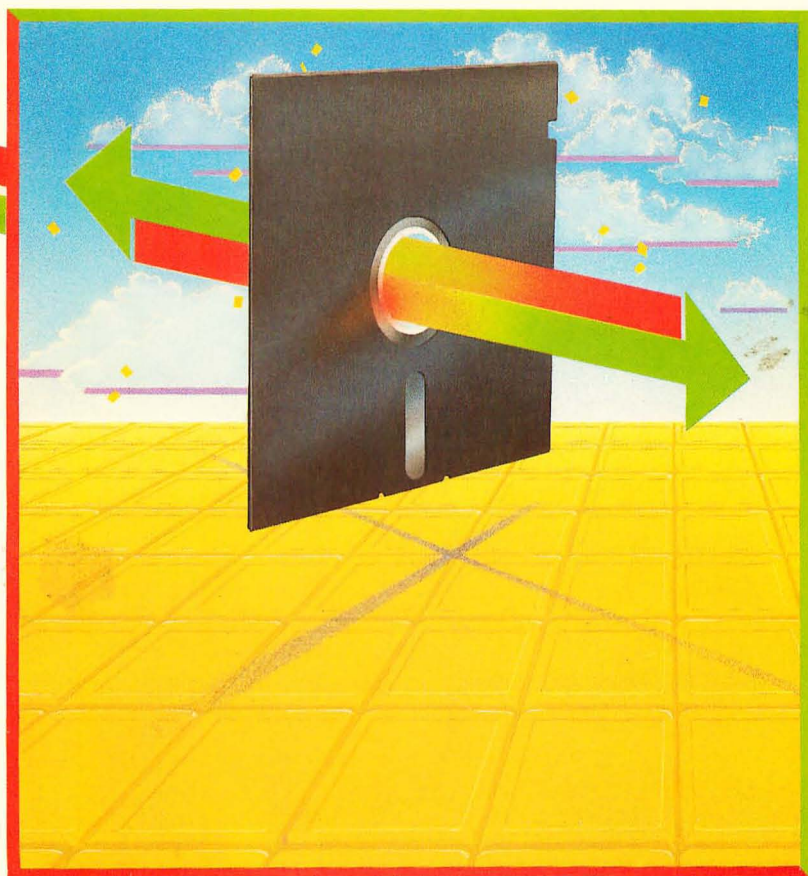


Computer Direct

312/382-5050

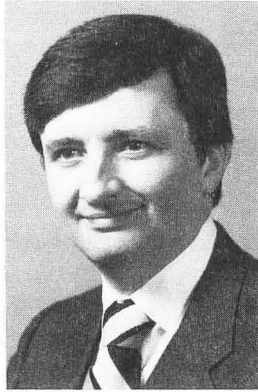
ATARI[®] BASIC Tutorial

Robert A. Peck





ATARI® BASIC Tutorial



Robert A. Peck has had over 12 years experience in the area of engineering product development and testing in hardware, software, firmware, and user documentation. He is a registered professional engineer in the State of Illinois, and is presently Manager of Technical Documentation for Amiga Corp., in Santa Clara, California. Previously, he has held managerial positions with Savin Information Systems and Memorex Corp., and was a Senior Project Engineer with Underwriters Laboratories, Inc., for nine years.

Mr. Peck received a bachelor's degree in electrical engineering from Marquette University in Milwaukee, Wisconsin (1969) and a master's degree in business administration from Northwestern University in Evanston, Illinois (1974). He has had numerous articles published in various trade magazines and has written operations manuals for several manufacturers, including ATARI[®], Inc.

ATARI[®] BASIC Tutorial

by

Robert A. Peck

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1983 by Howard W. Sams & Co., Inc.
Indianapolis, IN 46268

FIRST EDITION
FIRST PRINTING—1983

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-22066-0
Library of Congress Catalog Card Number: 83-50177

Edited by: *C. W. Moody*
Illustrated by: *R. E. Lund*

Printed in the United States of America.

PREFACE

Your ATARI®* Home Computer System is a tool you may use for many purposes. Among the uses are entertainment, home finance, letter writing, education, and many more.

As with many other tools, the more you know about it, the more useful it will become. It is really a general purpose tool, because it does what you tell it to do. This book will help you learn how to use the ATARI Home Computer by showing you, step by step, how the ATARI BASIC language can be used to control the computer.

In this book, you will find a large number of functioning examples that you can try on your machine. All of these examples have been tested carefully and will work on all of the ATARI Home Computers.

By reading the text and by trying the examples, you will be able to understand the way the various BASIC commands work. You may, if you wish, use some of the examples as a basis for your own programs.

In some cases, the text will start out with a short program that illustrates a certain point. Then additional program lines will

*ATARI® is a registered trademark of ATARI®, Inc., A Warner Communications Company.

PREFACE

be added to this functioning base program to perform additional functions. However, an attempt has been made throughout the book to keep the examples as short as possible. Those of you who may not have a data storage device available will therefore be able to follow along with the examples with little difficulty.

Throughout this book, the primary concentration of the program examples will be a user-interactive approach. In other words, the programs will be designed, as much as possible, to have you working *with* the computer, or for the computer to provide you with information, then asking what it should do next. This type of approach to programming is often called *user-friendly*. We hope you will find this approach helpful in the design of your own programs.

ROBERT A. PECK

CONTENTS

INTRODUCTION	9
--------------------	---

CHAPTER 1

INTERACTING WITH THE MACHINE, A FIRST APPROACH	11
--	----

Immediate Commands vs. Program Writing — The ATARI Screen Editor — Other Screen Editor Functions — Starting to Program — An Introduction to Variables — Starting to Make Decisions — Starting to Interact With the Machine — Compound Statements — Review of Chapter 1

CHAPTER 2

COMPUTERS COMPUTE	40
-------------------------	----

Other IF Tests — How to Make the Computer "Compute" — Operator Precedence — A New Value for a Variable — The INT Function — Other Functions Built-In to ATARI BASIC — Other Math Functions — Review of Chapter 2

CHAPTER 3

STRINGING ALONG	67
-----------------------	----

How ATARI BASIC Handles Strings — Other String Functions — String Comparison Features — The ASC and CHR\$ Functions — Review of Chapter 3

CONTENTS

CHAPTER 4

DESIGNING A PROGRAM	92
Planning a Program — Program Save and Load — Introduction to DOS — Making a Program Selection — Review of Chapter 4	

CHAPTER 5

PULLING DATA OUT OF DIFFERENT BAGS	116
The DIM Statement — FOR-NEXT Statements — Nesting Loops — More About the Accountant — The DATA Statement — The RESTORE Statement — The RND Function — Review of Chapter 5	

CHAPTER 6

MENU PLEASE	145
Accessing the Screen Editor from Within a Program — Absolute Cursor Positioning — The PEEK Statement — The POKE Statement — Game Controllers (Joysticks) for Menu Selection — How to Keep Control of the Machine During User Input — Review of Chapter 6	

CHAPTER 7

INTRODUCTION TO SUBROUTINES	177
Structure of a Subroutine Call — Screen Decoration Subroutine — More Uses for Subroutines — Review of Chapter 7	

CHAPTER 8

GETTING COLORFUL, GETTING NOISY	195
Graphics Capabilities — True Graphics — Sound Capabilities — Review of Chapter 8	
REFERENCES	216
INDEX	217

INTRODUCTION

Before starting anything with the computer, be sure everything is plugged together just as the manual states. The design of the ATARI® Home Computer Systems makes it difficult, if not impossible, to plug them together incorrectly, but check with the manual to be sure.

Before applying power to the ATARI® 400™, 800™, or 1200XL™ Home Computers, insert the ATARI BASIC cartridge into the cartridge slot. Because there is an interlock on the system for the cartridge, it is not really required to have the power off at this time, but it is just a good practice that you might want to adopt. (Some other computer manufacturers do not provide this protection and it will be necessary to turn off the power before inserting or removing an accessory. This prevents damage either to the accessory or to the computer.)

If you have a disk unit, be sure, before the disk is turned on, that a floppy disk is installed correctly. See your ATARI DOS Manual for the precautions to take with your disks. Once the DOS is loaded, you will be able to store your example programs on the disk for future reference.

If you have the ATARI® 810™ Disk Drive, or any other accessories such as the ATARI® 850™ Interface Module, turn on the power to these accessories before you turn on the power to the

computer. Then, when you turn on the computer, one of the first things it will do is to ask the accessories, "Who's out there?" and "Tell me how to handle your data." If the power to the accessory is off when the computer is turned on, it will not know the accessory is there and may not be able to use it later.

Now apply the power to the computer. You should see a solid blue screen (or gray if not on a color TV set). Then, after the accessories have had a chance to talk to the computer, you will see the computer tell you it is "READY." This means it is listening to the keyboard and wants to know what you want it to do next.

CAPS/LOWR KEY

ATARI BASIC only accepts commands in upper case (all capital letters). There is a key on the keyboard located between the **RETURN** and **SHIFT** keys labeled **CAPS/LOWR**. If you press this key, it performs a function similar to the unlock-shift function on a typewriter. All letters typed from then onward will be "small" letters. If you give the computer a command spelled in this form, it will not understand you. To return to the all capital letters mode, hold down the **SHIFT** key, then touch the **CAPS/LOWR** key. This has the same effect as a lock-shift function and puts the machine back in the correct state. When the power first comes on, or if you touch **SYSTEM RESET**, it will also go into the all capitals state.

Now turn to Chapter 1 to find out how to give the computer its instructions.

CHAPTER

1

Interacting With the Machine, a First Approach

Perhaps the most important thing the computer can do is provide the user with information. This information can come from many different sources, such as a data file on tape or disk, a series of computed numbers (answers to mathematical problems), or communications with other computers across telephone lines or another form of communications network.

In order to start some form of dialog with the machine, the computer must be told where to find the data you require, then it must be told what to do with it. In this, the first example you will try on the machine, you will be providing a complete instruction on a single line. The computer will follow that instruction, then again tell you it is READY for another.

The ATARI BASIC language has, as its commands, a set of English-like instructions that are each very closely related to the function it performs. For all of the BASIC commands, the computer can only obey the command if it is spelled exactly correctly. This is because there is only a limited amount of room in the cartridge and it therefore can have only a limited vocabulary.

For the program examples that will be shown in this book, after you type each line, press the **RETURN** key. This means you are

RETURNing control to the computer and it is to process the line of data input you have provided.

Try the following command as an example. Type it exactly as shown. If you make a typing mistake before you press **RETURN**, use the **DELETE BACK S** key to go back to correct your error. Type the following:

```
PRINT "HELLO"
```

and remember to press **RETURN**. The computer responds immediately by printing the word HELLO below your command line. Then it tells you that it is again READY for another command. In this PRINT command, the word HELLO is enclosed in quotes. You will use the quotes in the same way a writer uses them in a book, such as: He said, "THIS IS EXACTLY WHAT I WANT YOU TO PRINT." The computer takes everything literally and will print exactly what it finds between the quotation marks.

This is one example of the use of the PRINT command. You may use it to print many different things. It may also be used to print into a data file or to the line printer. For the early examples, however, printing to the screen will be the primary use of the command.

IMMEDIATE COMMANDS vs. PROGRAM WRITING

The command you have just given the computer caused it to perform the requested action immediately. This is therefore referred to as an *immediate* command. After the command has been performed, the computer will not remember that you gave the command. If you want to do it again, you have to type it in again or find another way to do the same thing.

There is a way you can make the computer remember a series of instructions. This is called *programming*. In programming, instead of the immediate mode of operation, you will be using the *deferred* mode. This means the computer will not execute the instructions immediately, but will wait for you to tell it to RUN the program.

As promised in the introduction, this book will attempt to minimize your typing. Therefore, it is time to introduce the functions

of the ATARI Screen Editor before programming is officially introduced.

THE ATARI SCREEN EDITOR

One of the functions built into your ATARI Home Computer allows you to modify the display on the screen if the computer is in the command mode (ready to accept either immediate or deferred commands). There is a rectangle on the screen which shows where the next character can be printed. This is called the *cursor*. When the **RETURN** key is pressed, if the cursor is within the same line in which a data display change has been made, the computer will obey the command contained on that line. This section will teach you how to use the Screen Editor.

Cursor Control

Look at the keyboard (Fig. 1-1). At the left side, there is a key labeled **CTRL** with a light-colored background. At the right side of the keyboard, there are four keys, each with the same light-colored background, and each with an arrow pointer on it (one points up, another down, another left, and the fourth points right). These keys are known as *cursor control* keys.

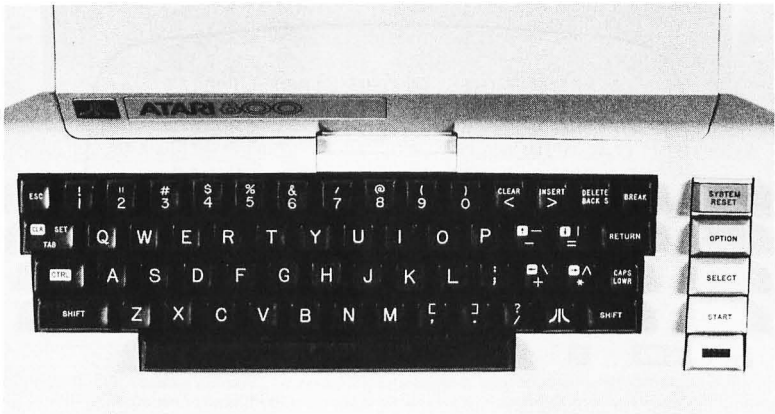


Fig. 1-1. The keyboard of the ATARI® 800.

The control key **CTRL** must be held down, just as a shift key is held down on a normal typewriter, to obtain the control functions that move the cursor. Just as with a shift key, if you touch the **CTRL** key alone, nothing happens; it just selects an alternate function for the key with which it is pressed.

Try it now. To move the cursor left, hold down the **CTRL** key, then touch the left-arrow key. To move it right, hold **CTRL** and touch the right-arrow key. Up and down are controlled by the up-arrow and down-arrow keys, respectively, with the **CTRL** key. (See Figs. 1-2 through 1-5.)

These keys have no permanent effect on the display. They only move the cursor to a different spot so you can change the screen contents using some other key or key combination. When the cursor is positioned over a letter or symbol on the screen, that item is displayed in reversed video (dark on light background) instead of normal video (light against dark background). This is done so you can still see the cursor, but also so you can still see which character is there. When the cursor is again moved, the original display is restored in the position it was before.

Notice that when the cursor leaves the screen at the top it reappears at the bottom, still traveling upward. Likewise, when it exits left or right it reappears at the opposite side, again moving in the original direction. You can take advantage of this to mini-

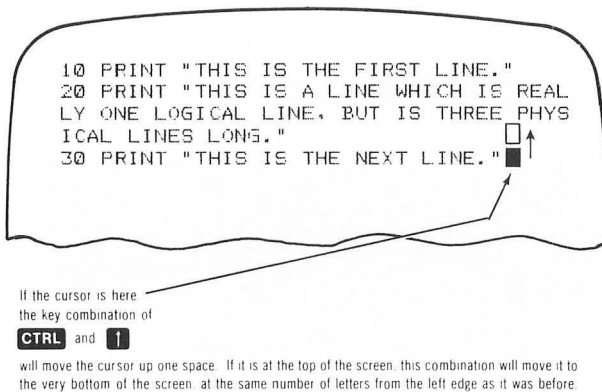


Fig. 1-2. Screen Editor function: cursor up.

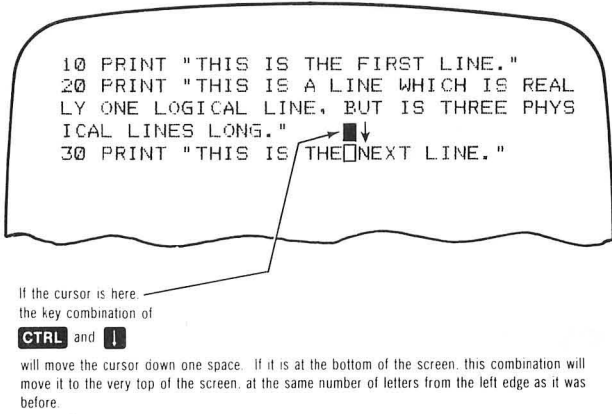


Fig. 1-3. Screen Editor function: cursor down.

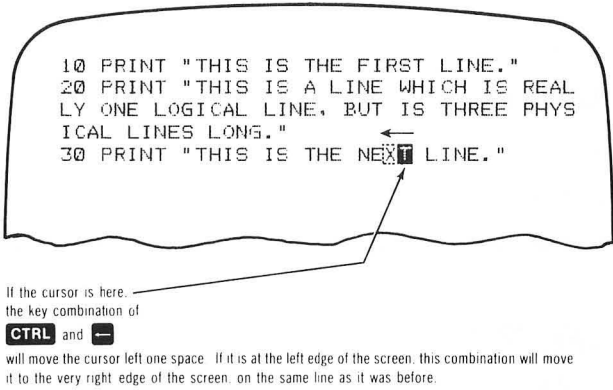


Fig. 1-4. Screen Editor function: cursor left.

mize your program editing time by moving the cursor in whichever direction it will travel the least to get to the item you want to change.

Notice that when you hold down **CTRL** and one of those keys for more than one-half second, the key begins to repeat. This function is common to all of the key combinations in the system. This is called the AUTOREPEAT function.

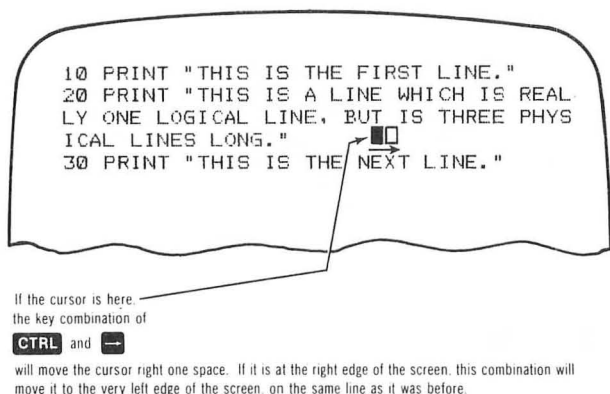


Fig. 1-5. Screen Editor function: cursor right.

Move the cursor onto the H of the line you originally typed (PRINT "HELLO"). In place of the word HELLO, type:

OTHER STUFF

The line should now read:

PRINT "OTHER STUFF"

If it does not, use the cursor controls to move to the incorrect items and retype them so that it looks exactly as shown. Leave the cursor somewhere within the same line you changed, then press **RETURN**. The computer responds by printing:

OTHER STUFF

in place of the word HELLO. This demonstrates that the computer has seen the new immediate command and has executed it.

How Long Is a Line?

When you type lines on your ATARI Home Computer, it is possible for you to type a line which is longer than the width of the screen. Each actual line on the screen is called a *physical line*. There are 24 physical lines of text that can be displayed on the screen.

If an ATARI BASIC line is longer than 38 characters, it will automatically continue on another physical line. This means that if you are typing a long line that contains more than 38 characters, if you want it to be all part of the same BASIC statement, do *not* press **RETURN** when you get to the right margin. Just keep typing and the cursor will return to the beginning of the next line. The Screen Editor will automatically accept the continuation on the following line as though it were one long continuous line.

If the ATARI BASIC line is longer than 76 characters, it will continue on the third physical line. The line may be composed of a maximum of 120 characters (114 if the normal left margin of two positions is used). ATARI BASIC will not accept any line with more than this number of characters and will “beep” at you when you near the end of the third line.

This long single line is called a *logical line*. As indicated in a previous paragraph, each logical line may be composed of up to three physical lines.

In the description of the Screen Editor, which follows in this chapter, you will see descriptions of inserting lines and deleting lines. All blank lines, if deleted, will make everything else on the screen move up one line.

If you place the cursor somewhere on the screen within a logical line composed of two or three physical lines, and execute the delete-line function, you will delete that *logical* line. This means that each time you do the delete-line function, one, two, or three lines will disappear from the screen.

The line-delete function does *not* cause ATARI BASIC to delete a program line from a program. Under the direct control of ATARI BASIC, this may only be done either by typing the line number to be deleted, then pressing **RETURN** (deletes that line only because it is replaced by a blank, and therefore is no longer remembered), or by typing NEW (which deletes all lines of a program).

Error Noticed After Hitting **RETURN**

What if you discover an error *after* you have hit the **RETURN** key? If it is a command that ATARI BASIC does not understand, it will repeat the line you just typed with the characters “ERROR—” in the first part of the line. Also, it will print a cursor character in

this error line over the first character that ATARI BASIC thinks does not belong.

To correct this error, move the cursor to the *original* line you typed. Move the cursor to the error and type in the correction. Use the character-insert or character-delete functions (see next section), if necessary, then hit **RETURN** again.

If the line is still not correct, ATARI BASIC will overwrite the old error line with a new one and show you where the new error is located. If the line is correct now, the cursor will simply be positioned on the next line (where the error line was printed). Do *not* press **RETURN** at this point! If you have been successful in correcting the error, move the cursor down, past the error-printed line, before making any more data entries.

ATARI BASIC uses the Screen Editor, which can read the line on which the cursor is located, to obtain its data input. If you press **RETURN** at this time, it would reread the error line, finding another error!

OTHER SCREEN EDITOR FUNCTIONS

Let's look at some other things you can do with the Screen Editor. (Refer to Figs. 1-6 through 1-9.)

Line Insert (Fig. 1-6)

Use the cursor control keys to move the cursor to the leading O of the newly printed line OTHER STUFF. Now hold down the **SHIFT** key, and press **INSERT**. The function **SHIFT INSERT** inserts a blank line at the cursor position. The words OTHER STUFF and everything else on the screen shift down by one line to make room for the new blank line.

Line Delete (Fig. 1-7)

Leave the cursor exactly where it was after you tried the line-insert function. Now hold down the **SHIFT** key and touch **DELETE**. The function **SHIFT DELETE** will remove the logical line on which the cursor is resting and move all other lines below it on the screen up one line. With this command you will have removed the blank line you inserted in the preceding example.

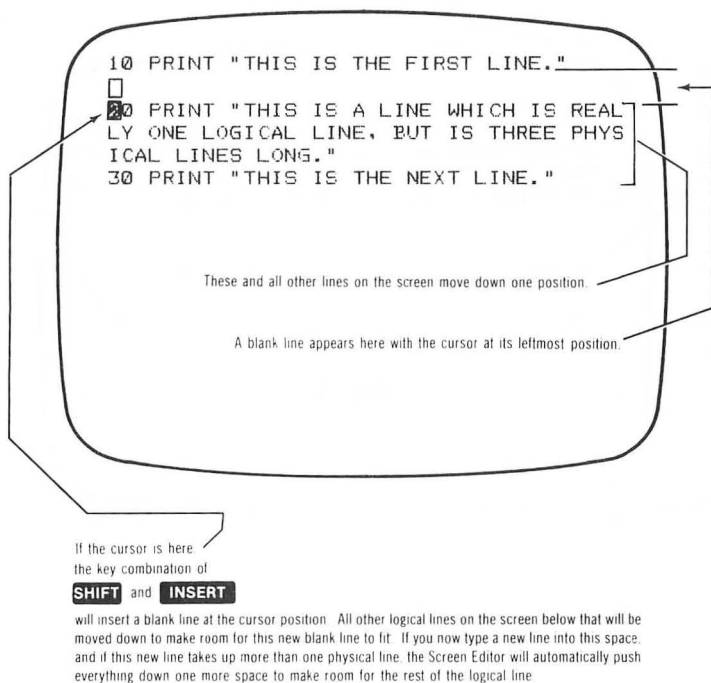


Fig. 1-6. Screen Editor function: line insert.

Character Insert (Fig. 1-8)

Move the cursor onto the O of the line you printed containing OTHER STUFF. Now hold down the **CTRL** key and press **INSERT**. The function **CTRL INSERT** takes all of the characters within the logical line where the cursor is sitting, moves them one character position to the right, and inserts a blank space where the cursor is located. The cursor doesn't move, so the next thing you could do is type a character in this blank space. This is why it is called a *character-insert* function.

Character Delete (Fig. 1-9)

Leave the cursor where it was and hold down the **CTRL** key again. Now touch the **DELETE** key. The **CTRL DELETE** function deletes the character on which the cursor is sitting and moves

ATARI BASIC TUTORIAL

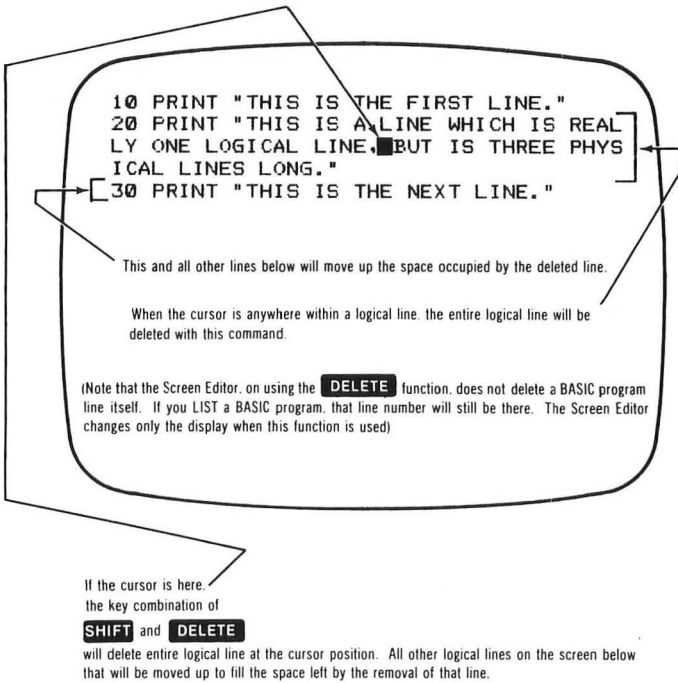


Fig. 1-7. Screen Editor function: line delete.

all of the other characters on that logical line one space to the left.

STARTING TO PROGRAM

Now that you have a way to move the cursor and have the computer accept a new version of your data, you can begin to program the machine.

The first step is to understand how the system tells whether a command line is an immediate command or a program line which it is to remember but not yet execute. The rule for ATARI BASIC is simple: *Any line which begins with a number between 0 and 32767 is treated as a program line. Any other type of line input is treated as an immediate command.*

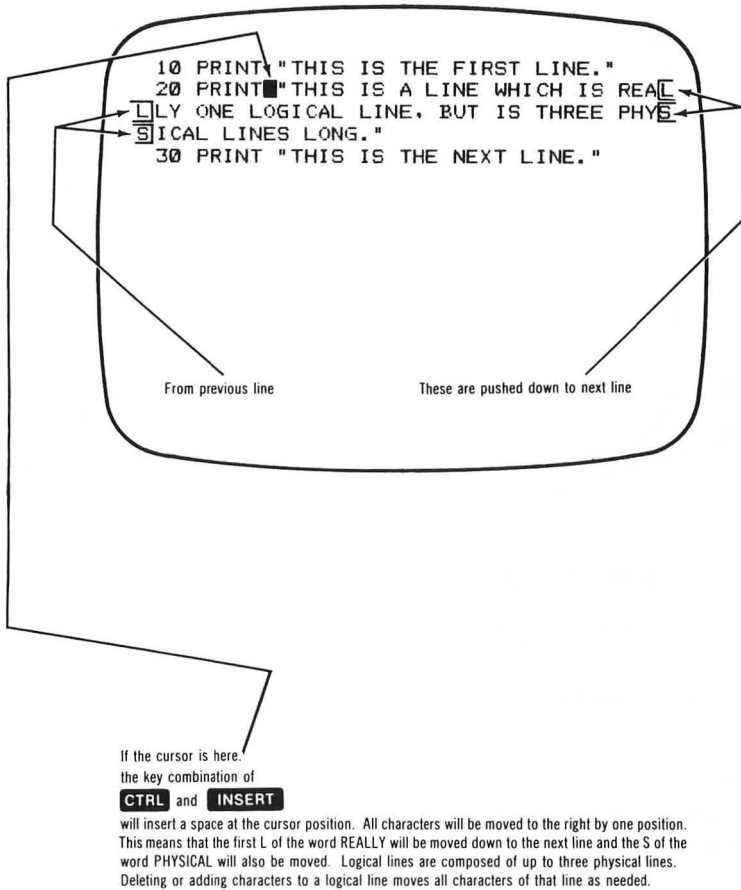


Fig. 1-8. Screen Editor function: character insert.

Let's use this fact to write the first line of your first program. Move the cursor up to the line on which you printed OTHER STUFF and position it over the O. Now use the line-delete function to erase the entire line.

If you now move the cursor up to the line on which you have the PRINT command and touch **RETURN** again, it will reprint OTHER STUFF on the line you just erased. However, if you use

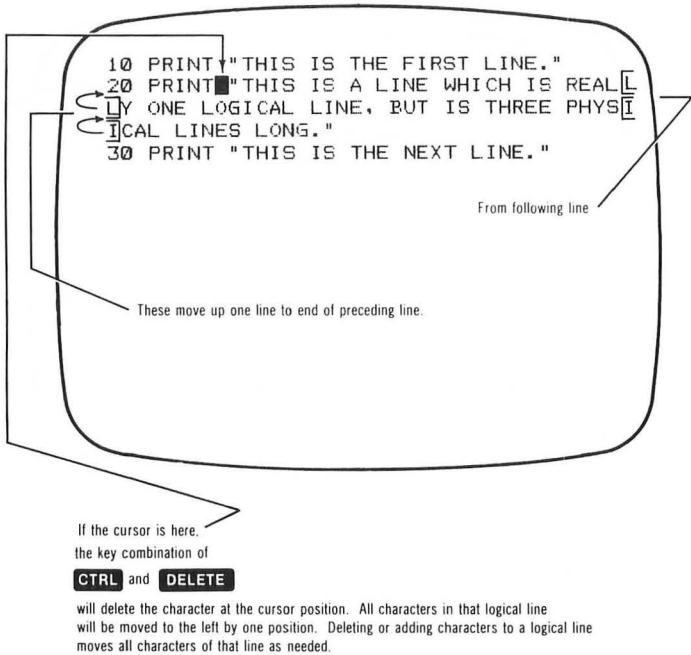


Fig. 1-9. Screen Editor function: character delete.

the instructions that follow, this command line will then become part of a computer program. To start, first move the cursor to the line where the machine printed OTHER STUFF, and use the line-delete function to erase it again. You will now use another of the ATARI Screen Editor commands to insert a character.

Move the cursor onto the P of the PRINT command line. Hold down the **CTRL** key, and touch the **INSERT** key twice. The entire line will be moved to the right by two spaces. In those spaces, type the digits 10 (making this line number 10). Now press **RETURN**.

Notice that this time the machine did *not* print OTHER STUFF on the line immediately below the command line. This is because the 10 placed in front of the command line told the machine to remember this line as part of a program.

Just to confirm that the machine will now remember this in-

struction, you will now use another function of the Screen Editor to clear the screen.

Clear Screen

Hold down either the **SHIFT** key or the **CTRL** key. Now press the **CLEAR** key. This Screen Editor function, **SHIFT CLEAR** or **CTRL CLEAR**, will erase all the contents of the screen and move the cursor to the upper left-hand corner. This is called the HOME position. Whenever this book refers to CLEAR SCREEN or HOME the cursor, this is the function you will do. Now type:

LIST

and touch **RETURN**. You should see the following:

```
10 PRINT "OTHER STUFF"  
READY
```

What you have just done is to list the contents of the program which the computer has in its memory. In later sections, you will be told how to add to a program, how to delete sections from it, and how to control the sequence of the program's functions. For now, however, let's tell the computer to execute this program. Type RUN and touch the **RETURN** key. The computer responds by printing:

```
OTHER STUFF  
READY
```

You have just RUN your first program!

When you typed the word RUN, it told the computer to go from command mode (listening to the keyboard for what to do next) into execute mode (reading its program for the commands to perform). The computer will stay in this execute mode until it reaches the end of your program or until it reaches an exit point (END point) which you have defined. Then it returns to the command mode again to wait for you to tell it what to do.

Programming in ATARI BASIC can be compared to making up a recipe which the computer is to follow. Continuing the comparison, if you would separate a recipe into a number of different steps, and write each step on a separate card, your recipe would

be very much like a computer program. Now imagine numbering each of the cards of your recipe. Place all recipe cards in the correct numerical order so that the lowest numbered card represents the first step, the next lowest is the next step, and so on to the highest numbered card.

When you follow this recipe in numerical order, you will be acting exactly like ATARI BASIC acts when it reads and executes a program. ATARI BASIC programs are executed by the machine performing the command at the lowest program line number, and continuing at the next lowest sequential line number, then the next, etc., proceeding to the end of the program. This line number sequence will continue unless you have placed some decision points within the program that will change the order of the command execution.

To demonstrate this sequential program execution, type in the following program lines, in the order shown:

```
30 PRINT "THEN LINE 30"  
10 PRINT "THIS PROGRAM EXECUTES LINE 10"  
40 PRINT "AND FINALLY LINE 40"  
20 PRINT "THEN LINE 20"
```

Remember to press **RETURN** to enter each of the lines. Now that the program has been entered, LIST it. Notice that the computer has taken all of the lines that you typed and placed them in numerical order. This is the order in which they will be executed. Whichever lines you add will be added to the program in the numerical sequence in which they belong, from lowest to highest number.

Going back to the recipe idea again, if you had your cards all in order, and someone suggested that step 5 could be changed to add a different ingredient, you might either erase and write over step 5 with a new version, or perhaps throw away the step 5 card entirely and write up a new one to be added to the recipe.

You will notice from the earlier example that the line 10 PRINT "OTHER STUFF" has now been replaced by the new line 10 which you just typed. This is because the computer is doing the same type of housekeeping that you would do for a recipe. It has replaced the old line 10 with the new one.

RUN this program and note the display. Again it demonstrates how the computer executes its program instructions if you don't specify any change to its normal sequential operation.

One of the ways you can change the order of command execution is to use the BASIC command called GOTO. The way this command appears when used by itself is as in the examples given here. Type in all five exactly as shown. This assumes that the program mentioned above is still in the machine, so these program lines will be added to it.

```
5 GOTO 30
35 GOTO 10
15 GOTO 40
45 GOTO 20
25 END
```

When you RUN this program, the command sequence will not be 10, 20, 30, 40 any more. It will instead be 5, 30, 35, 10, 15, 40, 45, 20, 25, and finally a return to the command level caused by the machine reading the END statement. The GOTOs have changed the order in which the machine has performed the instruction sequence.

This showed you a way you can directly define the sequence for the program statement execution. However, it does not do much. Later you will see some ways in which a GOTO can be used to save you some time in programming. For now, though, we will use the GOTO in combination with other ATARI BASIC statements.

Throughout the remainder of this book, you will be using many different program examples. Many of the examples given will be unrelated to the preceding example. Therefore, you will need to erase the old program from memory before trying to do the new program. This is easily accomplished with the ATARI BASIC keyword NEW.

When you type the word NEW, ATARI BASIC erases all references in its memory to the program that was there before. If the GOTO demo program is still present, LIST the program to the screen, then type NEW, then type LIST again. You will see that after typing NEW, the only response from the computer is READY.

This is because the program is gone. In the chapters that follow, you will be instructed to type the word `NEW` when the example to be shown does not relate to the preceding one.

Usually, when you are doing a project of some kind, you will find that there may be more than one way to perform the project. You would need some way to decide which pathway to take. The next section introduces one of the ways in which ATARI BASIC handles numbers. By using numbers, and by introducing another ATARI BASIC statement (the `IF` statement) immediately thereafter, you will be able to tell the computer to do something based on a decision it will make.

AN INTRODUCTION TO VARIABLES

If you want the computer to remember a number value and to tell you what that number is later when you ask for it, you must give the computer a name to associate with that number. This is similar to the way a person remembers something. As an example, if you have a friend named Ted and his age is 15, you may associate the words Ted and age, if they are used together, with the number 15. If you would express this so the computer could make the same association, it might read as with the following ATARI BASIC program line:

```
100 TEDSAGE = 15
```

or you can also say

```
100 THEAGEOFTED = 15
```

or abbreviate to

```
100 T = 15
```

The names you have given to this number are called *variable names*. By naming a variable, it directs ATARI BASIC to reserve space in memory. In this space, then, it will store any number you assign to that variable name. When you again ask the computer to tell you (or to use) the value of that variable, it will go to the same memory location each time to get the value to use. Likewise, if you tell it that the variable has a different value, that new value replaces the old value and will be used from then onward.

You can see how the computer keeps these values by typing the following immediate data entry lines:

```
ABC = 1
```

The computer responds:

```
READY
```

Now type:

```
PRINT ABC
```

This command says print the value of the variable called by the name ABC. The computer responds:

```
1  
READY
```

Now type:

```
PRINT B
```

telling the computer to print the value of the variable called B. The computer responds:

```
0  
READY
```

What happened here? You did not give any value to the variable name B. This meant there was no memory space assigned to hold a value named B. When ATARI BASIC searched the memory and did not find an assigned value for this variable, it assumed a value of zero. This value is what was printed. Now try:

```
PRINT ABC2
```

The computer again responds:

```
0  
READY
```

The variable ABC2 hasn't been defined either, so you get the same response. The reason you were asked to do this is to illustrate another point about ATARI BASIC. This is that *all characters* in a name are considered to be "significant." Each

variable name may be many characters long if you wish (limit about 110 characters for each name). This example shows that the variable ABC is different from ABC2.

If you assign two names to two variables, and if each name is 100 characters long, and the first 99 characters of each are identical, with the last different, ATARI BASIC will still be able to tell the difference and will store each in the correct memory location.

This is a situation that many other versions of BASIC do not provide. Most other BASICs only care about the first two characters. In other words, ATARI BASIC can easily tell the difference between the variable name TED and the variable name TEMPERATURE. However, other BASICs will treat both names as though they shared the same memory location or number storage space, which they would call location TE. This is mentioned here in case you will ever need to convert one of your ATARI BASIC programs to run on another machine.

Each of the variable names that you define must start with an alphabetic character, A–Z, and may contain up to 110 alphabetic characters or any of the numbers, 0–9. Examples of names which ATARI BASIC will accept are:

```
ABC1000
HHHHHHHHH
M$(1)
```

The last example is a special type of variable called a *string*. You will learn about strings in Chapter 3. The parentheses () shown in the last of the name examples are used to indicate an *array*. Arrays are covered in Chapter 5 of this book.

Arrays and strings were mentioned here because the names applied to the strings or arrays are part of the total number of names that ATARI BASIC can handle. You cannot define more than 128 different names in a single ATARI BASIC program. It will not accept any more and will say "ERROR 4 AT LINE XXX." XXX stands for the last line number that was read acceptably before you tried to insert the program line defining name number 129. ERROR 4 is the "too many variable names" error.

Now that you know what a variable is, you will be told how it can be used. When you ran the last demo program segment, you

printed the value of the variable using the statement PRINT ABC. This has been another way in which you can use the PRINT command. The previous way, if you recall, was to print a message to the screen. In that use, the message appeared in quotes within the command line.

Sometimes you will need to make up a line of information to the computer operator composed partially of your letter message, and a number that has been generated by the computer program. In this case, you will take the two forms of the PRINT command as you now know them and combine them to produce a single line of printed output. You can do this by using a semicolon (;) at the end of the first (or the first of many) PRINT statements. When the computer executes this line and sees the semicolon, it will keep the printing cursor (current print position) exactly where it was after printing the final character of the line you specified. Therefore, the next character to be printed will be placed in the next available position to the right of the character last printed.

If you do not use the semicolon, any line you ask the computer to print will cause a carriage return (print cursor will move to left-hand margin) and a line feed (print cursor will move down to next lower line from that last printed). An example program for showing how the semicolon is used is shown below. In this example, you will type NEW first to erase any old program lines which might have been in the memory before you started this section.

```
NEW
10 PRINT
20 A = 1
30 PRINT "THE VALUE OF VARIABLE A IS: ";
40 PRINT A
```

When you RUN this program, the computer executes line 10 first. This line says to PRINT, but it does not specify what is to be printed. Therefore, the computer prints nothing. But there is no semicolon present there, so the computer performs a carriage return/line feed (goes to leftmost position of next line) resulting in a blank line appearing on the screen between the command RUN and the message output line.

Line 20 defines the value of the variable called A. Line 30 prints

the first part of the message and, because of the semicolon, keeps the print cursor immediately following the last space in the message. This allows line 40 to place the value of A on the same line, resulting in the display output:

```
THE VALUE OF VARIABLE A IS: 1
READY
```

To make things a little more compact, you can combine the PRINT statements from lines 30 and 40 onto a single line. First, on a separate line type:

```
40
```

then **RETURN**. This will erase line 40. ATARI BASIC sees line 40 as blank because of the new data entered. A blank line does not have to be remembered. Now type:

```
LIST 30
```

then **RETURN**. The computer responds:

```
30 PRINT "THE VALUE OF VARIABLE A IS: ";
```

Use the cursor controls (reminder, **CTRL** arrow key combinations) to move the cursor to the space just after the semicolon. Then type an A at that spot, making the line read:

```
30 PRINT "THE VALUE OF VARIABLE A IS: ";A
```

With the cursor positioned within this line somewhere, press **RETURN**. This causes ATARI BASIC to accept this line as input, replacing the original line 30.

Now RUN this modified program. You will see that the printed result is exactly the same. Therefore, the semicolon, in ATARI BASIC, can be used to combine different types of data output onto a single line, and may be used within a single PRINT command if desired.

STARTING TO MAKE DECISIONS

Going back to the recipe example again, it might be written with decision points throughout the recipe. Examples that might be found in a typical recipe are:

- If the batter is too thin, then add a little flour;
- If the oven is not up to 400°, then don't start baking the item yet;
- If the item becomes brown on top, then test it for doneness; and so forth.

All of these different decisions may be expressed in the form: IF (some condition is true) THEN (do something) OR ELSE (continue doing the next thing).

ATARI BASIC can make decisions based on the same type of logic. In fact, the BASIC commands that are used for this decision making are called IF and THEN. There is no equivalent to the OR ELSE part of the example in ATARI BASIC. This condition is fulfilled by executing the next sequential command in case the IF condition turns out to be false.

The condition that can be tested may be as simple as a single number value or it may have many conditions combined into a single test. These multiple conditions are covered elsewhere in this book and in the Advanced ATARI BASIC Tutorial. For now, however, you will just use very simple tests for the IF statement, at least until you are adjusted to the way in which the commands function.

The IF statement must always have a THEN associated with it, contained within the same program line. The ATARI BASIC Editor will not accept a program line unless it contains the correct associated components. An IF statement alone is incomplete without the THEN. The computer will first make the test specified after the word IF and before the word THEN.

If the condition results in a false indication, or if the result of an arithmetic test of some kind is zero, BASIC will go directly to the next sequential statement. If the condition evaluates as true, or if an arithmetic test results in a nonzero value, whatever BASIC statements that are contained within the command line following the THEN are executed. The following example will show you one way the IF-THEN statement can be used. To try the program, just type the lines exactly as shown. As before, if you make any mistakes, use the Screen Editor cursor-move keys and functions to make the corrections.

```

NEW
10 N=3
20 PRINT "THE VALUE OF N IS: ";
30 IF N=1 THEN PRINT "ONE."
40 IF N=2 THEN PRINT "TWO."
50 IF N=3 THEN PRINT "THREE."
60 IF N=4 THEN PRINT "FOUR."
70 END

```

All this program does is translate the number value of the variable N into a written value. Program line 10 sets the value of N. Line 20 prints the first part of the data output line. It keeps the cursor one space past the colon because the PRINT statement was used with the semicolon. (Recall that this prevents the carriage return/line feed from happening so you can make up a line using different PRINT statements.)

Program lines 30 through 60 are the IF-THEN program statements. In words, each spells out what it is to do. If you run this program, you will notice that only one of the PRINT commands actually happens. This is because the value of N is exactly 3 in the example shown. Therefore only the PRINT command associated with the word "THREE" will be performed.

Another thing you should notice here is how very similar each of the program lines 30 through 60 are to each other. Did you remember to try to use the Screen Editor to save yourself some time in typing them in? Just to show you how you might have done this, you could have typed in:

```
30 IF N=1 THEN PRINT "ONE."
```

Then hit **RETURN**, hold down the **CTRL** and up-arrow keys to move to the 3 of line 30, then type a 4 in place of the 3, hold down the **CTRL** and right-arrow keys to move to the 1 and replace it with a 2, move to the O of the word ONE, and then type TWO." in place of the original word, then hit **RETURN** . . . and so on until you had entered the entire program from 30 to 60. It would have saved you some typing, right?

So now you know, if you see a program example which uses many lines that are similar to each other, you can use the Screen

Editor to save you some work. This could happen when you are typing a large program from a magazine or other printed source. Let's say that line 1500 is very similar to a line 30 you typed earlier and you want to use the Screen Editor to save some work. The LIST command can be used to help here. For example you can tell the machine to:

LIST 30

and it would respond with a listing of line 30 of the program. You could then use the Screen Editor to remake the printed listing of this line into the right form for line 1500, and press **RETURN** to enter this new program line. The original line 30 would be unchanged because you had not asked the machine to enter any changes there.

If you wanted to use the Screen Editor on all lines from 30 through 60, the LIST command also allows you to specify a range of lines. This form of the LIST command would look like this:

LIST 30,60

and would list all lines between and including 30 and 60. Since longer programs will not all fit into a single screen display, by using this form of the command you can look at pieces of your program at a time. These pieces can be as small as one single line if you specify one line number, or as large as the entire program if no line number range is given. Try each of the forms of the LIST command now on the program you just typed in:

LIST 30 (lists only line 30)

LIST 30,60 (lists lines 30–60 inclusive)

LIST (lists the whole program)

Now RUN this program. The output should read:

THE VALUE OF N IS: THREE.

Then it will print:

READY

Just to demonstrate that each of the values will cause the correct output, once the machine says READY, change program line 10 by typing:

```
10 N=2  
RUN
```

or,

```
10 N=1  
RUN
```

or,

```
10 N=4  
RUN
```

In each case, the computer will print the correct value if the program was typed as shown. Now try typing:

```
10 N=5
```

then RUN this version. It prints:

```
THE VALUE OF N IS:
```

but does not complete the statement. This is because all of the conditions of the IF statements were found to be false. Therefore, no value was filled into the statement.

STARTING TO INTERACT WITH THE MACHINE

This book promised that you would be able to have the machine work *with* you to get a job done. In the examples shown so far, the machine always came back to you and said READY. Then you had to change the instructions you gave it, and tell it to RUN again. Now it is time to show you how to keep the machine in the RUN mode; in other words, to stay within the program you write rather than the master program which only says READY all the time.

Using the same program that was just discussed, make the following changes. (Be sure to LIST the program to make sure the changes appear as shown.) When you do a LIST, do not be concerned if the spacings of the commands are not exactly as

you entered them. ATARI BASIC ignores all extra spaces except those that appear between a pair of quote marks. Those are always kept as you entered them. (Remember to use the Screen Editor to save typing.)

```
10 PRINT "PLEASE INPUT VARIABLE N"  
15 INPUT N  
30 IF N = 1 THEN GOTO 300  
40 IF N = 2 THEN GOTO 400  
50 IF N = 3 THEN GOTO 500  
60 IF N = 4 THEN GOTO 600  
70 PRINT N  
75 GOTO 10  
300 PRINT "ONE."  
310 GOTO 10  
400 PRINT "TWO."  
410 GOTO 10  
500 PRINT "THREE."  
510 GOTO 10  
600 PRINT "FOUR."  
610 GOTO 10
```

You have added a new type of statement to the program. This is in line 15. The INPUT statement, in ATARI BASIC, is used in a program when you want the machine to pause and ask for a number value to be typed at the keyboard. Once the number value has been typed, the computer operator must hit the **RETURN** key to return control to the machine. The cursor must be on the same line as the number that has been input to allow the machine to see the number correctly.

LIST your program so you can look at it. What does this program do? If you follow the statements closely, you will see the way it will look at the line numbers. Line 10 asks you for a numeric value. Once you type it in, line 20 prints the same lead-in message as before. Lines 30, 40, 50, and 60 each try to see if N is exactly the value that is in the IF-THEN test. If it matches, the English equivalent is printed. Then the program continues again asking for another value. This is caused by the GOTO statements that we added to change the sequence of the program.

Now RUN this program. The machine will ask you what value of N is to be used and will keep asking you for a new value in an endless loop. After each new value, it will see if it can translate the value to words. If it cannot do so, it will print the number value instead.

This program illustrates another idea about programming. That is, there should always be some way the machine should reply to a user request. Whether the typed data is good or bad, there should be some way the machine is supposed to respond in each case. If you don't tell it how to respond, you may give an incorrect reply to a question. (If all replies to a question are tested and found false, there should be a response marked to be used for the all-false condition.)

RUN the program and enter some different numbers, including the numbers 1, 2, 3, and 4. Watch the results it prints. Try the numbers 1000, -10, 9.99999999E+97, and 1.0E-98. These last two numbers are entered in what is called *scientific notation*.

The large number, 9.99999999E+97, means a number almost equal to 10 followed by 97 zeros. This is the largest number that can be handled by ATARI BASIC. Notice when you entered this number that it printed all of the digits when line 70 was executed. This indicates that ATARI BASIC can remember nine digits for any number, even a very large one. If you enter a number greater than this, you will exit the program to the command mode indicating an ERROR 8, saying the input data is not acceptable. You will also see an ERROR 8 if you try to give the machine an alphabetic input (ABC, etc.; any characters not strictly numeric or scientific notation as shown in the examples).

The small number, 1.0E-98, means a number 0.00000.....01, where the number of zeros between the decimal point and the 1 is 97. This is the smallest number ATARI BASIC can handle. If you enter a smaller number, then line 70 will print the value zero (0).

If you produce an ERROR 8, to continue to try to enter numbers you must tell the machine to RUN again. The question mark (?) on the screen is called a *prompt*. It is present to tell you that ATARI BASIC is waiting for your input from an INPUT statement.

Since, as you will see in later sections of this book, the INPUT

statement can be used for alphabetic as well as numeric inputs, it is good practice to tell your program user exactly what type of data is expected. If the system is expecting a number and gets a letter, you will get an error that will interrupt the program. There is a way to handle this error, though, called a TRAP statement. This will be introduced later in this book.

You would probably like to go on to something else now. But the machine keeps on asking you for more input. To do anything else, you will have to return to command mode from the execute mode. To do this, simply touch the **BREAK** key. The **BREAK** key will stop a running program, telling you "STOPPED AT LINE XXX." Here, XXX represents the line number the machine was about to execute when it saw you press **BREAK**.

COMPOUND STATEMENTS

The program you just performed did do some user question and answer work, but it involved a bit more typing than necessary. There is another feature in ATARI BASIC that you can use to make this program a bit shorter. This feature is called the *compound* statement.

A compound statement is a set of BASIC statements all with the same line number and all within the same "logical line." Each ATARI BASIC statement is separated from the preceding statement by a colon (:).

In ATARI BASIC, each of the "logical lines" takes up anywhere from one to three actual lines on the screen. Since each line is normally 38 characters wide, the maximum normal BASIC statement can be 3 times 38 characters, or 114 characters total.

Let's look at how using some compound statements would have made the program shorter (and less work). Use the Screen Editor to make changes to lines 30 through 60 as follows:

```
LIST 30,60
```

Then use the **CTRL** and arrow keys to move to the correct spot on each line to make the lines read as follows. Remember, you *must* touch **RETURN** after making the change on each line, *with the cursor within that line*, for ATARI BASIC to accept the

new changes. Since each **RETURN** keypress moves the cursor down one line, it will be easiest for you to make these changes if you start the change on line 30. Here is how these lines must read after the changes:

```
30 IF N = 1 THEN PRINT "ONE." :GOTO 10
40 IF N = 2 THEN PRINT "TWO." :GOTO 10
50 IF N = 3 THEN PRINT "THREE." :GOTO 10
60 IF N = 4 THEN PRINT "FOUR." :GOTO 10
```

The new program requires only the line numbers 10, 15, 20, 30, 40, 50, 60, 70, and 75. Since lines 300–610 are not being used, you could type:

```
300 RETURN
310 RETURN
```

and so forth to delete these unused lines. But for now, just leave them in.

RUN this new version of the program. The results are the same as before. Use the **BREAK** key to return to the command mode.

As you can see from this example, the other way does require much more typing and much more space. This other way may be needed if you need each IF-THEN test to perform many different things, and when the total of those things it must do will not fit in a compound statement on a logical line. (Remember that everything within the logical line following the word THEN is part of what will be executed if the IF statement has a "true" result.)

An example of a long compound statement is as follows:

```
200 IF N = 1 THEN B = 100:PRINT "THE NEW VALUE
OF B IS";B:PRINT "AND THE VALUE OF N IS";N
```

Notice that all of the BASIC statements from which the compound statement is made are complete and correct (each could have been used separately in a statement having a separate line number).

Because of the rules for the IF-THEN combination, all of the statements following the THEN will be executed, in the order given, if the IF condition is true. None of those statements will be executed if it is false.

REVIEW OF CHAPTER 1

Thus far, you have learned that:

1. Capital letters must be used for the ATARI BASIC commands.
2. Immediate commands do not have line numbers, program lines do have line numbers.
3. The ATARI Screen Editor can be used to save some work in a programming job. The **SHIFT**, **CTRL**, **DELETE**, **INSERT** and arrow keys are used to call the Screen Editor functions.
4. Line numbers in an ATARI BASIC program specify the normal sequence of execution of the program from lowest number to the highest (0–32767). The program sequence may be modified, in one way, by a GOTO statement.
5. A variable is a named memory area used to store a number value. There are two different types of variables, numbers and strings. A number variable can have a value anywhere from $1.0E-98$ to $9.99999999E+97$. ATARI BASIC saves nine significant digits.
6. A program can be LISTed anytime while in the command mode. You can LIST either one selected line number, a range of line numbers separated by a comma, or just say LIST (with no line number range) to list the entire program.
7. A program can be RUN once it is entered. To exit the program, if you have no other exit method installed, the **BREAK** key can be used.
8. Compound statements can be used to save typing (and time, as will be seen in later chapters).

If you are not familiar with one or more of the preceding review items, we suggest you consult the index and go back to reread the section which covers that subject. Then you may proceed to the next chapter.

Computers Compute

In this chapter, you will learn more about the test conditions for the IF statement introduced in Chapter 1. You will also learn how you can use the computer as a number processor. The ATARI Home Computer is, after all, performing most of its operations based on some mathematical function. You will therefore learn how to use it in this same way.

OTHER IF TESTS

In Chapter 1, you were shown an example where the IF statement tested an "equals" condition. The example was:

```
30 IF N = 1 THEN PRINT "ONE."
```

The ATARI BASIC IF statement can test other conditions also. For example, it can test if N is greater than 1, if N is less than 1, or if N is not equal to 1. These particular IF tests can be written in ATARI BASIC as follows: In ATARI BASIC, a right-arrow bracket (>) represents the "greater-than" condition. A left-arrow bracket (<) represents the "less-than" condition.

Each arrow bracket looks like the item it represents because whichever variable is at the "small end" of the arrow is being

tested to see if it is smaller than the item at the "large end" (open part of the bracket) of the arrow bracket. Therefore, an example of a test for a number being greater than 1 looks like:

```
14 IF N > 1 THEN PRINT "IT IS GREATER THAN 1."
```

And an example of a test for a number being less than 1 looks like:

```
18 IF N < 1 THEN PRINT "IT IS LESS THAN 1."
```

The conditional test for "not equal" must mean that one variable is expected to be either greater than or less than the other variable. In ATARI BASIC, this condition test is represented by a combination of the symbols for each test (greater than/less than) and appears as follows:

```
<>
```

When it is used in an ATARI BASIC statement, the statement, if N is not equal to 1, print something, looks like:

```
30 IF N <> 1 THEN PRINT "IT IS NOT EQUAL TO 1."
```

There are two other tests the IF statement can make. These are used to test if a number is greater than or equal to another number, or if a number is less than or equal to another. Again, the symbols for each are a combination of the symbols for the individual test conditions.

To test whether a number is greater than or equal to another, this symbol is used:

```
> =
```

An example of this is as follows:

```
35 IF N > = 1 THEN PRINT "N IS GREATER THAN OR EQUAL TO 1."
```

To test whether a number is less than or equal to another, this symbol is used:

```
< =
```

An example of this is as follows:

```
40 IF N < = 1 THEN PRINT "N IS LESS THAN OR EQUAL TO 1."
```

You might have wondered why these conditional IF statements are covered in a chapter called "Computers Compute." Well, the way the computer is able to tell if the conditional test is true is to perform a subtraction, and this is computing. It subtracts the right side of the comparison (in each preceding case, the number 1) from the left side of the comparison. Then the condition test is made on the result.

For example, the equals comparison says the IF statement is true if the result of the subtract is equal to zero. The less than or equal comparison says the IF statement is true if the result of the subtract is less than or equal to zero, and so forth.

When ATARI BASIC completes its evaluation of the comparison statement, it comes up with one of two possible values. A condition true produces a nonzero result. A condition false is assigned to a zero result.

In some cases, you would need to combine various tests to see if a number was in a certain range or if two numbers have specific values. A number range test could be written as:

```
40 IF N > 10 THEN IF N < 20 THEN PRINT "N IS OK RANGE"
```

Notice the construction of this statement. The first THEN is the companion to the first IF. When the first IF test is found not to be true, the second IF is not even executed because of the IF-THEN rules. ATARI BASIC would just have gone on to the next sequential statement. However, when the first IF is true, then the second IF is tested. When it is true also, the PRINT statement takes place. (The PRINT statement is the completion of the second THEN for the second IF.) This is a bit awkward. ATARI BASIC does, however, provide another way to combine these two tests. This is with a connecting statement called AND. If you used the AND statement, the preceding test would look like this:

```
40 IF N > 10 AND N < 20 THEN PRINT "N IS OK RANGE"
```

What this says is that *both* the test $N > 10$ must be true and the test $N < 20$ must be true, in order for the combination of the two to be true. If *either* is false, the combination test is false and the condition following THEN will not be executed. AND can only be used within an IF test.

Another combination test you might want to do is to see if one condition or another condition is true. ATARI BASIC provides this capability also. An example is shown here:

```
50 IF N < 10 OR N > 20 THEN PRINT "N IS OUT OF RANGE."
```

This is testing the same condition as in the previous example for AND, but it states it from another angle.

The combination conditional test is true if *either* of the condition tests is true. The keyword OR can only be used within an IF condition test.

Now that you have seen how the different options of the IF statement can be used, you may want to try an example program using this command. But before that, look at another command (the TRAP statement) that is going to be used as an error catcher. If you remember from the earlier program using the IF statement, it was possible to cause an error by giving the program some data that was not strictly a number. For example, if it asked you to input a number, and you typed XYZ then **RETURN**, it would exit the program with an ERROR 8. This would say that the input was not as it expected. A quick example of the use of a TRAP is shown here. Try it. These program lines will be used in the next couple of examples also.

```
NEW
10 TRAP 200
20 PRINT "PLEASE TYPE A NUMBER VALUE"
30 INPUT N
40 PRINT "THANK YOU"
100 GOTO 10
200 PRINT "SORRY, THAT DID NOT LOOK RIGHT!"
210 GOTO 10
```

RUN this program and see what happens when you try to enter either an alphabetic input or just a **RETURN** with no number entered. Instead of giving you an ERROR 8, the machine now tells you that you didn't enter a number correctly. Now the only way you will be able to exit the program is to touch the **BREAK** key or the **SYSTEM RESET**, because you have used the pro-

gram itself to catch the errors and to handle them. Still using this program, change line 210 to read:

```
210 GOTO 20
```

and RUN the program again. Now give it bad data twice. (Just enter XYZ **RETURN**, then just **RETURN**.) The first time you do it, it gives you the right message. The second time, though, it gives an ERROR 8 and leaves the program execute mode. ATARI BASIC requires that the TRAP be reset each time it is used. In this mode, it is much like a mousetrap. It must be freshly set each time it is to be ready for another use.

Add the following lines to the program. Be sure to change line 210 back to the original reading (210 GOTO 10). The program then will also illustrate some combination IF tests.

```
50 IF N > 0 THEN PRINT "THIS IS A POSITIVE NUMBER"
60 IF N > 100 AND N < 1000 THEN PRINT "IT IS BETWEEN 100
AND 1000"
70 IF N >= 1000 THEN PRINT "IT IS 1000 OR GREATER"
80 IF N = 0 THEN PRINT "THIS NUMBER IS ZERO"
90 IF N < 0 THEN PRINT "THIS IS A NEGATIVE NUMBER"
100 IF N <> 999 THEN GOTO 10
110 PRINT "IT WAS 999, EXIT TO COMMAND MODE"
999 END
```

This program will test the value of N that you enter, and print only those things about N that are true. If you wish, you may add some more tests and PRINT statements to this program which would test the size of a number less than zero. This program also uses the IF test to allow an exit to the command mode by testing if N is equal to 999 (tested in line 100). If it is equal to 999, then the IF test in line 100 fails (the not-equal test is false) and the program exits.

RUN the program and try some values of N. To return to command mode, type 999 then **RETURN**.

HOW TO MAKE THE COMPUTER "COMPUTE"

Your ATARI Home Computer can perform arithmetic and many different kinds of math functions. These functions will be dis-

cussed later in this chapter. First, though, you must learn how to write a math statement that ATARI BASIC will understand.

Normally, when people think about arithmetic problems, they would often write them, for example, as:

$$A + B - C = D$$

and then perform the operation in the order in which it is written. For all arithmetic statements, ATARI BASIC requires that the answer variable must be shown first. This means that the example must be written (if shown as a program line) as:

$$100 \quad D = A + B - C$$

ATARI BASIC wants you to tell it where the answer is to be stored, then to tell it how to calculate the answer. That is why the variable D is shown first.

The arithmetic operations ATARI BASIC can do using a single character as the operation indicator are as follows:

- Addition, for which you use the plus sign (+).
- Subtraction, for which you use the minus sign (-).
- Multiplication, for which you use the asterisk (*).
- Division, for which you use the slash mark (/).
- Exponentiation, for which you use the caret symbol (^).

Here are some examples of how an arithmetic problem might normally be written, and how it must be written for ATARI BASIC:

Addition:

$$\begin{array}{r} 5 \\ +3 \\ \hline 8 \end{array}$$

10 A = 5

20 B = 3

30 ANSWER = A + B

40 PRINT ANSWER

Or, if you want to do it another way (not use 10,20):

30 ANSWER = 5 + 3

40 PRINT ANSWER

Subtraction:

$$\begin{array}{r} 12 \\ - 5 \\ \hline 7 \end{array}$$

10 A = 12

20 B = 5

30 PRINT A - B

In this second example, you should notice that ATARI BASIC will print, as a number quantity, any value which is correctly specified. In this example, instead of assigning the result to a variable called ANSWER, you have just asked the machine to print whatever are the results, without saving them for any other use.

Multiplication:

$$\begin{array}{r} 6 \\ \times 7 \\ \hline 42 \end{array}$$

10 A = 6

20 B = 7

25 ANSWER = A * B

30 PRINT ANSWER

Or, if you wish:

30 PRINT A * B

Division:

$$\text{Divisor} \overline{\left| \begin{array}{l} \text{Answer} \\ \text{Dividend} \end{array} \right.} \text{ or } \text{Answer} = \frac{\text{Dividend}}{\text{Divisor}}$$

You can ask ATARI BASIC to do this in a statement, such as:

30 ANSWER = DIVIDEND / DIVISOR

Here, the direction of the slash mark makes it seem as though the item at the left is "over" the item at the right of the slash mark.

Exponentiation:

This is the arithmetic operation which is used to “raise a number to a power.” For example, 2 raised to the third power would be written this way:

$$2^3$$

and its value is $2 \times 2 \times 2 = 8$. You may have ATARI BASIC calculate this value this way:

```
100 ANSWER = 2 ^ 3:PRINT ANSWER
```

As with other types of commands, either numbers or variable names can be used in the arithmetic expressions. But the power to which the number is raised *must be an integer* (examples: 2, 3, -7), in other words, it must not have a fractional part (examples of powers which cannot be used: -2.12, 3.5). If you do not use an integer, your program will halt with an ERROR 3, a value error.

When you are using command mode, you may sometimes want to use your ATARI Home Computer as a calculator. Just tell it what numbers to calculate, and it will give you an immediate reply. For example, you can type:

```
PRINT 24 * 60 * 60
```

The machine will reply:

```
86400  
READY
```

This is a quick calculation of how many seconds there are in a day (24 hours in a day, times 60 minutes in an hour, times 60 seconds in a minute). Or you may try anything else of interest to you in this way.

OPERATOR PRECEDENCE

Each of the arithmetic symbols for addition, subtraction, and so forth are called *operators*. *Operator precedence* means the order of importance of each of the operators.

ATARI BASIC has rules about the order in which arithmetic

operations should be performed. It is possible for you to change the rules by using parentheses to tell ATARI BASIC exactly what to do first. The order of precedence for all arithmetic and logical operations is shown in Table 2-1. The precedence order shows which one is done first, before others on a lower level.

Table 2-1. Order of Precedence for Arithmetic Operations

Operator	Description	Meaning
()	Parentheses	Left and right parentheses tell ATARI BASIC that it should change the evaluation order. See Note 1.
	String Relationals	See Note 2.
-	Unary Minus	Refers to a negative number, such as -20.14. A unary minus is not a part of an arithmetic expression. Instead, it is closely tied to the number, indicating that it has a minus value.
^	Exponentiation	Raising a number to a power. This is done before any of the operators that follow.
* /	Multiplication and Division	These have equal precedence and are performed from left to right.
+ -	Addition and Subtraction	These also have equal precedence and are performed from left to right after multiplication and division.
	Number Relationals	See Note 3.
NOT	Logical NOT	Unary operator for a logical (true/false) operation. In other words, if variable V = 0, then NOT V has the logical value of 1 (NOT 0).
AND	Logical AND	See Note 4.
OR	Logical OR	See Note 4.

Note 1. Parentheses are used to change the order in which ATARI BASIC will evaluate an equation. For example, if you notice in the table, the multiply

operation is more important than the add operation. Let's look at an example:

$$D = A + B * C$$

ATARI BASIC would then evaluate this equation as B times C plus A, and store the results in variable D. It is because of this precedence that the multiply operation is done first. What if you wanted the add operation to be first? You must enclose the add operation in parentheses (), such as:

$$D = (A + B) * C$$

Whatever ATARI BASIC sees within parentheses will then be done before any other operation. This is because parentheses have the highest of all precedence.

Parentheses can also be used to group various operations, and may be "nested" inside of each other. An example of nesting is shown here:

$$E = (A * (B - C)) + D$$

a b cd

A left parenthesis adds a nest level, a right parenthesis subtracts one nest level. The small letters below the diagram show:

- (a) the start of nest level 1
- (b) the start of nest level 2
- (c) the end of nest level 2
- (d) the end of nest level 1

Level 2 is "inside" of level 1 and is, in this example, the innermost nest level. ATARI BASIC will perform all arithmetic it finds in the innermost nest level first, then the next innermost, the next, and so forth until it is outside all levels of parentheses. Inside of any single nest level, the other operator sequences will still be the same.

Note 2. String compare operations are shown in the next chapter.

Note 3. Number compare operators are those you saw discussed at the beginning of this chapter, namely:

$$> , < , = , >= , <= , <>$$

These comparison operators are not only used in IF statements, but they can also be used in an arithmetic statement. When they are used this way, the result of the use of a compare operator is either a 1 (representing the condition TRUE), or a 0 (representing the condition FALSE). Examples are:

$$X = A > B$$

where,

- X = 1 if A is greater than B
- X = 0 if A is less than B

$$Y = C <> D$$

where,

- Y = 1 if C is not equal to D
- Y = 0 if C is equal to D

Note 4. The logical operators, AND and OR, are used to combine logic tests into one single larger test. You already saw at the beginning of this chapter how the AND and the OR operators were used within an IF statement. They may also be used in an arithmetic statement in the same way. Here are a couple of examples.

$$G = D \text{ AND } E$$

If the absolute value (ignore the sign + or -) of the numbers D and E are *both* greater than or equal to one (1), then the value of G will be a 1. If the absolute value of *either* one is less than 1, the value of G will be a zero.

$$H = A \text{ OR } B$$

If the absolute value of *either* of the numbers A or B is greater than or equal to one (1), then the value of H will be a 1. Only if *both* A and B are less than 1 will H be a zero.

A NEW VALUE FOR A VARIABLE

When you define an arithmetic statement, such as

$$C = A + B$$

ATARI BASIC will perform the calculation which you define on the right side of the equals sign, no matter how complicated, then it will finally store the value in the variable named on the left side of the equals sign. Because of the order of calculations you may, if you wish, use the "old" value of the variable itself in the calculation. In fact, you will see this type of statement very often in programs, and it will appear like this:

$$D = D + 1$$

Where the new value of D is calculated to be the present value of D with one added to it.

Let's look at an example program that uses this type of arithmetic to do something useful. Type in the program exactly as shown. The NEW statement is shown, as usual, to tell the computer to erase all old program pieces which might still be lying in the memory.

```
NEW
10 D = 0
20 PRINT "A TABLE OF SQUARES":PRINT
30 PRINT "NUMBER",N-SQUARED"
```

```

40 D = D + 1
50 PRINT D,D*D
60 IF D <= 9 THEN GOTO 40

```

RUN this program. It will give you a table of squares for the numbers from 1 to and including 10. Line 10 sets the initial value of D (initializes D). Line 20 prints the heading for the table. The compound part of line 20 prints a blank line. Line 30 prints another header. Line 40 gives D a new value each time it is executed, using the old value as the starting point as explained above. Line 50 prints both the number and the square of the number (that number times itself). Line 60 tests the current value of D. The program will go to line 40 again until the value of D is greater than 9. By then, it will have completed the table from 1 to 10.

Line 50 also introduced a new ATARI BASIC control character in the PRINT statement. This is the comma. When you place a comma in a PRINT statement, it tells the computer to skip to the next print-tab stop on the screen.

The print-tab stops are not related to the Screen Editor tabs. Print tabs are fixed at screen locations 1, 11, 21, and 31 where column 1 represents the leftmost printing position on the screen. (The normal screen form leaves the leftmost 2 columns blank, starting printing in column 3 of the actual screen, with 38 actual printing columns available. Therefore the normal print-tab locations are at real columns 3, 13, 23, and 33 on the screen.)

By using the comma in the PRINT statement, you can see you have a neatly arranged display. In the chapter titled "Menu Please," you will see other ways of positioning the printing on the screen. Now make a change to line 60, so that it now will read:

```

60 IF D <= 9 THEN 40

```

Clear the screen and RUN the program again.

There is no difference in the way it runs now. This is because ATARI BASIC saw a number quantity after the THEN keyword. The rule here is that if there is nothing except a number quantity following a THEN, ATARI BASIC thinks that is a line number to which it must execute a GOTO. The GOTO is not needed. How-

ever, this number must be a "literal" number, such as 40 or 100 or 32500. It cannot be a variable if the GOTO keyword is deleted.

If the GOTO is left there, you may use it as part of a "computed GOTO." This would be used where the value following the GOTO is a variable. The value of the variable must be equal to one of the program line numbers so that it will really have a place to GOTO. Change the program to add line 5, and change line 60 as shown here:

```
5 M = 40
60 IF D <= 9 THEN GOTO M
```

Clear the screen and RUN the program again. There is no difference, because ATARI BASIC found a GOTO 40 when the value of M was evaluated at the IF test.

Before leaving this example program, make another change, this time in line 50. Change it to read:

```
50 PRINT D,D^2
```

Now you will be using the exponentiation function to calculate the squares of the numbers. RUN the program again. The answers look a little different, don't they?

Exponentiation uses a special arithmetic formula. Because ATARI BASIC can only save a total of 10 digits, we say it is "limited to 10-digit precision."

When the formula is done, there may be a need to save more than 10 digits sometimes (not only with this, but with any math function) to be sure the result is exact. Since only 10 digits can be saved, you may wish to do some "rounding" of the result before you print it. This depends on the type of problem you are doing, and has been shown here just to demonstrate that the answer you expect may not always be the one you will get because of what is called *round-off* error.

THE INT FUNCTION

How can you do the rounding? ATARI BASIC provides a function that will let you separate the whole numbers from the fractions. It is called the INT function. The INT function takes a number

value, and calculates the whole number part of that value. You use the INT function as follows:

$$N = \text{INT} (D)$$

where D is a variable which has some digits after the decimal point and you want N to be just the whole number itself, with nothing after the decimal point. The INT function automatically rounds *down* to the next nearest whole number. This means that the number 3.99999996 becomes 3 if you try the INT function. To round up to the next number, you would need to do something like this:

$$N = \text{INT} (D + 0.5)$$

As a matter of fact, let's try that in the example. Change line 50 again, this time to read:

```
50 PRINT D,INT( D ^ 2 + 0.5 )
```

RUN the program again. This time the numbers look OK. There is, though, another way you could take advantage of the rounding down which INT always does. Because of the rounding down, negative numbers get rounded "correctly," as an example shows here:

$$N = \text{INT} (-3.99)$$

This results in $N = -4$. So let's make one more change in line 50 of the demo program to use this fact:

```
50 PRINT D, - INT ( - ( D ^ 2 ) )
           d   c b a   ac
```

The preceding letters may help you to understand what is happening. The letters "a" mark the inner set of parentheses. This tells ATARI BASIC what function to do first (calculate the square of the number). The letter "b" points to the minus sign, saying that it must make the result negative. You need the parentheses here to change the order of operations. Since the unary minus is more important than exponentiation, without the parentheses ATARI BASIC would have performed it this way: $(-N)^2$, which would always result in a positive number.

The letters "c" mark the outermost level of parentheses, which enclose the number on which the INT function is supposed to operate. This makes the INT function work always on a negative number. Finally the "d" points to the final minus sign, saying turn the result of the INT function (which will be minus) into a plus number (since -1 times -1 equals $+1$).

RUN the program again. See that the results are still the same as before, it only used a different way of getting there.

Another Use for INT

Sometimes when you are doing a division problem, you would like to know the answer in terms of the quotient and the remainder. In ATARI BASIC, when you express a division problem, it would look like this:

```
40 RESULT = DIVIDEND / DIVISOR
```

If you want to express the result as QUOTIENT plus REMAINDER, you can use the INT function as shown in the following example. Notice the spelling that was used for "remainder." There is a reason for this. The word REM is a reserved word in ATARI BASIC. Everything ATARI BASIC sees following the letter combination REM will be treated as a "REMark" or comment. It is kept as part of the program, but anything on the logical line following the REM will be ignored when the statement is executed. The REM statements you will see in the programs that follow will be the method by which the programs will be "documented."

When you type an example program, you do not have to type in any of the REM lines or any of the comments following the REM if it occurs on a compound statement line. However it is good practice, when you design your own programs, to put in some REM statements here and there to tell yourself (and perhaps others who will use your program) what each section of the program is supposed to be doing.

Getting back to the division example, here is a program that will present division results in the form just mentioned:

```
10 TRAP 300
20 PRINT "SPECIFY WHOLE NUMBER DIVIDEND";INPUT DIVIDEND
25 REM YOU DONT HAVE TO TYPE THIS LINE OR ANY REM LINE.
```



```
30 REM IN LINE 20, THE SEMICOLON TELLS ATARI BASIC
35 REM NOT TO SPACE DOWN TO THE NEXT LINE. THIS
40 REM ALLOWS IT TO ACCEPT THE DIVIDEND ON THE SAME
45 REM LINE AS THE REQUEST ITSELF THE COLON IN
50 REM LINE 20 ALLOWS THE INPUT STATEMENT TO SHARE
55 REM THE PROGRAM LINE 20 WITH THE PRINT STATEMENT.
60 REM
70 PRINT "INPUT A WHOLE NUMBER DIVISOR ";:INPUT DIVISOR
80 PRINT:REM THIS PRINTS A BLANK LINE
90 RESULT = DIVIDEND / DIVISOR
100 WHOLEPARTOFRESULT = INT (RESULT)
105 REM YOU CAN ABBREVIATE ANY OF THE NAMES.
110 REM THE EXAMPLE USES THE LONG NAMES JUST
115 REM TO BE SURE YOU WILL UNDERSTAND WHAT THE
120 REM VARIABLES MEAN.
125 REM
130 FRACTIONPART = RESULT - WHOLEPARTOFRESULT
140 REM NOW TO GET THE REMAINDER, JUST MULTIPLY
145 REM THE FRACTION PART BY THE DIVISOR,
150 REM BUT ALSO MAKE SURE THIS RESULT IS A
155 REM WHOLE NUMBER.
160 REM
170 RMAINDER = INT ( FRACTIONPART * DIVISOR + 0.5 )
175 REM NOTICE THE SPELLING OF THE WORD ABOVE
180 REM SO ATARI BASIC DOESN'T IGNORE THE LINE.
185 REM ALSO USING THE 0.5 TO CAUSE ROUNDING UP
190 REM BECAUSE OF PRECEDENCE, THE MULTIPLY IS DONE
195 REM FIRST.
200 PRINT "THE DIVISION PROBLEM ";DIVIDEND;"/";DIVISOR
210 PRINT
220 PRINT "GIVES A RESULT OF ";WHOLEPARTOFRESULT
230 PRINT
240 PRINT "WITH A REMAINDER OF ";RMAINDER
250 PRINT
260 PRINT "TO EXIT, USE BREAK KEY":PRINT
270 GOTO 10
300 PRINT "THAT DID NOT LOOK RIGHT!"
310 PRINT "PLEASE TRY AGAIN":PRINT
320 GOTO 10
```

This program actually only consists of the following line numbers: 10, 20, 70, 80, 90, 100, 130, 170, and 200–320. The rest are all comments (remarks). It does what we started out to do, but something *is* missing. No, the error trap is there, but there is no check to see whether the input numbers, dividend or divisor, are whole numbers. If they are not whole numbers, you will get inaccurate arithmetic results. How can you be sure the user of the program does input whole numbers? You can do this by using the INT function again! See the following test which could be included in the program. Let's look at it as though it was a separate set of tests. Don't type in these lines. They are here just for discussion. In the section following this discussion, you will see a way to combine all of these tests into a single line!

In this example, the logic names are called PART1, PART2, and PART3. The logic test PART1 determines if the number has any fractional part or is a whole number. PART1 is true if it is a whole number. PART2 is called true if the number is nonzero. (You cannot divide by zero, and this test avoids an error.) PART3 is true if both PART1 and PART2 are true. If PART3 is not true, then there is an error and it should be trapped.

```

500 TRUE = 1
503 FALSE = 0
505 FRACTPART = N - INT (N)
510 PART1 = FALSE
512 PART2 = FALSE
514 PART3 = FALSE
520 IF FRACTPART = 0 THEN PART1 = TRUE
525 IF N <> 0 THEN PART 2 = TRUE
530 IF PART1 = TRUE AND PART2 = TRUE THEN PART3 = TRUE
535 REM LINE 530 COULD ALSO BE WRITTEN AS:
540 REM IF PART1 AND PART2 THEN PART3 = TRUE
550 IF PART3 = FALSE THEN GOTO 400
560 REM 550 COULD ALSO BE WRITTEN AS
570 REM IF NOT PART3 THEN GOTO 400
580 REM BECAUSE IF PART3 WAS FALSE, THEN THE
590 REM STATEMENT "NOT PART3" WOULD BE TRUE
600 REM AND THE GOTO 400 WOULD HAPPEN.

```

This set of statements, in sequence, then means that if the number is a whole number (has no fraction part) and if it is greater than zero, go on to the next statement. If it either has a fraction part or if it is equal to zero, then GOTO 400.

Here is an example of using the parentheses to combine a whole set of logic tests into a single line. This saves a lot of typing as you can see when comparing it to the previous example.

```

22 N = DIVIDEND
23 IF NOT(N <> 0 AND ( N - INT ( N ) ) = 0 ) THEN GOTO 400
24 REM d          cb      a ab   cd
400 PRINT "THAT WAS SUPPOSED TO BE A WHOLE NUMBER"
410 GOTO 310

```

Line 23 looks somewhat complicated, but if you break it down, the functions it performs are identical to those in the many-line example presented just ahead of this section. The small letters are used again to mark the various levels of parentheses so you can see what ATARI BASIC will do first.

Just remember, when looking at parentheses, start counting them from left to right. ATARI BASIC will look at them the same way. It will say, looking at the preceding example, "Left parenthesis, another left parenthesis, another left one, another left . . . aha, a right parenthesis . . . that must apply to the most recent left one I saw." (This forms the "innermost nesting level.") "Now I'm looking for the next right parenthesis, which will be a match for the second most recent left parenthesis I saw . . ." and so forth until it finds a match for each one. Once it finds a match for one, the entire enclosed quantity can be calculated and can be considered as though it was a single variable.

You will see in the following explanation how the levels of parentheses match up, using the left-to-right scanning as explained in the preceding paragraph.

- (a) Marks the parentheses which tell the INT function what number to operate upon.
- (b) Tells ATARI BASIC to calculate the value $N - \text{INT}(N)$; in other words, take away the whole number part and just leave the fraction.

- (c) Tells ATARI BASIC to test if the fraction part is greater than zero (or you could test if it was greater than .0001 or something else).
- (d) Tells ATARI BASIC to see if N is greater than zero AND if the fraction part is equal to zero meaning that it is a whole number not equal to zero.
- (e) If all of those things are true, then it is OK to do the next line. If any of them is false, the number value resulting from the AND will be a zero. When you add the NOT in front of a zero result, it makes the result NOT zero, which means true. When the IF statement sees a true result, it does what comes after the THEN. In this case, it goes to 400, the error handler.

Use of Keywords

In line 240 in the preceding sample program, the word "remainder" is spelled correctly. As long as it is within the quote marks, any of the ATARI BASIC keywords may be used within words or as part of a string expression. However, when you make up variable names, you cannot use these keywords as part of the names.

For any keyword it finds, ATARI BASIC will try to execute the function. Therefore, you should at least be familiar with the keywords themselves when you try to write your own programs, even if you may not be totally familiar with the functions of each one.

OTHER FUNCTIONS BUILT-IN TO ATARI BASIC

Certain arithmetic functions have been built-in to ATARI BASIC so that you don't have to write a lot of formulas for things which might often be done. These are described in this section.

SGN Function

The SGN function is provided to allow you to determine if a number is greater than, less than, or equal to zero. You would use it as follows:

```
200 S = SGN ( D )
```

The variable S becomes +1 if the variable D is positive, 0 if the variable D is zero, and -1 if the variable D is negative. When you use SGN, it saves writing up to three program statements, namely:

```
200 IF D > 0 THEN S = 1
201 IF D = 0 THEN S = 0
202 IF D < 0 THEN S = -1
```

Only one of the preceding IF tests will be true, so S will get the correct value. But, as you can see, it is easier to use the SGN function instead.

THE SQR() Function

ATARI BASIC provides a function that will calculate the square root of a number. It is written as follows:

```
B = SQR ( A )
```

The parentheses are a part of the expression of the function and *must* be used.

Try a square root demonstration program. It will not only cover square roots, but it will also make a couple of other statements about programming. Here it is:

```
NEW
10 PRINT "GIVE ME A NUMBER,"
15 PRINT "I WILL CALCULATE THE SQUARE ROOT"
20 PRINT
30 TRAP 200
40 INPUT N
50 SN = SQR (N)
60 PRINT "THE SQUARE ROOT OF ";N
70 PRINT "IS: ";SN
80 GOTO 10
200 PRINT "THAT DOES NOT COMPUTE!"
210 PRINT:GOTO 10
```

If you RUN this program, you must use the **BREAK** key or **SYSTEM RESET** to exit.

Look at the preceding program. You will see that line 50 has

been used to calculate the square root value, directly after the INPUT statement. Why? Well, this program allows two possible chances for an error to occur. The first chance is during the INPUT statement where, if you gave it a bad number, it could generate an ERROR 8. The second chance is during the SQR calculation where a bad value would generate an ERROR 3 (value error). The program uses the TRAP statement to catch both of the errors. Whichever error causes the TRAP into line 200 prints the message. The recycle back through lines 10–30 resets the TRAP again.

If line 50 had been left out, then line 70 would have to read:

```
70 PRINT "IS: ";SQR(N)
```

If you did this, then gave the SQR function a bad input number (such as -1), then the computer would print:

```
THE SQUARE ROOT OF -1
IS: THAT DOES NOT COMPUTE!
```

instead of just the message

```
THAT DOES NOT COMPUTE!
```

It is up to you how you trap and report the errors. But this example, at least, shows you what could happen to a calculation within a PRINT statement and how to prepare for that kind of error.

The ABS() Function

In the first example of the IF statement in this chapter, you saw a program which tested an input value, then printed all the things about it that were true. In that example, nothing much was done with the values less than zero. Let's look at what might have been done with them.

ATARI BASIC has a function called ABS. It is used to find the absolute value of any number. This value will always be a positive number. As an example, the function is written in this way:

```
200 C = ABS ( N )
```

The parentheses are part of the expression of the function and *must* be used. (The statement $C = ABSN$ will not call this func-

tion; instead it will equate C to the variable "ABS(N).") If N is a positive number, then $C = N$. If N is a negative number, then $C = (-1 \text{ times } N)$. (In other words, $C = \text{the value of } N \text{ with the minus sign changed to a plus sign.}$)

A partial example program is shown here. It does not include the error testing, but is complete enough to be RUN to show how to use the ABS() function. It will also show you some other ways to form an IF test.

```

NEW
10 PRINT "GIVE ME A NUMBER"
20 INPUT N
30 NA = ABS(N)
40 IF NOT N THEN PRINT "N IS ZERO":GOTO 10
50 IF N < 0 THEN PRINT "N IS NEGATIVE"
60 IF N > 0 THEN PRINT "N IS POSITIVE"
70 IF NA > 1000 THEN PRINT "ITS VALUE IS OVER 1000"
80 IF NA <= 1000 THEN PRINT "ITS VALUE IS LESS THAN OR
EQUAL TO 1000"
90 GOTO 10

```

Line 30 calculates the absolute value. In line 40, the THEN part of the line not only prints the statement about the value, but it also performs the GOTO function. This was done to prevent the IF test in line 80 from being performed. Can you tell why line 90 was added at the bottom of the program? Why didn't the program just end with line 80 reading as follows?

```

80 IF NA <= 1000 THEN PRINT "ITS VALUE IS LESS THAN OR
EQUAL TO 1000":GOTO 10

```

Well, if N was greater than 1000, and if there was no line 90 included, ATARI BASIC would evaluate line 80 as false, and never do anything within that same statement following the THEN. Therefore, it would run out of statements to execute and wind up back in the command (READY) mode.

There is another solution to this problem. That would be to change line 70 of the program to read:

```

70 IF NA > 1000 THEN PRINT "ITS VALUE IS OVER 1000":GOTO 10

```

which means there is a GOTO 10 as part of both a test that will be true and a test that will be false. Therefore, one of these paths to line 10 will be taken. Remember that this program did not use any input error testing. By now you should know how to put in error TRAPS.

OTHER MATH FUNCTIONS

ATARI BASIC includes a number of other math functions you can use. These are specified in this section. This book is not intended to give extensive examples for the use of all the math functions. Rather, it is intended to show you the correct way to write ATARI BASIC statements and what each of the keywords means.

In addition, a goal of this book is to show you exactly what the process of designing a program is all about. Therefore, these math functions show only how to write the statement correctly. Many BASIC users will use these functions far less than any others mentioned in this book.

CLOG Function

ATARI BASIC includes this function to calculate a value equal to the logarithm (base 10) of the variable or expression enclosed in parentheses. This function can be used as follows:

```
200 ANSWER = CLOG(VARIABLE)
```

The parentheses are required. (Note: There are small bugs in this routine which return incorrect values for CLOG(1) and CLOG(0). These are specified in your ATARI BASIC Reference Manual.)

LOG Function

ATARI BASIC includes this function to calculate a value equal to the natural logarithm (base e) of the variable or expression enclosed in parentheses. This function can be used as follows:

```
200 ANSWER = LOG(VARIABLE)
```

The parentheses are required. (Note: There are small bugs in this routine which return incorrect values for LOG(1) and LOG(0). These are specified in your ATARI BASIC Reference Manual.)

EXP Function

ATARI BASIC provides this function to calculate a value of e (approximate value 2.71828283) raised to the power you specify within the parentheses. As with other functions, the item enclosed in the parentheses can be a variable or an expression. This routine can only be used if the accuracy of the result is required to be six significant digits or less. An example of the way to use the EXP function is:

```
200 ANSWER = EXP (VARIABLE)
```

DEG / RAD Functions

For the trigonometric function descriptions that follow, the expression within the parentheses will be evaluated either in radians or in degrees. When the computer is first powered up, the RADians function is active. The words DEG and RAD are used to switch back and forth between the two. If you type

```
DEG RETURN
```

or use

```
200 DEG
```

in your program, then any trig functions will be evaluated in degrees from that time on. Likewise, if you type

```
RAD RETURN
```

or use

```
300 RAD
```

then any trig functions will be evaluated in radians.

SIN Function

This function calculates the sine of the angle specified in the parentheses. An example is shown below:

```
200 DEG
210 ANSWER = SIN(45)
220 PRINT ANSWER
```

This program calculates the sine of the angle 45°. This result is approximately 0.707. The parentheses are required. The value of SIN() will always be between 0 and 1.

COS Function

This function calculates the cosine of the angle specified in the parentheses. An example follows:

```
200 DEG
210 ANSWER = COS(60)
220 PRINT ANSWER
```

This program calculates the cosine of the angle 60°. This result is approximately 0.5. The parentheses are required. The value of COS() will always be between 0 and 1.

ATN Function

This function returns the arctangent of the variable value which you enclose in the parentheses. This function is the reverse of the tangent function in that it calculates the angle in degrees or radians from which the tangent value would have been derived. An example will show what this means:

Appendix E of the ATARI 400/800 BASIC Reference Manual shows that the tangent function can be calculated from the SIN or COS functions as follows:

$$\text{TANGENT} = \text{SIN}(X) / \text{COS}(X)$$

where X is the angle in degrees or radians. If $X = 45^\circ$, then the program shown here will calculate the tangent, then use the ATN function to prove that this is the angle which would produce this tangent value.

```
NEW
100 DEG: REM USE DEGREES
110 TANGENT = SIN(45)/COS(45)
120 PRINT ATN(TANGENT)
```

You will see that the answer is about 45°. (Again, the limits of precision within the machine may give an answer very close but not exactly correct.)

REVIEW OF CHAPTER 2

Thus far, you have learned that:

1. The IF test can be used to test many different conditions and combinations of conditions.

2. A TRAP statement can be used to prevent a bad input from affecting the program outcome, but the TRAP must be reset each time it is used.

3. ATARI BASIC can do various numerical calculations. In the process of doing the calculations, it evaluates your program lines from left to right, but also does certain operations before it does others. This is known as *operator precedence*.

4. Parentheses-enclosed expressions are evaluated from the innermost set to the outermost set. ATARI BASIC matches up parentheses by taking the last encountered left parenthesis as a mate to the earliest encountered right parenthesis. After each match-up, that entire quantity can be treated as though it is a separate variable.

5. Logic expressions, such as used in the IF tests, are called true if the value is nonzero, and false if the value is zero. An IF test will pass (that is, allowing its THEN part to be executed) only if the expression it tests is true.

6. A variable can be used as a counter (see "A New Value for a Variable") by adding a quantity to the old value of the same variable.

7. Following the THEN in an IF-THEN statement, a line number may be used, as long as the line number is an actual number and not a variable (such as: 40 and not something like: M).

8. If you wish to assign a variable that will tell ATARI BASIC where to go following an IF test, the THEN will be followed by a GOTO, and the line number following the GOTO may be a variable, such as:

```
IF ( N > 0 ) THEN GOTO M
```

where M is a variable equal to one of the actual line numbers in the program. This is known as a *computed* GOTO. (It does not need to be part of an IF-THEN test, it may be used wherever a GOTO can be used.)

9. The INT function always rounds *down* to the next nearest integer value.

10. Anything following the letter combination REM is ignored by ATARI BASIC. It is, however, stored as part of a program and will be there again when you list it. It merely does not execute any part of the statement following the REM.

11. You must be careful when choosing variable names to prevent ATARI BASIC from misinterpreting you. Any variable name cannot contain any of the reserved keywords.

12. ATARI BASIC has some math functions built-in to aid your calculations.

As with Chapter 1, if any of the preceding items seem unfamiliar to you, it would be a good idea to go back through this chapter and become familiar with them. This book was designed to build on previous examples wherever possible, both to save you typing and to logically build up your familiarity with ATARI BASIC, step by step.

CHAPTER

3

Stringing Along

The purpose of this chapter is to show you how ATARI BASIC handles and uses strings. The first thing you have to know here is:

"WHAT IS A STRING?"

Well, the easy answer to that question is shown by example. And the example is the question itself!

A string is a group of characters enclosed between a pair of double quotation marks. The characters that are used in a string in ATARI BASIC can be capital or small letters, numbers, arithmetic symbols, and just about anything else having an ATARI ASCII (ATASCII) value. (See Appendix C of your ATARI BASIC Reference Manual for the appearance of some of the symbols that might occur in a string.) In fact, the only character that ATARI BASIC cannot allow in a string is the double-quote (") character. This is because the first double quote marks the beginning of a string, and the second double quote signals ATARI BASIC that this is the end of the string.

Normally, one expects to see and use the alphabet, numbers, and punctuation signs in a string. The "normal things" will be what will be used for strings in this book.

You have seen strings before in the first two chapters, such as the strings "HELLO" and "OTHER STUFF." In the first two chapters, strings were used strictly for passing small messages to the user. In this chapter, you will see how strings can be used in other ways. For example, in the second chapter you saw an example program in which you had to type a number as a reply to a program question. It would be nice if instead of a number, the reply could be a word, such as YES or NO, or maybe just Y or N for short. Using strings in your program can provide this capability.

HOW ATARI BASIC HANDLES STRINGS

Strings in ATARI BASIC are handled a little differently than in other forms of BASIC. In particular, the various Microsoft® BASICs do not require that you save space for a string. ATARI BASIC *does* require that space be saved for each string you will use. No string in ATARI BASIC is allowed to be larger than the storage space you have set aside for it.

The DIM Statement

Here is an example program showing how you save space for a character string. It will also show what happens to the extra characters if there is not enough space to store them. The DIM keyword is the word that tells ATARI BASIC to save some space for the string. (The DIM statement has other uses in what is called *array* storage, but that is a more advanced topic and will be covered later.) Try this example and see for yourself what happens:

```
10 DIM A$(9)
20 A$ = "123456"
30 PRINT A$
```

Before you run the program, check a couple of notes on what you see here.

DIM is actually short for DIMension (but never use the whole word, ATARI BASIC won't understand it). This specifies the size of the memory in one "dimension." In this case, only one dimension, that of "length," is specified. String variables can only have

the dimension of length. When you do arithmetic with number arrays, you will see that numeric variables can also have the dimension of width. But again, that is a future subject.

You will notice that the string variable ends in a dollar sign (\$). This is common to all string variables and tells ATARI BASIC that it is a string. This means that you cannot make up a number variable name which ends in a dollar sign.

Line 20 assigns a value to A\$. Notice that since it is a string variable, the value assigned to it is a string and is enclosed in quotes. RUN this program and see what happens. The full set of characters assigned to A\$ has been printed by the PRINT statement. Now change line 10 to read:

```
10 DIM A$(3)
```

and RUN the program again.

Now notice that only the first three characters have been printed. ATARI BASIC has reserved only a space three characters long for the string called A\$. Any extra characters wouldn't fit into that space, so ATARI BASIC has thrown them away! This did not cause any error, it is just the way ATARI BASIC handles strings. Now, change line 10 back to DIM A\$(9), add the following line:

```
25 A$(7) = "789"
```

and RUN the program again.

You will see that ATARI BASIC allows you to merge together sets of strings into one longer string just by telling it the position number at which the add-on is to start. Now change line 25 to read:

```
25 A$(10) = "789"
```

and RUN the program again.

This time the program stopped with an ERROR 5. This means that you tried to assign a start position number that was larger than the dimension number you specified for the string array. Remember that it is OK to add extra characters which ATARI BASIC will just throw away, but you must specify a starting point

within the dimension of the string in the first place. Now change the example again to read:

```
10 DIM A$(9)
20 A$="XX"
40 PRINT A$
```

and RUN the modified program.

What does the string look like? Just a pair of XXs, right? Even though you have reserved nine spaces for the string, you only used the first two spaces. ATARI BASIC knows this and when you say PRINT A\$, it means print A\$, starting at character one, and print as many characters as the user has assigned to this string.

Now change the example program again. This time add line 30 shown here:

```
30 A$(8)="XX"
```

and RUN the program again. What did it print?

Yes, the XXs are where you would have expected them to be, but what is between the XXs? If you have followed along with the preceding examples, the result most likely reads:

```
XX34567XX
```

These are the leftovers from the previous program you tried where A\$ was defined as "123456789." This shows you that when a program starts (using the RUN statement), the string variable areas, which you define, may contain any kind of data, even graphics characters or other "garbage." ATARI BASIC does not clear out any data left in these areas either at program start or during the program run. Again, to further demonstrate that, change line 30 to read:

```
30 A$(3)="Z"
```

RUN the program, then change line 30 to read:

```
30 A$(9)="Z"
```

and RUN it once more. Notice that after the first change, the extra characters temporarily disappear. This is because the very last time that ATARI BASIC sees something added to a string, it says

that the last character position filled defines the end of the string. Therefore, when you tell it to print A\$ it prints from the first character to and including the "last known" character position.

The last change mentioned to line 30 shows you that even though ATARI BASIC temporarily changed its definition of where the string ended, it did not change any of the characters that were part of the earlier version of the string. This means that if you really want to be sure you know what characters are part of a long string, you must define them yourself for the full length of the string. Some programs will use strings in this way to build up a whole line of data for output. This is known as formatting an output line. You will see this subject later in the chapter titled "Menu Please."

Without running the program again, type the following command in direct mode:

```
PRINT A$(5) (and press RETURN)
```

Notice that this prints the string beginning at the fifth character, and prints all of the rest of the string. Again, once ATARI BASIC has been told to print the string, it begins where you tell it to start, then continues till it reaches the position indicated by the "current length."

Now, how about doing a little string "splitting"? Sometimes you might want to look at just one character, or maybe a group of characters that are part of a longer string. Try the following add-on to the preceding example. The whole example is shown here just in case you put the book away and have just picked it up again for this section. Here, the complete example will show a string split method, and includes all of the changes you made so far.

```
5 DIM B$(10)
10 DIM A$(9)
15 A$ = "123456789"
20 A$ = "XX"
25 A$(8) = "XX"
30 PRINT "CURRENT A$ CHARACTERS: ";A$
35 B$ = A$(4,6)
40 PRINT "CHARACTERS 4 THRU 6 ARE: ";B$
```

RUN this program. The second line should show the contents of B\$ (pronounced "B string") are "456." (No quotes will be displayed on the screen with the string value.)

Notice that you can, if you wish, combine the printing of a string variable with a character string on the same line, as in line 30. You can even use a PRINT statement to group together a number of different strings, such as by the statement:

```
PRINT A$;B$
```

or some similar type of statement. Again, the semicolon tells ATARI BASIC to stay in the last position after you print that section of the line . . . don't go on to the next line before you print what's coming next.

Line 35 says B\$ = A\$(4,6). This says start defining the characters for B\$ starting at character position 1 and use A\$ character number 4 as the starting character and A\$ character number 6 as the last character. Since this is three characters total, B\$ will have a length of three, and will be composed of the characters "456."

This type of command not only allows you to split strings but also to rearrange them. For example, to move characters from one section of a string to another, you might use the following type of program. Again, it has been kept as short as possible to save you some typing.

```
NEW
10 REM THIS PROGRAM MAY BE PART OF
20 REM A MAILING LIST PROGRAM
30 DIM A$(40),B$(40)
40 A$ = ".....":A$(11) = A$:A$(21) = A$:B$ = A$
50 PRINT "TYPE YOUR LAST NAME AND PRESS RETURN"
60 INPUT A$
70 PRINT "TYPE YOUR FIRST NAME AND PRESS RETURN"
80 INPUT B$
85 A$(26) = B$
90 PRINT "YOUR NAME IS STORED AS:"
100 PRINT A$
110 B$ = A$(26):B$(16) = A$
```

```

120 PRINT "WHEN YOU PRINT A MAILING LABEL"
130 PRINT "IT SHOULD APPEAR THIS WAY:"
140 PRINT B$
150 END

```

Line 30 reserves 40 spaces for each string. A\$ (remember, it is pronounced "A string") is going to be used to store your name in the form LASTNAME, FIRSTNAME just as you might see it on an index card. If someone was going to mail you a letter, though, it would look funny if he put your last name first. So this program will use the built-in ATARI BASIC string functions to swap the last name and first name before they are printed.

In line 40, we got a little fancy. Remember you saw that ATARI BASIC does not do anything to the memory areas assigned to string storage before these areas are used. Line 40 is in there to put some starting values into the memory so you won't see any garbage. A set of 10 periods is used between the quotes. So the first part of line 40 assigns the first 10 positions of A\$ to be 10 periods.

The second part of line 40 does a second part of setting up the value in A\$. (Remember compound statements . . . if not, recheck Chapter 2 again . . . use the colon to separate parts of the compound statement.) It takes the value currently in A\$, starting with character 1, and duplicates that value in character position 11, duplicates position 2 in position 12, and so forth until the length of A\$ is 20, and the first 20 characters are all periods. The third segment of line 40 assigns characters 21–40 the same as characters 1–20. The final segment initializes B\$.

But wait, there's something fishy here, isn't there? Earlier you saw that ATARI BASIC, when it is supposed to do something with a string, starts at the specified position (or the first position if it is not told where to start). This program starts with a string that is 10 characters long, and fills in the 11th character, then the 12th, and so on. As it is going onward, it is supposed to keep going until it reaches the end of the string. But this statement, A\$(11)=A\$, would seem to keep making the string longer and longer. Why doesn't it go on forever if the string is 10 characters long when it starts, then 11 characters long after one character is moved, then

12 and so on? It would seem that it would always be 10 characters behind any ability to perform this request.

Well, ATARI BASIC solves this problem by always remembering what the "current length" of the string happens to be before it starts anything like this. Then, only after the data move has happened does it change the "current length" to whatever it has become as a result of the move. Therefore, if it starts with a length of 10, it moves 10 characters only. Then it looks to see what the length has become and remembers this, after the move has happened.

Lines 60 and 80 are there to show you that you can use the INPUT statement to input a string variable, just as you used it in Chapter 2 to input number values. All you have to do is tell ATARI BASIC what kind of variable is to be input, and it handles the rest. Line 110 says:

```
110 B$ = A$(26):B$(16) = A$
```

Let's look at a picture of what this line does. First, let's assume that you entered A\$ to read:

```
LASTNAME
```

then

```
FIRST
```

It would be stored as:

```
LASTNAME.....FIRST
```

with the periods for the fillers. Periods don't appear after the word FIRST because the total length of A\$ was defined by the last character position that was filled.

Now the first part of line 110 does this (Here's a diagram of B\$ after the first part of line 110):

```
FIRST
```

It is five characters long, starting with character 26 from A\$, and running out to the current length of A\$, which would be 30 characters (moved numbers 26–30, inclusive). Now the second part of line 110 reads:

```
B$(16) = A$
```

This says to take all characters from A\$, one at a time, and assign them to B\$ starting at character position 16. This does the following: Old B\$ is:

FIRST

But the entire B\$ data area looks like this:

FIRST.....

because the area was initialized in line 40 with periods (40 total).

Now we add A\$ contents starting at position 16, (B\$ data area copied again just to show everything up close):

<— 40 characters long —>

FIRST.....

LASTNAME.....

(This line is down here to illustrate FIRST..... that ATARI BASIC “drops” the extra characters which go beyond the end of the available data space.)

When the lower line (containing LASTNAME) is stored into the memory space containing the upper line (FIRST...), the B\$ memory looks like this:

FIRST.....LASTNAME.....

<— 40 characters long —>

and it will print in the correct order as the program is supposed to do.

The LEN Function

In the preceding discussions, there are many references to the length of the string. Since ATARI BASIC knows what the length of each string happens to be, it might be useful for us to know this also. Why? Well, someday you might want to write a program that does crossword puzzle word searches. Let's say you have your “dictionary” set up so that you have all of the one-letter words first, then all of the two-letter words, the three-letter words, and so on. Then when you ask the program to give you all of the four-letter words starting with “g” and having an “e” in the fourth

position, you would not want the computer to waste its time searching all of the one-letter words first, then the two-letter words, and so on. In particular, it is a waste of time, especially if the language, such as BASIC, is a little slow on long searches. Therefore, you would need to know the length of the word you were looking for so the computer could start its search at the most efficient place in the dictionary.

Fortunately, ATARI BASIC provides the LEN function for this purpose. The LEN function returns a value which is the number of characters currently assigned to the string name you request. You write this function as follows:

```
10 N = LEN(A$)
```

where the complete name of the string variable is enclosed in the parentheses. Here is a very short program which demonstrates the LEN function:

```
NEW
10 DIM A$(120)
20 PRINT "TYPE SOMETHING AND HIT RETURN"
30 PRINT "I WILL TELL YOU HOW MANY CHARACTERS"
40 PRINT
50 INPUT A$
60 PRINT
70 PRINT "YOU TYPED ";LEN(A$);" CHARACTERS"
80 END
```

The value you get from the LEN function can be used like any other number value. It is an integer (whole number) and can be used in calculations or anywhere else a number is accepted. That is why the first example line showed the statement, $N = \text{LEN}(A\$)$.

When you look at the mailing list program again, you will see that if you used the LEN function, you might have done things a little differently. For example, the way the final printed name (first . . . last) is put together, there is a lot of space allowed between the first name and the last. Also, maybe you wouldn't want to print the dots there after all.

Let's look at how the LEN function could have helped to make

the final printed form a little better looking. In the example, the last name was read, then the first name. After the last name was read, ATARI BASIC knew that the length of A\$ was eight (LAST-NAME). A number variable could have been assigned to hold that value, let's say it was called LNAME. So to use the LEN function for this, you would have written:

```
65 LNAME = LEN(A$)
```

After the first name was read, ATARI BASIC knew the total length of A\$, and that was 30 characters (25 characters and dots combination allowed for the last name section, then 15 characters and dots allowed for the first name). Since you knew where the program started storing the first name (at character position 26), you can move only the right amount of characters into the area (B\$) from which you wish to print. Let's see how.

Please refer back to the mailing list program as we show these add-ons to it. The purpose of the add-ons is to show you how the LEN function can help. You may try to RUN the modified program if you wish. Add line 65 as shown above (repeated here):

```
65 LNAME = LEN(A$)
```

Retype line 110 to read:

```
110 B$ = A$(26,LEN(A$))
```

or

```
110 B$ = A$(26)
```

which, in this case, does the same thing (uses characters from 26–30). Then add lines 113, 114, 115, reading:

```
113 LFIRST = LEN(B$)
```

```
114 B$ (LFIRST + 1) = " "
```

```
115 B$ (LEN(B$) + 1) = A$(1,LNAME)
```

What are the functions of each of these lines? Line 113 says the number variable called LFIRST is set to the current length of B\$. This becomes a pointer to the last character you added into B\$, which here is the last character of the first name. Line 114 says starting with the next character position, add onto B\$ with a string

composed of two blank spaces (because that is what is contained between the two quote marks). So far, this builds B\$ into the first name, followed by two blanks. Line 115 says starting with the first character position following the two blanks, begin storing the characters of the last name. On the right side of the equals sign it shows the character limits, from 1 to the length of the last name (which line 65 calculated).

On the left side of the equals sign, it shows that we didn't assign any temporary variable name to the value LEN(B\$), as we had done with LFIRST and LNAME previously. Anywhere that a value is needed only once in a program, it is usually OK to use the equation which represents that value, especially since ATARI BASIC will calculate the value most anywhere it is properly placed. Since it is inside the parentheses, ATARI BASIC must do everything that is inside of the parentheses before it can find out what number is to be used. Therefore, this type of equation is OK to use.

There is another reason for sometimes using the equation rather than assigning a new temporary variable name. That is, the limit of 128 variable names which was mentioned in Chapter 1. However, it is not too likely that a beginning programmer will run into this limit very often. It is just a little something to show you now, so that you might recall seeing it "somewhere" when you do find such a problem.

Now if you RUN the mailing list sample program with those changes in place, you will see the output generated as:

```
FIRST LASTNAME <-(no dots here anymore)
—> <-(two spaces here)
```

and this looks a bit better than what we started with.

OTHER STRING FUNCTIONS

Let's now look at a couple of other string functions that ATARI BASIC has, namely the STR\$ and VAL functions. These functions are the link between number variables and string variables because by using them you can convert one kind to the other.

The STR\$ and VAL Functions

First we'll look at the STR\$ function. Its purpose is to convert a number variable into a string variable. You would write a line using this function in this way:

```
200 A$ = STR$(N)
```

The first condition is that A\$ has had a DIM statement near the top of the program so that there is some space reserved for ATARI BASIC to store the characters.

The second thing you should know is that N is a numeric variable. The value of N can be an integer or a real number (can have a decimal point). The value of N can be any value which ATARI BASIC considers within its maximum range (roughly $+1.0E+97$ to $1.0E-97$). In fact, if you give a number to this function having the "E" in it, it will also convert the "E" part to a string. So, for example, if $N=1.348E-17$, then the string value of N will be 1.348E-17, and so forth for any real number.

The VAL function provides exactly the opposite action, taking a string representation of a number value and converting it into a real number you can work with. You would use the VAL function in this way: Assume that A\$ contains the characters, 3.14159, then the line:

```
300 N = VAL(A$)
```

would assign the value of 3.14159 to the variable N.

What is the advantage of being able to go from string variables to numeric variables and back again? Well, let's look at a couple of example problems. Say you were asked to write a program that would take any input number and tell how many digits there were after the decimal point. How would you do it? Look at the number 123.1. This one would seem easy to do. First, use the integer function program that was developed in Chapter 2 to strip off the digits before the decimal point, then just work with the remainder (which will be all of the digits after the decimal point). So the remainder is 0.1. How can you use a program to tell how many digits there are in this number? First, you multiply the num-

ber by 10 (result is 1.0). Then use the integer function again to strip off the whole number and see if the remainder is zero. If it is not zero (such as if the number was 123.147), then add 1 to a counter and let the counter count the number of digits past the decimal point. Keep going until the remainder is zero.

This program is shown here for you to try if you wish. You will see that for the smallest number that ATARI BASIC can handle (about $1.0E-97$), this program runs through 97 passes before it finally discovers how many digits will be placed after the decimal point. (Recall that this number is a 1 preceded by 96 zeros, making a total of 97 digits to the right of the decimal point.)

There is a second program that follows this one. It will use the STR\$ function to do the same work. You will see that it takes only one pass through the program to do the same job. If time is important, it is usually best to choose the most efficient way to do a job.

Here's the first digit counter program, using the INT function as mentioned earlier:

```

10 N = - 1:REM SET DIGIT COUNTER TO FIRST VALUE
20 PRINT "INPUT A NUMBER, I'LL TELL YOU"
30 PRINT "HOW MANY DIGITS ARE TO THE"
40 PRINT "RIGHT OF THE DECIMAL POINT"
50 PRINT "NUMBER = ";INPUT X
60 N = N + 1
70 Y = INT(X)
80 Z = X - Y
90 IF Z = 0 THEN GOTO 200
100 X = Z*10
110 GOTO 60
200 PRINT "THERE WERE ";N;" DIGITS TO THE RIGHT"

```

Line 10 says to set up the value of N as -1 . The reason for this is that the first time the INT function is used, the program will dump all digits which are to the left of the decimal point. This means that the correct answer for an integer value should be "0" digits. Line 10 will cause this result.

Each run from line 60 through line 110 will then count one more digit, moving all the rest of the digits to the left by one slot, then

tossing them away ($Z = X - Y$), leaving only the remainder. As long as the remainder is not zero, the program will keep looping around, trying to count yet another digit after the decimal point.

A number like $1.23456E-96$, though, will not have any digits move until very near the end of the counting. That is because this number is 123456 preceded by 95 zeros. So it will be a batch of zeros that will always occur to the left of the decimal point for each multiplication, and the remainder will be nonzero until the very last digit has been counted.

Try the program. It will work on all positive and negative numbers, whether large or small. But notice that for the smallest ones, it could take up to 97 loops through the test part before it is able to report the results. Try the number $1.2345678E-97$. This one will take about 3 or 4 seconds to produce the result. That is OK, but it is one case where there is a more efficient way to do something. If your program has to do a lot of this kind of calculations, the fastest way will be the best for you most of the time.

Now you can look at and perhaps try the next program. It does the same thing as the last one, but uses the STR\$ function instead. You will see that it is much faster, even on the example number ($E-97$).

If you have already entered the earlier program, you can just enter this one by changing the lines from 60 to 160, and adding line 5. The rest of the program is exactly the same.

```

5 DIM A$(20)
10 N = -1:REM SET DIGIT COUNTER TO FIRST VALUE
20 PRINT "INPUT A NUMBER, I'LL TELL YOU"
30 PRINT "HOW MANY DIGITS ARE TO THE"
40 PRINT "RIGHT OF THE DECIMAL POINT"
50 PRINT "NUMBER = ";INPUT X
60 A$ = STR$(X)
70 DP = 0:E = 0:LEFT = 0:RIGHT = 0:P = 1
80 IF DP <> 0 THEN GOTO 95
85 IF A$(P) = "." THEN DP = P:GOTO 105
88 IF A$(P) = "E" THEN E = P:GOTO 120
90 LEFT = LEFT + 1:GOTO 105
95 IF A$(P) = "E" THEN E = P:GOTO 120

```

```

100 RIGHT = RIGHT + 1
105 P = P + 1:IF P <= LEN(A$) THEN GOTO 80
110 N = RIGHT:GOTO 200
120 P = P + 1:Q = LEN(A$)
130 Z = VAL(A$(P,Q))
135 IF A$(1,1) = "-" THEN LEFT = LEFT - 1
140 Z = Z + LEFT
150 GROUP = LEFT + RIGHT:N = GROUP - Z
160 IF N < 0 THEN N = 0
200 PRINT "THERE WERE ";N;" DIGITS TO THE RIGHT"

```

If you try this program, it will run much faster than the earlier version, even though it is longer. It will take no more than 12 or so calculations, regardless of the size of the number, to figure out how many digits there are. How does it work?

Line 60 sets A\$ equal to the string equivalent of the number you entered. Line 70 sets various number variables to zero. The number called DP represents the position of the decimal point in the string. It is set to zero to start because the search has not yet started. Likewise, the E variable is set to zero and will represent the position, if any, of an E in the string (if it is a very large or very small number, it will be printed with the E). Variables LEFT and RIGHT will represent how many digits are found to the left and right of any decimal point. They are zero to start also, as the search begins.

Line 80 is a test. It sees if the pointer has reached a decimal point previously. If so, no more LEFT digits should be counted. Line 85 checks to see if the current character, A\$(P,P), is a decimal point. If so, DP will now be set to a number other than zero. This line looks at the value of a single character of A\$ by using the index of (P,P). If you recall earlier, we used the numbers in the parentheses for a string to show which character positions to operate with, such as A\$(4,6). This meant take the characters starting with number 4 and continue on with the string until character 6 is used. In this case (P,P), it means take only one character. For example, (4,4) says it is a one-character string that we'll use for the comparison operation, and it is character 4 (starts with 4, ends with 4). Line 85 is comparing this one character at

the selected position to a decimal point. If it is a decimal point, the program needs to search no more. If not, the program will fall through to line 88.

Line 88 is also a test. Very large and very small numbers are printed by ATARI BASIC with an E in them. If the pointer into the string has found an E, it is also time to stop counting LEFT digits. Line 90 adds 1 to the pointer and sends the program back to the top again if another LEFT digit (digit to the left of a decimal point) has been found. Line 95 is identical to line 88. If the program finds the E character, it will have no more digits to count to the right of the decimal point.

Line 100 adds 1 to the count of RIGHT digits. Line 105 adds 1 to the character position pointer and checks to see that it is less than or equal to the number of characters in the string. If so, the program proceeds to examine another character.

If the program gets to line 110, it means that there was no E character encountered. Therefore, the number of digits to the right of the decimal point was equal to RIGHT. If the program found the end of the string before any decimal point was seen, then RIGHT = 0 and it reports it this way.

At line 120, the E character has been found, and a pair of digits is produced which will point to the start and the end, in the string, of the digits after the E character. For example, in the number 1.2345E-24, it points to the characters making up the -24. Line 130 then uses the VAL function to change the -24 characters into a number which you can add or subtract with other numbers. This number is called Z.

At line 135, the value of LEFT is adjusted in case there was a minus sign present as part of a number. Because of the way the number LEFT was counted, it is possible that the number of characters to the left of the decimal point should be one less than the number found. As an example, in direct mode you can type:

```
N = 100  RETURN
PRINT N
```

and the response will be:

```
100
```

But, if N is a negative value, the first character that will be printed will not be a "number," it will be a minus sign. For example:

```
N = -34
PRINT N
```

will respond:

```
-34
```

Since the STR\$ function operates the same way as printing to the screen, line 135 is used to test the first character to see if it is a minus sign. It tests A\$(1,1), which is the first character, with a LEN of 1. If it is a minus sign, then there is one less digit to the left of the decimal point than the value currently in variable LEFT, and it has to be adjusted to LEFT - 1.

Line 140 says Z = Z + LEFT. This adjusts the value found past the E character to "normalize" the number. This means that it places *all* characters after the decimal point and tells what the value of Z would be if that would happen.

Line 150 says GROUP = LEFT + RIGHT, meaning that the total group of digits which were handled and which are now to the right of a phantom decimal point are the total of the left-hand and right-hand digits combined. It also sets N = GROUP - Z. This takes the normalized E number and adds the total number of digits to it. The result is the total number of digits to the right of the decimal point.

Line 160 says that if N is less than zero, there must be zero digits to the right of the decimal point. Whatever value is determined is printed by line 200.

Let's look at an example number just to see why the program does what you see:

```
1.725E - 10
```

This value could also be represented as:

```
.1725E - 09
```

because moving the decimal point to the left by one subtracts one from the E-character value. This number, as normalized, now says there are 9 zeros to the right of the decimal point. We can

also see that there are 4 digits to the right of the decimal point in the E-representation of the number. Therefore, if the number was drawn out completely, it would have to look this way:

```
0.0000000001725
```

or a total of $9 + 4 = 13$ digits to the right of the decimal point. This is what the program will calculate and display as described.

Again, when you try the program, notice how much faster it is than the other version. Even though the program is longer, it takes much less time to do the job.

STRING COMPARISON FEATURES

In the preceding program, you saw a couple of uses of the comparison operators (the equals sign and the less-than-or-equals sign) in the IF statements. This time they were not being used on regular number quantities, but rather on string variables. In fact, *all* of the comparison operators are just as valid on string variables as on numeric variables. For example, you can try a small program like this:

```
NEW
10 DIM G$(10),H$(10)
20 G$="ABCDE"
30 H$="Z"
40 IF G$<H$ THEN PRINT G$;" BELONGS IN FRONT OF ";H$
50 IF G$ = H$ THEN PRINT G$;" AND ";H$;" ARE IDENTICAL"
60 IF G$>H$ THEN PRINT G$;" BELONGS BEHIND ";H$
```

Try it, then if you want to try some combinations, change lines 20 and 30 to read:

```
20 PRINT "PLEASE SPECIFY STRING 1";:INPUT G$
30 PRINT "PLEASE SPECIFY STRING 2";:INPUT H$
```

and add line 70:

```
70 IF H$ <> "STOP" THEN GOTO 20
```

The program, with these changes, will keep accepting input strings from you until you type the word STOP as string number 2. Then

it will return to BASIC direct command level. Now you have a convenient, English-language way to make your program stop without hitting **BREAK** or **SYSTEM RESET** or using any number tricks.

You may have some occasion to do alphabetic sorting at some time in the future. One thing you must know about these string comparison features is that they take things very "literally." They compare the first character to the first character, then the second character to the second character, and so on until one of the strings runs out of characters to compare.

If, at the time a string runs out, both strings are identical, the one with the larger number of characters will be considered to be the greater one. As an example, use the preceding program to compare a string 1 and string 2 of:

```
MOTH
MOTHER
```

You will see that MOTHER comes after MOTH, and is considered a higher number value. This is as it should be and is as a dictionary would list it. It does not matter, for example, if you had specified string 1 earlier as ZZZZZZZZZZ, then respecified it as MOTH. The string memory storage, if you remember, will have the word stored as MOTHZZZZZZ, which would certainly make it seem to come after MOTHER. But the comparison operators only care about the *current length* of the string variable. So MOTH will come first.

If, in the process of comparing the values, you had specified lines 20 and 30 as follows:

```
20 G$ = "ABC"
30 H$ = " Z"
```

you might think . . . "Z comes later than ABC in the alphabet, so I think I shall have to put it that way." ATARI BASIC, on the other hand, has no choice but to look at it on a character-by-character basis. It doesn't care that these are alphabetic characters, and can only go by the ATASCII number value assigned to the character itself. The ATASCII value for a blank space (which precedes the Z) is decimal 32. The ATASCII value for the capital letter A is

decimal 65. Therefore, the string with the Z in it, since it has a lower number value in a numeric placement, must come first!

Therefore, when you are trying to compare ATASCII character strings, make sure that the first characters are lined up the same way you want them to be interpreted. (This is called "left-justified," meaning butting-up against the left-hand edge of the space reserved for the word.)

THE ASC AND CHR\$ FUNCTIONS

ATARI BASIC has two more string-related keywords. These are ASC and CHR\$. VAL and STR\$ are related to number conversions only. For example, if you tried to perform a VAL function on a nonnumeric string, ATARI BASIC would give an ERROR 18 (an unexpected value was found).

The ASC function will return the decimal value of the single ATASCII character on which it operates. For example, the line:

```
PRINT ASC("A")
```

returns the value 65.

The CHR\$ function will return the character whose value is specified in the parentheses. For example the line:

```
PRINT CHR$(65)
```

prints the character A. In this manner, the ASC and CHR\$ functions are exact complements of each other. This can also be shown by the sample line:

```
PRINT CHR$(ASC("A"))
```

which also prints the character A. If you remember the rules on parentheses-enclosed items, the "innermost" function is done first, then each outermost function until the line is complete. In this case, the ASC function is done first, putting the value 65 into the CHR\$ parentheses. This then prints the letter A.

You will see strings used elsewhere in this book for various purposes. Most of them, due to the nature of the book, will be used to communicate somehow with the user.

In the chapter titled "Menu Please," you will see an example of

the use of the ASC function to simulate the VAL function, and another example will show you how to use the CHR\$ function to do the same thing that the STR\$ function will do.

Why would you want to do this? Well, let's say there was a case where you wanted a number of items to be entered on a single line. If you did not want to ask the user to enter a comma between each item to separate them, you might need to separate the string items yourself and find out if they were numeric or string data. These program pieces will give you an introduction to the use of these functions.

Such a function might be used in a program language translator where it is necessary to see if the item entered is a number or a variable name. In each case, the item would be treated differently.

The CHR\$ function can also be used to install characters in a character string, which are sometimes difficult to print otherwise. For example, you may, as a writer, want to print a line that would say:

HE SAID, "THAT WAS EASY."

If you recall at the start of this chapter, when you have a string, it is usually defined by a double quote at the beginning and a double quote at the end. It is, therefore, difficult to find a way to make a double quote as part of a string itself. You can do this, however, using the CHR\$ function. Taking the example line as something you want to print:

```
10 DIM A$(50)
20 A$ = "THAT WAS EASY."
```

(Note that if this is printed directly, the quotes disappear.)

```
30 PRINT "HE SAID, ";CHR$(34);A$;CHR$(34)
```

If this program is RUN, the results will be the sentence shown earlier. CHR\$(34) is the double-quote character.

You can find this, and other characters you can produce using the CHR\$ function, in Appendix C of your ATARI BASIC Reference Manual. The character produced is in the right-hand column, and the decimal number you give to the CHR\$ function is listed in the leftmost column. You will find the double quotes listed at decimal 34 of the table.

One additional thing you should know about the STR\$ and CHR\$ functions is the warning given in your ATARI BASIC Reference Manual. It is that in logical comparisons, do *not* use more than one STR\$ or CHR\$ in a logic equation. This means, for example, don't make an equation that says this:

```
IF STR$(N) > STR$(F) THEN GOTO 100
```

because this type of equation will not execute correctly. The reason for this is that ATARI BASIC uses a special memory area to hold the converted string value when the STR\$ function is performed. If you try to perform it twice in a single line, there will be a problem since the second use of the STR\$ function uses that same memory area. This will make the logic comparison operation result incorrect.

The way to avoid this problem is to store the results of the STR\$ (or CHR\$, same problem) somewhere before making the compare. This means that the example would be better worded this way:

```
10 DIM TEMPS(20)
20 TEMPS = STR$(N)
30 IF TEMPS > STR$(F) THEN GOTO 100
```

This example is valid because the results of the first STR\$ are already stored away when the second STR\$ is performed.

If this problem is present for a string comparison, why then is it not present for an arithmetic comparison? For example:

```
10 M = 10: N = 5
20 G = 2: H = 1
30 IF (M + N) > (G + H) THEN PRINT "GREATER"
```

If you RUN this example, you will see it print the word GREATER. If you add line 25 as follows:

```
25 G = 14
```

and RUN it again, the comparison will find two equal values and the word GREATER will not be printed. Or if you change line 25 to read G = 100 and RUN it again, then GREATER is not printed either. Why does this work and string comparisons using STR\$ don't work?

ATARI BASIC uses two temporary memory storage areas, called *accumulators*, to store the temporary results for the arithmetic type of operations. One is used for the left side of the compare operation, the other is used for the right side. Therefore, the compare can always be done correctly.

In the string operation, only one area is used for the temporary data, so the operation demonstrated in the example must be performed if you are doing a STR\$ or CHR\$ type of operation.

REVIEW OF CHAPTER 3

1. A string is a group of ATASCII characters. When defining a string variable, the characters used to define it are enclosed in double quotes ("").

2. Any ATASCII characters can be used in a printable string except the double-quote character itself. But this character may be inserted during the printing by using the CHR\$ function.

3. String variable names must end with the character "\$" (the dollar sign). If the variable is named A\$, then you pronounce its name as "A string" (not "A dollar sign").

4. You have to save space for each string using the DIM function, such as DIM A\$(1000), which will reserve 1000 spaces for character data that is part of A\$.

5. The current length of a string will be determined by the last defining statement which works on that string. You can find out the current length of the string using the LEN function.

6. When ATARI BASIC reserves space for a string or does any work on that string, it only works on the parts of the string which you specify are to be changed. Any other parts not included in the "scope" of the string usage will remain unchanged.

7. You can add onto a string just by specifying at which character position the add-on is to occur. You can look at pieces of a string just by specifying the beginning and ending character position for the group of characters you want to use. For example, position 6 *only* would be specified as A\$(6,6)—the string piece begins and ends at position 6. For another example, look at three characters beginning at position 3—A\$(3,5)—specifies a string length of three characters, numbers 3, 4, and 5.

8. For any string operation, the limits of the character positions to be used must be within the limits of the space you saved for the string using the DIM statement. Otherwise an error will occur.

9. If you specify a string and not use any parentheses, ATARI BASIC thinks you mean to use the whole string, from character position 1 through and including the length of the string. If you specify a string using one length number in parentheses, such as A\$(7), ATARI BASIC assumes that the string section you want to look at starts at character position 7 and goes to the length of the string. If you specify two numbers in the parentheses, then ATARI BASIC uses only those two positions as the character limits of the string; for example, A\$(3,6).

10. The VAL function can be used to find the value of a number that was originally entered as, or assembled internally as, a string. It is used where arithmetic must be performed on the value and where that arithmetic can only be done in the number mode, not the string mode. This can be used on a long string of data which really represents a number value.

11. The STR\$ function can be used to convert real number values into string values.

12. The ASC function can be used to convert a single string character into a number, representing the ATASCII value of that character.

13. The CHR\$ function can be used to convert a single ATASCII value into a character string character.

Designing a Program

In Chapter 3, one of the program examples contained a challenge . . . “How would *you* do this?” For that challenge, you had already been given a fair set of tools with which a program could be developed, but thus far no instructions on exactly *how* a program is developed. This chapter will give you a brief introduction to program planning.

PLANNING A PROGRAM

As you saw in Chapter 1, the real reason you have a computer is that it can do something for you. Because it works very fast, it can do many things in a short time. Because it can be programmed, it is very versatile. This section will concentrate on how you can decide how the machine should go about doing its job. To get started, let's look at the diagram in Fig. 4-1, which shows the job of the computer. This diagram basically holds true for most applications—from games, to accounting programs, control programs, and almost anything else the computer can do.

This chart can be generalized to a set of three smaller boxes, as shown in Fig. 4-2. The entire process of writing a program can be generalized to a form of the chart in Fig. 4-2.

Fig. 4-1. Flowchart showing basic job of computer.

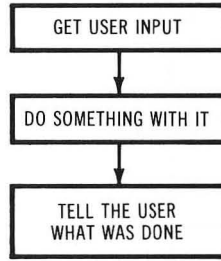
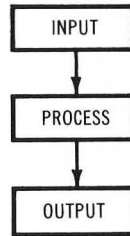


Fig. 4-2. Flowchart of Fig. 4-1 simplified.



As you are developing an application, it is not always necessary to do everything at once. It may be easier to break up the application into many small pieces. Then you may want to take each piece, make it function OK on its own, then finally "link" together all of the working pieces into a functioning program.

This technique of breaking things up and developing them individually is called *modular programming*. It is very useful because it is usually easier to work with and understand small sections of a computer program one at a time than to try to understand and get the bugs out of the whole thing at one time.

How then does the diagram in Fig. 4-2 apply to modular programming? Well, the block called INPUT can refer to the group of program pieces which lead into a particular module. The block called OUTPUT can refer to the group of program pieces which follow a particular module. So the program modules, once assembled, would look like Fig. 4-3. If you consider the output of one module to be the input of another, then the chart would look like Fig. 4-4. What you see here is a very simple version of a *flowchart*. You will see other flowcharts used in this text, just to get you used to organizing your programs properly.

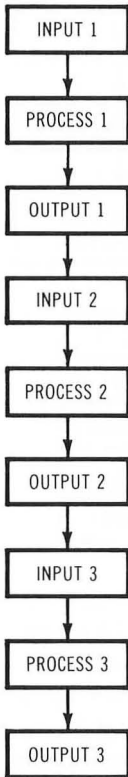


Fig. 4-3. Flowchart illustrating modular programming.

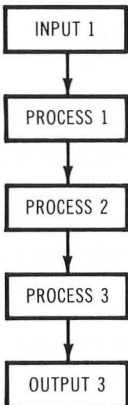


Fig. 4-4. Flowchart of Fig. 4-3 simplified.

In some cases, you will also see what could be called *flow narratives* for some programs. A flow narrative is a description of the flow of the program and might be used to substitute for the flowchart. Flow narratives are also used to emphasize the need for organizing things properly before trying to do the final program.

The brief drawing shown in Fig. 4-2 doesn't take into account that the process has to provide for decision making in most applications. You have been shown a decision-making tool, the IF statement, in Chapter 2. Now let's see how you would represent it on a flowchart.

The IF statement is represented by a diamond-shaped block, as shown in Fig. 4-5. An IF test has only one entry point (from the top), and should have only two possible exit points. One exit is for the overall result of all compound tests (many tests linked with a set of AND, OR, etc.) being true. The other exit is for the condition where one or more of the conditions tested is false, which makes the whole test false.

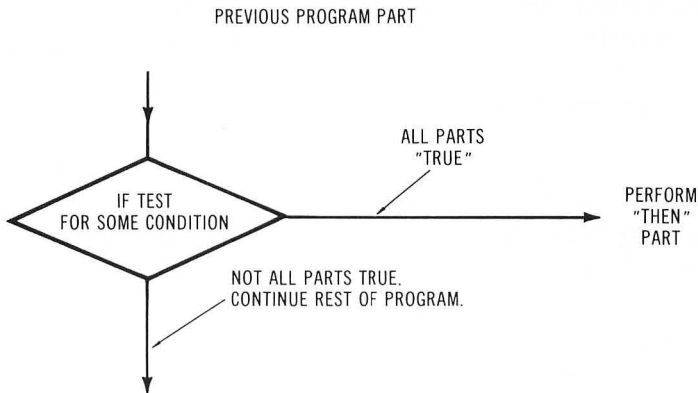


Fig. 4-5. Flowchart illustrating the IF test for decision making.

If you consider the structure of the BASIC program, the IF statement, if it fails, performs the next sequential statement (the one "directly below" it). If the test is true, the statement performs the THEN part, which is written "off to the side" of the original statement. Therefore, when you use the flowchart symbol for the

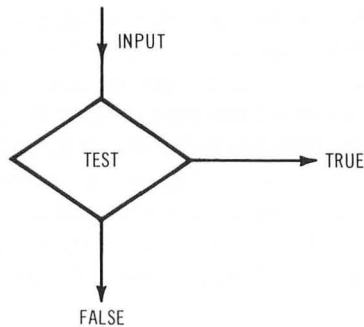


Fig. 4-6. Flowchart of Fig. 4-5 simplified.

IF statement, you should have the FALSE exit at the bottom, and the TRUE exit off to one side or the other, as shown in Fig. 4-6.

You already have a powerful set of programming tools. The rest of this book will be used to demonstrate them, and to add other tools that will make your job easier. Whenever a tool is added, you will see a comparison to the type of operation you might have had to perform if that tool had not been available. This will allow you to simplify some of the things you might already be doing where the tool is already available to make the job easier.

Just to demonstrate the effectiveness of the tools you have, let's look at a typical application and build a flowchart from it. Then from the flowchart, we will do the program.

An application most people would be familiar with is a check-book balancer. Let's look at what features it should have:

- (a) Ask for initial balance at program start.
- (b) Allow input of the following types of amounts:
 - Checks.
 - Deposits.
 - Service Charges.
 - Transfers (such as automatic overdraft protection provided through a credit-card link).
 - Withdrawals (such as credit-card-based autoteller cash access).
- (c) Give a running balance after each transaction.
- (d) Offer to do another transaction or quit.

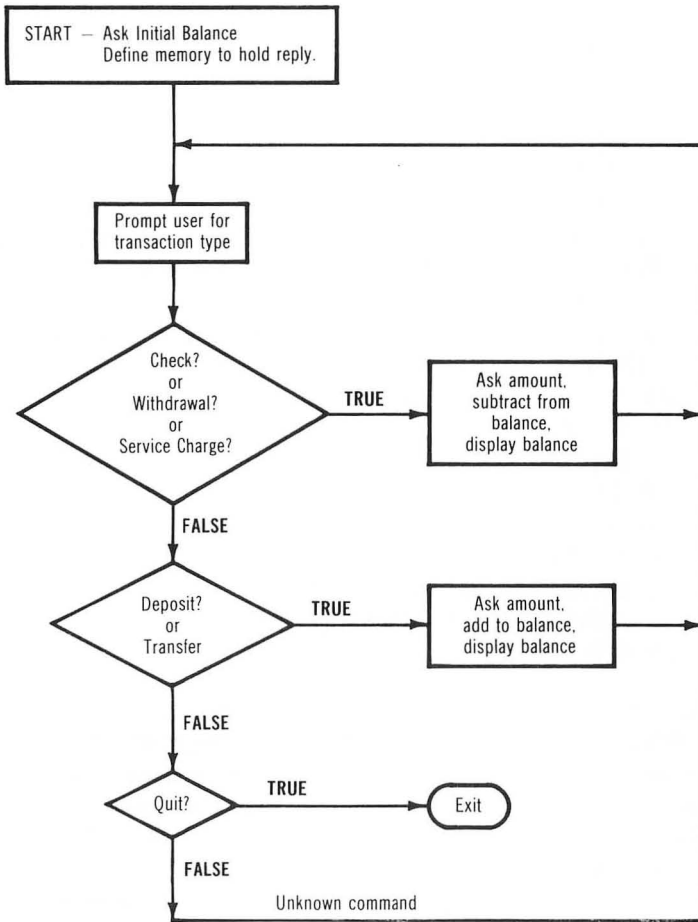


Fig. 4-7. Flowchart of checkbook balancing program.

Fig. 4-7 shows how this may be simplified on a flowchart. The structure of such a program becomes very simple once the flow has been defined:

```

10 DIM A$(20):REM SAVE SPACE FOR USER REPLY
20 PRINT "WHAT IS START BALANCE?"
30 PRINT "EXAMPLE 1234.56 THEN PRESS RETURN"

```

```

40 INPUT BALANCE
50 PRINT:PRINT
60 PRINT "TYPE FIRST LETTER, THEN RETURN"
70 PRINT "TO SELECT TRANSACTION TYPE"
80 PRINT
90 PRINT "(CHK,DEP,SRV,CHG,TRNSFR,WDRAWL,QUIT)"
100 INPUT A$
140 IF A$(1,1)="C" OR A$(1,1)="S" OR A$(1,1)="W" THEN GOTO 200
160 IF A$(1,1)="T" OR A$(1,1)="D" THEN GOTO 300
180 IF A$(1,1)="Q" THEN END
190 IF A$(1,1)<>"C" OR A$(1,1)<>"S" OR A$(1,1)<>"W" THEN
GOTO 100
200 PRINT "PLEASE SPECIFY AMOUNT"
210 INPUT AMOUNT
220 BALANCE = BALANCE - AMOUNT
230 GOTO 400
300 PRINT "PLEASE SPECIFY AMOUNT"
310 INPUT AMOUNT
320 BALANCE = BALANCE + AMOUNT
400 PRINT "CURRENT BALANCE IS: ";BALANCE
410 GOTO 50

```

This completes the program. All positive-type amounts (transfers, deposits) are added to the balance. All negative-type amounts (checks, withdrawals, service charges) are subtracted from the balance. There is a command that allows an easy exit to BASIC, the command called QUIT.

RUN the program and try it out. What happens if you give it an entry such as "JUNK" when it asks for the initial balance or any check amount? This is not a valid entry and halts the program.

Now, if you want to make the program more "bulletproof," you can install error checking here. The following TRAP statements would help this situation:

```

15 TRAP 15
55 TRAP 50

```

If you remember the TRAP statement, it tells ATARI BASIC to go to the line number specified in the TRAP if it finds an error during

the program execution. In this case, you are trying to trap bad data input. In the first case, the TRAP will spring if there is a bad input during data entry for the initial balance. In the second case, the TRAP will execute statement 50 if there is an error in the input of the amount. Each time the program passes through a loop to expect another input, the TRAP statement will be reset, ready to function again.

Now RUN the program. Any bad data input will simply ask you for the same data (correctly stated this time) to be entered.

PROGRAM SAVE AND LOAD

Now that you know how to plan your programs, you will probably be developing some of your own in the near future. As a matter of fact, you may want to save pieces of the programs you are learning from this book to use as parts of your own programs. This section will show you two different ways to save your programs to cassette or to disk, and two different ways to load them back into the computer.

If you are using the ATARI® 410™ Program Recorder as your main program storage device, you will be able to save and load programs at any time the machine is on. If you are using disk, however, you will always have to remember to TURN ON THE DISK UNIT BEFORE TURNING ON THE COMPUTER. As a reminder, when the power comes on the computer looks to see if there is a disk unit or other accessories connected to it. If it does not see them, it will not understand any commands you might give to try to talk to these accessories. Therefore, remember that the disk unit (or units) must be ON before the computer.

Make sure there is a disk in the disk unit. If you forgot to turn the disk unit on first, turn it on now. Then turn off the computer and turn it back on again. The disk unit will come on for a while, then go off again. This fixes things so that the computer will be ready to talk to the disk when you ask it to do so.

There are two different ways to save a program. One is what ATARI calls a *tokenized* form. This means the program is only readable by the machine because all of the BASIC keywords have been replaced by some numbers ATARI BASIC uses to

represent the words. This is the actual way the program appears in the memory system.

The tokenized version of the program takes up less memory than the original version where all of the keywords were spelled out. It is like a "shorthand" version of the program. When stored, the tokenized version will take up less disk or cassette memory space, and will save to and load from disk or cassette faster (less data to load or save).

Once you look at examples of loading and saving programs, you will then see examples of the other way of doing it and the advantages you might have. First, we will discuss the tokenized versions. Let's take a small program as an example. You may try any combination of ATARI BASIC statements you desire; these are just here for convenience and to prove that the programs can be stored on the cassette or disk. Here's a very short program you can practice with:

```
NEW
10 PRINT "THIS PROGRAM GOT SAVED OK"
20 PRINT
30 PRINT "AND IT RUNS OK ONCE I LOAD IT"
40 PRINT
```

You know what this program does. Now let's save it to cassette. Type CSAVE and press **RETURN**. The ATARI Home Computer will "beep" twice to tell you that it wants you to position the tape, using the forward or rewind keys, to the starting point you wish. The ATARI Program Recorder has a counter on it. If you rewind the tape all the way to the beginning, then use the fast-forward key, you can advance it to any position you desire your program to start. When you want to get your program back off the tape, you will have to position the tape to that same counter number. The ATARI Home Computer will expect the data to be there when it tries to read it.

The best kind of tape to use is a computer tape. This will usually not have any "leader" on the start or end. This allows the computer to start recording from the very beginning of the tape. Assuming you have started from tape counter position zero, and have positioned the tape to the very beginning, you can now

press and hold down the RECORD button, and then press the PLAY button. Both of them should stay down, but the tape will not yet start to record.

Now press the **RETURN** key on the computer. The tape unit will start and the program will be recorded. Once the tape has stopped, write down the tape counter value. Add about 10 counts to it, and you can use this new counter value as the position at which another program can be stored on the same tape. When the recording is complete, the computer responds:

READY

Now type NEW to erase the program in memory. Then type LIST to see that there is nothing there. Now use the REWIND key on the recorder to reposition the tape to the beginning. Type CLOAD and hit **RETURN**. The ATARI Home Computer will beep once to ask you to position the tape, then to press the PLAY button. When you press PLAY, nothing happens. Press the **RETURN** key on the computer.

Now the tape starts to run and your program will be loaded again. CLOAD means "Cassette-LOAD." Once the tape stops, type RUN to confirm that the tape loaded OK. If you hit any error conditions, check your ATARI Program Recorder manual or the front or back pages of your ATARI BASIC Reference Manual to see what they mean. If you have followed instructions, there should be no errors.

If you are going to save more than one program on a tape, you may want to name your programs to prevent the machine from loading something unless the names match. This is a different form of cassette save and load, which is not compatible with the first kind. In other words, if you save something with a CSAVE, you must load it again with a CLOAD, and not with the way you will now see.

Saving "Named" Programs on Cassette

Assuming you have a program now in the memory that you want to save (maybe the example showed earlier), you may type:

```
SAVE "C:TESTNAME"
```

Again you will hear two beeps, asking you to position the tape and press the RECORD and PLAY buttons together. Again you will press **RETURN** to start the SAVE, and again the tape unit will come on, record the program, and go off.

This is still the tokenized version of the program being saved. It is just a different way to do it. The difference here is that the tape "file" now has a name. A "file" is a name given to a group of data of some kind. You will also see files described where disk operations are specified. The sequence you typed:

```
SAVE "C:TESTNAME"
```

consisted of the command SAVE plus the file specification (file-spec). The file specification consists of the device type, in this case C:, which stands for the cassette recorder, plus the file name, which was TESTNAME.

A file name may be up to eight characters long and must start with one of the letters of the alphabet. The rest of it can be any combination of letters and numbers. ATARI BASIC also allows you to add something onto the file name. This is called the *extension* and it is just another way you can tell the difference between different types of data files.

The extension is formed by having a period or dot (.) following the eight (or less) characters of the name, then using up to three characters for the extension. The period is not actually a part of the extension, it is only the separator between the name and the extension. Here are some examples:

```
TESTNAME.BAS  
MYJOB.ASM  
LAST.OBJ
```

The extensions in the examples are called "BAS" for BASIC programs, "ASM" for assembly programs, and "OBJ" for special files called *executable* files. If you are interested in assembly language programming, you will see these last two types of files many times. These names are only suggestions; ATARI BASIC does not require that you use the extensions, it only makes your job easier to have them available.

The quotation mark (") is there to tell ATARI BASIC that there is

a name of a filespec coming up, and that it should get ready to send that filespec to the cassette handler (or the disk handler if a disk is being used).

Saving “Named” Programs on Disk

The same type of program save can be done using the disk units. In that case, you will issue the command:

```
SAVE "D:TESTNAME"
```

In this case, the device name is D:. This automatically assumes disk unit number 1. If you have more than one disk unit and you want to save on a unit other than number 1, you must tell it which one to use by specifying the unit number as part of the device name, such as:

```
SAVE "D2:TEST2.BAS"
```

Again, all of these saves will be done using the tokenized version of the file.

Loading “Named” Files from Cassette or Disk

The command required here is:

```
LOAD "C:TESTNAME"
```

if cassette, or:

```
LOAD "D:TESTNAME"
```

if disk unit number 1. You may try any BASIC program to practice this SAVE and LOAD. Then type NEW to erase it from memory, LOAD it again from the disk, and RUN it to be sure it does come back, or LIST it to see it again on the screen.

Listing Programs to Tape or Disk

There is another way you can save your programs to tape or disk. This is to LIST them on the tape or disk file. Why? Well, the program will not be saved in a tokenized form, so it will be more easily readable. But that is not the real advantage, as you will soon see. The advantage of listing your programs to tape or disk is that they can be entered later from the same device. What's so

special about the ENTER command? IT DOES NOT ERASE ANY PROGRAM CURRENTLY IN THE MEMORY. Instead, it treats the data coming in from the tape or disk as though you were entering it from the keyboard! This is just like moving a program into memory, then changing it the way you want, then running it in its customized form. Just to make this very clear, let's look at an example:

Let's say you have a title block . . . a group of BASIC statements you will always want to have as part of every program. It could include a copyright notice or just your name and address such as:

```
2 PRINT "I WROTE THIS PROGRAM AND"
4 PRINT "IT IS OK FOR PEOPLE TO"
6 PRINT "COPY IT AS LONG AS THIS"
8 PRINT "NOTICE STAYS WITH IT"
10 PRINT "C - 1983 J. SMITH"
```

Now those lines may be a real pain to type each time you write a new program, especially if they are much longer than this particular batch of lines. Let's assume you have placed these lines on a disk file by using the command:

```
LIST "D:MYTITLE.BAS"
```

Notice that this time the command line did not use the final quote mark. If you are using a direct command entry, and if this is the last item on the line where the direct command is entered, ATARI BASIC will not care if you use the final quote mark. It will automatically terminate the string you have entered when it sees the end-of-line character on the screen. The end-of-line will have been entered automatically when you hit the **RETURN** key.

(As a side note here, when you are writing a program you can, if you wish, leave out the ending quote mark at the end of a string definition or a PRINT statement, but it is not a good idea. For example, the line:

```
10 PRINT "THIS USES THE LAST QUOTE":GOTO 50
```

will execute correctly, but the line:

```
10 PRINT "THIS ONE LEAVES IT OFF :GOTO 50
```

will never go to line 50 because it will think that the part saying GOTO 50 is part of the string.)

Now let's return to the subject. If you issued the preceding LIST command, the data will be listed to the disk file named:

MYTITLE.BAS

Now, if all of the line numbers in that file were below line number 100, you could write any other program you wanted using no line number below 100, then add the title block to the program just by typing the command:

ENTER "D:MYTITLE.BAS"

with your new program already in the memory.

What this will do is leave the current program alone in memory and read the MYTITLE.BAS file as though you were typing it in yourself. As you may remember, if you type a line number that already exists, whatever you type will replace the earlier version of that line number. Or if you type a new line number, the contents of that line will become a part of your program. So as each of the lines of MYTITLE.BAS is read, if your new program has any of the same line numbers, they will be replaced by those in MYTITLE.BAS. When the file reading is complete, the program will consist of a combination of the lines from both files. In this way, you can easily add your title block to all of your programs.

Of course, this is not just limited to title blocks. If you have developed a set of instructions for your programs that can be used in common for all programs, such as a way of presenting a menu, for example, then this set of instructions may be used in the same way. When you get to the chapter titled "Menu Please," you will see a set of ATARI BASIC instructions that have been deliberately structured in this way for your convenience, if you care to use them in your own programs.

You have seen that the SAVE, LOAD, ENTER, and LIST instructions are all structured as:

(instruction) "D:NAME"

where the filespec is in quotes. This means that the filespec is a string. Therefore, this also means that ATARI BASIC will accept

the name of a string in this command in place of the quoted part. For example, if you have the program we've been using as an example (the four print statements), instead of the SAVE or LIST instructions you practiced with, you could have entered the following in direct mode:

```
DIM A$(20)
A$ = "D:TESTNAME"
SAVE A$
```

and the file would have been saved in the same way.

The next chapter covers ways to find data other than always asking the user to type it in. You will see more uses for the disk or cassette in that chapter as well. They are good for other things besides the programs. The next chapter is titled "Pulling Data Out of Different Bags" and that's just what you will be doing.

Once a program has been stored on disk using the SAVE command, there is one other way to get it back into memory and start it going. This way is to use the extended version of the RUN command that is structured just like the LOAD command, and looks as follows:

```
RUN "D:MYFILE"
```

The RUN command acts just like the LOAD command. It erases any program that might currently be in memory. Then it does the same thing that the LOAD command does. Finally, it starts the program at the lowest line number you have provided in the program.

The RUN command is normally used in direct mode, when you are deciding which program to do. But it is OK to use the RUN command from within a program also.

The type of program that would use the RUN command is a program that may control many other programs. What this means is that you may have a disk that has many games on it, or maybe many different programs that you have written to manage the home expenses. Instead of trying to remember the exact spelling of the names you have given to each of these programs, it would be nice if you could use the machine to do this for you.

Here are some examples of some programs you might have on the disk:

CHEKBOOK.BAS
FOODPLAN.BAS
CARSTUFFBAS
BOOKLIST.BAS

When you turn on the disk unit, then the computer, the disk unit runs for a while, then shuts off. Then the screen presents you with the word READY. This does not really remind you which programs you have on the disk, or tell you anything else about how to get going. Let's see what steps you would have to take normally, then you will be shown how to do a program that would save you some work.

Those of you who only have cassette recorders can skip this part if you wish and go on to the next chapter. Unfortunately, the cassette does not have a Disk Operating System, or DOS as it is called, so the things discussed here will not be helpful to cassette users.

If you are already familiar with ATARI DOS, you can skip forward to the section titled "Making a Program Selection" for a continuation of this discussion.

INTRODUCTION TO DOS

The ATARI Disk Operating System is introduced here to allow the first-time user enough experience so that he or she can follow along with the rest of the examples in the book.

Turn off the computer. Insert your ATARI MASTER DISKETTE into disk drive 1. (If you have more than one drive, make sure that you have set the addressing switches differently, and that one of them is set to address 1.) See your ATARI DOS Manual for more data on disk unit addressing. Now turn on the disk drive, then the computer. The disk unit turns on for a while, then off again. Now the screen says "READY." Type the command DOS, then hit **RETURN**. The disk comes on again and soon provides you with a menu showing the different kinds of things it can do, using one-

letter commands. In the chapter titled "Menu Please," you will see how to form your own selections like this. For now, though, let's just look at a couple of the commands and how they can be used.

The command "A" in ATARI DOS 1 and 2 selects the directory command. This will list the contents of the disk. The example which led into this discussion reminded you that you need to know what is on the disk before you can RUN any of its programs. This command would normally be used here. Type A, then hit **RETURN**. The screen will display the following data:

DIRECTORY--SEARCH SPEC, LIST FILE?

At this point we are not interested in anything except a complete list of what is on the disk. If you want more information about this command, please refer to your ATARI DOS Manual. For now, just type

D1:

then **RETURN**. Now the screen shows the following:

```
DOS          SYS 039
DUP          SYS 042
AUTORUN     SYS 001
625 FREE SECTORS
```

(This is the display for ATARI DOS 2.0S; your display may differ slightly.)

What this tells you is the name of each of the files on the disk, which are DOS.SYS, DUP.SYS, and AUTORUN.SYS, but it doesn't show the period between the parts of the name.

As you may recall earlier in the text, the name can be up to eight characters long; then, after a period, the name can have an "extension" which gives a clue as to what kind of file it is. In this case, the extension is called SYS meaning a "system" file, one that is needed to make DOS work.

The numbers next to the names tell you how many "sectors" each of these files occupies on the disk. A sector is an area on the disk which can be compared to a mailbox. Each sector, like

a mailbox, can only be stuffed with a certain maximum amount of data. Each disk can be compared to a post office, having a maximum number of mailboxes that can be used.

A new disk, when it is "formatted," will have available for use a maximum of 707 free sectors. This is comparable to putting up a post office, installing a number of mailboxes, then going around to each to see how good a job the construction crew has done.

In some cases, the boxes will have been damaged and cannot be used to store mail at all. In this case, the postmaster will not allow any customers to rent these boxes and will put a sign on them saying they are not available.

ATARI DOS does the same thing for disk sectors. After it writes data on a disk for the first time, it will look at the whole disk carefully, trying to find out if it can use all of the sectors on the disk. If not, it will mark them in its directory as unusable. This is an indication of the quality of the disk, and it can tell you (if there are many sectors marked unusable) that maybe you should not use this disk to store your programs or data.

Now it is time to make up a working disk to use for program saving. The screen menu of ATARI DOS now says:

SELECT ITEM OR **RETURN** FOR MENU

Press **RETURN**, then get a blank disk. You will prepare it for use now. Notice the difference between the blank disk and the MASTER DISKETTE. The blank has a little notch taken out of the side of it. This is called the *write-protect* notch. The ATARI Disk Drive cannot write on a disk if this notch is not there or is covered with a piece of opaque tape. After you develop some of your own programs, you may want to write-protect your program disk to prevent a program or data from being altered. To do this you will put a piece of opaque tape over the notch.

Now you have the blank disk. Remove the MASTER and insert the blank. Close the door carefully as always. Now select the item called "FORMAT DISK." This is item 1 of ATARI DOS 2.0S. Then press **RETURN**. ATARI DOS asks:

WHICH DRIVE TO FORMAT?

Type:

1 RETURN

or

D1 RETURN

or

D1: RETURN

(ATARI DOS 2.0S is very forgiving here and will accept any of these inputs.) ATARI DOS then replies:

TYPE "Y" TO FORMAT DRIVE 1

ATARI DOS is asking you if you are really sure you want to do this because *formatting will erase all data you already have on the disk*. If you started with a blank disk here, type Y, then **RETURN**. Otherwise, you may first want to check the directory to see if the disk is really blank or if there is still something on it you want to save.

If you typed Y, the ATARI Disk Drive will click and spin for a while, then return control to DOS. What it did was to entirely rewrite the disk contents on every track, then check to see how good the disk was. Again, using the post office example, ATARI DOS writes the equivalent of a box number (sector address) onto the various sections of the disk, and stores some data in each of those sections. Then it looks back to see if it can read the data it put there. This is why you will hear the ATARI Disk Drive take 40 fast steps (it is writing the data then), followed by 40 slow steps (when it tries to find and read the data in each sector).

Before you try to use the disk, you should do a directory list (item A of ATARI DOS 2.0S) on this blank disk. The screen should show 707 FREE SECTORS. If it does not, ATARI DOS found some of the sectors were not good. This may mean a flaw or a scratch on the disk and may be an indication of future trouble.

The primary concern here would be the total number of sectors you have available for your data, because ATARI DOS *can* use a slightly scratched disk, if the scratch happened *before* the data was put there. What this means is that when ATARI DOS

writes any data onto the disk, unless it is told otherwise, it will always read the disk again from each sector before going on to write another sector. As it uses each sector, ATARI DOS keeps a map showing which sectors are used and the sequence in which they are used for a data file. Generally speaking, if the data could be written there on the disk in the first place, the system was able to read it back at least once, and should still be able to read it in the future if there is no damage to the disk.

Now you have a blank *formatted* disk. If you want this disk to be the one you always start with for your programming, you must also put the DOS files on it. They take up some room, of course, but if you don't want to have to swap disks (put in MASTER, turn on system, take out MASTER, put in program disk, then go), you should have the DOS files on it. To do this, select the item that says:

WRITE DOS FILES

and press **RETURN**. ATARI DOS 2.0S will say:

DRIVE TO WRITE DOS FILES TO?

Type 1 (or D1 or D1:) then **RETURN**.

Again, ATARI DOS gives you a second chance, as in the FORMAT command, with the statement:

TYPE "Y" TO WRITE DOS TO DRIVE 1

Type Y, then **RETURN**. This will write DOS.SYS and DUP.SYS to the newly formatted disk.

MAKING A PROGRAM SELECTION

Those of you who were already familiar with ATARI DOS came here directly. Those of you who needed some introduction now have a blank formatted disk ready to use. If you are still in DOS, type B **RETURN**. This will run the ATARI BASIC language. The screen will clear and the computer will respond "READY."

Just so that you have some programs on the disk with which to practice, try the following. (If you already have some BASIC pro-

grams on your disk, you could use their names instead of these, but new users can practice with these examples.) Type the following lines:

```
5 DIM A$(10)
10 PRINT "THIS IS PART OF CHEKBOOK"
20 PRINT "CONTINUE OR END"
30 INPUT A$
40 IF A$(1,1) = "C" THEN RUN "D:SELECT"
50 IF A$(1,1) = "E" THEN END
60 GOTO 20
SAVE "D:CHEKBOOK"
```

Now type:

```
10 PRINT "THIS IS PART OF FOODPLAN"
SAVE "D:FOODPLAN.BAS"
```

Now type:

```
LIST 10
```

Use the control-arrow keys to move onto the F of the word FOODPLAN and type over it with the word CARSTUFF and hit **RETURN**. Remember, use the Screen Editor if it can save you some work.

Now, if the SAVE command is still visible on the screen, use the control-arrow keys to move onto the word FOODPLAN in that line and change it also to read CARSTUFF, then hit **RETURN**. Notice that the Screen Editor works with the SAVE also.

Do the sequence of changing line 10 and save the changed program for the word BOOKLIST also. Now you have four dummy programs on the disk, with the names CHEKBOOK, FOODPLAN, CARSTUFF, and BOOKLIST. They don't do much, but they will at least show you that the program that follows can be used to call each program individually. Your own program names will be used here instead, when you have done them.

This program must also be typed in, just as shown. It is the program master which will control the running of all of the others. The "flow narrative" for this program (used here in place of a flowchart) describes what the program should do.

Flow Narrative:

1. Set up any initial conditions.
2. Ask the user what program to RUN.
3. Present the possibilities.
4. Accept the input.
5. Is it first choice? If so, RUN it.
6. If not, is it second choice? If so, RUN it.
7. Same for all choices possible.
8. If command not found, go back to Step 2.

NEW

```

10 DIM A$(10),B$(1)
20 PRINT "WHICH PROGRAM DO YOU WANT?"
30 PRINT "TYPE FIRST 2 LETTERS, THEN"
40 PRINT "HIT RETURN"
50 PRINT
60 PRINT "PROGRAM CHOICE: ";INPUT A$
70 B$ = A$(1,1):REM SHORTER COMPARE IN NEXT LINES
80 IF A$(1,2) = "CH" THEN RUN "D:CHEKBOOK"
90 IF B$ = "F" THEN RUN "D:FOODPLAN.BAS"
100 IF A$(1,2) = "CA" THEN RUN "D:CARSTUFFBAS"
110 IF B$ = "B" THEN RUN "D:BOOKLIST.BAS"
120 PRINT "NO SUCH CHOICE FOUND":GOTO 20

```

Now save this program on the same disk with the command:

```
SAVE "D:SELECT"
```

If it saved OK, ATARI BASIC will return with the word READY. Now let's simulate what will happen when you sit down fresh at the keyboard. If this is the disk you have initialized to include DOS files as well as your BASIC files, when you insert the disk, turn on the ATARI Disk Drive, then turn on the computer, the word READY will appear. Now you can type:

```
RUN "D:SELECT"
```

and hit **RETURN**. You are presented with the lines from the SELECT program asking what program to do. Follow its instructions. You should see the disk unit come on, and the new pro-

gram you selected will replace the SELECT program and begin to run.

Each of the sample programs you saved allows you to type the first letter "C" for continue, or "E" for END. A continue instruction here tells the system to RUN the SELECT program again. This is known as linking programs together, where one program tells the computer to RUN another. You will see more of this in "Menu Please."

REVIEW OF CHAPTER 4

1. You can plan a program in advance. All it takes is to break down the application into a sequence of small steps consisting of INPUT, PROCESS, DECISION, and OUTPUT. Then write down the steps, and do the program using them as a guide.

2. If you are using ATARI DOS, you can prepare a new disk for program storage using the FORMAT disk command.

3. Then you can store programs on the disk using either the SAVE"D:NAME" or LIST "D:NAME" commands. The SAVE command stores the program in a special way that requires the computer to erase anything in memory while it is being LOADED or RUN. The LIST command saves the program in a form similar to direct typing, and does not destroy current memory contents when it is ENTERED back into the machine using the command ENTER"D:NAME".

4. To get back a program you saved using the SAVE command, you can either LOAD or RUN it, using the commands LOAD"D:NAME" or RUN"D:NAME".

5. To save a program on cassette, you must position the tape and remember the count at which the program is to begin, then issue the command CSAVE or the command SAVE"C:NAME". The computer beeps twice to tell you to push the RECORD and PLAY buttons together, then to push **RETURN** to record the program.

6. To load a program from cassette, you must position the tape to the same count at which it was started for the save, then issue the command CLOAD if it was saved using a CSAVE, or LOAD"C:NAME" if it was saved using a SAVE"C . . ." command.

The two different kinds of save command are not compatible . . . you cannot load one kind if it was saved using the other kind of command. The computer beeps once to ask you to push the PLAY button, then to press **RETURN** to load the program.

7. The LIST and ENTER commands can also be used with the tape unit.

CHAPTER

5

Pulling Data Out of Different Bags

In the preceding chapters, you were given enough BASIC tools to do most any of the straightforward tasks you might encounter. This chapter, and those that follow, will give you additional tools that will make your programming job easier or, in some cases, just more “effective.”

THE DIM STATEMENT

In the earlier chapters, you have seen the DIM statement used primarily as a way to save room for character strings as:

```
DIM A$(20)
```

which means reserve 20 places for a character string known as A\$ (remember “A string” as the pronunciation). This DIM statement also provides a way to define “arrays” of numbers.

An array is a group of items that may be related in some way. If you are performing some type of repeated operation on a group of items, it would be very tiring to have to write a program to refer once to each one of them. Such an example follows.

Let’s say you had 26 numbers to add together. You could choose to assign each one of the numbers to a different variable name;

for example, A for the first one, B for the second one, C for the third, and so on. But your program would become very long to write and you would probably haul out a calculator or just use a calculator-like program instead of writing your own. Let's see why by looking at what you might have written:

```

10 PRINT "WHAT IS VALUE OF A"
20 INPUT A
30 INPUT "WHAT IS VALUE OF B"
40 INPUT B
.
.
.
400 TOTAL = A + B + C . . . + Z
410 PRINT "TOTAL IS: ";TOTAL

```

Lines 10 and 20, lines 30 and 40, and so forth, are very much the same, and they look just like something that would be better done in some kind of repeating "loop" (you will see more about loops as this chapter progresses). But you can see that lines 20 and 40 are different because of the choice of the variable name.

Here is where the DIM statement can help you. Instead of giving the numbers different names all the way down, let's give the numbers an "index," which means a position within an array. Here is how to do it:

```
10 DIM NUMBER (26)
```

This statement will reserve a space in your ATARI Home Computer for 26 elements called NUMBER. Each one of the elements is going to be a number of some kind because the name used here does not end in a dollar sign (\$), so ATARI BASIC knows these are not string elements.

If you want to refer to any individual element in the array, you must specify which element it is by telling ATARI BASIC the index number. You do this as follows:

- First element in the array is: NUMBER(1)
- Tenth element in the array is: NUMBER(10)
- Last element in the array is: NUMBER(26)

The index is placed in the parentheses behind the array name and that tells ATARI BASIC which one to use.

Now, using this knowledge, let's take another look at the way the total program would be structured. (Note: You may remember that you've already seen a total-getting program in an earlier chapter . . . the main difference here is that each of the data entries is being saved along the way. There is a reason for this, and you will see it soon.) The flow narrative for this program would then be:

1. Ask for a value.
2. Put it away.
3. Add it to the total.
4. Is it the ending value? If not, go back to 1.

First we will look at how a program would be made up to do this flow narrative, then we will add error trapping. Then we will go on to other statement types which might be able to make the job easier. Here's a program piece that will do it:

```

10 DIM NUMBER (100)
20 REM SAVE SPACE FOR 100 NUMBERS
30 TOTAL = 0
40 INDEX = 1
45 REM START AT FIRST LOCATION
50 PRINT "INPUT ENTRY # "; INDEX
60 INPUT ZZ:NUMBER(INDEX) = ZZ
70 IF NUMBER(INDEX) = 9999 THEN GOTO 500
80 REM LINE 70 USES 9999 AS END CONDITION
90 IF INDEX > 100 THEN GOTO 500
100 REM LINE 90 ONLY ALLOWS 100 ENTRIES
105 TOTAL = TOTAL + ZZ
110 INDEX = INDEX + 1:GOTO 50
500 PRINT "TOTAL IS: ";TOTAL

```

Note line 60; ATARI BASIC won't let you write a statement such as INPUT A(N). Array "equates" cannot be used in an INPUT statement.

As in previous examples, you don't have to type the REM statements if you don't want to, but they are handy sometimes to

remind you what you were trying to accomplish, especially if you come back to the program many weeks or months later. But, if you are in a crunch for program space and you need to cut out every bit of extra information, you may want to eliminate the REM statements and try to put everything on the same line, if possible. Here is another version of the same program piece that does just that:

```

10 DIM NUMBER(100):TOTAL = 0:INDEX = 1
50 PRINT "INPUT ENTRY # ";INDEX:INPUT ZZ:NUMBER
(INDEX) = ZZ:IF NUMBER(INDEX) = 9999 THEN 500
90 IF INDEX > 100 THEN 500
100 TOTAL = TOTAL + ZZ
110 INDEX = INDEX + 1:GOTO 50
500 PRINT "TOTAL IS: ";TOTAL

```

This program piece reports the total when you either enter more than 100 numbers, or when you enter 9999 as the last one.

Now that we have this part of the program, let's look at error trapping. For a number entry program, the most common error is for someone to hit the **RETURN** key without entering a number. Another common error is to have a combination of letters and numbers entered in place of just numbers. Add the following three lines to the program:

```

40 TRAP 1000
1000 PRINT "THIS PROGRAM ONLY ACCEPTS NUMBERS"
1010 GOTO 40

```

This will give you error trapping for those two kinds of errors.

Now, the program can take 100 entries and can handle some kinds of data errors. Why should you want to save the entries as well as the total? Well, suppose you are an accountant, or just trying to get a total of your checks or bills. . .it would be nice to be able to ask the machine to tell you not only the total, but also to present you with a complete list of the numbers you entered so you can be sure they are right. (The total is no better than the numbers that went into it!)

Let's add this capability to the program to allow the user to see the data entered. Add the following lines to the program:

```

5 DIM Z$(1)
600 X = 1
610 PRINT "PRESS RETURN TO SEE LIST"
620 PRINT "OF NUMBERS ENTERED"
630 INPUT Z$
640 PRINT "ENTRY # ";X;" = ";NUMBER(X)
650 X = X + 1
660 IF X > INDEX THEN END
670 IF ((X/10) - INT(X/10)) > .09 THEN 640
680 PRINT "PRESS RETURN TO SEE NEXT GROUP":GOTO 630

```

These lines allow a way for the user to view the data entries 10 at a time. The item Z\$ is entered as a string of length 0 when the **RETURN** key is hit. If you tried to use a number entry here, the **RETURN** key would have caused an error, requiring another TRAP statement.

Line 600 sets up the initial value of the array pointer (index value) for X, then lines 650 and 660 control the branching back to do more. Line 670 is a test to see if there is a remainder when you divide a number by 10. It uses the INT function to check, for example, if 41/10 (which equals 4.10) is greater than the integer value of this division (which is 4) by at least 0.09. The reason that 0.09 is used instead of seeing if the result of X/10 compared to INT(X/10) is zero is that occasionally the arithmetic values can be a little imprecise, as demonstrated in Chapter 2, "Computers Compute." So it is usually safer to compare a result to something close to zero, rather than to believe it will always be shown as zero.

Now, once the total has been established, hitting **RETURN** will display 10 of the original input values, then 10 more for each **RETURN** until all have been displayed for checking against the original input. The program now looks like this:

```

5 DIM Z$(1)
10 DIM NUMBER(100):TOTAL = 0:INDEX = 1
50 PRINT "INPUT ENTRY # ";INDEX:INPUT
ZZ:NUMBER(INDEX) = ZZ:IF NUMBER(INDEX) = 9999 THEN 500

```

```

90 IF INDEX > 100 THEN 500
100 TOTAL = TOTAL + ZZ
110 INDEX = INDEX + 1:GOTO 50
500 PRINT "TOTAL IS: ";TOTAL
600 X = 1
610 PRINT "PRESS RETURN TO SEE LIST"
620 PRINT "OF NUMBERS ENTERED"
630 INPUT Z$
640 PRINT "ENTRY # ";X;" = ";NUMBER(X)
650 X = X + 1
660 IF X > INDEX THEN END
670 IF ((X/10) - INT(X/10)) > .09 THEN 640
680 PRINT "PRESS RETURN TO SEE NEXT GROUP":GOTO 630

```

The next section of this chapter shows you how to make this program even more useful.

FOR-NEXT STATEMENTS

Until now, anytime we've tried to do anything in a repeated form, we have always set up loop control variables. This was in the form of:

```

INDEX = 1
INDEX = INDEX + 1
IF INDEX < MAXINDEX THEN GOTO (somewhere)

```

ATARI BASIC allows an easier way to express this kind of "loop" control in the form of a FOR-NEXT statement pair. Let's look at two example program pieces, one with the index variable, and the other with the FOR-NEXT loop used. The example will be just a do-nothing counter which you will occasionally use just as a time delay. Here they are:

Loop With Index Variable Control

```

300 INDEX = 1
310 INDEX = INDEX + 1
320 IF INDEX < = 10000 THEN 310

```

This will produce a time delay of about 30 seconds, so if you run it, don't think that the machine has gone bad on you.

Loop With a FOR-NEXT Statement in Control

```
400 FOR N = 1 TO 10000
410 NEXT N
```

This will also take about 30 seconds to execute, but it does the same thing in a more efficient manner. How does it work? In the case of the index control variable, each time we got to the end of the loop we had to do a test, then tell ATARI BASIC where to go next. In the case of the FOR-NEXT statement combination, ATARI BASIC already *knows* where to go.

The statements that will be executed in a loop as a result of this structure are those contained between the FOR statement, which states the starting value of the control variable, and the NEXT statement, which automatically increments and tests it. Or, in diagram fashion:

FOR (variable = initial value)

<p>statements to be executed under control of this loop</p>

NEXT (variable name)

What the FOR-NEXT loop performs in terms of a flowchart is shown in Fig. 5-1. The value of 1 is the default. This means that if nothing else is specified, ATARI BASIC will use 1 as the adjustment value. To assign an alternate value, simply add the word STEP to the FOR statement, as the following example shows:

```
500 FOR N = 10 TO 100 STEP 10
510 PRINT N
520 NEXT N
```

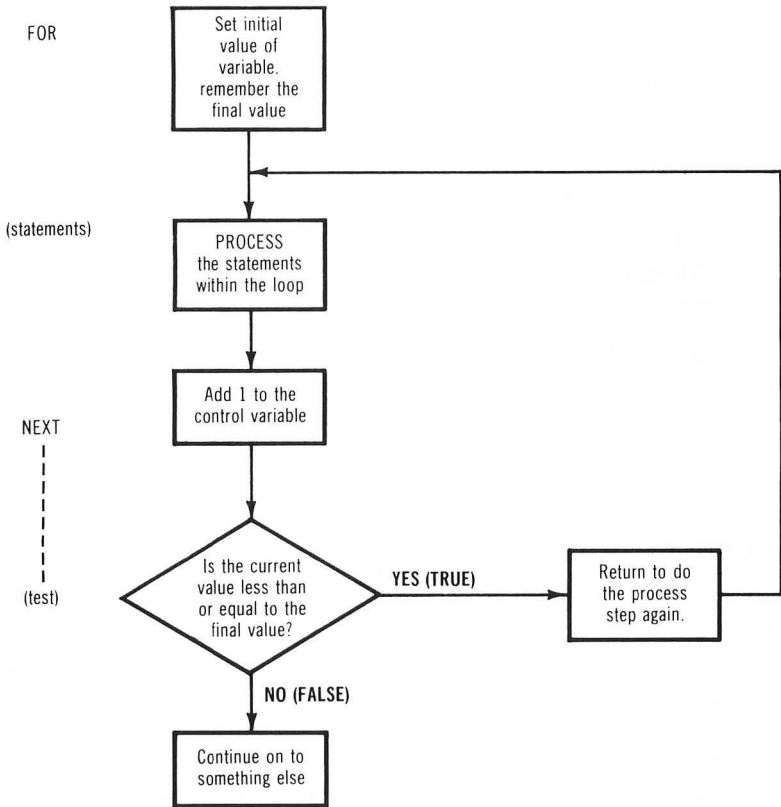


Fig. 5-1. Flowchart of FOR-NEXT loop.

This allows you to step the N value by 10 each time it performs all of the steps between the FOR and NEXT statements. This is a bit simpler than setting up the index and increment/test arrangement.

What about the case where you want to go down in value? You just specify a high starting value, a low ending value, and a negative step value, such as:

```
600 FOR N = 100 TO 10 STEP - 10
610 PRINT N
620 NEXT N
```

The last case is where you don't want to use an integer value (whole number) for the increment. Again, just tell ATARI BASIC that the step value is a real number, as follows:

```
700 FOR N = 3.47 TO 4.12 STEP 1.2E - 3
710 PRINT N
720 NEXT N
```

In each of these last three cases, the word NEXT causes ATARI BASIC to recall what the next STEP value is supposed to be. It then uses that for the value change, and uses the specified final value for the comparison.

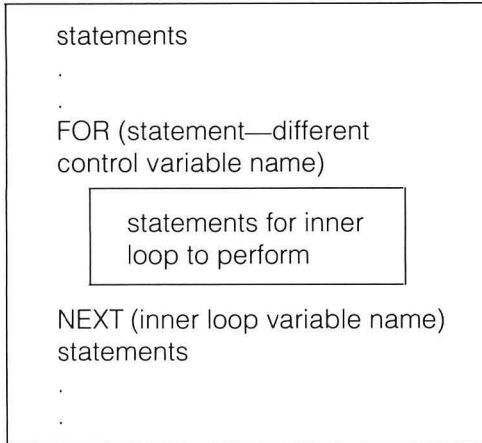
NESTING LOOPS

Sometimes you will want to do something to a group of things such as an array. And you will want to do that operation to both the rows and columns of the array. You may want to set up a loop of some kind that would set the row value, then have what is called an *inner loop* which would perform the function on all of the columns before the outer loop incremented to the next row. Or you may simply be handling a single array of some kind and just want to split up the handling of it in some convenient manner so that, for example, only 10 items are printed out at a time.

We have an excellent example of that case in the last program we did just ahead of the FOR-NEXT explanation. But before we get into how a nested loop can help this kind of operation, let's just look at the general rules that should be applied to nested loops. These are given in two forms, one a set of diagrams, the other a set of statements, both illustrating the same rules.

The following diagram illustrates that the statements comprising the "body" of a FOR-NEXT loop should be totally enclosed within the FOR-NEXT pair. Another FOR-NEXT pair can be "nested" within the first one if a different control variable name is selected for it. These loops can be nested to any number you wish to use.

FOR (statement)



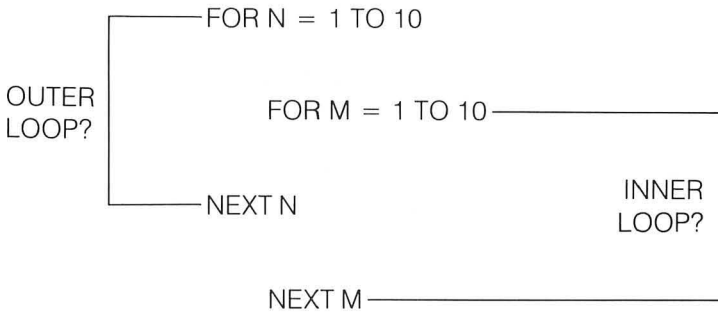
NEXT (outer loop variable name)

Another way to illustrate this is by the use of a set of word diagrams, which can illustrate multiple levels of nesting. The indentations from the left-hand margin indicate at what level you might be, and the control variables are named N1, N2A, N2B, N3, and so forth, to show the nesting level:

```

FOR N1 = START1 TO END1
  statements
  FOR N2A = START2A TO END 2A
    statements
  NEXT N2A
  statements
  FOR N2B = START2B TO END 2B
    statements
    FOR N3 = START3 TO END3
      statements
    NEXT N3
  statements
NEXT N2B
NEXT N1
    
```

What you *don't* want to do is construct a set of nested loops so that they overlap each other, as is illustrated in the following diagram:



You don't want to do this because the computer will become confused and will execute the entire outer loop first, for all initial values of M, then will hang up with an ERROR 13 because you tried to increase the outer loop value beyond its ending point. Try the following *bad example* of nested loops just to see why it is so important to do it right:

Problem: To generate a list of the first 100 numbers of a multiplication table.

Wrong Way:

```

5 HOWMANY = 0
10 FOR N = 1 TO 10
20 FOR M = 1 TO 10
30 PRINT M;" TIMES ";N;" = ";M*N
35 HOWMANY = HOWMANY + 1
40 NEXT N
50 NEXT M
60 PRINT "I HAVE PRINTED ";HOWMANY;" LINES."
  
```

RUN this and see what you get.

Right Way: Same as before, except create the loop nesting correctly by changing lines 40 and 50 as follows:

```
40 NEXT M
50 NEXT N
```

Now RUN it again. You can see how important it is that your loops are nested correctly.

MORE ABOUT THE ACCOUNTANT

Before the FOR-NEXT loops were explained, you saw an example program for helping an accountant balance his or her books. This example will continue here, showing how the FOR-NEXT loop can be used to build pieces of a helper program.

The way the program was before, it would be necessary to RUN it twice in order to get a pair of adding-machine-type lists of numbers to check. Let's look at a way to avoid running it twice.

If there are to be two sets of entries, with 100 numbers per entry, the addition process could be done once for each set of entries. Likewise, the presentation process should be done once for each. It would be possible to define two different arrays of numbers, such as:

```
DIM NUMBERSET1(100), NUMBERSET2(100)
```

But this would cause you to have to write at least part of the program statements twice, once for each number set. Let's change line 10 of the original program to read:

```
10 DIM NUMBER(200):TOTAL = 0:INDEX = 1
```

Now you can treat the first 100 numbers as part of number group 1 and the second 100 numbers as part of number group 2. If you want to work with either the first group or the second group, all you have to do is change the way ATARI BASIC points to which one of the values it will use. Look at the following example:

```
NUMBER(INDEX + OFFSET) = ZZ
```

This is a changed version of line 50 of the original program.

The number value in the parentheses (INDEX + OFFSET) will specify which of the values in the array NUMBER will be used. If the value of OFFSET is zero, then the program will run exactly as it did before, using the first 100 positions of the NUMBER array to store the results. However, if the value of the item called OFFSET is changed to 100, then the positions from NUMBER(101) through NUMBER(200) will be used throughout the program. Notice, though, that line 640 would have to be changed to read:

```
640 PRINT "ENTRY # ";X + OFFSET;" = "; NUMBER (X + OFFSET)
```

To set up for this, let's ask the accountant which one of the entries he is working on, this way:

```
30 OFFSET = 0
35 PRINT "WHICH NUMBER GROUP IS THIS? (1,2)";
38 INPUT Z$:IF Z$(1,1)="2" THEN OFFSET = 100
```

Then, to use this information correctly, here is the rest of the beginning part of the program:

```
3 TRAP 500:REM PRINT TOTAL IF ONLY HIT RETURN
5 DIM Y$(1),Z$(1), CHECK (200), NUMBER (200)
40 TOTAL = 0:INDEX = 1
50 PRINT "INPUT A NUMBER ";:INPUT ZZ
55 NUMBER(INDEX + OFFSET) = ZZ
60 IF NUMBER(INDEX + OFFSET) = 9999 THEN 500
90 IF INDEX > 100 THEN 500
100 TOTAL = TOTAL + ZZ
110 INDEX = INDEX + 1:GOTO 50
500 PRINT "TOTAL IS: ";TOTAL
501 TRAP 500:IF INDEX = 1 THEN 30
```

The FOR-NEXT loops can help in other ways to make the bottom part of this program a little less complicated. Let's see how. Here is the FOR-NEXT version of the data print part:

```
600 X = 1
610 PRINT "PRESS RETURN TO SEE LIST"
620 PRINT "OF NUMBERS ENTERED"
```

```

630 INPUT Y$
640 FOR OUTERLOOP = 1 TO 10
643 FOR INNERLOOP = 1 TO 10
646 PRINT "ENTRY # ";X + OFFSET;" = ";NUMBER(X + OFFSET)
650 X = X + 1:IF X = INDEX THEN GOTO 680
655 NEXT INNERLOOP
660 PRINT "PRESS RETURN TO SEE NEXT GROUP"
670 INPUT Y$:NEXT OUTERLOOP

```

This version of the program is more useful because it can handle two different entry groups independently. On completion of a set of entries, the total is given and the list of entries can be seen, 10 at a time. Look at the FOR-NEXT part in this program segment. Notice the correct nesting of the inner loop inside of the outer loop.

This program uses an extra counter, called X, which is used as a pointer into the NUMBER array. It is also used to list only as many entries as were made, even though the NUMBER array can hold many more. This example also shows you that it is OK to exit early from a FOR-NEXT loop if you need to.

Now look again at statement number 5. There is an extra array mentioned there. It is called CHECK, and there are 200 spaces reserved for it. Why was this put here? Well, instead of having the accountant check his figures from both of the data entries, why not let the computer do it for him?

Let's look at a typical data entry session. Only the array contents will be shown, just to keep it a bit shorter. Initially, let's assume that the CHECK array is all zeros.

ARRAY POSITION	NUMBER	CHECK
1	12.56	0
2	3.01	0
3	9.22	0
(first total	24.79)	
101	12.56	0
102	9.22	0
103	3.81	0
(second total	25.59)	

Now if you were the accountant, you would use the CHECK column to go down each of the lists and find the numbers that matched. Then the nonmatching entries could be used to point out where the error might be. The result would then appear like this:

ARRAY POSITION	NUMBER	CHECK
1	12.56	1
2	3.01	0
3	9.22	1
101	12.56	1
102	9.22	1
103	3.81	0

Here, a 1 in the check column indicates that there was a matching entry somewhere in the opposite entry group. If there is a zero there after checking, there was no match and this should be reported as an error. Let's see how to add this feature to the program.

First, it is necessary to make sure the whole check array has nothing but zeros in it. Here's how to do that:

```
700 FOR V = 1 TO 200:CHECK(V)=0:NEXT V
```

Once both sets of entries have been made, one list can be used as the base, and the other list can be used to compare it against. It does not matter in which order the entries have been made, if the entire list is searched until a matching entry with no checkmark is found, each can be matched up or declared as non-matching. The flow narrative for this is:

1. Find out which array has more values in it and use that index number to search both for matches.
2. Set up a loop to look at all entries in the first array.
3. Look at an entry in the first array.
4. Set up a loop to compare it to all of the entries in the second array, one at a time. If the checkmark item for the second array is zero, and if the numbers match, then make the checkmark item for *both* arrays at the pointer locations into a 1 and exit the outer

loop (return to Step 3 for the next entry in the first array). Otherwise, compare the next entry.

5. If the entire second array has been searched with no match, leave the checkmark position as a zero for this first-column position, and go on to the next one.

6. Once all of the entries have been matched up, go through both sets looking for whichever entries have a zero still in the CHECK position and report them as possible errors.

For the moment, just assuming that the correct number of items was entered both times, let's see how to perform this test:

```

800 FOR M = 1 TO INDEX
810 FOR N = 100 TO 100 + INDEX
820 IF NUMBER(M) = NUMBER(N) AND CHECK(N) = 0 THEN GOTO
840
830 NEXT N
835 GOTO 845:REM DO NOT SET CHECK IF NOT FOUND
840 CHECK(M) = 1:CHECK(N) = 1:REM FOUND MATCH
845 NEXT M

```

This part takes care of the checking, now to add the reporting of errors:

```

900 PRINT "HERE ARE THE UNMATCHED ITEMS"
905 PRINT "FROM THE FIRST SET OF ENTRIES"
910 FOR W = 1 TO 200
920 IF W <> 100 THEN 950
930 PRINT "HERE ARE THE UNMATCHED ITEMS"
940 PRINT "FROM THE SECOND SET OF ENTRIES"
950 IF W < INDEX THEN 980
960 IF W >= INDEX AND W <= 100 THEN GOTO 990
970 IF W >= 100 + INDEX THEN 1000
980 IF CHECK(W) = 0 THEN PRINT NUMBER(W)
990 NEXT W

```

Notice how this section of the program treats the whole array called NUMBER at one time, and likewise looks at each of the checkmark columns. If there are still any zeros (missing check

marks), then there is an unmatched number in that location and it should be reported.

Can you figure out the reason why lines 950 through 970 are used? The reason is because each number entry set has 100 reserved locations, but the accountant has not necessarily made 100 entries for each. Therefore, you can only ask the program to check the entries that were actually made, namely 1 through INDEX - 1, and 100 through 100 + INDEX - 1.

Finally, you have to know how to patch in the checking part of the program so it can be used. This is done by adding lines 680 through 690:

```
680 IF Z$(1,1)<>"2" THEN 30:REM CHECK 2ND ONLY
685 PRINT "DO YOU WANT TO CHECK ENTRIES (Y/N)"
690 INPUT Y$:IF Y$(1,1) <> "Y" THEN GOTO 30
```

For those of you who want to try this program, the whole text of the program developed is collected here with a couple of error traps added:

```
1 REM THE ACCOUNTANT'S HELPER
3 TRAP 500:REM RETURN ONLY GETS TOTAL
5 DIM Y$(1),Z$(1), CHECK(200), NUMBER(200)
10 TOTAL = 0:INDEX = 1
30 OFFSET = 0
35 PRINT "WHICH NUMBER GROUP IS THIS? (1,2)";
38 INPUT Z$:IF Z$(1,1) = "2" THEN OFFSET = 100
40 TOTAL = 0:INDEX = 1
50 PRINT "INPUT A NUMBER ";:INPUT ZZ
55 NUMBER(INDEX + OFFSET) = ZZ
60 IF NUMBER(INDEX + OFFSET) = 9999 THEN 500
90 IF INDEX > 100 THEN 500:REM TAKE 100 MAX
100 TOTAL = TOTAL + ZZ
110 INDEX = INDEX + 1:GOTO 50
500 PRINT "TOTAL IS: ";TOTAL
501 TRAP 500:IF INDEX = 1 THEN 30:REM NO ZERO ENTRIES
600 X = 1
610 PRINT "PRESS RETURN TO SEE LIST"
620 PRINT "OF NUMBERS ENTERED"
```

```
630 INPUT Y$
640 FOR OUTERLOOP = 1 TO 10
643 FOR INNERLOOP = 1 TO 10
646 PRINT "ENTRY # ";X + OFFSET;" = ";NUMBER(X + OFFSET)
650 X = X + 1:IF X = INDEX THEN GOTO 680
655 NEXT INNERLOOP
660 PRINT "PRESS RETURN TO SEE NEXT GROUP"
670 INPUT Y$:NEXT OUTERLOOP
680 IF Z$(1,1) <> "2" THEN 30
685 PRINT "DO YOU WANT TO CHECK LISTS (Y/N)"
690 INPUT Y$:IF Y$(1,1) <> "Y" THEN 30
700 FOR V = 1 TO 200:CHECK(V)=0:NEXT V
800 FOR M = 1 TO INDEX
810 FOR N = 100 TO 100 + INDEX
820 IF NUMBER(M) = NUMBER(N) AND CHECK(N) = 0 THEN GOTO
840
830 NEXT N
835 GOTO 845:REM DO NOT SET CHECK IF NOT FOUND
840 CHECK(M)=1:CHECK(N)=1
845 NEXT M
900 PRINT "HERE ARE THE UNMATCHED ITEMS"
905 PRINT "FROM THE FIRST SET OF ENTRIES"
910 FOR W = 1 TO 200
920 IF W <> 100 THEN 950
930 PRINT "HERE ARE THE UNMATCHED ITEMS"
940 PRINT "FROM THE SECOND SET OF ENTRIES"
950 IF W < INDEX THEN 980
960 IF W > = INDEX AND W < = 100 THEN GOTO 990
970 IF W > = 100 + INDEX THEN GOTO 1000
980 IF CHECK(W) = 0 THEN PRINT NUMBER(W)
990 NEXT W
1000 PRINT "PRESS RETURN TO SEE COMPLETE LIST"
1010 INPUT Y$
1025 PRINT "HERE IS A LIST OF EVERYTHING":PRINT
1050 PRINT "FIRST","CHECK","SECOND","CHECK"
1100 FOR J = 1 TO INDEX - 1
1110 PRINT NUMBER(J),CHECK(J),NUMBER(J + 100),CHECK(J + 100):
NEXT J
```

There are other things you can add to the program, such as different error trapping for bad data entries and saving the value of INDEX for both the first set and the second set of entries. (In the program, we just assume that the total number of entries to check is based on the last number counted for the variable called INDEX.) However, the purpose of the program was to show you how arrays can be used, along with the FOR-NEXT loops, and this program does use a few of them.

This chapter was titled "Pulling Data Out of Different Bags." One of the "bags" you saw initially was the user input. In other words, you asked the user for data. The next place from which you could pull data is the array.

THE DATA STATEMENT

Now you will see another of the ATARI BASIC statements used to provide yet another source of data for you to use. This is the DATA statement. The DATA statement provides you with a way to store data in your program, either to initialize an array (give everything a starting value) or to simply store number or character-string data for the program's use.

What is meant by initializing an array? Well, if you remember the early sections of this book where the string splitting operations were discussed, you may recall that ATARI BASIC does not store anything to any of the memory areas you might use for strings or numbers when the program starts. You just get whatever the memory has lying around in it. To refresh your memory, try the following example:

```
10 DIM A(100)
20 FOR N = 1 TO 100
30 PRINT "A(;"N;" ) = ";A(N)
40 NEXT N
```

Now RUN the program. If you have just turned on your machine, it is possible you will list all zeros. Then again, if you have been using it for a while, there is no figuring what kind of numbers will be listed.

If you did list all zeros, try this experiment afterwards. Type the direct command:

```
A(99) = 1234567
```

Then RUN the program again. Notice that the second to last line shows the value you just entered! This means that ATARI BASIC did not, on starting the program, change any of the values of the arrays. This illustrates that if you want to be sure of what values a number or an array has, *you* must initialize it before you use it.

Suppose you wanted to start an array with all zeros. That could be done like this:

```
10 DIM NUMBERS(100)
20 FOR N = 1 TO 100:NUMBERS(N) = 0:NEXT N
30 FOR N = 1 TO 100
40 PRINT NUMBERS(N)
50 NEXT N
```

The actual work, of course, is done by line 20. Line 10 was necessary so that ATARI BASIC would know how many spaces to save for the array called NUMBERS. The rest of the program is just to show us that the array did initialize correctly.

What would happen if it were necessary to initialize an array with a bunch of numbers not related to each other? Zeros were easy; so is a number progression where one number is related to the next by simple addition or something similar. Let's say, for example, that the numbers to be initialized were as follows: (Note that our program will not actually enter the following sequence; it is only for demonstration purposes.)

```
NUMBER(1) = 52
NUMBER(2) = 37
NUMBER(3) = 93
NUMBER(4) = 12
NUMBER(5) = 1
```

Since the numbers may not be related to each other in any perceptible way, but are perhaps needed by the program in that exact sequence, it would grow very tiring to have to enter a bunch of NUMBER(XX) = statements in the program just to ini-

tialize it. Even when you might use the Screen Editor to duplicate most of the information along the way, it still is difficult.

You can use the ATARI BASIC DATA statement to help you here. Using the preceding numbers as an example, the form of the DATA statement is as follows:

```
100 DATA 52,37,93,12,1
```

where each of the data elements is separated from the previous one by a comma.

To use the DATA statement in this example, another statement is used to read the data. This is the READ statement. When the system sees a READ statement, it looks for the next available DATA statement item that has not yet been read, then takes that item and places it into the variable named in the READ statement. If there has not been any READ statement performed previously, then the first data item read is the first data item in the first DATA statement the system can find.

Remember that ATARI BASIC will search from the very first line number in the program when it tries to find the DATA statements. Therefore, the first DATA item will be in the DATA statement that has the lowest line number. To illustrate how this works, try the following example:

```
20 DIM NUM(10)
30 FOR N = 1 TO 10:NUM(N)=0:NEXT N
40 FOR N = 1 TO 5:PRINT NUM(N):NEXT N
50 FOR N = 1 TO 5
60 NUM(N)=M
70 HEAD M
80 PRINT M
90 NEXT N
200 DATA 52,37,93,12,1
```

Line 30 puts zeros in the whole array. Line 40 prints out the first five elements to prove that zeros are stored there. Lines 50 through 90 pull five elements for the array out of the DATA statement, and line 80 prints them to prove that the DATA statement, with the READ statement, really works. Now change line 200 to read:

```
200 DATA 3.14, -2E21, .3, 0.0, .00004
```

and RUN the program again. This proves that the DATA statement works with whole numbers also.

Now change the program to read:

```

20 DIM A$(100)
30 FOR N = 1 TO 100:A$(N) = " ":NEXT N
40 PRINT A$
45 A$ = "":REM NO SPACES BETWEEN THE QUOTES (LEN = 0)
50 FOR N = 1 TO 5
60 READ A$
65 PRINT A$
70 NEXT N
200 DATA "STRING 1 ","STRING 2 ","STRING 3 ","S4 "
210 DATA "FINAL STRING OF 5"

```

and RUN it again. This time the DATA statement provides string variables to read instead of numbers.

For a string variable, anything can be enclosed in quotes if you wish. That is, anything other than the double-quote character. This means that even though the comma is normally used as the separator for the DATA items, it could be included in a string DATA item if it was within the pair of double quotes. So a string DATA item might look like this "FOR PARTS 1, 2, AND 3."

Now, using the preceding program, it is possible to read in a number of strings and make them all part of one large string. But, in its present form, there is no way to tell where one string piece leaves off and another begins.

One thing you might consider doing is to store a marker after each string. Then, as you try to print the string piece, count the markers as you go until you come to the one you want. But if the string gets longer and longer, the search time becomes proportionately longer!

Another alternative is to remember where each string piece starts and ends as it is added to the long string. (By the way, the reason this is being discussed here is that ATARI BASIC does not provide string arrays.) To do this memory work, first let's add the following DIM statement to the program:

```

15 DIM S(100), E(100)

```

letting S stand for the start position of the selected string variable, and E for the end position of the selected string variable. If there were only one S and one E, then a single string variable could be printed by specifying the print statement as:

```
PRINT A$(S,E)
```

If we use the first string variable, called "STRING 1 ", then S would have to be a 1 and E would have to be a 9, which translates into:

```
PRINT A$(1,9)
```

(Print all characters in the array between 1 and 9 inclusive.)

Now, suppose we keep track of the "index" numbers for the start and end positions of each of the strings. Then to print any one of them, say item X, if the start and end positions for each are stored in the S and E arrays, the PRINT statement will look like this:

```
PRINT A$( S(X) , E(X) )
```

The extra spaces are just there to make it easier for you to see what is enclosed in the parentheses. Now let's look at how to keep track of those numbers. Here is the revised program:

```
15 DIM S(100), E(100)
20 DIM A$(100), B$(20)
30 FOR N = 1 TO 100:A$(N) = " ":NEXT N
40 PRINT A$
45 A$ = "":REM NO SPACES BETWEEN THE QUOTES (LEN = 0)
50 FOR N = 1 TO 5
55 S(N) = LEN(A$) + 1
60 READ B$
62 A$(S(N)) = B$
65 E(N) = LEN(A$)
70 NEXT N
100 PRINT A$
110 FOR N = 1 TO 5
120 PRINT "A$( " ; S(N); " , " ; E(N); " ) = " ;
130 PRINT A$( S(N),E(N) )
140 NEXT N
200 DATA STRING 1 , STRING 2 , STRING 3 , 54
210 DATA FINAL STRING OF 5
```

After printing the entire string called A\$, this program prints out five lines which say, for example:

```
A$(1,9) = STRING 1
A$(10,18) = STRING 2
```

and so forth. This shows you that you can use single long strings as string arrays (groups of string items) just by keeping track of where, in the string, each one begins and ends. This not only works with DATA statements, you can also take string data from the user in the same way.

Notice that in line 120 the semicolon (;) is used several times. Remember, the semicolon tells ATARI BASIC that the printing cursor must remain exactly where it was when the preceding item has been printed. (Don't move it; don't go on to a new line.) You can put many things together on the same line this way if you wish. Line 120 demonstrates that even though the line is ended, the printing cursor still stays on the same line when line 130 is executed.

String arrays can also be used to help you save space, such as by somehow encoding what you want to say. A string array can hold a lot of different words for you, with another array pair holding the pointers to the words. Then, to print out a sentence, instead of having to store all of the words for all of the sentences you wish to use, in the correct sentence sequence, you might store the sentences as:

```
231, 42, 68, 2, 9, 18, 5
```

which might mean: (you are going)(down a)(long)(tunnel)(with) (three)(possible exits), where each of the word groups (not stored with parentheses, just there to show the possible groupings) might have been stored as shown, with the indexes into the word arrays selected by the numbers used.

In fact, the way that many of the early "adventure-type" games managed to store so many messages and instructions in so small a space was to store each message piece just once, then to use an index method to ask for each selected message piece to be printed out again. In other words, the message was listed as a series of numbers, representing a sequence of phrases.

This brings up another important point about DATA statements

. . . for ATARI BASIC, they can be just about anywhere in the program. They are actually nonexecutable statements, and are treated as REMarks in the program. ATARI BASIC will even permit a DATA statement or a REM statement to be used as the target for a GOTO statement, but please don't do that as it is very bad programming practice.

The other important point to be made about DATA statements is that ATARI BASIC treats all DATA statements as though they all occurred one right after the other, no matter how far separated in the program they might be, and no matter how the program may branch and jump and GOTO. For example, if there were eight data items in the program, in lines:

```
1 DATA 1,2,3
(MORE STATEMENTS)
500 DATA 4,5,6
(MORE STATEMENTS)
21000 DATA 7
(MORE STATEMENTS)
32500 DATA 8
```

Then ATARI BASIC would treat these statements no differently than if they had been entered in the first place as:

```
1 DATA 1,2,3,4,5,6,7,8
```

The reason this is mentioned is that some people, when writing programs, like to group the data near to where it is to be used, such as:

```
REM THIS IS GROUP ONE DATA
(Loop FOR READ GROUP ONE)
DATA
DATA
DATA
(OTHER STATEMENTS)
REM THIS IS GROUP TWO DATA
(Loop FOR READ GROUP TWO)
DATA
DATA
```

This structure is accepted by ATARI BASIC although it may not be accepted by an ATARI BASIC compiler. (Some BASIC compilers insist that all DATA statements be grouped together and placed as the last line numbers in the program.) It just provides you with different ways you can write your programs. (One way may be easier to understand, the other way may be required by the compiler if used. It is your choice how to organize your DATA statements.)

THE RESTORE STATEMENT

ATARI BASIC, as you ask it to perform READ statements, keeps a pointer that tells it which of the DATA items was last read. Then, for the following READ statement performed, it points to the next available DATA item. There may be cases where the DATA statement is not used just to initialize an array, but the data must be used in some repeated fashion anyhow. Instead of writing many multiples of the DATA statements, ATARI BASIC allows you to tell it to RESTORE the pointer to the first item again and reuse the data. A simple example of this is shown here:

```

10 DATA 6,5,4,3,2,1
20 FOR N = 1 TO 6
25 PRINT "THE FIRST ";N;" DATA ITEMS ARE:"
30 FOR M = 1 TO N
40 READ NUM
50 PRINT NUM;:IF M <> N THEN PRINT ",";
60 NEXT M
70 PRINT:PRINT:REM 2 BLANK LINES
80 RESTORE
90 NEXT N

```

RUN the program and see the effect that the RESTORE statement has. It reused the group of data items.

Now change the program by adding the following lines, just to see another couple of uses for this (you may think of many more):

```

22 SUM = 0:MULT = 1
45 SUM = SUM + NUM:MULT = MULT*NUM

```

```
65 PRINT:PRINT "THEIR SUM IS ";SUM
68 PRINT "ALL MULTIPLIED TOGETHER = ";MULT
```

RUN it again just to see how something extra has been added.

THE RND FUNCTION

In the programs we've done so far, we have always asked the user to make some kind of decision, which told the machine exactly what to do next. This may not always be the best way to go. Say, for example, you are designing a game of some kind. If the game always makes the same decision each time a person plays it, it may not be very interesting for long. Once you play it a number of times, you would know what it would do. If something is too easy to beat, it is not interesting any more. Therefore, you must throw in a few "twists" at times.

In the process of making decisions, or in pulling data out of different bags, so to speak, you can use the RND function provided by ATARI BASIC. This function provides a number between zero and one. The number is said to be *random*, which means unpredictable. If you base some of the program decisions on an unpredictable number, then the game you design could always be somewhat fresh, with many different possible decisions being made each time that section of the game is performed. Here is how to call the RND function:

```
A = RND(X)
```

where X can be any number or variable name (if it is not used by ATARI BASIC). But there must be no more than one variable name or number present in the parentheses. The result is assigned to be the value of variable A (or whatever name you use). If you execute this statement, or better still, try a one line program such as:

```
1 PRINT RND(X):END
```

then each time you specify RUN, a different random number, between zero and one, will be printed.

This is not very practical, initially. In most programs, you will

have a need for random numbers between certain other ranges. For example, you may want to print one of 10 possible messages. For this you will need a number between 1 and 10. And especially if there are *only* 10 messages, the numbers cannot be allowed to go outside of this range for indexing the messages (ATARI BASIC would give an out-of-range error). Therefore, the random number you get must be converted. Let's see how.

The general formula for getting a random integer between X and Y is to first take the range $Y - X + 1$, where Y is greater than X. For example, with the numbers 1 and 10, the range of possible values (if only integer values are to be used) is 10. (The actual values are 1,2,3,4,5,6,7,8,9, and 10.) Then take this number (10) and multiply it by the random number received, such as:

$$10 * \text{RND}(X)$$

This gives a number from 0 to 10 (10 times the original range of 0 to 1).

If the numbers are truly random, they will be somewhat evenly distributed. This means that about one out of each 10 will be in each one of the ranges (averaged over many numbers selected). This means that of 1000 numbers chosen, about 100 will be between 0 and 1, about 100 between 9 and 10, etc.

The numbers between 0 and 1 are not useful here, because they are outside the range of 1 to 10. Also, there are no numbers generated in the actual number 10 itself. Therefore, the result must be adjusted to use all of the numbers we want, by adding one to each number we get, then making it a whole number. This is the way it is done:

```

10 DIM N(10)
20 FOR M = 1 TO 10:N(M)=0:NEXT M
30 REM:SET COUNTERS TO ZERO
35 FOR M = 1 TO 1000
40 A = INT(RND(X)*10) + 1
50 N(A) = N(A) + 1
60 PRINT M, A
70 NEXT M
80 PRINT:PRINT "HERE ARE THE COUNTS OF NUMBERS"
90 PRINT "IN EACH RANGE OF NUMBERS"
```

```
100 PRINT  
110 FOR M = 1 TO 10:PRINT "COUNTER # ";M;" = ";N(M)  
120 NEXT M
```

If you RUN this program many times, the counters will differ each time. If you increase the outer loop value to 10000 instead of 1000, the counts may be somewhat closer in actual percentages, if the counter is truly random.

Now, how can a random number generator be used to direct the grabbing of data from different areas? Well, if you set up an array, the random number generator can be used to select the index into the array and pick a different number each time. If the array contains word pointers, such as the S and E arrays described in the preceding example, then selecting a random value for the pointer array will select a random word to be printed. You can use your own imagination for other applications.

REVIEW OF CHAPTER 5

1. The command used to reserve space either for a string or a set of numbers is the DIM command.
2. More than one item can be specified in the DIM command, as long as each is separated from the previous one by a comma.
3. Arrays are groups of numbers that are related to each other in some way. The array has a single name and many individual pieces, each of which can be accessed using an index number to tell which one of the pieces to use. The index number must be less than or equal to the maximum number reserved by the DIM statement for the array.
4. A FOR-NEXT statement is a very handy way of making some sequence of statements repeat. It also allows a STEP part of the sequence to tell the direction and the amount to step the control variable. The FOR-NEXT loops must be properly nested in order to function correctly.
5. Arrays may be used together to accomplish various tasks, as shown in the accountant's helper program.
6. The RND (random number generator) function can be used to generate different kinds of results each time a program is run.

CHAPTER

6

Menu Please

In this chapter, we are going to add some more ways of making a program presentable to the user. You already have a good amount of basic tools with which to construct a program. Now you will learn things to make it look better. The first tool you will be using is the Screen Editor. If you remember, you saw it introduced before in Chapter 1. There, though, you were only taught to use it to make up your programs. The Screen Editor can also be used from within your programs. The following section tells how.

ACCESSING THE SCREEN EDITOR FROM WITHIN A PROGRAM

Accessing the Screen Editor from within a program requires what is called an *escape sequence*. This means that the key labeled **ESC** (for escape) is part of the group of keystrokes that must be used to access the Screen Editor functions. Any Screen Editor command normally performed by pressing the **CTRL** key along with some other key can also be done using an escape sequence from within an ATARI BASIC program.

When the escape sequence is used, the program as LISTED

on the screen will not actually show the escape character. But it will show the character in a command sequence that will actually perform the Screen Editor function you want to do. These each relate to the character that is printed on the top of the key, which is how you normally find it in the first place when you use the Screen Editor. So this makes it easier.

Clearing the Screen

If you recall from Chapter 1, clearing the screen is done by pushing the **CTRL** key down, then touching the **CLEAR** button. (This one has no character printed on it, but it does have the word CLEAR.)

Clearing the screen from within a program requires putting the required escape sequence into a string, such as:

```
10 PRINT "␣"
```

where the character sequence that is actually inserted in the string is **ESC** then **CTRL CLEAR**. From this point onward in this chapter, the notation **CTRL** (something) means that the **CTRL** key is held down, and the other key is pressed.

The character that goes into the string is a kind of "kinked-up-left" arrow. This is a graphics symbol that means "home" the cursor, or clear the screen and move the cursor to the uppermost left-hand corner of the screen.

The reason an escape sequence is used is to allow you to LIST your program to examine the things you are telling it to do. If the "real" clear-screen character was to be "printed," it would make it impossible to LIST that part of your program. (The screen would clear each time, so the escape sequence is used.)

Moving the Cursor One Position Down

This function, if you remember, was called from the direct command mode by using the **CTRL** key along with the down-arrow key. To put this function into your program, put the following escape sequence into a string in a PRINT statement:

```
ESC CTRL (down-arrow key)
```

ATARI BASIC prints a down-arrow graphics character in your string and executes this command (in graphics mode 0) when the string is printed. (The graphics commands are introduced in a later chapter, "Getting Colorful.")

Moving the Cursor One Position Up

This function was called from the direct command mode by using the **CTRL** key along with the up-arrow key. To put this function into your program, put the following escape sequence into a string in a PRINT statement:

ESC CTRL (up-arrow key)

ATARI BASIC prints an up-arrow graphics character in your string and executes this command (in graphics mode 0) when the string is printed.

Moving the Cursor One Position Left

This function was called from the direct command mode by using the **CTRL** key along with the left-arrow key. To put this function into your program, put the following escape sequence into a string in a PRINT statement:

ESC CTRL (left-arrow key)

ATARI BASIC prints a left-arrow graphics character in your string and executes this command (in graphics mode 0) when the string is printed.

Moving the Cursor One Position Right

This function was called from the direct command mode by using the **CTRL** key along with the right-arrow key. To put this function into your program, put the following escape sequence into a string in a PRINT statement:

ESC CTRL (right-arrow key)

ATARI BASIC prints a right-arrow graphics character in your string and executes this command (in graphics mode 0) when the string is printed.

Notice that the last four functions are called *cursor moves*. This means that as they move around on the screen, they don't do anything to whatever characters may already be on the screen. They only move the cursor to a new position, potentially useful for input or for output at that new position.

Moving the Cursor Back One Position, Erasing Character

Normally, when you use the **DELETE/BACK S** key in direct mode, it is like the backspace key on a typewriter. You want to go back and type over a mistake of some kind, replacing what you did before with something new. You can use the same function in your program by putting the following escape sequence into a string in a PRINT statement:

ESC DELETE/BACK S

When ATARI BASIC sees this sequence, it will put into the string a graphics character that looks like a left-facing triangle pointer instead of the left-arrow, which normally represents a cursor move only. This is done to allow you to recognize what will happen.

What might this be used for? Well, maybe you have formed a nice menu on the screen (menu being a selection of programs to do), or maybe you are simply accepting a line of input from a user. Now, perhaps this person enters a bad input and your program knows it cannot use it. How do you make the screen pretty again? One way you could correct the error is to completely clear the screen, tell the user he or she made a mistake, then completely rewrite the screen again. This is not necessarily the best way. Let's explore some others.

First, let's look at using the **DELETE/BACK S** escape sequence just shown. We'll do this with a short sample program that will clear the screen, then print a sample line and demonstrate the function.

```
5 PRINT "◄":REM CLEAR SCREEN (ESC, CTRL-CLEAR)
10 DIM A$(100)
20 A$="THIS STRING IS GETTING SHORTER AND SHORTER AND
SHORTER AND SHORTER"
30 PRINT A$;
```

```

40 FOR N = 1 TO 66
50 PRINT " ◀ "; :REM CHAR FROM ESC THEN DEL/BACK S
60 FOR X = 1 TO 500:NEXT X
70 NEXT N

```

When you enter the program, don't forget the semicolons, otherwise the operation of the program won't make any sense. The semicolons keep the cursor on the same line so the delete/backspace function you put into the PRINT statement in line 50 can do its job properly. (By the way, line 60 is a FOR-NEXT statement with nothing to do inside of the loop. A do-nothing loop is often used as a time delay, and that is what it is doing here so you can see what is happening.) Now RUN the program. What happens is just what the string says!

How would this be used in a program on user data input? You could ask the user for a number or a word of some kind as the reply to a question. Then, instead of asking your program to accept a number directly, for example, you could get "smart" and accept the entire input as a string variable. This means that in one part of the program you might have the statement:

```
DIM REPLY$(40)
```

and in another part of the program, where the input is to be taken, the statement:

```
INPUT REPLY$
```

Then you could use your string-splitting instructions such as:

```
NUM$ = REPLY$(X,Y)
```

which would take the piece between and including positions X and Y in the string, and make them part of a separate string, etc. Then, if you are expecting a number, you could use the VAL function, and so forth. But, we are getting ahead of ourselves. Let's do an input example later, and put in all of the error trapping and so on when we do that. But for now, how do we make the display pretty again, once we find that the input is actually bad?

Before we go on, though, let's discuss one more point. That is, be sure you remember the way the Screen Editor treats "logical

lines" of data. As you ran the preceding sample program, did you notice that after the program did a delete/backspace on the second line, it continued at the last character of the previous line? This is a logical line that you printed. It was printed without any end-of-line characters in it because of the semicolon (;) at the end of each line. If you wish, you can print very long logical lines. The Screen Editor will handle all of them in the same way as just happened (just like one long continuous line).

But, you must also be aware that the Screen Editor does have a limit on the length of a line it thinks is part of a logical line, even though you may print a longer one. The limit is 120 characters, and this is reduced by as many characters as the indent from the left-hand side of the screen (put there for many TVs because of overscan, if you remember). Typically, then, the maximum length of physical/logical string that can be handled by the Screen Editor is 38×3 or 114 characters.

The limit on the number of characters was mentioned because we are going to look at more than one way of making the screen pretty again. First, let's look at some kind of a data entry form. Then let's try some of the cursor move activities on it. The following program will clear the screen, then it will print a couple of questions on the screen. When you have a menu on the screen, it is sometimes more effective in communicating with the user than if you just print your questions one at a time. Here it is:

```

10 DIM A$(100),B$(100)
20 PRINT "␣":REM ESC THEN CTRL-CLEAR
30 FOR N = 1 TO 3:PRINT " ↓ ";NEXT N
35 REM LINE 30 WAS AN ESC THEN CTRL-DOWN ARROW
38 REM GO DOWN 4 SPACES
40 PRINT "NAME: "
50 PRINT:PRINT:PRINT:PRINT
60 PRINT "AGE: "
70 FOR N = 1 TO 6:PRINT " ↑ →";NEXT N
72 REM THIS IS THE CURSOR UP AND CURSOR RIGHT
75 INPUT A$
80 FOR N = 1 TO 4:PRINT " ↓ →";NEXT N
90 INPUT B$

```


RUN this program and see what the menu presentation looks like. Try a couple of data entries. The example was kept as simple as possible so that we can work with it. Now enter the direct command:

```
PRINT A$
```

and see what you get. It printed out exactly what you entered. In this case, the Screen Editor began the line immediately at the current cursor position, and accepted everything you typed thereafter.

With this kind of menu, what happens if you type a name reply that is two or three lines long? Try it! When you type the reply long enough to go more than one line, did you see the AGE: entry move down one line to make room on the screen for the entry? And, when you hit **RETURN** to accept the entry, did you notice that the cursor moved down to exactly the right position on the screen as it normally did? This happened because of the Screen Editor again. The Screen Editor automatically makes room for the coming line by pushing everything down one space, for a maximum of three lines or 114 characters total as discussed earlier. You see, the cursor motion controls you have put into this demonstration program are "relative-motion" controls. In other words, the cursor will move in some way relative to where it is now. Since everything on the screen moved down, including the position at which your carriage return happened, the cursor wound up at the right position for the second data entry.

ABSOLUTE CURSOR POSITIONING

In contrast to what you just used for the data entry, here is a modified version of the same program. You will notice that it is shorter, but performs the same function. The exception is in the cursor positioning method.

```
10 DIM A$(100),B$(100)
20 PRINT "▲":REM ESC THEN CTRL-CLEAR
30 POSITION 2,4
40 PRINT "NAME: "
50 POSITION 2,8
```

```
60 PRINT "AGE: "  
70 POSITION 7,4  
75 INPUT A$  
80 POSITION 7,8  
90 INPUT B$
```

Notice that if you RUN this version, it does not respond correctly to the case where you enter more than one line of "name" data. This is because absolute cursor positioning is being used. The program does not "realize" that the data has been moved on the screen, and will put the cursor in the wrong place to get the next data entry. First we'll describe absolute positioning, then later we'll look at the different ways you could compensate for this kind of problem.

The ATARI graphics mode 0 screen we have been using for all of our data entry and program development is composed of 24 lines of 40 character positions on each line. When the Screen Editor is being called directly to place the cursor and to accept characters, it pays attention to the left-hand margin setting and reduces the effective width of the screen automatically. However, ATARI BASIC does have the keyword POSITION available for placing the cursor on the screen anywhere you want. The POSITION keyword expects to have with it a pair of X and Y positions specified. These can be calculated values or fixed numbers.

The range of numbers for the X part of the POSITION statement goes from 0 (meaning the farthest left-hand edge of the screen and ignoring any left margin) to 39 (meaning the farthest right-hand edge of the screen). The range of numbers for the Y part of the POSITION statement goes from 0 (meaning the topmost line of the screen) to 23 (meaning the bottommost line of the screen). In a diagram, it would look like Fig. 6-1. The margin mentioned earlier starts at X position 2 on the screen and operates normally for Screen Editor data handling. However, when you use the POSITION statement, the left margin is overridden. You can use columns 0 and 1 also.

The reason these numbers are referred to as 0 to 39 and 0 to 23, instead of 1 to 40 and 1 to 24, is that the values in the X and

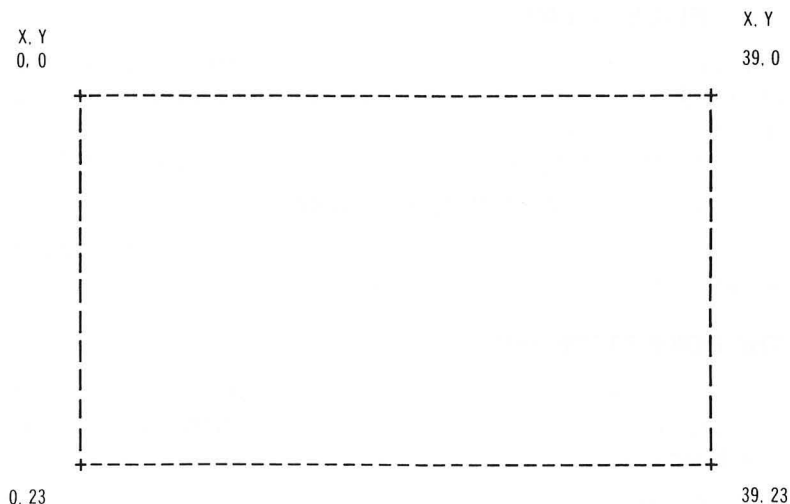


Fig. 6-1. The range of X and Y numbers for the POSITION statement.

Y coordinates can be used directly in an equation to locate screen data in the memory according to the formula:

$$\text{DATA_ON_SCREEN} = \text{SCREEN_DATA_START} + 40 \times Y + X$$

But that is a more advanced topic and will not be covered in this book.

Now you have three possible choices for data entry:

1. Print a line for each question, accept the answer there, or
2. Print a menu, then move around on it using relative cursor motions, or
3. Print a menu using fixed cursor positioning and move around it to the next data entry point using relative cursor motions.

So far, it looks as though items 2 and 3, if used together, might offer you some of the easiest menu handling. But let's look at some other possibilities first; maybe there are still some better ways to do things and keep "control" of what the user is going to see on the screen.

THE PEEK STATEMENT

It is time now to introduce the PEEK statement. This allows you to look directly at the contents of any memory location. The ones we will be primarily interested in are those that have something to do with controlling the system. The format of this statement is:

```
ANYNAME = PEEK(MEMORYLOCATION)
```

where the value given by the PEEK function to ANYNAME is anywhere from 0 to 255. An example is shown later.

THE POKE STATEMENT

If it is necessary that we not only look at something in the memory, but also to put something there for control purposes, the POKE statement is used. Its format is:

```
POKE MEMORYLOCATION,VALUE
```

where the VALUE, which is placed in a memory location, is anywhere from 0 to 255. Any value outside of this range will cause an error.

Let's say that you wanted to give somebody instructions which said:

```
PRESS OPTION TO CHANGE DIFFICULTY
PRESS SELECT TO CHANGE GAME
PRESS START TO BEGIN GAME
```

The following sequence of ATARI BASIC statements will read the option switches and will do the selected function on request. The example is here only to demonstrate the technique. This may give you some ideas for how it could be used in your programs:

```
10 DIFFICULTY = 1000:GAMESELECT = 2000:BEGIN = 4000
15 INFOX = 2:INFOY = 23:REM LAST LINE POSITION
20 REM DEFINES ACTUAL LINE NUMBERS FOR A VARIABLE
30 REM GOTO-TYPE STATEMENT AS SHOWN BELOW
35 REM LINE 15 DEFINES WHERE ERROR MSGS ARE PUT
300 PRINT "PRESS OPTION TO CHANGE DIFFICULTY"
310 PRINT "PRESS SELECT TO CHANGE GAME"
320 PRINT "PRESS START TO BEGIN GAME"
```

Now to see if any one of those keys is pressed, we have to write a loop that will stay in one place “forever” until the user obeys one of the three instructions we have provided. Here it is:

```
400 POKE 53279,8
```

readies the machine to read the console switches,

```
410 SW = PEEK(53279)
```

reads the current state of the switches,

```
420 IF SW = 7 THEN 400
```

If the value is 7, it means that one of the switches is pressed.

```
430 IF SW = 3 THEN GOTO DIFFICULTY
```

```
440 IF SW = 5 THEN GOTO GAMESEL
```

```
450 IF SW = 6 THEN GOTO BEGIN
```

```
460 POSITION INFOX,INFOY
```

```
470 PRINT "ONLY ONE SWITCH AT A TIME PLEASE"
```

```
480 GOTO 400
```

Line 430 defines the value you will find in location 53279 if only the **OPTION** switch is pressed. Line 440 defines the value if only the **SELECT** switch is pressed. Line 450 defines the value you will find if only the **START** switch is pressed. If the value 7 is found there, it means that none of the switches is pressed, so this program piece goes back to look again. This assumes that this is the *only* instruction group provided for the user on the screen.

If there is any value other than those tested, the program warns that only one switch at a time should be pressed. This is because each bit presents one “low bit” to the total to be shown here. The **START** switch creates a low on bit 1, making the total $7 - 1 = 6$ if only that switch is pressed. The **SELECT** switch creates a low on bit 2, making the total $7 - 2 = 5$ if only the **SELECT** switch is pressed. The **OPTION** switch creates a low on bit 4, making the total $7 - 4 = 3$ if only the **OPTION** switch is pressed. Therefore, any combination of these switches will make the total a different value. If you wanted to, you could use this information to report which combination was selected, and to make decisions based on that combination.

You will also notice that decision lines 430, 440, and 450 have used the names DIFFICULTY, GAMESEL, and BEGIN instead of the line numbers where the routines may be found. This is, as described in an earlier chapter, the variable-GOTO statement which ATARI BASIC allows. It is more descriptive and shows what is happening in the program. Using this kind of "internal documentation" is sometimes as good as putting in a lot of REMarks to tell what is going on. It lets you, and others, understand the operation of your program because you have related the program names and things it is doing to the actual BASIC instruction code that does the operation.

GAME CONTROLLERS (JOYSTICKS) FOR MENU SELECTION

Suppose, instead of or in addition to the keyboard, you wanted to use the joysticks to select what was to happen next? ATARI BASIC provides a set of keywords for reading these as well. We will relate them here to their possible use in menu selection, but you can use the values you see here to apply to games and other applications.

The ATARI® 400™ and 800™ Home Computers provide four possible positions into which joysticks can be connected. The ATARI® 600XL™, 800XL™, 1200XL™, 1400XL™, and 1450XLD™ Home Computers provide two positions. To be compatible with all units, we will primarily concentrate on the first two joysticks, accessed through the ATARI BASIC keyword STICK. STICK is actually a function, and needs a variable name to make it work. For example, STICK(0) and STICK(1) are the first and second joystick ports on the unit, respectively.

We will look at two different demonstration programs here. The first is used strictly to show you what the values of the different positions of the STICK can be. This is done with a simple graphics demonstration. The second demonstration program uses the joysticks to move a pointer from one position to another, and introduces another ATARI BASIC keyword, STRIG. First, let's clear the screen:

```
10 PRINT "⌘"
```

Now, for this program, we only want to move the cursor around a bit, and not leave anything permanent on the screen. So the program will be designed just to move the cursor around and leave it there until something different happens. Notice in the program that there is always a PRINT statement following each cursor move. This is because ATARI BASIC does not actually move the cursor to a new location determined by a POSITION statement until a PRINT statement has been issued. Here is another part of the program:

```

20 POSITION 2,3:REM LINE 3, SECOND COLUMN
30 PRINT,10,14,6
35 POSITION 2,8
40 PRINT,11,15,7
45 POSITION 2,13
50 PRINT,9,13,5
60 REM GIVES A DISPLAY
70 POSITION 2,20
80 PRINT "PLUG INTO FIRST PORT, MOVE STICK"
90 PRINT "CURSOR FOLLOWS STICK, VALUE = STICK(0)"
95 REM TELL USER WHAT TO DO

```

Now, we must provide a way for the computer to tell us it knows what value is being read from the joystick. In order to do this graphically, we must know where to put the cursor. In each case, we want the cursor to be near the number it is reading. Since that would take a lot of IF-THEN combinations, let's use an array of X and Y coordinates to show where the cursor should be for each reading of the STICK. First, reserve some space for the sets of X and Y values:

```
5 DIM X(16),Y(16)
```

Then, read in the values. Since they only range from 5 to 15, these are the only values we will have to read into the array:

```

120 FOR N = 5 TO 15:READ P:X(N)=P:NEXT N
130 FOR N = 5 TO 15:READ P:Y(N)=P:NEXT N

```

Lines 120 and 130 read the X values first, then the Y values.

Now we must provide the data for the READ statements to work

on. In this case, we will keep the DATA statements in the middle of the program rather than moving them to the end. As mentioned when the DATA statement was first introduced, it is treated just like a REM statement, and is basically ignored by anything but the READ statement.

Note that this also means you cannot combine anything else with the DATA statement and expect it to work. As a separate program sometime, try the following:

```
1 DATA 1,2,3:PRINT "I SAW THAT PART"
2 END
```

The DATA statement is treated the same as the REM statement; anything within the same line following the DATA statement is ignored.

Here is the data, corresponding to the X and Y cursor positions for where the numbers 5,6,7,(8),9,10,11,(12),13,14,15 are printed on the screen. Of course there is no 8 or 12, but the array just uses a place holder for them.

```
140 DATA 30,30,30,0,10,10,10,0,20,20,20
150 DATA 13, 3, 8,0,13, 3, 8,0,13, 3, 8
```

Now for the endless loop that will just take in the value of the STICK(0) function, and put the cursor in the right spot. Then it will loop back forever, doing it again.

```
200 M = STICK(0)
210 POSITION X(M),Y(M)
220 PRINT " ";:REM ONE BLANK SPACE
230 GOTO 200
```

Follow the instructions (if you have a joystick, that is) and see the cursor move to show the value that is being read. Make sure you hold the joystick pointing the right way, with the forward arrow pointing forward, and the cursor will move on the screen exactly the same way you are moving the stick.

In the second demonstration program, we will use this stick reading to move a cursor on the screen with a menu selection on it. Then, pushing the trigger button on the joystick will be the signal that the menu selection currently chosen should be exe-


cuted. That will introduce another ATARI BASIC keyword, STRIG. Let's start with a clear screen again:

```
10 GR.0
```

This is a slightly different version of clear screen. It says to go into graphics mode 0. In other words, reinvent the screen display. This takes a little longer than the PRINT statement because the clear-screen function just sends blanks to the data area of the screen, while the GR.0 (or, spelled out completely, GRAPHICS 0) command actually redefines all of the screen instructions as well as setting up the graphics 0 data area. You will see more on this subject in the chapter on "Getting Colorful."

Now, to set up a phony menu from which a program can be selected:

```
20 DIM A$(20),B$(20),C$(20),D$(20),E$(20),F$(20)
30 A$="PROGRAM ONE":B$="PROGRAM TWO"
40 C$="PROGRAM THREE"
```

For the following lines, just before you type the word PROGRAM, touch the ATASCII key (), then touch it again just before you touch the ending quote on each of the strings in lines 50 and 60. This will make the letters appear in reverse video, which is the way the program is to be done:

```
50 D$ = "PROGRAM ONE":E$ = "PROGRAM TWO"
60 F$ = "PROGRAM THREE"
```

Now, to print these items as a menu, let's use the same line positions as in the joystick example . . . lines 3, 8, and 13 from the top:

```
100 POSITION 2,3:PRINT A$
110 POSITION 2,8:PRINT B$
120 POSITION 2,13:PRINT C$
```

Now print the user instructions:

```
150 POSITION 2,18
155 PRINT "PLUG IN STICK 0, MOVE IT"
160 PRINT "TO SELECT MENU ITEMS,"
165 PRINT "PUSH TRIGGER TO EXECUTE PROGRAM"
```

We are going to do one more thing that is important to menu selection items: make the cursor invisible; it will just distract the user in this program.

180 POKE 752,1:REM MAKE CURSOR DISAPPEAR

The plan for the program is to have all three of the menu selection items initially in regular video. Then, depending on the position of the stick, the invisible cursor should be repositioned, and the menu selection at that cursor position should be rewritten in reverse video to highlight that this is the selection we want.

In the previous example, the cursor was moved to the position corresponding to the stick position. This would be undesirable in this case since we want to be able to let go of the joystick after making a menu pointer move, and not have to hold it there while we press the button to select that item. Therefore, we will be sampling the stick position and comparing the old position to a new position to see if the user has indicated some kind of move.

Also, we will be waiting a time between each look at the stick so it doesn't "circle" rapidly through the menu selections, making it impossible to stop at one correctly. (Sometimes BASIC is slow, and sometimes it can seem very fast.)

First, let's assume that the joystick is in the center position and nobody is moving it. At the start of the program, one of the program selections must be automatically selected. Let's make it the center one, just for an example:

300 POSITION 2,8:PRINT ES:Y = 8

which makes the center selection appear in reverse video, and:

210 OLDMOVE = 0

which says that this is the last known position of the joystick. (We haven't even looked at it yet, but this is program start time, and line 210 says that the input device is silent at the start.)

Now, for reading the joystick, we have nine different kinds of possible readings. The menu select should respond the same way whether the user pushes the stick forward, or forward and to the right, or forward and to the left, so we must distinguish between the various values and call all forward movements the

same, treat all center positions the same, and treat all rear movements the same. Let's see how:

```
350 T = STICK(0)
```

This saves us some typing during the test parts.

```
360 UP = - 5 : DOWN = 5
```

This tells us the direction of motion for the POSITION statement.

```
370 IF T - 2*(INT(T/2)) = 0 THEN MOVE = UP:GOTO 400
```

Notice that all of the up-movement joystick readings are even multiples of 2. This statement takes the whole number, divides it by 2, and then sees if there is any fractional part as a result. The numbers 6, 10, and 14 will have no fractional part after dividing by 2.

If it wasn't a move up, maybe it was a move down:

```
380 TA = INT(T/2)
```

```
385 IF TA - 2*INT(TA/2) = 0 THEN MOVE = DOWN:GOTO 400
```

This is how we can tell the movements apart.

The move-down values are 5, 9, and 13. The center values are 7, 11, and 15. When they are divided by 2 and the fractional part is discarded, the values become:

DOWN = either 2, 4, or 6.

CENTER = either 3, 5, or 7.

With these new values, we can say that if the result is odd, the stick is centered; if it is even, the stick is pulled down.

Since line 370 took care of the UP condition, and lines 380 and 385 took care of the DOWN condition, then the only condition left is the CENTER condition, which is:

```
390 MOVE = 0
```

where each of the values of MOVE will indicate how many positions on the screen we should move from the current one, based on the position of the stick. If you remember, at the beginning of the program the menu was printed at locations 2,3; 2,8; and 2,13 on the screen. So, if the current pointer says we are at location

2,8; then a value of +5 or -5 applied to the Y part of the POSITION statement will make us move to either the bottom or the top printed line.

Now, let's look at how to move just one position at a time on the screen:

```
400 IF MOVE = OLDMOVE THEN 340
```

This says that if the stick is held in any one position, don't do anything except go back to line 340. Here is line 340:

```
340 FOR M = 1 TO 50:NEXT M:OLDMOVE = MOVE
```

This is just a time delay before we fall into line 350, which looks at the stick again. What this says is that if the user is holding the stick in a forward or a reverse position, then make only one move per movement of the stick. In other words, center the stick first, then move it forward again to select an up motion of the pointer, or downward to select a down motion of the pointer.

The next line of the program forces it to do nothing if the stick is centered. The stick is automatically centered if nobody is touching it, so there should be no action on the screen at this time.

```
445 IF MOVE = 0 THEN 340
```

Now if it was not a no-move, it must be a move, but which way? For any kind of move, before we move, first we have to reprint the line in regular video instead of reverse video. Then we can go to the new line, print it in reverse video, and return to scan the stick again.

This next part of the program is not intended to be the most efficient in the way it handles the strings. It is just written in segments that try to be very understandable.

```
450 IF MOVE = 5 AND Y = 13 THEN 340
455 REM DON'T MOVE FARTHER DOWN THAN LAST LINE
460 IF MOVE = -5 AND Y = 3 THEN 340
465 REM DON'T MOVE FARTHER UP THAN FIRST LINE
470 IF MOVE = 5 AND Y = 8 THEN GOTO 600
480 IF MOVE = 5 AND Y = 3 THEN GOTO 700
490 IF MOVE = -5 AND Y = 13 THEN GOTO 800
```

```

500 REM THIS CASE IS MOVE = -5 AND Y = 8
510 POSITION 2,8:PRINT B$:REM RESTORE OLD
520 Y = 3:POSITION 2,Y:PRINT D$:GOTO 340
600 POSITION 2,8:PRINT B$:REM RESTORE OLD
610 Y = 13:POSITION 2,Y:PRINT F$:GOTO 340
700 POSITION 2,3:PRINT A$:REM RESTORE OLD
710 Y = 8:POSITION 2,Y:PRINT E$:GOTO 340
800 POSITION 2,13:PRINT C$:REM RESTORE OLD
810 Y = 8:POSITION 2,Y:PRINT E$:GOTO 340

```

Last, we need to add something that gets us into the program we want to run if the trigger button is pressed:

```
345 IF STRIG(0) = 0 THEN 1000
```

The ATARI BASIC STRIG function always returns a value of 1 if the trigger button is not pressed, and 0 if the button is pressed. The trigger on the second port can be accessed by STRIG(1). Likewise, the other two ports on the ATARI 400 and 800 Home Computers have trigger functions named STRIG(2) and STRIG(3).

Let's add something for the program to do, now that the selection part is finished:

```

1000 GR.O
1010 PRINT "NOW LOADING . . . ";
1020 IF Y = 3 THEN PRINT A$
1030 IF Y = 8 THEN PRINT B$
1040 IF Y = 13 THEN PRINT C$
1050 POSITION 2,20:END

```

For your program, you might want to make the subject of the THEN part of lines 1020, 1030, and 1040 something different, such as:

```
1020 IF Y = 3 THEN PRINT A$:LOAD "D:PROGRAM1.BAS"
```

or THEN GOTO 2000 (start of a program piece), or something else.

The important part of menu selection is to try to keep the user-entered errors to a minimum, and to tell the user what is happening along the way. It is very distressing if a program begins a

long set of calculations, or maybe a program load or search, and doesn't tell the user what is going on. After designing a program, you will know how long an operation should take and tell the user about it by adding another output line somewhere. If you don't, the user may think that either the program or the machine has gone bad.

HOW TO KEEP CONTROL OF THE MACHINE DURING USER INPUT

Earlier in this chapter, the ways of keeping the screen pretty were mentioned. We went from accepting direct strings of user input to using the function keys and the joysticks as input. These last two certainly help to keep control over the appearance of the screen. However, there are times when nothing less than an input string will really serve the purpose. Here, then, is the way to handle that:

Before now, whenever you were looking for a user input, you used the INPUT statement. That always meant that the user was in complete control of what went onto the screen, and that your program never got control back until the **RETURN** key was pressed. You have seen how this could mess up the screen display, especially if the user hit some cursor-move keys during data inputs. The better way to handle this is to have *you* control exactly what will be input, and exactly what will appear on the screen.

You can read the keyboard directly, using the following command:

```
KEY = PEEK(764)
```

If there is no key pressed, this memory location in your ATARI Home Computer will contain the number 255. If a key has been pressed, however, the computer will contain another number. That number is called the *internal key code*, and it relates to the position of the key on the keyboard, as well as whether either the **CTRL** or the **SHIFT** keys have been pressed.

Try the following program. It is a continuous loop that will read the keyboard location and tell you what key was found to be

pressed last (the internal key code only). Following this example is an explanation of how the key code relates to the normal ATAS-CII input that the INPUT statement normally receives.

```
10 KEY = PEEK(764)
20 PRINT "INTERNAL KEYCODE FOR THAT KEY IS: ";KEY
30 GOTO 10
```

When you RUN this, you will notice that it continues to display the number representing the last key you entered. This is because it operates as a "latch" and does not "clear" itself. If you want to read a different key each time, you must clear it yourself. The ATARI Operating System normally does this for you when you use the INPUT statement, but now you are in direct control.

Change the program to read the following, demonstrating the clearing of the keyboard location between reads:

```
10 KEY = PEEK(764)
15 IF KEY = 255 THEN 10:REM NO REPORT IF NO KEY
20 PRINT "INTERNAL KEYCODE FOR THAT KEY IS: ";KEY
25 FOR M = 1 TO 50:NEXT M
26 REM CALL THIS AUTOREPEAT DELAY
28 POKE 764,255:REM PUT 255 AT 764 TO CLEAR IT
30 GOTO 10
```

Now, when you press any key, it reports the internal key code assigned to it. Try any key with the **SHIFT** key held down. It is different than with the key alone. Try any key with the **CTRL** key held down. It, too, is different. This is how the ATARI Home Computer can tell the difference.

From an internal key code viewpoint, you will notice that a regular A = 63, a **SHIFT** + A = 63 + 64 = 127, and a **CTRL** + A = 63 + 128 = 191. And so it goes for all of the keys on the keyboard. A **SHIFT** + **CTRL** + key generally provides 192 + key code, but this group of combinations is not really supported by the ATARI Operating System, so it is best left alone.

For our purposes, we will be concerning ourselves primarily with the alphabetic and numeric keys, since these are the ones most likely to be used for user input. One thing to notice is that the combination **CTRL** + the "1" key does not respond. This is

a special operating system function that says "stop the listing to the screen" and will not report the key code as long as the ATARI Operating System is in control.

If your menu has on it a set of items that perhaps says:

PRESS LETTER TO SELECT PROGRAM

- A. CHECKBOOK
- B. ACCOUNTANT
- C. PAKMANIA

You would then have a way other than using the INPUT statement to get the user response. Such a menu program would look like this:

```

10 GR.0:REM CLEAR SCREEN
15 POKE 752,1:REM MAKE CURSOR VANISH
20 PRINT "PRESS LETTER TO SELECT PROGRAM"
30 POSITION 2,8
40 PRINT "A. CHECKBOOK":PRINT:PRINT
50 PRINT "B. ACCOUNTANT":PRINT:PRINT
60 PRINT "C. PAKMANIA"
80 POKE 764,255:REM CLEAR BEFORE READ
90 KEY = PEEK(764)
100 IF KEY = 63 THEN RUN "D1:CHEKBOOK.BAS"
110 IF KEY = 21 THEN RUN "D1:ACCOUNT.BAS"
120 IF KEY = 18 THEN RUN "D1:PAK.BAS"
130 GOTO 90:REM IF NO KEY, TRY AGAIN

```

Another thing you could offer, of course, is "HIT ANY OTHER KEY TO SEE MENU #2" and continue on with another display with other menu choices, perhaps even including RUN another menu program!

So this gives you direct control over single inputs, but it does require that you know in advance what the relationship of the key codes is to the characters you want to input. What we're saying here is that the key codes seem to have no direct relationship to the ATASCII character set (see the Appendix in your ATARI BASIC Reference Manual titled "ATASCII Character Set").

In particular, the ATASCII for an "A" is supposed to be decimal

65, B is 66, C is 67, D is 68, and so forth. When you give a command to PRINT CHR\$(65), you print the letter A, CHR\$(66) prints the letter B, and so on. But these key codes, if you were reading the keyboard directly to "user-input" a character string, don't give the same numbers. For example, A is 63, B is 21, C is 18, and so on. An interpreting program would become long if only key codes could be used this way. How do we make the conversion?

One way would be to provide a set of tables in each program, such as CODE(64) and CONVERTED(64), with DATA statements for each one to provide the correct translation. But that is the long way! The ATARI Operating System already provides a conversion table for you to use. It contains all of the conversion codes and may be referenced from BASIC. If you want to use this approach, the correct sequence would be:

```
300 POKE 764,255:REM CLEAR
310 KEY = PEEK(764):REM READ IT
320 IF KEY > 192 THEN 300:REM IGNORE CTRL + SHIFT
325 REM ALSO IGNORES VALUE 255 = NO KEY
330 ATASCII = PEEK(KEY + KEYTABLESTART)
```

Then use the ATASCII value in any way you would normally use a key entry. The value you get this way is an integer value, comparable to the value you get with the statement:

```
ATASCII = ASC(Z$)
```

where Z\$ is a one-character string. To make this value into a string character, you would have to:

```
STRINGPART$ = CHR$(ATASCII)
```

Or, to add it onto a string, once you were satisfied that it was one of the possibly correct values:

```
A$(LEN(A$) + 1) = CHR$(ATASCII)
```

which would tack it onto the back of an existing string.

But, to be able to use this approach, you would have to know where the internal ATASCII key code conversion table is located. Depending on the version of the machine you own, the table will

be located in different places. The technique of interpreting the values remains the same, but you must locate the table start in your own machine. You can do this with the following program:

```

10 X = 53245
20 FOR M = 1 TO 200
30 FOR N = 1 TO 196
40 X = X + 1
50 Y1 = PEEK(X):Y2 = PEEK(X + 1):Y3 = PEEK(X + 2)
60 IF Y1 <> 108 THEN 100
70 IF Y2 <> 106 THEN 100
80 IF Y3 <> 59 THEN 100
90 PRINT "TABLE STARTS AT: ";X:END
100 NEXT N
110 PRINT "SEARCHING FOR TABLE START AT: ";X
120 NEXT M

```

Once you have this number, called TABLESTART (of course, you can call it anything you like), you can use it to confirm that the values are as you would have expected. For example, you know that the key codes for A, B, and C are 63, 21, and 18. You can confirm that you can look them up in the key code table by doing the following experiment:

```

20 PRINT "PLEASE INPUT A KEYCODE"
30 INPUT KEYCODE:IF KEYCODE > 255 THEN 20
40 ATASCII = PEEK(TABLESTART + KEYCODE)
50 PRINT "THE ACTUAL ATASCII OF THAT KEYCODE IS: ";
60 PRINT ATASCII
70 PRINT: PRINT "AND THE CHARACTER IT REPRESENTS IS: ";
80 PRINT CHR$(ATASCII)
90 PRINT
100 GOTO 20

```

and try the following key codes:

63	21	18	(small a, b, and c)
127	85	82	(SHIFT + A, B, and C)
191	149	146	(CTRL + A, B, and C)

Notice the sequence of the ATASCII codes which come back out of the table. They are in direct numerical sequence. If you had tried the key codes for the numerals 0 through 9 (50, 31, 30, 26, etc.), you would also find that they are in numerical sequence. This was intentional when the character sequence was chosen. You will see how this fact can be used in the following example.

To show a practical use for what we just developed, let's take another look at the program piece that we did early in this chapter (asked for name and age). This time, since we are going to be controlling the input, there will be no problem with the absolute or relative cursor motion. We will print onto the screen something related to what the user enters, and only accept a keystroke if it is one of the ones we are expecting.

```

5 REM NEW VERSION OF NAME/AGE PROGRAM
10 DIM A$(30):REM ROOM FOR MAX NAME SIZE
20 GR.0:REM CLEAR SCREEN
30 POSITION 2,4
40 PRINT "NAME: "
50 POSITION 2,8
60 PRINT "AGE: "
70 POSITION 2,20:REM GIVE INSTRUCTIONS
80 PRINT "ENTER LAST NAME, COMMA, FIRST NAME"
90 PRINT "THEN PRESS RETURN"
100 POSITION 8,4:REM PUT CURSOR THERE FOR INPUT
101 PRINT " ":REM ONE BLANK SPACE

```

Now the input should be managed by the key code input method shown earlier in this chapter. For now, we will add in the key code routine here, then to control which part of the program is using the routine, we will use a "flag" so we know where to go back to when done. In a later chapter, you will see how to reuse routines without tricks as will be used here.

Here is the key code translator routine repeated from earlier. Remember that you must substitute your own value of KEYTABLESTART or this won't work right:

```

1300 KEY = PEEK(764)
1304 IF KEY>192 THEN 1300:REM IGNORE CTRL + KEY

```

```

1308 POKE 764,255:REM CLEAR FOR NEXT ONE
1312 ATASCII = PEEK(TABLESTART + KEY):REM USES
TABLESTART NO. FROM EARLIER PROGRAM
1320 IF KEY < 64 AND KEY <> 32 AND ATASCII > 64
THEN KEY = KEY + 64:REM MAKE ALL UPPER CASE
1330 ATASCII = PEEK(TABLESTART + KEY)
1340 IF FLAG = 1 THEN 120
1350 IF FLAG = 2 THEN 280
1360 PRINT "ERROR IN FLAG SELECT":END

```

In the following section, we will be trying to determine if the user has indeed done what we asked; that is, to enter a last name, a comma, and a first name.

One thing that was added to the key code part was to convert all characters received to upper case (all key codes except the comma have been converted to a code between 64 and 127). This means that the user need not hold down the **SHIFT** key, and so on. And even if he did, the letter group would still be accepted the same way—as all capitals.

So, now the key reading routine can be entered from the current section of the program:

```

105 CFLAG = 0
110 FLAG = 1:GOTO 1300:REM GET AN ATASCII VALUE
120 IF CFLAG = 1 AND ATASCII = 44 THEN 2000
125 REM NO MORE THAN ONE COMMA IN NAME LINE
130 IF ATASCII = 44 THEN CFLAG = 1:GOTO 160
135 IF ATASCII = 155 THEN 200:REM USER HIT RETURN
140 IF ATASCII >= 65 AND ATASCII <= 90 THEN 160
145 REM ENTER ATASCII CHARACTER BETWEEN A AND Z
150 GOTO 110:REM IF NOT, THEN TRY AGAIN
160 POKE 53279,7:REM MAKES CLICK TO TELL USER INPUT
ACCEPTED
170 PRINT CHR$(ATASCII);

```

Line 170 prints the character and keeps the cursor right where it is for the next acceptable character to come along.

Now, we have to make provisions for accepting the string for storing the name:

```
12 A$ = "":REM NOTHING BETWEEN QUOTES (LEN(A$)=0)
```

and then a way to collect the characters we are receiving, but not more than 29 total:

```
180 IF LEN(A$)>28 THEN 3000
```

where 3000 is the location which should report that too many characters were entered and maybe give the user a second try at it.

```
190 A$(LEN(A$) + 1) = CHR$(ATSCII):GOTO 110
```

Line 190 adds the character to the back of the string we are putting together, and goes back for more.

The next part should get an age from 0 to 99 (we are assuming a typical user, but let's go from 000 to 999 just in case). The inputs must be numbers, not letters, so after getting the ATASCII values, each must be in the range of 48 through 57. (See the ATARI BASIC Reference Manual, ATASCII Character Set Appendix, or simply PRINT ASC("0") then PRINT ASC("9") to confirm these numbers.)

First, we have to change the instructions to the user, then put the cursor in the right spot and ask for the input. Here is one of the places where we can use those "embedded control codes," Screen Editor commands to get rid of the previous instructions, and then to print our new instructions there.

```
200 POSITION 2,20:REM GO TO THE INSTRUCTION LINE
```

```
210 PRINT " ↑↑↑ ";
```

These string characters in line 210 are inverse video up-arrows. They were entered, after the first double quote, by the key sequence:

```
ESC SHIFT + DEL ESC SHIFT + DEL ESC SHIFT + DEL
```

This line contains the deferred Screen Editor command to delete the line on which the cursor is presently sitting. This is entered three times, to get rid of all three lines there. Each line lower moves up to fill the space. Therefore, since the cursor doesn't move in between, all three lines are deleted with a single command line.

Now that the old command lines have disappeared, we can print the new ones:

```
220 PRINT "AGE . . . NUMBERS ONLY . . . 000-999"
230 PRINT "THEN TOUCH RETURN"
```

and put the cursor at the right point to accept the numbers:

```
240 POSITION 8,8:PRINT " ";
```

Now to enter the key input routine, with a bit of setup first:

```
250 FLAG = 2:REM TELL WHERE CAME FROM
260 AGE = 0
270 GOTO 1300:REM GET INPUT
280 IF ATASCII = 155 THEN 400:REM IF RETURN HIT, PROGRAM
    ENDS
290 IF ATASCII >= 48 AND ATASCII <= 57 THEN 320
300 GOTO 270:REM OUT OF RANGE, NO KEY CLICK
320 POKE 53279,7:REM CLICK KEY = OK
330 NXTDIGIT = ATASCII - 48
340 AGE = AGE*10 + NXTDIGIT:REM CALCULATES AGE
345 IF AGE > 999 THEN 4000
```

Line 345 provides for an error report if the user enters a number too large with no **RETURN**.

```
350 PRINT CHR$(ATASCII);
```

Line 350 prints the character and leaves the cursor in position for the next character.

```
360 GOTO 270:REM GET NEXT CHARACTER
```

Then the last thing to do is to exit gracefully:

```
400 POSITION 2,20
410 PRINT " ■■■ ■ ":REM KILL MESSAGES
420 PRINT "THANK YOU":END
```

Lines 2000, 3000, and 4000 should also be added somewhere to tell the user that he (2000) can't have more than one comma in the line, or (3000) can't enter more than 29 characters in the name, or (4000) can't enter an age greater than 999. For each


```

260 AGE = 0
270 GOTO 1300:REM GET INPUT
280 IF ATASCII = 155 THEN 400
290 IF ATASCII >= 48 AND ATASCII <= 57 THEN 320
300 GOTO 270:REM OUT OF RANGE, NO KEY CLICK
320 POKE 53279,7:REM CLICK KEY = OK
330 NXTDIGIT = ATASCII - 48
340 AGE = AGE*10 + NXTDIGIT
345 IF AGE > 999 THEN 4000
350 PRINT CHR$(ATASCII);
360 GOTO 270:REM GET NEXT CHARACTER
400 POSITION 2,20
410 PRINT " █ █ █ ":REM KILL MESSAGES
420 PRINT "THANK YOU ":END
1300 KEY = PEEK(764)
1304 IF KEY > 192 THEN 1300:REM IGNORE CTRL + KEY
1308 POKE 764,255:REM CLEAR FOR NEXT ONE
1312 ATASCII = PEEK(TABLESTART + KEY):REM USES
TABLESTART NO. FROM EARLIER PROGRAM
1320 IF KEY < 64 AND KEY <> 32 AND ATASCII > 64
THEN KEY = KEY + 64:REM MAKE ALL UPPER CASE
1330 ATASCII = PEEK(TABLESTART + KEY)
1340 IF FLAG = 1 THEN 120
1350 IF FLAG = 2 THEN 280
1360 PRINT "ERROR IN FLAG SELECT":END
2000 POSITION 2,22:PRINT "*****";
2005 PRINT "ONLY ONE COMMA ALLOWED"
2010 POSITION 8,4
2020 FOR N = 1 TO LEN(A$) + 1
2030 PRINT " ";
2040 NEXT N:REM ERASE USER INPUT
2050 A$ = "":GOTO 100
3000 POSITION 2,22:PRINT "*****";
3005 PRINT "NAME FIELD ONLY 29 MAX"
3010 POSITION 8,4
3020 FOR N = 1 TO LEN(A$) + 2
3030 PRINT " ";

```



```

3040 NEXT N:REM ERASE USER INPUT
3050 A$="":GOTO 100
4000 POSITION 2,22:PRINT "****";
4005 PRINT "AGE MUST BE UNDER 999."
4010 POSITION 8,8
4020 FOR N=1 TO 4
4030 PRINT "█";:REM ESC CTRL + DEL
4035 REM LOOKS NEATER TO HAVE STRING PULLED BACK
4038 REM TO ORIGIN THAN BEING WIPED OUT BY CURSOR
4040 NEXT N:REM ERASE USER INPUT
4050 GOTO 240

```

REVIEW OF CHAPTER 6

1. The process of forming a menu can be aided by using the built-in functions of the ATARI Screen Editor.

2. The Screen Editor functions are put into character strings by the **ESC** key, followed immediately by the **SHIFT** + key or **CTRL** + key, which would normally perform a screen edit function in direct mode.

3. Absolute cursor positioning can be specified for placement of text on a graphics 0 background, and one can move around the screen if desired using relative cursor motion.

4. The PEEK statement is used to look directly at memory locations. The value found in any location will vary from 0 through 255. PEEK is a function, and to use it, you say X=PEEK(MEMORY), where the memory value is from 0 to 65535.

5. The POKE statement is the opposite of PEEK. It is used to put something into memory. To use it, you say simply POKE MEMORY,CONTENTS, where the memory address is as in item 4, and the contents is from 0 to 255.

6. The condition of the option switches can be determined by reading (doing a PEEK) from location 53279. By doing a POKE 53279,7 you can also make the keyboard on the ATARI 400/800 Home Computers make a clicking sound.

7. The joysticks can be read using the ATARI BASIC keyword STICK(N), where N=0,1,2, or 3 representing the four possible

plug-in positions. The triggers can be read using the keyword STRIG(N). These also are functions, so they need to be read by specifying ANYVARIABLE = STRIG(N) or ANYVARIABLE = STICK(N).

8. It is possible to maintain control over exactly what the user inputs to your program by reading the keyboard directly instead of using INPUT statements. This is done by reading location 764 and translating the key code found there into something you can use.

CHAPTER

7

Introduction to Subroutines

At the end of Chapter 6, there was an example of a keyboard reading routine which was entered from two different points in the program. In both cases, we wanted to read the keyboard without using the INPUT statement because it meant better control of the user input.

Because it was entered from two different points in the program, it was necessary to include a variable called FLAG, which had a different value for each entry point. Then, at exit, the value of FLAG was tested and the keyboard reader returned to the correct routine based on the value of FLAG.

There is another way this could have been done. In fact, the concept you are about to see is very widely used. This is the concept of *subroutines*. A subroutine is, as its name implies, a routine that "serves" another routine . . . a routine that is called upon by another one and performs a task. On completion of the subroutine's task, it RETURNS to the calling routine.

ATARI BASIC allows one subroutine to serve many master routines if desired. This is because ATARI BASIC always "saves the RETURN address" of the calling routine so that, at exit, it can RETURN to the correct place. In other words, when a subroutine is given control, ATARI BASIC saves the previous running status

of the machine (address of the calling routine). Then when the subroutine is finished, ATARI BASIC can restore the machine operation to the way it was before the subroutine was "called."

STRUCTURE OF A SUBROUTINE CALL

A diagram of a typical operation is shown in Fig. 7-1. This is an example of another menu program, using the keyboard reader built as a subroutine. Let's examine the typical structure of a subroutine.

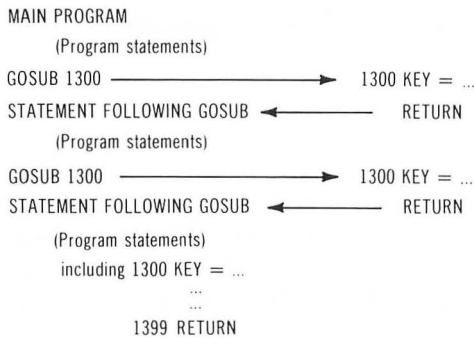


Fig. 7-1. Diagram of a typical subroutine structure.

Fig. 7-1 shows a number of arrows, coming from different parts of the program, going to what looks like a copy of statement 1300, then coming back. In fact, there is only one subroutine with that line number in the program. The diagram is only intended to show that the GOSUB statement can be used from anywhere in the program, and wherever it is used, ATARI BASIC will return control to the next statement in the sequence following the GOSUB itself.

As indicated in Fig. 7-1, a subroutine call consists of the ATARI BASIC keyword GOSUB, followed by the line number that is the beginning of the subroutine itself. When the subroutine is finished doing what it was designed to do, it uses the ATARI BASIC key-

word RETURN. Here is an example program you can try that will show you how subroutines are used:

```

10 PRINT "STARTED AT LINE 10"
20 PRINT "DID A GOSUB 6000 FROM 20":GOSUB 6000
30 PRINT "RETURNED TO NEXT LINE 30"
40 PRINT "DID ANOTHER FROM 40":GOSUB 6000
50 PRINT "AND GOT BACK TO NEXT LINE OK"
60 END:NEED THIS ELSE WILL FALL INTO 6000
70 REM WITH NOBODY TO RETURN TO.
6000 PRINT:PRINT "      GOT TO 6000 OK"
6010 PRINT
6020 RETURN:REM RETURN TO WHOEVER CALLED IT

```

That was a real do-nothing program, but it did illustrate that when the RETURN is performed, it returns to whichever place it came from, just as though it never went anywhere at all.

Because we were relating the subroutine calls to the keyboard routine before, let's look at how it would appear as a subroutine:

```

1300 KEY = PEEK(764)
1304 IF KEY > 192 THEN 1300
1308 POKE 764,255
1312 ATASCII = PEEK(TABLESTART + KEY)
1320 IF KEY < 64 AND KEY <> 32 AND ATASCII > 64 THEN
KEY = KEY + 64
1300 ATASCII = PEEK(TABLESTART + KEY)
1340 RETURN

```

Notice that the only difference between this and the original version is that line 1340 now says RETURN, instead of a flag test. This means that if you wanted to, you could use this subroutine for many lines on the menu. Then, for each time it was used, you wouldn't have to add a different flag test line or any flag setting lines into the routines that used KEY.

Usually, you would tend to use subroutines where there was a long group of program statements that you might want to do many times in the program, in many different places. In these cases, it is much easier to type GOSUB 10000 (or whatever num-

ber) than it is to type each of the groups of lines wherever they might be needed. But you don't have to limit the use of GOSUB to long sets of lines. You can use it wherever it is convenient.

Another thing that may make it more convenient for you is that ATARI BASIC allows you to use variable names in the GOSUB as well as the GOTO statement. This means that once you know at which line number you have the starting location of the subroutine, you can define a variable name to be equal to that line number. Let's see how this could be used.

Let's say you wrote a program that had a lot of cursor motion statements in it. The cursor control statements always need the PRINT statement, along with the quotes, the **ESC**, and then the **CTRL** + something. It would be tedious to have to write this sequence many times in a program. So this is a case where a one-line subroutine can help you write your programs. Here is an example:

```
13000 PRINT " ↓ ":RETURN:REM ESC CTRL + UP
14000 PRINT " ↓ ":RETURN:REM ESC CTRL + DN
15000 PRINT " → ":RETURN:REM ESC CTRL + RT ARROW
16000 PRINT " ← ":RETURN:REM ESC CTRL + LT ARROW
17000 PRINT " ✖ ":RETURN:REM ESC CTRL + CLEAR
```

These are now the subroutines you can call to move the cursor on the screen. To use them effectively, and to tell the user more about what is happening in your program, you could now give names to the subroutines, such as:

```
10 UP = 13000:DOWN = 14000:RIGHT = 15000
20 LEFT = 16000:BLANK = 17000
```

Now, in the body of the program you can use:

```
500 GOSUB BLANK:REM CLEAR THE SCREEN
```

or

```
670 FOR N = 1 TO 5:GOSUB DOWN:NEXT N
```

to move the cursor down five spaces, and so on.

Whenever possible, it is usually best to put things into the programs that will explain exactly what is happening, especially

for a beginning programmer. This is called *internally documenting* the program. If a program is properly documented internally, you will be able to find out what it was supposed to do. If you come back to the program months or even days later, it will be easier for you to follow the train of thought if the variables and the sequences mean something in relation to what the program is trying to accomplish.

There is another benefit you can get from using subroutines; that is, you can develop a program in pieces. A typical structure might be:

```
DO JOB NUMBER 1
DO JOB NUMBER 2
SET UP LOOP FOR-NEXT STEP
DO JOB NUMBER 3
LOOP BACK IF NOT DONE
DO JOB NUMBER 4
```

which could translate to:

```
10 GOSUB 400
20 GOSUB 750
30 FOR N = 1 TO LOOPLIMIT
40 GOSUB 18720
50 NEXT N
60 GOSUB 23000
(rest of program, including subroutines)
32000 END
```

What this kind of structure means is that some of these subroutines may be developed separately, and stored separately on your disk.

While you are trying to get each subroutine ready to function, you might have a main program consisting of one or two lines which set up the initial conditions needed by the subroutine, if any, then a single line consisting of a GOSUB to that subroutine, followed by an END. Such a subroutine debugging program might look like this:

```
10 DIM A$(5),V(5)
20 DATA A,B,C,D,E
```

```

30 FOR N = 1 TO 5:HEAD A$(N,N):NEXT N
40 FOR N = 1 TO 5
50 PRINT "PLEASE INPUT VARIABLE ";A$(N,N)
60 INPUT V(N)
70 NEXT N

```

Then, suppose the subroutine you are trying to test has variable names X, Y, Z, T, and H. The next few lines of the test program would then be:

```
80 X = V(1):Y = V(2):Z = V(3):T = V(4):H = V(5)
```

and

```
90 GOSUB 21000
```

assuming that subroutine 21000 was the number of the one being tested. Then, assuming that you had to give the subroutine several sets of numbers in order to see if it was working correctly, the next line would be:

```
100 GOTO 40
```

An alternative to line 80 for naming the variables for test would be to eliminate it entirely and change the DATA statement to read:

```
20 DATA X,Y,Z,T,H
```

which, of course, is simpler.

Just to make sure you are not confused about subroutines and how to construct them, here is another definition for a subroutine: *A subroutine is a statement, or a group of statements, which terminates with a RETURN keyword.* This means that even the following can be called a subroutine:

```
5000 RETURN
```

The other thing you must remember about subroutines is that it is not possible to perform the RETURN statement unless some other part of the program has gone there using the GOSUB statement. The mistake that beginning programmers often make is to put the subroutines in the program somewhere, use them, then

have the program, during its normal sequencing, “bumble into” them. The following example shows what you should watch out for:

```

300 GOSUB 2000
    (more statements)
640 GOSUB 2000
    (more statements)
1900 REM END OF PROGRAM NORMAL OPERATIONS
1950 REM BUT NO BASIC "END" STATEMENT USED
1980 REM AND NO GOTO ANYWHERE . . . NEXT ACTION
1990 REM IS TO FALL DOWN DIRECTLY INTO 2000
2000 REM SUBROUTINE START
2200 RETURN

```

What will happen here is that the program, although seemingly finished when it gets to line 1990, will continue to execute at the next line, which is part of the subroutine! You may never have told ATARI BASIC that the subroutine was to be performed, but then without the END statement before the subroutine, it doesn't know any better. BASIC always executes the next sequential statement if it is not told to do otherwise.

This points out an important point about the use of the END statement. END does not have to be reserved for the very last statement (highest numbered statement) in your program. Instead, the word END tells ATARI BASIC that the processing job is ENDED and it may return control to the direct command mode again.

What kind of process is good for use as a subroutine? The answer to this question is *anything* that makes the job easier. The rest of this chapter will be dedicated to providing some examples of subroutine construction and use. You will probably think of many more.

SCREEN DECORATION SUBROUTINE

In Chapter 6, “Menu Please,” we concentrated on how to accept data gracefully from the user. This routine will aid in that

presentation by providing a way to decorate the menu. Assuming that you have already placed the various menu choices on the screen, it might make the screen more presentable if there was a border of all asterisks(*) surrounding the data entry area. The following routine will provide the capability to draw rectangular enclosures at any point on the screen, in any character, including graphics symbols. It does not include all of the error checking you might feel is necessary, but then *you* are the user. You can feed the correct numbers through the program you design to use the subroutine. The main program provided with the subroutine here only shows you one of the ways it can be tested. First, the concept itself . . . a flow narrative if you prefer:

1. The screen in graphics 0 is 40 characters wide by 24 characters high. When addressing this screen with the POSITION statement, these limits must be specified as 0 to 39 horizontal, and 0 to 23 vertical to avoid generating an error.

2. For the sake of the test program, reserve the bottom four lines of the display for the instructions. This means limit the second part of the POSITION statement to the range 0 to 19.

3. Because of the way the ATARI Screen Editor prints characters to the screen, limit the first part of the POSITION statement to the range 0 to 38. (If you print something in column 39, the rest of the lines on the screen will be moved down to make room for a new line that the Screen Editor thinks is coming next.)

4. Use loops to print an upper line, a lower line, and two enclosing lines, making sure that none of the areas printed fall outside of the specified margins. If any do fall outside, then only print the part that "should be visible."

Following the subroutine is a short test program constructed of just a few lines, as suggested earlier in this chapter.

First, assume that the user-defined limits for a box to be drawn on the screen are X1 and X2 for the X direction, and Y1 and Y2 for the Y direction. Then, to draw a line across the screen, keep the Y part of the POSITION statement constant, and vary the X part from X1 to X2:

```
20000 FOR X = X1 TO X2
```

Now make sure that the value of X is in bounds:

```
20010 IF X < 0 OR X > 38 THEN 20090
```

Make sure that Y is in bounds, too:

```
20015 Y = Y1
20017 IF Y < 0 OR Y > 19 THEN 20040
```

Now position, then print the upper line if it is in bounds:

```
20020 GOSUB 23000 : REM A GOSUB INSIDE ANOTHER
```

where subroutine 23000 is composed of the following statements:

```
23000 POSITION X,Y
23010 PRINT Z$(1,1);
23020 RETURN
```

Now that we've defined that this subroutine will print a character, it is time to define the space for the character in the memory:

```
10 DIM Z$(1)
```

Now continue with the rest of the routine.

The next part of the program must print the lower part of the box, meaning the lowest line across the box bottom if it is in bounds:

```
20040 Y = Y2
20045 IF Y > 19 OR Y < 0 THEN 20090
20050 GOSUB 23000
```

Finally, select the next position. (This will draw one character at a time of the upper and bottom parts of the box, then go on to the next character.)

```
20090 NEXT X
```

Now, the next part of the subroutine should draw the sides of the box if they are in bounds:

```
21000 FOR Y = Y1 TO Y2
21010 IF Y < 0 OR Y > 19 THEN 21040
21015 X = X1
```

And make sure X is in bounds:

```
21020 IF X < 0 OR X > 38 THEN 21040
21030 GOSUB 23000
```

Last, change X to X2 to draw the rightmost edge of the box:

```
21040 X = X2
```

And make sure that all of the X parts are in bounds:

```
21050 IF X < 0 OR X > 38 THEN 21090
21060 GOSUB 23000
```

Now do the next value:

```
21090 NEXT Y
```

Finally, RETURN from this subroutine to the calling routine:

```
22000 RETURN
```

Here is a small test program that can be used to demonstrate this subroutine

```
1 PRINT "▲": REM CLEAR THE SCREEN (ESC, THEN CTRL +
CLEAR)
20 POSITION 2,20:PRINT "CHARACTER = ";:INPUT Z$
30 POSITION 2,21:PRINT:PRINT:REM CLEAR LINES
40 POSITION 2,21:PRINT "SPECIFY X1,X2":INPUT X1,X2
50 POSITION 2,22:PRINT "SPECIFY Y1,Y2":INPUT Y1,Y2
60 GOSUB 20000
70 GOTO 20
```

If you RUN this program, you can enter single characters, then an X1 comma X2 in the range of 0 to 38, and a Y1 comma Y2 in the range of 0 to 19, and this subroutine will draw a box on the screen in that character. If you specify X1,X2,Y1, or Y2 outside of their ranges, the box will have only the limits that can actually fit on the screen.

MORE USES FOR SUBROUTINES

There are, of course, many other applications for subroutines, but we will cover just one “officially” before proceeding to the next chapter, since that one covers an introduction to graphics and sound. There are some programs there that make use of subroutines as well.

The following will introduce a handy subroutine that converts from memory value to binary. Binary numbers are what the computer normally uses to represent everything it is doing. This subroutine will allow you to split any number into its binary parts. Then another test program will be shown which demonstrates one of the uses of this binary subroutine.

First, though, you will have to know just what a binary number is. In the world of the computer, all it really knows are two things, on and off, represented by the numbers 1 for on and 0 for off. It is by a combination of these numbers, 1 and 0, that the computer makes up larger numbers. Some of these larger numbers are used to make up control instructions for the machine, and some others are just considered as data. At this time, we will only be concentrating on the numbers used as data.

When the computer wants to represent a number, it must combine a batch of 1s and 0s in some way to accurately tell what the number might be. It does this using the “powers-of-2” rule, as follows:

Each bit position in a multibit word represents a specific power of 2; that is, 2 to the zero power, which is 1; 2 to the first power, which is 2; 2 to the second power (means 2×2), which is 4; 2 to the third power (means $2 \times 2 \times 2$), which is 8; then any whole number can be represented by an addition of some of the powers of 2.

Just to prove this, take a random whole number from 0 to 1000. Following is a list of the various powers of 2. Pick the numbers out of the list which add up to your selected number. You will soon see that there is no whole number less than or equal to 65535 that you cannot represent by a combination of powers of 2. Notice that you need to take only *one* occurrence maximum

of any single power of 2. This is where the binary number system comes into play. Here is the table:

Power of 2	Value (decimal)
15	32768
14	16384
13	8192
12	4096
11	2048
10	1024
9	512
8	256
7	128
6	64
5	32
4	16
3	8
2	4
1	2
0	1

Once a binary number has been translated from the decimal, it is represented by a string of binary digits whose positions correspond to the power of 2 which each represents, as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	(powers of 2)
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	(binary number)

This binary number is the largest one that can be represented with 16 binary digits, because they are all 1s. This number represents decimal 65535, because it is the sum of all of the numbers in the preceding table. When you make up a binary number, and you want to translate it to decimal, you start out with a total of zero. Then for each 1 in the binary number, you add the value of its position in the number (from the table) to the sum until all binary digits have been accounted for.

To convert from decimal to binary, on the other hand, is a slightly easier task, since there is no table needed. The resulting numbers will either be 0s or 1s. Converting from decimal to binary consists of taking a number and dividing it by 2. If there is a fraction left over, the fraction will be a 1. If the number divides

evenly, the remainder of the division will be zero. The remainder of the division is, in each case, part of the binary number you are trying to convert. Then the remainder is discarded (actually made part of the binary number) and the number that remains (half of the previous value less any fractional part) is divided by 2 again until the result is zero. Then what is left is the binary value. Let's look at a typical sample number, say 200:

$$\begin{aligned}
 200/2 &= 100 \text{ remainder} = 0 \\
 100/2 &= 50 \text{ remainder} = 0 \\
 50/2 &= 25 \text{ remainder} = 0 \\
 25/2 &= 12 \text{ remainder} = 1 \\
 12/2 &= 6 \text{ remainder} = 0 \\
 6/2 &= 3 \text{ remainder} = 0 \\
 3/2 &= 1 \text{ remainder} = 1 \\
 1/2 &= 0 \text{ remainder} = 1
 \end{aligned}$$

Note that the division continued until the result of the division by 2 was zero. Copying down the binary number in the reverse order of its remainders, the result is:

1 1 0 0 1 0 0 0

Referring to the table again, you can see that this is the value of $128 + 64 + 8 = 200$, which is correct! This technique works for any whole number.

This technique is used in the program that follows. There is a reason for the splitting of a number into its binary parts. A sample program demonstrating this will follow immediately after the subroutine description.

The memory system of your ATARI Home Computer is eight bits wide. This means that the contents of any of the different memory locations in this machine can be represented by a series of eight binary digits. This also means that the maximum decimal value that can be found in any memory location is 255. If you recall when PEEK and POKE were introduced, the maximum value specified was 255. Now you know how this number came about.

If we are going to split a number from one value into eight values, the splitting technique might be similar for all of the eight

different pieces. So it would be most efficient to use an array to hold the eight pieces. Let's define an array called BIN:

```
5 DIM BIN(8)
```

Now we are going to define two subroutines, one that uses another one.

The first subroutine is one that does the dividing by 2 and then defines that the number is now half of what it was to start with (see the explanation of binary splitting in the preceding section). This subroutine is as follows:

```
2000 Y1 = INT (Y/2)
2010 BIT = Y - Y1*2
2020 Y = Y1
2030 RETURN
```

Line 2000 takes the value of Y, divides it by 2, and deliberately throws away the remainder; line 2010 says that the remainder bit equals the original value minus the value divided by 2 multiplied by 2 after the remainder was discarded; finally, line 2020 says the new value of Y is half what it started with. Then a RETURN is at line 2030.

This subroutine does everything shown in the preceding example except for providing a place for putting the values together. That is done by the routine that uses this subroutine (assuming, for now, that all numbers to be converted are 255 or less):

```
1000 FOR N = 1 TO 8
1010 GOSUB 2000
1020 BIN(N) = BIT:REM SAVE THE BIT CALCULATED
1030 NEXT N
```

Now for a short routine which can test this subroutine group:

```
10 PRINT "ENTER INTEGER BETWEEN 0 AND 255"
20 INPUT D
22 IF D = 0 THEN 26
24 IF D < 1 OR D > 255 THEN 10
26 IF Y - 2 * INT (Y/2) <> 0 THEN 10
```



```
30 Y = D
40 GOSUB 1000
```

This does the conversion, now for a presentation routine:

```
50 SUM = 0
60 FOR N = 8 TO 1 STEP - 1
```

Here is the first time we're demonstrating that the steps can be minus as well as plus.

```
70 SUM = SUM*10 + BIN(N)
80 NEXT N
```

This makes a binary number into a decimal number consisting of a string of binary digits.

```
90 PRINT "BINARY EQUAL IS: ";SUM
100 GOTO 10
```

The reason we chose to reconvert the real binary number into a decimal "equal" was because it is one of the easiest ways to "suppress leading zeros." This means that if there are any zeros in front of the value, they don't get printed by ATARI BASIC. This prevents the number 1 from being printed as 00000001.

However, there are times when you *want* to see all of the zeros, both leading and included zeros. Then, instead of the preceding lines 50 through 100, you would use something like the following:

```
50 V = ASC("0")
55 PRINT "THE FULL BINARY VALUE IS: ";
60 FOR N = 8 TO 1 STEP - 1
70 PRINT CHR$( V + BIN(N));
80 NEXT N
90 PRINT
100 GOTO 10
```

This prints all of the zeros, both within the value as well as the leading zeros. You may find statements 50 and 70 interesting. We have converted from a character value to an ordinary number. Then, we used the ordinary number to calculate a character value to print.

Let's change the program one more time, then RUN it again. This time the PRINT statement will print either a blank space or an asterisk. It is this latest version that will be used in the demonstration program that follows. Again, lines 50 through 100 are what we change, as follows:

```

55 PRINT "THE BINARY VALUE APPEARS AS: ";
60 FOR N = 8 TO 1 STEP - 1
70 IF BIN(N)=0 THEN PRINT " ";
75 IF BIN(N)=1 THEN PRINT "*";
80 NEXT N
100 GOTO 10

```

Now, we will use the last part of the demonstration program for another purpose, as a subroutine to something else. Just for the sake of completeness, the entire final version of the program is contained here.

The purpose of this program is to show you how the characters are formed on the screen. You will see that the ATARI Home Computer stores the character set as a series of eight memory locations in the Operating System area of the memory. When it wants to produce a character on the screen, part of the ATARI Home Computer called ANTIC figures out which character it is, then goes into the memory to find the bits of which the character is made. This program does the same thing.

The character set for the computer starts at location 57344, and contains 1024 total locations.

```


1 GR.0 : REM CLEAR THE SCREEN
3 DIM BIN(8):REM SAVE SPACE FOR BINARY RESULT
15 X = 57344:REM START OF CHAR. SET
20 FOR CH = 0 TO 127:REM 128 CHARACTERS
22 PRINT "THE CHARACTER BELOW BEGINS AT ";X
25 FOR LN = 0 TO 7:REM 8 LINES EACH CHARACTER
30 Y = PEEK(X)
35 GOSUB 1000
40 GOSUB 60
42 X = X + 1
45 NEXT LN:REM DO NEXT LINE OF THIS CHARACTER

```

```

48 PRINT
50 NEXT CH:REM DO NEXT CHARACTER
52 END
60 FOR N=8 TO 1 STEP - 1
70 IF BIN(N)=0 THEN PRINT " ";GOTO 80
75 IF BIN(N)=1 THEN PRINT "*";
80 NEXT N
85 PRINT
90 RETURN
1000 FOR N=1 TO 8
1010 GOSUB 2000
1020 BIN(N) = BIT:REM SAVE THE BIT FOUND
1030 NEXT N
2000 Y1 = INT (Y/2)
2010 BIT = Y - Y1*2
2020 Y=Y1
2030 RETURN

```

If you wish, you can substitute line 75 with a cursor character instead of the asterisk. You can embed a cursor into the PRINT statement by pushing the ATASCII key (), then the space bar, then the ATASCII key again, just after the first quote and after the last one. This will make a more solid display.

Subroutines have many other possible uses. You will see more of them used in Chapter 8.

REVIEW OF CHAPTER 7

1. Subroutines can be a useful way of performing repeated sequences of statements.
2. A subroutine is simply one or more statements which terminate in the ATARI BASIC keyboard RETURN.
3. No matter from where in a program a subroutine is entered, it automatically knows where to go back to when the RETURN is issued.
4. It is possible to structure a program so that one subroutine calls another, which calls another, etc.
5. If you develop a number of different routines, each of which

works by itself, and if you want to do a program that calls for a number of these functions, it may be possible to construct a program as a series of subroutine calls. This may be the next best thing to a program that writes programs.

CHAPTER

8

Getting Colorful, Getting Noisy

This chapter will give you an introduction to the capabilities of the ATARI Home Computer in both graphics and sound. Until now, we have only been using one color, specifically white on blue for the printed background. The ATARI Home Computer has a wide range of colors that can be displayed, and a large group of graphics modes in which the displays can be produced. Now we can look at some of these capabilities and see how they might be included in some of your programs.

GRAPHICS CAPABILITIES

The text screen that we have been using is called graphics 0. Its basic capabilities are 24 lines of 40 characters each. In addition to being used as a text printing screen, it may also be used as a graphics screen for drawing lines and shapes. But this subject will be discussed later, after demonstrating some of the other graphics modes and their color capabilities. Plotting and drawing in graphics 0, 1, and 2 are a bit confusing and it would be better, perhaps, to get the color handling straight before looking at the "exceptions." For now, let's just treat the graphics 0 mode as a

text screen and show the ATARI BASIC keywords that may affect what you see on the screen.

You have already used the POSITION and PRINT statements on this text screen in the menu chapter. Just for a refresher, try the following program. It demonstrates the outer limits of the positioning area and labels those limits. It is for convenience that $X=38$ was chosen here; sometimes when you select 39, the Screen Editor will add a blank line where it isn't wanted.

Note that this program will be used as the basis for the explanation and demonstration of *all* of the graphics modes the ATARI Home Computer can display (from 0 through 11, anyway). All that will be done is to change a couple of lines here and there, then RUN the program again.

You will see that some of the graphics modes have a higher resolution than others. This means that some graphics modes put more dots on the screen, which means that you can make pictures or characters of finer detail in these modes. Here is the demonstration program for graphics mode 0:

```

10 DIM X(4),Y(4),P(4),Q(4)
20 GRAPHICS 0
30 FOR N = 1 TO 4
35 READ V:X(N)=V
40 READ V:Y(N)=V
45 NEXT N
85 FOR N = 1 TO 4
90 READ V:P(N)=V
95 READ V:Q(N)=V
100 NEXT N
110 FOR N = 1 TO 4
120 POSITION X(N),Y(N)
130 PRINT "+ ";
140 NEXT N
150 FOR C = X(1)+1 TO X(2)-1
160 POSITION C,Y(1):PRINT "- ";
170 POSITION C,Y(3):PRINT "- ";
180 NEXT C
200 FOR N = 1 TO 4

```

```

210 POSITION P(N),Q(N)
220 PRINT X(N);", ";Y(N)
230 NEXT N
1000 DATA 2,2,38,2,2,20,38,20
1100 DATA 3,3,34,3,3,19,34,19
9999 END

```

What this program does is outline the boundaries of the screen, then print, near the corners, the positions at which the boundaries have been established.

The first FOR-NEXT loop reads the values of the X and Y positions where the plus signs are to be placed, from DATA statement 1000. The second FOR-NEXT loop reads where on the screen the X and Y identification numbers are to be placed. The third FOR-NEXT loop is used to actually position the data, and the last one is used to label it correctly. Note that we are printing one of the things that was also used for a plotting coordinate.

The color you see on the screen is a light blue. It can be changed to other colors and other brightnesses. However, instead of explaining colors with graphics 0, it is much easier to demonstrate the color capabilities using graphics 1. Try the demonstration program that follows. It shows five different colors on the screen and will allow us to more easily explain about "color registers" and their use.

If you want to simply leave the previous program in memory, you can just type this one in directly. Then, add the line:

```
1 GOTO 4000
```

and RUN the program. As mentioned earlier, additional modifications will be made to the previous program, so this piece can be demonstrated on its own:

```

4000 GR.1:REM NOTE THAT THIS ABBREVIATION IS OK
4010 DIM Z$(20),G(4),H(4)
4020 Z$ = " COLOR REGISTER "
4021 G(4) = 160:G(3) = 128:G(2) = 32:G(1) = 0
4022 H(4) = 96:H(3) = 128:H(2) = - 32:H(1) = 0
4030 FOR R = 0 TO 3
4040 FOR L = 1 TO LEN(Z$)

```

```
4050 IF ASC(Z$(L,L))<65 THEN P = H(R + 1):GOTO 4070
4060 P = G(R + 1)
4070 PRINT#6;CHR$(ASC(Z$(L,L)) + P);
4080 NEXT L
4090 PRINT#6;CHR$(R + 48 + P)
4100 NEXT R
```

When you RUN this program, your display should appear as follows:

```
COLOR REGISTER 0
COLOR REGISTER 1
COLOR REGISTER 2
COLOR REGISTER 3
```

in the top section of the screen, and

```
READY
```

at the bottom.

What you have just produced with this program is a split screen graphics display, which is what is called for by ATARI BASIC, GRAPHICS 1. Note that if you didn't want the split screen, and that the entire screen was to appear as the upper part, you would have added 16 to the graphics mode number. In other words, line 4000 would have been GRAPHICS 1 + 16. Try it if you wish. However, for the discussion that follows, please replace the original line. Here is how the display got there:

In line 4000, you set up the graphics mode that you see. Line 4010 reserved space for the character string and for the "offset" numbers. Lines 4021 and 4022 set the values of the offsets. These offset numbers will be discussed shortly. Line 4030 sets up a FOR-NEXT loop for writing the register number to the screen (register 0, 1, 2, and 3). Line 4040 sets up another FOR-NEXT loop to individually treat each one of the characters to be sent to the graphics area of the screen. The rest of the program is dedicated to "adjusting" the value of the character to be printed so that it appears in the correct color.

The graphics area of the screen is treated differently from the text area (the last four lines you see in graphics mode 1). You

can print anything into the text area just using the regular PRINT statement. Each time you print a line, the text screen "scrolls" upward within that 4-line area just like it does in graphics mode 0; it is just that now there are only four lines to handle this way.

Graphics mode 1 is also a text graphics mode. You can print any letter text into this area by using the PRINT statement, but instead of the word PRINT alone, you must use the "word" PRINT #6;. Examples of this are:

```
PRINT#6;"VALUE IS ";N
PRINT#6;A;" ";B
PRINT#6;" COLOR REGISTER ";R
```

Note that the PRINT#6; command cannot be used in direct command mode once the program is ENDED; it will cause an ERROR 133. This is because this command is a special one in which the #6 represents an "IOCB number," which stands for Input/Output Control Block. The statement PRINT#6; says to ATARI BASIC that the character which would normally have been sent to the screen by the normal PRINT statement is to be sent through the IOCB to whichever device is being controlled by the IOCB. In this case, the device is the graphics area of the screen. The Screen Editor, then, is controlling only the bottom four lines, and it will scroll, and so forth, normally.

This program modified the way the PRINT statement was used because it now allows us to look at how certain of the ATARI BASIC commands affect the color that appears on the screen. At the moment, what you should see on the screen has COLOR REGISTER 0 printed in a gold color, COLOR REGISTER 1 in light green, COLOR REGISTER 2 in blue, and COLOR REGISTER 3 in a light red. (Your television may be adjusted somewhat differently, but these should be similar to the colors you see.) This was printed for you so that you could see how the SETCOLOR statement works.

There are five different "Playfield" color registers in the ATARI Home Computer, and four different "Player/Missile" color registers as well. We won't be doing much in this book with the last four registers, except for a demonstration in graphics modes 9 and 10.

A register is simply one of the memory locations within the computer where some piece of information can be stored. In this case, each register controls the part of the machine that controls the color on the screen when a certain color register is selected. You will see more about this later (see PLOT, DRAWTO).

The first four color registers are called 0, 1, 2, and 3. Their contents determine the color that will be shown on the screen when a particular "Playfield" data is shown on the screen. The complicated PRINT#6; statement you saw earlier was just used to make sure that the items we were printing would come out in the colors we chose.

Note that in graphics mode 1 you can only print a limited part of the character set. This is shown in Table 9-6 of the ATARI BASIC Reference Manual, labeled "Internal Character Set." Basically speaking, the characters that can be printed in this mode (without any character set redefinition) are those from internal character numbers 0 through 63. This covers the alphabet (all capital letters), all numbers, and most basic punctuation marks. This is a set of 64 characters representing the internal character set values from 0 to 63 when the PRINT#6; statement is interpreted. Each memory location can contain any number from 0 to 255, which is four times the number of characters that graphics mode 1 will allow to be printed. What happens to the other 3/4 of the number values? Well, you see the result on the screen. It prints the same character set, but in four different colors. What the "offset" numbers did was to adjust the value of each character when printed to make sure it would appear in the right color.

Now, we can use this information to explain what you see on the screen. (Make sure the program to display the color registers is still there, displayed on the screen.) Notice the bottom section of the screen. This still looks like the text screen mode we used prior to this chapter. It contains white letters on a blue background. Now look up into the graphics area of the screen. Notice that the data "COLOR REGISTER 2" is also printed in blue. This is because graphics mode 0 uses the color contained in color register 2 for the color which appears as its background display.

Now we can look at the ATARI BASIC command SETCOLOR. You use the SETCOLOR command to put a value in the color

register. Any item on the screen that is supposed to be displayed using that color register will appear in the color and brightness you have put into this color register. The way the SETCOLOR command appears is as follows:

SETCOLOR REGISTERNUMBER,KOLOR,BRITENESS

(I used a rotten spelling because ATARI BASIC uses the real word COLOR as one of its commands and I didn't want to confuse either you or ATARI BASIC.) REGISTERNUMBER is a number in the range 0, 1, 2, 3, or 4. (You have not seen register 4 yet; it will be shown soon.)

KOLOR stands for any number from 0 to 15. These are the basic hues that can appear on the screen. Each of them can appear in eight different brightness levels, allowing, for most modes, the effect of 128 different colors that could be chosen to appear on the screen. (Graphics mode 1 allows five different colors on the screen simultaneously.)

BRITENESS is any *even* number from the choices 0, 2, 4, 6, 8, 10, 12, 14, where 0 means not really dark. The only color that will allow you an almost black is color 0 (gray, or no color). Its lowest brightness level gives the appearance of black (no brightness). The maximum value for brightness is 14, which means so bright it is almost white. Odd numbers are treated the same as the next lowest even number, so only even values should be used to distinguish the brightness levels. Try the typical direct command:

SETCOLOR 2,6,2

Notice that it changed the color of the data "COLOR REGISTER 2" into a reddish purple. Notice that it also changed the color of the background of the text area of the screen to the same color. This demonstrates that graphics mode 0 gets its background color from color register 2. Now issue the direct command:

SETCOLOR 1,6,10

This means use color register 1, change the color to purple, but make the brightness much higher than that of register 2. Notice the difference between the brightness of the display of color registers 1 and 2. Now issue the command (but before you touch

RETURN, make sure you are watching the text area of the screen at the bottom so you can watch what happens to the text display):

```
SETCOLOR 1,6,14
```

Did you see what happened? The text got much brighter against the background. This indicates that graphics mode 0 gets its background color from color register 2 and its brightness from color register 1.

Here is a chart showing the colors selected when the SETCOLOR command KOLOR is used:

KOLOR Value	Color Selected
0	gray
1	light orange (gold)
2	orange
3	red-orange
4	pink
5	red
6	purple-blue
7	blue
8	blue
9	light-blue
10	turquoise
11	green-blue
12	green
13	yellow-green
14	orange-green
15	light-orange

You may experiment by issuing various direct SETCOLOR commands, but notice that for each time you do, it might be best to experiment only with color registers 0 and 3. This is because 1 and 2 affect the text screen appearance, and if the brightness of 1 is close to the brightness of 2, and the color of 1 is close to the color of 2, it might be next to impossible to read the text area!

Instead of all of that typing, you could try the following program. Again, it is short and can share the memory space with the

other two programs you have already entered. To perform this program, simply type RUN once you have the program typed in:

```

5000 PRINT "PRESS ANY KEY TO CONTINUE"
5005 FOR N = 0 TO 15
5010 FOR M = 0 TO 14 STEP 2
5020 POKE 764,255
5030 C = PEEK(764)
5040 IF C = 255 THEN 5030
5050 FOR X = 1 TO 50:NEXT X:REM SMALL DELAY
5060 SETCOLOR 0,N,M
5070 PRINT "COMMAND WAS: SETCOLOR 0,";N;",";M
5080 PRINT
5090 PRINT "PRESS ANY KEY TO CONTINUE"
5100 NEXT M
5110 NEXT N

```

Now RUN the program and follow instructions. There will be 128 different color combinations displayed in the graphics area line labeled "COLOR REGISTER 0."

We have already established what color registers were used for the display of graphics mode 0. We have just shown also how different text characters can be displayed in the graphics mode 1 screen. If you use the adjustment factors shown in the lines 4000–4100 program section, you will be able to PRINT#6; into the graphics 1 top section in any one of four different colors you select yourself. But, we have stated earlier that graphics mode 1 allowed five colors to be displayed at one time. Where is the fifth color? Well, the answer to that is the same as the answer to "How do I use color register 4?"

Color register 4 establishes the background color for all graphics modes. This is different from the background of graphics mode 0, because its background is really the color of Playfield number 2. It really looks as though the color of Playfield number 2 has been drawn as a set of squares on the screen, butting up against each other so well there is no space between them. The characters in graphics mode 1 appear then as "inverse video" plots against the graphics mode 2 color.

Let's try a couple of experiments to show what color register 4 does. This assumes that the display for the previous program is still on the screen. Type the direct command:

```
SETCOLOR 4,2,0
```

This says to use color register 4, set it to orange, and make it a dark orange. That has lit up the whole screen in this orange color! Thus, color register 4 controls the background color.

Notice that this color exists around the outer edges of the text area also. If you were to now type the command GR.0, then again the command SETCOLOR 4,2,0, you would see that orange color now is the border around the text area here. Border/background are, therefore, controlled by color register 4. Now change line 4000 of the sample program to read:

```
4000 GRAPHICS 2
```

and RUN the program again.

Graphics 1 gives you a graphics area that is normally used for printing text material, and a text screen of four lines at the bottom. Graphics 2 does the same, but in different sized letters. Graphics 1 will allow you to print up to 20 lines of these larger letters (20 characters per line instead of 40) or, if you are using graphics 1 + 16 (deletes the text area), lets you use 24 lines of characters.

In graphics 2, the letters are twice as tall as in graphics 1, and twice as wide as in graphics 0. So graphics 2 allows up to 10 lines of 20 characters each, or graphics 2 + 16 allows up to 12 lines.

Each time a PRINT#6; command terminates without a semi-colon, it says there is an end of line there and that the next line to be printed using the PRINT command should appear on the next line. Examples are:

```
PRINT#6
```

This prints one blank line into the graphics area.

```
PRINT#6;"THIS LINE IS HERE"
```

This prints THIS LINE IS HERE, then whatever line comes next starts in the first position of the next line.

When you get to the bottom of the "page" in the graphics area, it does not scroll up like the graphics 0 area. Before adding any more lines of characters, you must clear the screen first. None of the other Screen Editor modes will work on this graphics area (cursor up, cursor left, and so forth), but the screen clear command *will* work, as:

```
PRINT#6;"␣"
```

which, as you may recall, was printed by the command sequence: quote **ESC CTRL + CLEAR** quote.

TRUE GRAPHICS

Now we come to the first "true" graphics mode, namely mode 3. It is called a true graphics mode because points can be plotted against a background, and lines can be drawn with those points.

Here is a complete reprinting of the original program shown at the beginning of this chapter, changed for use with graphics mode 3 instead:

```
5 REM GRAPHICS 3 DEMO PROGRAM
10 DIM X(4),Y(4),P(4),Q(4)
20 GRAPHICS 3
30 FOR N = 1 TO 4
40 READ V:X(N)=V
50 READ V:Y(N)=V
60 NEXT N
70 FOR N = 1 TO 4
80 READ V:P(N)=V
90 READ V:Q(N)=V
100 NEXT N
110 FOR N = 1 TO 4
120 COLOR 1
130 PLOT X(N),Y(N)
140 NEXT N
200 FOR C = X(1) + 1 TO X(2) - 1
```

```
205 COLOR 2
210 PLOT C,Y(1):COLOR 3:DRAWTO C,Y(3)
230 POSITION C,Y(3)
240 PRINT#6;" - ";
300 NEXT C
255 END
1000 DATA 0,0,39,0,0,19,39,19
1100 DATA 1,1,15,1,1,18,15,18
```

RUN this program. Graphics 3 has a resolution of 40 blocks across by 20 blocks down. (Graphics 3+16 deletes the text window and gives a resolution of 40 by 24.) Each of these blocks can be in any one of four possible colors, where the color numbers are not exactly as shown in the previous example. Specifically, when you want to select a color to use, the color number does not correspond exactly to the color register you set in the first place. This will be explained later.

In all of the graphics modes from 3 through 8, when you want to put a block of color on the screen (or a point of color—it depends on how small the blocks are), you must first tell ATARI BASIC which color you want to work with. It is like selecting a pen that contains this color of ink. Each time you put that pen down on the paper, it produces a blotch of that color. Once you select a new pen, you can produce a new color. This is done with the ATARI BASIC statement COLOR. An example of the COLOR statement is:

```
205 COLOR 2
```

which means to select a color from the color register group to use for any time an item is placed on the screen. The following chart details what number specified in the COLOR statement selects which color register. Unfortunately, if you say COLOR 1, it does *not* select the color you specified in the SETCOLOR statement for register 1, so you must refer to this table or simply remember which goes with which. Note that this information is also available in the ATARI BASIC Reference Manual in a chart titled "MODE, SETCOLOR, COLOR TABLE."

FOR MODES 3, 5, AND 7:

The command COLOR 0 selects COLOR REG.4
 The command COLOR 1 selects COLOR REG.0
 The command COLOR 2 selects COLOR REG.1
 The command COLOR 3 selects COLOR REG.2

Graphics modes 3, 5, and 7 are the four color graphics modes. When you are trying to place points on the graphics area, you can "PLOT" points in one of three possible colors. So you could think of these as colors 1, 2, and 3 (corresponding to color registers 0, 1, and 2). The PLOT command is the instruction to ATARI BASIC to place a colored block onto the screen at the X,Y coordinates specified with PLOT. That block is to be of the color that was last specified in the COLOR statement. The PLOT statement basically has the same effect as the combination of:

POSITION X,Y

and

PRINT CHR\$(KOLOR)

An example PLOT statement looks like this:

PLOT X,Y

which assumes that it has been preceded somewhere by a statement of:

COLOR x

so it knew what color to use. If x is not specified, COLOR 0 is used (background).

For these graphics modes, we can consider COLOR 0, being the background or border, as "no color." This essentially means that if you plot in other colors against this background, then later use COLOR 0, those points now plotted will disappear against the background.

One of the things you may notice about the preceding program is that line 240 is still the same as it was before, in the mode 0

demonstration program. Even though we are now using COLOR and PLOT, the POSITION and PRINT statements still function in this graphics mode. Notice also that in line 205, a color is selected for the top line of the drawing. In line 210, a color is selected for the middle section of the drawing. But a different color appeared along the bottom line. This is because we are using the PRINT statement with the character "-". The effect of using a character with this graphics mode is to select the color automatically without a separate COLOR statement. All you need to do is to print a character that has the right combination of "bits" and it will perform the same task as separately specifying a COLOR and then PLOTting a point. The letters P,Q,R, and S work just fine for this sort of thing. For example, in line 240, substitute a P for the hyphen and RUN the sample program again. Now the bottom line is "empty" because it has been plotted in the background color. The character P selected "COLOR 0" which is really color register 4 per the chart shown earlier. This is the background color.

Likewise, the letters Q, R, and S when printed into this mode, or modes 5 or 7, have the same effect as selecting COLOR 1, 2, or 3, respectively, prior to a PLOT statement. Try them if you wish. Now change line 20 to read:

20 GRAPHICS 4

and RUN the program again. You will see the display shrink to 1/4 of its original size. You can plot twice as many points in both the X and Y directions in graphics 4 as you can in graphics 3.

One other command that is new to this demonstration program is the DRAWTO command. It takes the last graphics point plotted, wherever it might be on the screen, and draws a best possible connection between that last point and the new point specified, using whatever color is in effect at the time. It does not matter in which color the last point was plotted. For example, a point may be plotted in the background color, then a line may be drawn in another color from that "invisible" point to a new point, in a visible color if desired. An example would be to add the following lines to the latest program, which now demonstrates graphics 4:

```
255 COLOR 0:REM USE BACKGROUND COLOR
300 PLOT 60,0:DRAWTO 65,35
310 COLOR 1:DRAWTO 25,35
```

and RUN it again. Notice that only one line segment has been made visible on the screen. The first segment kept the computer busy for a little while, but we only got to see the second line segment because the first was plotted in the background color. Now change line 20 to read:

```
20 GRAPHICS 6
```

and RUN it again.

Both graphics 4 and 6 are two-color modes. Each can show just the background color (specified by SETCOLOR 4, but selected by the command COLOR 0), and the color from color register 0 (specified by SETCOLOR 0, selected by the command COLOR 1). Using mode 4, you can plot 80 by 40 dots on the screen in the single color against any selected background. Using mode 6, you can plot 160 by 80 dots, as you can see from the change in the size of the display. If you specify mode 4 + 16 or mode 6 + 16, the text window disappears and you can plot 80 by 48 dots (mode 4 . . . now mode 20), or 160 by 96 dots (mode 6 . . . now mode 22) in that single color. Now change line 20 to read:

```
20 GRAPHICS 5
```

and RUN it again. This gives you the same size display and the same number of plotting points as mode 4, but four colors are allowed.

If you now change line 20 to read:

```
20 GRAPHICS 7
```

and RUN it again, it gives you the same size display and number of points as mode 6, but four colors are allowed. In each case of four colors used, the color selections are made as shown in the chart for mode 3.

Modes 6 and 7 have the ability to produce 160 dots across by

80 dots down (or 96 if mode + 16 is specified). Why would anybody use a two-color mode when a four-color mode is available having the same number of dots? Well, the four-color mode takes nearly twice as much memory to store the screen image. In some cases, when the program is very large, it may be critical that the memory usage be kept to a minimum for the display.

Anyway, just to prevent you from thinking that the different displays for the higher numbered modes are simply smaller, you should make a change in the DATA statement, line 1000, so that it can reflect the full size of the display. The contents of the DATA statement for each mode can include the outer limits of the mode, such as, for example, mode 7.

Mode 7 can plot 160 points in the X direction and, if the text window stays there, 80 points in the Y direction. The upper left-hand corner of the graphics "window," as it is called, is at 0,0, which means $X=0$, $Y=0$. Therefore, the plotting capability is from 0,0 to 159,0 as the upper left and right corners, to 0,79 to 159,79 as the lower left and right corners. Make sure that line 20 reads

```
20 GRAPHICS 7
```

and then change the DATA statement to include all of the limits for this mode:

```
1000 DATA 0,0, 159,0, 0,79, 159,79
```

You don't have to put in the spaces, it was just done to make it easy for you to see the relationship between the data and how it was included in the statement.

Now RUN the program again. This time the graphics 7 screen is drawn to its limits along the top, bottom, and side edges. If you have still included lines 255 through 310, you *will* see the formerly invisible line now. That is because it is plotted against a color which is not the background. Finally change line 20 to read:

```
20 GRAPHICS 8
```

and RUN it again. This mode has the highest resolution of all, specifically 320 dots across by 160 dots down (192 down if graphics 8 + 16, which is graphics 24, is used). This mode is also

limited in the color selections, but its limitations are more similar to graphics 0 than any other mode. In particular, graphics 8 can have only one color, with two different luminances or brightness values. The color of the background for mode 8 will be that specified in color register 2, with the brightness of the background also determined by that register. The graphics point that is plotted will have the same color as the background color (from register 2), but will have the brightness specified in register 1 (refer to the early discussions of mode 0, for example).

Practice, if you wish, using the POSITION or PLOT statements, along with the DRAWTO statement to form lines in the various graphics modes. One such application might be the drawing of curves. Here is a sample program just to get you thinking of other possible applications:

```

1 REM: THIS PROGRAM DRAWS A CIRCLE
2 GR.7
3 T8 = 80:T4 = 40:T2 = 20
4 COLOR 1
5 SETCOLOR 1,14,14
10 FOR V = 0 TO 6.28 STEP 0.06
20 X = T8 + T2*SIN(V)
30 Y = T4 + T2*COS(V)
40 PLOT X,Y
50 NEXT V

```

In this program, T2 specifies the radius of the circle, and T8 and T4 specify the position of the center of the circle in X and Y coordinates, respectively.

Now, a final note before leaving this brief introduction to graphics is the use of PLOT and COLOR in modes 0, 1, and 2. It was specified that these graphics modes were the exceptions to the rule that the COLOR statement normally represented the selection of which color register was to be used to show something on the screen. Well, in modes 0, 1, and 2 the COLOR information is what is used to determine what is placed on the screen all right, but it is CHARACTER data instead of color-register data. This may be illustrated with this short program:

```

5 REM COLOR DATA IN MODES 0, 1, AND 2 DEMO
10 GR.0:REM COMPUTER WILL SPELL IT OUT WHEN LISTED
15 POKE 752,1:REM MAKE CURSOR VANISH
17 PRINT " ":REM GET RID OF EXTRA CURSOR
20 COLOR ASC("A")
30 PLOT 0,0
40 DRAWTO 39,20
50 POKE 752,0:REM CURSOR COMES BACK

```

You can repeat this for mode 1 also, with no changes other than line 10 to show mode 1, and line 40 changed to read:

```
40 DRAWTO 19,20
```

since each character is twice as wide and has only 20 characters across. Try mode 2 also, changing line 10 and then changing line 40 to read:

```
40 DRAWTO 19,10
```

SOUND CAPABILITIES

The ATARI Home Computer has a set of four sound generators that may be used to produce interesting sound effects. This book will explain how the sound generators can be used, then leave it up to you to experiment with various combinations of sounds to find those that might be useful.

The four sound generators are called 0, 1, 2, and 3. You can turn them on using the ATARI BASIC statement SOUND. These generators are interesting because any combination of one, two, three, or four may be going at the same time if you wish. Also, once you start them up, they keep going on their own until *you* switch them off or until a program ends. (The only other thing that normally affects them is any transfers of data to or from the disk or cassette, because parts of the sound generators are used for communicating with these devices.)

The SOUND statement is structured as shown in the following example:

```
SOUND VOICE,PITCH,DISTORTION,VOLUME
```

where VOICE is any number from the group 0, 1, 2, or 3. It tells which of the four sound generators should be given the command contained in the rest of the group of numbers.

PITCH is any number from 0 to 255. A high number results in a low note, a low number results in a high note. Table 10.1 of your ATARI BASIC Reference Manual contains the pitch values for the common musical notes for three octaves of the musical scale.

DISTORTION is any even number from 0 to 14, which says what kind of distortion should be applied to the tone this generator produces. A value of 10 gives an almost pure tone, where any other value gives varying degrees of distortion from a buzzing sound to a hiss.

VOLUME is a number from 0 to 15 which says how loud this generator should be. A value of zero effectively turns off the generator. A value of 15 is very loud, with other values representing sound levels in between. The ATARI BASIC Reference Manual cautions that if more than one sound generator is used, then the total of all of the VOLUME values should not exceed 32. This is to keep the system operating in a range where the tones produced can be sent out correctly without unintentional distortion. Try the following direct command:

```
SOUND 0, 121, 10, 8
```

This turns on sound generator 0. The tone it produces is middle C because the number 121 was selected for PITCH. The distortion is minimum (almost a pure tone) because the number 10 was selected for DISTORTION. The volume value is 8, roughly a "normal" volume level.

Use the Screen Editor cursor move controls to move the cursor up to the sound line, and change the value of the volume from 8 to 6, then press **RETURN**. Notice that the sound level decreases. But also notice that the generator continues to run. This allows you to set a value, then have the program do other things with the sound still running.

Now move the cursor up again, this time change the sound generator number to 1, and touch **RETURN**. Now two different generators are sending out the same note.

Now move the cursor up again, this time change the pitch

value to 144 and touch **RETURN**. Now there are still two generators, but this time they are playing different notes. You can experiment with many different combinations of PITCH, DISTORTION, and VOLUME.

Here is a short program that will vary just the PITCH, so you can hear what range of PITCH the computer can produce. Each of the four sound generators is used for all possible pitches. All four sound alike, so it doesn't matter which one you choose to use or in which order.

```

5 REM SOUND STATEMENT TESTER
10 FOR N=0 TO 3:REM N IS GENERATOR NUMBER
20 FOR M=1 TO 255: REM M IS PITCH
30 D = 10 : REM D IS DISTORTION
40 V = 8: REM V IS VOLUME
50 SOUND N, M, D, V
60 NEXT M
70 NEXT N

```

REVIEW OF CHAPTER 8

1. ATARI BASIC allows the screen to show many different kinds of displays. These displays are called by the ATARI BASIC statement GRAPHICS X, where X represents a number from 0 to 15 (ATARI 600XL, 800XL, 1200XL, 1400XL, 1450XLD) or 0 to 11 (ATARI 400/800). In addition, modes 1 through 8 can be presented without a "text window" by adding 16 to the graphics mode number (modes 17–24).

2. ATARI BASIC can control the color produced on the screen in various areas by the use of the statement SETCOLOR P,Q,R, where P is the color register number, Q is the actual color number, and R is the brightness of that color.

3. ATARI BASIC selects the color to be used to plot a point on a graphics screen or a character on a character screen using the COLOR statement. The value used with the COLOR statement, for example:

```
COLOR Z
```


must be 0 through 255 for graphics 0, 1, 2, 17, and 18; 0 through 1 for the two-color modes 4, 6, 20, 22, and 8; 0 through 3 for the four-color modes 3, 5, 7, 19, 21, and 23; and 0 through 15 for the special modes 9, 10, and 11.

4. The ATARI sound registers can be started using the ATARI BASIC SOUND command as:

SOUND REGISTER,PITCH,DISTORTION,VOLUME

where the REGISTER numbers are 0 to 3, the PITCH can be 0 to 255, the DISTORTION can be even numbers from 0 to 14 with a value of 10 giving little distortion, and the VOLUME can be 0 to 15 with 0 the softest and 15 the loudest.

REFERENCES

1. *ATARI® BASIC Reference Manual*. ATARI®, Inc., Sunnyvale, CA, 1979.
2. Poole, Lon. *Your ATARI® Computer*. Osborne/McGraw-Hill, Berkeley, CA, 1982.
3. Chadwick, Ian. *Mapping the ATARI®*. COMPUTE! Books, Greensboro, NC, 1983.
4. COMPUTE! Magazine. *COMPUTE!'s First Book of ATARI®*. COMPUTE! Books, Greensboro, NC, 1981.
5. COMPUTE! Magazine. *COMPUTE!'s Second Book of ATARI®*. COMPUTE! Books, Greensboro, NC, 1982.

INDEX

A

- Absolute cursor positioning, 151-153
- Absolute-value (ABS) function, 60-61
- Accumulators, 90
- AND operator, 42, 48, 50
- Arctangent (ATN) function, 64
- Arithmetic operations, 44-47
 - addition, 45
 - division, 46
 - exponentiation, 47
 - multiplication, 46
 - operator precedence, 47-50
 - subtraction, 46
- Array(s), 116-120, 134-136
 - index, 117-118
 - initializing, 134-135
- ASC function, 87-88
- ATARI Disk Operating System, 107-111
 - formatting a blank disk, 109-111
 - linking programs on disk, 111-114

ATASCII, 67

- key code conversion table, locating, 167-168
- AUTOREPEAT function, 15

B

- Binary numbers, 187-193

C

- Call, subroutine, structure of, 178-183
- Character strings, 67-90
 - ASC function, 87-88
 - CHR\$ function, 87-89
 - comparison features, 85-87
 - definition of, 67
 - LEN function, 75-78
 - saving space for, 68-70
 - STR\$ function, 78-79
 - VAL function, 78-79
- CLOAD command, 101
- Color registers, 197-205
- COLOR statement, 206-211

Common logarithm (CLOG)
 function, 62
 Compound statements, 37-38
 Computed GOTO, 52
 Cosine (COS) function, 64
 CSAVE command, 100-101
 Cursor control, 13-16
 absolute positioning, 151-153
 from within a program, 145-151

D

DATA statement, 134-141
 Decimal-to-binary conversion,
 187-193
 Deferred mode, 12
 Degrees (DEG) function, 63
 DIM statement, 68-70, 116-117
 Disk Operating System (DOS),
 107-111
 formatting a blank disk, 109-111
 linking programs on disk, 111-114
 DRAWTO statement, 208-212

E

END statement, 25, 183
 ENTER command, 104, 105

F

Flowchart, 93
 Flow narrative, 95
 FOR-NEXT loops, 121-134
 nesting, 124-127

G

Game controllers (joysticks),
 156-164
 GOSUB statement, 178-183
 GOTO command, 25
 Graphics capabilities, 195-212
 true graphics, 205-212

I

IF-THEN statement, 31, 40-44
 Immediate commands, 12-13
 Index, array, 117-118
 INPUT statement, 35-37
 Integer (INT) function, 52-54
 Internal key code, 164-165

J

Joysticks, 156-164

K

Keyboard, 13
 Key code
 conversion table, ATASCII,
 locating, 167-168
 internal, 164-165
 Keywords
 ABS, 60-61
 AND, 42, 48, 50
 ASC, 87-88
 ATN, 64
 CHR\$, 87-89
 CLOAD, 101
 CLOG, 62
 COLOR, 206-211
 COS, 64
 CSAVE, 100-101
 DATA, 134-141
 DEG, 63
 DIM, 68-70, 116-117
 DRAWTO, 208-212
 END, 25, 183
 ENTER, 104, 105
 EXP, 63
 FOR, 121-134
 GOSUB, 178-183
 GOTO, 25
 GRAPHICS (n), 195-212
 IF, 31, 40-44
 INPUT, 35-37
 INT, 52-54
 LEN, 75-78
 LIST, 23-25, 33, 103-105
 LOAD, 103

Keywords—cont.

LOG, 62
 NEW, 25
 NEXT, 121-134
 NOT, 48, 58
 OR, 43, 48, 50
 PEEK, 154
 PLOT, 207-211
 POKE, 154
 POSITION, 151-153
 PRINT, 12
 PRINT#6;, 199-205
 RAD, 63
 READ, 136, 141
 REM, 54
 RESTORE, 141
 RETURN, 179-183
 RND, 142-144
 RUN, 12, 23, 106
 SAVE, 102-103
 SETCOLOR, 199-204
 SGN, 58-59
 SIN, 63-64
 SOUND, 212-214
 SQR, 59-60
 STEP, 122-124
 STICK, 156
 STRIG, 156, 159, 163
 STR\$, 78-79
 THEN, 31, 40-44
 TRAP, 43-44, 98-99
 use of, 58
 VAL, 78-79

L

Length (LEN) function, 75-78
 Line length, 16-17
 logical line, 17
 physical line, 17
 LIST command, 23-25, 33, 103-105
 LOAD command, 103
 Loops
 FOR-NEXT, 121-134
 IF-THEN, 121
 nesting, 124-127

M

Math functions

ABS, 60-61
 ATN, 64
 CLOG, 62
 COS, 64
 DEG, 63
 EXP, 63
 INT, 52-54
 LOG, 62
 RAD, 63
 SGN, 58-59
 SIN, 63-64
 SQR, 59-60

Modular programming, 93

N

Natural logarithm (LOG) function, 62
 Nesting loops, 124-127
 NEW command, 25
 NOT operator, 48, 58

O

Operator precedence, arithmetic operations, 47-50
 OR operator, 43, 48, 50

P

PEEK statement, 154
 Physical line, 16-17
 PLOT statement, 207-211
 POKE statement, 154
 POSITION statement, 151-153
 Powers of 2, 187-189
 PRINT command, 12
 PRINT#6; command, 199-205
 Print-tab stops, 51
 Programming, 20-26
 definition of, 12
 Program planning, 92-99
 Program save and load, 99-114
 CLOAD command, 101
 CSAVE command, 100-101

Program save and load—cont.
 Disk Operating System (DOS),
 107-111
 formatting a blank disk, 109-
 111
 linking programs on disk,
 111-114
 ENTER command, 104, 105
 LIST command, 103-105
 listing programs to tape or
 disk, 103-105
 LOAD command, 103
 loading "named" files from
 cassette or disk, 103
 RUN command, 106
 SAVE command, 102-103
 saving "named" programs on
 cassette, 101-103
 saving "named" programs on
 disk, 103
 tokenized form, 99-100
 Prompt, 36

R

Radians (RAD) function, 63
 Random (RND) function, 142-
 144
 READ statement, 136, 141
 REM statement, 54
 RESTORE statement, 141
 RETURN statement, 179-183
 Round-off error, 52
 RUN command, 12, 23, 106

S

SAVE command, 102-103
 Scientific notation, 36
 Screen Editor
 accessing from within a
 program, 145-151
 character erase, 148-149
 clear screen, 146
 cursor down, 146-147
 cursor left, 147

Screen Editor—cont.
accessing from
 cursor right, 147
 cursor up, 147
 functions, 13-23
 character delete, 19, 22
 character insert, 19, 21
 clear screen, 23
 cursor down, 15
 cursor left, 15
 cursor right, 16
 cursor up, 14
 line delete, 18, 20
 line insert, 18, 19
 SETCOLOR command, 199-204
 Sign (SGN) function, 58-59
 Sine (SIN) function, 63-64
 Sound capabilities, 212-214
 Square-root (SQR) function, 59-
 60
 STICK function, 156
 STRIG function, 156, 159, 163
 Strings, character, 67-90
 ASC function, 87-88
 CHR\$ function, 87-89
 comparison features, 85-87
 definition of, 67
 LEN function, 75-78
 saving space for, 68-70
 STR\$ function, 78-79
 VAL function, 78-79
 Subroutines, 177-193

T

TRAP statement, 43-44, 98-99
 True graphics, 205-212

U

User input, maintaining control
 of, 164-175

V

VAL function, 78-79
 Variables, 26-30

More Books for ATARI® Owners!

ATARI® BASIC TUTORIAL

Leads you through the practical ins and outs of programming in ATARI BASIC, including color graphics and sound, on all ATARI home computer systems. Uses short, workable program elements that become more complex as your knowledge increases. Has many self-documenting programs. By Robert A. Peck. 224 pages, 6 × 9, comb. ISBN 0-672-22066-0. © 1983.

Ask for No. 22066\$12.95

MOSTLY BASIC: APPLICATIONS FOR YOUR ATARI®, Book 1

Thirty-eight fascinating and useful ATARI BASIC programs that run on ATARI 400, 800, or 1200XL computers! Includes 10 educational, 7 on business and investment, 6 for the home, 6 using graphics and sound, a Tarot card reader, and 4 utilities. By Howard Berenbon. 184 pages, 8½ × 11, comb. ISBN 0-672-22075-X © 1983.

Ask for No. 22075\$12.95

MOSTLY BASIC: APPLICATIONS FOR YOUR ATARI®, Book 2

Another goldmine of ATARI BASIC programs you can key into ATARI 400, 800, or 1200XL computers! Includes two dungeons as part of 11 educational programs; a monthly budget, food analysis, and weekly calendar plus 8 more home applications; a series on Money and Investment; 2 programs on ESP; and a Dungeon of Danger! By Howard Berenbon. 264 pages, 8½ × 11, comb. ISBN 0-672-22092-X. © 1983.

Ask for No. 22092\$15.95

THE KIDS' COMPUTER IQ BOOK

Usable by the whole family as a booster in computer literacy! Offers a basic foundation in computer science while it covers the hows and whys of today's computers and forms a good introduction to problem-solving and programming. By Eileen Buckholtz and Dr. Joanne Settel. 152 pages, 5½ × 8½, soft. ISBN 0-672-22082-2. © 1983.

Ask for No. 22082\$5.95

WHAT DO YOU DO AFTER YOU PLUG IT IN?

Complete tutorial covers microcomputer hardware, software, languages, operating systems, data communications, and more, followed by solutions to the practical problems you'll meet while using them. By William Barden, Jr. 200 pages, 5½ × 8½, soft. ISBN 0-672-22008-3. © 1983.

Ask for No. 22008\$10.95

HOWARD W. SAMS CRASH COURSE IN MICROCOMPUTERS (2nd Edition)

An actual self-study course in one lay-flat volume, completely updated with new chapters on 16-bit micros and BASIC programming, expanded coverage of applications software, new photos, and more. Lets you learn about microcomputers and programming fast! No previous computer knowledge necessary. By Louis E. Frenzel, Jr. 320 pages, 8½ × 11, comb. ISBN 0-672-21985-9. © 1983.

Ask for No. 21985 \$21.95

USER'S GUIDE TO MICROCOMPUTER BUZZWORDS

Handy quick-reference that provides an understanding of the basic terminology you need to become "computer literate." Contains many illustrations. By David H. Dasenbrock. 112 pages, 5½ × 8½, soft. ISBN 0-672-22049-0. © 1983.

Ask for No. 22049 \$9.95

HOW TO MAINTAIN AND SERVICE YOUR SMALL COMPUTER

Shows you simple procedures you can use to sharply reduce problems and downtime with your Apple® II, TRS-80®, or other small computer. Also shows you how to diagnose what's wrong, find the faulty part, make simple repairs yourself, and deal with the repair shop when professional servicing is necessary. By John G. Stephenson and Bob Cahill. 208 pages, 8½ × 11, soft. ISBN 0-672-22016-4. © 1983.

Ask for No. 22016 \$17.95

MEGABUCKS FROM YOUR MICROCOMPUTER

Shows you how to make money using your creative talents through your microcomputer, and includes details for doing your own writing, reviewing, and programming. By Tim Knight. 80 pages, 8½ × 11, soft. ISBN 0-672-22083-0. © 1983.

Ask for No. 22083 \$3.95

ELECTRONICALLY SPEAKING: COMPUTER SPEECH GENERATION

Learn to generate synthetic speech with a microcomputer. Includes techniques, a synthesizer overview, and advice on problems. By John P. Cater. 232 pages, 5½ × 8½, soft. ISBN 0-672-21947-6. © 1983.

Ask for No. 21947 \$14.95

USING COMPUTER INFORMATION SERVICES

Shows you how to use your microcomputer to communicate with the national computer networks. Clearly explains what's available, how to retrieve it automatically, how to use your computer as a powerful communications tool, and more. By Larry W. Sturtz and Jeff Williams. 240 pages, 5½ × 8½, soft. ISBN 0-672-21997-2. © 1983.

Ask for No. 21997 \$12.95

These and other Sams Books and Software products are available from better retailers worldwide, or directly from Sams. Call 800-428-SAMS or 317-298-5566 to order, or to get the name of a Sams retailer near you. Ask for your free Sams Books and Software Catalog!

Prices good in USA only. Prices and page counts subject to change without notice.

ATARI is a registered trademark of ATARI, Inc.



TO THE READER

Sams Computer books cover Fundamentals — Programming — Interfacing — Technology written to meet the needs of computer engineers, professionals, scientists, technicians, students, educators, business owners, personal computerists and home hobbyists.

*Our Tradition is to meet your needs
and in so doing we invite you to tell us what
your needs and interests are by completing
the following:*

1. I need books on the following topics:

2. I have the following Sams titles:

3. My occupation is:

<input type="checkbox"/> Scientist, Engineer	<input type="checkbox"/> D P Professional
<input type="checkbox"/> Personal computerist	<input type="checkbox"/> Business owner
<input type="checkbox"/> Technician, Serviceman	<input type="checkbox"/> Computer store owner
<input type="checkbox"/> Educator	<input type="checkbox"/> Home hobbyist
<input type="checkbox"/> Student	Other _____

Name (print) _____

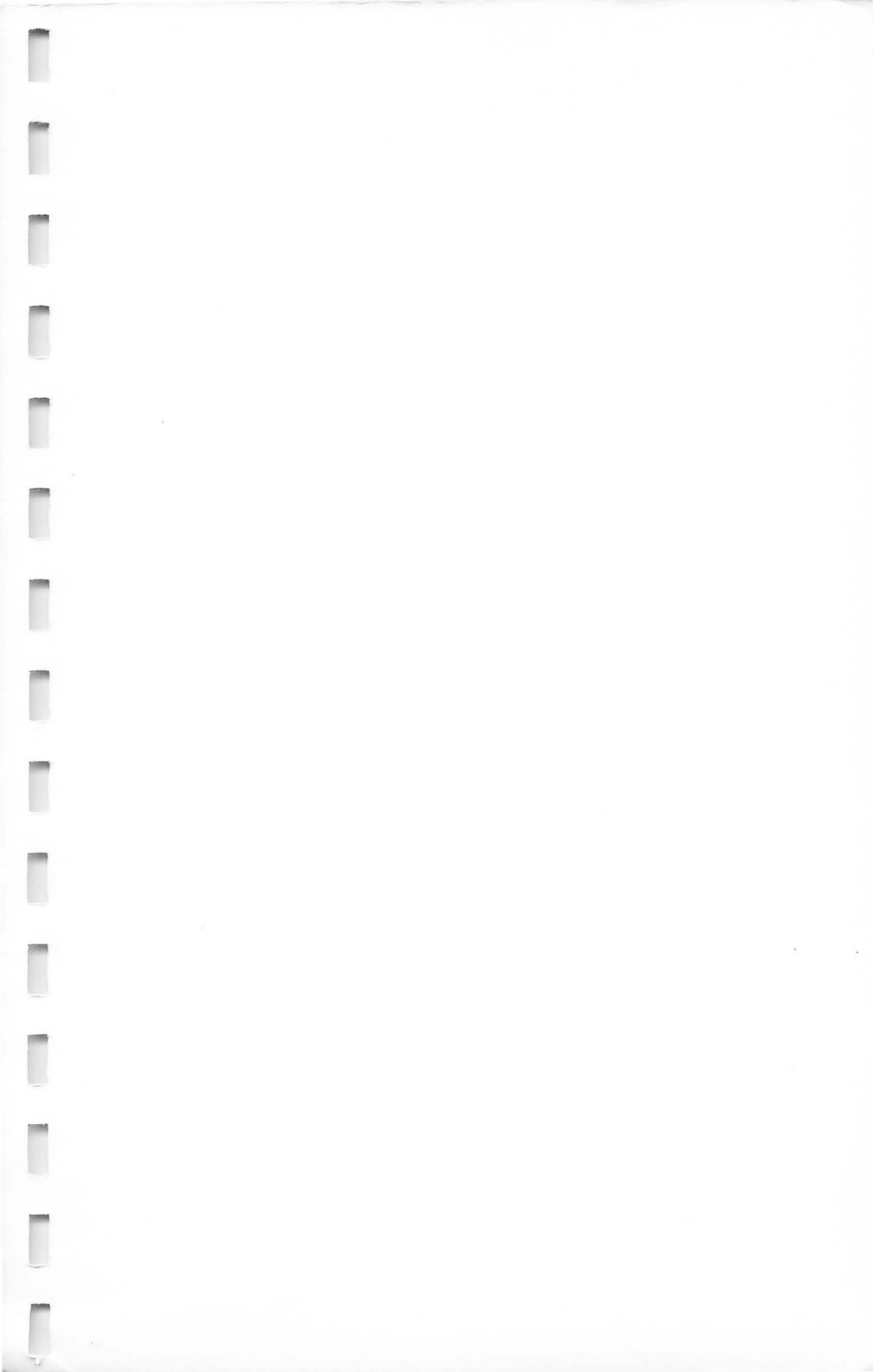
Address _____

City _____ State _____ Zip _____

Mail to: **Howard W. Sams & Co., Inc.**

Marketing Dept. #CBS1/80
4300 W. 62nd St., P.O. Box 7092
Indianapolis, Indiana 46206

22066



ATARI[®] BASIC Tutorial

- Written specifically for owners and users of ATARI[®] Computer Systems.
- Guides the reader step by step through the ATARI BASIC language—from the most elementary concepts to more advanced programming techniques.
- Follows a progressive format—each chapter builds on knowledge acquired in previous chapters.
- Concentrates on a user-interactive approach in order to have the reader working *with* the computer while progressing through the book.
- Enables users who have perhaps run games and prepackaged programs on their machines to learn how to produce useful programs of their own.
- Contains numerous examples of debugged, self-documenting programs, including programs to demonstrate the color graphics and sound capabilities of the ATARI Computer Systems.

Computer Direct

We Love Our Customers

Box 1001, Barrington, IL 60010