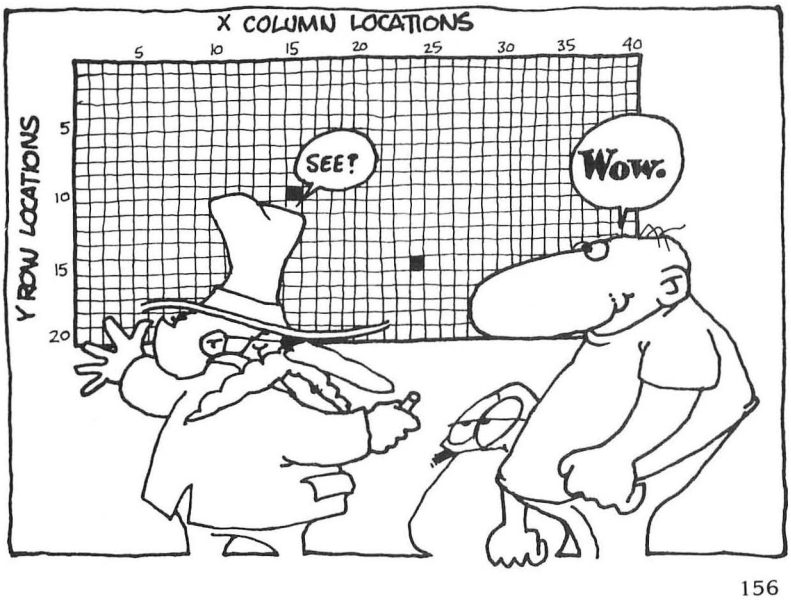
If you check out the chart you'll see that *low-resolution* graphics modes 0 and 3 + 16 each have a total of 960 (40X24) empty holes waiting to be filled. Not too many holes, that's why they're called "low-resolution modes."

On the other hand, *high-resolution* mode 8 has 69120 (360X192) teeny-weeny holes waiting to be filled in with color. Wow!



Coordinates

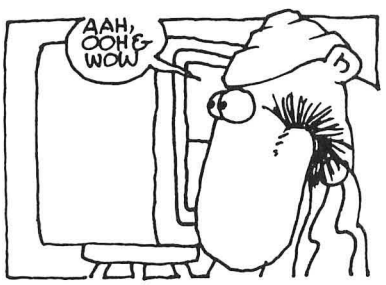
I'm glad you're here, Junior. It's simple! Each hole (or pixel) is assigned its own two-number location called a *coordinate*. This nifty system works just the same as the map you used to rescue Lawrence of Newark on page 110. Here's how the coordinate system applies to Atari graphics:



The numbers across the top of the illustration assign a value to each column; these are called "X" locations. The numbers down the illustration's side assign a value to each row; these are called "Y" locations. *Each pixel is identified by its two-number location coordinate.* The pixel's column location is *always* stated first, followed by a comma, then the pixel's row location, like this: X,Y.

Snidely's filled in the holes at location 15,10 (X,Y) and at location 24,15 (X,Y) to illustrate this simple concept.

Loads of Modes



Selecting the right graphics mode requires some artistic pizzazz (or lots of pizza!). I hold my thumb up in front of the screen, like Van Gogh appraising his canvas, and make comments — "Ahh, ooh, aargh, grunt snuffle, and wow, what a nice thumb!" — until I've choosen the right graphics mode.

This method works great for artistic types like me. But will it work for you? Even if you are an accomplished computer artist with a great thumb, you will enjoy the following fascinating section. It's loaded with some real "artistic" tricks of the trade, and gives you the complete rundown on each graphics mode, plus some very exciting graphics tricks.

Pixel Modes 3 through 8 — Using the COLOR and PLOT Statements

If you wander back to Appendix E, and peek at the all-inclusive graphics chart, you'll see that the pixel modes are separated into three groups.

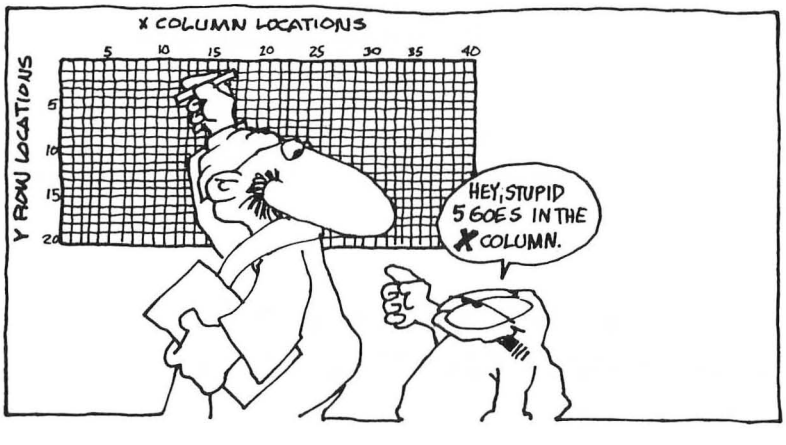
- Four-color graphics
- Two-color graphics
- High-resolution graphics

Using the COLOR statement to add color to your display in these modes is real easy. I'll use it in graphics mode 3 to give you the idea.

Graphics mode 3 or its full screen version, 19 (3 + 16), is a four-color graphics mode. This simply means that you have one background color and three pixel colors to work with. The COLOR and PLOT statements go together like Clyde and sand — they're inseparable. In graphics mode 3, you use COLOR 0 to "paint" the background of your picture and COLOR's 1,2, and 3 to "draw" on your screen in vibrant hues!

The PLOT command fills a pixel on your screen with the color of your choice, at the coordinates of your choice. COLOR and PLOT are used together like this:

COLOR 1:PLOT 5,10



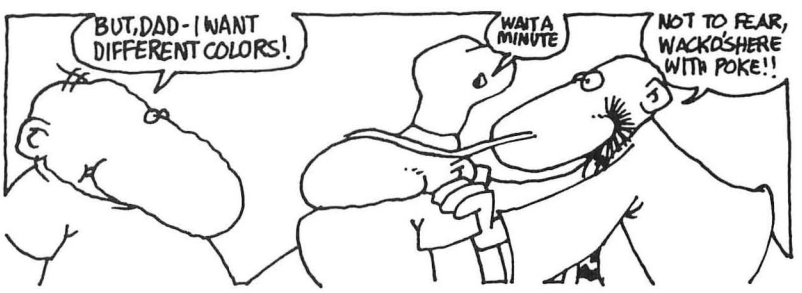
Here's a short program that PLOTs an orange pixel (COLOR 1) on your screen in graphics mode 3.

Color and Plot

```
10 GRAPHICS 3
20 COLOR 1
30 PLOT 0,0
```

Replace line 20 first with COLOR 2, then with COLOR 3. COLOR 2 appears as light green and COLOR 3 as a small blue box. Amazing! The four colors are preset:

COLOR 0—Black (background)
COLOR 1—Orange
COLOR 2—Light Green
COLOR 3—Blue



Have no fear, Dr. Wacko's here with our old friend POKE to rescue you from futzing around with the COLOR statement.

Remember, a POKE statement's got two numbers. Study the illustration to the left.

You use the POKE statement to stuff information directly into a special location inside your Atari computer's memory. Certain locations, when they hold information about colors, are sometimes called *registers*. The first number in the POKE statement tells the computer what register you want to stuff with information. The second number is the stuff that you're POKEing. Sound familiar?

Take a look at this chart. It shows the POKEs used to control each of the colors available:

	POKE REGISTER NUMBER	USE COLOR
GRAPHICS 3,5,7 (FOUR COLOR)	708	1
	709	2
	710	3
	712	4
GRAPHICS 4,6 (TWO COLOR)	708	1
	712	0
GRAPHICS 8 (1 COLOR, TWO LUMINANCES	709	1
	712	-
	710	0

By adding a number between 0 and 255 you can change the color of each PLOTed pixel. The added POKE color number changes both the pixel's *color* and its *brightness*. To get different shades of color, you simply add or subtract 2 from the color number.

Now, while you are still dazzled by all those vibrant colors let's do some more programming. Clear your screen, type NEW, then retype the preceding Color and Plot program and change line 10 to read:

```
10 GR. 3:POKE 708,99
```

Now, when you run your new Color and Plot program you'll see that the color of the little dot has changed to *pugnacious purple*!

Go wacko with this short program. By POKEing 708 with any number between 0 and 255 you can change the color of the pixel that you've PLOTed using COLOR 1. Experiment with other COLOR statements and registers. Replace the COLOR statement in line 20. Make it COLOR 2, and change line 10 to read:

```
10 GR. 3:POKE 709,195
```

Play with these concepts until you've added color to your deserted world, and have a good grasp of how the different color POKEs control each PLOTed color.

DRAWTO the Casbah

Now that you're all wacked-out on color, I'll show you how to use DRAWTO to draw lines on the screen. But

before you use DRAWTO you'll have to PLOT a beginning coordinate. DRAWTO is always used with PLOT like this:

```
PLOT X1,Y1:DRAWTO X2,Y2
```

Put the Color and Plot program back together again and add this line:

```
40 DRAWTO 39,19
```

Again, fool around with this new program, then we'll mess it up some more.

OK? Now, using the same program, add this line:

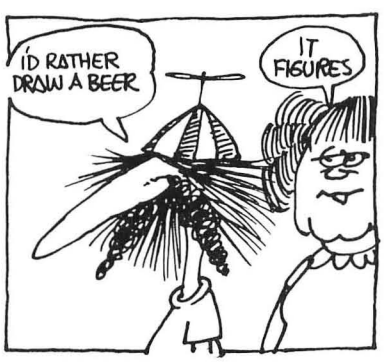
```
15 POKE 708,99
```

And change line 40 to read:

```
40 DRAWTO 0,19:DRAWTO 39,19:DRAWTO
   39,0:DRAWTO 1,0
```

Now, when you run your modified program, a strip of purple borders the edge of the screen. It's *pugnacious purple* again because we've POKEd 708, COLOR 1's register, with 99. Remember?

Line 40 shows that you can "chain" DRAWTO commands one after another. You can draw anything imaginable, even a deposed despot, by combining these PLOT, DRAWTO, and COLOR techniques!



Camel Racetrack

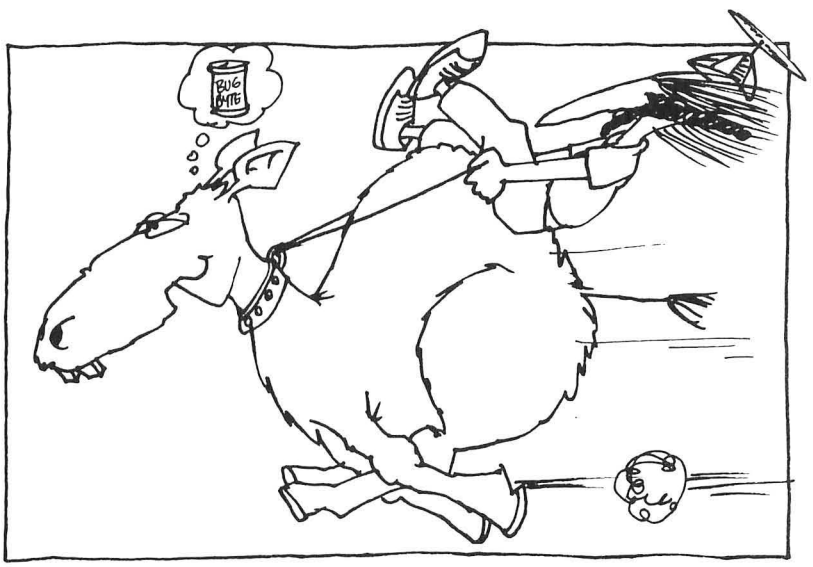
Ready to put your graphics knowledge to use? If you are, here's one of Clyde's favorite programs: Camel Racetrack. Enter it, run it, then we'll take a tour of the circuit.

Camel Racetrack

```
 10 GRAPHICS 19
 20 COLOR 1
 30 FOR A=0 TO 4
 40 PLOT A*2,A*2
 50 DRAWTO 39—A*2,A*2
 60 DRAWTO 39—A*2,23—A*2
 70 DRAWTO A*2,23—A*2
 80 DRAWTO A*2,A*2
 90 NEXT A
100 COLOR 0
110 PLOT 19,2:DRAWTO 19,21
120 PLOT 20,2:DRAWTO 20,21
130 PLOT 2,11:DRAWTO 37,11
140 PLOT 2,12:DRAWTO 37,12
150 COLOR 2
160 PLOT 10,10
170 DRAWTO 29,10
180 DRAWTO 29,13
190 DRAWTO 10,13
200 DRAWTO 10,10
210 COLOR 3
220 PLOT 11,11
230 DRAWTO 28,11
240 PLOT 11,12
250 DRAWTO 28,12
1000 GOTO 1000
```

Racing through Camel Racetrack

Now, armed with your knowledge of graphics modes and commands, put on your racing silks and we'll trot through Camel Racetrack.



In line 10, I've selected "windowless" graphics mode 19 (3 + 16).

To speed up the drawing process I often use the handy FOR/NEXT loop. Lines 20 through 90 of Camel Racetrack quickly draw five COLOR 1 lines of decreasing lengths from the top to the bottom of the screen.

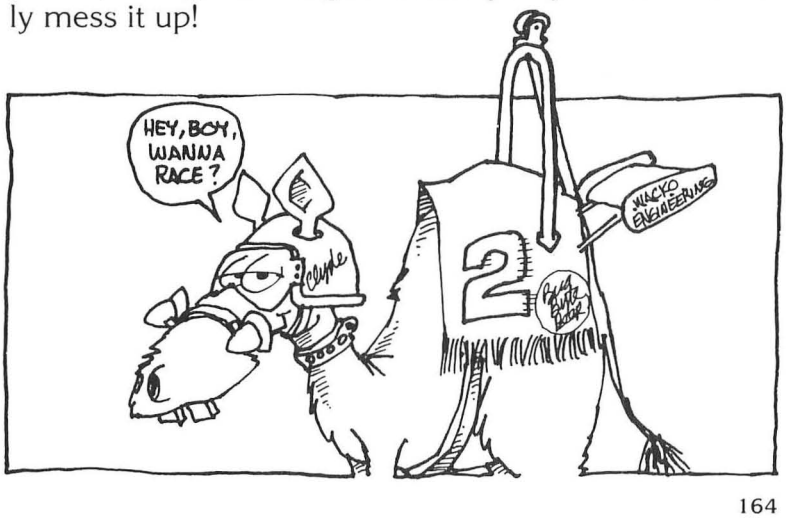
Here are lines 10 through 90 of the Racetrack program. I've slowed down the drawing process by adding a pause FOR/NEXT loop in line 85 so you can watch the screen being drawn. Run this short program, watch it race around in circles, and you'll get the picture — or a headache!

Camel Racetrack FOR/NEXT loop Demo

```
 10 GRAPHICS 19
 20 COLOR 1
 30 FOR A=0 TO 4
 40 PLOT A*2,A*2
 50 DRAWTO 39—A*2,A*2
 60 DRAWTO 39—A*2,23—A*2
 70 DRAWTO A*2,23—A*2
 80 DRAWTO A*2,A*2
 85 FOR PAUSE=0 TO 200:NEXT PAUSE
 90 NEXT A
100 GOTO 100
```

The balance of the program, lines 100 through 250, is broken into sections headed by a COLOR statement and followed by a bunch of PLOT and DRAWTO statements. COLOR 0, the background color, is used to "cut away" sections of the lines that were first drawn. COLORs 2 and 3 are used to draw the design in the center of the screen.

Now that you know what you are doing (I wish I could say the same), use your new skills to modify your racetrack to your heart's content. Change its colors; put it in a different graphics mode; put it on a different planet; change its design. Go completly wacko — really mess it up!



Sifting Through the Text Modes

Graphics Mode 0

Graphics mode 0 is usually used to display text, but stick with Dr. Wacko, kid, and I'll show you how to go bzzrrk in this mode (if you'll be a little patient).

Just so you'll recognize it when you see it, graphics mode 0 is that blue display with a black background that you see when you first turn on your computer.

Here's some basic information about graphics 0:

Graphics command: **GRAPHICS 0** or its abbreviated form **GR.0** turns on graphics mode 0.

Screen size: Graphics mode 0 has a screen size of 40 characters across by 24 characters down, and is usually used to display text.

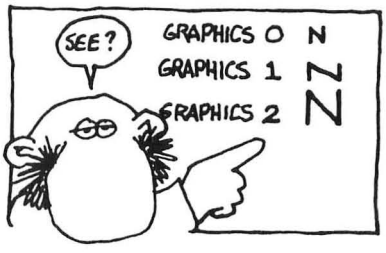
Luminance: It can display one color having two degrees of brightness. To change the text character's brightness just POKE 709 with a number between 0 and 255, like this: **POKE 709,30**.

Screen and border color: If you want to change the screen's color, POKE 710 with a number between 0 and 255, and if you want to change the color of the screen's border, POKE 712 with a number.

Graphics Modes 1 and 2: El Biggo Texteroonio

Graphics modes 1 and 2 are also used to display text. Trip between graphics modes 0,1, and 2 in my Selecting-the-Stage program, and once you get back up you'll see that letters and symbols printed in graphics mode 1 are twice the width of those printed in graphics 0, but are the same height.

Letters and symbols printed in graphics mode 2 are the same width as graphics mode 1 characters, but twice as tall.



Here's the scoop on graphics modes 1 and 2:

Graphics commands: **GRAPHICS 1** or **GRAPHICS 2** or their abbreviated forms **GR.1** or **GR.2** are used to turn on these two graphics modes.

Screen size: Graphics mode 1's split screen measures twenty characters across by twenty characters down (20X20). Its full-screen size is twenty characters across by twenty-four characters down (20X24).

Graphics mode 2's split screen size is twenty characters across by ten characters down (20X10). Its full screen size is twenty characters across by twelve characters down (20X12).

Colors: Both graphics modes can display up to five colors: Four character colors and one background color.

Plotting in the Text Modes

Many of my most colorful tent shows were presented in the text modes. I often use either graphics mode 1 or 2 because each offers *five colors*. I've even used graphics mode 0 a few times.

Now I'll show you how to PLOT in these text modes so you'll be ready to wow them at the oasis.

Here's a short, but brilliant example:

Text Mode Plotting

```
 10 GRAPHICS 1
 20 GOTO 40
 30 POKE 752,1:COLOR 32:PLOT 2,0
 40 COLOR 87
 50 PLOT 0,0
 60 COLOR 97
 70 PLOT 1,1
 80 COLOR 227
 90 PLOT 2,2
100 COLOR 203
110 PLOT 3,3
120 COLOR 111
130 PLOT 4,4
```

First run this program as is and you'll see "WACKO" printed in brilliant colors.

Now change line 10 to: **10 GRAPHICS 2**, and run it again — BIGGER letters!

Now try it in graphics mode 0 by deleting line 40, then changing line 10 to: **10 GRAPHICS 0**.

What's happening here? The number you put behind the COLOR statement, like COLOR 87, is the ATASCII value (see page 225) of the letter or symbol that will be printed at the PLOT coordinates! In line 40 of my brilliant program, **COLOR 87** prints the first letter of my name, *W*, at coordinates 0,0.

Refer to the Hotski-Totski Chart on page 222 to modify this program and put your name on the silver screen!

Getting Into POSITION

Oh, before I forget (and I never do that!), you can also use PRINT (in GR.0) or PRINT #6; (in GR.1 or GR.2) with the POSITION statement to place characters on the screen. Try this one-liner:

```
10 GRAPHICS 1:POSITION 5,5:PRINT
   #6;CHR$(87); CHR$(97);CHR$(227);
   CHR$(203); CHR$(111)
```

Yup. That's me!

PRINT #6 and El Biggo Texteroonio

In graphics modes 1 and 2, uppercase, lowercase, and inverse video letters, numbers, and symbols normally appear on your screen as *uppercase* text. But, here's the exciting part: Each has *its own distinct color*!

Type in and run this short program, and you'll get the picture. Underlined letters indicate inverse text. Press the perverse inverse character key with the Atari symbol (or diagonal white/black markings) to enter these characters (see page 19). Press it again to return to normal text.

Colorful Letters

```
10 GR.2
20 POSITION 3,5
30 PRINT #6;"El bIggO tEXt"
```

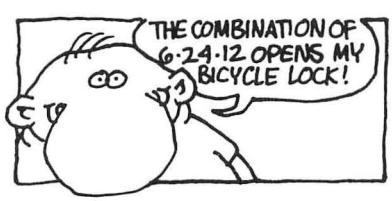
Isn't that shocking! Now let's have some fun.

First, change line 10 to read **GRAPHICS 1** and run it again. Neato!

OK, now we come to the devious part. I'm going to show you how to *mess up all those colors*!

Grab Clyde and safari on over to Table A on page 170. There yet?

Let's give it a try. Type: **POKE 708,99 [RETURN]**. All the letters that were entered as standard uppercase letters — the E in "EL", the I in "BIGGO", and the E in "TEXT" have all miraculously changed color. They're all putrid purple! Eeeyuk!



POKE other numbers into location 708 by changing the number after 708. Try combinations like 708,23; 708,185; and so on, so forth, and to wit.

Figure out that chart yet? You did! Then you realize that you can change the other letter's colors in the same manner.

Here's the explanation Slow Poke gave me:

POKE 710,XX changes the color of inverse uppercase letters.

POKE 709,XX changes the color of standard lowercase letters.

POKE 711,XX changes the color of inverse lowercase letters.

One additional thought. You can also change the background color from black to any color your heart desires by POKEing 712 with any number between 1 and 255 (0 is black so don't waste your time entering it). POKE 712,185 turns the background to my favorite color, oasis green.

TABLE A Color in Graphics Modes 1 and 2

ATASCII Value for Color Register 0	1	2	3	Character	ATASCII Value for Color Register 0	1	2	3	Character
32*	0	160	128	[ ]	50	18	178	146	[2]
33	1	161	129	[!]	51	19	179	147	[3]
34	2	162	130	[”]	52	20	180	148	[4]
35	3	163	131	[#]	53	21	181	149	[5]
36	4	164	132	[$]	54	22	182	150	[6]
37	5	165	133	[%]	55	23	183	151	[7]
38	6	166	134	[&]	56	24	184	152	[8]
39	7	167	135	[’]	57	25	185	153	[9]
40	8	168	136	[(]	58	26	186	154	[:]
41	9	169	137	[)]	59	27	187	None	[;]
42	10	170	138	[*]	60	28	188	156	[<]
43	11	171	139	[+]	61	29	189	157	[=]
44	12	172	140	[,]	62	30	190	158	[>]
45	13	173	141	[-]	63	31	191	159	[?]
46	14	174	142	[.]	64	96	192	224	[@]
47	15	175	143	[/]	65	97	193	225	[A]
48	16	176	144	[0]	66	98	194	226	[B]
49	17	177	145	[1]	67	99	195	227	[C]

ATASCII Value for Color Register				Character
0	1	2	3	
68	100	196	228	D
69	101	197	229	E
70	102	198	230	F
71	103	199	231	G
72	104	200	232	H
73	105	201	233	I
74	106	202	234	J
75	107	203	235	K
76	108	204	236	L
77	109	205	237	M
78	110	206	238	N
79	111	207	239	O
80	112	208	240	P
81	113	209	241	Q
82	114	210	242	R
83	115	211	243	S
84	116	212	244	T
85	117	213	245	U
86	118	214	246	V
87	119	215	247	W
88	120	216	248	X
89	121	217	249	Y
90	122	218	250	Z
91	123	219	251	[
92	124	220	252	\
93	None	221	253	]
94	126	222	254	^
95	127	223	255	_

* 155 selects the same character and color register as value 32.

POKE 708,X FOR COLOR REGISTER 0 (uppercase)
POKE 709,X FOR COLOR REGISTER 1 (lowercase)
POKE 710,X FOR COLOR REGISTER 2 (inverse uppercase)
POKE 711,X FOR COLOR REGISTER 3 (inverse lowercase)

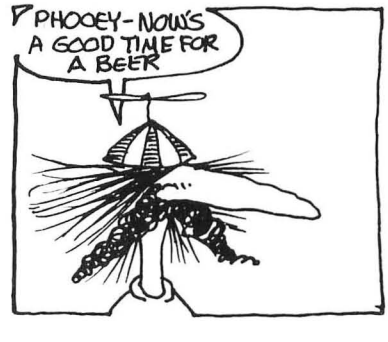
LOCATE and Collisions

After you finish marching through this desert of knowledge you might go beserk like me and decide to try your hand at a little arcade-game design. If you do, you'll want lots of action! When a camel bumps into a palm tree, you'll probably want your computer's screen to display a blinding flash and sound off with an ear-splitting explosion — **BAAROOOOOM!** When a date falls on a character's head, you'll want to hear the action — **BONG!** One way to let the other elements in an arcade game program know that a collision has occurred is to use the LOCATE statement. LOCATE is used in this format: **LOCATE X,Y,Z.**

X is the column location of the collision, Y is the row location, and Z is the value that's encountered at location X,Y. (You don't have to use Z; any variable other than X or Y will work just fine.)

In graphics modes 3 through 8 the value placed in Z at each location is the number that follows the COLOR statement: either 0, 1, 2, or 3.

In graphics modes 0 through 2 the value placed in Z at each location will be the ATASCII value of the character that's encountered. For example, if the letter A is located at coordinates 5,5 in graphics mode 0 (**LOCATE 5,5,Z**), the value placed in Z will be 65. (Now's a good time to refer again to the Hotski-Totski chart on page 222.)



Here's a short program that shows the basic LOCATE concept. My infamous model of the universe program at the end of this chapter uses LOCATE to help me simulate the universe!

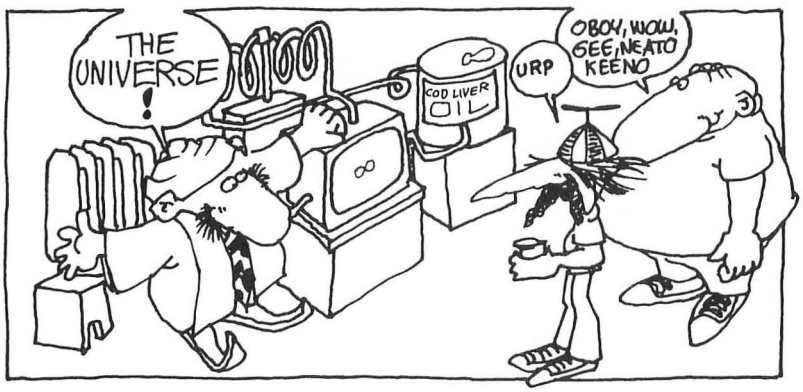
LOCATE Demo

```
 10 GRAPHICS 3
 20 COLOR 1
 30 PLOT 0,0
 40 COLOR 2
 50 PLOT 1,1
 60 COLOR 3
 70 PLOT 2,2
 80 LOCATE 0,0,A
 90 LOCATE 1,1,B
100 LOCATE 2,2,C
110 LOCATE 3,3,D
120 PRINT A,B,C,D
```

After you run this program three colored squares will appear at the upper left of the screen. The corresponding COLOR statement numbers (1, 2, 3, and 0) are LOCATEd, then printed in the text window.

COLOR 1 (register 0 — orange) is plotted at 0,0.
COLOR 2 (register 1 — green) is plotted at 1,1.
COLOR 3 (register 2 — blue) is plotted at 2,2.
COLOR 0 (register 4 — black) is the background color.

Wacko's Infamous Model of the Universe!



And now, here's the wondrous program you've been waiting for. I've titled it "Universe." It was developed at the Wacko Institute of Alchemy many years ago to check out the Big Bong Theory, and demonstrate the improbability of life occurring in a random environment.

Here's the experiment you'll watch on your screen after you enter and run UNIVERSE.

Two randomly controlled colored squares bounce around on Atari's graphic mode 3, two-dimensional screen.

Each square is controlled by three random generators:

1. X coordinate (1-39, inclusive)
2. Y coordinate (1-19, inclusive)
3. COLOR (1-3, inclusive)

A condition, or "Law of Life" is set. In this creative experiment, "life" occurs when both squares arrive at adjacent positions (line 100) at the same time. When they arrive, one square must be orange (**COLOR 1**), and the other square must be blue (**COLOR 3**).

Some Real Heavyweight Stuff

Since this is a simple model (for simple wackos), Petunia told me that the probability of "life" occurring can be calculated by using a simple binomial distribution method. (Easy for her to say. Wheew!)

Her calculations made two assumptions.

Her calulations assume that the random movement of each square and its color will be *truely random*. Because *true random behavior* is real hard to simulate on a computer, her calculations might be off a little bit.

Her calculations also assume that each square will behave independently. Do its own thing, so to speak.

Petunia's binomial distribution calculations for this model show that there is a probability of 2,470,864.5:1 that "life" will occur in each cycle. This means that "life" should statistically occur after 2,470,864.5 cycles. But, "life" may occur during the first cycle, or "life" may never occur at all. Uh oh!

As a matter of scientific interest, if each cycle's duration was 1 second, life "should" occur in 28 days!

One reason for baffling you with such alchemy is to demonstrate how versatile your computer really is. Modeling, from stress analysis (I'm usually under a lot of stress) to a model of the universe, is all within the realm of possibility. That chunk of machinery in front of you is very powerful, if you use your imagination!

There's another reason I'm showing you this great program; it uses many of the elements that you have learned in this chapter. It's also a great example of how to use LOCATE. So without any further "pad dew" (water on a camel's foot), here's Universe:

Universe

```
 0 GRAPHICS 3:POKE 752,1
 10 X = INT(RND(0)*39) + 1
 20 Y = INT(RND(0)*19) + 1
 30 Z = INT(RND(0)*3) + 1
 40 Q = INT(RND(0)*39) + 1
 50 R = INT(RND(0)*19) + 1
 60 S = INT(RND(0)*3) + 1
 65 COLOR 0:PLOT 15,10:DRAWTO 16,10
 70 COLOR S:PLOT Q,R
 80 COLOR Z:PLOT X,Y
 90 LOCATE 15,10,C:LOCATE 16,10,D
100 IF C = 1 AND D = 3 THEN GOTO 100
105 IF C = 1 THEN O = O + 1
107 IF D = 3 THEN B = B + 1
110 POKE 77,0
120 COLOR 0:PLOT X,Y:PLOT
    Q,R:N = N + 1:SOUND 0,X + Y + Q,8,15:?
    CHR$(28);
130 PRINT "ORANGE:";O;" BLUE:";B;" CYCLES:";N
135 COLOR 2:PLOT 15,10:DRAWTO 16,10
140 GOTO 10
```

The Universe Explained (Almost)

Here's a quick journey through all but the SOUND portion of Universe.

Line 0 selects graphics mode 3 and turns off the cursor.

Lines 10 through 60 spit out numbers used to randomly place the squares on the screen and change their colors.

Line 65 erases the two green squares that were drawn in line 135, by drawing the background color (COLOR

0) from coordinates 15,10 to 16,10. These two green squares show where the two random squares must land for "life" to occur.

Lines 70 and 80 PLOT the two randomly colored squares at random locations on the screen.

Here's the exciting part of this program! The two LOCATE statements in line 90 and the IF/THEN statement in line 100 "look" for "life." If C = 1 AND D = 3, the program stops at line 100 — the conditions for "life" have been met!

Lines 105 and 107 update the ORANGE and BLUE readout at the bottom of the screen.

After around ten minutes, if no keys are pressed, the screen goes into what is called the "attract" mode; it starts flashing different colors. In line 110, **POKE 77,0** disables the attract mode.

Line 120 erases the last PLOTed position of each colored square with COLOR 0, it sounds off, then prints CHR$(28) (an up cursor) in the text window to keep the display from scrolling.

Line 130 prints the number of times the ORANGE and BLUE squares land on the "life" coordinates. It also prints the number of CYCLES the program runs through.

Line 135 draws two green squares on the screen at the "life" coordinates.

Line 140 sends the program back to line 10 to repeat the cycle.

CONGRATULATIONS! Pat yourself on the back, have a wild and crazy party! You understand and can now *almost* design programs like this one.

I said "almost" because I will be introducing the SOUND statement used in Universe in the next chapter. But you've come a long way!

You know how to select the proper graphics mode for a program like UNIVERSE, and how to use the COLOR, PLOT, and DRAWTO statements to draw almost anything you'd like. And you understand the amazing LOCATE statement!

Are you ready to make a lot of noise with your computer? Then, put in your earplugs and turn the page!

5
Desert ZOUNDS!

Close your eyes and do a little desert dreaming. If you listen real closely you might hear the hot wind of the Sahara blowing grains of sand across the dunes, the gentle rustling of palm leaves blown by an evening breeze, or the errie sound of flutes, lambskin drums, and a dancer's clacking cymbals.

Open your eyes! (Ooops! I forgot that you can't read this with your eyes closed. Oh, well.)

If you heard all those sounds when you closed your eyes you've really got a great imagination. All I ever hear are Clyde's grunts and Junior's questions.

Now that your eyes are open (I hope), and you've retured to the "real" world, close the door, put cotton in your ears, and get set for some real audio excitement as we delve into the wild world of Atari sound!

Your Atari can sing, or meow, in as many as four voices. You can use each voice solo, or you can be creative and combine a few, or all four to achieve remarkable sound effects, and some beautiful music.

But the best thing about the Atari SOUND statement is that it's so easy to use.

A typical SOUND statement looks like this:

SOUND 0,100,10,15

Type this in and press RETURN. When you have heard enough, type **END [RETURN]**.

Here's what each number in the SOUND statement controls:

VOICE: SOUND 0

Your Atari can sing in harmony, using up to four voices, and you select the voice you'd like to hear by assigning a value to the first number following the SOUND command. SOUND 0, SOUND 1, SOUND 2, and SOUND 3 are all available.

PITCH: SOUND 0,100

Select each voice's pitch by setting the second number to a value between 0 and 255. The pitch I've selected in my example is 100.

DISTORTION: SOUND 0,100,10

The third number in a SOUND statement varies distortion, from pure tones to gobbledygook. Examples of pure tones are 10 and 14. Other even-numbered values (0, 2, 4, 6, 8, and 12) add different amounts of noise and distortion to your tone. But the number controlling the distortion must be an even value between 0 and 14.

VOLUME: SOUND 0,100,10,15

Vary the loudness of each voice by setting the fourth number to a value between 1 and 15. The value 15 is the LOUDEST, while 1 is just a whisper.

Before I show you how to program barrooms, bongs, and arrghs, type in and play around with this short program. It will help you become acquainted with the basic sound concepts you've just learned.

BASIC Sound

```
10 ?:? "ENTER Pitch, Distortion and Loudness.
   Press RETURN after each entry."
20 TRAP 70:INPUT P,D,L
30 SOUND 0,P,D,L
40 IF PEEK(53279)< >6 THEN GOTO 40
45 . **CHECK OUT SMOKEY PEEKS'S LIST OF
   GREAT PEEKS FOR MORE ON
   PEEK(53279)**
50 SOUND 0,0,0,0
60 ?:?: "ZOUNDED GREAT!":GOTO 10
70 ? "YOU MADE A BOO-BOO. TRY
   AGAIN!":GOTO 10
```

A question mark appears on the screen when you run this glamorous program. First enter the *pitch*, then the *distortion*, and finally the *loudness*. Hit RETURN after each entry.

Press START when you've heard enough.

I always enjoy playing with combinations of pitch and distortion. Captain Action showed me these interesting combinations. Try them, and then create some of your own astounding sounds:

Dune Buggy: PITCH = 100, DISTORTION = 4
Desert hum: PITCH = 100, DISTORTION = 6
???: PITCH = 100, DISTORTION = 8
Biplane: PITCH = 250, DISTORTION = 12

A Cacophony of Multivoiciferous Zounds

Want to make a cacophony of multivoiciferous zounds? Just replace lines 30 and 50 in the BASIC Sound program with:

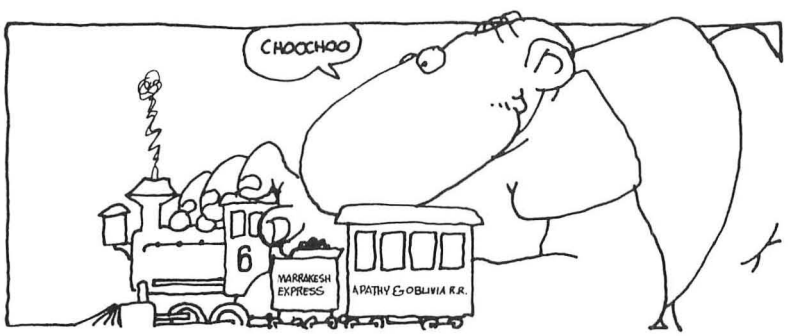
```
30 SOUND 0,P+4,D,L:SOUND
   1,P+8,D,L:SOUND 2,P+12,D,L:SOUND
   3,P+16,D,L
50 FOR OFF=0 TO 3:SOUND OFF,0,0,0:NEXT
   OFF
```

Line 50 is a neat routine that turns all four voices off.

See what happens when you use a whole bunch of voices? You get a whole bunch of noise! Change the values that are added to the pitch in each SOUND statement. Wilder noise!

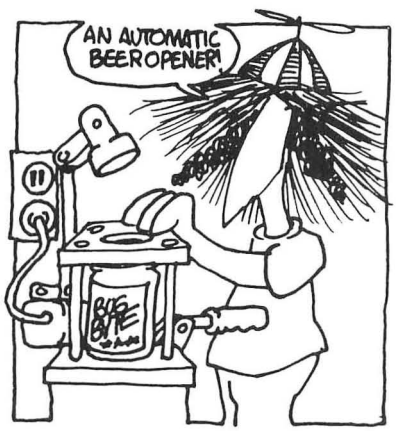
Now That You Understand ZOUND

Now that you understand how the SOUND statement works, it's time to show you how to use SOUND in your programs. But first, here are some extra-special sound treats for you to enjoy. Spend a little time working through each program and then we'll get started.



The Marrakesh Express

```
10 . MARRAKESH EXPRESS
20 GRAPHICS 17:POKE 712,148:POSITION
   1,10:PRINT #6;"MARRAKESH EXPRESS"
30 FOR X = 15 TO 0 STEP - 1 - P:SOUND
   1,0,0,X
40 R = INT(RND(0)*300) + 1
50 IF R = 30 THEN SOUND 3,36,10,10:SOUND
   2,48,10,10:GOSUB 90:SOUND
   3,0,0,0:SOUND 2,0,0,0
60 NEXT X:P = P + 0.03
70 IF P > = 5 THEN P = 5
80 GOTO 30
90 POKE 77,0:POSITION 8,12:PRINT #6;
   "toot":FOR A = 1 TO 400:NEXT A:POSITION
   8,12:PRINT #6;"          ":RETURN
```



Captain Action's Design

```
10 GRAPHICS 17
20 FOR X = 10 TO 100:SOUND 0,X,10,
   10:SOUND 1,X - 2,10,8:SOUND
   2,X + 2,10,12:NEXT X
30 SOUND 1,0,0,0:SOUND 2,0,0,0
40 POSITION 4,11:PRINT #6;
   "BAROOOOMMM!"
50 FOR DECAY = 15 TO 0 STEP - 0.5:SOUND
   0,100,8,DECAY:FOR B = 1 TO 20:POKE
   712,B:NEXT B:NEXT DECAY
60 GRAPHICS 1 + 32:POKE 712,148
70 POKE 752,1:PRINT "Captain Action designed
   this one!"
80 PRINT :PRINT " Press START to blow up
   again!"
90 IF PEEK(53279)< >6 THEN GOTO 90
100 GOTO 10
```

SOUND and the IF/THEN Statement

The IF/THEN statement is often used to introduce sound into a program. It's easy to use. IF, something happens in your program, or game — a certain key is pressed, a ball hits a wall or a rocket ship takes off — THEN, the SOUND statement is activated. And glorious sound accompanies and enhances the action!

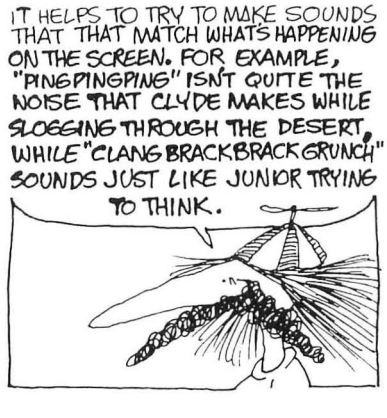
Here's a short and simple example that shows how the IF/THEN works with SOUND:

```
10 PRINT "Press START for a real blast!"
20 IF PEEK(53279)< >6 THEN GOTO 20
30 SOUND 0,150,10,15
40 FOR WAIT = 1 TO 500:NEXT WAIT
50 SOUND 0,0,0,0
60 GOTO 10
```

Now that you're an expert Sahara sound maker, fool around with the SOUND statement in line 120 of the Universe program. Change it to your heart's desire. Captain Action was able to make the computer utter some real far out and spacey sound to accompany the action. It sounded like utter nonsense to me — you can do better then that! See if you can add more SOUND statements, more than one voice, to really make the program come alive — *PLUNK*!

Decay of the PING

A *ping* sound is often heard out here in the desert — when I drop an olive pit down a well, when sand pings against the side of my tent, or when Junior gets verbal. I wanted to simulate this exciting sound on my Atari computer.



A pinging effect is achieved by *decaying* the sound: that is, playing the sound several times and reducing the volume of each play.

Here's how to produce a decaying sound. You'll find many uses for this effect if you decide to design arcade games, create music, or baffle your camel.

Ping!

```
10 FOR DECAY = 15 TO 0 STEP —.8:SOUND
   0,60,10,DECAY:NEXT DECAY
```

A decay decreases a note's volume while sustaining its pitch.

Vary the rate decay rate by changing the negative value that follows the word STEP in the Ping! program. The smaller the negative number, like —.05, the longer the decay.

Play around with this concept a little. You should be able to get Ping! to sound like a note played on a piano!

A decay's opposite number is called an *attack*. An attack increases a note's volume while sustaining its pitch. Just change the program to read

```
10 FOR ATTACK = 0 TO 15 STEP .8:SOUND
   0,60,10,ATTACK:NEXT ATTACK
```

and you'll be attacked!

Let's Go to a Ping Subroutine

In many programs the same sound is called for over and over. In one of the stranger arcade games I designed, the player presses the START key to flip through a variety of screens and maps. A *ping* sound effect ac-

companies each screen change. Rather than repeating the routine each time, you should use a subroutine to create the *ping*.

Type in and run this little gem to get the idea.

```
  5  . *START key PEEK is on line 10!*
 10 IF PEEK(53279)=6 THEN GOSUB 100
 20 GOTO 10
 30 . *HERE COMES THE SUBROUTINE!*
100 FOR DECAY=15 TO 0 STEP -.8:SOUND
     0,60,10,DECAY:NEXT DECAY
200 SOUND 0,0,0,0:RETURN
```

Every time you press the START key, the program branches to the subroutine in line 100. A *ping* sounds, then in line 200 the sound is turned off and the program returns to line 10.

If you are using the same SOUND statement throughout your program, it's easier to GOSUB to a SOUND subroutine. You'll save yourself the effort and time of typing in the same SOUND statements many times, and your program will be more efficient and use less memory.

Changing SOUND with DATA

Complex and changing sounds can be made by READing values into the SOUND statement(s).

Here's a gruesome example. I've named this short program Clyde's Lament. Listen and you'll understand why.

Clyde's Lament

```
  0 . *** Read PITCH & DISTORTION values
      from DATA into SOUND Statement ***
 10 FOR X = 0 TO 3:READ P,D
 20 SOUND 0,P,D,12
 25 . *** PAUSE ***
 30 FOR PAUSE = 1 TO 200:NEXT PAUSE
 40 NEXT X:RESTORE:GOTO 10
 50 . *** The DATA is in pairs: (PITCH,
      DISTORTION) ***
100 DATA 60,2,85,10,150,6,100,8
```

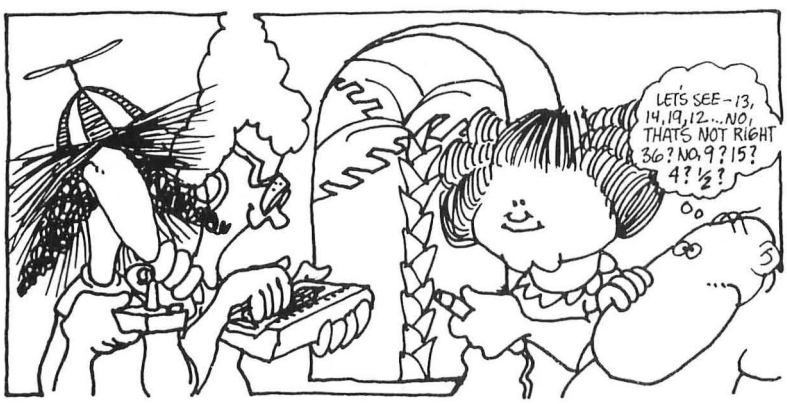
Even though Snidely's wild about Clyde's Lament, most camels (except Clyde) have a more refined musical sense. (No offense to your sense intended, Clyde.) They deserve to hear the real McCoy. Music with a beat. Music they can relate to. Music that sounds like music! It's all possible with Atari SOUND, but that's another book.

Now that you've developed a sound foundation, add sound and weird noise to all your programs. It'll give them a professional touch, and wake up your caravan.

6
Weaving The Perfect Flying Carpet, or, The Art of Programming

If you've got a strong constitution, walk into a hot stuffy room filled with wacked-out programmers (all munching stale tuna fish sandwiches), give them the same problem, and they'll all solve it in their own unique and creative way.

I'll give you an example. I walked into a room full of wackos (and wackettes) and asked them to create a simple version of the game Black Jack (21) to play against the computer.

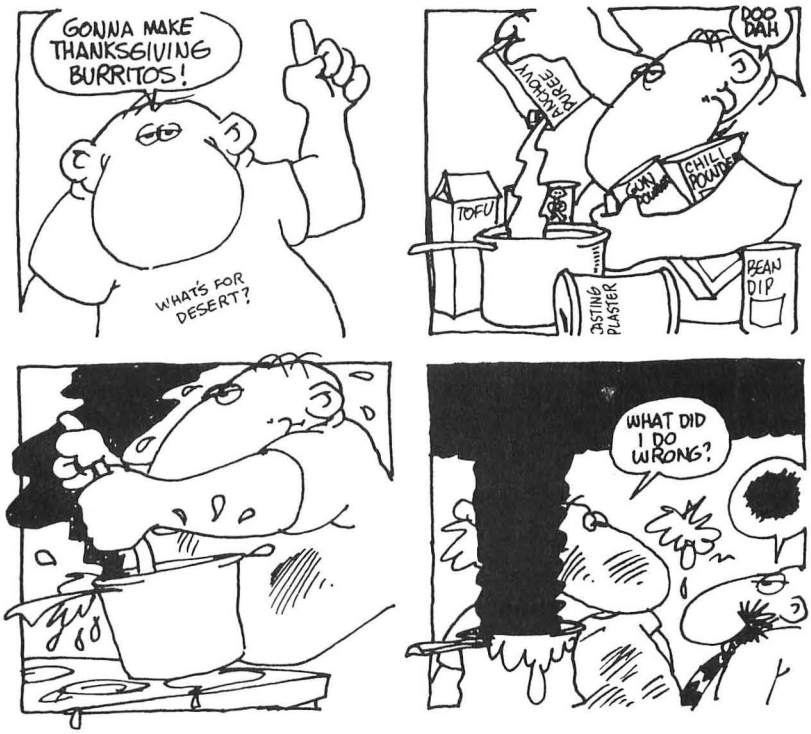


Captain Action's first thought was to use a joystick for the player/computer interface. Ms. Peeky felt that touching the keys was more reassuring, and Petunia went high-tech and decided to get information into the computer by using a light pen. Junior had no comment.

Right off the bat, I knew that these four weirdos would design their programs differently. But it goes even deeper than individuality. The logical thought process

that each programmer goes through while creating a program is unique. He or she can effectively solve the same problem in many different ways.

Pay Attention to Detail, Especially When Making Anchovy Burritos



Programming is just like making anchovy burritos for Thanksgiving dinner. You've got to work out a recipe, go shopping for the ingredients, then put it all together and cook it up. If you've paid close attention to detail, your burritos will be scrumptious. If not, you may end up with burned pots, disgruntled relatives, and lots of former friends.

To show you what goes on in a programmer's brain, I'll let you take a glimpse into mine as I tell you how I designed the Mini Word Processor listed on page 206. (A *word processor* is just a computerized typewriter. I designed my computerized typewriter after the keys of my old electric got jammed with sand.)

Know the Rules of the Game and Define the Problem

I always write down what I want my program to do *before* I start doing any programming. I do this so I'll have a written framework upon which to build my program. Defining the problem — by creating a written description of it — is always an essential first step, regardless of what type of program you are designing. This first step may require more of your brainpower and attention to detail than all of the actual programming! Here's what happened to me....

I wanted my word processor to have six principal features:

1. Show text on the screen.
2. Include at least some rudimentary editing functions.
3. Store text to disk or cassette and load it back into the computer when needed.
4. Print out the contents of the screen onto a printer.
5. Erase the contents of memory and the screen at the press of a key.
6. Be friendly to writers. All options should be easy to remember and easy to use.

I originally thought that designing a simple word processor would be, well, simple. But after writing down all the ingredients I needed, and mulling over the programming consequences, I discovered that I had never really looked closely at the mechanics of a word pro-

cessor. I had never broken it down and examined all the little details.

To write a workable program, I had to examine every feature to gain a thorough and complete understanding of it. In the process, I discovered that there was more to a word processor than meets the eye, and I now have a much better understanding of how a word processor operates.

Programming on The Right Side of The Brain

The process of really looking at a programming problem is the same process that artists go through when they paint still-lifes or portraits. Artists must "see" every minute detail of the subject before they can create a picture on canvas.

A Short Exercise in Seeing

Here's a short exercise in *seeing*. It involves what is called "experiential" learning (learning by doing). Once you're through, you will have experienced total vision, and understand the relationship between art and the art of programming.

Take a piece of paper and a pencil and sit down at the kitchen table. Lay the paper in front of you and place your hand at the side of the paper. Now, take a really close look at that hand. See every line, every curve, every shadow. In a few minutes you'll know more about your hand than you thought possible.

Starting at any point, move your eyes very, very slowly over the edge of your hand. As your eyes do this let your pencil draw this outine on your paper. Really get into "seeing" every curve and shape that make up the outline of your hand. Take your time. Go slowly. Really concentrate.

When you are finished drawing the outline of your hand, go back and fill in all the other details: the curves of your fingernails, the small creases in your hand, even the pores and minute strands of hair on the top of each finger. You'll be pleasantly surprised at the results. This drawing (if you're not already an artist) is probably the most detailed and realistic you've ever done. And all because you *really saw* your hand for the first time!

The Wacko Side of Your Brain

While you were drawing, you may have noticed that you lost sense of time and didn't hear that radio or TV playing in the background or any external sounds. You were completely engrossed in your work. That's because you were using what I call the "Wacko (creative or right) side of your brain." You've also learned to appreciate the effort and attention to detail that goes into a work of art, such as your drawing of your hand.

But what does this have to do with programming? EVERYTHING! When you are programming or designing a program, you need to "see" the final result with the detail artists "see" before they begin painting pictures. *When you program, you must use the Wacko side of your brain!*

"Like, Totally, Really!"

You must totally *see* your programming problem, whether your are designing a word processor, a game, or something totally wacko. By seeing the problem, you gain a total understanding of the problem and can successfully design a program that effectively solves the problem. Total involvement helps you appreciate good programming in yourself and in others.

Now That You Understand The Ground Rules

Once I've defined a program and have an in-depth understanding of its functions, I review each function, apply my knowledge of the BASIC language, and add a touch of wackiness.

You can start working with any of the steps you've written down. But I usually start at the top, with the first one, and work my way down the list. This is the same way that your computer executes a program. So, working this way makes it easier to put together the total program. As you get deeper into the programming you may want to change the order of things to make your program more efficient. Go for it! This is part of the creative programming process.

The Modular Approach

I examine and work through each step, treating it as a small program unto itself, which I call a "module." I write a "shopping list" of programming ingredients needed to make each module of the program work on their own, then formulate a recipe that blends its ingredients.

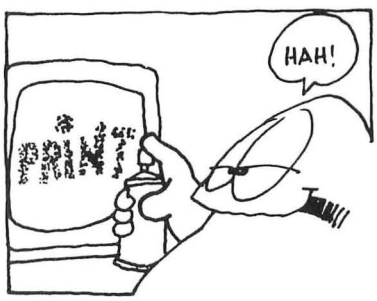
After I've completed a module, I first test it by itself, and then within the entire program . Sometimes I'll design a module directly on the computer. I enter all the programming, then experiment with my module (adding a pinch of this and pinch of that) until it "tastes" just right. This method also works when making spaghetti sauce!

A Word Processor, Step-By-Step

Enough of this conceptualizing stuff. Let's design a word processor! (If you'd like to look at the finished product before we begin, just flip to page 206, type in the program, and start writing that great novel.) All set?

OK, but I'd like to point out one thing before we start designing and programming. The line numbers I use below to explain each module are the same I've used in the finished word processor. This will help you see how it all fits together.

MODULE 1: Print to the Screen



The first thing a word processor must do is print text on the screen. So, I designed Module 1 to do just that.

Module 1

```
 50 CLR :DIM S$(20),L$(20),B(2500)
 60 X = 1:GRAPHICS 0
 70 CLOSE #1:OPEN #1,4,0,"K:"
 80 GET #1,A
 85 . ***OPTIONS***
160 ? CHR$(A);
170 B(X) = A
180 X = X + 1:GOTO 80
```

Line 50 DIMensions **S$** for use in the Save Text Module that comes later and **L$** for use in the Load Text Module. Most important, it DIMensions the array **B** that's used to store the text in line 170 of this module.

Type in and run Module 1. You've got a very rudimentary word processor. Each character you type is printed on your screen, and stored in array **B**!

The OPTIONS, like printing, saving, loading, erasing, and editing the text will fit between lines 80 and 160 of the program.

MODULE 2: Edit

Everything was hunky-dory until I discovered that each time I corrected text by pressing the DELETE BACK S key, the ATASCII value of the DELETE key, 126, was stored in array **B**.

This just wouldn't fly. I realized that if I tried to print or save the text, these extra characters would get in the way and bollix up the works. I had to come up with a way to delete each character on the screen, without placing the ATASCII value of the DELETE key into the array.

Here's the solution I finally came up with, after much experimentation, and futzing around:

Module 2

```
100 IF A = 126 THEN ? CHR$(A);:X = X - 1:GOTO
    80
```

I inserted this short module into Module 1, and solved the problem. Here's how it works. If you press the DELETE BACK S key, **A** equals 126. When this happens the program prints the control character for DELETE on the screen, and the cursor moves one space to the left and erases the character it lands on.

X = X – 1 is the heart of this module. Each time the cursor does its thing on the screen, the array counter is set *back* one notch.

To finish things off, the program returns to line 80 to **GET** the next character.

The solution looks pretty straightforward here. But it took me quite a bit of experimentation to arrive at the result I was looking for. One lesson I learned while working through this problem was to have a very clear idea of the result I wanted *before* I tried to find a solution. It all gets back to defining the problem!

MODULE 3: Saving Text

I wanted to design this module to make saving text as easy as possible. Here's how I wanted it to work:

1. Press OPTION + "S" or "s" and the screen clears
2. A message appears on the screen asking for a device and filename.
3. You enter info and press RETURN.
4. If you make a mistake in entering info, the program goes back to line 70 for another try.
5. Otherwise, text that is stored in array **B(X)** is saved to the proper device. The text is printed on the screen during this process.
6. After the text is stored, the program goes back to line 70 to start again.

Once I had written down what I wanted to accomplish, I started programming. Here's the result:

Module 3

```
115 . GOTO the Save Text Routine
120 IF A = 83 AND PEEK(53279) = 3 OR A = 115
    AND PEEK(53279) = 3 THEN GOTO 200
195 . Save Text Routine
200 CLOSE #1:OPEN #1,12,0,E:"
210 POSITION 3,10:?"[SAVE]
    DEVICE:FILENAME ";
220 INPUT #1,S$
230 CLOSE #1:TRAP 370:OPEN
    #2,8,0,S$:POKE 712,195:POKE 710,195:?
    CHR$(125)
240 FOR T = 1 TO X—1
250 PUT #2,B(T):? CHR$(B(T));:NEXT T
260 CLOSE #2:?CHR$(253);:POKE
    710,148:POKE 712,148:GOTO 70
```

GOTO the Save Text Routine. In line 120, if a you press an uppercase letter "S," **A** equals 83. If you press a lowercase "s," **A** equals 115. And, if the OPTION key is pressed at the same time, the program goes to the *Save Text Routine* beginning on line 200.

The Save Text Routine. In line 200, channel #1 is first closed, then opened to allow writing to and reading from the screen editor. This action also clears the screen and sets the stage for the printing in line 210 and the **INPUT**, less question mark, in line 220.

Line 210 prints a message on the screen asking for a device and filename.

Line 220 waits for an **INPUT**, then assigns it the string variable name **S$**.

Line 230 closes the screen editor (channel #1), then uses a TRAP statement to trap any input errors to line 370. A channel is then opened to the device you specified in line 220, and finally, the screen is colored green and cleared.

The **FOR/NEXT** loop in Lines 240 and 250 takes each character from array **B(T)** and **PUT**s it into the file on whatever device is selected.

When the **FOR/NEXT** loop runs out of characters, the program branches to line 260 where channel #2 is closed and a buzzer sounds to let you know that the saving process is complete. The screen turns blue, and the program goes back up to line 70.

MODULE 4: Loading Text

To keep things simple, I designed this module to load text like the module that saves text. Here's how I wanted it to work:

1. Press OPTION + "L" or "l" and the screen clears.
2. A message appears on the screen and asks for a device and filename.
3. You enter info and press RETURN.
4. If the you make a mistake in entering info, the program goes back to line 70 for another try.
5. Otherwise, text is loaded into array **B(X)**. The text is printed on the screen during this process.
6. After the text is loaded into the computer the program goes back to line 70.

By making the save and load features of this word procesor similar, I eased my programming chores and made the program easy to use. Here's the Module 4 part of this program:

Module 4

```
135 . GOTO the Load Text Routine
140 IF A = 76 AND PEEK(53279) = 3 OR A = 108
    AND PEEK(53279) = 3 THEN
    B(X) = 32:GOTO 270
265 . Load Text Routine
270 CLOSE #1:OPEN #1,12,0,"E:"
280 POSITION 2,10:? "[LOAD]
    DEVICE:FILENAME ";
290 INPUT #1,L$
300 CLOSE #1:TRAP 370:OPEN
    #2,4,0,L$:OPEN #1,8,0,"S:":POKE
    712,195:POKE 710,195:X = 1
310 TRAP 350:GET #2,A
320 PUT #1,A
330 B(X) = A:X = X + 1
340 GOTO 310
350 CLOSE #1:CLOSE #2
360 ? CHR$(253);:POKE 712,148:POKE
    710,148:GOTO 70
```

GOTO the Load Text Routine. This routine is almost identical to the GOTO Save Text Routine you've just seen. There's one small difference, however. Before this routine goes to line 270 (the Load Text Routine), it enters a blank space (32) into array **B(X)**. After much experimentation, I discovered that this space was needed for proper screen formatting. Remove **B(X) = 32** from line 140, run the program, and you'll see what I mean.

The Load Text Routine. This routine is almost identical to the Save Text Routine. But, it **GET**s characters from either disk or cassette and **PUT**s them into array **B(X)**.

MODULE 5: Printer Routine

I decided that it would be nice to add a feature so you can print the contents of the screen to a printer. And here's the module that does this:

Module 5

```
 88 . GOTO Printer Routine
 90 IF A = 80 AND PEEK(53279) = 3 OR A = 112
    AND PEEK(53279) = 3 THEN GOTO 190
185 . Printer Routine
190 LPRINT CHR$(155):CLOSE #1:OPEN
    #2,8,0,"P:":FOR T = 1 TO X - 1:PUT
    #2,B(T):NEXT T:CLOSE #2:GOTO 70
```

GOTO the Printer Routine. In line 90, if you press an uppercase letter "P," **A** equals 80. If you press a lowercase "p," **A** equals 112. And, if you press the OPTION key at the same time, the program goes to the Printer Routine beginning on line 190.

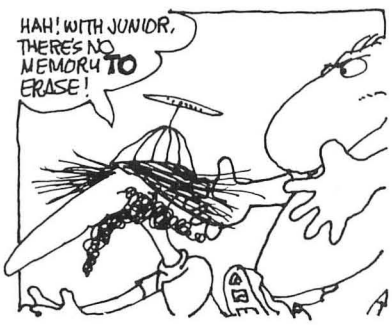
Printer Routine. The Printer Routine on line 190 performs several functions:

1. It prints a carriage return on the printer so printing begins at the left side of the paper.
2. It closes channel #1 then opens channel #2 to the printer.
3. The **FOR/NEXT** loop **PUT**s the contents of the array onto the printer. It goes to **X – 1** so the extra carriage return or space at the end of the text isn't printed.
4. When printing is completed, it closes channel #2 and goes back to the main program beginning on line 70.

Ending Text with a Carriage Return

While experimenting with the Printer Routine, I was surprised to discover that I had to end the text to be

printed with at least one carriage return. When I didn't press RETURN before printing, the printout wasn't the same as the information printed on the screen. This taught me a very valuable lesson: *Always "plug modules in" and test them before moving on."*



MODULE 6: Clear Screen and Erase Memory

I designed this short module so you can clear the screen and memory by pressing the CTRL key plus the uppercase letter "E." I selected the letter "E" to represent "Erase," so the operation would be easy to remember.

Module 6

```
150 IF A = 5 THEN GOTO 50
```

Simple, isn't it? If you press the CTRL key and the "E" key **A** equals 5, and the program goes to line 50 and begins again.

Module 7: One Final Improvement

Since most novels are longer than twenty-four screen lines, I decided to incorporate a feature that lets you scroll a document to *read* and *review* it.

Look at the routine first, then I'll explain how it works and how it's used.

Module 7

```
130 IF A = 18 THEN ?CHR$(125);:FOR T = 1 TO
    X—1:? CHR$(B(T));:NEXT T:GOTO 70
```

If you press the CTRL and "R" keys at the same time, **A** equals 18 and the screen is cleared. Then, the

FOR/NEXT loop is used to print the contents of the array on the screen. When the printing is finished, the program goes back up to line 70.

Here's how to use the "Read" feature:

1. Press the CTRL key plus the "R"key to start the scrolling
2. Press the CTRL key plus the "1" key to pause; press them again to continue scrolling.

Error Handling

The routine on line 370, below, was put into the program after I added the routine on line 130 that lets you scroll long text down the screen.

```
370 ? CHR$(125);:A = 18:GOTO 130
```

In essence, if an error occurs in line 230, the program goes to line 370, branches up to line 130, and finally to line 70 to start again. This may seem like the long way around, but if you review the Text Scrolling Routine, it will all become clear.

The Word Processor That Ate Cleveland

As I designed this program I kept wanting to add more features. I thought it would be nice to add a "wrap-around" feature to automatically shift (wrap) the last word of a line to the left margin of the next line. This common feature on most profesional word processors eliminates pressing RETURN at the end of each line.

I also wanted to add printer functions that would automatically center text, line up text evenly on the right-hand side of the printout, and underline or highlight words.

I also thought that incorporating a "search and replace" feature would be real helpful. And, most of all, I wanted to make use of all the editing functions; not just the DELETE BACK S.

But I knew that if I added all these features, the program would be extremely long and difficult to explain, and you wouldn't have a chance to experiment on your own!

Keep the Cards and Letters Coming!

I'd like to see your modified versions of this word processor. So, if you make some additions, please send me a short note and your program listing. Dr. Wacko will answer you. Trust me!

Adding Some Documentation

Once you've finished your program, worked out the bugs, and really think it's got possibilities, it's time to write operating instructions so your friends can use it. To give you the idea, here's the documentation for Wacko's Amazing Word Processor.

Wacko's Amazing Word Processor

This word processor lets you create text (up to 2500 characters in length) on your screen, save it to either disk or cassette, and print the contents of the screen to a printer.

Easy Operating Instructions

1. *Getting started.* A title page appears on your screen after you load and run your word processor. Just press the START key and you're ready to begin typing.

2. *To save your text.* Press the OPTION key *and* the "S" key simultaniously, and your screen will look like this:

[SAVE] DEVICE:FILENAME

Type a letter "D", a colon (:), and a filename, then press RETURN to save your text to disk. Your entry should look like this:

D:FILENAME.EXT [RETURN]

Type a letter "C", followed by a colon (:), then press RETURN to save your text to cassette. No file name is required.

3. *To Load text into the word processor.* Press the OPTION key *and* the "L" key and your screen will look like this:

[LOAD] DEVICE:FILENAME

Type a letter "D", a colon (:) and the name of the file stored on disk, then press RETURN to load the text into the word processor. Your entry should look like this:

D:FILENAME.EXT [RETURN]

Type a letter "C", followed by a colon (:), then press RETURN to load the text from cassette into the word processor. No file name is required.

4. *To print text on your printer.* Press RETURN at the end of the text you want to print. Then press the OPTION key *and* the "P" key and your text will be printed out. Don't forget to turn on your printer!

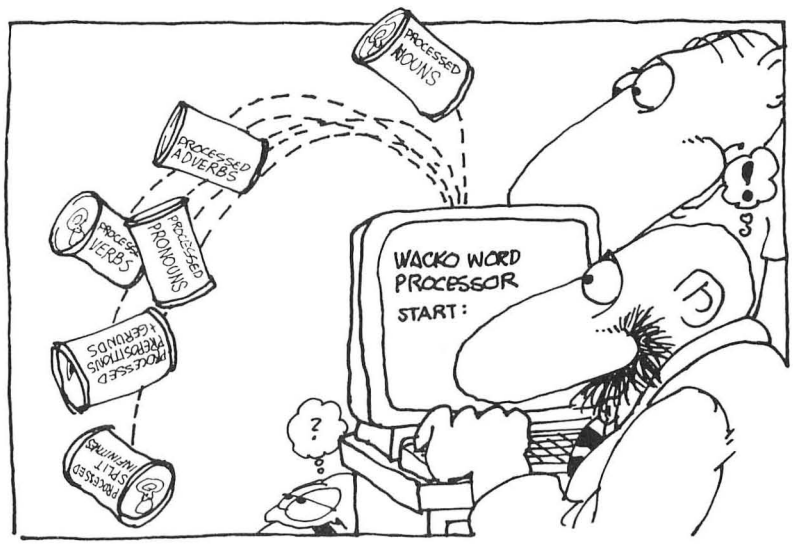
5. *Editing*. The only editing function available on this word processor is the DELETE BACK S key. Each time you press the DELETE BACK S key the cursor moves back one space and deletes one character.

The cursor will travel up the screen a maximum of three lines before stopping. If you continue to press the DELETE BACK S key, characters are still being deleted from the word processor's memory. Use the Read and Review option to see your edited text.

6. *Read and review*. The read and review option is used during editing to see edited text, and to scroll and read text that is longer than the length of the screen.

Press the CTRL key *and* the "R" key to begin scrolling. Press the CTRL key *and* the "1" key to stop scrolling. Press CTRL and "1" again to continue scrolling.

Now that you know how to design and use a word processor, here's the program you've all been waiting for. Type it in carefully, run it, and write that great novel or a nasty letter to me.

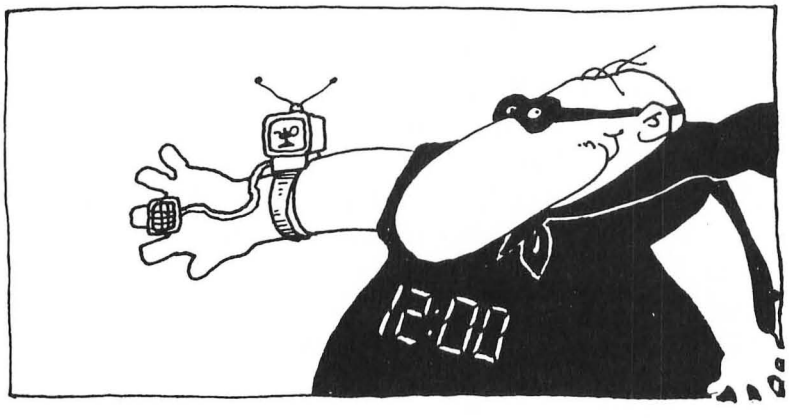


Wacko's Amazing Word Processor

```
 10 GRAPHICS 1 + 16:POKE 712,99:POSITION
    5,8:PRINT #6;"DR WACKO'S":POSITION
    7,10:PRINT #6;"AMAZING"
 20 POSITION 3,12:PRINT #6;"WORD
    PROCESSOR"
 30 POSITION 5,20:PRINT #6;"PRESS start"
 40 IF PEEK(53279)< >6 THEN GOTO 40
 50 CLR :DIM S$(20),L$(20),B(2500)
 55 REM ***PRINT TO SCREEN***
 60 X = 1:GRAPHICS 0
 70 CLOSE #1:OPEN #1,4,0,"K:"
 80 GET #1,A
 85 REM ***OPTIONS***
 90 IF A = 80 AND PEEK(53279) = 3 OR A = 112
    AND PEEK(53279) = 3 THEN GOTO 185:REM
    Print—out. Press RETURN after text!
100 IF A = 126 THEN ? CHR$(A);:X = X—1:GOTO
    80:REM Editing Function
110 IF A = 155 THEN ? CHR$(171):
    B(X) = A:X = X + 1:GOTO 80:REM
    Carraige return
120 IF A = 83 AND PEEK(53279) = 3 OR A = 115
    AND PEEK(53279) = 3 THEN GOTO 200:REM
    Save file
130 IF A = 18 THEN ? CHR$(125);:FOR T = 1 TO
    X—1:? CHR$(B(T));:NEXT T:GOTO 70:REM
    Read long text. CTRL + 1 Stops scroll
140 IF A = 76 AND PEEK(53279) = 3 OR A = 108
    AND PEEK(53279) = 3 THEN B(X) = 32:GOTO
    270:REM Load file
150 IF A = 5 THEN GOTO 50:REM CTRL + E Clears
    the screen
155 REM ***PRINT TO SCREEN***
160 ? CHR$(A);
170 B(X) = A
180 X = X + 1:GOTO 80
185 TRAP  380: REM***PRINTER***
```

```
190 LPRINT CHR$(155):CLOSE #1:OPEN
    #2,8,0,"P:":FOR T = 1 TO X—1:PUT
    #2,B(T):NEXT T:CLOSE #2:GOTO 70
195 REM ***SAVE***
200 CLOSE #1:OPEN #1,12,0,"E:"
210 POSITION 3,10:? "[SAVE]
    DEVICE:FILENAME ";
220 INPUT #1,S$
230 CLOSE #1:TRAP 370:OPEN #2,8,0,S$:POKE
    712,195:POKE 710,195:? CHR$(125)
240 FOR T = 1 TO X—1
250 PUT #2,B(T):? CHR$(B(T));:NEXT T
260 CLOSE #2:? CHR$(253);:POKE 710,148:POKE
    712,148:GOTO 70
265 REM ***LOAD***
270 CLOSE #1:OPEN #1,12,0,"E:"
280 POSITION 2,10:? "[LOAD]
    DEVICE:FILENAME ";
290 INPUT #1,L$
300 CLOSE #1:TRAP 370:OPEN #2,4,0,L$:OPEN
    #1,8,0,"S:":POKE 712,195:POKE 710,195:X = 1
310 TRAP 350:GET #2,A
320 PUT #1,A
330 B(X) = A:X = X + 1
340 GOTO 310
350 CLOSE #1:CLOSE #2
360 ? CHR$(253);:POKE 712,148:POKE
    710,148:GOTO 70
365 REM ***SAVE/LOAD ERRORS***
370 ? CHR$(125);:A = 18:GOTO 130
380 ? CHR$(125);:POKE 710,34:POSITION 4,10:?
    "Please Turn Printer/Interface On!";
390 FOR WAIT = 1 TO 500:NEXT
    WAIT:A = 18:POKE 710,148:GOTO 130
```

A Special Bonus Coder/Decoder



Once you've created a message using Wacko's Amazing Word Processor, you can protect it from prying eyes with the next two bonus programs: Coder and Decoder.

Here are the two program listings. Just enter and save each one, then I'll tell you how to protect your text.

Coder

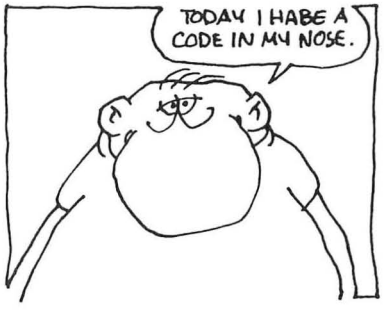
```
 10 CLR :CLOSE #1:? CHR$(125):X = 1
 15 POKE 710,0:POSITION 15,1:?
    "***CODER***":?
 20 DIM C(15000),A$(20),C$(20)
 30 ? "ENTER TODAY'S CODE:";
 40 TRAP 10:INPUT D
 50 ? :? "CLEAR TEXT —
    DEVICE:FILENAME.EXT"
 60 TRAP 10:INPUT A$
 70 OPEN #1,4,0,A$
 80 TRAP 110:GET #1,A
 85 ? CHR$(A);
 90 X = X + 1:C(X) = A + D
100 GOTO 80
```

```
110 CLOSE #1:? :? CHR$(253):? :X = 1
120 ? "CODED TEXT —
    DEVICE:FILENAME.EXT"
130 TRAP 10:INPUT C$
140 OPEN #1,8,0,C$
145 IF C(X)—D = 91 THEN GOTO 180
150 TRAP 180:X = X + 1:PUT #1,C(X)
170 GOTO 145
180 CLOSE #1:? CHR$(253):? "FINITO!"
```

Decoder

```
 10 CLR :X = 1:CLOSE #1:? CHR$(125)
 15 POKE 710,0:POSITION 15,1:?
    "***DECODER***":?
 20 DIM C(15000),A$(20),B$(20)
 30 ? "ENTER TODAY'S CODE:";
 40 TRAP 10:INPUT D
 50 ? :? "CODED TEXT —
    DEVICE:FILENAME.EXT"
 60 TRAP 10:INPUT A$
 70 OPEN #1,4,0,A$
 80 TRAP 110:GET #1,A
 90 X = X + 1:C(X) = A—D
 95 ? CHR$(C(X));
100 GOTO 80
105 ? CHR$(125)
110 CLOSE #1
120 ? :? CHR$(253)
130 ? "CLEAR TEXT —
    DEVICE:FILENAME.EXT"
135 X = 1
140 TRAP 10:INPUT B$
150 OPEN #1,8,0,B$
155 IF C(X) = 91 THEN GOTO 190
160 TRAP 190:PUT #1,(C(X))
180 X = X + 1:GOTO 155
190 CLOSE #1
200 ? CHR$(253):? "FINITO!"
```

To Code Your Text



Important: When you compose your text, type at least one Left Bracket "[" as the last character. Both programs recognize the left bracket as the end of text.

Run the Code program and you'll be asked to "ENTER TODAY'S CODE." Enter any *whole* number you'd like and press RETURN. (Remember this number. You'll need it to decode your text!)

Next, you'll be asked for the device and filename of the text (stored on disk or cassette) that you want coded. Your screen will look like this:

CLEAR TEXT — DEVICE:FILENAME.EXT

If your text is stored on a disk, respond by entering the device symbol "D", followed by a colon (:), and the name of the text file stored on your disk.

D:FILENAME.EXT

If your text is stored on cassette, just enter "C", followed by a colon. (**C:**). No file name is required.

Press RETURN, and text from the file you've selected will scroll down your screen.

You'll hear a buzzer, and the words

CODED TEXT — DEVICE:FILENAME.EXT

will appear on your screen.

Enter the storage device and the filename (disk only) you've chosen for your coded text. When the word "FINITO!" appears on your screen, a new file of coded text has been created on your disk or cassette.

To Decode

Run the Decode program and you'll be asked to "ENTER TODAY'S CODE." Enter the same number you used when coding your text. Press RETURN.

Next, you'll be asked for the device and filename of the text (stored on disk or cassette) that you want to decode. Your screen will look like this:

CODED TEXT — DEVICE:FILENAME.EXT

If your text is stored on a disk, respond with the device symbol "D", followed by a colon (:), and the name of the text file stored on your disk.

D:FILENAME.EXT

If your text is stored on cassette just enter "C", followed by a colon. (**C:**). No file name is required. Press RETURN.

You'll hear a buzzer, and the words

CLEAR TEXT — DEVICE:FILENAME.EXT

will appear on your screen.

Enter the storage device and the filename (disk only) you've chosen for your decoded text. When the word "FINITO!" appears on your screen, a new file of decoded text has been created on your disk or cassette.

Enjoy your new secret Coder/Decoder!

DR. WACKO PRESENTS ATARI BASIC

WANT MORE? WE GOT MORE! RUN DOWN TO YOUR LOCAL STORE AND GET DR. C. WACKO'S MIRACLE GUIDE TO DESIGNING AND PROGRAMMING YOUR OWN ATARI COMPUTER ARCADE GAMES!

APPENDIX A: UH OH! ERROR MESSAGES

Your Atari computer lets you know that there are problems by printing numbers on the screen. Here's what those numbers mean.

2 **Not Enough Memory.** There is not enough memory left in RAM for your BASIC program or variables.

3 **Weird Numeric Value.** A numeric value that was expected to be positive is negative, or a value is way out of wack.

4 **Too Many Variables.** You've used too many variables! A program can only have a maximum of 128 different variable names.

5 **A String is Too Long.** Your program tried to use a string that is longer than the DIMensioned string length.

6 **Not Enough DATA.** A READ statement tried reading past the end of the DATA statement's list of values and dropped off the edge of the program.

7 **Number Greater Than 32767.** Oops! You can't use a value greater than 32767.

8 **INPUT Boo Boo.** You tried to INPUT a non-numeric value (such as a letter, punctuation mark, or control character) into a numeric variable.

9 **Array or String DIMension Error.** Your DIM statement might contain a string variable or array that you've already DIMensioned; or you've tried to use an array larger than 32,767 bytes; or your program tried using an unDIMensioned string variable or array or an array element that doesn't exist.

10 You're Getting Too Fancy. There are too many GOSUBs, or an expression has too many levels of parentheses. (((par)(enth)eses)))...good grief!

11 Numeric Zonkers. Your program tried to divide by zero or refer to a number larger than $1X10^{98}$ or smaller than $1X10^{-99}$.

12 Can't Find That Line!. A GOSUB, GOTO, IF-THEN, ON-GOSUB or ON-GOTO tried to branch to a nonexistent line number and got lost.

13 A NEXT Without a FOR. Your program bumped into a NEXT statement, but there wasn't a complementary FOR statement for it to refer to. It got confused.

14 A Line is Too Long. A statement is too complex or too long for BASIC to handle on one logical line.

15 A GOSUB or FOR Statement Vanished. A NEXT or RETURN statement looked for a FOR or GOSUB and couldn't find it.

16 The Program That Never RETURNed. A RETURN statement tried to return but couldn't find a matching GOSUB.

17 Garbage in Garbage out. Something weird happened. A strange POKE, machine language routine, or faulty RAM caused your program to try to execute completely meaningless garbage.

18 Strange String. A string doesn't start with a valid character (like a number), or a string in a VAL statement is not a numeric string.

19 The Program's Too Big. The program you're trying to load won't fit into the available RAM.

20 Change The Channel. The program tried to use channel 0 or tune in on a channel larger than 7.

21 It's Not in LOAD Format. You tried using a LOAD statement to load a program that you stored by using an ENTER or CSAVE statement.

128 BREAK It Up. You pressed the BREAK key while the computer was thinking.

129 The Channel Is Already Open. The program tried using a channel that was already being used.

130 The Nonexistent Device. The program tried using a device that doesn't exist. For example, D:, P:, and S: do exist; Q: doesn't.

131 Reading from a Write-Only Device. The program tried to read from a device that you can only write to, such as a printer.

132 Bad XIO Grammar. Something's wrong with an XIO (Input/Output) command.

133 The Channel's Not Open. The program tried to "speak" to a device before OPENing it.

134 Bad Device or Channel Number. The program can only use channels 1 through 7.

135 Writing to a Read-Only Device. The program tried to write to a device that you can only read from.

136 End Of File. The program bumped into the end of a file or tried to read a sector on a disk that wasn't part of the opened file.

137 Truncated Record. The program tried to read a record longer than 256 characters and truncated it.

138 Time Out. A specified external device (like the disk drive or printer) doesn't respond. Are they turned on?

139 A Device Can't Perform. The disk drive, program recorder or ATARI 850 Interface Module can't perform a command.

140 Serial Bus Problem. There's a serial bus problem. Your diskette or cassette may be defective.

141 The Cursor is Out of Range. The cursor tried to whiz off the side of the screen.

142 Another Serial Bus Problem. There's a serial bus problem. Check your diskette or cassette, it may be defective.

143 Yet Another Serial Bus Problem. Are you sure your diskette or cassette is OK?

144 The Disk Is Protected!. You can't write to a diskette because it is physically write-protected or its directory is scrambled.

145 Disk Drive Comparison Glitch. The disk drive found a difference between what it wrote and what is was supposed to write. (I often have the same problem) Or, there is a problem with the screen handler.

146 It Tried To Do The Impossible. The program failed while trying to do the impossible, like inputting from the printer or outputting to the keyboard.

147 Not Enough RAM for Graphics. There's not enough memory space in RAM for all those fancy graphics.

150 Too Many Serial Ports Are Open. You can only open one serial port to one channel at a time.

160 I Don't Know That Drive Number. Drive numbers can only be D:, D1:, D2:, D3:, or D:4. If you tried something like "D:5", you got this error.

161 There Are Too Many Files Open. You can only open three disk files at the same time.

162 The Disk Is Full (Burp!). There's no more room on the diskette.

163 What's Happening?. While the computer was inputting or outputting data, it found and error and can't determine its cause or recover from it. Wheew.

164 POINTed in The Wrong Direction. A POINT statement inadvertently moved the file pointer outside of the opened file. You might also get this error if the disk file is garbled.

165 The Old File Name Error. A file name contained weird characters, started with a lowercase letter or used improper wild-card characters.

167 It's Locked Up. The disk files can't be written to or erased because they're locked up. (Le' me outa here!!)

169 The Directory Is Full. The disk directory only has room for sixty-four names.

170 I Can't Find That File. You've got to be precise. If the file name you use isn't exactly the same as the name on the directory, you'll get this error.

APPENDIX B: STORING AND RETRIEVING YOUR PROGRAMS

If You Have A Disk Drive

There are two ways to store and retrieve your programs from disk.

SAVE Your Program

The first, and most common, way to store a program on disk is to **SAVE** it by using the following BASIC format:

SAVE"D:FILENAME.EXT"

Type the **SAVE** command first, then enclose the *device* symbol, *filename*, and *extension* in quotation marks.

DEVICE

In this example the *device* is **D:**, which means that you are saving your program to disk drive 1. If you want to save your program to another disk drive, disk drive 2 for example, just add the drive number(2) after the "D" like this: **D2:**.

FILENAME

The *filename* can be any name of your choice, but you have to follow these rules:

1. It can never exceed eight (8) characters.
2. It must begin with a letter; it cannot begin with a number or symbol.

Here are some examples:

MYFILE is OK.

2MYFILE begins with a number and cannot be used.

MYGREATFILE is too long.

EXTENSION

Using an *extension* is optional. It can help you keep track of the type of program you've saved, or to further identify the program.

If you want to use an *extension*, just place a period (.) after the *filename*, then type your *extension*.

The EXTENSION also has a couple of rules you must follow:

1. It cannot be longer than three (3) characters.
2. It may begin with a number.

Here are a few examples of program names with *extensions*:

FILENAME.BAS — "BAS" is the *extension*, and is shorthand for "BASIC."

FILENAME.LST — "LST" is the *extension*, and is shorthand for "LIST."

FILENAME.OBJ — "OBJ" is the *extension*, and is shorthand for "OBJECT CODE" (machine language program).

The LOAD Command

The **LOAD** command loads programs in **SAVE**ed format from your disk into your computer. Here's an example:

LOAD"D:FILENAME.EXT" [RETURN]

LIST Your Program

When you want to merge one program with another, a variation of the familiar **LIST** command comes in real handy.

To **LIST** your program to disk, use the following BASIC format:

LIST"D:FILENAME.EXT"

Type the **LIST** command first, then enclose the device symbol, *filename*, and *extension* in quotation marks.

Merge Two Programs

Here's a short example that shows how to merge two programs.

First, type this one-line program and LIST it to your disk. Name this program, "ONE".

10 PRINT "WACKO";

Now, use the NEW command to clear out your computer's memory, type this one—liner, and LIST it to your disk. Name this program, "TWO."

20 GOTO 10

Use the NEW command again, and we'll merge the two short programs.

The ENTER Command

The **ENTER** command loads programs in LISTed format from your disk into your computer.

First, ENTER the program called "ONE," then enter "TWO" like this:

1. **ENTER"D:ONE"** [RETURN]
2. **ENTER"D:TWO"** [RETURN]

Now, to make sure both programs are in your computer type "LIST" and press RETURN.

All set? OK, RUN the program and go absolutely wacko!

If You Have A Program Recorder

To SAVE/LOAD and LIST/ENTER programs to and from your program recorder use the device symbol "C:", but *don't* use a *filename* or *extension*. Here are some examples:

```
SAVE"C:"
LOAD"C:"

LIST"C:"
ENTER"C:"
```

You can also SAVE and LOAD programs to and from your program recorder with these two commands:

```
CSAVE
CLOAD
```

APPENDIX C: ATASCII CODES

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
0		CTRL-,	8		CTRL-H
1		CTRL-A	9		CTRL-I
2		CTRL-B	10		CTRL-J
3		CTRL-C	11		CTRL-K
4		CTRL-D	12		CTRL-L
5		CTRL-E	13		CTRL-M
6		CTRL-F	14		CTRL-N
7		CTRL-G	15		CTRL-O

APPENDIX C: ATASCII CODES

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
16		CTRL-P	31		ESC/CTRL-*
17		CTRL-Q	32		SPACE BAR
18		CTRL-R	33		SHIFT-1
19		CTRL-S	34		SHIFT-2
20		CTRL-T	35		SHIFT-3
21		CTRL-U	36		SHIFT-4
22		CTRL-V	37		SHIFT-5
23		CTRL-W	38		SHIFT-6
24		CTRL-X	39		SHIFT-7
25		CTRL-Y	40		SHIFT-9
26		CTRL-Z	41		SHIFT-0
27		ESC/ESC	42		SHIFT-*
28		ESC/CTRL--	43		+
29		ESC/CTRL-=	44		,
30		ESC/CTRL-+	45		-

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
46	.	.	61	=	=
47	/	/	62	>	>
48	0	0	63	?	SHIFT-/
49	1	1	64	@	SHIFT-8
50	2	2	65	A	A
51	3	3	66	B	B
52	4	4	67	C	C
53	5	5	68	D	D
54	6	6	69	E	E
55	7	7	70	F	F
56	8	8	71	G	G
57	9	9	72	H	H
58	:	SHIFT-;	73	I	I
59	;	;	74	J	J
60	<	<	75	K	K

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
76	L	L	91	[	SHIFT-,
77	M	M	92	\	SHIFT-+
78	N	N	93	]	SHIFT-.
79	O	O	94	^	SHIFT-*
80	P	P	95	–	SHIFT--
81	Q	Q	96	♦	CTRL-,
82	R	R	97	a	(LOWR) A
83	S	S	98	b	(LOWR) B
84	T	T	99	c	(LOWR) C
85	U	U	100	d	(LOWR) D
86	V	V	101	e	(LOWR) E
87	W	W	102	f	(LOWR) F
88	X	X	103	g	(LOWR) G
89	Y	Y	104	h	(LOWR) H
90	Z	Z	105	i	(LOWR) I

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
106	j	(LOWR) J	121	y	(LOWR) Y
107	k	(LOWR) K	122	z	(LOWR) Z
108	l	(LOWR) L	123		CTRL-;
109	m	(LOWR) M	124		SHIFT- =
110	n	(LOWR) N	125		ESC/CTRL-<
111	o	(LOWR) O	126		or ESC/SHIFT-< ESC/BACK S
112	p	(LOWR) P	127		ESC/TAB
113	q	(LOWR) Q	128		(^) CTRL-,
114	r	(LOWR) R	129		(^) CTRL-A
115	s	(LOWR) S	130		(^) CTRL-B
116	t	(LOWR) T	131		(^) CTRL-C
117	u	(LOWR) U	132		(^) CTRL-D
118	v	(LOWR) V	133		(^) CTRL-E
119	w	(LOWR) W	134		(^) CTRL-F
120	x	(LOWR) X	135		(^) CTRL-G

APPENDIX C: ATASCII CODES

Decimal Code	ATASCII Character	Keys to Create Character
136		(↑) CTRL-H
137		(↑) CTRL-I
138		(↑) CTRL-J
139		(↑) CTRL-K
140		(↑) CTRL-L
141		(↑) CTRL-M
142		(↑) CTRL-N
143		(↑) CTRL-O
144		(↑) CTRL-P
145		(↑) CTRL-Q
146		(↑) CTRL-R
147		(↑) CTRL-S
148		(↑) CTRL-T
149		(↑) CTRL-U
150		(↑) CTRL-V
151		(↑) CTRL-W
152		(↑) CTRL-X
153		(↑) CTRL-Y
154		(↑) CTRL-Z
155	EOL	(↑) RETURN
156		ESC/SHIFT-BACK S
157		ESC/SHIFT->
158		ESC/CTRL-TAB
159		ESC/SHIFT-TAB
160		(↑) SPACE BAR
161		(↑) SHIFT-1
162		(↑) SHIFT-2
163		(↑) SHIFT-3
164		(↑) SHIFT-4
165		(↑) SHIFT-5

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
166	&	(inv) SHIFT-6	181	5	(inv) 5
167	'	(inv) SHIFT-7	182	6	(inv) 6
168	(	(inv) SHIFT-9	183	7	(inv) 7
169	)	(inv) SHIFT-0	184	8	(inv) 8
170	*	(inv) SHIFT-*	185	9	(inv) 9
171	+	(inv) +	186	:	(inv) SHIFT-;
172	,	(inv) ,	187	;	(inv) ;
173	-	(inv) -	188	<	(inv) <
174	.	(inv) .	189	=	(inv) =
175	/	(inv) /	190	>	(inv) >
176	0	(inv) 0	191	?	(inv) SHIFT-/
177	1	(inv) 1	192	@	(inv) SHIFT-8
178	2	(inv) 2	193	A	(inv) A
179	3	(inv) 3	194	B	(inv) B
180	4	(inv) 4	195	C	(inv) C

APPENDIX C: ATASCII CODES

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
196	D	(↑) D	211	S	(↑) S
197	E	(↑) E	212	T	(↑) T
198	F	(↑) F	213	U	(↑) U
199	G	(↑) G	214	V	(↑) V
200	H	(↑) H	215	W	(↑) W
201	I	(↑) I	216	X	(↑) X
202	J	(↑) J	217	Y	(↑) Y
203	K	(↑) K	218	Z	(↑) Z
204	L	(↑) L	219	[	(↑) SHIFT-,
205	M	(↑) M	220	\	(↑) SHIFT-+
206	N	(↑) N	221	]	(↑) SHIFT-.
207	O	(↑) O	222	^	(↑) SHIFT-*
208	P	(↑) P	223	_	(↑) SHIFT--
209	Q	(↑) Q	224	♦	(↑) CTRL-.
210	R	(↑) R	225	a	(↑) (LOWR) A

Decimal Code	ATASCII Character	Keys to Create Character	Decimal Code	ATASCII Character	Keys to Create Character
226	b	(inv) (LOWR) B	241	q	(inv) (LOWR) Q
227	c	(inv) (LOWR) C	242	r	(inv) (LOWR) R
228	d	(inv) (LOWR) D	243	s	(inv) (LOWR) S
229	e	(inv) (LOWR) E	244	t	(inv) (LOWR) T
230	f	(inv) (LOWR) F	245	u	(inv) (LOWR) U
231	g	(inv) (LOWR) G	246	v	(inv) (LOWR) V
232	h	(inv) (LOWR) H	247	w	(inv) (LOWR) W
233	i	(inv) (LOWR) I	248	x	(inv) (LOWR) X
234	j	(inv) (LOWR) J	249	y	(inv) (LOWR) Y
235	k	(inv) (LOWR) K	250	z	(inv) (LOWR) Z
236	l	(inv) (LOWR) L	251		(inv) CTRL-;
237	m	(inv) (LOWR) M	252		(inv) SHIFT- =
238	n	(inv) (LOWR) N	253		(inv) ESC/CTRL-2
239	o	(inv) (LOWR) O	254		(inv) ESC/CTRL-BACK S
240	p	(inv) (LOWR) P	255		(inv) ESC/CTRL->

APPENDIX D: SMOKEY PEEK'S POKES & PEEKS

This appendix includes some of my favorite memory locations. You'll find them useful as you continue to explore the power of your Atari computer.

The PEEK function lets you "look" into a memory location and read its contents. The POKE statement lets you stuff information directly into a memory location.

16 and 53774: Used together *after each Graphics statement*, they disable the BREAK key. Program example:

```
10 POKE 16,64:POKE 53774,64:GOTO 10
```

20: PEEKing this location will give you a source of ever-changing numbers. Program example:

```
10 POKE 710,PEEK(20):GOTO 10
```

580: POKEing 580 with 1 (**POKE 580,1**) at the beginning of your program purges the existing program from memory when you press the SYSTEM RESET key.

Color Registers: 708 to 712

708: Controls color register 0.

709: Controls color register 1.

710: Controls color register 2.

711: Controls color register 3.

712: Controls color register 4.

752: POKE 752 with 0 (**POKE 752,0**) to get rid of the cursor. Use **POKE 752,1** to put the cursor back on the screen.

764:PEEK(764) returns a different value for each key you press. You can use PEEK(764) in an IF/THEN statement to tell your program to do something when you press a specific key. Here's what I mean.

First RUN this program, press some keys, and watch the results:

```
10 PRINT PEEK(764):GOTO 10
```

Now we'll use this PEEK in a short example:

```
10 IF PEEK(764) = 62 THEN GOTO 30
20 GOTO 10
30 FOR X = 1 TO 20:SOUND 0,50,6,10:NEXT
   X:SOUND 0,0,0,0
40 POKE 764,255:GOTO 10
```

Just press the "S" key a few times and listen to the glorious results!

POKEing 764 with 255 (**POKE 764,255**) in line 40 resets this location back to its normal setting.

53279: PEEK(53279) tells which special function key you have pressed. Oasis 5 shows how to get the most from this very special PEEK, which also shows up in the Wacko Word Processor. Here's a handy chart that shows you how to use this great PEEK:

Values of PEEK(53279)

VALUE	KEY(S) PRESSED
0	OPTION, SELECT and START
1	OPTION and SELECT
2	OPTION and START
3	OPTION
4	SELECT and START
5	SELECT
6	START

Just enter and RUN this short program, press the special-function keys, and watch the results:

```
10 PRINT PEEK(53279):GOTO 10
```

APPENDIX E: GRAPHICS MODE CHART

	GRAPHICS MODE	DISPLAY TYPE	AVAILABLE COLOR	SCREEN SIZE (COLUMNS x ROWS)	COLOR REGISTER NUMBER(S)			COLOR, REGISTER NUMBER AND POKE	MEMORY USED (BYTES)
					CHARACTERS OR PIXELS	BACKGROUND	BORDER		
TEXT MODES	0	STANDARD TEXT	1 color & VARIABLE CHARACTER LUMINANCE	40 x 24	1 (VARIABLE LUMINANCE ONLY)	2 (BLUE)	4 (BLACK)	—	993
	1	DOUBLE-WIDTH TEXT	5	20 x 20 (SPLIT) 20 x 24 (FULL)	0, 1, 2, 3	4	4	(SEE TABLE 2.2)	513
	2	DOUBLE-WIDTH DOUBLE-HEIGHT TEXT	5	20 x 10 (SPLIT) 20 x 12 (FULL)	0, 1, 2, 3	4	4	(SEE TABLE 2.2)	261
PIXEL MODES	3	FOUR COLOR GRAPHICS	4	40 x 20 (SPLIT) 40 x 24 (FULL)	0, 1, 2	4	4	COLOR 0, REGISTER 4 POKE 712,0	273
	5		4	80 x 40 (SPLIT) 80 x 48 (FULL)	0, 1, 2	4	4	COLOR 1 REGISTER 0 POKE 708,40 COLOR 2 REGISTER 1 POKE 709,202	1017
	7		4	160 x 80 (SPLIT) 160 x 96 (FULL)	0, 1, 2	4	4	COLOR 3 REGISTER 2 POKE 710,148	3945
	4	TWO COLOR GRAPHICS	2	80 x 40 (SPLIT) 80 x 48 (FULL)	0	4	4	COLOR 0 REGISTER 4 POKE 712,0	537
	6		2	160 x 80 (SPLIT) 160 x 96 (FULL)	0	4	4	COLOR 1 REGISTER 0 POKE 708,40	2025
	8	HIGH RESOLUTION GRAPHICS	1 color & VARIABLE CHARACTER LUMINANCE	360 x 160 (SPLIT) 360 x 192 (FULL)	1 (VARIABLE LUMINANCE AND PHASE SHIFT color)	2	4	COLOR 0 REGISTER 2 POKE 710,148 COLOR 1 REGISTER 1 POKE 709,202	7900

INDEX

C 4702

> $12.95 FPT USA

Dr. C. Wacko Presents

ATARI BASIC

and the

Whiz-Bang Miracle Machine

Does Atari BASIC make you a little dry in the mouth? Wondering whether the only thing writing your own programs will ever do for you is make you hot under the collar? Did reading that last BASIC programming book feel like a jog across the Sahara at high noon? Dr. C. Wacko (and friends) to the rescue!

Yes, folks, back from safari by popular demand, it's *Dr. C. Wacko Presents Atari BASIC and the Whiz-Bang Miracle Machine.* Now, even people who don't know figs about computers can learn to program and have fun with their Atari. "But real programming can't be fun," you say? Enter the sun-drenched world of the good doctor and you'll not only get a great tan, but you'll:

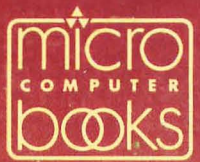
- grunt your way through Basic BASIC Training and get in shape for your programming adventure
- wander through the Great White Expanse, drinking from the Well of Programming Knowledge at every oasis
- don sunglasses and enter the world of Atari graphics
- create enchanting sounds and haunting melodies with Atari sound
- weave the perfect flying carpet while learning the Art of BASIC programming

Create word processors, computer files, games, encoders and decoders, and even models of the universe? Without knowing a byte of BASIC? Yup, Dr. C. Wacko and his band of desert renegades make it all possible.

Take the

Wacko Challenge!

Pick up any other BASIC programming book on the shelf. No, not that one, the one over there. Open it. Now open *Dr. C. Wacko Presents Atari BASIC and the Whiz-Bang Miracle Machine.* Which one will you trust to guide your programming career? Well, you're not alone, since surveys show that BASIC programmers prefer the Wacko method 3–1.



ADDISON-WESLEY PUBLISHING COMPANY

ISBN 0-201-11491-7