Basic ATARI BASIC®

for the 400, 800, and XL computers

JAMES S. COAN
& RICHARD KUSHNER

Basic ATARI® BASIC

JAMES S. COAN
& RICHARD KUSHNER

Equipment Needed

To use the programs in this book you will need the following:

- An Atari 400, 800, or 1200XL (each with the BASIC cartridge); or the Atari 600XL, 800XL, 1400XL, 1450XLD
- A TV (color recommended)
- An Atari 410 Program Recorder or an Atari 810 Disk Drive (both optional)

1	2	3	4	5	6	7	8	9	PRINTING
84	85	86	87	88	89	90	91	92	YEAR

Preface

The Atari family of computers is composed of very popular, inexpensive microcomputers with capabilities that just a few years ago were available only on machines costing thousands of dollars more. Attached to a color television, they offer incredible color graphics and sound capabilities. They contain a powerful microprocessor that can be directed by a number of different computer languages to perform many useful and entertaining feats. Connected to a tape recorder or disk drive, they permit the storage of data and programs for future use, thus flawlessly handling many filing, listing, and record-keeping functions. You can save programs that you write for later recall without having to type in all the program statements again. With a printer you can list programs and output the results of calculations in whatever form you desire, limited only by your ingenuity. These inexpensive computers can perform virtually all the same functions as their far more expensive and elaborate cousins. In addition, microcomputers are becoming an important part of educational programs in many elementary and secondary schools. It is clear that an early introduction to computing in the home will benefit all school-age children.

Large numbers of programs are available for use on your Atari computer. You can play all manner of entertaining, colorful games; you can perform complex financial calculations and display the results as printed tables, pie charts, or bar graphs; you can write a letter to your representative in Congress and use your printer to prepare it as though you had your own private secretary; you can interact with your computer through its keyboard, joysticks, paddles, light pen, or homemade widget. You can purchase programs to do all this and much more.

You can also write your own programs. There is really no secret to writing programs, no initiation rite that is required to enable you to open magical doors. The computer understands a limited set of instructions that can be combined to take the programmer from a problem in need of a solution to a program that solves it.

This book will introduce you to programming in general and to programming specifically on Atari computers. These are really two distinct items. Programming can be approached on a very general basis, with a discussion of how best to carry out certain commonly needed functions (for example, counting). The topics of how to structure programs for comprehensibility, readability, ease of use, and faster execution could all be addressed, with the result that you would then have a passing acquaintance with programming, but wouldn't know how to apply it to an Atari computer. On the other hand, the Atari computer could be approached by itemizing all its BASIC commands, with examples showing how they are used. You would know the meaning and uses of the individual commands, but not how to join them together into programs.

Neither of those approaches, however, would be fair. Anyone, including you, can write programs for an Atari. In addition to practical examples, discussions of

the powerful features incorporated into Atari computers and problems (with solutions available) to practice on, you need some guidance in how to write good programs that meet your needs. That guidance is what this book aims to provide. We will lead you through programming on the Atari, covering all the main features of your computer and offering many additional helpful hints as we go. We will give you some practice in developing your own programs and aid you in turning loose both your creative genius and the terrific capabilities of your Atari computer.

Atari BASIC is what this book is all about. Although there are at least half a dozen other programming languages available for the Atari, BASIC is by far the most popular. By using a set of simple English commands, BASIC permits the beginner to interact easily and comfortably with the computer. There is very little that you will ever want to do on your Atari that cannot be handled with BASIC. For those who have already had some contact with other versions of BASIC (and there are about as many versions as there are computers), we forewarn you that Atari BASIC has some distinctive features. These features will all be described as we develop our understanding of the BASIC language.

This book begins with the assumption that you have access to an Atari computer, either with BASIC built in or with a BASIC plug-in cartridge, and that you know how to hook up your computer and whatever peripheral devices you have. This information is available in the material that was packed with your computer.

We encourage you to experiment. There is no way that you can damage your computer by typing in an incorrect command. The very worst thing that can happen is that, under some circumstances, the computer will "lock up"—that is, it will refuse to respond to any key that is pressed (including, sometimes, the system reset key). If this happens to you, you will have to turn the computer off and then back on to regain control. In the process you will have lost any program that was in the computer memory. Since most of the programs in this book are very short, you can easily retype the lost information. You will make lots of mistakes and be greeted with many error messages. That's a sure sign that you are learning! It is rare for all but the shortest programs to run as expected the very first time.

We hope that what you learn from this book will lead you to develop many programs for your personal use and pleasure. Like most things in life, progress in programming comes through a series of little steps, building one on the other until you reach your goal. Don't just read the book. The only way to become proficient at BASIC programming is to try the examples in the book and work out the solutions to the problems at the end of various sections. Working with the book and the computer together will prove to be a much more rewarding experience than either of them alone. The "hands on" learning method will make an impression on your memory that will carry over into all your future programming efforts.

This book begins with short, complete programs that are carefully and gently built into larger programs that solve larger problems. Chapter 1 starts us on entering data and obtaining results from the computer. Chapter 2 introduces some ideas for planning a program. Chapter 3 introduces Atari graphics and more BASIC commands. Chapter 4 offers a potpourri of BASIC features and programming

techniques. Chapter 5 presents strings, and Chapter 6 covers numeric arrays and how to simulate string arrays. Chapter 7 is a collection of miscellaneous applications. The Atari tape recorder and its use in storing programs and sequential data files is the topic of Chapter 8. Chapter 9 discusses input and output using a disk drive and features random-access files. Chapter 10 covers sound generation and its use with music and sound effects. The final chapter of the book presents the multitude of Atari graphics modes in detail and some advanced graphics methods.

Each chapter is followed by a Programmer's Corner, which highlights special features or advanced programming ideas. Programmer's Corner 1 elaborates on the Atari screen editor and the graphics characters. The discussion of the editor continues in Programmer's Corner 2, where error trapping is also covered. Programmer's Corner 3 extends the discussion of graphics in that chapter and introduces BASIC keyword abbreviations, while Programmer's Corner 4 investigates the attract mode and improved input for programs. Programmer's Corner 5 shows how to check whether a key has been pressed and how to modify the character set. Sorting, making a built-in clock, and using the START, SELECT, and OPTION keys are the topics of Programmer's Corner 6. A menu program is featured in Programmer's Corner 7. Programmer's Corner 8 continues the theme of that chapter with a discussion about overcoming tape loading problems and automating tape loading. Programmer's Corner 9 presents a detailed description of the disk utility commands, and in Chapter 10 the Programmer's Corner shows how to translate sound into music. The final Programmer's Corner presents a technique for modifying the screen to display more than one graphics mode at one time.

We have also included eight appendixes for your use. Appendix A lists many memory locations useful to the BASIC programmer. Appendix B discusses several "bugs" in Atari BASIC. Appendix C contains a word description of the error code numbers that Atari BASIC uses. A useful reading list of magazines and books of value to Atari owners is featured in Appendix D. Appendix E contains a table of the decimal, graphic, and keystroke representations of all the characters that your Atari can display on the screen (the ATASCII codes). Appendix F discusses features unique to the XL series of Atari computers and how to use them in your own BASIC programs. An ordered listing of the titles of all the programs in this book is contained in Appendix G. Finally, Appendix H presents solutions to all of the even-numbered problems.

<div style="text-align: right">

James S. Coan
Richard A. Kushner

</div>

New Hope, PA
High Bridge, NJ

Contents

Chapter 1

Introduction to BASIC on an Atari Computer 1

PROGRAMMER'S CORNER 1

Chapter 2

Writing a Program 24

PROGRAMMER'S CORNER 2

Chapter 3

Beginning Atari Graphics and Much More ...   42

Chapter 4

Miscellaneous Features and Techniques 60

Chapter 5

Character Strings and String Functions 75

PROGRAMMER'S CORNER 5

Chapter 6

Arrays 95

PROGRAMMER'S CORNER 6

Chapter 7

Using What We Know: Miscellaneous Applications 121

PROGRAMMER'S CORNER 7

Chapter 8

The Tape 142

PROGRAMMER'S CORNER 8

Chapter 9

The Disk 154

PROGRAMMER'S CORNER 9

Chapter 10

Sound 188

PROGRAMMER'S CORNER 10

Chapter 11

Graphics, Graphics, Graphics 208

PROGRAMMER'S CORNER 11

Appendix A

Useful Memory Locations for the BASIC Programmer

Chapter 1

Introduction to BASIC on an Atari Computer

A *program* is a set of instructions that causes a computer to perform in a predictable way. The process of writing those instructions for a computer is called *programming*. We can write programs to accomplish an amazing variety of tasks. The Atari can perform a wide range of arithmetic operations. It can be programmed to play music on its built-in speaker or to draw graphs, in color no less. Paddles or joysticks can be used to provide a continuous range of responses by moving a lever or rotating a dial. We can even write programs to respond to a light pen drawing on a TV screen. There are many ways in which Ataris are being used to help students learn subject matter unrelated to computers. The same computer can be used to keep track of all kinds of data necessary in the operation of a small business.

Every instruction used in programs has its own precise definition; the total collection of these instructions is called a *computer language*. Each instruction in the language has a form associated with it. This form is called the *instruction syntax*. The syntax of each instruction that we enter into the computer must be one of those that the computer recognizes. For example, the computer will reject the instruction QUIT, whereas it will find END perfectly acceptable and will indeed end upon encountering the END instruction. Even though QUIT and END have similar meanings in English, the computer won't treat them the same. Words that make up the language are called *keywords*. END is a BASIC keyword.

The Atari is capable of working with several languages. The language presented in this book is Atari BASIC. It resembles the BASIC that was developed at Dartmouth College by John G. Kemeny and Thomas E. Kurtz. BASIC is designed so that people ranging from rank amateurs to advanced engineers can quickly and easily write programs pertinent to problems of their own interest.

1-1...Getting Started in Atari BASIC

There are a number of things that we need to know, all at once, to get going. The READY that appears when you first power up the computer and at the completion of most commands issued from the keyboard is your computer's sign that it is ready to perform your next assignment. The solid block that appears beneath the READY, called the *cursor*, indicates where on the screen your next typed input will appear. You can move the cursor around the screen by holding down the CTRL key and any one of the four keys with the arrows on them. If you tap an arrow key once, the cursor will move one space in that direction. If you hold down any of the arrow keys, after about two seconds the cursor will continue to move in that direction until you release the key. The powerful screen editing capabilities of your Atari computer will be covered more fully at the end of this chapter. In the meantime, let's write our first Atari BASIC program.

```
100 PRINT "HERE IS AN EXAMPLE"
110 PRINT "OF A PROGRAM IN"
120 PRINT "ATARI BASIC"
```

Program 1-1. Our first program in Atari BASIC.

There you have it; not much to it, but a lot of preliminaries were covered in getting even this far. Notice that we typed NEW before we began to type in our program. NEW erased any BASIC program in the Atari's memory and prepared the computer for new input. Naturally, you should never type NEW unless you really mean it. The old program is not recoverable. We will cover saving programs for future use in the next section.

Our program consists of three statements. Each statement is labeled with a line number. Line numbers may be any integer from 0 to 32767. After the second cursor, we typed

```
100 PRINT "HERE IS AN EXAMPLE"
```

and pressed the RETURN key. The Atari responded by moving the cursor down one line and back to the left edge of the display. At this point the computer has stored our line 100 in its memory. We then typed in the next two lines in the same manner. Each of these statements is an example of a PRINT statement. When the program is RUN, each PRINT statement is an instruction to the computer that something is to be displayed on its screen.

```
RUN
HERE IS AN EXAMPLE
OF A PROGRAM IN
ATARI BASIC

READY
```

Figure 1-1. Execution of Program 1-1.

Next, we typed RUN and pressed the RETURN key. In this case, as with the

NEW instruction, we did not assign a line number to the instruction. The presence of a line number means that the current line is to be stored for later use by the computer. The absence of a line number means that the computer will immediately process whatever is on the line as an instruction. The RUN instruction causes the computer to process the instructions of the program stored in the computer's memory. That is what is meant by "running a program."

The RUN command will begin execution of your program at the lowest-numbered line. We can also begin execution at some other line number, say line 120, by entering GOTO 120 and then pressing RETURN. Note, however, that in a complex program this may result in problems because information needed by the program has been bypassed.

When the Atari runs out of instructions in the stored program, it simply displays READY and the cursor and politely waits for you to tell it what to do next. If we now type RUN again the Atari will display the same three-line message on the TV.

Note the difference between the letter "oh" and the digit zero. The Atari uses an oval with a slash through it for the digit zero and an open oval for the letter "oh." You might type "ohs," zeros, and eights so that you can study them on the monitor or TV. You will find the zero key between "9" and " < " in the top row of keys, while the "oh" is in the second row from the top between the "I" and "P" keys.

Here is a way to change the displayed message:

```
11Ø PRINT "OF A PROGRAM"

115 PRINT "WRITTEN IN"
```

Figure 1-2. Changing Program 1-1.

We have changed line 110 by retyping it. We have inserted a new line, numbered 115. By choosing a line number between two existing line numbers, we have told the computer that we want line 115 to be processed after line 110 and before line 120. It is a good idea to allow intervals in your line numbering. This allows space for additions to your program in the future. It is also a good idea to number lines in regular increments, for example, by tens or hundreds, as this makes entering programs easier. No matter what order you type the program lines in, the computer will arrange them in increasing order. Now, if you tell the computer to follow the instructions of the new program, you get:

```
RUN
HERE IS AN EXAMPLE
OF A PROGRAM
WRITTEN IN
ATARI BASIC

READY
```

Figure 1-3. Execution of the modified Program 1-1.

We call the process of carrying out the instructions of program statements

execution. Thus, when we type RUN, we are telling the computer to execute the program.

At this point, we have a program that we have created in two distinct steps. We first entered three lines, and some time later we entered two lines. One of those two lines replaced a line of the earlier program, and the other added a new instruction line. The resulting program contains four lines. Let us now look at the program in its entirety by using the LIST instruction.

```
LIST

100 PRINT "HERE IS AN EXAMPLE"
110 PRINT "OF A PROGRAM"
115 PRINT "WRITTEN IN"
120 PRINT "ATARI BASIC"
```

READY

Figure 1-4. Demonstrate LIST.

The instructions NEW, RUN, and LIST are commonly called *commands* because they are used to command the computer to manipulate the program as an entity rather than perform a program instruction.

What happens when we make typing errors? If we type LOST instead of LIST, the Atari will display the following error message:

```
ERROR- LOST
```

No harm has been done, the computer just doesn't understand what you want it to do. You simply have to correct the request and retype. If you type

```
100 PRIMT "HERE IS AN EXAMPLE"
```

you will get the message

```
100 ERROR- PRIMT "HERE IS AN EXAMPLE"
```

with the left quotation mark appearing as an inverse display. Once again, all you have to do is retype the line correctly and nothing is lost. You can mess up your program with input errors, but you can't hurt the computer. We will discuss Atari error messages as they come up. Appendix C lists the most common error numbers encountered in BASIC and their meanings. A complete list is included in your Atari Reference Manual.

We can look at a single statement by using an extension of the LIST command.

```
LIST 100
```

will display only line 100 of our program, if it exists. LIST 100,200 will display all of the lines in our program from 100 to 200, inclusive. Now retype the line and reexecute the program. If we type

```
120 PRINT "APPLE BASIC"
```

when we meant to type

```
120 PRINT "ATARI BASIC"
```

we have a different kind of error, which the computer will never find for us. Similarly, if we instruct the computer PRINT B when we really meant PRINT A, the computer will not inform us of our mistake. The value of B will be displayed where we expected to see the value of A. Thus it is important to evaluate our results for correctness.

...SUMMARY

A computer language is a defined set of instructions that convey specific meanings to the computer. In BASIC on an Atari, each instruction of a program begins with a line number. The PRINT statement is used to display a message on the computer monitor.

The NEW command prepares BASIC for a new program, the RUN command causes the Atari to carry out the instructions of the program stored in its memory, and the LIST command displays the stored program on the monitor. LIST 100 displays the line numbered 100, while LIST 100,200 displays all lines in the interval from 100 to 200, including 100 and 200.

1-2...The Atari Editor

Your Atari BASIC comes equipped with a very powerful editor for making changes in what appears on your TV screen. Your programming efforts will be greatly facilitated if you understand and use these commands. We have already described how the CTRL key, in conjunction with the arrow keys, can be used to move the cursor around the screen. We will soon see how useful this can be in editing a BASIC program.

There are a number of other commands available to us from the keyboard that are also useful in program editing. The SHIFT and CAPS/LOWR keys are useful in getting into and out of lowercase. BASIC will only accept commands that are typed in uppercase, so anything you type first appears with uppercase letters. You may, however, want to print out a message on the screen in lowercase, or a mixture of upper- and lowercase. Pressing the CAPS/LOWR key once will put you into lowercase. Your Atari keyboard then functions just like a typewriter—letters are typed in lowercase unless you hold down the SHIFT key while you press the letter key. If you want to return to uppercase only, hold down the SHIFT key while you press the CAPS/LOWR key. Try the following example (then press RETURN):

```
PRINT "This is UPPERCASE and lowercase."
```

Not bad, huh? This ability to function much like a typewriter is very useful in word-processing applications. Also, displays using only uppercase letters are difficult to read.

We have here used the *immediate mode* of communication with the computer. Any time you type a command with no line number, the computer will execute it immediately. For example, when we used PRINT above, the computer carried

out our request at once. Any instruction that can be used in a program statement can also be used in the immediate mode.

The CTRL key allows each key to function in more ways than one. For example, we already saw how to use CTRL with the arrow keys to move the cursor, while pressing the same keys alone produces the characters −, +, *, and =. For entering and editing programs we will find the following combinations of keys to be of great value:

CTRL and CLEAR/ < will clear the screen and put the cursor in the top left corner.

CTRL and INSERT/ > will open up one space in the typed line each time it is used. You can use this to make room for inserting instructions within a line.

CTRL and DELETE/BACK S will delete the character immediately following the cursor. Also note that using the DELETE/BACK S key alone will delete the character just before the cursor and move the cursor one space to the left. The text to the right of the cursor, however, is not moved back to fill in the space created by erasing one character.

CTRL and CAPS/LOWR will switch the keyboard into its graphic character mode. We'll make use of this mode at the end of this chapter.

SHIFT and DELETE/BACK S deletes whatever line the cursor is on; continuing to do this will erase lines as they are brought up the screen to the line the cursor is on. If the line deleted in this way, however, happens to be part of a numbered program line, it will not be deleted from the computer's memory.

To delete a line from a program, simply position the cursor on an empty line, type in the number of the line to be deleted and press RETURN. Try LISTing a program after doing this and you will see that the line has, indeed, been deleted.

The RETURN key has been used here to tell the computer that it should accept the line you have just typed in and either store it in memory (if it has a line number) or execute it (if it doesn't). The computer will accept the line even if the cursor is in the middle of the line when you hit RETURN. For example, the following line has a typing mistake:

```
PRINT "I don't spelll very well."
```

To correct the mistake, you could retype the entire line, making sure that you corrected the error without making any other errors in the process. But there is a much simpler method that is far less likely to make things worse than they were in the first place. First move the cursor up to the line that is the computer's display of your mistake and use SHIFT and DELETE/BACK S to remove it from the screen (we'll explain why we do this shortly). Then use the CTRL and arrow keys to move the cursor up to the mistake in the line beginning with PRINT in the original instruction and use either CTRL and DELETE/BACK S or DELETE/BACK S alone (depending on where we put the cursor). Then press RETURN and the Atari will print

```
I don't spell very well.
```

Keep in mind that you don't need to move the cursor to the end of the line after making a correction or addition in order to have the whole line accepted by the computer. This is true even if the line you are editing fills more than one line on the screen.

The reason we took care to delete the printed line from the screen before we edited our mistake was that otherwise the computer would have printed our corrected line directly over the incorrect line. In this case, you would have seen the same mistake appear, even though it would not be your fault. Go back and run through the previous example without removing the incorrectly printed line and you will see what we mean.

A similar error in a program line would be corrected the same way, except there would be no need to delete the incorrectly printed line, since RETURN would not cause the line to be executed in this case. Remember that this is because numbered lines are interpreted as part of a program and will not be executed until a RUN command has been given.

This is a good opportunity to discuss line length. The Atari will normally display 38 characters per line. A program line, however, can include three "screen" lines. Type in the following line (don't press RETURN until the very end):

```
PRINT "This line of printing will have
more than one screen line in it,  in f
act, it will have three screen lines."
```

Did you notice the "beep" when you got to the letter "l" near the end? The Atari was warning you that you had just seven characters to go before you reached the end of the line length that it would accept. Also notice that when the line was printed on the screen, the word "than" was split up in an inconvenient place. The Atari will carry out your PRINT statement by placing the first 38 characters (including spaces) on line one, the next 38 on line two, and the rest on line three. A useful trick to keep words from being broken up in such unsightly places is to enter the line as follows:

```
PRINT "This line of printing will have
more    than one screen line on it,   in
fact itwill have three screen lines."
```

Writing the line in this manner matches the line length to be printed to the width of the display available. What we have done is add or remove spaces between words to "force" words in the second and third lines to line up with T in the first line (T is the first character in the PRINT statement). You will find this to be helpful in getting your screen displays to line up neatly and be more readable.

The ⅄ key on the 800 and 400 and the ◢ key on the XL series switches the keyboard between normal and inverse characters. Screen displays in inverse characters can be effectively used for emphasis. Remember that if you accidentally type a BASIC command in inverse video, the Atari will give you an error message, even if you used uppercase.

1-3...Saving and Loading Programs

Before we go any further, we need to cover how to go about saving and loading programs. Without these capabilities, your Atari would be very limited. Who among you would like to have to retype a program every time you wanted to use it? No hands up? We don't blame you.

...Tape

Tape users can save programs using CSAVE or SAVE "C:" or LIST "C:", where the "C:" refers to the Atari cassette recorder. CSAVE and SAVE "C:" perform identically, while LIST "C:" saves programs in a different format and also permits the saving of portions of a program by using an alternate form of the command— LIST "C:",A,B, where A is the beginning line number and B is the ending line number of the program section that you wish to save. In all of these cases, there will be two beeps from the computer to notify you that you should position the tape in the recorder, press the RECORD and PLAY switches, and then press RETURN to start the saving process. One important note for cassette users. It is *highly* recommended that you issue an LPRINT command before you save any program to tape. You will receive an error message when you do this (unless you have a printer and it is turned on), but this doesn't matter. What you accomplish is that the program is more reliably saved.

The most convenient way to SAVE programs on tape is to rewind the tape to the beginning, reset the counter on the recorder, and then press RECORD and PLAY together. If you use long tapes, note the counter reading after the program is saved and use this number as the starting point for the next program. Then, when you want to load a program, you can rewind the tape, reset the counter, and use fast forward to move to the known starting number for the program of interest. This can be a nuisance for a tape that has many programs, but it works. Better yet is to use short tapes (called C-10 for ten minutes per tape, five on a side). Then it's easy to find a program by rewinding and saving only one program per tape side. Remember to reset the counter whenever you rewind the tape and *only* then.

If you used CSAVE or SAVE"C:" to save your program then you load it with CLOAD or LOAD "C:". If you used SAVE "C:" you can also have the program load and run by using RUN "C:". If you used LIST "C:" to save, you use ENTER "C:" to load the program. In all of these cases, there will be one beep from the computer to notify you that you should position the tape, press the PLAY key on the recorder and then press RETURN to load the program.

The computer has no way of checking if you pressed the correct buttons for saving or loading. You will, however, receive an error message if the process fails.

...Disk

If you are using a disk drive, your first order of business is to "format" the disk—that is, to prepare the surface of the disk so that the computer knows where it is writing information. To do this follow the sequence:

Type DOS and press RETURN.

When the DOS menu appears, type I and press RETURN.

Type 1 to indicate you want to format a disk in drive 1.

Type Y and press RETURN to verify that you really do want to format a disk.

When the menu reappears, the disk is formatted.

Type H and press RETURN to indicate you want to write DOS on this disk.

Type Y and press RETURN to verify your choice.

When the BUSY light on the drive goes out, the disk is ready for your use.

Type B to get you back into BASIC.

Note that when you type DOS you will be erasing any program currently in the computer's memory, so you should format disks before you type in or load any programs that you will want to save. It is generally a good practice to save any programs before going to DOS to carry out any of its many functions. We will discuss DOS in detail in Chapter 9.

Disk drive users save programs by using SAVE "DN:FILENAME.EXT" or LIST "DN:FILENAME.EXT", where N is the disk drive number (1–4) and FILENAME.EXT is the name you have assigned to that program. If N is not given, the computer assumes that it is 1, so that those with one disk drive need not specify this number.

A portion of a program is saved to disk with LIST "DN:FILENAME.EXT",A,B where A is the starting line number and B is the ending line number of the section to be saved. FILENAME.EXT consists of a program name of up to eight letters and an optional three-character extension after the decimal point. Lowercase letters and spaces and other symbols are not permitted. FILENAME must begin with a capital letter and should contain only capital letters and numbers. If you have saved a program, you can load it with LOAD "DN:FILENAME.EXT" or you may have it load and begin to run by itself with RUN "DN:FILENAME.EXT". The meaning of N and FILENAME.EXT are the same as above. If you have LISTed a program, you load it with ENTER "DN:FILENAME.EXT".

Note that all the commands with LOAD or RUN in them erase anything in the computer's memory before executing. On the other hand, however, using ENTER leaves memory intact and loads right over an existing program, line for line. Thus, if the program in memory had a line 100 and the program being ENTERed also has a line 100, the new program line will replace the old one. This can be useful for merging portions of programs together.

If you want to enter a program without overlaying an existing program in memory, type NEW and press RETURN before using the ENTER command.

This is only a brief description of how to save and load programs. We wanted to put information on how to save and load programs very early in this book because without it, using a computer can quickly become a very frustrating experience. This is precisely what we are trying to avoid. There will be more details and information on saving data from a program later.

1-4...Performing Calculations in Atari BASIC

Now that we know how to display messages, we might like to have something for the messages to talk about. The Atari's ability to perform calculations is readily available to us.

```
100 PRINT "THE NUMBERS ARE:"
105 PRINT "234.56 AND 43901"
110 PRINT "THE SUM IS"
120 PRINT 234.56+43901
130 PRINT "THE DIFFERENCE IS"
140 PRINT 234.56-43901
150 PRINT "MULTIPLY THEM"
160 PRINT 234.56*43901
170 PRINT "NOW DIVIDE"
180 PRINT 234.56/43901
```

Program 1-2. Calculations in Atari BASIC.

Here we have a program that performs addition, subtraction, multiplication, and division of two numbers. As you can see in lines 120, 140, 160, and 180, BASIC uses +, −, *, and / as the symbols for these arithmetic operations. Below is an execution of this program.

```
RUN
THE NUMBERS ARE:
234.56 AND 43901
THE SUM IS
44135.56
THE DIFFERENCE IS
-43666.44
MULTIPLY THEM
10297418.5
NOW DIVIDE
5.34293068E-03

READY
```

Figure 1-5. Execution of Program 1-2.

Notice that the product displays nine or ten digits, depending on your model. This is the maximum precision of Atari BASIC. That is not to say that all answers are accurate to this many digits. Sometimes the computer has to round things off. So, it is up to you to verify the accuracy of computed results. In addition to evaluating the accuracy of the computations that the computer carries out, you will need to know the accuracy of the numbers that you give the computer to work with.

Never enter numbers with commas. The comma is used in a different way in Atari BASIC, to separate different numbers, rather than to set off digits within a given number.

For the division problem of our program, something else interesting happens.

We get 5.34293068E-03 as the answer. This is another way of writing .00534293068, which takes 11 digits to express. Atari BASIC uses this scientific notation for displaying very small and very large numbers. Any value less than 0.001 or greater than 9999999999 will be displayed in this manner. Atari BASIC limits numbers to a range from −9.99999999E97 to 9.99999999E97. That should be entirely adequate for our needs for some time to come.

The following is a little program to demonstrate a few numbers printed in scientific notation.

```
100 PRINT "EXAMPLES OF SCIENTIFIC NOTA
TION"
120 PRINT ".0001",";"= ";.0001
125 PRINT ".00058293","= ";.00058293
130 PRINT ".00123456789","= ";.0001234
56789
140 PRINT "11234567890","= ";112345678
90
150 PRINT "3939382827347456","= ";3939
382827347456
```

Program 1-3. Demonstrate scientific notation.

In Program 1-3 we have used single PRINT statements to print two items on one line. Look at line 125. There you will see that the first thing to be printed is enclosed in quotes. This instructs the computer to print .00058293 (see Program 1-3). Then a comma appears. A comma as used here is an instruction to the computer to display the next character beginning in the next field. In Atari BASIC the first field starts with the first column on the screen and successive fields come every ten columns after that. Our left margin is in column 2. The next field comes in column 12, then 22, and so forth. The program then first looks to see where the cursor is located after printing .00058293 (in column 13) and moves over to the next field (column 22), where it prints " = ". Next it sees the semicolon. A semicolon is used to separate items in a PRINT statement when we want the computer to keep printing without skipping any columns on the screen. Therefore, immediately after printing the equals sign and one space it will print its representation of the number .00058293 in scientific notation. We have used commas and semicolons to separate different items in a program. Symbols used in this way are called *delimiters*.

Now let's look at the display produced by the above program.

```
RUN
EXAMPLES OF SCIENTIFIC NOTATION
.0001       = 1E-04
.00058293           = 5.8293E-04
.00123456789        = 1.23456789E-03
11234567890         = 1.12345678E+10
3939382827347456    = 3.93938282E+15

READY
```

Figure 1-6. Execution of Program 1-3.

Notice the second line printed. Even though line 120 in our program looks like all the others, the result doesn't line up neatly on the screen like all the others. This is because the cursor found itself in column 8 after printing .0001 and, therefore, the comma instruction only sent it to column 12 (the next field). Change line 120 to read as follows and then rerun the program; you will see a much neater display.

```
120 PRINT ".0001",,"= ";.0001
```

The two commas instruct the computer to move over two fields before printing the equals sign. No matter how experienced you become with your Atari, there will always be times when screen output will have to be readjusted to make it more readable.

We will indeed get used to the 38-character screen on the Atari. However, we are not limited to 38 characters on the printed page. Therefore, we will present most of our program listings in a wider format. If you type exactly what is displayed in the programs of this book, BASIC will take care of the rest. We present here Program 1-3 reformatted without any line breaks.

```
100 PRINT "EXAMPLES OF SCIENTIFIC NOTATION"
120 PRINT ".0001",,"= ";1E-04
125 PRINT ".00058293","= ";5.8293E-04
130 PRINT ".00123456789","= ";1.23456789E-03
140 PRINT "11234567890","= ";1.12345678E+10
150 PRINT "3939382827347456","=";3.93938282E+15
```

Program 1-4. Demonstrate Program 1-3 without line breaks.

We may want to work with a screen that has some value other than 38 characters per line. We can easily adjust both the left and the right margins. Memory locations 82 and 83 hold the value of the left and right margins (normally 2 and 39, respectively). The following instructions will change the left margin to the 0th and the right to the 10th column.

```
POKE 82,0
POKE 83,10
```

The format of a POKE statement is to give the memory location, then a comma, then the value to be put there. We can do this in either the immediate or the deferred modes of operation. Using POKE allows us to put any value from 0 to 255 in any memory location, although in most cases only a few values are meaningful and some values will cause problems. This then becomes the value the computer uses until we again change it. (Some operations of the computer will change the values in certain memory locations, but that is not of concern at this time.) The opposite of POKE is PEEK, which allows us to find out what value is currently stored in any memory location without changing it. Throughout this book, we will introduce other handy POKEs and some PEEKs. Appendix A contains a summary of many of the more useful ones.

1-5...Numeric Variables

We can do some interesting things with what we know at this point, but the real power of the computer begins to emerge when we can save the results of calculations without having to recalculate.

A variable may be thought of as a pigeonhole or a mailbox in which we may save the value of an intermediate result as a computer program goes about solving our problem for us. For example, suppose we establish an hourly wage and save it in "WAGE". Then we take the number of hours worked and save the value in "HOURS". Next, we might find the net pay by multiplying WAGE by HOURS and then saving it in "PAY".

Or we might want to take the average of some numbers. Program 1-5 uses numeric variables to do just that.

```
90 REM * CALCULATE A SIMPLE AVERAGE
100 LET SUM=34+45+65+89+91+56
110 LET NUMBEROFVALUES=6
120 LET AVERAGE=SUM/NUMBEROFVALUES
130 PRINT "AVERAGE =";AVERAGE
140 END
```

Program 1-5. Calculate a simple average.

If we want to calculate an average for a different set of values, we need only retype lines 100 and 110 of this simple program.

We have used three variables in this program: SUM, NUMBEROFVALUES, and AVERAGE. We are free to choose a wide variety of names for variables. In Atari BASIC, variable names must start with an uppercase letter, which may be followed by any uppercase letter or any digit or combination. The length of a variable name may be as long as will fit on one program line if you choose. Unlike many other versions of BASIC, Atari BASIC recognizes all the characters in a variable name. Thus, NUMBEROFVALUES1 and NUMBEROFVALUES2 will be stored in two distinct pigeonholes in the computer's memory. It is an excellent idea to give meaningful names to variables so that the program has meaning to others and also to you if you want to modify it at some time in the future. WAGES = 10000 will have much more meaning than W = 10000. Very long names (like NUMBEROFVALUES), however, are a bother to type in if they appear several times in a program. Also, longer variable names take up more memory for their storage, meaning less memory is available for your program. Atari BASIC allows up to 128 variables in any program, so you will rarely be at a loss for variables. You should avoid using BASIC keywords, such as LIST, NEW, PRINT, and END as variables in your program. You can usually get away with this on an Atari, but it is good programming practice and common sense to avoid it. Stick to all the other words in the dictionary for your variables.

You should also be on the lookout for variable names that start with keywords. Note the following line:

```
100 LISTER=A
```

You can type this line in and hit RETURN and not receive an error message. However, if you list this line you will receive

```
100 LIST ER=A
```

because LIST is a keyword and BASIC is trying to help you out by putting a space between the keyword and the rest of the program line.

Each of the statements 100, 110, and 120 of Program 1-5 is an example of an *assignment statement* in BASIC. The effect of statement 100 is that the Atari will calculate the sum of the six numbers shown there and store it in a slot labeled "SUM". Line 110 causes the value 6 to be stored in a pigeonhole labeled "NUMBEROFVALUES". Line 120 causes the computer to divide the value found in the slot labeled "SUM" by the value found in the slot labeled "NUMBEROFVALUES" and place the result in a slot labeled "AVERAGE".

```
RUN
AVERAGE =63.3333333

READY
```

Figure 1-7. Execution of Program 1-5.

The assignment statement in BASIC may take one of two forms.

```
100 LET X=15
```

and

```
100 X=15
```

are functionally equivalent. In practice, most programmers drop the use of LET. However, many beginners find it helpful to include the LET keyword while learning BASIC.

1-6...The INPUT Statement

BASIC includes the INPUT statement as one means of providing data for a program to work on. When BASIC encounters an INPUT statement, it causes the computer to wait for data to be typed at the keyboard.

When the statement

```
200 INPUT X
```

executes, it will display a question mark as the signal to us that we are to type in a single number.

```
200 INPUT X,Y,Z
```

will also display a question mark. However, we have provided for three values to be entered, and the computer will insist on getting all three.

Suppose we enter only one. BASIC will gently prod us by displaying a question mark repeatedly until we have entered the proper amount of data.

Suppose we enter too much data. Then Atari BASIC will ignore the extra data and proceed with the rest of the program.

Suppose we just hit the RETURN key or type a letter instead of a number. BASIC will display the message

ERROR— 8 AT LINE 200

which is an INPUT STATEMENT ERROR. We have asked for input as a numeric variable and RETURN or letters are not acceptable. In the next chapter we will learn how to protect a program against all sorts of input errors.

Appendix C contains the English equivalent of the most common Atari error codes. As time goes by and your familiarity with Atari BASIC grows, you will become proficient at interpreting these error codes. Atari BASIC is a very powerful BASIC considering its size. However, in order to incorporate all the nice features, such as its fabulous editing capabilities, it was necessary to skimp in other areas, and that is why the error codes are numbers rather than words.

Suppose we have the following record of gasoline purchases for a brand-new car:

GALLONS	ODOMETER
19.3	230.3
12.7	456.7
17.7	709.4
11.1	895.5
13.8	1131.6

We want a program that will calculate the mileage for each tankful of gasoline. We have been careful to fill the tank each time. Since we do not know whether the tank was full when we got the car, we should discard the figure for the first purchase. What we do know is that 12.7 gallons took us 226.4 miles (456.7 − 230.3), 17.7 gallons took us 252.7 miles (709.4 − 456.7), and so on. Program 1-6 asks the right questions and performs the miles-per-gallon calculation for us.

```
100  PRINT "FIRST READING";
110  INPUT MILEAGE1
195  PRINT
200  PRINT "GALS, READING";
210  INPUT GALLONS,MILEAGE2
220  MILEAGE=MILEAGE2-MILEAGE1
230  MILESPERGALLON=MILEAGE/GALLONS
240  PRINT MILESPERGALLON;" MPG"
250  MILEAGE1=MILEAGE2
260  GOTO 195
```

Program 1-6. Calculate gasoline mileage.

Note the use of the semicolon at the end of lines 100 and 200. This enables us to compose a single line of display on the screen from several lines in a program. The semicolon instructs the computer *not* to move the cursor down one line and to the left margin. Thus the question mark displayed by the INPUT statement at line 110

will appear immediately following the G in READING from line 100, and the question mark displayed by the INPUT statement at line 210 will appear immediately following the G in READING from line 200.

It is always a good idea to display a label for an INPUT request. By this we mean you should have the program print a message stating what the INPUT wants (such as MILEAGE, GALLONS, etc.). We may know right now what that question mark means, but nobody else will know, and next week we probably won't remember.

Since we want the number of miles traveled, the program must subtract the previous reading from the current one. This is done in line 220. MILEAGE1 is the miles traveled, GALLONS is the number of gallons used and MPG is the number of miles per gallon. Once the computer has calculated the number of miles traveled, the current reading becomes the previous reading for the next item of data to be entered. This is the purpose of line 250. Line 195 is referred to as a blank PRINT. A blank PRINT will be displayed as a blank line. We can use this to adjust the spacing for better legibility.

...GOTO

We must introduce the GOTO statement at this point. The GOTO 195 you see at line 260 is an instruction to the computer to execute the statement numbered 195 next in sequence. In this way, we are able to control the order in which BASIC executes the statements of a program.

```
RUN
FIRST READING?230.3

GALS, READING?12.7,456.7
17.82677165 MPG

GALS, READING?17.7,709.4
14.27683615 MPG

GALS, READING?11.1,895.5
16.76576576 MPG

GALS, READING?13.8,1131.6
17.10869565 MPG

GALS, READING?
STOPPED   AT LINE 210
```

Figure 1-8. Execution of Program 1-6.

Clearly the value 17.82677165 shown in the execution of Program 1-6 is more precise than 12.7, 456.7, or 230.3, but it is not more accurate. We may safely say that we got about 17.8 miles per gallon for the first calculation. The results of a calculation can never be more accurate than the data. Soon we will learn how to round off results to any desired precision.

What did we do to deserve the message

```
STOPPED   AT LINE 210
```

We can halt execution of a program by pressing the BREAK key in response to an INPUT request. This is okay for programmers, but soon we will see better ways to exit our programs. We should not require others who will be using our program to use the BREAK key in this way. In fact, we will learn how to disable the BREAK key so that inadvertently pressing it will not mess up the program. We will also learn several good ways to terminate input.

1-7...READ and DATA in Atari BASIC

There are numerous ways to provide programs with data. We have entered numbers into our programs by including them in PRINT statements, by assigning values to variables with the assignment statement, and by programming with the INPUT statement. Now we add READ and DATA to the list for Atari BASIC.

The READ statement assigns values to variables by using a DATA statement as the source. If there is more than one item of data in a DATA statement the items are separated from one another with a comma. BASIC will always interpret a comma as the end of that particular piece of data.

READ and DATA are always coordinated to solve the problem at hand. It is not sensible to have one without the other. Let's simply convert the INPUT-based program on gasoline mileage to an equivalent program using READ and DATA. In this case the data will not be entered from the keyboard during execution, so we display the gallons and miles traveled along with the miles-per-gallon figure. The logic here is identical to the logic of our Program 1-6.

```
90 PRINT "CALCULATION OF CAR MILES PER GALLON"
95 PRINT
100 READ MILEAGE1
200 READ GALLONS,MILEAGE2
220 MILEAGE=MILEAGE2-MILEAGE1
230 MILESPERGALLON=MILEAGE/GALLONS
240 PRINT MILESPERGALLON;" MPG"
250 MILEAGE1=MILEAGE2
260 GOTO 200
900 DATA 230.3
910 DATA 12.7,456.7
920 DATA 17.7,709.4
930 DATA 11.1,895.5
940 DATA 13.8,1131.6
```

Program 1-7. Program 1-6 with READ and DATA.

Note that we have arranged the DATA so that the numbers are grouped to look like the table of values first presented. The computer doesn't care how many or how few lines we use for DATA. The important point is that the values in DATA statements be in the correct order. We may arrange the DATA so that it is well

organized for humans to read. Since the computer will never be confused, we should take care to make things better for us.

```
RUN
CALCULATION OF CAR MILES PER GALLON

17.8267716 MPG
14.2768361 MPG
16.7657657 MPG
17.1086956 MPG

ERROR-    6 AT LINE 200
```

Figure 1-9. Execution of Program 1-7.

We do indeed get the expected calculation results. However, we have also triggered an Atari error message because we ran out of DATA when the program was expecting more. We will find a couple of ways to handle the end-of-data condition in our programs as we progress.

...SUMMARY

We have covered a lot of ground in this first chapter. Very soon, nearly everything presented here will be second nature to you. Many of the error numbers will be quickly translated by your brain into their English equivalent; the others can easily be looked up in Appendix C. You do need to remember things like NEW, LIST, RUN, PRINT, LET, line numbers, GOTO, INPUT, variables, quotes, and READ and DATA.

Atari BASIC allows numbers from $-1E97$ to $+1E97$.

Practice with the various editing commands will pay off in handsome dividends in future ease of entering and editing programs. There is always the hard way to edit—by retyping the entire line—but the easy cursor control movement and insert and delete capabilities of Atari BASIC will make the job much easier.

The PRINT statement in BASIC is used to display labels in quotes, numeric values expressed literally, and values stored in variables individually or in combination by separating items with semicolons or commas.

Variables are used in programs to retain numeric values during program execution. Variable names must begin with a capital letter and may consist of capital letters and digits intermixed after the first character. Variable names may be as long as three screen lines and certainly should be long enough to make your program easily understandable to yourself and others. It is best to avoid using any BASIC keywords as variable names.

Values may be assigned to variables using the BASIC assignment statement. The use of LET in such statements is optional. Values may also be assigned through the keyboard using INPUT statements.

The companion statements READ and DATA are used for storing data within the program itself.

We have four commands for program manipulation thus far. RUN calls for our program to be executed. GOTO 100 calls for our program to be executed beginning at line 100; under some circumstances, it can lead to erroneous results. LIST displays our program or a segment of our program. NEW clears the BASIC work area for new projects.

We can halt a program waiting for INPUT or at any point during its execution by pressing the BREAK key.

Programs can be saved to and loaded from tapes using CSAVE or SAVE"C:" and CLOAD or LOAD"C:" or RUN"C:". Portions of programs are saved and loaded with LIST"C:" and ENTER"C:". The corresponding disk commands are SAVE "D:FILENAME.EXT" with RUN "D:FILENAME.EXT" and LIST "D: FILENAME.EXT" with ENTER "D:FILENAME.EXT".

Problems for Chapter 1 ....

You should not feel that you must limit yourself to the problems offered here. As you get some programming under your belt, you should find lots of interesting problems to try on the computer. Learning to program is unique in that the computer will provide you with a measure of your success. You do not need an answer book or a teacher for that. The real joy of learning anything comes when you begin to formulate the problems, solve them, and verify that your solutions are correct all on your own (it helps to have a computer).

At this point you can write programs to print messages of all kinds, request data from the keyboard, and perform a variety of arithmetic operations.

1. Write a program to display the sum of 123.45, 654, 1920, 114423, and .01.
2. Write a program to display the sum of five numbers to be supplied during execution.
3. Write a program to print a decimal value for 2/3.
4. Assign 1/3 to the variable X. Display the value of X, 3*X, and X + X + X.
5. Write a program to display decimal values for 1/7, 2/7, through 6/7.
6. Write a program to find a value for the following expression:

$$\frac{1/2 + 1/3}{1/3 - 1/4}$$

7. Write a program to find the sum of the first ten counting numbers.
8. Write a program to find the product of the first ten counting numbers.
9. Write a program to print out the message "HAPPY BIRTHDAY" five times, alternating between uppercase and lowercase letters and with one blank line between each printed line.
10. Write a program to print the message "HAPPY BIRTHDAY" diagonally beginning in the upper left of the screen and putting each letter on a different line.

PROGRAMMER'S CORNER 1

...RESTORE

When your program requests input via a READ command, the computer will scan
the program from the beginning to find the first DATA statement and start there.
The next READ statement will cause data to be read starting from where the last
READ statement ended, and so on through the program. There are times when you
will want to read in the same data several times. One way to do this, of course, is to
have another set of DATA statements that repeat the data. A much easier way is
to use RESTORE in the form

```
100 RESTORE L
```

where L is the program line number where you want the next READ command to
start. If no line number is specified, the next READ statement will begin with the
first DATA statement. Note that the next READ statement will carry on from here
and *not* go back to where the data "pointer" was before our RESTORE.

...LIST

We have used the LIST command to display a program that is stored in the
computer's memory. In a long program, we may be looking for a particular section
in order to make some modifications. This is very difficult with the LISTed
program scrolling up the screen and lines disappearing off the top of the display. If
you hold down CTRL and press "1" (one) the LISTing will halt. Doing this again
will continue the LISTing. Thus, you can stop and start the display until you find the
program lines you are looking for. Once you find them, you can then press the
BREAK key to leave you in immediate mode where you stopped the listing so you
can edit those lines that appear on the screen, or let the listing continue to the end
and then list only those lines of interest. If you happen to know the general area in
the program that you want to have displayed on the TV, you can still use a
statement like

```
LIST 150,250
```

which will list line numbers 150 through 250, inclusive.

Another of the keys to be aware of is the SYSTEM RESET key. The reason for
this is that inadvertently pressing this key will do things that you don't want to have
happen in the middle of your program execution. The computer will clear the
screen, initialize itself, and go into the immediate mode, but it will *not* erase your
program from memory. Trying to continue a program from some middle point
after a SYSTEM RESET will generally be unsuccessful.

...CONT and STOP

Whenever a program is halted before its conclusion, it is possible to continue by
using the command CONT. This will continue execution starting with the

beginning of the next line after the line at which the program stopped. It is possible that part of this last line was not executed, which can lead to problems if, for example, the program was expecting two pieces of INPUT, but was halted after only getting one. The missing value will be assumed to be zero and may be used in subsequent calculations. So, be wary of this type of problem.

It is good practice to run through a program that does calculations using simple numbers to be sure that the program does what you want it to do, rather than what you have told it to do. This process can be made easier by inserting the STOP statement at various points in the program. Upon encountering STOP the program will do just that: it will finish everything up to the STOP and then go into the immediate mode. You can then PRINT the value of any variable used by the program in order to check out the calculations up to that point. You can do this in the immediate mode with simple PRINT statements. For example, PRINT X,DAY, 3*MILES will result in the computer printing on the screen the current values of the variables X and DAY and 3 times the variable MILES. You may then use CONT to go on to the next STOP. This is one procedure for "debugging" your programs— that is, for ensuring that there are no nasty "bugs" contained in the program that will lead to errors during program execution.

...More on Numbers and Calculations

One of the arithmetic operations not yet covered is exponentiation, the process of raising a number to a power. For example

```
2^3 =2*2*2 = 8
```

The "∧" sign (SHIFT and "*" key together) is used for exponentiation. Go ahead and type PRINT 2^3 in the immediate mode. Surprised, weren't you? Some versions of Atari BASIC give back 7.99999999 rather than 8! This is because the computer does not actually multiply 2 times itself three times when you ask for exponentiation. Some other calculations that, by the rules of common math, should give nice even results will similarly give close, but not exact, approximations. Generally, when we are doing a complex mathematical calculation, this type of problem will not affect us, because the final result will not show up this very small difference. We can easily, however, make up a simple example of how this difference will matter.

```
10 LET X=X^3
20 LET Y=2*2*2
30 IF X<>Y THEN PRINT "CURSES, FOILED AGAIN"
40 END
```

The "<>" in line 30 is just the way of saying "not equal," so this line says: IF X does not equal Y THEN PRINT 'CURSES, FOILED AGAIN!'

From the above discussion, we know that this message will indeed be printed with some Ataris. Other, more serious programs may also be tripped up by this type of calculation that exists in Atari BASIC. For those who have this problem, we will see a way around it in the next chapter.

...Precedence of Operations

How does BASIC evaluate the following expression:

```
4 * 3 + 2 ^ 3 - 1/4
```

Without some guidelines, it is possible to get many different results from this calculation, all depending on where we start and in what order we perform the calculations. Atari BASIC has a built-in order in which it will perform all calculations. We currently have at our disposal five mathematical operations— addition, subtraction, multiplication, division, and exponentiation. From among these, BASIC will always do exponentiation first, then multiplication and division, then addition and subtraction. Given a choice between operations of equal precedence, such as multiplication and division, it will start on the left and work to the right. Looking at our example above, 2^3 will first be calculated, making the expression to be evaluated

```
4 * 3 + 8 - 1/4
```

Multiplication and division are next in priority, so starting on the left, it will evaluate 4 * 3, leaving

```
12 + 8 - 1/4
```

The division follows, giving

```
12 + 8 - .25
```

Finally, the addition and subtraction are carried out to finish the calculation.

We can control the order of calculations by using parentheses around expressions. This is our way of telling the computer to ignore its normal hierarchy of operations and do things the way we want them. Calculations within parentheses will be carried out before any other operations are performed on them. Within a particular set of parentheses the normal order of operations will be followed. A good rule of thumb is that, if you're not sure how the computer is going to carry out a certain calculation, use parentheses to direct the order of operations. And always try out calculations with known numbers to see if things look right.

...Atari Graphics Characters

You have probably stumbled across the strange looking set of characters you find if you hold down the CTRL key and press any of the letter keys on the keyboard (or the period, semicolon, or comma). These shapes can be used as they are or combined to make figures or pictures for your programs. They can be put in PRINT statements just like text and displayed on the screen. For example, if you want a small box, type

```
100 PRINT "CTRL+Q CTRL+E"

110 PRINT "CTRL+Z CTRL+C"
```

The " + " means you should hold down the CTRL key while you press the letter key. What you will see between the quotation marks will be the graphics symbols themselves rather than the letters shown above. When you RUN this program you will get the small box desired. To get a bigger box you can use CTRL plus R or N to extend the top and bottom or SHIFT and " = " to extend the sides. You can even create a sort of graph paper with the following program.

```
90  REM * USING THE GRAPHICS SYMBOLS.
91  REM *
92  REM * IN ALL THE FOLLOWING LINES
93  REM * C+ MEANS HOLD DOWN CTRL KEY
94  REM * AND PRESS NEXT LETTER KEY.
95  REM * S+ MEANS HOLD DOWN SHIFT KEY
96  REM * AND PRESS NEXT LETTER KEY.
97  REM * ALL GRAPHICS SYMBOLS GO
98  REM * BETWEEN QUOTATION MARKS.
100 PRINT
110 PRINT
115 REM * IN LINE 120: C+Q (C+R C+W REPEATED
11 TIMES) C+R C+E
120 PRINT "┌┬┬┬┬┬┬┬┬┬┬┐"
125 REM * IN LINES 130,150,170,190: (S+=
REPEATED 12 TIMES) S+=
130 PRINT "│ │ │ │ │ │ │ │ │ │ │ │"
135 REM * IN LINES 140,160,180: C+A (C+R C+S
REPEATED 11 TIMES)
136 REM * THEN C+R C+D
140 PRINT "├┼┼┼┼┼┼┼┼┼┼┤"
150 PRINT "│ │ │ │ │ │ │ │ │ │ │ │"
160 PRINT "├┼┼┼┼┼┼┼┼┼┼┤"
170 PRINT "│ │ │ │ │ │ │ │ │ │ │ │"
180 PRINT "├┼┼┼┼┼┼┼┼┼┼┤"
190 PRINT "│ │ │ │ │ │ │ │ │ │ │ │"
195 REM * IN·LINE 200: C+Z (C+R C+X REPEATED
11 TIMES) C+R C+C
200 PRINT "└┴┴┴┴┴┴┴┴┴┴┘"
```

Program 1-8. Using Atari graphics.

If you typed in the above program, you may be thinking that there must be an easier way. Take heart—there is, once we learn commands that allow us to do repetitive things with one instruction. You'll then be able to make boxes of almost any size as well as perform other useful tasks.

If you want to just play with the graphics symbols on the screen you can press CTRL plus CAPS/LOWR to lock the keyboard into the graphics symbol mode. Now all the keys will display their graphic equivalents without your having to press CTRL first. To get out of this mode, press SYSTEM RESET or SHIFT plus CAPS/LOWR.

Chapter 2

Writing a Program

2-1...Planning Your Program

Computer programs are linear. That is, they define a single step at a time. Many problems brought to the computer for solution are nonlinear in nature. At various places in the program we may have to do different things, depending on the input or the result of a particular calculation. Many computerized processes are outlined by using large charts in which each item represents a complete subsystem consisting of a whole collection of very long computer programs. We need to develop some concepts that will help us begin with the big ideas and systematically arrive at a completed project whose smallest elements are computer-program statements.

Good programming requires a plan. The planning should be completed before' any program statements are written down. You should write out the entire program on paper before you sit down to type it into the computer. Major changes in program organization are easy to deal with before the program has been typed into the computer, because there is less inertia to overcome. Once a program has been typed into the computer, part of the problem becomes how to make the desired change while preserving as much as possible of what exists. While the program is still written out longhand, such changes are much easier. Certainly, we can easily write a program to add two numbers without much fuss. The plan can be in our head and we can "write down" the program statements directly at the computer keyboard. But try writing a system to launch a satellite or a system to do payroll—or even a program to find all prime integers from 1000 to 2000. Good planning requires a complete understanding of the problem. What is known? What is the question? What will be the form of the solution? How do I get from the known information to the solution?

...Counting on the Computer

Let's start with something simple—developing a plan for getting the computer to count. This is a good first problem, since it is something we are familiar with. A thorough understanding of the problem at hand is essential for writing computer programs. It is highly unlikely that we can write a program to solve a problem we do not understand.

We usually count by starting with the number 1 and repeatedly adding 1 to get the next counting number in sequence. That is easy! There are only two ingredients here: beginning and adding 1 repeatedly.

We can begin with a statement such as

```
110 COUNTER1=1
```

But how do we add 1? Here is one way:

```
120 COUNTER2=COUNTER1+1
125 COUNTER1=COUNTER2
```

In mathematics, the equals sign (=) usually asserts that two expressions have the same value. However, assignment statements in BASIC use the equals sign for a special purpose. The equals sign is used to assign the result of a calculation on the right to a variable named on the left. This allows us to eliminate line 125 and combine it with line 120

```
120 COUNTER1=COUNTER1+1
```

The variable COUNTER1 contains one value before the execution of this statement and another value following execution of this statement. The prior value is replaced by the new value. A variable may not store two values simultaneously. Even though pigeonholes and mailboxes may hold more than one item, variables cannot.

Now we need to tell the computer to repeat the work of line 120 over and over again. We resist any possible urge to include a statement 130 COUNTER1 = COUNTER1 + 1, and so on. To count to 100 this way would require more than 100 statements. Computers are supposed to save work, not make things harder. The way to repeat the action of line 120 over and over again is to include the following line:

```
130 GOTO 120
```

This will put the computer into a *loop*. Now we have three lines that would indeed cause the Atari to count, beginning with 1. We will use the variable COUNTER, not COUNTER1, since we saw that a variable can be incremented in one program statement.

```
110 COUNTER=1
120 COUNTER=COUNTER+1
130 GOTO 120
```

Program 2-1. First counting program.

However, we have overlooked an important ingredient. We will never know which number the computer is up to at any particular time, except when it gets to 9.99999999E97—that is, when it reaches the limit on numbers that the Atari can handle. What we need here is to display each number as the computer gets to it. Therefore, we will insert a PRINT statement between lines 110 and 120. In order for this statement to be executed every time the computer adds 1, the GOTO statement at line 130 must be changed. The loop must include the PRINT statement as shown in Program 2-2.

```
110 COUNTER=1
115 PRINT COUNTER
120 COUNTER=COUNTER+1
130 GOTO 115
```

Program 2-2. Counting with display.

Several comments and one warning are in order here. Program 2-2 has no natural termination. If you execute this program, it will run for a very long time. You will have to hit the BREAK or RESET button, pull the plug, or wait for BASIC to overflow. In order to make this a useful counting program, we need to replace the unconditional statement, 130 GOTO 115, with one that can make a decision. This brings us to the IF . . . THEN statement.

...IF ...THEN

Our counting program would be more useful if we had a way for it to terminate when some predetermined number has been reached. BASIC has the ability to alter the order in which statements are executed depending on the outcome of a decision. This is called a *conditional transfer*. Suppose we want the computer to count to 7 and quit. In this case, we want to GOTO 115 on the condition that COUNTER is less than or equal to 7. That is easy in BASIC:

```
130 IF COUNTER<=7 THEN GOTO 115
```

Line 130 will do the job for us. Here "less than or equal to" is symbolized with ($<$ =). The ($<$) symbol represents "less than" and the (=) symbol represents "equal to."

...REM: What's It All About?

While all of our programs are clear to us at the time that we write them, it is difficult to come back to an old program and recall all of the clear thoughts that we had way back when. BASIC offers the REM statement so that we may include REMarks as part of the program. The computer will ignore all REM statements during program execution, but it will list them along with the others in response to the LIST command. Not only will those REM statements remind us about our own old programs, but they will be invaluable to others reading our programs. No program should be considered complete without REM statements. Some programmers

consider REM statements so vital to the program-development process that they write them first. Not a bad idea!

REM statements should describe the action of the program or a segment of a program. REMarks like "LOAD Y WITH 17" actually detract from the readability of a program, whereas "INITIALIZE LOW TEMPERATURE CUTOFF" describes the function of part of a program. Our REM statement should note that the program will count from 1 to 7. And now we have a counting program to type into the Atari and RUN.

```
100 REM * COUNTING FROM 1 TO 7
110 COUNTER=1
115 PRINT COUNTER
120 COUNTER=COUNTER+1
130 IF COUNTER<=7 THEN GOTO 115
```

Program 2-3. Counting from 1 to 7.

When entering BASIC program lines into the computer, you should generally use spaces just as you would in regular noncomputer writing. The LISTing you get back, however, may not look the same as what you typed in. Atari BASIC will remove spaces between mathematical expressions, so that line 120 is LISTed without any spaces at all. It will also insert a space after each line number and all BASIC keywords, even if you left them out when you typed in the line. Your input may usually contain extra spaces without confusing the computer.

```
RUN
1
2
3
4
5
6
7

READY
```

Figure 2-1. Execution of Program 2-3.

When using IF . . . THEN statements, six options are available:

   $<$ less than
   $<=$ less than or equal to
   $=$ equal to
   $<>$ not equal to
   $>$ greater than
   $>=$ greater than or equal to

These symbols are called *relational operators*. Any BASIC expression may appear on either side of a relational operator.

Counting is a process that pervades computer programming. We do it all the

time. How many players? How many problems? Count the number of scores so that we may compute the average for this lab test. Count the number of lab tests so that we may compute the average for this lab run of this test. The examples go on endlessly. We might be interested in counting only the odd numbers. How would we change our counting program above to do that? That is easy—just change line 120 to read:

```
120 COUNTER=COUNTER+2
```

Now don't forget to change the REM statement to reflect the new function of the program:

```
100 REM * ODD INTEGERS FROM 1 TO 7
```

Misleading REM statements are terrible. The extra time it takes to get the REM statements right will pay off in the end. Suppose we have a problem that requires even integers. In this case, line 110 should set the value of COUNTER to start at 2 or whatever we require as the first even integer. Again, note that the REM statement should reflect the function of the program.

Our little program has four important components.

1. We initialize the counting variable.
2. Some action is programmed. In our example, we display the current value of the counter.
3. The counter is incremented.
4. We test the value of the counter to determine whether or not to loop back and repeat the programmed action.

Most counting routines are used for some higher purpose than merely displaying the current value of the counter. Suppose we have a relative who has promised to give us five times our age in dollars on each of our first 21 birthdays. We might like to know the total number of dollars we will have received upon reaching 21. This problem can be solved with the logic of our little counting program.

Here the programmed action consists of adding five times COUNTER for each year. We will use the variable DOLLARS for this. We initialize DOLLARS to zero. We test for 21 rather than 7. When the IF test fails, the program should print the value of DOLLARS with an appropriate label. The following is such a program.

```
50 REM * TOTAL $5 EACH YEAR ON EACH BIRTHDAY
100 DOLLARS=0
110 COUNTER=1
120 DOLLARS=DOLLARS+5*COUNTER
140 COUNTER=COUNTER+1
150 IF COUNTER<=21 THEN GOTO 120
160 PRINT "$";DOLLARS;" AFTER 21 YEARS"
999 END
```

Program 2-4. Birthday dollars.

Look at line 100. It turns out that BASIC automatically sets the values of all variables to zero when we type the RUN command. So, for our little problem, DOLLARS

would be initialized to zero for us. However, it is good programming practice to include an assignment statement anyway. Having that statement in the program makes the meaning of line 120 less mysterious. When programs are LISTed the line break for REM statements sometimes makes the message hard to read. You may make them more readable by using several statements with the breaks where you want them. Or, after you list the program you may find that you can insert spaces so that the breaks come between words. This is something that you will get used to with a little practice.

```
RUN
$1155 AFTER 21 YEARS

READY
```

Figure 2-2. Execution of Program 2-4.

Another counting problem will help to reinforce some of the things we are learning. Suppose you are the inspector in a packaging plant. Quality control requires that for any lot to be accepted, the average weight for five packages selected at random must be at least 180 grams. You want to write a program that asks the right questions and accepts or rejects the lot. For this problem we have the four components listed above. In this case, the programmed action is a little more complex. We print a label for the INPUT request, request INPUT, and add the entered weight to a variable designated for keeping track of the total weight for the five packages. This can be done with the statement

```
235 TOTAL=TOTAL+WEIGHT
```

where TOTAL is the running total weight and WEIGHT is the weight of each package in turn. Before we enter any package weights the value of TOTAL must be 0. When the value of the counter has passed five, we will calculate the average in AVERAGE. If the value of AVERAGE is less than the required 180 then we want a reject message. Otherwise, we want an accept message. Program 2-5 does it all.

```
100 REM * CHECK AVERAGE PACKAGE WEIGHT
105 REM * FOR 180 GRAM MINIMUM
200 TOTAL=0
210 COUNTER=1
220 PRINT "WEIGHT ";COUNTER;" ";
230 INPUT WEIGHT
235 TOTAL=TOTAL+WEIGHT
240 COUNTER=COUNTER+1
250 IF COUNTER<=5 THEN GOTO 220
260 AVERAGE=TOTAL/5
270 IF AVERAGE<180 THEN GOTO 290
275 PRINT "ACCEPT THIS LOT"
280 GOTO 295
290 PRINT "REJECT THIS LOT"
295 END
```

Program 2-5. Package-weight monitor.

We have included a REM statement describing the purpose of the program. Look at line 200. Note again that we have initialized a variable to zero even though we could let BASIC do it for us. Later, if we want to include this routine as part of a more complex program, the value of TOTAL will be reset to 0 every time this program segment is exercised. Failure to include such a statement would cause the value of TOTAL to grow ever larger as more and more lots are sampled. Thus the program would erroneously accept every sample after the first. (Doubtless, the computer would be blamed for this obvious programmer error.)

Note that we have selected COUNTER for counting, TOTAL for totaling, WEIGHT for the package weight, and AVERAGE for the average. Selecting variable names carefully will make the meaning of each program statement clearer. Don't use A9 for weight or TF for counting. They have no relationship to what they represent and will make it difficult to follow the logic if you want to expand or change the program at some future date.

Line 270 determines which message will be displayed according to the average weight. Line 280 assures that we get exactly one message. Let's run the program.

```
RUN
WEIGHT 1  ?182
WEIGHT 2  ?190
WEIGHT 3  ?180
WEIGHT 4  ?179
WEIGHT 5  ?177
ACCEPT THIS LOT

READY
```

Figure 2-3. Execution of Program 2-5.

To check another lot, simply run the program again.

If this is all that the program does, it might be more practical to use a hand-held calculator. In practice, there are many more factors to consider in the above problem. While it may be illegal for packages to be underweight, it is unprofitable to sell overweight packages. So, in addition to checking for minimum average, our program ought to check for any package over a certain weight, say 185. Furthermore, there may be a legal minimum weight, say 178.

The program can also easily be modified to process several batches of data. Simply change

```
280 GOTO 295
```

to

```
280 GOTO 200
```

and replace

```
295 END
```

with

```
295 GOTO 200
```

Now, how do we terminate execution of the program? We may enter a value of 0 to indicate that there are no more batches to process. Then the statement

```
232 IF WEIGHT=0 THEN GOTO 999
```

may be used to divert program execution to statement 999 for a weight of 0. We had better include the statement

```
999 END
```

to avoid the following error message:

```
ERROR- 12 AT LINE 232
```

which is a "LINE NOT FOUND" error, since we sent the computer to line 999 and that line does not exist.

While we can always use the RESET key in response to an INPUT request, it is not desirable to depend on this method. RESET is more for programmers to use during program development. Our programs should provide for more orderly control. We often want further computing after the last INPUT item. RESET terminates the program, but the use of a special data value whose presence we can check for (sometimes called *dummy* data) will allow us to direct the program to continue processing.

...SUMMARY

Know your problem well before coding your solution program. Have a plan. It is easier to make major changes on paper than it is to make major changes in a program that has already been typed into the computer.

The IF . . . THEN statement may be used to determine the next statement to be executed while the program is running.

Use REMarks freely, but properly. Don't state the obvious. State the purpose of a statement or group of statements. It is vital that REM statements be accurate. Make sure that your program documentation keeps up with any changes in your program at all times. It is very frustrating to sort through program code that does not agree with the documentation. This can be worse than no documentation at all. However, don't use that comment as an excuse to omit REM statements.

Artificial values (dummy data) may be used as data to control what statements will be executed next.

Problems for Section 2-1 . . . .

At this point, we know enough about BASIC to program solutions to a wide variety of problems. We could find the sum of the counting numbers from 1 to 100, or from A to B as long as we don't exceed 1E97. We could find sums of even integers or odd

integers or those divisible by five, and so on. We could do something as simple as having the Atari display "I LIKE BASIC" some specified number of times. Use your imagination. You needn't limit yourself to the problems listed here. If you have an Atari to yourself, then you can answer all of those "I wonder what would happen if . . ." questions with, "I'll try it." The computer never raises its voice or remembers our "dumb" questions; it just beeps and tells us what is wrong.

1. In the birthday problem (Program 2-4), have the computer print the amount received for this birthday and the total so far for each birthday.

2. In the package-inspection problem (Program 2-5), make the changes necessary to repeat processing for many batches of data in a single execution. Insert at least one blank PRINT statement so that the batches are separated on the screen.

3. Rewrite the package-inspection program (Program 2-5) to test for a minimum package weight of 178 grams and a maximum package weight of 183. Have the program report the reason for rejecting a lot and repeat for another batch.

4. Four test scores were 100, 86, 71, and 92. What was the average?

5. Write a program to count the number of odd integers from 5 to 1191, inclusive.

6. Write a program to find the number and the sum of all integers greater than 1000 and less than 2213 that are divisible by 11. (Start with 1001.)

7. Rewrite Program 1-9 in Programmer's Corner 1 to ask for INPUT to get a number for the height of the graph paper. Use an IF . . . THEN statement to check that the graph paper has the desired height. Try it out with a height of 10 and also 50.

8. A person is paid $.01 the first day, $.02 the second day, $.04 the third day, and so on, doubling each day on the job for 30 days. Write a program to calculate the wages for the 30th day and the total for the 30 days.

9. Write a program to print the integers from 1 to 15, paired with their reciprocals.

10. A customer put in an order for four books that retail at $10.95 and carry a 25% discount, three records at $4.98 with a 15% discount, and one record player for $59.95 on which there is no discount. In addition, there is a 2% discount allowed on the total order for prompt payment. Write a program to compute the amount of the order.

11. In the song "The Twelve Days of Christmas," gifts are bestowed upon the singer in the following pattern: the first day she received a partridge in a pear tree; the second day two turtledoves and a partridge in a pear tree; the third day three French hens, two turtledoves, and a partridge in a pear tree. This continues for 12 days. On the twelfth day she received $12 + 11 + . . . + 2 + 1$ gifts. How many gifts did she receive altogether? Another way to ask this question is to ask: If she had to return one gift each day after the first, on what day would she return the last gift?

12. For Problem 11, have the computer print the number of gifts on each of the 12 days and the total up to that day.

13. George took tests in two courses. For the first course the scores were 83, 91, 97, 100, and 89. For the second course the scores were 65, 72, 81, and 92. Write a program that will compute both test averages. You will need two dummy-data values. One value will signal the end of this set of scores, and the other will signal the end of this execution of the program.

2-2...Random Events

How do they get the computer to flip coins, deal cards, or roll dice? These things are really very easy to simulate. All we need is the ability to generate numbers at random. BASIC includes exactly what we need for this. RND(X) produces a random number. RND is a little "black box" in BASIC that brings forth a number at random each time it is mentioned. Atari BASIC returns decimal numbers in the range 0 to .9999999999. The number enclosed in the parentheses is called the *argument* and can have any value at all, but it must be there. The random numbers generated will not depend on this value. In this book we will use RND(0), but you can substitute any value you desire in place of the 0. Let's look at a BASIC program to print ten random numbers.

```
100 REM * GENERATE A FEW RANDOM NUMBERS
200 I=1
230 PRINT RND(0)
240 I=I+1
250 IF I<=10 THEN 230
999 END
```

Program 2-6. Generate ten random numbers.

We have built a little counting routine that enables us to print RND(0) ten times. Here is a sample RUN of our program.

```
RUN
0.0843200683
0.537124633
0.259521484
0.0305480957
0.944244384
0.71685791
0.304061889
0.343460083
0.411056518
0.608581542

READY
```

Figure 2-4. Execution of Program 2-6.

33

Now we will adapt this new ability to flip a coin. Let's flip it 38 times to just fill one line of the screen without moving to the next line. There are three parts to this problem. We need to count to 38, generate a random flip, and print an H or a T depending. We know all about counting. We can decide whether to print an H or a T if we know how to tell which came up. All that remains is to organize how to distinguish heads from tails. We want half of each. So if we designate all of the random numbers from 0 to .4999999999 as heads and all of the numbers from .5 to .9999999999 as tails, the problem is solved. We merely test RND(0) in an IF . . . THEN statement. If we get less than .5 we branch to a statement that displays an H; otherwise we "drop through" to a statement that displays a T. Following the PRINT "T" statement, we must be sure to put in a GOTO statement to divert execution around the PRINT "H" statement. Here is a program to do just that.

```
198 REM * FLIP A COIN 38 TIMES
200 FLIPS=1
230 IF RND(0)<0.5 THEN GOTO 270
250 PRINT "T";
260 GOTO 280
270 PRINT "H";
280 FLIPS=FLIPS+1
290 IF FLIPS<=38 THEN GOTO 230
999 END
```

Program 2-7. Flip a coin 38 times.

```
RUN
HTHHTTTHTHHHHTTHHTHHTHHTTTHTHHTTTHHHT
```

READY

Figure 2-5. Execution of Program 2-7.

There you have it. We have accomplished what we set out to do. However, we really want to know as much about BASIC as possible. So, let's probe further.

. . .A RaNDom Exploration

The random-number generator may be bent to our needs in many ways. We have chosen to select two equal halves by forming a boundary at .5. This works fine for flipping a coin, but suppose we want to roll a die. Now there are five boundaries. We get numbers like .1666666667 and .8333333333. There is a much better way. If we multiply all numbers in the range of 0 to 1 (including 0 and excluding 1) by 6 then we get results in the range from 0 to 6 (including 0 and excluding 6). Then we could successively test to see if the result is less than 1, then less than 2, through 6, to get a value for the face of a die. This will certainly work, but it is not recommended. Once again Atari BASIC comes to the rescue. This time it is INT(N) that makes life simpler.

...INT(N)

INT(N) is a special function for developing an integer value that is the greatest integer less than or equal to the argument. Thus, INT(3.9876919) = 3, INT(4) = 4, and INT(−9.8) = −10. So, if we simply generate random numbers in the range from 0 to 5.999999999 then we can apply INT(N) to get integers in the range from 0 to 5. We merely add 1 to the values 0 to 5 to get values in the range 1 to 6. This is, of course, exactly what we want for rolling dice. Bingo—another problem solved. Let's look at Program 2-8 to roll a die ten times.

```
198 REM * ROLL A DIE TEN TIMES
200 I=1
210 VALUE=RND(Ø)*6+1
220 PRINT VALUE,INT(VALUE)
230 I=I+1
240 IF I<=10 THEN GOTO 210
999 END
```

Program 2-8. Roll a die ten times.

```
RUN
1.94400024          1
2.95794677          2
2.03811645          2
1.23355102          1
2.96746826          2
4.70770263          4
5.71047973          5
6.2300415           6
1.28765869          1
1.13687133          1

READY
```

Figure 2-6. Execution of Program 2-8.

There is a formula that is useful to know that allows you to generate random numbers in any desired range. You can have numbers from 1 to 100 or 5 to 23 or whatever your particular application requires. The formula is

$$X = INT (RND (0) * (A − B + 1)) + B$$

where A is the largest number we want and B is the smallest. For example, to generate random numbers from 23 to 67 (A = 67 and B = 23 in the formula), what we are doing is first generating numbers from 0 to 43 and then moving them up from zero by 23 (which, in this case, is B) so that they span the range from 23 to 67 as required. Try this out with a few examples of your own and you will begin to appreciate the usefulness of this formula.

...IF ... THEN Revisited

IF . . . THEN statements are used so frequently in BASIC to transfer program control that an abbreviated form exists.

```
240 IF I<=10 THEN 210
```

may be used in place of line 240 in our die-rolling program above. Using this new form of the IF . . . THEN statement, our coin flipping of Program 2-7 may be rewritten as follows:

```
198 REM * FLIP A COIN 38 TIMES
200 FLIPS=1
230 IF RND(0)<0.5 THEN 270
250 PRINT "T";
260 GOTO 280
270 PRINT "H";
280 FLIPS=FLIPS+1
290 IF FLIPS<=38 THEN 230
999 END
```

Program 2-9. Program 2-7 showing shortened IF . . . THEN.

...SUMMARY

RND(X) provides a source of random numbers from 0 to .9999999999. With appropriate manipulation, it is possible to use this function to generate random numbers within any desired range of numbers.

IF . . . THEN has an abbreviated form that we may use for conditional transfers. IF $X < 5$ THEN 230 will transfer control to line 230 if $X < 5$ is true.

Problems for Section 2-2 . . . .

1. Modify the coin-flipping program (Program 2-9) to repeat the 38 flips five times.

2. Modify the coin-flipping program (Program 2-9) to count the number of times tails comes up in 38 flips.

3. Write a program to flip a coin 1000 times. Count the number of tails. You might choose not to display H and T.

4. Write a program to roll two dice ten times.

5. Write a program to provide math drill problems in addition. Request limits and the number of problems by using INPUT statements. Display the number of correct answers at the end.

2-3...A Better Way to Count (FOR and NEXT)

Having written numerous counting loops, we imagine that there is some more compact method for doing this. After all, just about everything we do seems to involve counting of some sort.

...BASIC Loops

FOR and NEXT in BASIC automate the control functions of a program loop. Thus our earlier program to count from 1 to 7 becomes Program 2-10.

```
100 REM * COUNTING WITH FOR ... NEXT
110 FOR COUNTER=1 TO 7
115 PRINT COUNTER
130 NEXT COUNTER
999 END
```

Program 2-10. Program 2-3 with FOR . . . NEXT.

Statement 110 automatically establishes the limits on COUNTER as 1 and 7. Statement 130 automatically adds 1 to the value of COUNTER and tests to determine if COUNTER is less than or equal to 7. The value of COUNTER will be 8 when execution reaches line 999 of this program. If you want to save the last used value of the loop variable, then you need a statement such as 120 COUNTER2 = COUNTER in this program.

It is important to note that the statements between FOR and NEXT will always be executed at least once. If we program the statement FOR X = 4 TO 1, then the loop will be executed for X = 4. The NEXT statement will add 1 to 4, getting 5, and then find that X is greater than 1, and execution will "drop through," behaving in exactly the same way as our "hand-built" loops. To count from A to B by twos, simply code FOR COUNTER = A TO B STEP 2. We may use STEP −3 or even N. The steps are not restricted to integers. Any decimal value may be used—you can STEP by 0.1 or 3.33. If no value of STEP is given, a value of 1 is assumed by BASIC.

The FOR and NEXT statements provide several important benefits. FOR and NEXT loops execute faster than the identical hand-built variety. Their use reduces the number of ideas that we have to store in our heads as we write our programs. Those simple BASIC keywords embody the more complex controls actually used to construct the loop itself without requiring us to think about the detail each time that we use them, thus freeing our mental processes for solving the specific problem at hand. The ability to make a small number of program statements represent complex solutions greatly simplifies the writing of correct computer programs.

Now we can think about some of the counting loops we have looked at before. Consider the birthday dollars program (Program 2-4): in the original program, we had a line 110 COUNTER=1. That line happened to be the opening statement of a counting loop, but that statement could set the value of COUNTER to 1 for zillions of reasons. On the other hand, the statement

```
110 FOR COUNTER=1 TO 21
```

is crystal clear. It can mean only one thing: we are going to do something 21 times. In exactly the same manner, NEXT COUNTER conveys much more information to the person reading the program than

```
150 IF COUNTER<=21 THEN GOTO 120
```

FOR and NEXT are designed to go together. Don't try to initialize a loop with

```
100 COUNTER=1
```

and later close it with

```
200 NEXT COUNTER
```

If you do, when you RUN the program, you will get the error message

```
ERROR- 13 AT LINE 200
```

which is Atari BASIC's way of signalling a "NEXT WITHOUT FOR" error.

Occasionally, you will be sure you have a loop to repeat something several times. But, alas, it only happens once, and the computer sends no error messages. While the computer requires that a NEXT statement be preceded by a FOR statement, it does not necessarily report that a FOR statement was not followed by a NEXT statement. Now you know.

...SUMMARY

FOR and NEXT are paired up to control program loops in BASIC. For A = B TO C STEP D opens a loop by assigning the value of B to A. Each iteration of the loop is accomplished by adding the value of D to the value of A. When the value of A "goes past" the value of C, the loop is done. NEXT A causes the next iteration of the loop that was opened with the FOR A . . . statement. If the step value is not specified, it is assumed to be 1.

Problems for Section 2-3 . . . .

For each of the problems here, use FOR and NEXT where appropriate.

1. Modify the package-inspection program (Program 2-5) to use FOR and NEXT.

2. Write a program to count the number of odd integers from 5 to 1191 inclusive.

3. Write a program to find the number of and the sum of all integers greater than 1000 and less than 2213 that are divisible by 11. (Start with 1001.)

4. A person is paid $.01 the first day, $.02 the second day, $.04 the third day, and so on, the amount doubling each day on the job for 30 days. Write a program to calculate the wages for the 30th day and the total for the 30 days.

5. Write a program to print the integers from 1 to 15, paired with their reciprocals.

6. Do the "Twelve Days of Christmas" problem using FOR and NEXT.

7. For Problem 6, have the computer print the number of gifts on each of the 12 days and the total up to that day.

8. Modify the coin-flipping program (Program 2-9) to repeat the 38 flips five times.

9. Modify the coin-flipping program (Program 2-9) to count the number of times tails comes up in 38 flips.

10. Write a program to flip a coin 1000 times. Count the number of tails. You might choose not to display H and T.

11. Write a program to roll two dice ten times.

12. Write a program to provide math drill problems in addition. Request limits and the number of problems using INPUT. Display the number of correct answers at the end.

13. Examine the following program:

```
100 FOR I=1 TO 1.3 STEP 0.1
110 PRINT I
120 NEXT I
```

What values do you think it will display? Run it. Do you get what you expect? Write a program to display the four values you expected.

PROGRAMMER'S CORNER 2

Besides terminating a program during execution, the BREAK key serves another function. It provides you with a way to cancel a line being entered into a program, just as RETURN provides a way of accepting a line. Pressing the BREAK key on any part of a line being edited will move the cursor off that line and allow you to start over.

Accidentally hitting the BREAK key during the RUN of a program can be a real annoyance, since it abruptly brings things to a halt and frequently means that you have to start over. You can disable the BREAK key with the following POKE statements.

```
POKE 16,64
POKE 53774,16
```

This is not total protection, because some operations of the computer (including pressing SYSTEM RESET) will cancel out the effect of these POKE statements. However, for most programs, it will at least provide a simple, albeit imperfect, protection against accidental interruption of your program.

The TAB key can also be useful in editing program statements. Use of TAB will move the cursor in jumps across a line without erasing the characters that the cursor passes over. Of course CTRL and the arrow keys also do this, but it takes two hands.

...Error Trapping

Any program that requires input from the user, and many do, is open to "crashing" because the user inputs information that the program will not accept. A program looking for numeric input will stop in its tracks if the user inputs a letter of the alphabet instead or just presses the RETURN key. You will want to go to great pains to ensure that your programs do not halt because of this type of input. One

important lesson is to be sure to let others try out your programs. As the author, you know exactly what type of input a program requires and will be inclined to provide only that type of input. On the other hand, a first-time user may not understand what is being asked for and innocently hit the wrong keys. The following program protects against incorrect input.

```
 90 REM * INPUT PROTECTION DEMONSTRATION
110 PRINT "PICK A NUMBER FROM 1 TO 20"
120 TRAP 200
122 INPUT NUMBER
123 PRINT NUMBER
125 IF NUMBER<1 OR NUMBER>20 THEN 999
130 PRINT "GOOD GUESS! THAT'S MY NUMBER, TOO."
140 PRINT
150 GOTO 110
200 TRAP 40000
205 PRINT "INVALID INPUT. TRY AGAIN."
210 PRINT
220 GOTO 110
999 END
```

Program 2-11. Input protection demonstration.

```
RUN
PICK A NUMBER FROM 1 TO 20
?INVALID INPUT. TRY AGAIN.

PICK A NUMBER FROM 1 TO 20
?16
GOOD GUESS! THAT'S MY NUMBER, TOO.

PICK A NUMBER FROM 1 TO 20
?45

READY
```

Figure 2-7. Execution of Program 2-11.

Notice the TRAP 200 in line 120. We are telling the computer that if any problem occurs, it should transfer the program to line 200 instead of simply stopping and reporting an ERROR code. The TRAP statement must be executed before the error occurs. Also, once the TRAP statement is used it is cancelled out and can only be reactivated by another TRAP statement. In our case, our program is a loop that will go back to line 110 and thus will reactivate the TRAP statement when execution arrives again at line 120. If the TRAP statement is tripped, the program goes to line 200. The normal procedure of stopping and giving an ERROR message is deactivated and our program is now responsible for handling what happens next. TRAP N, where N is between 32768 and 65535, will cancel the TRAP statement. We have used TRAP 40000 because it is in this range and easy to type. Once the error is trapped, we display a warning message and go back to try new

input. Now, in spite of erroneous input, our program will continue to execute. We will make use of TRAP statements in later programs to catch other types of errors.

...Clearing the Screen

You may have noticed and been somewhat annoyed by how much gets printed on the screen during execution of some of our programs. This is particularly true of programs that loop around many times, each time asking for similar input. Sometimes it is useful to have all the values on the screen at the same time, but most often it merely clutters up the display. Atari BASIC has a way to clear the screen at any time. Holding down the SHIFT or CTRL key and pressing the CLEAR key will instantly clear the screen and position the cursor in the upper left corner. Or you can type

```
PRINT CHR$(125)
```

(We'll get into this strange CHR$ in the next chapter.) Add a line number and this also works within a program. Although SHIFT or CTRL plus CLEAR won't work in a program (try it and you'll see why!), if you press the ESC key before keying it in you will see a left curving arrow that the computer will interpret as a clear screen command when the program is RUN. These commands needn't be used by themselves. We could just as well use

```
100 PRINT " We have just cleared the screen!"
```

which will clear the screen and then type the message. We will use this capability to make our screen displays more attractive.

...The Built-In Buzzer

There is a buzzer available for use in your programs. From immediate mode just press CTRL and 2 together and you'll hear it. In a program, if you type

```
100 PRINT CHR$(253)
```

or

```
100 PRINT "ESC CTRL+2"
```

you'll hear the buzzer when the program runs. In the second case above, you will see a right curving arrow between the quotation marks when you type in the keys noted.

...Repeating Keys

Holding down any key on your Atari keyboard (except SHIFT, BREAK, and the ones on the far right of the keyboard) will cause the character to repeat continually after about two seconds. You will find this very handy, particularly with the SPACE bar.

Chapter 3

Beginning Atari Graphics and Much More

3-1...Graphics Capabilities

...A Graphics Example

It is one thing to program a computer to simulate the roll of a die and display a numeric result. It is another to program a computer to display a realistic picture of the die. In many programs, the presence of a graphics display can enhance a program's impact and even make it easier to use. In games, of course, the graphics are often the whole show. Your Atari computer can produce dazzling graphics in many shapes and forms. In this chapter we will begin a gentle introduction to graphics and also introduce you to some more advanced programming instructions. To do this we will take our example of rolling dice and use the Atari's graphics to display the die on the screen. Before we can do this, we need to cover some basic concepts of Atari graphics that will serve as a foundation for a more detailed look at graphics, which will appear in Chapter 11.

...The Atari Graphics Modes

Depending on which model of Atari computer you have, and how old it is, you will have either 9, 12, or 15 graphics modes. These include three text modes, six regular graphics modes, and three or six special graphics modes. These differ in a number of ways that we will cover in our later discussion. Suffice it to say here that the biggest difference between these modes is the degree of detail (resolution) that they are capable of displaying on the TV screen and whether or not they are primarily intended for displaying text. So far we have been using graphics

mode 0 for all our work. This is the graphics mode that the computer is in when you turn it on; it is one of three modes that are designed for use mainly with text (along with modes 1 and 2).

...Atari Colors

For our dice rolling example, we will be using graphics mode 3, which is the lowest resolution of the nontext graphics modes (which go from mode 3 up to mode 15 on some models). Mode 3 is a four-color mode, three for the foreground and one for the background.

Figure 3-1 shows the 16 colors on the palette from which we may choose our colors. In our programs we can change colors whenever we choose. One simple command will instantly change anything drawn in a particular color to another of our choosing. The 16 colors available may each be shown in eight different intensities, thus giving us a tinted palette of 128 colors. We assign the colors by using the statement

```
SETCOLOR A,B,C
```

where A is the color register, B is the number of the color desired (0–15), and C is an even number from 0 to 14 representing the intensity, with 14 as the brightest. In graphics mode 3, the color register may have the values 0, 1, 2, or 4. Register 4 is our background; registers 0, 1, and 2 are available to us for drawing on the screen. We do *not* have to assign colors with the SETCOLOR statement if we are willing to make use of the default colors that are built into Atari BASIC (those that are already assigned when the computer is turned on). In graphics mode 3 we have the following default colors:

REGISTER	COLOR
0	Orange
1	Light green
2	Dark blue
4	Black (background)

SET COLOR VALUE	COLOR	SET COLOR VALUE	COLOR
0	Gray	8	Blue
1	Light orange	9	Light blue
2	Orange	10	Turquoise
3	Red-orange	11	Green-blue
4	Pink	12	Green
5	Purple	13	Yellow-green
6	Purple-blue	14	Orange-green
7	Blue	15	Light orange

Figure 3-1. The Atari colors.

We mention here just once that the colors you see on your TV will be a function of the way your TV (and to a lesser extent, your Atari computer) are adjusted. Of course, if you are working with a black-and-white TV, different colors will appear as different shadings and you will have to experiment to find combinations that will produce effective graphics displays for your programs. We strongly recommend the use of a color TV to allow you to show off the Atari graphics in their best light (so to speak!).

Once you have informed your computer which colors you want to use (by placing the desired color numbers in the color registers using SETCOLOR), you can select the color to be used at a particular time with

```
COLOR A
```

where A is a number that tells the Atari which SETCOLOR register you want to use. Now you might think that it would be logical to use 1 for color register 1, 2 for color register 2, and so on. Because of considerations in the development of Atari BASIC, however, things are not that simple or logical. Instead, you will have to remember that in graphics mode 3 you call for the use of color registers 0, 1, 2, or 4 with the COLOR statement by using the values 1, 2, 3, or 0, respectively. Thus, color register 0 (default color of orange) is called by a COLOR 1 statement and color register 4 by COLOR 0. You must use a COLOR statement to draw on the screen even if you are using the default colors.

...Plotting Points and Drawing Lines

We are now able to change the color of any register at will and select that register for use in drawing. Next we need to look at how we tell the computer where we want to draw on the screen. We do this using the PLOT and DRAWTO statements. PLOT X,Y is the way we move our paintbrush to the position that is X units to the right of the upper left-hand corner and Y units down. Figure 3-2 shows just what we mean. Graphics mode 3 permits us to display 40 "dots" across and 20 "dots" down the screen. The upper left corner is labeled 0, 0; each location on the screen has a unique set of X and Y values. Shown on the figure are two more (X, Y) points, (12, 5) and (4, 12). Notice that, contrary to usual graphing procedures, we count from the upper left corner of the display. The PLOT statement moves our marker to the desired location and fills in that position with a mark of the desired color. Once our pen is "down" we can use DRAWTO X1,Y1 to draw from this location to the new location that is labeled (X1, Y1).

We can use graphics mode 3 and our ability to draw on the screen to write a program to display all the colors available. We'll make the program draw horizontal bars on the screen and then we will cycle through all the colors and brightnesses using two FOR . . . NEXT loops. Within the loops we will use the SETCOLOR statement to change the color of the bars on the screen. To help us keep track of what color and brightness we are looking at, we will PRINT these values in the text window at the bottom of the screen. Look at Program 3-1.
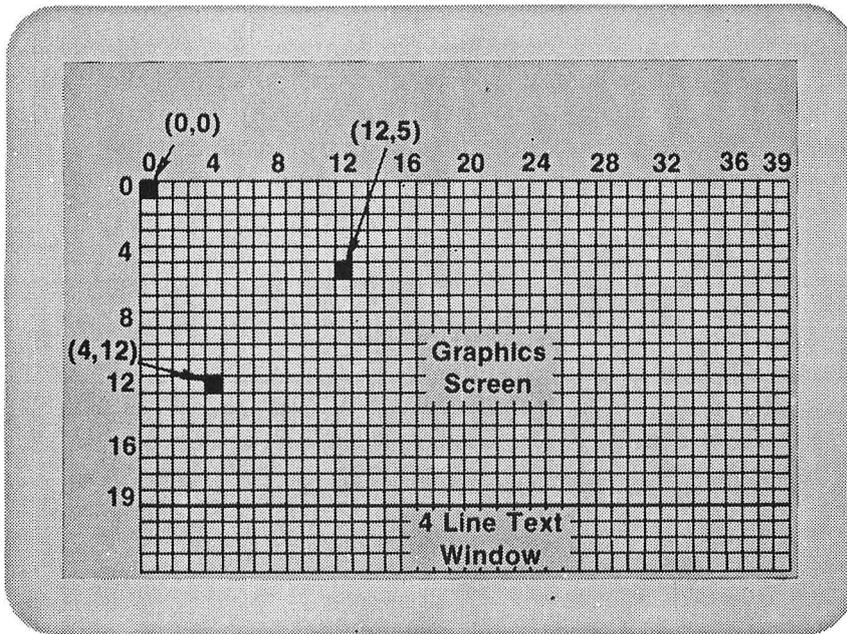
Figure 3-2. Graphics mode 3 with two points PLOTted.

```
0 REM * GRAPHICS 3 COLOR BARS
5 GRAPHICS 3
7 SETCOLOR 0,0,0
10 COLOR 1
20 FOR LINE=2 TO 18 STEP 2
30 PLOT 2,LINE
40 DRAWTO 36,LINE
50 NEXT LINE
60 FOR COL=0 TO 15
75 FOR LUM=0 TO 14 STEP 2
77 SETCOLOR 0,COL,LUM
78 PRINT "SETCOLOR 0,";COL;",";LUM
80 FOR DELAY=1 TO 250
85 NEXT DELAY
87 PRINT :PRINT
90 NEXT LUM
100 NEXT COL
```

Program 3-1. Graphics mode 3 demonstration.

Notice the use of meaningful variable names to help us understand the programming used.

...Drawing a Die

Let's use this information to set up how we want our six die faces to look. The sketch in Figure 3-3 shows them all, with numbers indicating the X and Y values that are involved. We will initially use the default colors. We'll draw our "1" die as a 7-by-7 square in the upper left corner with the "1" spot in the center of the die face.
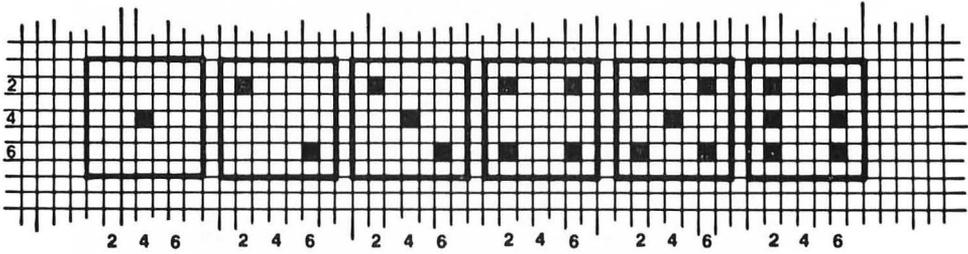
Figure 3-3. Six die faces with corresponding coordinates.

```
98 REM * THE "1" FACE OF A DIE
100 GRAPHICS 3
110 COLOR 1
120 FOR I=1 TO 7
130 PLOT 1,I
140 DRAWTO 7,I
150 NEXT I
160 COLOR 2
170 PLOT 4,4
180 END
```

Program 3-2. Draw the "1" face of a die.

Notice that what we have done is draw a blank die and used the PLOT statement to color in one location with a different color (in this case, COLOR 2). We could, of course, have written the program as a series of PLOT and DRAWTO statements, but putting the whole thing in a loop makes the program much easier to write. Also note that our screen is divided into two parts. We have our 40 by 20 graphics mode 3 screen at the top and a four-line graphics mode 0 text screen at the bottom. Type in the following statement (in the text window at the bottom of the screen) in the immediate mode to see our die change color to blue:

```
SETCOLOR 0,8,6
```

or try

```
SETCOLOR 0,8,10
```

for a lighter blue. Next try

```
SETCOLOR 4,12,6
```

to change the background to green. We have drawn the die using COLOR 1, which called for color register 0; by changing B and C in SETCOLOR 0,B,C we can have

any color and intensity we want. SETCOLOR and COLOR are like the multiplication tables that you learned in third grade: they can only be learned by practice. It may be helpful in relating SETCOLOR to COLOR to point out that the number used in a COLOR statement is one more than the SETCOLOR register desired (where one more than 4 gets you back to 0).

We can expand the graphics screen to cover the entire TV screen (eliminating the text window) merely by adding 16 to the number we specify in the GRAPHICS statement. This is true for all the various graphics modes available. Thus, either

GRAPHICS 19

or

GRAPHICS 3+16

will give us a full screen GRAPHICS 3 display. Try putting this change into Program 3-2 and then RUN the program again. If you watched carefully, you saw a glimpse of our die; then the screen cleared and we were back in graphics mode 0 everywhere. This will always happen when a program finishes executing if we use full-screen graphics. Generally this will not be a problem, because your program will go on to do other things. As long as the program does not end (or receive an instruction to change graphics modes), you will stay in the graphics mode assigned. For our short demonstration programs, however, it is a problem, and we will therefore keep the split screen display, which avoids this problem by keeping the top portion as a graphics screen. However, the four-line display at the bottom is not very handy for making changes in a program, since we can't see enough of the program at one time. It is much better to type GRAPHICS 0 (or its abbreviation GR. 0) in the text window, which will clear the screen and put us back in graphics mode 0 with its 24-line display. We can then do our editing and again RUN our modified work of art to see it.

Our "1" die is pretty nice. How do we get a "3?" Simply add the following two statements and RUN the new program:

165 PLOT 2,2

175 PLOT 6,6

There are seven positions on the face of the die where a dot may appear: 2, 2; 2, 4; 2, 6; 6, 2; 6, 4;, 6, 6; and 4, 4. By properly selecting from among these seven, we may draw any of the six faces.

...SUMMARY
We can use SETCOLOR and COLOR statements to select the color for use in drawing on the screen after we have used the GRAPHICS statement to determine which graphics mode we are going to work in. In graphics mode 3, we have a 40-by-20 display to work on. This may be expanded to 40 by 24 by adding the value 16 to our GRAPHICS value. We have 16 colors with eight intensities for each one. We can plot points with PLOT statements and draw lines with DRAWTO statements.

The positions on the screen are all measured from the upper left corner, with the first value referred to by PLOT and DRAWTO statements being measured across the top of the screen and the second being measured down the left edge. We can erase the graphics screen and regain the full screen for text editing by typing GRAPHICS 0 or GR. 0.

Problems for Section 3-1 . . . .

1. Write a program to display a die showing the "6" face in the upper right corner of the screen.

2. Write a program to display a pair of dice, one showing a "1" and the other showing a "3".

3. Write a program to request a number from 1 to 6 and display the appropriate die (try using colors if you have a color TV).

4. Write a program to draw a bar graph picturing the following dollar sales in thousands for a nine-week period:

WEEK	SALES
1	17
2	12
3	15
4	6
5	12
6	18
7	15
8	14
9	15

3-2 . . . Divide and Conquer (Subroutines)

Once we have written the code to display a die of a particular color having a particular face value in a particular place, it is hard to be inspired to write a new code to display that same die in another location or another color. And it is even less exciting to consider displaying five more dice this way. When we find ourselves writing routine after routine, each one only a slight variation of the one just finished, programming becomes tedious. The more experience we gain in programming, the more opportunity we will have to utilize what we have already done. Often a current problem is only a slight variation of an old, already solved one.

   If we want to display an orange die and then a blue die in the same location, the only thing that changes is the color. Clearly, it is a nuisance to duplicate the code that does the actual graphing. We can easily isolate that code and direct the computer to execute it at will by using GOSUB and RETURN.

...GOSUB and RETURN

GOSUB 1000 causes the computer to execute line 1000 next regardless of the next statement numerically in sequence. However, GOSUB 1000 differs from GOTO 1000 in that GOSUB remembers its place in the program. When a RETURN statement is encountered, execution resumes following the most recent GOSUB. The program statements that begin with the line number following the keyword GOSUB and ending with a RETURN statement are grouped and referred to as a *subroutine*. Thus GOSUB means "GO do the SUBroutine."

Atari BASIC also permits the line number specified in a GOSUB statement to be a variable. We could use

```
120 GOSUB DRAWONE

130 GOSUB DRAWTWO
```

as long as we had statements such as

```
10 DRAWONE=1000

20 DRAWTWO=1100
```

earlier in the program. Notice how much more descriptive the second method is. When you look at this program weeks from now, you will remember what each part is doing much more clearly.

For our orange die followed by a blue die, we need to have the program pause between the two displays. Otherwise, things will happen so quickly that we will not see the first die. This pause can be accomplished with a time-waster FOR . . . NEXT loop that does nothing else. The problem is solved in seven easy steps as follows:

1. Enable graphics mode.
2. Set orange color.
3. Display the die.
4. Waste some time.
5. Set blue color.
6. Display the die.
7. End.

Putting off for the moment writing the actual die-display subroutine, let's look at a program to control the display of orange and blue dice. See Program 3-2a.

```
98 REM * CONTROL DIE DISPLAY
100 GRAPHICS 3
110 COLOR 1
120 GOSUB 1000
152 FOR DELAY=1 TO 1500
154 NEXT DELAY
160 COLOR 3
170 GOSUB 1000
190 END
```

Program 3-2a. The control segment of a die-drawing program.

49

We have been able to embody a group of statements in the single statement

```
120 GOSUB 1000
```

Again, we have a method for organizing our thoughts more easily by concentrating many computing steps in a single statement. We can think of

```
GOSUB 1000
```

as "display a die" without having to think about the actual BASIC statements required to do the display. Look at lines 152 and 154. Those two lines make up a delay loop. For a longer delay, use a value larger than 1500. Without a delay, we would barely see the first die because it would be so quickly replaced with the second die.

Finally, the display routine is very easy. We may simply select those statements from our earlier die-drawing program and use appropriate line numbers. We may concentrate on the display without having to think about other parts of the program. We know that the first line should be numbered 1000 and that the last statement should be RETURN. See Program 3-2b.

```
998 REM * DISPLAY A "1" DIE
1000 FOR I=1 TO 7
1010 PLOT 1,I
1015 DRAWTO 7,I
1020 NEXT I
1030 COLOR 2
1040 PLOT 4,4
1050 RETURN
```

Program 3-2b. Subroutine to display a "1" die.

Programs 3-2a and 3-2b together make up a complete program to display the "1" die in two different colors with a brief delay in between.

It is important to realize the impact of the END statement at line 190 in Program 3-2a upon the subroutine beginning at line 1000. Without the END statement, the program will continue through the following lines until it reaches line 1050. You will then be greeted with

```
ERROR- 16 AT LINE 1050
```

because it is improper to execute a RETURN statement without a matching GOSUB. The END statement in line 190 assures that the routine at line 1000 is not executed an extra time.

Soon we will see that it is useful to separate the pieces of the program even further by using another subroutine to display the dots on the die. This will enable us to set a color for the dots easily and to plot any of the six possible faces.

...Make It Handle the General Case

Wouldn't it be nice to be able to display a die anywhere on the screen? With the idea of subroutines well in hand, this new twist is easy. All we need is to "send" to our

subroutine values that specify where a corner of the die is to be. Whatever this "corner" location is, we merely add it to the coordinates of our die "number."

We have already used this idea. We "sent" different color codes to the same subroutine at line 1000 in Program 3-2. Using X and Y as the horizontal and vertical position of the upper left corner of the die, we get the following subroutine to display the "1" anywhere on the screen.

```
998 REM * DISPLAY A "1" DIE
1000 FOR I=1 TO 7
1010 PLOT X+1,Y+I:DRAWTO X+7,Y+I
1020 NEXT I
1030 COLOR 2
1040 PLOT X+4,Y+4
1050 RETURN
```

Program 3-3. Drawing a "1" anywhere on the screen.

However, we must assure that the values of X and Y place the entire die within the 40-by-20 graphics screen. That means that X may range from 0 to 33 and Y is limited to values from 0 to 13 for our 7-by-7 die face. If we exceed either of these limits we will end up trying to draw at a location not permitted on our GRAPHICS 3 screen, and we will get the message

```
ERROR- 141
```

Now the final piece of the puzzle will fit into place as soon as we write six subroutines—one for each of the six possible faces of a die. Numbering the first lines of our subroutines 1100, 1200, . . . , 1600 will help to identify the purpose of each subroutine. Thus:

```
98 REM * PLOT ONE
1100 PLOT X+4,Y+4
1110 RETURN
1198 REM * PLOT TWO
1200 PLOT X+2,Y+2
1210 PLOT X+6,Y+6
1220 RETURN
     :
1598 REM * PLOT SIX
1600 PLOT X+2,Y+2
1610 PLOT X+2,Y+4
1620 PLOT X+2,Y+6
1630 PLOT X+6,Y+2
1640 PLOT X+6,Y+4
1650 PLOT X+6,Y+6
1660 RETURN
```

Now we may remove lines 1030 and 1040 from our die-display subroutine. The display separates nicely into showing the background and plotting the spots. These

two functions are now done with distinct subroutines. GOSUB 1000 displays the background. GOSUB 1100 through GOSUB 1600 may be used to display one through six spots on the die. We can set the colors independently. Once a die has been drawn on the screen, we can set color 0 and call upon the background display routine to erase the die, spots and all.

It is clear that once we have a number, such as R, that tells us how many dots to plot on a die, we need a way to branch to the appropriate subroutine. Thus, we wish to execute just one of the following statements:

```
GOSUB 1100
GOSUB 1200
GOSUB 1300
GUSUB 1400
GOSUB 1500
GOSUB 1600
```

We could do that with the following logic:

```
910 IF R<>1 THEN 920
912 GOSUB 1100
914 GOTO 990
   .
   .
   .
950 IF R<>5 THEN 960
952 GOSUB 1500
954 GOTO 990
960 IF R<>6 THEN 990
962 GOSUB 1600
964 GOTO 990
990 RETURN
```

All that typing is cumbersome, however, and it requires 18 statements to make a very simple decision. Our goal is always to simplify things. We could eliminate the six GOTO 990 statements, which are not essential because the IF . . . THEN statements will ensure that we don't draw the wrong die. That would leave us with 12 statements, but we still have a choppy structure that is unnecessarily long and difficult to read (for humans—the computer doesn't care).

...Another Visit with IF . . . THEN

We can use a new feature of IF . . . THEN statements to simplify the decision on which of the six die-display subroutines to execute. This new feature makes it possible to achieve the same result with six simple BASIC program lines.

Any BASIC statement may follow THEN in an IF . . . THEN statement. We may execute just one of the die-display subroutines with the following code:

```
910 IF R=1 THEN GOSUB 1100
920 IF R=2 THEN GOSUB 1200
930 IF R=3 THEN GOSUB 1300
940 IF R=4 THEN GOSUB 1400
950 IF R=5 THEN GOSUB 1500
960 IF R=6 THEN GOSUB 1600
```

Not only is this shorter to type, but it is much clearer to read. For any value of R in the range 1 to 6, just one of the IF . . . THEN tests comes out true. The other five come out false. Thus, the computer executes all six IF tests no matter what. But the computer is very fast, and the five false results will not delay execution noticeably for our present problem. Combining this feature with random numbers, we can program a wide variety of events.

Problems for Section 3-2 . . . .

1. Write a program to display a die face showing a "6" in the upper right corner of the graphics screen.
2. Write a program to display a random die face in the upper left corner of the screen.
3. Display a random die face, leave it for a few seconds, and then erase it.
4. Display two dice at random next to each other in the lower left corner.
5. Write a program to display a blinking die. Let it blink ten times, then leave the display on the screen.
6. Display a few dice at random in random locations on the screen to simulate physically rolling the dice. Then display a pair of dice at random and leave them on the screen.
7. Take our "1" die program (Program 3-3) and make it cycle through all 15 possible colors (with the intensity at 6) for the die spots.
8. Repeat Problem 7, but have it keep the original colors and loop through the possible intensities.

3-3 . . . BASIC Multiple Features

. . . GOSUB Revisited

At first we saw how we might use 18 statements to implement branching to one of six die-display subroutines. We reduced this to six easier-to-read lines using an extended feature of IF . . . THEN—executing any statement if the tested expression is true. Now we reduce this even further with a new feature of the GOSUB statement.

In BASIC, we can do the work of the six IF statements in one line by using the multiple GOSUB capability.

```
910 ON R GOSUB 1100,1200,1300,1400,1500,1600
```

If R = 1, GOSUB 1100 is executed. If R = 2, the computer executes GOSUB 1200, and so on. Should the value of R be less than 1 or greater than 6, statement 910 will be ignored. However, vaues less than 0 or greater than 255 will be rewarded with

```
ERROR- 3 AT LINE 100
```

which signifies a bad value error.

Atari BASIC has another powerful method for handling this sort of statement.

We have branched to one of several subroutines (by indicating the starting line number of that subroutine), depending on the value of a variable. We can also branch to a subroutine whose starting line number is given by the result of a calculation. Thus our previous ON . . . GOSUB statement could have been written

```
910 GOSUB 1000+100*R
```

For R values of 1 to 6 these two statements will behave in exactly the same way. In the second case, however, we will get a LINE NOT FOUND error (ERROR 12) if R is less than 1 or greater than 6. We can guard against this outcome by checking the value of R before we get to line 910. In this example, there is little reason to use the second method. In more complex programs, however, the ability to branch to a calculated line number can be of great value.

Now even the relatively simple six-statement logic used to branch to the proper spot-plotting subroutine has been reduced to a single statement. Lest we get the idea that all programs can be reduced by at least one statement (and therefore eliminated entirely), be assured that there is a limit to the features available in BASIC or any other computer language. Computers are finite and therefore limits do exist. Computers and computer languages are amazing, but they cannot perform magic.

...Nested GOSUBs

We put subroutines to good use in the die drawing of the last section. It is worth noting that some plot statements were repeated. A 3 is just a 1 superimposed on a 2. Thus:

```
1300 PLOT X+2,Y+2
1310 PLOT X+6,Y+6
1320 PLOT X+4,Y+4
1330 RETURN
```

becomes:

```
1300 GOSUB 1200
1310 GOSUB 1100
1320 RETURN
```

A 4 is a 2 with two extra spots and can be plotted as follows:

```
1400 GOSUB 1200
1410 PLOT X+2,Y+6
1420 PLOT X+4,Y+2
1430 RETURN
```

Similarly, a 5 is a 1 superimposed on a 4, and a 6 is a 4 with two extra spots.

In the above example, we have called one subroutine from within another. This is often very useful. Subroutines within subroutines are called *nested subroutines*. We may nest subroutines up to a theoretical maximum of 128 deep in Atari BASIC. This is theoretical because the actual maximum is decreased by one for each

variable in the program. In any event, we are unlikely to run into the problem of going over the limit on nested loops in any program we write.

What have we accomplished by all of this? There are two distinct benefits. We have made the spot-display subroutines more compact. Thus we can fit more of the program on the screen at once. Once again, we have made the programming process more orderly through careful packaging. We have also included each of the spot-plotting statements exactly once. This reduces the possibility of error. If we have made an error—say the central spot is misplaced—we need look for only a single PLOT statement to fix it for all die faces containing that spot. Any practice that makes programs easier to read or easier to change, or gives them better structure, is to be encouraged.

...GOTO Revisited

GOTO has the same multiple line-number branching capability as GOSUB. Thus the single Atari BASIC statement:

```
100 ON N1 GOTO 310,320,330,340,350,360,370
```

replaces seven IF . . . THEN statements.

Suppose we have a situation in which we want to execute line 1000 if N1 = 3, 1100 if N1 = 6, and 1200 if N1 = 11. Do not be tempted to use the multiple GOTO or GOSUB capability of BASIC in such a situation—it is much clearer to code three IF . . . THEN statements. Using 11 line numbers, only three of which are real, is very confusing to anyone reading your program. Don't do it! Even you won't understand it next week.

...Multiple Statements

The ability to place several program statements on a single numbered line has some useful applications. Suppose we have a subroutine at line 500 that requires us to set values for A, B, and C. This will generate several sets of lines of the following form:

```
100 A=5
110 B=9
120 C=3
130 GOSUB 500
```

Where certain statements naturally belong together, it is nice to be able to place them all on the same line. Using the colon (:) to separate statements, we may use the following equivalent code:

```
100 A=5:B=9:C=3:GOSUB 500
```

While it may be very nice to place several statements on the same line, there may be good reasons not to. Doing so can make the line a little harder to edit, and it is more likely to make the program harder to read when LISTed lines appear on two screen lines with a break in the middle of a word. This capability should be used with caution.

Line numbers do require space in the computer's memory. Occasionally, a

program grows to the point where it is too big for the available memory. One method of reducing the amount of memory a program requires is to use multiple statements on each line. In doing this to an existing program you must be careful that you don't change the logic of the program by incorrectly combining lines that are referenced by a GOTO, IF . . . THEN, or GOSUB statement.

...Multiple Statements and IF . . . THEN

BASIC allows multiple statements following IF . . . THEN statements. A statement such as

```
100 IF A=5 THEN B=6:C=11
```

is perfectly legal. That statement will execute both B = 6 and C = 11 when A = 5, and neither B = 6 nor C = 11 if A does not equal 5.

If you are not comfortable with this logic, you may use the following code to implement the same logic:

```
100 IF A<>5 THEN 120
110 B=6:C=11
120 rest of program
```

The fact that you know of a certain feature does not mean that you should use it frequently or even at all. It is good to have a broad collection of capabilities available for use in the appropriate situation. It is also good to be aware of as many features as possible so that you can understand other people's programs. Know the language and use it well. It is a mistake to bend the logic of a program so that you can use some cute program statement. Cute or tricky programs are difficult to read. Some programmers like to embed tricky logic in their programs so that "nobody will ever figure it out." That is just why you should not do it. Even you will never figure it out later when you want to change it.

For example, your original program may have looked like this:

```
     .
     .
     .
500 IF X=3 THEN GOTO 750
510 Y=7
520 Z=5

     .
     .
     .
```

If we use multiple statements per line without considering the logic of the program, we can end up with:

```
500 IF X=3 THEN GOTO 750:Y=7:Z=5
```

In the first case, if X < > 3 then we will go on and get to Y = 7 and Z = 5. But in the condensed version, if X < > 3 we will go on to the rest of the program and the

56
•••

values of Y and Z will never be assigned (and hence, they will be assumed to be 0). So be very careful to consider what the program is supposed to do when you begin to compress a program by putting multiple statements on one line.

...SUMMARY

Using GOSUB . . . RETURN allows us to divide a program into logical blocks that simplify writing and understanding the code. You may use variables and calculated values in GOSUB calls. Variables and arithmetic expressions may also be combined with IF . . . THEN statements in structuring your programs. ON X GOSUB. . . and ON X GOTO. . . are other variations on this theme. Atari BASIC permits multiple statements on one line with a colon separating them. This feature must be used judiciously so as not to mess up the program logic or make the program difficult to understand.

PROGRAMMER'S CORNER 3

...Graphics Mode 5 and Graphics Mode 7

Our understanding of graphics mode 3 carries over directly to graphics mode 5 and graphics mode 7. Anything you did with graphics mode 3 can be done with these new modes. The SETCOLOR and COLOR statements are identical. The only difference is that while mode 3 gives you a 40-by-24 screen (with the text window at the bottom removed), mode 5 gives you an 80-by-48 screen and mode 7 gives you a 160-by-96 screen. This means that you can show much greater detail with these higher-resolution screens with the same four-color display. The penalty that you pay is the amount of memory required by the computer to maintain the screen. Mode 3 requires only 432 bytes, but mode 5 requires 1176 and mode 7 needs 4200. The differences in resolution are shown in the following program (use BREAK to stop the show).

```
90 REM * DEMO FOR GR. 3,5 AND 7
100 XMAX=40
105 X=INT(XMAX/2)
110 YMAX=24
112 Y=INT(YMAX/2)
115 MODE=3
120 GOSUB 1000
130 XMAX=80
135 X=INT(XMAX/2)
140 YMAX=48
142 Y=INT(YMAX/2)
145 MODE=5
150 GOSUB 1000
160 XMAX=160
165 X=INT(XMAX/2)
170 YMAX=96
```

```
172  Y=INT(YMAX/2)
175  MODE=7
180  GOSUB 1000
900  GOTO 100
998  REM * OUR DRAWING SUBROUTINE
1000 GRAPHICS MODE+16
1010 L=2
1015 PLOT X+L,Y+L
1025 GOSUB 2000
1030 DRAWTO X-L,Y+L
1035 GOSUB 2000
1040 DRAWTO X-L,Y-L
1045 GOSUB 2000
1050 DRAWTO X+L,Y-L
1055 GOSUB 2000
1060 DRAWTO X+L,Y+L
1070 L=L+2
1080 IF L>=YMAX/2 THEN 1100
1090 GOTO 1015
1100 FOR I=1 TO 500:NEXT I
1105 FOR I=1 TO 100
1110 C=INT(RND(0)*3)+1
1120 GOSUB 2000
1130 NEXT I
1150 RETURN
1998 REM * ROUTINE TO VARY COLORS AND BRIGHTNESS
2000 C=INT(RND(0)*3)+1
2005 REGISTER=C-1
2010 IF C=0 THEN REGISTER=4
2015 BRITE=INT(RND(0)*8)+1
2017 IF BRITE/2<>INT(BRITE/2) THEN BRITE=BRITE+1
2020 SETCOLOR REGISTER,INT(RND(0)*15),BRITE
2025 COLOR C
2030 RETURN
```

Program 3-4. Demonstration of graphics modes 3, 5, 7.

Program 3-4 also demonstrates how easily and quickly you can change the colors. The subroutine from lines 2000 to 2030 is used to randomly pick the register, color, and intensity for a SETCOLOR statement. Note also that line 2017 guarantees that the value for the intensity will be an even number. The color is changed as the drawing progresses (lines 1000 to 1090), then there is a pause (line 1100) and then the colors are randomly varied again (lines 1105 to 1130). The use of other graphics modes will be covered in detail in Chapter 11.

...BASIC Keyword Abbreviations

All the BASIC keywords we have learned, as well as those we will be learning, can be typed in abbreviated form. We have already learned that ? can be used to represent PRINT. This is standard in nearly all forms of BASIC. REM can be

shortened to R., or just a period (.). All the other abbreviations are made from the words themselves, using one or more letters, beginning with the first letter of the keyword and then ending with a period. Thus LIST is abbreviated L., COLOR is C., PLOT is PL., POKE is POK. The reason POKE is not just P. is that P. is reserved for POINT (a command we haven't gotten to yet) and PO. invites confusion with POSITION (another command to be discussed later). The most useful abbreviations are the single-letter ones; here is a list of them:

KEYWORD	ABBREVIATION
COLOR	C.
DATA	D.
ENTER	E.
FOR	F.
GOTO	G.
INPUT	I.
LIST	L.
NEXT	N.
OPEN	O.
POINT	P.
PRINT	?
REM	R. or .
SAVE	S.
TRAP	T.
X10	X.

Type in the following line and then LIST it on the screen.

```
100 F.I=1TO1000:N.I
```

Although the original is almost incomprehensible, Atari BASIC provides the full names (and appropriate spaces) when you use the LIST command. The only exception to this is that "?" for PRINT is left alone. So, if your typing is not terrific, abbreviations may speed up line entry and prevent some typographical mistakes by letting BASIC complete the words when you LIST the program. In the immediate mode, L. (LIST) and GR. (GRAPHICS) are probably the most useful of the abbreviations.

59

Chapter 4

Miscellaneous Features and Techniques

Certain calculations and other processes are required so frequently in programming that high-level languages like BASIC supply them in convenient packages. Many of these packages are called *functions*. Some of them are called *operators*. And some are just plain features. These tools are a tremendous convenience in any computer language.

We have already used the INT function in some of our earlier programs in Chapter 2. Remember? INT(X) returns the greatest integer that is less than or equal to X. INT(5.699) = 5 and INT(−4.091) = −5. When we are working with decimal numbers it is often useful to round off results. We will explore some other uses for INT in this chapter.

RND is a package that gives us access to random numbers in a program. We used RND to good advantage in Chapter 2. RND may be used to add interest and variety to games. This function is invaluable for writing simulation programs. We can write a program to model a real-life situation. By changing various factors in a proposed solution to a business problem, we can predict results without imposing poor judgment upon a frustrated public. We may confine our failures to unpublicized runs of a computer program.

These BASIC packages and numerous others will reveal themselves as extremely useful.

4-1 . . . Atari Numeric Functions
ABS, SGN, RND, SQR, and INT

It takes several BASIC statements to determine whether a number is positive, negative, or zero:

```
890 REM * DETERMINE +, 0, OR -
900 IF X>0 THEN S=1
910 IF X=0 THEN S=0
920 IF X<0 THEN S=-1
930 RETURN
```

Once we have written such a subroutine, we should test it. Then, every time we need such a calculation in another program, we must type the entire subroutine. The built-in SGN function determines whether a number is positive, negative, or zero with one statement.

```
130 S=SGN(X)
```

In just the same way we can determine the absolute value of a number in BASIC with the ABS function, which evaluates a number but ignores whether its value is positive or negative. Thus

```
ABS(6.35) = ABS(-6.35) = 6.35
```

and

```
ABS(32-16) = ABS(16-32) = 16
```

Not only are these functions useful in that they save us a lot of programming effort and typing time; they provide some meaning to the statement in which they appear. SGN(X) conveys the idea that we are interested in the sign of the number, while $X = T$:GOSUB 900 fails to convey just why we are invoking the subroutine at line 900 and that the result is returned in S. We would have to read the code beginning at line 900 or put in REM statements to understand the meaning.

The number in parentheses following the function name is called the *argument* of the function. This value is "passed" to the function, and the result is returned in the entire expression. The argument may be a single number (such as 3.25876), a calculated value (such as 3.25*400/16), or an expression involving variables (such as $(X \wedge 3*COST-.75)$, where the values of the variables were evaluated in some prior calculation.

Just as BASIC includes LET, GOSUB, END, IF . . . THEN, and FOR . . . NEXT, it includes features such as INT, RND, SGN, and ABS as elements of the language. This means that the necessary programming has been done for us and incorporated into BASIC. There are many advantages to this. The programming has been tested for us. The features will generally execute much faster than if we write the same calculations in BASIC. This is especially true for trigonometric and logarithmic functions.

For general programming, the most common functions are ABS, SGN, RND, SQR, and INT. Functions that come with the language are sometimes called *built-in* functions.

As discussed earlier, ABS(X) returns the absolute value of X, and SGN(X) returns −1, 0, or +1 according to whether the value of X is negative, zero, or positive. RND is the random-number generator. RND(X) returns random decimal numbers in the range from 0 to 1, including 0 and excluding 1. The variable X is

referred to as a *dummy* variable, since its value does not matter, but a variable must be present anyway.

SQR(X) returns the square root of X. We could also code X∧.5 to represent "X to the one-half power," but SQR is convenient and executes faster. Of course the value of X must not be negative. A negative argument in the SQR function will incur the wrath of BASIC. If we insist on coding a statement such as

```
100 PRINT SQR(-4)
```

we will be subjected to the following message when we RUN the program:

```
ERROR- 3 AT LINE 100
```

which indicates a BAD VALUE error.

Once we gain familiarity with how these functions work, they will come to mind as they are needed when we are thinking about ways to solve computer problems.

Suppose we are interested in finding the factors of integers. A factor of an integer is a number that divides it evenly with no remainder. Right away the INT function should come to mind. We may program the computer to compare INT(N/D) with N/D. If they are equal, then D divides into N with no remainder and D is a factor of N. If INT(N/D) does not equal N/D, then D is not a factor of N. For example:

```
INT(69/5) = 13
```

while

```
69/5 = 13.8
```

Clearly, 13 and 13.8 are not equal, so 5 is not a factor of 69. On the other hand:

```
INT(69/23) = 3
```

and

```
69/3 = 3
```

Twenty-three is a factor of 69 and so is 3 (since 69/3 = 23).

To find the largest factor of 1946, all that we have to do is write a little program that tries all of the values from 1945 down to 2. The first one that is a factor is the largest factor. Display it and terminate the program. While we are at it, we might just as well make this a somewhat general program. Let's make our program request a value for testing. See Program 4-1.

```
90 REM * FIND THE LARGEST FACTOR OF A NUMBER
100 PRINT "FIND LARGEST FACTOR OF";
110 INPUT N
120 FOR DENOM=N-1 TO 2 STEP -1
140 IF N/DENOM<>INT(N/DENOM) THEN 180
150 PRINT DENOM:END
```

```
180 NEXT DENOM
200 PRINT N;" IS A PRIME NUMBER"
```

Program 4-1. Find largest factor.

Note line 140. If we have a divisor that does not go without a remainder, then we perform the next test. If not, then we have the largest factor. Display it and quit.

```
RUN
FIND LARGEST FACTOR OF?1946
973

READY
```

Figure 4-1. Execution of Program 4-1.

There is something about this program that may not be obvious unless we witness the execution. The computer has to work for more than 20 seconds before producing the answer for N = 1946. And it would delay for more than 40 seconds for N = 1949, which is a prime number (that is, it has no factors other than 1 and itself). The smaller the first factor, the longer the delay. Surely we could find the largest factor of 1946 more easily than this.

Decimal division on a computer takes time. We could save one division for each value of N by assigning N/DENOM to an intermediate variable:

```
135 Q=N/DENOM
140 IF Q<>INT(Q) THEN 180
```

The time saving is about 10%. While this might be worth doing, we should also carefully examine the method we have chosen for solving this problem.

Take the case of 1946. The largest factor is 973, and the smallest factor is 2. We could simply test our factors beginning with 2. When we have found the smallest factor, the largest factor may be found by division. Thus we have gone from 973 trial values in the FOR . . . NEXT loop of Program 4-1 to a single trial for this particular value of N. We have also gone from 20 seconds to a small fraction of one second. That is an improvement worth working on. What if we enter 1949? This new method will require 1947 trial values of DENOM and just over 40 seconds to execute. So, this method only helps for values of N that have factors. We should continue asking questions and making observations that may lead to an improved method that also works for prime integers.

Let's return to the observation that the largest factor of 1946 is 973 and the smallest is 2. How are the rest of the factors paired? Take a look at this list:

2	973
7	278
14	139
139	14
278	7
973	2

There are six pairs of factors. Each pair appears twice. How can we determine when we have found all of the unique pairs of factors? For every factor less than or equal to the square root of a number, the other factor will be greater than or equal to the square root. This can be shown mathematically, but we will take it on faith. Once we are convinced of that, the rest is easy. We need only test divisors up to the square root. Simply change

```
120 FOR DENOM=N-1 TO 2 STEP-1
```

to

```
120 FOR DENOM=2 TO SQR(N)
```

and change

```
150 PRINT DENOM:END
```

to

```
150 PRINT N/DENOM:END
```

This change in strategy reduces the number of tests for N = 1949 from 1948 to 43. That is significant and worth incorporating into our program. We can also use the intermediate variable Q to store N/DENOM. Thus:

```
150 PRINT N/DENOM:END
```

becomes

```
150 PRINT Q:END
```

See lines 120, 135, 140, and 150 of Program 4-2.

```
90 REM * FIND THE LARGEST FACTOR OF A NUMBER
USING SQR(N)
100 PRINT "FIND LARGEST FACTOR OF";
110 INPUT N
120 FOR DENOM=2 TO SQR(N)
135 Q=N/DENOM
140 IF Q<>INT(Q) THEN 180
150 PRINT Q:END
180 NEXT DENOM
200 PRINT N;" IS A PRIME NUMBER"
```

Program 4-2. Find largest factor using SQR (N).

...Rounding Decimal Results

Another use for INT comes up when we work with dollars and cents where calculations come out in fractional cents. We would like always to round figures off to the nearest cent for printing. Anything that is .5 cents or more is "rounded up," and anything less than .5 cents is "rounded down."

We can convert dollars and cents to cents by multiplying by 100. Then if we add .5 cents, all values from .0 to .49 will become values in the range from .5 to .99, while all values in the range from .50 to .99 will become values in the range from 1.0 to

1.49. If we next apply INT, all decimal portions that were less than .5 disappear, and all values that were .5 or more result in one cent being added. Then we get from cents back to dollars and cents by dividing by 100. Thus we can round values to the nearest cent with a statement such as

```
200 NUMBER1=INT(NUMBER*100+.5)/100
```

If we follow through what this formula does, we'll get an added feel for how it works. Let's assume NUMBER = 345.58679, so that when we multiply by 100 we get 34558.679. Adding 0.5 gives 34559.179. When we take the INT of this value we get 34559; dividing by 100 gives 345.59, which leaves us with two decimal places as desired and correctly rounds off what we have eliminated. If we wanted to round off to three decimal places, we would carry out exactly the same procedure, only we would multiply by 1000 and later divide by 1000.

We can easily write a little test program to verify our solution for rounding values to the nearest cent (and incidentally for rounding any values to the nearest hundredth). See Program 4-3.

```
100  REM * DEMONSTRATE ROUNDING
140  PRINT "DATA","ROUNDED VALUE"
150  READ NUMBER
160  IF NUMBER=-9999 THEN END
200  NUMBER1=INT(NUMBER*100+0.5)/100
210  PRINT NUMBER,NUMBER1
220  GOTO 150
900  DATA 3.09123,4.94561
910  DATA 2390,-1.5102
920  DATA .0009,-1.4861
990  DATA -9999
```

Program 4-3. Rounding to the nearest hundredth.

We have included the labeling of line 140 to give the display some meaning.

```
RUN
DATA          ROUNDED VALUE
3.09123       3.09
4.94561       4.95
2390          2390
-1.5102       -1.51
9E-04         0
-1.4861       -1.49

READY
```

Figure 4-2. Execution of Program 4-3.

Note that this approach also handles negative values correctly. It is always a good idea to verify that our programs work properly for a wide variety of values. Even though the current problem doesn't require a particular class of values, it is desirable to test the program for them anyway. It is much easier to put the finishing

touches on a routine while we are familiar with the problem than to return to it months later when we discover that we really do want to handle those previously unwanted values.

We can also use rounding to get around the problem seen earlier where some versions of Atari BASIC do not produce integer values in cases where we know they ought to. Exponentiation is the best known case. We know that $2 \wedge 2$ is $2 \times 2 = 4$ and that $2 \wedge 3$ is $2 \times 2 \times 2 = 8$. We also know that Atari BASIC may give us 3.99999996 or 8.00000001 instead. We avoid this problem by using rounding.

```
100 Y=INT(2^2+.05)
```

or

```
100 Y=INT(2^3+.05)
```

...Compound Interest

Suppose we have $100 in a savings account at 5.5% interest, compounded daily. How much will that be at the end of one year? We can easily write a little program to calculate that. There is a formula that gives compound amounts very nicely:

$$A = P (1 + I)^N$$

where

A = Amount
P = Principal
I = Interest rate per interest period
N = Number of interest periods

The raised N in the formula above indicates "to the power." Program 4-4 shows the above formula in BASIC.

```
100 REM * CALCULATE DAILY COMPOUNDED INTEREST
200 PRINCIPAL=100
210 INTEREST=0.055/365
220 PERIODS=365
300 AMOUNT=PRINCIPAL*(1+INTEREST)^PERIODS
310 PRINT AMOUNT
```

Program 4-4. Compound interest by formula.

Remember that "∧" is used as the symbol for "to the power" on the Atari. This symbol is generated by using SHIFT + "*" on the keyboard.

```
RUN
105.653819
```

```
READY
```

Figure 4-3. Execution of Program 4-4.

Now, since we have enough trouble buying anything with a whole cent, let alone

.3819 cents, we might as well round that value off to the nearest cent. We can do that easily by replacing line 310 with

```
310 PRINT INT(AMOUNT*100+.5)/100
```

This program tells us what our amount will be at the end of the year. What the program doesn't tell us is what has happened to the buying power of our money due to inflation. It doesn't tell us of the federal, state, and even city income taxes we may have to pay on the interest. However, a savings account is still better than hiding the money in a mattress.

That compound interest formula works just fine if we are going to put $100 in the bank and leave it there. But suppose we decide to put $20 into the account on the first of each month. For simplicity, let's consider that each month has 30 days and that the year has 360 days. Let's put $100 in the bank on January 1 and then put $20 in on the first of the month each month all year. We can handle this nicely with a FOR . . . NEXT loop going from 1 to 12. See Program 4-5.

```
100 REM * ADD $20 EACH MONTH
200 PRINCIPAL=100
210 INTEREST=0.055/360
220 PERIODS=30
300 FOR MONTH=1 TO 12
310 PRINCIPAL=PRINCIPAL+20
320 AMOUNT=PRINCIPAL*(1+INTEREST)^PERIODS
330 PRINCIPAL=AMOUNT
340 NEXT MONTH
350 PRINT "$100 PLUS $20 EACH MONTH $ ";
360 PRINT INT(AMOUNT*100+0.5)/100
```

Program 4-5. Compound interest with money added each month.

Note that in line 330, the amount at the end of each month becomes the principal for the next month.

```
RUN
$100 PLUS $20 EACH MONTH $ 352.94

READY
```

Figure 4-4. Execution of Program 4-5.

...SUMMARY
ABS, SGN, RND, SQR, and INT are commonly used built-in functions available in BASIC. ABS(X) returns the absolute value of X. SGN(X) returns -1, 0 or $+1$ according to whether X is negative, zero, or positive. RND(X) returns a random number in the range 0 to .9999999999 and INT(X) is very useful in rounding off numbers.

Problems for Section 4-1 . . . .

1. Write a program to find all prime factors of an integer by rewriting the guts of Program 4-2 as a subroutine and calling it repeatedly. Eliminate duplications.

2. Write a program to compare the effect of considering the banking year to have 360 days instead of the 365 on the real calendar. Use 5.5% and 12.5% interest on a principal of $100,000.

3. Compare daily compounding with monthly compounding for $1000 at 5.5% and 12.5% interest for one year.

4. Write a program that asks the user for the interest rate, principal, and interest periods per year as input and calculates the result. This program can be used for lots of "what if" calculations merely by changing the values used.

5. Compound interest may also be calculated without the formula given in this section. We may simply build a loop that adds the interest at the effective interest rate once for each period in the time that the money is on deposit. Write a program to calculate interest this way and compare your results with those in the programs of this section. Compare a 365-day year with a 360-day year.

4-2 . . . Some Special Features

There are lots of features that we could get by without. In fact, many versions of BASIC do not include some of the packages we will be opening in this section. These features do, however, make life interesting and programming easier.

. . . POSITION

This moves the cursor to any desired location on the screen. It has the form

```
100 POSITION ROW,COLUMN
```

The cursor will not move to this position until you actually access the screen (for example, with a PRINT statement). You can use this command to neatly arrange your screen displays, although this is only one of several ways to accomplish this.

. . . TAB

There are other possibilities for positioning the cursor before printing on the screen. We have already used commas in print statements and seen how they move the cursor. You can also use the TAB key to position the cursor. Pressing the TAB key in immediate mode will move the cursor across the screen, stopping at columns 7, 15, 23 and every eight columns after that to the end of the logical line (which is three

screen lines). You can use the same capability in a program by pressing ESC and then the TAB key, giving a "➤" symbol on the screen. Each time this is done, the cursor moves to the next tab stop. Just as with a typewriter, you may also delete tab stops or add new ones. Tabs are deleted from particular positions by the key sequence CTRL and TAB. They can be added anywhere along a row with SHIFT and TAB. Preceding either of these with ESC permits their use in a BASIC program. You will see a strange assortment of arrows and arrow-like symbols on the screen as a result of these sequences. Setting tabs can be useful for lining up columns of information on the screen.

If you just want to locate a few things on the screen, it is easier to use POSITION or the following POKE:

```
100 POKE 85,COLUMN
```

where COLUMN is the column where you want the printing to start (in the row where the cursor is currently located). Some other versions of BASIC have a TAB statement that merely performs the same function as the above POKE.

...FRE

FRE(X) is a function that returns the amount of free memory in bytes. The value of X doesn't matter, but a value of 8 or 9 is handy because these numbers are near the parenthesis keys on the keyboard. A byte corresponds to a single character in memory. Your Atari will have a minimum of 16K (16384) bytes of memory, although not all of this will be available to your programs. You need to consider the memory available to you when you are working on very large programs or on programs that use large arrays or very long strings (both of which will be discussed in upcoming chapters). When you are checking the amount of memory available after writing part of a program, be sure to RUN the program first so that all statements that use or reserve memory have been executed.

...PADDLE and STICK

If you have an Atari 400 or 800, you have four ports on the front of the computer. If you have any other model of Atari computer you have two such ports on the side. These ports are designed to accept input from paddles or joysticks. From left to right the ports are numbered 0, 1, 2, and 3 (or 0 and 1 if there are only two). A pair of paddles or one joystick can use each port. Thus, PADDLE 0 and PADDLE 1 will refer to the paddles plugged into the leftmost port and STICK 1 refers to the joystick in the second port. Both paddles and joysticks can be used to input values to a program.

The PADDLE function returns a value from 1 to 228, depending on the rotation of the paddle. All the way clockwise gives a 1 and all the way counterclockwise gives 228. These numbers are measurements of the electrical resistance of a potentiometer (variable resistor) that is not quite calibrated to match the full rotation, so 228 will begin to appear well before the knob is turned fully counterclockwise. From fully clockwise to this point the values will vary pretty

uniformly. The paddle also has a button that is read by the PTRIG function. This returns a 1 if the button is not pushed and a 0 if it is. Here's a simple program that reads the paddle in port 0 and displays the value it has read.

```
5 REM * USING PADDLES FOR INPUT
10 PRINT "J":REM * CLEAR SCREEN
15 POKE 752,1
20 POSITION 2,10
30 PRINT "THIS PROGRAM ACCEPTS INPUT FROM THE"
35 PRINT "PADDLE AND DISPLAYS THE VALUE."
40 PRINT "PUT THE PADDLE IN PORT 0 AND USE THE"
45 PRINT "BREAK KEY TO STOP."
48 FOR DELAY=1 TO 1500:NEXT DELAY
50 PRINT
55 X=PADDLE(0)
60 PRINT "PADDLE VALUE IS ";X;
70 IF PTRIG(0)<>0 THEN PRINT :GOTO 80
75 PRINT " AND Trigger PUSHED"
80 GOTO 55
```

Program 4-6. Using paddles for input.

Notice the POKE 752,1 in line 15. This will make the cursor invisible and is used to produce a screen display with fewer distractions on it. POKE 752,0 will make the cursor visible again. When you try this program you will notice how sensitive the value printed is to the position of the paddle. The paddle is not a good device to use if you are looking for small precise changes in values, unless you put in delays to give the user time to carefully select the desired value or use other tricks to ensure a more stable response.

As an aside, note that we have used ESC SHIFT and CLEAR (" **J** ") to clear the screen.

The joystick is used more often than the paddle as an input method, especially in games. Its use, however, can go far beyond that. The position of the joystick is requested with

100 DIRECTION=STICK(0)

which looks much like the paddle statement. The value returned will be one of nine values, eight for eight different directions and one for straight up (that is, no direction). Figure 4-5 shows the values returned to the computer for all of these possible directions.

Here's a program that checks on the joystick position and tells us which direction it is leaning towards.

```
5 REM * USING THE JOYSTICK FOR INPUT
10 PRINT "J":REM * CLEAR SCREEN WITH ESC CTRL+
CLEAR
15 POKE 752,1
20 POSITION 2,10
30 PRINT "THIS PROGRAM ACCEPTS INPUT FROM THE"
35 PRINT "JOYSTICK AND DISPLAYS THE VALUE."
```

```
40 PRINT "PUT THE JOYSTICK IN PORT Ø AND USE"
42 PRINT "THE BREAK KEY TO STOP."
45 FOR DELAY=1 TO 1500:NEXT DELAY
50 PRINT
55 DIRECTION=STICK(Ø)
60 PRINT "THE VALUE IS ";DIRECTION;" WHICH IS ";
70 GOSUB 500
80 GOTO 55
499 REM DIRECTIONS SUBROUTINE
500 IF DIRECTION=14 THEN PRINT "NORTH"
510 IF DIRECTION=10 THEN PRINT "NORTHWEST"
520 IF DIRECTION=6 THEN PRINT "NORTHEAST"
530 IF DIRECTION=9 THEN PRINT "SOUTHWEST"
540 IF DIRECTION=5 THEN PRINT "SOUTHEAST"
550 IF DIRECTION=7 THEN PRINT "EAST"
560 IF DIRECTION=11 THEN PRINT "WEST"
570 IF DIRECTION=13 THEN PRINT "SOUTH"
580 IF DIRECTION=15 THEN PRINT "UPRIGHT"
590 RETURN
```

Program 4-7. Using the joystick for input.
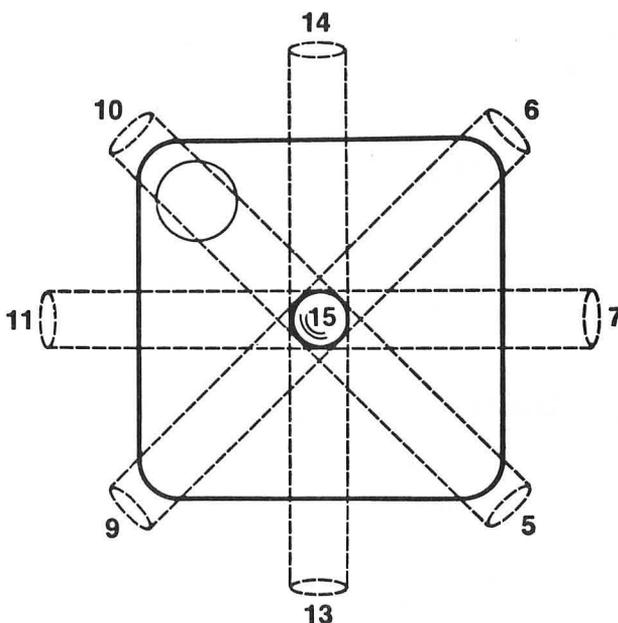
Figure 4-5. Joystick values.

The subroutine at line 500 uses the direction numbers returned to assign a name to the value, which is then printed along with a number. The possibilities for joystick use are limited only by your imagination. For example, the joystick can be used by

disabled persons to select letters and numbers that are on the keyboard without pressing the keys.

The joystick also has a button, which is read with

```
200 BUTTON=STRIG(Ø)
```

where STRIG(0) refers, as expected, to the leftmost port; STRIG is short for stick trigger. This can also be incorporated in a program. Here's a simple drawing program using graphics mode 3, where the joystick controls the "paint brush" and the button controls whether we draw or erase. (Erase merely means draw with the background color.)

```
10  GRAPHICS 3
20  S=STICK(Ø):T=STRIG(Ø)
25  IF T=Ø THEN COLOR Ø
27  IF T=1 THEN COLOR 1
30  IF S=7 THEN X=X+1
40  IF S=11 THEN X=X-1
50  IF S=14 THEN Y=Y-1
60  IF S=13 THEN Y=Y+1
70  IF X>39 THEN X=39
80  IF X<Ø THEN X=Ø
90  IF Y>19 THEN Y=19
100 IF Y<Ø THEN Y=Ø
110 PLOT X,Y
115 FOR I=1 TO 25:NEXT I
120 GOTO 20
```

Program 4-8. A GRAPHICS 3 drawing program.

There are many elaborations that can be added to a program like this, one of the most obvious being some indication of where the "brush" is when we are in the erase mode. Also you might try using the button to cycle through the colors, so you can draw with three colors and erase with the fourth.

4-3...Other Atari BASIC Functions

SIN(X) and COS(X) will return the trigonometric values we would expect. When you turn on your computer, or after a NEW or RUN command, your Atari will assume that you are working in radians, where π radians = 3.1416 radians = 180 degrees. To work in degrees, simply use DEG. You will then have all the computations in degrees. Go back to radians with RAD. You can derive all the other regular trig functions from SIN and COS. You also have the inverse tangent function, ATN(X), which works in degrees or radians just like SIN(X) or COS(X). It remains for the programmer to determine the correct quadrant where that is a problem.

There are a few more functions available for mathematical use. EXP(X) raises e (2.71828183) to the Xth power, LOG(X) returns the natural (base e) logarithm of X, and CLOG(X) returns the common (base 10) logarithm of X.

4-4...Logical Operators in BASIC

...AND, OR, and NOT

Often in a program there are several conditions that may determine the next course of action. We might want to execute a subroutine if AVERAGE > 95 and SCORE < 70. We can do this with AND.

```
300 IF AVERAGE>95 AND SCORE<70 THEN GOSUB 900
```

will do the job. AND is one of the three *logical operators* in BASIC. BASIC evaluates the expression "AVERAGE > 95". If that expression is true, BASIC sets its value to 1. If that expression is false, BASIC sets its value to 0. The same goes for "SCORE < 70". We can even assign logical values to variables.

```
290 L1=AVERAGE>95:L2=SCORE<70

300 IF L1 AND L2 THEN GOSUB 900
```

This is equivalent to the single statement 300 above. In line 290, the value of L1 is set to 1 if AVERAGE > 95 is true and 0 is AVERAGE > 95 is false. Similarly, L2 becomes 1 or 0. And finally, in line 300, L1 AND L2 becomes 1 or 0. We can even assign L1 AND L2 to another variable if that suits our purpose.

PROGRAMMER'S CORNER 4

...The Attract Mode

If we use our joystick or paddle routines for long periods of time, we will notice that all of a sudden the screen starts to randomly change colors on its own. There is a timer at memory location 77 that does this. This is a carry over from arcade games, where the flashing colors will attract attention (hence the name attract mode). With game machines, the concern was that if a display with fixed colors was left on the screen for long periods of time, then there might be damage to the television screen. With the computer, if there is no keyboard input for nine minutes, the attract mode begins. You can prevent this from happening by using

```
POKE 77,0
```

in a line of your program that is regularly executed. If you want to force the computer into the attract mode, just use POKE to place the value 255 into this location.

...More Error TRAPping

We have used TRAP to catch errors on INPUT by redirecting the program to a line that displays a message and then goes back and requests input again. We can actually check what the error was and take action depending on the nature of the error. For example, the error may be that the cursor is out of range for the graphics mode that you're in, or an input is illegal, or a calculated value is invalid, or any

number of other reasons. Memory location 195 stores the error number and memory locations 186 and 187 together hold the line number where an error occurred. You can find the error number with

```
100 ERRORNUM=PEEK(186)
```

and the line number with

```
110 LINENUM=PEEK(186)+PEEK(187)*256
```

It is good practice to use this to check that the error encountered is indeed the error you were protecting against and not some other one. For example, you may have a TRAP on an INPUT statement so that letters are not input when numbers are required. The error that is expected if your TRAP is tripped is ERROR −8; you can check this with PEEK (195) to see if an 8 is there and, if it is, print your BAD INPUT message.

A word of warning about the use of TRAP statements. When you first write a program, it is best to either use the above to check what the error is, or else leave out all TRAP statements until the program is up and running with no obvious errors. The reason for this is that when you are debugging a program, you want to see what the errors are so you can correct them. A simple TRAP of input will send any error that occurs to your subroutine, print out the warning message, and then go back for a new input, without giving you a clue as to what actually went wrong.

...Neater INPUT

There will be many times when there is a maximum length for the INPUT that is being requested from the user. We can display on the screen what this maximum is by showing a series of, say, dashes on the screen and then positioning the cursor so that the input occurs right above these marks. We reposition the cursor (which is really just the point where the next character will be printed) in a program by using CTRL and the arrow keys to move around the screen and preceding these commands with ESC. For example, Program 4-9 requests INPUT and shows you how many characters you may use.

```
90 REM * INPUT WITH UNDERLINING
100 PRINT "Input number of widgets (<1000000)"
110 PRINT
120 PRINT " ────────":REM 7 CTRL+M
130 PRINT "↑↑↑":REM 3 ESC CTRL+UP ARROW
140 INPUT WIDGETS
150 IF WIDGETS>9999999 THEN PRINT "TOO BIG, TRY
AGAIN."
160 PRINT
170 GOTO 100
```

Program 4-9. Positioning the cursor for neater input.

In line 120, CTRL and M is used to make our underline mark; then the up arrow in line 130 puts the cursor at the start of this line. Note that we allowed a space in line 120 for the question mark (?) that comes along with the INPUT statement.

Chapter 5

Character Strings and String Functions

Most of our work has used numbers and calculations. However, we have printed messages and labels by enclosing them in quotation marks in PRINT statements. The ability to handle nonnumeric data is important in working with computers. Such data are referred to as *string data*. String data may contain any of the letters, digits, and special characters available on the computer. Thus, string data comes in character strings.

Strings may be used for a name-and-address mailing list, for instructions telling how to use a computer program, as labels to make the displayed results more understandable, or as part-identification labels in an inventory-control system. We might simply use strings to make a game program more conversational. We can ask the player's name and use it later in displayed messages. BASIC provides a variety of features that make the handling of string data very convenient. There are string variables, which enable us to store and manipulate character strings. Using string variables and string functions, we can manipulate individual characters and groups of characters. We can even print a string in reverse order just for fun.

5-1...Atari BASIC Strings

Atari BASIC provides string variables and several useful string manipulation functions. A string variable is distinguished from a numeric one by using a dollar sign ($) as the last character in the variable name. We may work with string variables in many of the ways in which we work with numeric variables. For instance, any of the following statements may appear in a program:

```
100 LET VALUE$="FIRST"

100 READ A$

100 INPUT NAMEOFITEM$

110 PRINT FILE$
```

String variable names may be as long as one program line, 114 to 120 characters. String variables themselves may contain any number of characters, up to the limit of the memory that is available in the computer. The most important consideration in handling string variables in Atari BASIC is that you *must* dimension the string before a value is assigned to it, using a statement such as

```
100 DIM VALUE$(10)
```

or

```
100 DIM VALUE$(10),A$(250),NAME$(25),FILE$(2500)
```

The second example reserves space for four strings of different lengths. Always dimension your string variables with a value that is at least as large as the longest string you expect to give that name. If we had used the statement above to DIMension VALUE$, and later had a statement

```
200 VALUE$="YOUR MONEY'S WORTH"
```

the actual VALUE$ string stored in memory would only consist of the first ten characters. What you will be doing with DIM statements is setting aside a place in memory to hold the string when you get around to giving it a value in your program.

Now, in order to execute READ A$ we must, of course, provide a corresponding DATA statement, just as with numeric variables. If we want to include a comma in the string, then we should enclose the string in quotation marks. Without the use of quotation marks, the comma is interpreted as the end of the current DATA item. Program 5-1 READs string DATA and PRINTs it for us to see.

```
100 DIM A$(30)
105 READ A$
120 PRINT A$
130 GOTO 105
495 REM
500 DATA George M. Cohan,Abe Lincoln
510 DATA Joan of Arc
```

Program 5-1. READ . . . DATA with strings.

The comma in line 500 is interpreted by Atari BASIC as a data separator or delimiter. We could have provided the same data for this program by typing as follows:

```
500 DATA George M. Cohan,Abe Lincoln,J
oan of Arc
```

Notice that the computer breaks up the DATA statement after the "J" in Joan and pushes the remaining characters to the next line as we type.

For short data items we could avoid having unnatural breaks in DATA statements by placing each data item on a single line. Doing this will take up additional memory. However, we are writing very short programs that don't require much memory. So, we won't worry about memory use until we are writing very long programs. The most readable form follows:

```
100 DIM A$(30)
105 READ A$
120 PRINT A$
130 GOTO 105
495 REM
500 DATA George M. Cohan
510 DATA Abe Lincoln
520 DATA Joan of Arc
```

Program 5-2. Program 5-1 with reformatted DATA.

It is always worth a little effort to make programs more readable. As we gain experience with programming, this will come automatically.

```
RUN
George M. Cohan
Abe Lincoln
Joan of Arc

ERROR-    6 AT LINE 105
```

Figure 5-1. Execution of Program 5-2.

The error message (which is an OUT OF DATA error) is a little disturbing. Good programs will never produce that message! In some situations, programs that end with an error message will fail to perform as desired. We should always provide for an orderly program termination. In this case we may simply add an artificial string-data item to the data list. Such a data item is sometimes called *dummy data*. We will use this artificial data item as a signal to the program that all the data have been read. After line 100 and before line 120 we compare A$ to the signal data. When "STOP" is used as the terminating signal the final program looks like Program 5-3.

```
100 DIM A$(30)
105 READ A$
110 IF A$="STOP" THEN 900
120 PRINT A$
130 GOTO 105
495 REM
500 DATA George M. Cohan
510 DATA Abe Lincoln
```

```
520 DATA Joan of Arc
599 DATA STOP
900 END
```

Program 5-3. Using dummy data to terminate program execution.

Now our little demonstration program terminates in an orderly way. Of course, the actual signal is arbitrary; all that matters is that we select some value that will not be a real DATA item and test for that value.

Atari BASIC permits us to compare strings and put them in order in much the same way we compare numbers using IF . . . THEN statements. The sequence used is known as ATASCII, Atari's version of ASCII (American Standard Code for Information Interchange). For numbers and letters ATASCII and ASCII are the same, but some of Atari's special characters differ from their ASCII equivalents or have no ASCII equivalent. Numbers run from 0 to 9, followed by letters in alphabetical order, uppercase before lowercase and regular printing before inverse printing. We can easily write a short program to demonstrate comparisons of strings.

```
90 REM * COMPARE STRINGS FOR ORDER
95 DIM A$(30),B$(30)
100 PRINT
110 PRINT "INPUT A$";
120 INPUT A$
130 IF A$="STOP" THEN 240
140 PRINT "INPUT B$";
150 INPUT B$
160 IF A$<B$ THEN 220
170 IF A$=B$ THEN 200
175 REM
180 PRINT A$;" IS GREATER THAN ";B$
190 GOTO 100
195 REM
200 PRINT A$;" IS EQUAL TO ";B$
210 GOTO 100
215 REM
220 PRINT A$;" IS LESS THAN ";B$
230 GOTO 100
235 REM
240 END
```

Program 5-4. String comparison in Atari BASIC.

If you play around with this program a while, you will soon discover that strings are compared starting with the leftmost character and proceeding to the right until a difference is found. Then the string with the larger ATASCII value is declared to be greater. If all the characters of one string match another string, but the second string still has characters left over, then the second string is considered to be greater. Here are a few results from a RUN of Program 5-4.

```
RUN

INPUT A$?WHAT'S THIS
INPUT B$?WHAT'S THAT
WHAT'S THIS IS GREATER THAN WHAT'S THAT

INPUT A$?WHAT'S THIS
INPUT B$?WHAT'S WHAT
WHAT'S THIS IS LESS THAN WHAT'S WHAT

INPUT A$?WHAT'S WHAT
INPUT B$?WHAT'S WHAT
WHAT'S WHAT IS EQUAL TO WHAT'S WHAT

INPUT A$?STOP

READY
```

Figure 5-2. Execution of Program 5-4.

All of the comparison operators for numeric comparisons are available for string comparisons.

We can use Atari BASIC to access groups of characters or individual characters in a string by using subscripts. For example, if

```
DAY$="SUNMONTUEWEDTHUFRISAT"
```

then we can display "SUN" with the statement

```
200 PRINT DAY$(1,3)
```

Likewise, we could use PRINT DAYS$(19,21) to display "SAT". DAYS$(X,Y) defines all the characters from the Xth position through the Yth position in the string. Program 5-5 displays the names of the days of the week.

```
80 REM * DISPLAY THE DAYS OF THE WEEK
100 DIM DAYS$(21)
120 DAYS$="SUNMONTUEWEDTHUFRISAT"
140 FOR K=1 TO 7
150 J9=3*K-2
160 PRINT K,DAYS$(J9,J9+2)
190 NEXT K
200 END
```

Program 5-5. Display the days of the week.

Look carefully at line 150. The idea here is to start at positions $1, 4, \ldots, 19$, since the name of each day is three characters in length. On day 1 we get $J9 = 3 * 1 - 2 = 1$, and we display characters from there to $J9 + 2 = 1 + 2 = 3$. On day 7 we get $3 * 7 - 2$, which is 19, and we display the characters in positions 19 through 21.

```
RUN
1          SUN
2          MON
3          TUE
4          WED
5          THU
6          FRI
7          SAT

READY
```

Figure 5-3. Execution of Program 5-5.

Clearly, if DAY$(4,6) calls for characters 4,5, and 6, then DAYS$(7,7) calls for the seventh character. Likewise, DAYS$(J,J) calls for the Jth character. Now we can access the characters of strings individually. The reverse procedure may also be carried out. DAYS$(J,J) = "A" assigns the letter A as the Jth character in the string DAYS$ and DAYS$(5,7) = "ABC" assigns ABC as the fifth, sixth, and seventh characters, respectively.

...Single Subscript

If we leave out the second subscript, then something quite different happens. Examine Program 5-6, which displays A$(K) for various values of K in line 130.

```
80 REM * DEMONSTRATE THE USE OF A SINGLE SUBSC
RIPT IN A STRING
100 DIM A$(25)
110 INPUT A$
120 FOR K=1 TO 25
130 PRINT A$(K)
140 NEXT K
150 END
```

Program 5-6. Single string subscript.

We can display a substring starting from a particular character of the string to the end by using a single subscript. A$(1) calls for the entire string. A$(4) calls for the substring beginning with the fourth character and extending to the end. Thus as the value of K increases by one for each step of the FOR . . . NEXT loop we get one less character at the beginning of the string.

```
RUN
?HERE WE GO
HERE WE GO
ERE WE GO
RE WE GO
E WE GO
 WE GO
WE GO
```

```
E GO
  GO
GO
O

ERROR-    5 AT LINE 13Ø
```

Figure 5-4. Execution of Program 5-6.

The error message in the run of Program 5-6 indicates that we have attempted to access characters beyond the length of A$.

Displaying A$(K) works well until we specify a value of K "off the end" of the string. In Program 5-6 above, calling for the display of A$(11) when A$ contained only ten characters caused this STRING LENGTH error. We can easily avoid this by using the LEN( ) function to measure the number of characters in a string.

...The LEN( ) Function

LEN(A$) measures the number of characters actually stored in the string variable A$. Even though we may have dimensioned A$ to 250, the LEN( ) function will count only the number of characters that are really stored there. So, in our little program above, instead of having the FOR . . . NEXT loop in line 120 go from 1 to 25, we should code the following line:

```
12Ø FOR K=1 TO LEN(A$)
```

It is never good to allow a program to terminate in an error condition. This can cause other serious problems. Knowing the number of characters stored in a string variable enables us to tell the program when to stop.

...String Comparison

BASIC allows us to compare strings for equality by using IF . . . THEN statements. The equals sign (=) is used to test for equality, and less than/greater than symbols (< >) are used to test for inequality. Each of the following statements is valid:

```
1ØØ IF A$=B$ THEN 135
1ØØ IF A$<>B$ THEN 24Ø
1ØØ IF A$(2,4)=A$(7,9) THEN END
1ØØ IF T$(1,3)=S$(K,K+2) THEN 2ØØ
```

Using what we know at this point, we can display the characters of a string in alphabetical order. Suppose we enter the alphabet as a string constant so that we may compare each letter of the alphabet with each of the letters of some string entered from the keyboard during a program RUN. First we will look for A's, then for B's, and so on until we have looked for Z's. Each time we find that the current letter in the entered string does not match the current letter of the alphabet, we skip

81

to the next letter in the entered string. Each time we find a match, we print the matched character.

```
90 REM * OUR 1ST PROGRAM TO ALPHABETIZE
CHARACTERS OF A STRING
100 DIM ALPHA$(26),B$(40)
110 ALPHA$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 PRINT "ENTER YOUR STRING:"
130 INPUT B$
190 REM * WE ALPHABETIZE BY USING A SAMPLE
ALPHABET
200 FOR K=1 TO 25
210 FOR J=1 TO LEN(B$)
220 IF ALPHA$(K,K)<>B$(J,J) THEN 240
230 PRINT B$(J,J);
240 NEXT J
250 NEXT K
900 END
```

Program 5-7. Alphabetizing in Atari BASIC.

The decision to display the current character is made in line 220. The actual display occurs in line 230. We might code lines 220 and 230 in the following single line:

```
220 IF ALPHA$(K,K)<>B$(J,J) THEN PRINT B$(J,J);
```

```
RUN
ENTER YOUR STRING:
?BIRTHDAY
ABDHIRTY

READY
```

Figure 5-5. Execution of Program 5-7.

Our little program seems to have done its job. But what will happen if we enter characters that are not letters of the alphabet? Let's try it.

```
RUN
ENTER YOUR STRING:
?WHAT IS GOING ON HERE?
AEEGGHHIINNOORSTW

READY
```

Figure 5-6. Another execution of Program 5-7.

We can see that any characters not included in ALPHA$ are simply ignored. In many situations that is exactly what we would want.

...Concatenation

There will be times when we will want to construct one string from another

string or strings. For instance, we might have a situation where someone's last name is stored in LAST$ and the first name in FIRST$ and we want a new string, NAME$, to contain the entire name, first name first. It is a simple matter to set NAME$ equal to FIRST$ with

```
40 NAME$=FIRST$
```

Now, how do we insert a space between the names? We use a statement of the following form:

```
150 NAME$(LEN(NAME$)+1)=" "
```

Think of line 150 as saying the following: "Find the length of the string called NAME$. Then move one character further and, beginning at this point, append whatever follows the equals sign onto NAME$." In a similar fashion, after we add the space, we append the last name with

```
160 NAME$(LEN(NAME$)+1)=LAST$
```

The one thing to look out for is that the string NAME$ must be dimensioned large enough to accommodate all of the characters in FIRST$, LAST$, and the space added in line 150. If you should fail to provide for this, Atari BASIC will do one of two things. If there is room for more characters in NAME$, it will append as many as will fit and ignore all extras beyond that. If NAME$ is full, you'll get ERROR −9 (ARRAY or STRING DIM error). So simply make sure that your strings are adequately dimensioned for the job you intend to do. The process of adding characters to a string is called *concatenation*.

Suppose we have a name in NAME$ as described above. That is, NAME$ = "JOHN JONES". What would it take to write a program to create a new string containing the name, last name first, followed by a comma, a space, and the first name? All we have to do is find the space with a loop. Once we have found the space, it is a simple matter to rearrange the parts of the string in the desired order. Consider Program 5-8.

```
90 REM * REARRANGE NAME FROM FIRST NAME
FIRST TO LAST NAME FIRST
100 DIM NAME$(30),X$(31)
110 PRINT
120 PRINT "NAME- FIRST NAME FIRST"
130 PRINT "ENTER HERE";:INPUT NAME$
135 IF NAME$="STOP" THEN 900
200 FOR I=1 TO LEN(NAME$)
210 IF NAME$(I,I)=" " THEN 300
220 NEXT I
230 PRINT "ENTER A SPACE BETWEEN NAMES"
240 GOTO 110
290 REM
300 X$=NAME$(I+1)
310 X$(LEN(X$)+1)=", "
320 X$(LEN(X$)+1)=NAME$(1,I-1)
```

```
400 PRINT X$
410 GOTO 110
900 END
```

Program 5-8. Rearranging names with BASIC strings.

By dimensioning X$ to one character more than NAME$ we guarantee enough space. Line 210 tests for the first space in NAME$. Then lines 300 through 320 rearrange the name string. See the execution of Program 5-8.

```
RUN

NAME- FIRST NAME FIRST
ENTER HERE?JOHN JONES
JONES, JOHN

NAME- FIRST NAME FIRST
ENTER HERE?STOP

READY
```

Figure 5-7. Execution of Program 5-8.

There are a couple of things that could be done to improve our program. Suppose there is more than one space in the entered name. The program should reject it. Suppose someone enters last name first with a comma. The program should reject that also. It is left as an exercise to make these improvements.

...SUMMARY

Atari BASIC provides string variables for storing character strings in a program. String variable names may be up to three program lines in length and may be assigned with INPUT, READ . . . DATA, or assignment statements. The only limit on the length of strings is the amount of memory available. Strings may be placed in order by using IF . . . THEN statements. Strings may be concatenated by using a statement of the form A$(LEN(A$)+1)=B$ to add B$ onto A$.

Problems for Section 5-1 . . . .

1. Write a program that requests the user's name and responds with, "HELLO THERE 'YOUR NAME'", using the entered name where 'YOUR NAME' appears here.

2. Enter several words in DATA statements. Write a program that will display the data item that comes earliest in the alphabet. Be sure to use dummy data.

3. Enter several words in DATA statements. Write a program that will display only the word that is alphabetically last in the list.

4. Often in programs we want to ask the user questions for which only "YES" and "NO" are acceptable answers. Since we might want to do this at many points in the same program, it is useful to write one subroutine that sets a numeric variable to "1" for "YES" and "0" for "NO." Write such a subroutine.

5-2...String Functions in Atari BASIC

...ASC( )

Every character is stored in computer memory as a number. The numbers used by Atari BASIC are derived from the ASCII (American Standard Code for Information Interchange) character set and are referred to as ATASCII codes. The values 65 through 90 are used for the letters "A" through "Z"; 97 through 122 are used for "a" through "z." We may find the ATASCII value for any string character with the ASC( ) function.

ASC(A$) is the value used by BASIC for the first character in the string variable A$. We can learn the value of the letter "T" by typing

```
PRINT ASC("T")
```

BASIC will reply, "84." The actual values won't be important to us for most of our programs. The important concept here is that there is an order and that it places letters alphabetically in the correct sequence.

...CHR$( )

CHR$(X) returns the character whose ATASCII code is X. CHR$(90) is Z, while the character for 32 is a space. Try the following program to see the characters and their ATASCII equivalents.

```
90 REM * THE ATARI CHARACTER SET
92 GRAPHICS 0
95 PRINT "    THE ATARI CHARACTER SET"
96 PRINT :PRINT :PRINT
97 POKE 752,1
100 FOR I=0 TO 255
105 IF I=125 OR I=27 OR I=155 THEN 120
107 IF I=28 OR I=29 OR I=30 OR I=31 THEN PRINT
CHR$(27);CHR$(I);:GOTO 120
108 IF I=126 OR I=127 OR I=156 OR I=157 OR I=158
OR I=159 THEN PRINT CHR$(27);CHR$(I);:GOTO 120
110 PRINT CHR$(I);
120 NEXT I
```

Program 5-9. Atari ASCII characters.

85
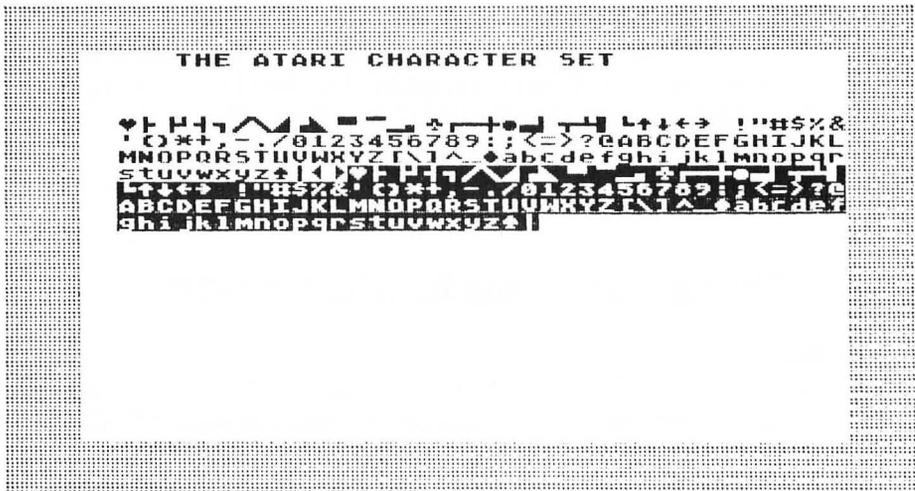
Figure 5-8. Execution of Program 5-9.

Did you notice the "beep" when you ran the program? That was CHR$(253), which you can use anytime you want to get the immediate attention of the user of your program. Don't overdo it, however, because it can get on a person's nerves. You'll notice that we have deliberately excluded some values from the above program in order not to mess up the display. The missing characters would move the cursor into areas that we already wrote on. To see this, RUN the program again, but eliminate lines 105, 107, and 108.

One use of the CHR$ function is to allow us to put quotation marks in strings, when normally that would be illegal, since we use quotation marks to signal the beginning and the end of a string. For example,

```
100 PRINT "THIS IS A VERY "STRANGE" STRING"
```

will result in an error message, but the following statement will work.

```
100 PRINT "THIS IS A VERY ";CHR$(34);"STRANGE"
;CHR$(34);" STRING"
```

...STR$

The STR$ function converts a numeric value to string format. STR$(N) converts the internal binary code used to represent the numeric value of N into the ATASCII code used for each of the digits. Let's examine the effect of a statement such as

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

While N stores a numeric value that we may command the computer to use in arithmetic calculations, T$ stores the digits of the number N as string characters. Thus STR$ permits us to convert any numeric value to a form that can be manipulated by using string functions of Atari BASIC.

...VAL

VAL is the reverse of the STR$ function. VAL(A$) converts a character string of digits in A$ into the binary format used for storing numbers. If the first character could not be part of a number, an "0" is returned. If the function is successful in converting the beginning of a string, it continues until it finds an impossible character. When this happens VAL simply stops processing and returns the value up to that point. For example:

```
VAL("12 DAYS OF VACATION")
```

will convert to

```
12
```

This function handles scientific notation just fine. The value will be converted into the standard form for Atari BASIC. Thus:

```
VAL("123E-1")
```

will convert to

```
12.3
```

There they are: ASC, CHR$, LEN, STR$, and VAL. Combined with concatenation, they form a powerful set of tools to manipulate strings. Now let's use some of them.

Suppose we are working on a program to prepare financial reports. This means that we will be printing numbers that represent money in dollars and cents (or yuan and fen or whatever). BASIC doesn't care what the units of our numeric values might be. As far as BASIC is concerned, one dollar and 20 cents is 1.2. We would like to display that amount as 1.20. So, our first task is to write a routine that will convert numeric values like 1.2 to string values like 1.20. We must also deal with values that come out to fractional cents, so that we can handle 381.2961 properly. Fundamentally, we are faced with a formatting problem.

Let's write a subroutine that accepts a number in M1 and returns a string in D$. Then we can easily write a control routine to test it.

One way to make sure that a number like 1.2 has a trailing 0 is to multiply it by 100. So, 1.2 becomes 120. Of course, we must later insert the decimal point in the proper position. Our new number represents money in cents. Multiplying 381.2961 by 100 produces 38129.61. We need to round this off to the nearest cent. That can be done by adding .5 and eliminating the fractional portion of the resulting number. We saw in the last chapter that INT is made for such a purpose. So, we may calculate the money values in cents with a statement such as

```
M9=INT(M1*100+.5)
```

Notice that we have left the value of M1 unchanged. It is a good idea to write subroutines that leave the input values intact.

Next, we can convert the number of cents from a numeric value to a string with

```
X$=STR$(M9)
```

Now, this string has no decimal point. We know that the two right-hand digits represent cents and must appear to the right of a decimal point. Further, we know that the remaining digits represent dollars and must appear to the left of the decimal point. We may create the D$ string from these three pieces: dollars, decimal point, and cents. A decimal point may be included in one of two ways: enclose a decimal point in quotes or use CHR$(46). We find the code for a decimal point by PRINTing ASC("."). The number of digits in the dollar portion may be found by using the LEN function:

```
D9=LEN(X$)-2
```

Summing up:

```
Dollars        = X$(1,D9)
Decimal point  = CHR$(46)
Cents          = X$(D9+1,D9+2)
```

All that remains is to build the output string by concatenating these three portions. See Program 5-10.

```
999  REM * FORMAT DOLLARS AND CENTS
1000 M9=INT(M1*100+0.5)
1010 X$=STR$(M9)
1020 D9=LEN(X$)-2
1030 D$=X$(1,D9)
1040 D$(LEN(D$)+1)="."
1050 D$(LEN(D$)+1)=X$(D9+1,D9+2)
1060 RETURN
```

Program 5-10. Formatting subroutine.

As mentioned earlier, we used "." instead of CHR$(46) in line 1040.

Now we can write a small control program to test our subroutine. This will require an INPUT statement to enter test values with some dummy value to terminate the input and a PRINT statement to display results. See Program 5-11.

```
90  REM * TEST FORMATTER
100 DIM X$(10),D$(10)
120 PRINT "TEST VALUE";
130 INPUT M1
135 IF M1=-9999 THEN END
140 GOSUB 1000
150 PRINT M1;" = ";D$
160 PRINT
170 GOTO 120
```

Program 5-11. Control routine to test Program 5-10.

It is a good idea to provide a special value of M1 that will allow us to exit the program without having to press the BREAK key or the SYSTEM RESET key. The value −9999 serves that purpose in this program.

```
RUN
TEST VALUE?1.2
1.2 = 1.20

TEST VALUE?-381.2961
-381.2961 = -381.30

TEST VALUE?19
19 = 19.00

TEST VALUE?381.29499
381.29499 = 381.29

TEST VALUE?-9999

READY
```

Figure 5-9. Execution of Program 5-11.

Our program works well for the sample input values. However, consider what happens if the value of M1 is less than one dollar. How could we add a dollar sign? How could we put commas in to mark off thousands? Accountants like to put negative numbers in brackets. How could we do this? Some of these things are left as problems.

...SUMMARY

The string functions ASC, CHR$, LEN, STR$, and VAL have been presented. ASC(A$) returns the numeric code for the first character of A$, and CHR$(A) returns the character whose code is A. LEN(A$) returns the number of characters in A$. STR$(A) converts the numeric value of A to string characters, and VAL(A$) converts the string characters to numeric representation. Strings can be pieced together by concatenation. Portions of strings may also be manipulated by using M$(X,Y) to read from the Xth through the Yth character of the string M$. We may also assign the Xth through the Yth characters of M$ to another string variable by declaring M$(X,Y)=A$.

Problems for Section 5-2 . . . .

1. Modify Program 5-11 to handle amounts less than one dollar.

2. Modify Program 5-11 to place a dollar sign ($) to the left of the first digit in the formatted result.

3. Modify Program 5-11 to insert commas to mark off thousands.

4. Correct Program 5-11 to properly display 0.00 if the amount is 0.

5. Modify Program 5-11 to enclose negative values in angle brackets. That is, -1.43 should display as $\langle 1.43 \rangle$.

6. Write a program to perform the reverse conversion: the string (1,234.56) should convert to the numeric value -1234.56.

89

7. Problems 1 to 5 could be worked cumulatively. The result could be a program that performs all of the tasks described in the five problems. Do this.

8. Write a program to display messages on the screen so that they scroll horizontally across the screen. Use DATA statements to supply the messages.

9. Given the date in yy/mm/dd form, display the date as Month dd, 19yy. That is, 82/12/31 becomes December 31, 1982. You may want to test for bad dates like 82/04/31.

PROGRAMMER'S CORNER 5

...Checking the Last Key Pressed

There will be times when you want the user of your program to read and understand what is on the screen before going on in the program. In cases like this, putting in a simple delay loop may make the delay too short to read the program or too long, so that the user gets bored waiting. A nice way to take care of this problem is to use memory location 764, which holds the internal code for the last key pressed. Now, this code does not correspond to the ATASCII value, but for this purpose we don't care. We just want to know if a key has been pressed. If it has, we will go on with the program; if it hasn't, we'll continue to wait. This is done as follows:

```
90 REM * CHECKING FOR A KEY TO BE PRESSED
100 PRINT "THIS WILL STAY ON THE SCREEN UNTIL"
110 PRINT "YOU PRESS A KEY."
120 POKE 764,255
130 IF PEEK(764)<>255 THEN 150
140 GOTO 130
150 POKE 764,255
155 PRINT
160 PRINT "AHA!!!! YOU PRESSED A KEY."
200 END
```

Program 5-12. Checking for a key to be pressed.

...Changing the Character Set

The characters that appear when you type on the screen have to come from somewhere. Somehow the key that is pressed is translated into a command that causes the computer to find the right character and display it. The process is actually reasonably straightforward. Each character, if you look at it closely, is made up of an arrangement of dots. These are defined on an 8-by-8 grid. The letter "B" is shown in Figure 5-10.

1 2 8	6 4	3 2	1 6	8	4	2	1	Total
								0
								124
								102
								124
								102
								102
								124
								0

Figure 5-10. The letter B of the Atari character set.

However, it isn't stored in memory like this. There it is stored as a sequence of eight numbers, one for each row of the grid, starting at the top. The columns are identified by successive powers of two, starting from the right, as also shown in Figure 5-10. Each row is then the sum of the numbers for all the squares that are "colored in" by the particular character. In this way, each character has a unique set of eight numbers representing it. So our "B" is stored as 0,124,102,124,102,102,124,0. So it is also with all the other 127 characters of the character set; there are eight numbers for each and they run consecutively. A slight complication is that they do not run in ATASCII order, however. They are stored in order by an internal code which, thank goodness, bears some resemblance to the ATASCII order. Here's how the internal codes relate to ATASCII:

ATASCII	INTERNAL CODE	ATASCII	INTERNAL CODE
0–31	Add 64	128–159	Add 64
32–95	Subtract 32	160–223	Subtract 32
96–127	Same as ATASCII	224–255	Same as ATASCII

The location where the character set is stored is kept in memory location 756. If we multiply what is there by 256 we find the start of the character set. So all we need do is enter

```
PEEK(756)*256
```

and we have the first number of the first character in the set. Looking at our table above, the first character must be the one with an ATASCII value of 32 (since its

internal code is 0), which is a space. Since the ATASCII value of our "B" is 66, its internal code is $66 - 32$ or 34. Therefore, it will start $34*8$ numbers (bytes) beyond the start. Let's check that out.

```
90 REM * CHECK THE LETTER "B"
100 CHARSET=PEEK(756)
120 GRAPHICS 0
130 CHARSTART=CHARSET*256
140 STARTB=CHARSTART+8*(66-32)
150 PRINT "The letter B is made up of:"
160 FOR I=STARTB TO STARTB+7
170 PRINT PEEK(I)
180 NEXT I
```

Program 5-13. The eight-number representation of the letter B.

```
RUN
The letter B is made up of:
0
124
102
124
102
102
124
0


READY
```

Figure 5-11. Execution of Program 5-13.

Sure enough, that agrees with our drawing.

We can't go in and change any of these values to redefine characters, much as we might like to. The characters are stored in ROM, which is Read Only Memory, and its contents can't be changed. We can, however, do the next best thing: we can store a different value in location 756 and fool the computer into thinking the character set starts somewhere in a part of memory that we have control over. Then we just have to put our new character set there and that's it. Not quite! The computer makes pretty indiscriminate use of the memory we use, moving things here and there as it sees fit. Thus, our new characters would get messed up in no time, except for one more good thing. Memory location 106 holds a value that tells the computer where the top of memory is, and it won't go beyond this limit. So we need to move the top of memory down, to fool the computer again, and then use the protected portion above this new top value to store our character set. The value in location 756 is actually the value for the top of memory divided by 256; it can only be incremented by multiples of 256. This is just fine because we need $256*8$ spaces to fit in all of the character set. So we will make these changes and then, once we have the character set where we can get at it, we'll go ahead and change a character. For

an example, we'll redefine the dollar sign ($) as a box with a dot in the middle. Figure 5-12 shows the eight values we need to have.

1 2 8	6 4	3 2	1 6	8	4	2	1	Total
								0
								0
								126
								66
								90
								66
								126
								0
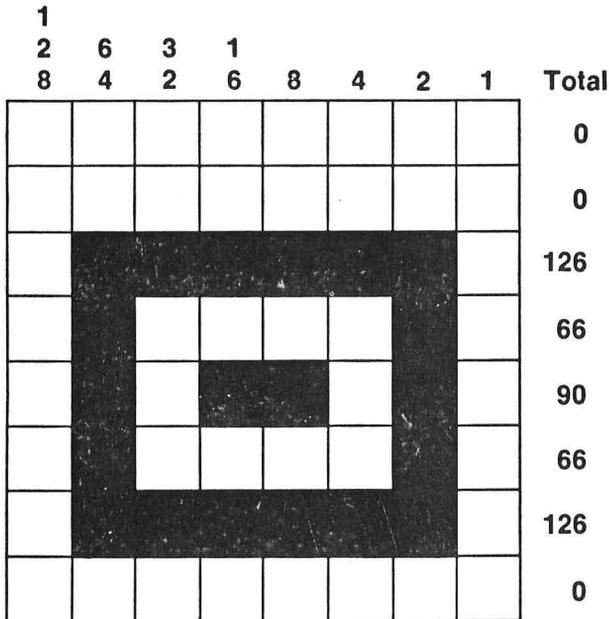
Figure 5-12. The "$" redefined.

The ATASCII value for "$" is 36, so these values need to go (36—32)*8 = 32 bytes from the start of our list. Here's our program.

```
90 REM * REDEFINE $ CHARACTER
100 TOP=PEEK(106)-4
110 POKE 106,TOP
120 GRAPHICS 0
130 CHARGO=TOP*256
132 CHARFROM=PEEK(756)*256
135 PRINT "Moving the character set.  Hold on.."
140 FOR PLACE=0 TO 1023
150 POKE CHARGO+PLACE,PEEK(CHARFROM+PLACE)
160 NEXT PLACE
170 POKE 756,CHARGO/256
180 FOR I=0 TO 7
182 READ A:POKE CHARGO+I+(4*8),A
184 NEXT I
186 PRINT "OUR NEW CHARACTER........"
188 PRINT "$ $ $ $ $ $"
190 PRINT "NOT BAD!"
192 END
200 DATA 0,0,126,66,90,66,126,0
```

Program 5-14. Redefine the $ character.

Go ahead and use the "$" key. It will stay redefined unless we change graphics modes or hit SYSTEM RESET or turn off the computer. You can just as easily redefine keys to make mathematical symbols, foreign alphabets, or space ships. It's just as straightforward to redefine a whole bunch of the characters. You can even get clever and make more detailed shapes by redefining characters to be parts of a bigger shape and then PRINTing the combination to get the whole shape. We can also reserve more of memory for character sets and store several variations. Then, merely by putting different values in memory location 756, we can instantly switch from one character set to another. Simple animation may be done in just this way.

Chapter 6

Arrays

We have been using variables to store values one at a time. Such variables are referred to as "simple" variables. We have been able to perform marvelous feats on the computer with simple variables. We will accomplish even more with array variables. An array variable allows us to designate a collection of data values with a single variable name. Now, instead of designating the scores of the players in a five-player game as S1, S2, S3, S4, and S5, we may use an array variable. S(X) may be used to refer to the score of the Xth player. S(X) is read "S sub X." We may use the same variable name for an array as for a simple variable. You may want to refrain from doing this to avoid confusion, however. The value in parentheses is called a *subscript*. Each data value in the array is called an *element*. Using an array we could do the scoring for all five players with the same little segment of our BASIC program.

Arrays are used for storing information that naturally belongs together. Tax tables, pricing structures, inventory information, and life insurance premiums are all appropriate for arrays. There are many times when an array is useful for storing information about the workings of the program itself. We may use arrays for storing test scores, temperatures, random numbers, and lists of all kinds.

If we are working with prime numbers up to 1000, it may be desirable to store them all in an array. Even though we could recreate the sequence at any time, it might be more convenient to have them all right there at the flick of a subscript.

Arrays allow us to have many pieces of information available for use at one time. Recall that Atari BASIC allows the programmer to have 128 variables in any program. For most programs this is more than enough. But if we have a program that manipulates the batting averages of all the players in the National League, we

95
. . .

would be in trouble if each player was represented by one variable. The name of an array takes up only one variable name, no matter how many items there are in the array. We also use arrays because they allow easy storage and retrieval of values. By their very structured nature they help the programmer keep careful track of many pieces of information.

6-1...Numeric Arrays (One Dimension)

We may immediately benefit from the array concept by simply referring to array variables as needed. If we want the sixth element of T to be 5, we simply code a statement such as

```
200 T(6)=5
```

We may readily use arrays in every way that we have been using simple variables. PRINT or IF . . . THEN may directly use array variables. READ and INPUT require special treatment. There are some rules, however, that we need to get clear first.

Rule 1: All arrays must be DIMensioned before they can be used, no matter how many elements the array will contain. For example,

```
100 DIM A(20),FILE(240),A120(120)
```

will reserve space for three arrays.

There is a big difference between DIMensioning strings, which we have already done, and DIMensioning arrays. DIM A$(100) means that we want to set aside space in memory for a single string called A$ that would contain a maximum of 100 characters. DIM A(100), on the other hand, will reserve space in memory for 101 *different* numeric values. These are named A(0), A(1), A(2), . . . , A(100). Notice that the array starts with the zeroth element. Computers prefer to count from 0 rather than 1. You are free to use or ignore this zeroth member of the array.

Rule 2: You cannot READ or INPUT numeric values directly into an array. This means that the following statements are not permitted:

```
100 READ A(20)
100 INPUT A(20)
```

Instead, you take an indirect approach:

```
100 READ ITEM
110 A(20)=ITEM
```

or

```
100 INPUT ITEM
110 A(20)=ITEM
```

We have used ITEM as a temporary variable to help us READ or INPUT data into our array.

Rule 3: Never assume that the elements of an array that you have created are initialized by the computer to be 0 when you DIMension the array.

This is a fairly small point, but one that can get you into trouble. The space set aside for your array may contain some "garbage" numbers, that is, numbers other than zero. If it is important in your program that your array contain all zeros (or any particular value), then put the values you want there before you use the array with a loop such as

```
100 FOR I=1 TO 20
110 A(I)=0
120 NEXT I
```

Now let's look at a problem that makes use of arrays. In a given week we record the daily high temperature:

Sunday	72
Monday	78
Tuesday	76
Wednesday	79
Thursday	85
Friday	85
Saturday	71

There are any number of questions we might ask. We might want to know the average of all the temperatures in our list or the highest and lowest temperatures. By using an array we can easily find the answers. Let's READ the data into elements 1 through 7 of an array named TEMP.

The average is easy. We just add up the seven temperatures and divide by seven. We may use TOTAL for the total. The first value of TOTAL is the temperature for the first day.

We may find the highest and lowest temperatures by using two variables, HI and LO. These may be initialized as the temperature of the first day, as it is initially both the highest and the lowest temperature. We will then compare the other temperatures with these starting values and use the results of these comparisons to replace HI and LO as needed until we have gone through all the values.

The solutions for the three questions regarding temperatures each call for setting initial values and then performing some operation on each of the six days after the first—that is, Monday through Saturday. So our program will have a section to set up all of these initial values and a section with a loop that performs some calculation for each of the three questions. See Program 6-1.

```
90 REM * ENTER THE TEMPERATURES IN ARRAY TEMP
95 DIM TEMP(7)
100 FOR J=1 TO 7
105 READ ITEM
110 TEMP(J)=ITEM
120 NEXT J
145 REM * SET UP INITIAL CONDITIONS
```

```
150 TOTAL=TEMP(1)
160 HI=TEMP(1):LO=TEMP(1)
190 REM
200 FOR J=2 TO 7
210 TOTAL=TOTAL+TEMP(J)
230 IF TEMP(J)>HI THEN HI=TEMP(J)
240 IF TEMP(J)<LO THEN LO=TEMP(J)
250 NEXT J
290 REM
300 PRINT "AVERAGE TEMPERATURE: ";TOTAL/7
310 PRINT "HIGHEST TEMPERATURE: ";HI
320 PRINT " LOWEST TEMPERATURE: ";LO
890 REM
900 DATA 72,78,76,79,85,85,71
990 END
```

Program 6-1. Find average, highest, and lowest temperatures.

```
RUN
AVERAGE TEMPERATURE: 78
HIGHEST TEMPERATURE: 85
  LOWEST TEMPERATURE: 71
```

READY

Figure 6-1. Execution of Program 6-1.

The next question that might be asked is, "How many times did the temperature increase, decrease, and remain unchanged?" We might now use the variables I, D, and U for this. We might want to know on what days the highest and lowest temperatures occurred. These are left as exercises.

Suppose we wish to simulate drawing numbers from a hat. We can easily do it with random numbers, provided that we may return each number to the hat before drawing the next one. If we want to simulate drawing without replacement, then we need a way of keeping track of what has been drawn. Here is an ideal application for an array. We simply set each element of an array equal to 1 and make the value 0 when that element has been selected. If the selected element is 1 then we know that it is available for use: use it and set it to 0. If a selected element is 0 then we know that it is not available for use and we must select again. Let's look at a program for drawing five numbers at random from among ten. See Program 6-2.

```
90 REM * DRAWING FIVE NUMBERS AT RANDOM FROM
AMONG TEN
95 DIM A(10)
100 FOR J=1 TO 10
110 A(J)=1
120 NEXT J
190 REM
200 FOR J=1 TO 5
210 R=INT(RND(0)*10+1)
250 IF A(R)=0 THEN 210
```

98
...

```
260 PRINT "     ";R;
270 A(R)=0
280 NEXT J
290 PRINT
300 END
```

Program 6-2. Drawing five numbers at random from among ten.

```
RUN
     4    10    6    7    3

READY
```

Figure 6-2. Execution of Program 6-2.

By all appearances our program works just fine. It might be interesting to evaluate how well it does work. One measure of quality is the number of unusable random numbers generated. We can easily insert a counting variable to determine this. This is also left as an exercise.

Considering the problem set before us, the trial-and-error method of the above program is not really a serious flaw in design. Drawing five numbers from among ten, or even drawing ten from among ten, does not require major computer resources. However, what happens when we increase the numbers? Suppose we want to draw 100 from among 100? It is worth investing some effort to eliminate the trial and error entirely.

Here is a plan that allows us to use every random number selected. First initialize the elements of the array as follows:

```
100 FOR J=1 TO 10
110 R(J)=J
120 NEXT J
```

This means that each element stores one of the numbers in the range 1 to 10. Next, select a random number in the range 1 to 10 and use that value as the subscript, say S. Now display R(S), replace R(S) with R(10), and select a random number in the range 1 to 9. Since either we are on the first draw or we have replaced R(S), we will not need to determine if it has been used: we know it has not been used. Since we have moved R(10) into a lower-numbered element, we may select from among fewer elements and still include all of the remaining numbers in the next random selection. The second time through we move R(9) into the selected element. We simply repeat the select-display-replace sequence until the desired number of random draws have occurred.

We need to calculate the number of elements remaining. As the draw number (J) goes from 1 to 5, the number of elements remaining goes from 10 to 6. Thus, we can calculate the last element with

```
210 L=10-J+1
```

See Program 6-3.

99

```
90 REM * DRAWING RANDOM NUMBERS WITHOUT
REPLACEMENT
92 REM * WITH NO TRIAL AND ERROR
95 DIM R(10)
100 FOR J=1 TO 10
110 R(J)=J
120 NEXT J
190 REM
200 FOR J=1 TO 5
210 L=10-J+1
230 S=INT(RND(0)*L+1)
240 PRINT "    ";R(S);
250 R(S)=R(L)
270 NEXT J
290 PRINT
300 END
```

Program 6-3. Drawing efficiently without replacement.

Notice that the element is printed in line 240 and then replaced in line 250. L is always the number of active elements in the array. Even if we happen to select the Lth element, this method continues to function properly—the Lth element will be assigned to itself, and no harm is done.

```
RUN
    8    1   10    6    7

READY
```

Figure 6-3. Execution of Program 6-3.

Problems for Section 6-1 . . . .

1. Modify the daily temperature program (Program 6-1) to tabulate the number of times the temperature increased, decreased, or remained unchanged.

2. Modify the daily temperature program (Program 6-1) to determine on which days the highest and lowest temperatures occurred.

3. In the first program that draws numbers from a hat (Program 6-2), insert a variable to count the number of unusable numbers generated. Run the program several times to obtain a range of values.

4. Do Problem 3, drawing 10 numbers from among 10.

5. Modify Program 6-3 to select 100 numbers from among 100.

6. Fill a 20-element array with twice the value of the subscript. Display all of the elements in order and in reverse order.

7. Fill one array with the values 6, 3, and 9. Fill a second array with the values 2, 8, 6, and 5. Display all possible pairs of one element from each array. There are 12 pairs.

8. Fill two arrays as in Problem 7. Fill a third array with all elements from these two arrays with no duplicates.

9. Fill a 100-element array with random numbers. Count the number of increases, decreases, and no changes. Calculate the average.

6-2...Numeric Arrays (Two Dimensions)

We have seen that one-dimensional arrays may be used to organize data in a list. We can also use two subscripts to arrange data into tables of all kinds. We might be interested in the temperature at 6:00 A.M., 12:00 noon, and 6:00 P.M. for a week. For this we need an array with two subscripts. Such an array is referred to as *two-dimensional*. We will use one dimension to represent the days of the week and the other to represent the three different times of day. Let's look at a program to find the average daily temperature using three readings a day. See Program 6-4.

```
90 REM * FIND AVERAGE TEMP
95 DIM TEMP(7,3)
100 FOR DAY=1 TO 7
110 FOR READING=1 TO 3
120 READ TEMP
125 TEMP(DAY,READING)=TEMP
130 NEXT READING
140 NEXT DAY
175 REM
180 PRINT "        TEMPERATURE"
190 PRINT "Day 6AM 12N 6PM Avg."
200 FOR DAY=1 TO 7
202 PRINT DAY;"    ";
205 TOTAL=0
210 FOR READING=1 TO 3
220 TOTAL=TOTAL+TEMP(DAY,READING)
230 PRINT TEMP(DAY,READING);"   ";
240 NEXT READING
250 PRINT TOTAL/3
260 NEXT DAY
980 REM
1000 DATA 76,79,75,72,77,76
1020 DATA 74,79,81,75,80,83
1040 DATA 80,77,70,68,65,65
1060 DATA 65,67,76
```

Program 6-4. Find daily average temperature.

Note that in line 95 we have DIMensioned an array with

```
95 DIM TEMP(7,3)
```

Think of this as a sheet of paper with seven columns (for the days of the week) and three rows (for the three daily readings). We could just as well have used

101
• • •

TEMP(3,7), as long as we keep track of how we access the array to enter and retrieve data. As with one-dimensional arrays, you must DIMension an array before it can be used.

```
RUN
        TEMPERATURE
Day 6AM  12N  6PM  Avg.
1    76   79   75   76.66666666
2    72   77   76   75
3    74   79   81   78
4    75   80   83   79.33333333
5    80   77   70   75.66666666
6    68   65   65   66
7    65   67   76   69.33333333

READY
```

Figure 6-4. Execution of Program 6-4.

...Zero Subscripts

The zero subscript is always available. In many programming situations the zero subscript is a great convenience. The zero term of a polynomial is easily represented in this way. The positions reserved for the zero subscripts are there whether we use them or not. For most programs the impact of zero subscripts is minor. When you are writing large programs on a machine with little memory, however, it may become necessary to use them just to get the program to fit.

...SUMMARY

An array enables us to manage a number of variables by using one variable name. Arrays may be one- or two-dimensional and are created with statements of the form DIM ARRAY(X) for one-dimensional arrays or DIM ARRAY(X,Y) for two-dimensional arrays. The array size is one larger than the number used because the computer starts counting with zero. Array elements can be used in BASIC statements wherever a simple numeric variable can be used. With arrays we will often find it convenient to use FOR . . . NEXT loops to process all elements or a block of the elements.

Problems for Section 6-2 . . . .

1. Write a routine for Program 6-4 to find the maximum temperature for each of the three reading times (6:00 A.M., 12:00 noon, and 6:00 P.M.).

2. Write a routine for Program 6-4 to find the maximum temperature for each day.

3. Write a routine for Program 6-4 to find the average temperature for each of the three reading times (6:00 A.M., 12:00 noon, and 6:00 P.M.).

6-3...Creating String Arrays

Several implementations of BASIC allow you to have arrays of strings that can be used and manipulated just like arrays of numeric variables. Atari BASIC does not have this feature. Recall that DIM A$(X) has a very different meaning for strings than DIM A(X) has for numbers. A$(X) defines a string named "A$" that will have a maximum length of X characters, while A(X) defines an array to be called "A" that will have at most X + 1 numeric items in it.

This doesn't mean that we can't make something that will have the properties of a string array and use that instead. Atari BASIC does permit strings to have any length right up to the amount of memory that is available, which will make a very long string indeed. We will use this capability to make an array of string data where, instead of the strings being stacked one beneath the other, the strings will be one right after the other in a single long string. We will need to impose one important limitation: all of the strings must be the same length. This will permit us to find where each little string is within the large string. This sounds impractical, of course. Why should all the strings just happen to have the same length? They won't, of course, but we will force them to be that way by adding blank spaces to each of them to get them all up to the length of the longest one.

Consider, for example, the days of the week. Check them over and you'll find that Wednesday is the longest, with nine letters. We will, therefore, DIM a string for our array with a length of $9 * 7 = 63$ characters for all the days of the week. Program 6-5 reads the days of the week from DATA statements, puts them into one string, and then accesses the pieces of that string to recall the individual days.

```
90 REM * READ AND DISPLAY DAYS OF THE WEEK
95 DIM WEEK$(63),DAY$(9)
100 FOR DAY=1 TO 7
110 READ DAY$
115 IF LEN(DAY$)<9 THEN DAY$(LEN(DAY$)+1)=" "
:GOTO 115
120 WEEK$(LEN(WEEK$)+1)=DAY$
130 NEXT DAY
190 REM
200 FOR DAY=1 TO 7
210 PRINT WEEK$(9*(DAY-1)+1,9*(DAY-1)+9)
220 NEXT DAY
230 END
990 REM * DAY DATA
1000 DATA SUNDAY
1010 DATA MONDAY
1020 DATA TUESDAY
1030 DATA WEDNESDAY
1040 DATA THURSDAY
1050 DATA FRIDAY
1060 DATA SATURDAY
```

Program 6-5. Display the days of the week.

```
RUN
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY

READY
```

Figure 6-5. Execution of Program 6-5.

In line 95 we DIM two strings, DAY$ for one day and WEEK$ for the entire week. In line 115 we add on spaces if the string for a particular day is not as long as the string for the longest day, Wednesday. If the string is shorter, line 115 keeps adding spaces until it reaches a length of nine characters. Line 210 then pulls apart this long string to get at the days. We know that each day in the string occupies nine characters (including extra spaces). Thus, the first day starts at WEEK$(1,1), the second at WEEK$(10,10), and so forth. Our calculation takes the next nine characters in WEEK$, starting at each of these values, and prints them. We will be using this sort of calculation to locate portions of strings again and again, so go through it with several of the days to see how it works. We had our DAY loop run from 1 to 7, since the days run from 1 to 7, but this is really arbitrary. We had to put (DAY-1) into the calculation in line 210 to correct for this. If we had made the DAY loop go from 0 to 6, we could have simplified the calculation a little. People don't like to count from zero, however, so we avoided it here.

Once the string data are stored as part of a long string, we may manipulate them in many ways. It may be that on a report we want the days of the week spelled out in one place and abbreviated in another. We can do this with our ability to take any portion of the string we want. To get the abbreviations of the days, we merely need to take the first three characters of each day name. We already know where each name starts. We do this with a slight change in line 210.

```
210 PRINT WEEK$(9*(DAY-1)+1,9*(DAY-1)+3);"   ";
WEEK$(9*(DAY-1)+1,9*(DAY-1)+9)

RUN
SUN   SUNDAY
MON   MONDAY
TUE   TUESDAY
WED   WEDNESDAY
THU   THURSDAY
FRI   FRIDAY
SAT   SATURDAY

READY
```

Figure 6-6. Execution of modified Program 6-5.

104
•••

Recall that in Program 6-4, to average the three temperatures taken each day for a week we labeled the days of the week from 1 to 7. We now have the ability to produce a more readable report. Having the days of the week listed by number runs counter to the fact that nearly everyone uses names for them. We may modify our program to label each line with the day. If we use the full day names, then we have to deal with the fact that not all names have the same number of letters. We can handle this by using comma spacing, but then the day names will take up too much space for our compact report. Instead, we'll use our newfound ability to pick out the abbreviations. See Program 6-6.

```
90 REM * FIND AVERAGE TEMP
95 DIM WEEK$(63),DAY$(9),TEMP(7,3)
100 FOR DAY=1 TO 7
110 READ DAY$
115 IF LEN(DAY$)<9 THEN DAY$(LEN(DAY$)+1)=" "
:GOTO 115
120 WEEK$(LEN(WEEK$)+1)=DAY$
122 FOR READING=1 TO 3
124 READ TEMP
126 TEMP(DAY,READING)=TEMP
130 NEXT READING
140 NEXT DAY
175 REM
180 PRINT "     TEMPERATURE"
190 PRINT "Day 6AM 12N 6PM Average"
200 FOR DAY=1 TO 7
202 PRINT WEEK$(9*(DAY-1)+1,9*(DAY-1)+3);" ";
205 TOTAL=0
210 FOR READING=1 TO 3
220 TOTAL=TOTAL+TEMP(DAY,READING)
230 PRINT TEMP(DAY,READING);"   ";
240 NEXT READING
250 PRINT TOTAL/3
270 NEXT DAY
990 REM * DAY DATA
1000 DATA SUNDAY,76,79,75
1010 DATA MONDAY,72,77,76
1020 DATA TUESDAY,74,79,81
1030 DATA WEDNESDAY,75,80,83
1040 DATA THURSDAY,80,77,70
1050 DATA FRIDAY,68,65,65
1060 DATA SATURDAY,65,67,76
```

Program 6-6. Display average daily temperature with day names.

Look at the DATA section. We have included the days of the week right in with the temperature data. Doing it this way helps to clearly document which temperatures go with which day.

```
RUN
     TEMPERATURE
Day  6AM  12N  6PM  Average
SUN  76   79   75   76.66666666
MON  72   77   76   75
TUE  74   79   81   78
WED  75   80   83   79.33333333
THU  80   77   70   75.66666666
FRI  68   65   65   66
SAT  65   67   76   69.33333333

READY
```

Figure 6-7. Execution of Program 6-6.

This report is easy to read. We do not wonder whether day 1 is Sunday or Monday. Four of the averages are displayed with ten digits. We might want to round those values off to the nearest tenth. If it is important to have the day names spelled out, then we could easily change the appearance of the report by using POSITION X, Y or POKE 85 (remember that one?).

Suppose we have a record store and are using a computer to help calculate sales slips for us. Each record is marked with a letter from H to P. This letter is assigned according to the price of the record. Thus, H is the label on every $2.99 record, and I is the label on every $3.45 record. We can easily write a program using arrays to calculate a total sale for us.

We can enter the correspondence between letters and prices into the program by using READ . . . DATA statements. Two arrays will be required—one of our long strings containing the letter codes and a numeric array for the prices. It is a simple matter to arrange the data so that the position in our string corresponds to the price for that code in our numeric array. Placing the data in DATA statements makes it easy to add new codes or change prices. We will use "!" as the signal to stop reading data. It is always a good idea to leave a gap in line numbers between the real data and the termination signal. See Program 6-7.

```
 90 REM CALCULATE SALES SLIPS
100 DIM CODE$(26),ITEM$(1),PRICE(26),RECORD$(1)
200 FOR I=1 TO 26
205 READ ITEM$,PRICE
210 IF ITEM$="!" THEN 250
215 CODE$(LEN(CODE$)+1)=ITEM$
220 PRICE(I)=PRICE
230 NEXT I
250 N1=I-1
290 REM * REQUEST INPUT AND CALCULATE HERE
300 PRINT "(TYPE '!' TO STOP)"
310 TOTAL=0:NUMBER=0
320 PRINT "RECORD CODE: ";
330 INPUT RECORD$
335 IF RECORD$="!" THEN 500
```

```
340 FOR J=1 TO N1
350 IF RECORD$=CODE$(J,J) THEN 400
360 NEXT J
370 PRINT "NOT FOUND - REENTER"
380 GOTO 320
400 TOTAL=TOTAL+PRICE(J)
410 NUMBER=NUMBER+1
420 GOTO 320
490 REM
500 PRINT
510 PRINT "RECORDS: ";NUMBER
520 PRINT "  TOTAL: $";TOTAL
900 END
1000 DATA H,2.99,I,3.45
1010 DATA J,3.69,K,3.99
1020 DATA L,4.49,M,4.99
1030 DATA N,5.99,O,6.99
1040 DATA P,7.99
1100 DATA !,0
```

Program 6-7. Total price in record store.

Program 6-7 is set up in four segments. The first segment, from lines 100 to 250, reads in the price data. The second segment, from lines 300 to 420, handles the entry of figures for each sale. Lines 500 to 520 display the final results. And the fourth segment is the DATA in lines 1000 to 1100.

```
RUN
(TYPE '!' TO STOP)
RECORD CODE: ?H
RECORD CODE: ?P
RECORD CODE: ?P
RECORD CODE: ?O
RECORD CODE: ?L
RECORD CODE: ?A
NOT FOUND - REENTER
RECORD CODE: ?!

RECORDS: 5
  TOTAL: $30.45

READY
```

Figure 6-8. Execution of Program 6-7.

...Geography

Let's write a program to play Geography, a simple game for two or more players. We will write a program for a person to compete with the computer. Each player says the name of a place whose first letter is the same as the last letter of the name chosen by the previous player. Of course, the first name can be any place at all. If I

107

say Boston, then you might say New York. That fits the rule, because Boston ends with an "n" and New York begins with one. The next player might think of Kansas. No name may be used a second time. The first person unable to think of an appropriate name drops out.

We can easily program the computer so that it "remembers" all of the names used. The computer will have a very limited "vocabulary" to choose from early in the game, and will lose very quickly. The more games the computer plays, however, the tougher it will be to beat.

We will put the names of the places one after the other in one long string called NAMES$. We will create a numeric array, START( ), to keep a file of where each place name in NAMES$ starts and another, FINISH( ), to keep track of where each one ends. We do this because this particular game makes use of the first and last characters in the place names. We will also use a numeric array to tell us if a specific name has been used. Let's set up a numeric array AVNAMES( ), for AVailable NAMES, so that 1 indicates that the name in the corresponding position of the NAMES$ string is available for use and a 0 means that the name has already been used in this game. Thus if AVNAMES(5)=1, then NAMES$(START(5),FINISH(5)) may be used. We can enter a few names into the NAMES$ string by using DATA statements. This way the computer has some names to start with. Let's allow the computer to select the first name.

It may sound like a big job to produce a program that performs as described. We can easily trim the job down to size by spending a little extra time organizing before we generate any BASIC program statements. Think about the steps in the game. There are six easily defined segments in our program.

1. Read the names into NAMES$ and fill the START and FINISH arrays.
2. Display the instructions.
3. Initialize the AV array to all ones.
4. Have the computer begin the game.
5. Process the person response.
6. Prepare the computer response.

Each of these six jobs may be programmed as a subroutine. The advantages of doing it this way are tremendous. When we first test our completed program it will be easy to spot which subroutine is not performing properly. Once we are satisfied that our program is working well, it will be a simple matter to determine which subroutines we need to modify or replace to change the program so that the names are stored in a file on disk or tape.

Let's begin by writing the control routine that will manage the six subroutines listed above. In thinking about this routine we need to handle the situation when the computer runs out of names in item number six. We can save the computer response

in a string variable and save "QUIT" when the computer quits. This thought leads us to think about letting the person quit at any time.

We select CP$, computer play, for the computer's response and PP$, player play, for the person's response. Further, we may give the player the option to play another game. We will provide for 100 names, which is about as many as a 16K Atari can handle. If you have more memory, you are free to increase this value. See Program 6-8a.

```
10 DIM NAME$(20),NAMES$(2000),START(100),
FINISH(100)
15 DIM AVNAMES(100),PP$(20),A$(3),CP$(20)
21 NAME$=""
25 GOSUB 8000:REM * READ NAMES ARRAY
30 GOSUB 9000:REM * INSTRUCTIONS
35 GOSUB 4000:REM * INITIALIZE AVAILABLE NAME
ARRAY
40 GOSUB 7000:REM * COMPUTER STARTS
50 GOSUB 6000:REM * PERSON RESPONDS
55 IF PP$="QUIT" THEN 75
60 GOSUB 5000:REM * RESPONSE OF COMPUTER
65 IF CP$<>"QUIT" THEN 50
75 PRINT "DO YOU WANT TO PLAY AGAIN (Y/N)";
80 INPUT A$
90 IF A$(1,1)="N" OR A$(1,1)="n" THEN END
120 GOTO 30
```

Program 6-8a. Control routine to play Geography.

The six steps have become six subroutines at lines 8000, 9000, 4000, 7000, 6000, and 5000. The choice of line numbers is arbitrary. Now we are well prepared to write each individual subroutine.

We read the names in 8000. The place names are entered in DATA statements. We check on the current length of NAMES$ in 8011 and use this to establish values for START and FINISH as the data are read. Line 8020 adds (concatenates) the place names onto NAMES$. We choose to provide the signal data "DONE". See Program 6-8b.

```
7998 REM * READ NAMES
8000 I9=0
8010 READ NAME$
8011 LONGSIZE=LEN(NAMES$)
8012 I9=I9+1:START(I9)=LONGSIZE+1
8015 IF NAME$="DONE" THEN 8080
8017 SIZE=LEN(NAME$):FINISH(I9)=LONGSIZE+SIZE
8020 NAMES$(LEN(NAMES$)+1)=NAME$
```

```
8025 GOTO 8010
8080 N0=I9-1
8090 RETURN
8100 DATA NEW YORK,CHICAGO,PHILADELPHIA,BOSTON
8590 DATA DONE
```

Program 6-8b. Read names into an array for Geography game.

Notice that line 8080 saves the number of names in the array in numeric variable N0.

Instructions are simple enough. We can just display a little description on the screen. Think about that. How fast do people read? We must provide a way for the fast reader to move on and allow the slow reader a chance to finish. We can do this by asking the person to tell the program when he or she is ready. See Program 6-8c.

```
8998 REM * INSTRUCTIONS
9000 PRINT "}This program will play a
geography"
9005 PRINT "game with you. You will take
turns"
9010 PRINT "with the computer. Each of you
will"
9015 PRINT "be trying to think of names of
places"
9020 PRINT "such that the first letter of
your"
9025 PRINT "name is the same as the last
letter"
9030 PRINT "of the previously used place
name.":PRINT
9035 PRINT "PLEASE USE CAPITAL LETTERS ONLY."
:PRINT
9040 PRINT "ARE YOU READY (Y/N)"
9045 INPUT A$
9050 IF A$(1,1)<>"Y" AND A$(1,1)<>"y" THEN
9045
9060 PRINT "}":REM ESC CTRL+CLEAR
9080 RETURN
```

Program 6-8c. Geography game instructions.

The wording of instructions is somewhat subjective. Instructions should tell the user what to expect. We carefully note that responses should be in capital letters only. The program could have been made to handle responses in upper- or lowercase letters, but that would have just complicated our example here. In line 9060 we clear the screen after the player has finished reading the instructions.

Initialization of the AVNAMES array beginning at line 4000 is very straightforward. See Program 6-8d.

```
3998 REM * INITIALIZE AVAILABLE NAMES ARRAY
4000 FOR J9=1 TO N0
4010 AVNAMES(J9)=1
```

```
4020 NEXT J9
4090 RETURN
```

Program 6-8d. Initialize available-names array.

To start the game at line 7000 we have the computer select at random a name from the names array. The place must be recorded as used and the CP$ string variable is loaded with the name selected. See Program 6-8e.

```
6998 REM * COMPUTER BEGIN THE GAME
7000 X9=INT(RND(0)*(N0-1))+1
7028 CP$=NAMES$(START(X9),FINISH(X9))
7030 PRINT "FIRST PLACE : ";CP$:AVNAMES(X9)=0
7090 RETURN
```

Program 6-8e. Begin Geography game.

Once the computer has produced a place name, the program proceeds to the person response subroutine.

We agreed to have the person's response stored in PP$. The person's response must pass a number of tests. It ought to have at least two characters. That is handled with the LEN( ) function. The first letter of the person's response must match the last letter of the computer place name. We do this in line 6030 by comparing the first character of the player's response, PP$(1,1), with the last character of the computer place name, CP$(LEN(CP$),LEN(CP$)). If PP$ passes these two tests then we must see if it is in the list of names stored in the NAMES$ string (lines 6040-6050). If PP$ is in the list, has it been used during the latest game (line 6100)? If it is not in the list, then we put it in the list (lines 6065-6073). See Program 6-8f.

```
5998 REM * PERSON GO
6000 PRINT
6010 PRINT "   YOUR TURN: ";
6011 INPUT PP$
6012 IF PP$="QUIT" THEN 6190
6015 IF LEN(PP$)>1 THEN 6030
6020 PRINT "NAME TOO SHORT ":GOTO 6010
6030 IF PP$(1,1)=CP$(LEN(CP$),LEN(CP$)) THEN
6040
6035 PRINT "NO MATCH":GOTO 6010
6040 FOR I9=1 TO N0
6045 IF PP$=NAMES$(START(I9),FINISH(I9)) THEN
6100
6050 NEXT I9
6055 IF N0<100 THEN 6065
6060 PRINT "NO MORE ROOM FOR MORE NAMES":GOTO
6010
6065 N0=N0+1
6067 LONGSIZE=LEN(NAMES$):SIZE=LEN(PP$)
6068 IF LONGSIZE+SIZE>=2000 THEN 6060
6069 START(N0)=LONGSIZE+1
6071 FINISH(N0)=START(N0)+SIZE-1
```

```
6073 NAMES$(LEN(NAMES$)+1)=PP$:AVNAMES(NØ)=Ø
6080 GOTO 6190
6098 REM * "FOUND NAME"
6100 IF AVNAMES(I9)=1 THEN 6150
6110 PRINT "USED ALREADY":GOTO 6010
6150 AVNAMES(I9)=Ø
6190 RETURN
```

Program 6-8f. Person-response subroutine for Geography.

In the event that someone runs enough games to build the names array up to 100 names, line 6060 of this subroutine will display a message and request another name.

Finally, the computer response subroutine at line 5000 completes the program. We simply search the NAMES$ string (using the values in the START and FINISH arrays) for a place name that has the proper first letter and that has not been used in the latest game. If no such name is found, save the word "QUIT" in CP$. See Program 6-8g.

```
4998 REM * COMPUTER RESPONDS
5000 FOR I9=1 TO NØ
5005 LAST=LEN(PP$)
5010 IF NAMES$(START(I9),START(I9))=PP$(LAST,
LAST) AND AVNAMES(I9)=1 THEN POP :GOTO 5050
5015 NEXT I9
5020 PRINT :PRINT " I have run out of names!"
5025 CP$="QUIT"
5030 GOTO 5090
5050 CP$=NAMES$(START(I9),FINISH(I9))
:AVNAMES(I9)=Ø
5060 PRINT "    I CHOOSE: ";CP$
5090 RETURN
```

Program 6-8g. Computer-response subroutine for Geography.

Note the POP in line 5010. Whenever you have the possibility of leaving a loop before it has had a chance to be completed it is a good idea to use a POP statement. This clears a value that is not needed out of memory, since such values can accumulate in the course of a long program. Since our program is doing comparisons as it goes through this loop, there will be lots of opportunities to exit the loop before it has run its course.

The program does not verify that the names are actually legitimate place names. That is left to the honor of the player. This same program allows the player to change the rules of the game. We could just as well use people's names or a computer glossary. In that case, we would want to change the instructions and the DATA statements.

Notice that in the computer response subroutine at line 5000 the entire list is scanned for names. Since every name that is added to the list during the game is by definition not available for the remainder of this game, the program need not do

this. We could establish another variable to hold the number of names at the beginning of the current game.

We list the complete program here for your convenience.

```
5 REM * PLAY GEOGRAPHY
10 DIM NAME$(20),NAMES$(2000),START(100),
FINISH(100)
15 DIM AVNAMES(100),PP$(20),A$(3),CP$(20)
21 NAME$=""
25 GOSUB 8000:REM * READ NAMES ARRAY
30 GOSUB 9000:REM * INSTRUCTIONS
35 GOSUB 4000:REM * INITIALIZE AVAILABLE NAME
ARRAY
40 GOSUB 7000:REM * COMPUTER STARTS
50 GOSUB 6000:REM * PERSON RESPONDS
55 IF PP$="QUIT" THEN 75
60 GOSUB 5000:REM * RESPONSE OF COMPUTER
65 IF CP$<>"QUIT" THEN 50
75 PRINT "DO YOU WANT TO PLAY AGAIN (Y/N)";
80 INPUT A$
90 IF A$(1,1)="N" OR A$(1,1)="n" THEN END
120 GOTO 30
3998 REM * INITIALIZE AVAILABLE NAMES ARRAY
4000 FOR J9=1 TO N0
4010 AVNAMES(J9)=1
4020 NEXT J9
4090 RETURN
4998 REM * COMPUTER RESPONDS
5000 FOR I9=1 TO N0
5005 LAST=LEN(PP$)
5010 IF NAMES$(START(I9),START(I9))=PP$(LAST,
LAST) AND AVNAMES(I9)=1 THEN POP :GOTO 5050
5015 NEXT I9
5020 PRINT :PRINT " I have run out of names!"
5025 CP$="QUIT"
5030 GOTO 5090
5050 CP$=NAMES$(START(I9),FINISH(I9)):AVNAMES(I9)=0
5060 PRINT "    I CHOOSE: ";CP$
5090 RETURN
5998 REM * PERSON GO
6000 PRINT
6010 PRINT "    YOUR TURN: ";
6011 INPUT PP$
6012 IF PP$="QUIT" THEN 6190
6015 IF LEN(PP$)>1 THEN 6030
6020 PRINT "NAME TOO SHORT ":GOTO 6010
6030 IF PP$(1,1)=CP$(LEN(CP$),LEN(CP$)) THEN
6040
```

```
6035 PRINT "NO MATCH":GOTO 6010
6040 FOR I9=1 TO N0
6045 IF PP$=NAMES$(START(I9),FINISH(I9)) THEN
     6100
6050 NEXT I9
6055 IF N0<100 THEN 6065
6060 PRINT "NO MORE ROOM FOR MORE NAMES":GOTO
     6010
6065 N0=N0+1
6067 LONGSIZE=LEN(NAMES$):SIZE=LEN(PP$)
6068 IF LONGSIZE+SIZE>=2000 THEN 6060
6069 START(N0)=LONGSIZE+1
6071 FINISH(N0)=START(N0)+SIZE-1
6073 NAMES$(LEN(NAMES$)+1)=PP$:AVNAMES(N0)=0
6080 GOTO 6190
6098 REM * "FOUND NAME"
6100 IF AVNAMES(I9)=1 THEN 6150
6110 PRINT "USED ALREADY":GOTO 6010
6150 AVNAMES(I9)=0
6190 RETURN
6998 REM * COMPUTER BEGIN THE GAME
7000 X9=INT(RND(0)*(N0-1))+1
7028 CP$=NAMES$(START(X9),FINISH(X9))
7030 PRINT "FIRST PLACE : ";CP$:AVNAMES(X9)=0
7090 RETURN
7998 REM * READ NAMES
8000 I9=0
8010 READ NAME$
8011 LONGSIZE=LEN(NAMES$)
8012 I9=I9+1:START(I9)=LONGSIZE+1
8015 IF NAME$="DONE" THEN 8080
8017 SIZE=LEN(NAME$):FINISH(I9)=LONGSIZE+SIZE
8020 NAMES$(LEN(NAMES$)+1)=NAME$
8025 GOTO 8010
8080 N0=I9-1
8090 RETURN
8100 DATA NEW YORK,CHICAGO,PHILADELPHIA,BOSTON
8590 DATA DONE
8998 REM * INSTRUCTIONS
9000 PRINT "⅃This program will play a
     geography"
9005 PRINT "game with you. You will take
     turns"
9010 PRINT "with the computer. Each of you
     will"
9015 PRINT "be trying to think of names of
     places"
9020 PRINT "such that the first letter of
     your"
```

114

```
9025 PRINT "name is the same as the last
letter"
9030 PRINT "of the previously used place
name.":PRINT
9035 PRINT "PLEASE USE CAPITAL LETTERS ONLY.":
PRINT
9040 PRINT "ARE YOU READY (Y/N)"
9045 INPUT A$
9050 IF A$(1,1)<>"Y" AND A$(1,1)<>"y" THEN
9045
9060 PRINT "ō":REM CLEAR SCREEN WITH ESC CTRL+
CLEAR
9080 RETURN
```

Program 6-8. Play a Geography game.

...SUMMARY

We have seen that it is possible to simulate arrays of strings in a number of ways. All the methods have involved using one long string to store all the pieces one after the other. We have used blanks to force all the pieces to have the same length, so that each piece starts and ends at an easily calculated place in the long string. We have also used numeric arrays to store the beginning and ending position of each piece, in which case the pieces can be of any length. We have seen in the sample programs that it is possible to coordinate numeric values with string data to allow arbitrary pieces of the long string to be selected.

Problems for Section 6-3 ....

1. In Program 6-8, which plays Geography, notice that the loop beginning at line 5000 scans every name in the list. None of the names that have been added in this most recent game may be used by the computer, because they have all been used by the human player. Fix the program so that the computer scans only those names it "knows" at the start of the most recent game. (Suggestion: establish a new variable, N2, which represents the number of names at the beginning of the current game.) Don't be tempted to change line 6040.

2. Modify the computer response subroutine (Program 6-8g) so that the computer randomly selects a starting point in the names array. Be sure that if no name is found the computer scans from the beginning of the array to the random starting point.

115

3. Sometimes it is interesting simply to rearrange strings for display purposes. Write a program that enters the days of the week in a string array and displays them in the following format:

```
S   M   T   W   T   F   S
U   O   U   E   H   R   A
N   N   E   D   U   I   T
D   D   S   N   R   D   U
A   A   D   E   S   A   R
Y   Y   A   S   D   Y   D
        Y   D   A       A
            A   Y       Y
            Y
```

4. Write a program to enter a collection of names in a string array. Find the element that comes first alphabetically. Display it and its position in the array.

PROGRAMMER'S CORNER 6

...Sorting Numeric Arrays

There is often a need to put the items in a numeric array in ascending or descending order. Arrays are ready-made for sorting, since all the elements have easily addressable names. There are many methods for sorting; we will apply one of the simpler, easier-to-understand sorts to this problem. The sort we will use looks at each pair of values, beginning at the start of the array, and swaps their position in the array if the first is larger than the second (if we want to put the values in ascending order). It continues through the list, swapping as it goes until it reaches the end. At this point, the largest value has moved to the last position in the array. It then goes back to the start again and does the same thing, stopping at the next to the last value, since we already know that the largest value is the last value. This continues until the array is rearranged in ascending order. A similar method works for descending order, only here the swap occurs if the first value is less than the second. Let's see how it works.

```
10 REM * SORTING ROUTINE FOR ARRAYS
20 PRINT "(1) ASCENDING SORT"
30 PRINT "(2) DESCENDING SORT"
40 INPUT SORT:PRINT
90 REM * SORTING ROUTINE FOR ARRAYS
100 DIM A(100)
110 FOR I=1 TO 10
120 A(I)=INT(RND(0)*200)
130 NEXT I
140 FOR I=1 TO 10
```

```
150 PRINT A(I);" ";
160 NEXT I
170 PRINT
190 N=10
200 FOR I=1 TO N-1
210 IF SORT=2 AND A(I)>=A(I+1) THEN 250
215 IF SORT=1 AND A(I)<=A(I+1) THEN 250
220 TEMP=A(I)
230 A(I)=A(I+1)
240 A(I+1)=TEMP
250 NEXT I
256 FOR I=1 TO 10
257 PRINT A(I);" ";
258 NEXT I
259 PRINT
260 N=N-1
270 IF N>0 THEN 200
300 END
```

Program 6-9. Sorting routine for arrays.

```
RUN
(1) ASCENDING SORT
(2) DESCENDING SORT
?1

104 55 11 103 33 85 70 111 121 1
55 11 103 33 85 70 104 111 1 121
11 55 33 85 70 103 104 1 111 121
11 33 55 70 85 103 1 104 111 121
11 33 55 70 85 1 103 104 111 121
11 33 55 70 1 85 103 104 111 121
11 33 55 1 70 85 103 104 111 121
11 33 1 55 70 85 103 104 111 121
11 1 33 55 70 85 103 104 111 121
1 11 33 55 70 85 103 104 111 121
1 11 33 55 70 85 103 104 111 121

READY
```

Figure 6-9. Execution of Program 6-9.

We began by asking the user to choose an ascending or descending sort. We then filled our array with random numbers (lines 110–130) and printed them out. We then proceeded to sort, the order depending on the user's choice. To show the process, we had the program print out the current order after each pass through the array. Note the N counter, which we used as our moving marker to indicate how many elements we had to compare each time through the array. Although this sort is reasonably fast for our array of ten elements, it becomes quite slow as the array becomes large. Try it with 100 elements (change lines 110, 140, 190, and 256) and

you'll see what we mean. Wouldn't it be a good idea to rewrite the program, using a variable for the loop length to allow the program to be easily changed?

If you run this program several times, you'll notice that on many occasions the sorting is complete several passes before the program finishes. We could put in a statement to check whether any swaps took place at all on a given pass and then end if none occurred, since we would have been finished at that point.

...The Built-In Atari Clock

With three simple PEEKs, the programmer can access a timer that is built into Atari computers. Memory locations 18, 19, and 20 represent a useful timer. Location 20 increments by 1 every sixtieth of a second until it reaches 255; it then increments location 19 by 1 and resets itself to 0. When location 19 reaches 255 it increments location 18 by one and resets itself to 0. This permits us to time intervals from a sixtieth of a second to more than 80 hours. We simply use POKE statements to enter zeros into these three locations to reset the timer, and then have the program periodically check the time until it reaches some predetermined value. And there we have a timer.

```
1 REM * A ONE MINUTE TIMER
3 PRINT "}ATARI ONE MINUTE TIMER"
4 PRINT
5 PRINT "Ready...Set...GO!":PRINT
6 N=10
7 POKE 18,0:POKE 19,0:POKE 20,0
10 TIME=INT((PEEK(18)*65536+PEEK(19)*256+
PEEK(20))/60)
15 IF TIME=60 THEN 100
20 IF TIME/10=INT(TIME/10) AND TIME=N THEN
PRINT CHR$(253);TIME;"SECONDS HAVE PASSED."
:N=N+10
30 GOTO 10
100 PRINT "ONE MINUTE IS UP!"
105 PRINT CHR$(253)
107 PRINT CHR$(253)
110 END
```

Program 6-10. Atari clock demonstration.

```
RUN
ATARI ONE MINUTE TIMER

Ready...Set...GO!

10 SECONDS HAVE PASSED.
20 SECONDS HAVE PASSED.
30 SECONDS HAVE PASSED.
40 SECONDS HAVE PASSED.
```

118

```
50 SECONDS HAVE PASSED.
ONE MINUTE IS UP!
```

READY

Figure 6-10. Execution of Program 6-10.

Notice the use of CHR$(253) in lines 20, 105, and 107. This causes the computer's speaker to "beep" as a reminder that a ten-second interval has elapsed.

Line 10 is our timer, which increments TIME by 1 every second. We have used the value N to check when we reach multiples of ten seconds and then print out our messages. If we did not use N, our message would have been printed several times, since the loop will check TIME more than once per second.

You may use this sort of timer for delay loops in programs if it is important that the delay time be some fixed value.

Delay loops of the form

```
100 FOR I=1 TO 1000:NEXT I
```

produce different delays depending on where they are in a program and how long the program is.

...The START, SELECT, and OPTION Keys

As any good Atari owner knows, these keys are very useful in many of the commercial games that are on the market. That is, after all, what they were put there for in the first place. Programs other than games may also make use of these keys. Memory location 53279 holds a value that indicates which key or combination of keys has been pressed. Table 6-1 shows the value returned for each combination of keys pressed.

KEY(S) PRESSED	VALUE RETURNED							
	0	1	2	3	4	5	6	7
Option	X	X	X	X				
Select	X	X			X	X		
Start	X		X		X		X	

Table 6-1. Keys pressed and values returned.

A 0 in location 53279 means that all keys have been pressed, a 1 indicates that the OPTION and SELECT keys have been pressed, and so forth. If we use a POKE statement to enter a value of 7 into this location we clear it. These keys can be used to accept responses from the user in any BASIC program. Here's a program that checks to see which of these keys (single or combinations) have been pressed.

```
80 REM * TEST THE CONSOLE KEYS
83 DIM BLANK$(38)
86 BLANK$="
   ":REM 37 SPACES
88 POKE 752,1
90 PRINT "⅃":REM ESC CTRL+CLEAR
```

```
91 POSITION 2,5
92 PRINT "OK, PRESS ANY COMBINATION OF KEYS"
93 X=7
94 POKE 53279,7
95 FOR CHECK=1 TO 30
96 KEY=PEEK(53279)
97 IF KEY<X THEN X=KEY
98 NEXT CHECK
100 POSITION 2,10
110 PRINT BLANK$
115 POSITION 2,10
120 IF X=0 THEN ? "YOU PRESSED OPTION, SELECT
AND START"
130 IF X=1 THEN ? "YOU PRESSED OPTION AND
SELECT"
140 IF X=2 THEN ? "YOU PRESSED OPTION AND
START"
150 IF X=3 THEN ? "YOU PRESSED THE OPTION KEY
ONLY"
160 IF X=4 THEN ? "YOU PRESSED SELECT AND
START"
170 IF X=5 THEN ? "YOU PRESSED THE SELECT KEY
ONLY"
180 IF X=6 THEN ? "YOU PRESSED THE START KEY
ONLY"
190 IF X=7 THEN ? "YOU PRESSED NO KEYS YET!"
200 PRINT
220 GOTO 91
```

Program 6-11. Reading the START, SELECT, and OPTION keys.

It is difficult to press more than one key evenly so that both make contact at the same time. Also, the computer takes very little time to read a memory location. If we only check once for what is pressed, we will usually get only the first key that makes contact. Our program makes use of the fact that combinations of keys give lower response values. We begin by assuming that no keys are pressed (X = 7). We then check 30 times, storing only the lowest value we find in these 30 checks. You can change line 95 to make as many checks as you want. You will find, however, that values much smaller than 30 give incorrect responses when you press more than one key, while much larger values make for a slow response.

Chapter 7

Using What We Know: Miscellaneous Applications

7-1...Looking at Integers One Digit at a Time

In general, the more detailed the control we have over a number in the computer, the more complex the problems we can expect to be able to handle. We also will find that as we learn more about what goes on inside the computer, we will be able to apply more elegant solutions to problems. It is common to store a different piece of information in each digit of a number. It is also common to group digits in twos or threes for this purpose. Part numbers, serial numbers, and course numbers are just a few examples of this. In this section we will develop methods of breaking up numeric values into their separate digits.

...Using Successive Division in Atari BASIC

Consider the number 2789. The 2 means 2000, which may be written $2 * 10^3$; the 7 means 700, which may be written $7 * 10^2$; the 8 means eight tens, which may be written $8 * 10^1$; and the 9 means nine units, which may be written $9 * 10^0$. Looking at the numbers step by step,

$$2789 = 2 * 10^3 + 789$$
$$789 = 7 * 10^2 + 89$$
$$89 = 8 * 10^1 + 9$$
$$9 = 9 * 10^0 + 0$$

This is an example of the general relationship

$$N = I * 10 ^ E + R$$

Compare this statement to what we have done above and you will see that the equations match—the variable I is each digit of our number, E is the power of ten, and R is our "remainder." If we restate this general equation in BASIC, we get

$$I = INT(N/10 \wedge E)$$

where I is the integer factor. If we solve the original equation for R we get

$$R = N - I * 10 \wedge E$$

Carefully study Program 7-1.

```
100 PRINT "INPUT AN INTEGER";
110 INPUT N
115 IF N>999999999 THEN PRINT "TOO BIG!":GOTO
100
120 IF N=0 THEN END
130 FOR E=9 TO 0 STEP -1
140 T=10^E
150 I=INT(N/T)
160 PRINT I;"   ";
170 R=N-I*T
180 N=R
190 NEXT E
200 PRINT :PRINT
210 GOTO 100
```

Program 7-1. Access digits by successive division.

Note that in each case the new N is the old R and the value of E is decreased by 1 for each cycle (or iteration). For ten-digit integers the value of E will have to begin at 9 and go to 0. A FOR . . . NEXT loop with STEP −1 will do this. Note line 140. In that line we simply save the value of 10∧E. Exponentiation is a slow process, and there is no need to have the computer do it twice for each value of E.

```
RUN
INPUT AN INTEGER?123456789
0   1   2   3   4   5   6   7   8   9

INPUT AN INTEGER?999
0   0   0   0   0   0   0   9   9   9

INPUT AN INTEGER?0

READY
```

Figure 7-1. Execution of Program 7-1.

...Using STR$

A very easy method for accessing the individual digits of a number is to use the STR$ function. Once we store a number as a string, we can use our ability to access individual digits with VALUE$(I,I), where I ranges from 1 to however long the

number is, to pick a number apart as we see fit. It becomes very easy to pick out any starting point and any number of digits. We can scan the number to look for a decimal. We can use the LEN function to find how many characters it takes to display the number. For demonstration purposes, let's write a little program to display each digit of a number individually.

```
90 DIM A$(20)
100 PRINT "ENTER A NUMBER"
110 INPUT N
120 IF N=0 THEN END
130 A$=STR$(N)
140 FOR I=1 TO LEN(A$)
150 PRINT A$(I,I);" ";
160 NEXT I
170 PRINT :PRINT
180 GOTO 100
```

Program 7-2. Using STR$ to separate numeric digits.

```
RUN
ENTER A NUMBER
?695.43147
6 9 5 . 4 3 1 4 7

ENTER A NUMBER
?147896325523698741
1 . 4 7 8 9 6 3 2 5 5 E + 1 7

ENTER A NUMBER
?0

READY
```

Figure 7-2. Execution of Program 7-2.

Note the second number entered in the run of Program 7-2. Since it is represented in exponential notation for display purposes, that is the format used by STR$( ). In the case of decimal numbers and exponential notation we will have to construct more logic to determine the actual numeric value represented by a particular digit according to its position in the number.

...SUMMARY

We have seen two methods for picking apart numbers digit by digit in a computer. We have explored the use of successive division and the use of the STR$ function to do this. Successive division gave some insight into how numbers are constructed. Using the STR$ function proved to be a much easier way to easily access any individual digit in any order.

Problems for Section 7-1........................

1. Modify Program 7-1 so that leading zeros are not displayed. Be careful that you don't eliminate all zeros!

2. Write a program to construct an integer by reversing the digits of an entered integer. Place the result in a numeric variable and PRINT its value.

3. Find all three-digit integers that are prime. Form new integers by reversing the digits and see if the new number also is prime. Print a number only if it and its reverse number are prime. There are 43 pairs of numbers, some of which appear twice.

4. Do Problem 3, but eliminate duplicates.

7-2...Number Bases

The day-to-day world of business, commerce, and general communications reckons in the familiar base-ten number system. The ultimate reckoning of the computer, however, is in base two. Base two requires only the two digits 0 and 1. Computers may represent a 1 with a positive voltage level or a magnetized state and a 0 with a zero voltage level or a demagnetized state.

Because computers work only in base two, it is useful to be familiar with this number system. The base-two number system is also referred to as the *binary number system*. Always keep in mind that a number is a number is a number. The number does not change by virtue of being expressed in a different system. As we change from one base to another, we are using different symbols to name the same numbers. In the binary number system, there are only two possible digits.

As we learned in the previous section, numbers in the base-ten system are constructed from successive powers of ten. Thus 1234 is represented as $1*10^3 + 2*10^2 + 3*10^1 + 4*10^0$. Similarly, in the base-two system, numbers are represented as successive powers of two. Thus 10011010010 converts, in the base-ten system, to

$$
\begin{array}{rcr}
1 * 2^{10} & = & 1024 \\
+0 * 2^9 & = & 0 \\
+0 * 2^8 & = & 0 \\
+1 * 2^7 & = & 128 \\
+1 * 2^6 & = & 64 \\
+0 * 2^5 & = & 0 \\
+1 * 2^4 & = & 16 \\
+0 * 2^3 & = & 0 \\
+0 * 2^2 & = & 0 \\
+1 * 2^1 & = & 2 \\
+0 * 2^0 & = & 0 \\
\end{array}
$$

Add it all up and you will get 1234, the same number we represented above in base-ten form.

Addition in base two is very simple. Either there is a "carry" as the result of two 1s being added or there is not. Thus:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 1 = 10$

Multiplication is also simplified by having only two possible digits. When you multiply by 1, the digits shift according to the position of the 1; when you multiply by 0 the result is 0. When you multiply by 1 in the rightmost position, the shift is 0. When you multiply by 1 in the second position from the right, the shift is 1. If we choose to number the positions from right to left as 0, 1, 2, 3, . . . N, then the shift equals the position of the 1.

$1 * 101001 = 101001$	(shift of 0)
$10 * 101001 = 1010010$	(shift of 1)

and

$1000 * 101001 = 101001000$	(shift of 3)

Thus:

```
   10          11011
 * 10       *    101
  100          11011
               00000
              11011
            10000111
```

Note that in the second multiplication example there is a carry across several positions when the intermediate products are added.

One disadvantage of the binary number system is that it takes so many digits to represent numbers. For instance, 15 in base ten is written 1111 in binary, and 127 in base ten is written 1111111 in binary. However, this is a disadvantage only to humans. Computers are not fazed by this. In fact, the computer is very good at accessing individual bits and turning them on or off one at a time. The number 255 in base ten is written 11111111 in binary. It requires eight binary digits to represent the number 255. Each *binary digit* is referred to as a *bit*. Bits are collected into groups of eight to form *bytes*. Atari computers are eight-bit machines—that is, they use electronic circuits in sets of eight to represent numbers and instructions in memory. Everything the computer does is stored in a byte or a group of bytes. This is why a number of the limits for Ataris are 255, one less than 2^8.

It is useful to always keep in mind that each digit of any integer represents an integral power of the base. So the digits in binary represent

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, etc.

in base ten, corresponding to bit positions

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, etc.

in binary.

...Decimal to Binary

Let's get started by writing a program to convert decimal to binary. If the base-ten number we have is odd, then the first base-two digit on the right is a 1. If we have an even base-ten number, then the first base-two digit on the right is a 0. Now, to move the base-two decimal point one space to the left, we divide our base-ten number by two and ignore the decimal part. We can ignore the decimal part by simply chopping it off. This process for eliminating the decimal part of a number is called *truncation*. If the truncated result is 0, then we are finished. If the truncated result is not 0, then we repeat the process for the next binary digit. Consider the process for 53:

	53	is odd	1
divide by 2 and truncate	26	is even	01
divide by 2 and truncate	13	is odd	101
divide by 2 and truncate	6	is even	0101
divide by 2 and truncate	3	is odd	10101
divide by 2 and truncate	1	is odd	110101
divide by 2 and truncate	0		

We have finished and 53 in base ten = 110101 in base two.

Now we simply need to work out a way to print the results, and a program will be forthcoming. The method we use is to store the digits in a 16-element array as we determine them. We store the rightmost (or low-order) digit in the 16th element, the 2nd digit in the 15th element, and so forth until finished. Later this can be expanded to accommodate larger numbers.

We will store the digits in an array. For any base-ten number, if division by two comes out even, then the corresponding base-two digit is 0. If division by two leaves a decimal portion, then the corresponding base-two digit is 1. This we can easily do with 2 lines of code:

```
310 IF I/2=INT(I/2) THEN A(J)=Ø
320 IF I/2<>INT(I/2) THEN A(J)=1
```

Line 310 enters a 0 in the Jth element if the integer is evenly divisible by two and line 320 enters a 1 in the Jth element if the integer is not evenly divisible by two. Examine Program 7-3.

```
100 REM * CONVERT DECIMAL TO BINARY
110 DIM A(16)
200 PRINT "ENTER AN INTEGER";
205 INPUT I
210 IF I<=Ø THEN 999
220 IF I<655536 THEN 300
230 PRINT "TOO LARGE":PRINT :GOTO 100
```

```
298 REM * LOAD THE ARRAY
300 FOR J=16 TO 1 STEP -1
310 IF I/2=INT(I/2) THEN A(J)=0
320 IF I/2<>INT(I/2) THEN A(J)=1
340 I=INT(I/2)
360 NEXT J
398 REM * DISPLAY RESULTS
400 FOR J=1 TO 16
410 PRINT A(J);" ";
420 NEXT J
455 PRINT :PRINT
460 GOTO 200
999 END
```

Program 7-3. Decimal to binary using successive division.

Our program checks to see that the number entered is between 0 and 65536 and then proceeds to do the conversion. You might want to add a check for accidental string input (which will crash the program) with a TRAP statement. We leave this as an exercise. Finally, the program displays the digits that have been stored in the array.

```
RUN
ENTER AN INTEGER?127
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1

ENTER AN INTEGER?32512
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

ENTER AN INTEGER?0

READY
```

Figure 7-3. Execution of Program 7-3.

...Binary to Hexadecimal

The hexadecimal number system, or *hex* for short, reckons in base sixteen, using 16 possible digits. The hex digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So 10 in hex is 16 in base ten, and EF in hex is 14*16 + 15*1 or 239 in base ten. Whereas the place values for base-ten representation are 1, 10, 100, 1000, and so on, and the place values for binary representation are 1, 2, 4, 8, and so on, the place values for hexadecimal representation are 1, 16, 256, 4096, and so on. (Note that in all numbering systems the place values are really 1, 10, 100, and 1000, when expressed in the notation of the numbering system itself. 1, 10, 100, and 1000 in hex are written as 1, 16, 256, and 4096 in base-ten notation.) It takes four binary digits to form a hex digit:

```
1011    0001    in binary
  B       1   = B1 in hex
```

So, two hexadecimal digits may be used to represent any number stored in one byte, and four hexadecimal digits may represent two bytes. This is very convenient for use with an eight-bit machine.

The hexadecimal numbering system offers some advantages when working with a computer. B1 is more compact and much easier to read than 10110001. There are some parameters associated with computers that are just plain easier to remember in hex than in base ten. Computer memory is often blocked off in segments containing 16384 bytes each. That is 16 times 1024, or 4000 bytes in hex. One common unit of measure for computer memory is the kilobyte, abbreviated K. One K is 1024 bytes. So, for a 64K machine, the four 16K segments begin at 0000H, 4000H, 8000H, and C000H, where the H signifies that the number is in hex notation. Those numbers are much easier to remember than 0, 16384, 32768, and 49152.

...Hexadecimal to Decimal

The conversion from decimal to hex is exactly analogous to the conversion from decimal to binary, except that we have to work out how to get the extra digits A through F into the picture. Since the extra-digit problem also occurs in the hex-to-decimal conversion, this is where we start.

Let's convert 1B3A hex to decimal:

The digit A in the 1's column represents	10
The digit 3 in the 16's column represents	48
The digit B in the 256's column represents	2816
The digit 1 in the 4096's column represents	4096
	6970

Thus, 1B3A in hex equals 6970 in base ten.

To work in hex, our programs must have a way to accept hex input and display hex output. Obviously this cannot be done with numeric variables. We may store the 16 hex digits in a string variable. All hex input should be checked to verify that no bogus digits have been entered. Let's start by writing a program that simply requests hex input, verifies it, and displays the verified number.

```
100 REM * DEVELOP HEX INPUT/OUTPUT
110 DIM H$(16),N$(10)
130 H$="0123456789ABCDEF"
140 GOSUB 400:REM *REQUEST & VERIFY
150 PRINT N$
190 GOTO 140
398 REM * REQUEST & CALL VERIFY
400 PRINT :PRINT "HEX NUMBER";:INPUT N$
410 L=LEN(N$)
420 IF L=0 THEN END
430 IF L<5 THEN 440
432 PRINT "TOO MANY DIGITS!"
434 GOTO 400
440 GOSUB 700
```

```
450 IF FLAG=0 THEN 490
460 PRINT "BAD FORMAT!":GOTO 400
490 RETURN
698 REM * VERIFY HEX STRING
700 FLAG=0:REM * GOOD INPUT
710 FOR J=1 TO L
720 FOR K=1 TO 16
730 IF H$(K,K)=N$(J,J) THEN 760
740 NEXT K
750 FLAG=1:REM * BAD INPUT
755 GOTO 790
760 NEXT J
790 RETURN
```

Program 7-4. Hex input/output.

```
RUN

HEX NUMBER?ABCD
ABCD

HEX NUMBER?AFAF
AFAF

HEX NUMBER?HEX
BAD FORMAT!

HEX NUMBER?FF
FF

HEX NUMBER?0
0

HEX NUMBER?

READY
```

Figure 7-4. Execution of Program 7-4.

Now, how do we get the computer to "know" that an A is 10, a B is 11, and so on? Since the digits are not numeric, we have this problem even for 0, 1, and the other base-ten digits as well.

This problem is not so tough as it might seem at first. Line 730 of Program 7-4 gives us all the information we need. The value of K there tells us which digit in the sample string H$ matches the Jth digit of the input string. If $K = 1$ then the digit in H$ is a 0; if $K = 16$ then we come up with F. So, subtracting 1 from K gives us the values from 0 to F corresponding to 0 to 15. Then, knowing which digit we are on tells us which place that digit represents, so we know what power of 16 to use.

The digit value is $K - 1$. The place is $L - J$. So the base-ten value is

$$(K - 1) * 16 \wedge (L - J)$$

129

We simply need a numeric variable in which to accumulate this information. Using this information, the subroutine at line 700 could easily return the base-ten value of the hex input. Simply set a numeric variable to 0 at about line 705 and accumulate at line 760, while moving NEXT J to line 770. This is left as an exercise.

...SUMMARY

We have seen that the rationale for base two or binary is that the digits 0 and 1 can be represented as electrical states of one sort or another. The hexadecimal number system is convenient because it correlates so nicely with data as they are stored in computer memory. While it takes eight digits to represent a byte of computer memory in binary, it requires only two hexadecimal digits. All conversion techniques rely upon determining the position of a particular digit and its actual value.

Problems for Section 7-2. . . . .

1. Write a program to convert hex to binary.
2. Modify Program 7-3 to eliminate leading zeros and display the result with no spaces.
3. Modify Program 7-4 to do the conversion as described in this section (hex to decimal).

7-3...Miscellaneous Problems for Computer Solution

We offer a few interesting problems for computer application here. Do not limit yourself to the problems suggested. You should be bringing your own problems to the computer by now. Although it is important to include suggested problems in any book, you will find that a tremendous satisfaction comes from developing your own ideas on the computer.

...Problems of General Interest

1. There is an old number puzzle about cows, pigs, and chickens that lends itself nicely to computer solution. A farmer has exactly $100 to spend on animals. He wants to buy at least one cow, at least one pig, and at least one chicken. Cows are $10 each, pigs are $3 each, and chickens are $.50 each. How many of each must he buy to have exactly 100 animals?

At first, this looks like an easy algebra problem. One soon finds that we have only two equations with which to solve a problem having three unknowns. This is where the computer comes in. We simply try all combinations of cows, pigs, and chickens until these equations are satisfied:

$$10 * CO + 3 * PI + CH / 2 = 100$$
$$CO + PI + CH = 100$$

By observing that there must be many more chickens than either pigs or cows we could solve this by hand, using trial and error. But we still might become frustrated with the number of calculations required.

The key to this problem is to realize that each of the three numbers we are looking for must be an integer. We could easily write a program with three nested FOR . . . NEXT loops where CO goes from 1 to 10, PI goes from 1 to 33, and CH goes from 2 to 100 by twos. If we do that, we will find that the program has to "think" for some time. We can greatly speed things up by using more of the information available to us. Clearly, if there must be at least one of each animal, there cannot be ten cows or 33 pigs or 100 chickens. There could be no more than nine cows, no more than 29 pigs, and no more than 98 chickens. We can derive the greatest speed improvement by using the fact that once the number of pigs and cows to try has been established, we can find the number of chickens from

$$100 - CO - PI$$

Next we check this number to see that it is even, since the price per chicken is \$0.50. We can test to see if CH divided by 2 in an integer. Write a program to solve this puzzle.

2. Sometimes it is fun to try to guess a number that someone else is thinking of. It is fairly easy to program a computer to play this simple game. Have the computer request the largest number from the user. Then the program should compute a random number in the range from 1 to the largest number. Next the program should ask for guesses from the user. Each guess should be checked. If the number is less than 1 or greater than the upper limit a message should put the user back on the right track. If the number is a correct guess, the program should say so and display the number of guesses required to get it right. The program should also note whether the actual number is higher or lower than the most recent guess.

3. There are many famous chess puzzles that are appropriate for computer solution. A notable one is the eight-queens problem. In how many ways can eight queens be placed on a chessboard so that no queen attacks another?

This puzzle may be solved by using one eight-element array. Placing a queen in a position of the array assures that no two queens occupy the same row. A queen may be placed in the row by entering its column number there. Now we assure that no two queens occupy the same column by avoiding duplicate column numbers in the eight-element array. Finally we check for diagonal attack by noting that for two queens at positions (X,Y) and (X',Y'), one diagonal is shared if $X - X' = Y - Y'$, while the other diagonal is shared if $X + X' = Y + Y'$. We need to have the computer test this for each queen in every column of one row. Write a program to print the positions of all queens for each solution. Note: Your solution program may take a long time to produce results.

131

For more about the eight-queens problem see the October 1978 and February 1979 issues of *BYTE* magazine.

4. It is always instructive to learn about the cost of homeownership. Aside from the ongoing costs of painting, fixing the roof, real estate taxes, and insurance, there is the ever-present mortgage interest. Most mortgages are set up so that the monthly payment stays constant. In the beginning, there is a large interest payment and a small payment toward the principal. At the end, the interest payment is small and more goes toward the principal. The following formula may be used to calculate the monthly payment PA:

$$PA = P\frac{I(1 + I)^N}{(1 + I)^N - 1}$$

where:

P   is the principal
I   is the monthly interest rate
N   is the number of months

Write a program to request the principal, annual interest rate, and number of years. Have the program display the monthly payment, the total amount paid, and the total interest paid.

...Math-Oriented Problems

1. Every positive integer may be expressed as the sum of the squares of four integers. Zero may be included as one or more of those integers to be squared. For example:

$$1 = 0^2 + 0^2 + 0^2 + 1^2$$

Write a program to find all sets of four such integers for a requested integer. Be careful about efficiency in this one. Test your solution with small integers before trying large ones!

2. Suppose you have to find the greatest common factor of 23902 and 15096. What would you do? The famous mathematician Euclid would have found the remainder after dividing 23902 by 15096, which is 8806. Then he would have found the remainder after dividing 15096 by 8806, which is 6290. Then he would have continued this pattern as follows:

$$23902 = 1 * 15096 + 8806$$
$$15096 = 1 * 8806 + 6290$$
$$8806 = 1 * 6290 + 2516$$
$$6290 = 1 * 2516 + 1258$$
$$2516 = 2 * 1258 + 0$$

Next, Euclid would have reasoned that since the remainder of the last

division was 0, the greatest common factor must be the last divisor, in this case 1258. This method required only five iterations. How many would it have taken using other methods?

3. The sieve of Eratosthenes is an ingenious method for generating prime integers. Write down all the integers from two to the desired upper limit. Now keep the first number and cross out all its multiples. Now keep the next number that has not been crossed out and cross out all its multiples. Repeat this process until there are no more numbers to cross out. The remaining numbers are prime.

There are two areas in this algorithm that are pitfalls for unnecessary extra processing. First, if the first multiple in any case has already been crossed out, then all other multiples will also have been crossed out. Second, we only have to check for integers that have not been crossed out up to the square root of the largest number in the original range.

This algorithm can easily be implemented in an array. First, enter the integers from two to the upper limit into the array elements two through the upper limit. Next, use a FOR . . . NEXT loop to access the multiple positions in the array. Set the contents of any element to be crossed out to zero. Finally, PRINT all subscript positions for which the element is not zero.

4. A perfect number is an integer the sum of whose proper factors is the integer itself. The proper factors of 15 are 1, 3, and 5. The sum of the factors of 15 is 9. Therefore 15 is not a perfect number. The proper factors of 6 are 1, 2, and 3. The sum of the proper factors of 6 is 6. Thus 6 is called a perfect number. Write a program to find the first four perfect numbers. Since the fifth perfect number is 33,550,336, and there is a significant amount of execution associated with determining "perfectness," we would be unwise to test each integer up to that one! Even the first four will take some time to find in BASIC. It turns out that there don't seem to be any odd perfect numbers, so let's test only even numbers.

5. Euclid was an active mathematician! He concluded that all possible even perfect numbers are of the form

$$N = 2^{(E-1)} * F$$

where

$$F = 2^E - 1$$

and F is an odd prime.

Using Euclid's algorithm, write a program to calculate perfect numbers. Try a range of 2 to 15 for E.

6. Pythagorean triples are sets of three integers that can be the sides of a right triangle. By the Pythagorean theorem, the sum of the squares of the two smaller integers equals the square of the largest one. The first Pythagorean triple is 3, 4, 5. Write a program to generate Pythagorean triples.

7. The number π has fascinated mathematicians for many centuries. Values for π may be calculated in a variety of ways. The following sequence is known to approach the value of π:

$$4(1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 \ldots)$$

Write a program to evaluate this sequence for several large numbers of terms.

8. There are many sequences that approach π as the number of terms increases. The following is another one:

$$2 + 16 \left[\frac{1}{1 * 3 * 5} + \frac{1}{5 * 7 * 9} + \frac{1}{9 * 11 * 13} + \cdots \right]$$

Write a program for this sequence. If you also did Problem 7, which sequence converges faster? ("Converges" means, "approaches the correct value.")

9. One method for approximating the value of π derives from the fact that the area of a circle is known to equal πr^2 where r is the radius of the circle. A circle having a radius of 1 has an area of π. Thus, if we inscribe a circle in a square having a side of 2 and examine one quarter of the figure, we have a quarter circle inscribed in a square with side of 1.
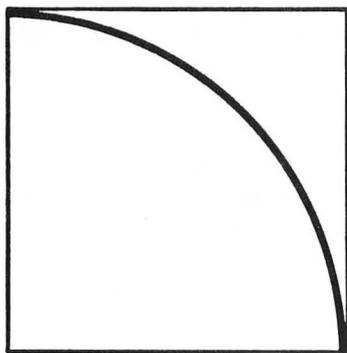
The area of the square is 1 and the area of the quarter circle is $\pi/4$. If we have some way of measuring the area of the quarter circle, then we simply multiply that number by 4 to get an approximation of π.

If we generate random values between 0 and 1 for X and Y, we will always get a point in the square. Sometimes we will get a point in the circle. The ratio of the number of times the point falls in the circle to the number of points selected is proportional to the areas of the quarter circle and the square. If we got 80 points in the circle out of 100 points selected, then the approximation of π we come up with a 4 * (80/100) or 3.2. Write a program to calculate π in this way for 100, 500, 1000, and 5000 random points. Assume that if a point lands on the circle then it counts as part of the circle.

You might experiment to see whether or not excluding points that fall on the circle has an impact on how fast the value you get approaches the known value of π ($\pi = 3.1415926536$ . . .).

PROGRAMMER'S CORNER 7

...Writing a Program Menu

It is common practice where programs are run on a fast video display to present the user with a list of options. Usually the options are numbered and the user simply enters the number of the preferred option, followed by a carriage return.

A sample menu might look like this:

```
1)  PLAY TIC TAC TOE

2)  SUPER LO-RES DEMO

3)  QUIT
```

Figure 7-5. Sample menu.

Note option 3. This is very important. By providing this option right in the menu we give the user proper control of the program. Many of the programs for sale in computer stores and by mail order are "menu-driven"—that is, they have a menu. The quality of menus is not consistent. One of the most common problems is the failure to include an option to terminate the program. With such programs, we have to press BREAK or SYSTEM RESET to exit. Sometimes we even have to shut the computer off and back on again to run other programs. In some cases this is done to make it difficult for the user to make unauthorized copies of a program.

Another common affliction is that entering something not in the list of choices produces a messy display. In some cases the menu even begins to disappear from the screen if we enter several out-of-range choices. With some programs, pressing an extra key before the menu even appears on the screen produces surprising results. We will endeavor to write a menu program that avoids all of these problems.

...Developing the Menu Routine

Each of the options in the menu may be a subroutine in a single program, or each may be a separate program. That doesn't matter much. What we want to do here is develop a good menu-processing routine.

First of all, the screen should be cleared with GRAPHICS 0, so we have a full, blank screen on which to display our menu.

Next, we should give some thought to how to take the choices from the keyboard. We can use an INPUT request or we may use a technique that processes the keystrokes directly, with no need to hit the RETURN key.

135

INPUT is quick and easy to code. However, if we code

```
940 INPUT X
```

and the user enters anything other than a numeric value, BASIC will stop the program and display an error message. We can take care of this possibility with a TRAP statement to catch errors and request that the input be reentered. We could also use

```
940 INPUT X$
```

and then convert the response to a numeric value with the VAL function. This nicely handles the situation where the user fails to enter a numeric response. In any program where we can expect the user to know enough to press the RETURN key this is a good method to use.

...GET, OPEN, and CLOSE

However, the ultimate in control comes if we read the keyboard directly. We can do this with the GET statement. This can be used to "get" one character from the keyboard at a time. Actually, the value gotten is the ATASCII value for the character, which will be a number from 0 to 255. In order to use GET, we must first open a channel to the keyboard that will be used by GET as a path for obtaining the value. We do this with

```
122 OPEN #N,A,B,"device"
```

where N refers to the channel number, A is the type of action to be taken, B is usually 0 but must be in the expression, and "device" is the character that represents what part of the system we want to communicate with. N can be a number from 0 to 7, but 0, 6, and 7 are used by the computer for special purposes and ought to be avoided. We can always make use of any number from 1 to 5 and should restrict our use to these values. Generally A is 4 for input only, 8 for output only, and 12 for input and output. For example, we would use 8 with a printer, since you only output to a printer. Similarly, if we want to communicate with the keyboard we use 4, since you only get input from a keyboard. The "device" code for the keyboard is "K:" and so our statement becomes

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

The companion statement to OPEN is

```
125 CLOSE #1
```

When you OPEN a channel, it remains open until you CLOSE it or until a program terminates with an END statement or by running out of statements. You should be careful, however, of programs that are terminated with STOP or by using the BREAK key, because they do not close any channels that are open.

There is another precaution you should take when using OPEN statements in a program. If we come back to the OPEN statement during the program, we will get ERROR —129, the code that indicates that the channel is already open. If you

attempt to open a channel that is already open you will always get an error message. One way to get around this is to put the OPEN statement up near the beginning of your program and never execute it again. A much better way is to always precede any OPEN statement with a CLOSE statement, to make sure that the channel is available. In our program we will do this and also take the extra precaution of using OPEN just before we need it and CLOSE as soon as we are done with the keyboard input. In a complex program, these actions will prevent a lot of frustration as errors are debugged.

Once we have written our menu routine, we can plan future programs that use it. All we have to change will be the names of the options and the control routine. Program 7-5 does it all.

```
90 DIM A$(30),N(10)
98 REM * TEST THE MENU SUBROUTINES
100 GOSUB 9400
110 GOSUB 9000
120 IF CHOICE<>N0 THEN 130
121 ? "↑↑  ARE YOU SURE (Y/N)?";
122 CLOSE #1:OPEN #1,4,0,"K:"
124 GET #1,X
125 CLOSE #1
126 IF CHR$(X)="Y" OR CHR$(X)="y" THEN
POKE 752,0:END
128 GOTO 110
130 PRINT "YOU CHOOSE - ";
133 GOSUB 10000+CHOICE*10
135 PRINT A$
140 FOR I=1 TO 1000:NEXT I
150 GOTO 110
8994 REM
8996 REM * "DO THE MENU"
8998 REM * RETURN THE SELECTION IN S
9000 GRAPHICS 0
9001 POKE 752,1
9002 POKE 85,10
9003 PRINT "* * M E N U * *"
9004 PRINT
9006 FOR I=1 TO N0
9008 PRINT I;")   ";
9010 GOSUB 10000+I*10
9014 PRINT A$
9018 PRINT
9019 NEXT I
9020 PRINT "YOUR CHOICE: ";
9025 CLOSE #1:OPEN #1,4,0,"K:"
9030 GET #1,X
9032 IF X>48 AND X<=N0+48 THEN 9035
9033 PRINT "NOT OFFERED":CLOSE #1
9034 FOR I=1 TO 1000:NEXT I:GOTO 9000
```

137

```
9035 PRINT CHR$(X)
9040 PRINT
9045 CLOSE #1
9050 CHOICE=X-48
9090 RETURN
9396 REM
9398 REM * INPUT THE # OF CHOICES
9400 READ N0
9410 RETURN
9900 DATA 9
10010 A$="PLAY TIC TAC TOE":RETURN
10020 A$="SUPER GRAPHICS DEMO":RETURN
10030 A$="SWELL SOUND EFFECTS":RETURN
10040 A$="GOLF EXTRA":RETURN
10050 A$="COMPLICATED ARITHMETIC":RETURN
10060 A$="NEXT OPTION":RETURN
10070 A$="ANOTHER ONE":RETURN
10080 A$="THIS IS THE LAST OPTION":RETURN
10090 A$="QUIT":RETURN
```

Program 7-5. Process a menu.

We have set up this menu program so that each option is assigned to a string A$. Lines 9000 to 9090 take care of displaying the menu and accepting a response from the keyboard. In line 9001 we turn off the cursor for a cleaner display. We then print our heading, "** MENU **", and display the choices. Take careful note of line 9010. We have made good use of the ability of Atari BASIC to use calculated values for GOSUB routines. Here's how it works: the options are assigned beginning at line 10010 and are numbered by ten. The loop that reads them starts with 1, which gives the GOSUB statement in 9010 a value of $10000 + 1 * 10 = 10010$. The subroutine at this line number merely assigns A$ and then RETURNs to the main body of the program, where we then print out A$. The next loop, we get GOSUB 10020, and so on until we get the last item in the menu. We could also have created one long string (as we did earlier in the Geography game) that would contain all the possible selections and then accessed pieces of the string to get the individual items. We saw then, however, that this can be cumbersome, although when many items are concerned it may be the only practical way. We present this other method here for you to consider in your own programs. We use the same programming in line 133 to display the user's choice.

Note how the user's choice is handled in lines 9020 to 9090. We first CLOSE and then OPEN our channel and then use a GET statement to retrieve a character from the keyboard. The GET statement will cause the computer to wait until some key is pressed before allowing the program to go any further. If you look in Appendix E, you will see that the ATASCII decimal code for 1 is 49 and for 9 is 57. So in line 9032 we check to see that the value gotten is in this range. If it isn't, we print a message, CLOSE the channel, and ask for a new input. This very neatly takes care of pressing wrong keys. If the input is okay, we use CHR$(X) to print out the character and then

subtract 48 from X to obtain a numeric representation. We then use this in line 133 to display the name of the choice.

In lines 120 to 126 we allow the user to stop using this program. If "QUIT" is the choice, in line 121 we ask that this be verified; if it is we END. Otherwise we go back and again display the menu. To prevent the screen from scrolling upward and messing up our display, we make use of the two arrows in line 121. This moves the cursor up two lines before printing the message. We write over what was on that line, and the only precaution is to be sure that we arrange our message, using extra spaces if necessary, to completely blot out what was on that line before.

We might want more than nine options. One method for handling this is to break the selections into two categories, so that each category can have up to nine options. This is not a bad idea anyway. Simply include an earlier question that tells the program which category to offer. This structure can be extended to provide various "levels" of menu, where some selections bring forth another menu offering another selection. Finally, a selection takes the user into the process or program desired.

If you simply must have more than nine options, then you might try using hex digits to get up to 15 options, or use the alphabet. You can probably squeeze about 20 selections on one screen; if the names are short, you could display them two to a row, fitting 40 selections on the screen.

After obtaining the selection from the menu, you can go ahead and have the program RUN the selection. For tape users this will mean asking the user to get the tape cued up and press the right keys; for disk users you can just RUN the program with

```
150 RUN "D:GOLF"
```

or whatever the program is called.

...Redefined Characters Revisited

In Programmer's Corner 5 we looked at redefining the Atari character set. We can now apply what we have learned in this chapter to improve our understanding of the Atari character set. Before going further you ought to look over that material to refresh your memory.

We learned that each character is stored as eight consecutive integers and that the order in which the characters are stored is not the same as the ATASCII character order. We saw how the "B" was constructed and how the eight numbers that represent the "B" are obtained. If you take a look at Figure 7-6, a modification of Figure 5-1, you will see that each horizontal line of the letter "B" is just the binary representation of the number, where a 1 means than that space is lit up and an 0 means that it is blank. Since we now know how to convert decimal numbers to binary numbers, we can update our program that showed us the numerical representation of that "B" so that it will produce an enlarged picture of exactly how the letter is displayed. In fact, we can make the program show us any character in the character set. Here's a program that does just that:

```
90 REM * FIND ANY CHARACTER
95 GRAPHICS 0:POKE 82,2
100 DIM CHAR$(1),A(8)
140 PRINT "PRESS <RETURN> TO END."
150 PRINT "INPUT THE CHARACTER TO FIND ";
:INPUT CHAR$
152 IF CHAR$="" THEN END
155 X=ASC(CHAR$)
160 GOSUB 500
165 PRINT
167 POKE 85,10:PRINT "1          "
168 POKE 85,10:PRINT "2631       "
169 POKE 85,10:PRINT "84268421"
170 START=57344+X*8
180 FOR I=START TO START+7
190 N=PEEK(I)
195 PRINT N;:POKE 85,10:GOSUB 300
198 PRINT
200 NEXT I
210 PRINT
220 GOTO 150
290 END
298 REM * LOAD THE ARRAY
300 FOR J=8 TO 1 STEP -1
310 IF N/2=INT(N/2) THEN A(J)=0
320 IF N/2<>INT(N/2) THEN A(J)=1
330 N=INT(N/2)
340 NEXT J
348 REM * DISPLAY RESULTS
350 FOR J=1 TO 8
360 IF A(J)=0 THEN PRINT " ";:REM INVERSE
SPACE BETWEEN QUOTES
365 IF A(J)=1 THEN PRINT " ";:REM REGULAR
SPACE BETWEEN QUOTES
370 NEXT J
380 RETURN
500 IF X>127 THEN X=X-128
510 IF X>31 AND X<96 THEN X=X-32:RETURN
520 IF X<32 THEN X=X+64
530 RETURN
```

Program 7-6. The eight-number representation of any character.

This program is a combination of two earlier programs—the one that found characters in the set stored in memory and the one that converted decimal numbers to binary. We first ask what character the user wants to see and then go to the subroutine at line 500 to determine the internal code for that character (refer to the table in Programmer's Corner 5). Try out a few numbers and you'll see that the logic of this routine produces the correct value (and can be used to avoid having to look up values in a table). Knowing the internal code, we are then able in line 170 to

locate the start of the character of interest and then print out the decimal values and their binary equivalent to show the actual character. You might think about going the reverse route—writing a program that allows you to draw characters in an 8-by-8 grid and then displays the decimal numbers that represent that character. This would be valuable if you were designing your own characters.

128	64	32	16	8	4	2	1	Total
0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	124
0	1	1	0	0	1	1	0	102
0	1	1	1	1	1	0	0	124
0	1	1	0	0	1	1	0	102
0	1	1	0	0	1	1	0	102
0	1	1	1	1	1	0	0	124
0	0	0	0	0	0	0	0	0

Figure 7-6. Binary representation of the letter "B".

141

Chapter 8

The Tape

The Atari program recorders provide an economical, reliable method of saving programs onto standard-quality audio tapes. Without a means of saving programs that you have developed, your Atari becomes more of a burden than a benefit. With the ability to save programs that can be used later without having to retype the entire program, your Atari can be a pleasure to use. You will want to have either the program recorder or the more expensive—but faster and more versatile—alternative, the disk drive. In Chapter 1, we briefly discussed how to save and load programs with the program recorder. Our purpose in introducing this topic early on was to make it easy for beginners to save their valuable programs. Let's go over what we learned then and add some details.

8-1...Storing and Retrieving Programs with the Program Recorder

There are three commands used to save programs to tape: CSAVE, SAVE "C:", and LIST "C:". All three require the following preliminaries:

1. Type LPRINT or LP. and press RETURN.
2. Insert the tape into the recorder.
3. Rewind the tape to its start.
4. Reset the tape counter.
5. (Optional) Move the tape forward to some desired tape counter setting.

When you type LP. and press RETURN you will receive the message ERROR—138 (if you don't have a printer turned on), but that can be ignored. Using this command will avoid a bug that sometimes shows up and prevents successful saving of programs on tape.

You are then ready to instruct the computer to save your program. CSAVE and SAVE "C:" will save the entire program on tape. LIST "C:" can also be used to save the entire program; or you can save a portion of the program with

```
LIST "C:",A,B
```

where A and B are the starting and ending line numbers, respectively, of the section of the program to be saved. The use of LIST "C:" also saves the program (or section) in a format that can later be loaded back in without erasing any program currently in the computer's memory. This permits you to merge two or more programs, which cannot be done with programs saved with CSAVE or SAVE "C:".

After you type in one of these commands, the computer will respond with two beeps, signaling you to press the RECORD and PLAY keys together and then press RETURN. If your TV sound is turned up, you will hear 20 seconds of a continuous pitch followed by alternating bursts of varying sounds and silence as the program is saved to tape. When the process is completed, you will see the READY prompt on the TV screen. You should then press the STOP lever to release the RECORD and PLAY levers. Make a note on the tape label of the program you have saved and the tape counter reading at the start and end of the recording. You need the first number so that you can later position the tape for loading the program back into the computer; the second number is necessary so that you will know where it is safe to save another program on the same tape without writing over an existing program.

Loading programs from the tape back into the computer begins much like saving a program. You don't need to type "LP.", but you do need to position the tape at the start of the program. To do this, rewind the tape, set the counter to zero, and then move forward to the tape counter number that you recorded on the tape label. The next command depends on how the program was saved to tape. If you used CSAVE, load with CLOAD; with SAVE "C:", use LOAD "C:"; and with LIST "C:" or LIST "C:",A,B, then use ENTER "C:". Note that the starting and ending line numbers (A and B) are not necessary when you are ENTERing a program from tape.

Any of these commands will cause the computer to respond with one beep, prompting you to press the PLAY lever and then RETURN. You will then hear a continuous tone followed by spurts of sound and silence as the program loads. The READY prompt will follow a successful load. An alternative to the LOAD "C:" command is to use

```
RUN "C:"
```

which has the effect of loading and running a program saved on tape in one step. This command can only be used with SAVE "C:". You can set up a series of programs that are stored on tape with SAVE "C:" to run each other by making the last line of each one

```
1000 RUN "C:"
```

Of course, you would use whatever line number was appropriate for the program. If you do this, you should also PRINT a message on the screen telling the user to press the PLAY lever and then the RETURN key.

You can also instruct the computer to execute RUN "C:" or CLOAD or ENTER "C:" without hitting the RETURN key by using the following code:

```
100 POKE 764,12
110 RUN "C:"
```

or CLOAD or ENTER "C:". Memory location 764 stores the internal code for the last key pressed. Since 12 is the internal code for the RETURN key, by executing POKE 764,12 the computer will see this value when its checks the keyboard after the next command; finding a 12 in location 764, it will proceed to load in your program. In this case you need only give a prompt to position the tape somewhere earlier in your program.

8-2...Storing and Retrieving Data with the Program Recorder

Just as we can load programs from tape and save programs to tape, we can read data from tape and write data to tape. We have seen several programs that used DATA statements to READ in information that the program required. We could have easily done the same thing by READing a file from tape. Files are useful in many situations. For example, we might have a list of names or a set of numbers used by several programs; if they are stored in one file on tape we can avoid having to type the list as part of each program.

To write data to a tape file we go through the same initial procedure described above for positioning the tape and noting the starting counter number. We then use

```
OPEN #N,8,0,"C:"
```

in our program to open a channel to the recorder for the purpose of writing. The #N is the channel number; we may use any number from 1 to 5. The 8 indicates that we are writing. Immediately after opening the channel, we should include the following lines in our program:

```
100 FOR I=1 TO 128
110 PUT #N,0
120 NEXT I
```

The function of this code is to write a 128-byte-long record to the tape. The reason for this is that the OPEN command to the recorder causes a leader to be written to the tape; the computer then does not shut off the recorder motor until a 128-byte

record is sent to the recorder. If the program doesn't take care of doing this, there will be problems later when you try to read the data. When we read in the data we will start by reading and discarding this first 128 bytes, because we know it precedes our real data.

Once we have opened a channel and written our 128-byte dummy file, we are ready to write data to the tape. We do this with

```
PRINT #N;NUMBER or PRINT #N;A$
```

to write whole numbers or records; or

```
PUT #N,X
```

to write single characters, where X is the ATASCII value (0-255) of the character being sent. It is generally more useful to use PRINT statements. The form of PRINT statement given above will also automatically send an end-of-line character (CHR$(155)) after the record or number. Using the program statement

```
PRINT #N;N; or PRINT #N;A$,
```

will not result in an end-of-line character. This will prove to be a problem when we want to read in the records or numbers, because the computer uses the end-of-line character to know when it has come to the end of that record or number. It is advisable to always provide an end-of-line character between records or numbers, either by putting them on separate program lines, by putting several PRINT statements on one line separated by colons (:), or by actually printing CHR$(155) as the last character in the record or number. When we are finished writing data to the tape we always use CLOSE #N. This ensures that all the records are sent to the recorder.

We retrieve data written on tape by using an INPUT statement. First, however, we must OPEN a channel, this time for reading. We do this in a program with

```
OPEN #N,4,0,"C:"
```

where the 4 signifies reading. Then we use

```
INPUT #N;NUMBER
```

or

```
INPUT #N;A$
```

to read in individual records that are separated on the tape by end-of-line characters. It is important to emphasize that the end-of-line characters are what the computer looks for to determine where individual records on the tape end. Without them, it will keep right on reading, assuming that everything that follows is part of the record it was sent to read. Once again, when we are done reading from the tape we finish up with CLOSE #N.

Let's look at a demonstration program that writes strings and numbers to tape using PRINT and then reads them back using INPUT.

```
90 REM * TAPE INPUT/OUTPUT DEMONSTRATION
100 DIM A$(30)
102 PRINT "POSITION TAPE FOR WRITING."
104 PRINT "THEN PRESS <START>."
105 PRINT "PRESS <RETURN> AFTER BEEPS."
107 IF PEEK(53279)<>6 THEN 107
110 OPEN #1,8,0,"C:"
120 FOR I=1 TO 128
130 PUT #1,1
140 NEXT I
198 REM * PRINT TO TAPE
200 A$="This is tape data #"
210 FOR I=1 TO 5
215 PRINT A$;I
220 PRINT #1;A$;I
230 NEXT I
250 X=100:Y=3.14159265:Z=3.15E+21
260 PRINT X:PRINT Y:PRINT Z
270 PRINT #1;X:PRINT #1;Y:PRINT #1;Z
280 CLOSE #1
298 REM * READ FROM TAPE
300 PRINT "POSITION TAPE TO READ"
305 PRINT "THEN PRESS <START>."
308 PRINT "PRESS <RETURN> AFTER BEEP."
310 IF PEEK(53279)<>6 THEN 310
315 OPEN #1,4,0,"C:"
320 FOR I=1 TO 128
330 GET #1,X
340 NEXT I
350 FOR I=1 TO 5
360 INPUT #1;A$
370 PRINT A$
375 NEXT I
380 INPUT #1;X:PRINT X
390 INPUT #1;Y:PRINT Y
400 INPUT #1;Z:PRINT Z
410 CLOSE #1
```

Program 8-1. Tape INPUT/OUTPUT demonstration.

```
RUN
POSITION TAPE FOR WRITING.
THEN PRESS <START>.
PRESS <RETURN> AFTER BEEPS.
This is tape data #1
This is tape data #2
This is tape data #3
This is tape data #4
This is tape data #5
100
3.14159265
```

```
3.15E+21
POSITION TAPE TO READ
THEN PRESS <START>.
PRESS <RETURN> AFTER BEEP.
This is tape data #1
This is tape data #2
This is tape data #3
This is tape data #4
This is tape data #5
100
3.14159265
3.15E+21

READY
```

Figure 8-1. Execution of Program 8-1.

We have printed on the screen and then written to tape a series of five different strings and then three different numbers, which were then read back in and again printed on the TV to verify our coding. Notice how we write a dummy 128-byte record in lines 120 to 140 and then read it back in and ignore it in lines 320 to 340. Lines 380 to 400 could just as well have been

```
380 INPUT #1;X,Y,Z
390 PRINT X:PRINT Y:PRINT Z
```

That is, commas are acceptable between items when you are entering data or records (although they should be avoided for output).

...Converting GEOGRAPHY to Tape

We can now use this information to convert our Geography game so that it reads the list of places from tape and, when we have finished playing, saves the new, enlarged list back onto tape for future use. Because we wrote the original program using subroutines it will be an easy task to modify it now for use with the program recorder.

We'll begin with a short program to write a file that contains the four names that the computer has at the start of the game. See Program 8-2.

```
10 REM * INITIALIZE THE PLACES FILE FOR THE
GAME OF GEOGRAPHY.
20 REM * WE ARE ENTERING THE FOUR LARGEST
CITIES IN THE USA.
90 N0=4
92 PRINT "POSITION THE TAPE AND THEN"
94 PRINT "PRESS THE <START> KEY."
95 PRINT
96 PRINT "PRESS <RETURN> AFTER THE BEEPS."
98 IF PEEK(53279)<>6 THEN 98
100 OPEN #4,8,0,"C:"
110 PRINT #4;N0
```

```
120 PRINT #4;"NEW YORK"
130 PRINT #4;"CHICAGO"
140 PRINT #4;"LOS ANGELES"
150 PRINT #4;"PHILADELPHIA"
160 CLOSE #4
```

Program 8-2. Write names to a file for Geography game.

This program uses the methods just discussed for saving records on tape. Note that we provide an on-screen prompt to lead the user through the steps of saving the data on tape (lines 92–96). Once this program has done its job, we won't need it any longer.

Our main Geography program will need a routine to read these records from tape at the start of the game. The original program used DATA statements to do the job, but we will modify that to retrieve the data from tape. See Program 8-3.

```
90 REM * DISK DIRECTORY FROM BASIC
100 DIM FILE$(17)
110 OPEN #1,6,0,"D:*.*"
120 TRAP 200:INPUT #1;FILE$
130 PRINT FILE$
140 GOTO 120
200 CLOSE #1
210 END
```

Program 8-3. File-reading subroutine for Geography game.

Once we enter place names, we add them to our long string, keeping track in two arrays (START and FINISH) of where each one starts and ends, exactly as we did in the original program (lines 8015–8040). We refer to a subroutine at line 10000 in line 8000 of this program. This subroutine supplies the prompts needed to get the tape ready. One routine is used for both reading and writing by assigning either the word READ or WRITE to the string CP$ before going to line 10000 and then using this word in that routine. See Program 8-4.

```
9998 REM * PROMPTS FOR RECORDER
10000 PRINT "⅁":REM ESC CTRL+CLEAR
10005 PRINT "POSITION THE TAPE TO ";CP$;" THE
FILE"
10010 PRINT "OF PLACES,THEN PRESS THE <START>
KEY."
10020 PRINT
10030 PRINT "PRESS <RETURN> AFTER THE BEEP(S)."
10040 IF PEEK(53279)<>6 THEN 10040
10050 RETURN
```

Program 8-4. Prompt the user to ready the tape.

Notice line 10040. Here we use PEEK to access memory location 53279; if the computer returns a 6 we know that the START key has been pressed.

We also need to add a new routine to write the places file back onto tape when the game is finished. This will be much like the program that created the small

148

record file that we first wrote. We will be writing the long string that contains all the place names accumulated at the end of the game to a file that will be available for the next game. Here again, we provide prompts so that the user can get the tape ready and press the right levers.

```
8498 REM * UPDATE PLACES FILE
8500 CP$="WRITE":GOSUB 10000
8505 CLOSE #3:OPEN #3,8,0,"C:"
8510 PRINT #3;N0
8520 FOR I9=1 TO N0
8530 PRINT #3;NAMES$(START(I9),FINISH(I9))
8540 NEXT I9
```

Program 8-5. Update the places file for Geography game.

Of course, we will want to save the list of places right after the program is saved on the tape, so that we can RUN the program and have it load in the places without us having to reposition the tape. By having the program accept a RETURN command without our having to press that key (this is where we need to use POKE 764,12), we can ensure that what follows on tape is the game data.

```
10 DIM NAME$(20),NAMES$(2000),START(100),
FINISH(100),AVNAMES(100),
PP$(20),A$(3),CP$(20)
21 NAME$=""
25 GOSUB 8000:REM * READ NAMES ARRAY
30 GOSUB 9000:REM * INSTRUCTIONS
35 GOSUB 4000:REM * INITIALIZE AVAILABLE NAME
ARRAY
40 GOSUB 7000:REM * COMPUTER STARTS
50 GOSUB 6000:REM * PERSON RESPONDS
55 IF PP$="QUIT" THEN 75
60 GOSUB 5000:REM * RESPONSE OF COMPUTER
65 IF CP$<>"QUIT" THEN 50
75 PRINT "DO YOU WANT TO PLAY AGAIN (Y/N)";
80 INPUT A$
90 IF A$(1,1)<>"N" AND A$(1,1)<>"n" THEN 30
100 GOSUB 8500
105 PRINT "I'm ready to play whenever you are.
"
110 END
120 GOTO 30
3998 REM * INITIALIZE AVAILABLE NAMES ARRAY
4000 FOR J9=1 TO N0
4010 AVNAMES(J9)=1
4020 NEXT J9
4090 RETURN
4998 REM * COMPUTER RESPONDS
5000 FOR I9=1 TO N0
5005 LAST=LEN(PP$)
5010 IF NAMES$(START(I9),START(I9))=PP$(LAST,
```

149
...

```
 LAST) AND AVNAMES(I9)=1 THEN POP :GOTO 5050
5015 NEXT I9
5020 PRINT :PRINT " I have run out of names!"
5025 CP$="QUIT"
5030 GOTO 5090
5050 CP$=NAMES$(START(I9),FINISH(I9))
     :AVNAMES(I9)=0
5060 PRINT "    I CHOOSE: ";CP$
5090 RETURN
5998 REM * PERSON GO
6000 PRINT
6010 PRINT "    YOUR TURN: ";
6011 INPUT PP$
6012 IF PP$="QUIT" THEN 6190
6015 IF LEN(PP$)>1 THEN 6030
6020 PRINT "NAME TOO SHORT ":GOTO 6010
6030 IF PP$(1,1)=CP$(LEN(CP$),LEN(CP$)) THEN
     6040
6035 PRINT "NO MATCH":GOTO 6010
6040 FOR I9=1 TO N0
6045 IF PP$=NAMES$(START(I9),FINISH(I9)) THEN
     6100
6050 NEXT I9
6055 IF N0<100 THEN 6065
6060 PRINT "NO MORE ROOM FOR MORE NAMES":GOTO
     6010
6065 N0=N0+1
6067 LONGSIZE=LEN(NAMES$):SIZE=LEN(PP$)
6068 IF LONGSIZE+SIZE>2000 THEN 6060
6069 START(N0)=LONGSIZE+1
6071 FINISH(N0)=START(N0)+SIZE-1
6073 NAMES$(LEN(NAMES$)+1)=PP$:AVNAMES(N0)=0
6080 GOTO 6190
6098 REM * "FOUND NAME"
6100 IF AVNAMES(I9)=1 THEN 6150
6110 PRINT "USED ALREADY":GOTO 6010
6150 AVNAMES(I9)=0
6190 RETURN
6998 REM * COMPUTER BEGIN THE GAME
7000 X9=INT(RND(0)*(N0-1))+1
7028 CP$=NAMES$(START(X9),FINISH(X9))
7030 PRINT "FIRST PLACE : ";CP$:AVNAMES(X9)=0
7090 RETURN
7998 REM * READ PLACES FILE
8000 CP$="READ":GOSUB 10000
8002 OPEN #3,4,0,"C:"
8005 INPUT #3;N0
8010 FOR I9=1 TO N0
8015 LONGSIZE=LEN(NAMES$)
8020 START(I9)=LONGSIZE+1
8025 INPUT #3;NAME$
```

150
. . .

```
8030 SIZE=LEN(NAME$)
8035 FINISH(I9)=LONGSIZE+SIZE
8040 NAMES$(LEN(NAMES$)+1)=NAME$
8045 NEXT I9
8050 CLOSE #3
8090 RETURN
8498 REM * UPDATE PLACES FILE
8500 CP$="WRITE":GOSUB 10000
8505 CLOSE #3:OPEN #3,8,0,"C:"
8510 PRINT #3;N0
8520 FOR I9=1 TO N0
8530 PRINT #3;NAMES$(START(I9),FINISH(I9))
8540 NEXT I9
8550 CLOSE #3
8590 RETURN
8998 REM * INSTRUCTIONS
9000 PRINT "5This program will play a
geography"
9005 PRINT "game with you. You will take
turns"
9010 PRINT "with the computer. Each of you
will"
9015 PRINT "be trying to think of names of
places"
9020 PRINT "such that the first letter of
your"
9025 PRINT "name is the same as the last
letter"
9030 PRINT "of the previously used place
name.":PRINT
9035 PRINT "PLEASE USE CAPITAL LETTERS ONLY."
:PRINT
9040 PRINT "ARE YOU READY (Y/N)"
9045 INPUT A$
9050 IF A$(1,1)<>"Y" AND A$(1,1)<>"y" THEN
9045
9060 PRINT "5":REM ESC CTRL+CLEAR
9080 RETURN
9998 REM * PROMPTS FOR RECORDER
10000 PRINT "5":REM ESC CTRL+CLEAR
10005 PRINT "POSITION THE TAPE TO ";CP$;" THE
FILE"
10010 PRINT "OF PLACES,THEN PRESS THE <START>
KEY."
10020 PRINT
10030 PRINT "PRESS <RETURN> AFTER THE BEEP(S).
"
10040 IF PEEK(53279)<>6 THEN 10040
10050 RETURN
```

Program 8-6. File-oriented Geography game.

151
. . .

...SUMMARY

Atari program recorders are valuable for saving programs and data on tapes. We can use CLOAD or LOAD "C:" to load in programs that have been saved with CSAVE or SAVE "C:", respectively. We can also use RUN "C:" to both load and run a program that was saved with SAVE "C:". We can use ENTER "C:" to load programs or parts of programs saved with LIST "C:". Data records may be saved to tape after using OPEN #N,8,0,"C:" with PRINT #N;NUMBER or PRINT #N;A$. You should always write a blank 128-byte-long record to the tape after an OPEN statement to prevent later loading problems. Records may later be retrieved after OPEN #N,4,0,"C:" with INPUT #N;NUMBER or INPUT #N;A$. In both cases you should use CLOSE #N when the transfer is finished.

PROGRAMMER'S CORNER 8

...Tape Loading Problems

Loading programs from tape can be a trying experience. It is important that the tape be positioned accurately so that there is not too much or too little leader tape before the actual program data. This can be particularly tricky when there are several programs on a tape. The problem is compounded because the Atari does not permit saving programs by name and instead uses only the presence of a 20-second leader to signal the location on the tape where a program starts. You can, however, use the trick given in the preceding section to help in finding programs stored on tape. We will LIST the name of the program on tape in a REM statement and follow this with the actual program. This LISTed name will be used as a marker to find the program on tape. The REM statement should be

```
1 REM PROGRAM NAME HERE
```

with one space between 1 and REM and one space between REM and the program name. This is important because we will INPUT this LISTed file and take anything starting with the seventh character as the name. If we match the name we're looking for, we'll want to load in the file that follows. If we don't, we'll need to read in and discard the file that does follow and then try again with the next LISTed named file. We will check whether we've reached the end of an unwanted file by using PEEK to access memory location 63. This location stores information on whether or not the end of a tape file has been found. A value other than zero in this location means that the end of the file has been reached. This will be our signal to check the next name file to see if it matches the name we're looking for. We will continue to do this until we find the desired program or reach the end of the tape. Program 8-7 provides all of these functions.

```
90 REM * USING PROGRAM NAMES WITH TAPE
100 POKE 752,1
110 DIM NAME$(40),TAPE$(128),PROG$(30)
```

```
120 ? "⅃":REM * CLEAR SCREEN WITH ESC CTRL+
CLEAR
125 PRINT "PRESS PLAY BUTTON ON RECORDER, THEN
"
130 PRINT "ENTER PROGRAM NAME";:INPUT NAME$
140 PRINT "SEARCHING........."
150 POKE 764,12
160 OPEN #1,4,0,"C:"
170 TRAP 300:INPUT #1,TAPE$:PROG$=TAPE$(7,
LEN(TAPE$))
180 IF TAPE$(7,6+LEN(NAME$))=NAME$ THEN 200
190 GOTO 300
200 TRAP 210:INPUT #1,TAPE$
210 CLOSE #1
220 PRINT :PRINT "FOUND PROGRAM. LOADING....."
230 POKE 764,12:CLOAD
300 PRINT :PRINT "PROGRAM NOT FOUND..."
310 TRAP 350:INPUT #1,TAPE$
350 CLOSE #1
360 PRINT :PRINT "ADVANCING TO NEXT PROGRAM.."
370 PRINT "THIS PROGRAM WAS ";PROG$
380 POKE 764,12:OPEN #1,4,128,"C:"
390 TRAP 400:INPUT #1,TAPE$:GOTO 390
400 IF PEEK(63)<>0 THEN CLOSE #1:GOTO 140
410 GOTO 390
```

Program 8-7. Finding a program by name on tape.

We assign the search name to NAME$ in line 130. We then use POKE 764,12 to turn on the recorder to begin the search, using a TRAP statement to catch potential errors. There are several other places in this program where we use TRAP statements to prevent errors from stopping the program (see lines 200, 310, and 390). Also notice that in line 170 we assign to PROG$ any characters read into TAPE$ from the seventh one on. Then in line 180 we check to see if we have matched our search criterion. If we match, we load the program at line 230. If not, we jump to line 300, using lines 380 to 410 to bypass the unwanted file. When line 400 shows that we have reached the end of the tape file we go back to line 140 and begin to search again. Notice that line 230 uses CLOAD. Depending on how the programs were saved on tape, you could use LOAD "C:" or RUN "C:" instead.

To use this program you will need to LIST the appropriate REM statement just before the program is saved. This can most easily be done by making the first line of your program the REM statement with the program name and then using LIST "C:",1,1 to store just this line on tape. Then, without repositioning the tape, you should CSAVE (or SAVE "C:") your entire program. You will now be able to store a series of programs with their names on a tape and load them in with ease.

Chapter 9

The Disk

9-1...Getting Started

DOS stands for *Disk Operating System.* In what follows, we will be referring exclusively to DOS II, the sequel to DOS I. It is unlikely that you will come across any disks that use DOS I. There is also a DOS III that functions very much like DOS II and also supports double-density disk drives. DOS II consists of DOS.SYS and DUP.SYS. We will discuss DUP.SYS at the end of this chapter. If your disk drive is turned on, it is DOS.SYS that the computer will look for when it is first turned on. This is the file that allows you to save programs on disk. This single capability often justifies the purchase of a disk drive. With a disk system we may save many programs on one disk and retrieve them later using program names. Using DOS, we may request that the computer display the name of each program on a given disk for us. This is a tremendous advantage over cassette tapes. The disk is also about 20 times as fast and much more reliable.

Furthermore, we can also easily use a disk to store data. The ability to store data on a disk turns our computer into a powerful data processing system. We can then use an Atari, for example, to handle a mailing list with hundreds of names. We can write statistical data with one program and use that data in any number of other programs.

Data stored on a disk is referred to as a *data file.* Programs stored on disk are files also, but programs are recognized by BASIC as special.

We discussed some disk preliminaries in Chapter 1 to enable us to save programs on disk. Let's review what we covered there:

1. In order to save a program (or data) on disk, the disk must first be formatted. This is a process whereby the disk is marked so that the computer can keep track of what goes where on the disk surface. This is done in BASIC by first typing DOS. Note that this will erase any program currently in the computer's memory. When the disk command menu appears on the screen, type "I" and respond to the "WHICH DRIVE" question with the number of the drive containing the disk to be formatted. Then answer the verification question with "Y" and the disk will be formatted. This process erases everything that may be currently on the disk, so be very careful! After this point, programs and data may be stored on the disk. It is good practice to put opaque tape over the notch on the disk edge and remove it only when you really want to format or write on a disk. With tape over this notch, the computer will not be able to write to or erase the disk.

2. Once a disk is formatted, we may put DOS.SYS and DUP.SYS on the disk if we so desire by typing "H" and then again answering the drive number and verification questions. At this point, the computer will recognize this disk on power-up.

3. If you then type "B" you will be back in BASIC. After you finish with your current program and wish to save it, you type

        SAVE "DN:FILENAME.EXT"

or

        LIST "DN:FILENAME.EXT"

The first will save the file on disk drive #N with the file name FILENAME.EXT. The "N" is optional if you are using disk drive 1. FILENAME.EXT has this form: up to eight letters (capitals and numbers only, with a capital letter first and no spaces) plus an optional three-character extension. The name does not have to be eight characters long in order to use an extension. The second method, using LIST, saves the file in a different form; the difference will be clear when we discuss loading these files.

4. Loading a SAVEd file is done with the statement

        LOAD "DN:FILENAME.EXT"

to load the file into computer memory. Alternatively,

        RUN "DN:FILENAME.EXT"

will LOAD and RUN the program with this name. In both cases, files currently in memory are erased before the commands are executed. If you try to LOAD or RUN a file that was LISTed to disk, you will get ERROR— 21. A LISTed file is loaded with

        ENTER "DN:FILENAME.EXT"

Files in memory are not erased when programs are loaded with ENTER. Rather, the new program merges with any existing program in memory, replacing any commonly numbered lines. This makes it possible to save numerous subroutines on disk and use them in many programs without having to retype them for each new use. This is most easily done by numbering subroutine lines using numbers greater than 30000 and having each subroutine use different line numbers. It is easy then not to use these large lines numbers in your main program, so that there will be no line number conflict when you ENTER the subroutine.

...What Is a File?

A file is simply some area of the disk where we can save data. We can, of course, also save programs on disk. When we save programs, DOS does everything for us. When we save data in a data file, it is up to us to organize the data.

One of the aspects of data files that encourages mystery is that they are invisible. Well, so are programs during execution. We have found that we could do fantastic things with programs even though we could see nothing going on until the final printed result. We will now expand our capabilities tremendously by using programs to create and access data files. We can LIST a program, but we are going to have to write programs to "LIST" any data files we create.

Data can be organized in a sequential format or arranged for random-access. Some statements and techniques are the same for both sequential and random-access files. We will be using OPEN, CLOSE, PRINT, INPUT, PUT, and GET with both types of files.

9-2...Sequential Files: An Introduction

Sequential files are easy to set up and use. We simply do everything starting at the beginning. If we want to place 15 items in a file, then we simply write them in order from 1 to 15. If we later wish to read the fourteenth item, then we read them all in order, beginning with item 1 and stopping when we get to item 14.

...OPEN, CLOSE, PRINT, INPUT, PUT, GET

These statements are our window to the disk file for output and input. We must always OPEN a file before we transfer any data. We use

```
OPEN #N,X,Ø,"DN:FILENAME.EXT"
```

where N refers to the channel number to be used for communication. We'll remind you again that 0, 6, and 7 are used by the computer, so restrict your values for the channel to 1 to 5. The X is a number that indicates the type of exchange that is to take place. It can have the following values:

12 = input and output; 4 = input; 8 = output; 9 = append to end of file; 6 = read disk directory.

156

The 12 is used for random-access files; we'll discuss that in a later section of this chapter. The 4 and 8 are what we will be using for sequential files, where 4 will be used to read in our file from disk and 8 will be used to write back to the disk after we have finished making additions, deletions, and changes to the file. The 8 will erase the file with the name FILENAME.EXT before it writes a new file onto the disk. The 9 is used to append data onto the end of an existing file, but it also has a big disadvantage. Each and every time that a file is opened with this command, a new disk sector (containing 128 bytes) is appended onto the end of the file, even if you are only going to write one short piece of data to the disk. It doesn't take too many disk accesses to build up a very large file, filling all of the available space on a disk. For this reason, we will use 8 to write a new file each time we are ready to output to the disk and do all our appending while in our BASIC program.

We can use OPEN with 6 to read the disk directory while in BASIC. The following program does just that.

```
90 REM * PUT AND GET WITH THE DISK
100 DIM A$(60)
110 A$="THIS IS A TEST OF PUT AND GET!"
115 PRINT A$
120 OPEN #1,8,0,"D:TESTFILE"
130 FOR I=1 TO LEN(A$)
140 X=ASC(A$(I,I))
150 PUT #1,X
160 NEXT I
170 PUT #1,155
180 CLOSE #1
200 OPEN #1,4,0,"D:TESTFILE"
210 TRAP 260:GET #1,X
220 IF CHR$(X)="!" THEN 250
230 PRINT CHR$(X);
240 GOTO 210
250 PRINT
260 CLOSE #1
270 END
```

Program 9-1. Disk directory from BASIC.

Since we can't know how many files are on the disk being interrogated, we have used a TRAP statement so that when we run out of file names we will smoothly exit from the program with no error messages.

...Writing Data to the Disk with PRINT

Once our file has been opened with

```
OPEN #1,8,0,"D:TESTFILE"
```

we are ready to send data to the disk. We can do this with PRINT or PUT. Using PRINT to the disk functions just like using PRINT to the screen. It will print whatever numbers follow it or whatever text is enclosed in quotation marks.

Commas between items to be PRINTed will put extra spaces in the file just as they put extra spaces on the screen. For this reason, we will avoid using commas, since this is a waste of disk space. A semicolon also has the same function as on the screen: it acts to concatenate the PRINTed items in the disk file. Thus,

```
PRINT #1;1;2;3
```

will put 1, 2, and 3 in consecutive order on the disk, with no spaces between them and an end-of-line character (CHR$(155)) after the 3. We will shortly discuss the importance of this end-of-line character.

When we are done PRINTing to the disk, we always use

```
CLOSE #N
```

to finish up. This is very important for two reasons:

1. We avoid an ERROR— 129 message if we later try to OPEN this file (which will remain OPEN without a CLOSE statement).
2. We may not get all of our data PRINTed to disk unless we use it. This is because we are really writing first to a small reserved memory buffer in the computer when we write to the disk. This buffer is only transferred to the disk when it becomes full or when we use CLOSE. Thus any partially full buffer will be lost unless a CLOSE statement is executed.

...Reading Data from the Disk with INPUT

INPUT works with data that has been PRINTed to disk. Once again, we must first open the file, this time for input, with

```
OPEN #1,4,0,"D:TESTFILE"
```

We then read the data by using

```
INPUT #1;N
```

or

```
INPUT #1;A$
```

depending on what kind of data we are reading from the disk. In the second case, we must have DIMensioned a string with this name before we get to this statement.

When you use INPUT to read from disk, the only way the computer knows it has come to the end of the file is when it sees an end-of-line character. Now we see the importance of this item in our PRINT statement. In general, it is a good rule to always separate items PRINTed to disk with end-of-line characters. These will automatically be placed there if the PRINT statement does not terminate with a comma or semicolon. A colon (:) between several statements on one line is acceptable, or separate lines for each PRINT statement can be used, or we can even print an end-of-file character (CHR$(155)) to the disk.

Once again we terminate our INPUT with a CLOSE #N statement.

...Some Examples Using PRINT and INPUT

Let's look at two examples to flesh out the above ideas. First, we'll write some

numbers and strings to the disk and then read them back in. We'll print both transactions on the screen to check that we were successful.

```
90 REM * PRINT AND INPUT WITH THE DISK
100 DIM A$(60)
105 PRINT "HERE IS THE OUTPUT TO THE DISK"
110 OPEN #1,8,0,"D:TESTFILE"
120 A$="This is the first item of our file"
125 PRINT A$
130 PRINT #1;A$
140 PRINT #1;"This is the second item of our
file"
145 PRINT "This is the second item of our file"
150 READ A$
155 PRINT A$
160 PRINT #1;A$
170 NUMBER=23456
175 PRINT "The fourth item is ";NUMBER
180 PRINT #1;"The fourth item is ";NUMBER
190 CLOSE #1
195 PRINT
196 PRINT "HERE IS THE INPUT FROM THE DISK"
200 OPEN #1,4,0,"D:TESTFILE"
210 INPUT #1;A$
220 PRINT A$
230 INPUT #1;A$
240 PRINT A$
250 INPUT #1;A$
260 PRINT A$
270 INPUT #1;A$
275 PRINT A$
280 CLOSE #1
300 END
1000 DATA This is the third item of our file
```

Program 9-2. Demonstration of PRINT# and INPUT#.

```
RUN
HERE IS THE OUTPUT TO THE DISK
This is the first item of our file
This is the second item of our file
This is the third item of our file
The fourth item is 23456

HERE IS THE INPUT FROM THE DISK
This is the first item of our file
This is the second item of our file
This is the third item of our file
The fourth item is 23456

READY
```

Figure 9-1. Execution of Program 9-2.

We READ one piece of output from a DATA statement and some was generated in the program itself. Both types of output are treated in exactly the same way.

Now let's look at a case where we use semicolons to concatenate items in the disk file. Then we'll read them back and see the result.

```
90 REM * DISK WRITE WITH SEMICOLONS
100 DIM A$(40),B$(40),C$(40),D$(120)
110 A$="I never met an Atari "
120 B$="owner who I "
130 C$="didn't like."
135 PRINT A$:PRINT B$:PRINT C$
140 OPEN #1,8,0,"D:TESTFILE"
150 PRINT #1;A$;B$;C$
160 CLOSE #1
170 PRINT
200 OPEN #1,4,0,"D:TESTFILE"
210 INPUT #1;D$
220 PRINT D$
230 CLOSE #1
```

Program 9-3. Concatenation of strings in a disk file.

```
RUN
I never met an Atari
owner who I
didn't like.

I never met an Atari owner who I didn't like.

READY
```

Figure 9-2. Execution of Program 9-3.

Notice that we did get the combined data when we used INPUT to read it from the disk file.

...Using PUT and GET with Disk Files

These two statements also work together. They both operate on one character at a time. PUT will send the ATASCII value (0–255) of one character to the disk and GET will read a single character from disk. GET will accept the ATASCII value of any character since it, too, only handles numbers from 0 to 255. PUT and GET are useful because they can be used to read a file one character at a time and check each character as it is read. You can then stop when a specific character is found, go off in the program and do something, and then come back and read characters from where you left off. For example,

```
90 REM * PUT AND GET WITH THE DISK
100 DIM A$(60)
110 A$="THIS IS A TEST OF PUT AND GET!"
115 PRINT A$
120 OPEN #1,8,0,"D:TESTFILE"
```

```
130 FOR I=1 TO LEN(A$)
140 X=ASC(A$(I,I))
150 PUT #1,X
160 NEXT I
170 PUT #1,155
180 CLOSE #1
200 OPEN #1,4,0,"D:TESTFILE"
210 TRAP 260:GET #1,X
220 IF CHR$(X)="!" THEN 250
230 PRINT CHR$(X);
240 GOTO 210
250 PRINT
260 CLOSE #1
270 END
```

Program 9-4. Demonstration of PUT and GET to disk.

We have simply sent a message to the disk one character at a time and then read it back in the same way, stopping when we got to the "!". PUT and GET can be useful in this way when we are looking at data files whose structure is unknown.

Of course, we generally will store more than one string in a file. We may store hundreds of names in a file. Let's think about the logistics of building a file with a large number of names in it. If the file is to be truly very large, we have to consider that it may not fit on a single disk. As a practical matter, we will not be concerned with this for some time in our programming career. Writing the names into a file is no problem—we simply write a program that writes entry after entry.

We could use the TRAP statement and simply read until we run out of data, but there might be other sources of error in our program. It is much better to write the number of items in the file itself. We can simply make the first item in the file be the number of items to follow.

Remember the program we wrote to play Geography in Chapter 6? We wrote that program using subroutines so that it would be easy to convert to store the names in a disk file. This way, we can arrange to have the computer "remember" place names from one day to another. Let's first write a little program to store the four beginning names. See Program 9-5.

```
10 REM *INITIALIZE THE PLACES FILE FOR THE
GAME OF GEOGRAPHY-WE ARE ENTERING THE FOUR
LARGEST CITIES IN THE USA.
90 N0=4
100 OPEN #4,8,0,"D:PLACES"
110 PRINT #4;N0
120 PRINT #4;"NEW YORK"
130 PRINT #4;"CHICAGO"
140 PRINT #4;"LOS ANGELES"
150 PRINT #4;"PHILADELPHIA"
160 CLOSE #4
```

Program 9-5. Write names to a file for Geography game.

161

When we run this program, we will have a file containing five items. The first item will be a 4 and the next four items will be four city names. It is important to note that the number we wrote to the file will be converted to the characters of the number, just like STR$. Thus, if N0 = 25, then there will be a 2 and a 5 in the file. However, we may retrieve the value with INPUT N0, just the same. Or in some special situation we might want to retrieve that number in a string variable. As the number representing how many names goes from one to two digits, the space required to store it in the file goes from one character to two. So when we rewrite the entire file, each of the place names will be located one character position further along in the file. We can program solutions to many problems without even realizing this. However, as we seek more elegant solutions, information of this kind will be important to have.

Let's now convert the Geography game from Chapter 6 to store names in a file. We simply need to replace the READ . . . DATA statements with a subroutine that reads the place names from the file into a long string and provide a subroutine that writes out all of the names to the file at the end of this series of games.

The original version reads the names from DATA statements in a subroutine at line 8000. So we may simply replace the subroutine with a new one that reads the names from a file. See Program 9-6a.

```
7998 REM * READ PLACES FILE
8000 OPEN #3,4,0,"D:PLACES"
8005 INPUT #3;N0
8010 FOR I9=1 TO N0
8015 LONGSIZE=LEN(NAMES$)
8020 START(I9)=LONGSIZE+1
8025 INPUT #3;NAME$
8030 SIZE=LEN(NAME$)
8035 FINISH(I9)=LONGSIZE+SIZE
8040 NAMES$(LEN(NAMES$)+1)=NAME$
8045 NEXT I9
8050 CLOSE #3
8090 RETURN
```

Program 9-6a. File-reading subroutine for Geography game.

Since there was no cleanup at the end of the original Geography game, our routine to write the names to the file at the end will be a new subroutine. Let's put it at 8500. Then the two file subroutines will be near each other in the final program. See Program 9-6b.

```
8498 REM * UPDATE PLACES FILE
8500 CLOSE #3:OPEN #3,8,0,"D:PLACES"
8510 PRINT #3;N0
8520 FOR I9=1 TO N0
8530 PRINT #4;NAMES$(START(I9),FINISH(I9))
8540 NEXT I9
8550 CLOSE #3
8590 RETURN
```

Program 9-6b. Write names to the file in the Geography game.

We can easily incorporate these two subroutines into the original Geography program. Next, we must modify the end-of-game logic to execute the subroutine at 8500 if the game just finished will be the last. We'll also have the computer provide a friendly goodbye message. All of this is provided by the four lines of Program 9-6c.

```
90 IF A$(1,1)<>"N" AND A$(1,1)<>"n" THEN 30
100 GOSUB 8500:REM * REWRITE THE NAME FILE
105 PRINT "I'm ready to play whenever you are."
110 END
```

Program 9-6c. Changes in the control routine to convert Geography to use a file.

We present the complete program here for your convenience. See the complete Program 9-6.

```
5 REM * GEOGRAPHY WITH DISK SEQUENTIAL FILE
10 DIM NAME$(20),NAMES$(2000),START(100),FINISH
(100),AVNAMES(100),PP$(20),A$(3),CP$(20)
21 NAME$=""
25 GOSUB 8000:REM * READ NAMES ARRAY
30 GOSUB 9000:REM * INSTRUCTIONS
35 GOSUB 4000:REM * INITIALIZE AVAILABLE NAME
ARRAY
40 GOSUB 7000:REM * COMPUTER STARTS
50 GOSUB 6000:REM * PERSON RESPONDS
55 IF PP$="QUIT" THEN 75
60 GOSUB 5000:REM * RESPONSE OF COMPUTER
65 IF CP$<>"QUIT" THEN 50
75 PRINT "DO YOU WANT TO PLAY AGAIN (Y/N)";
80 INPUT A$
90 IF A$(1,1)<>"N" AND A$(1,1)<>"n" THEN 30
100 GOSUB 8500:REM * REWRITE THE NAME FILE
105 PRINT "I'm ready to play whenever you are."
110 END
3998 REM * INITIALIZE AVAILABLE NAMES ARRAY
4000 FOR J9=1 TO N0
4010 AVNAMES(J9)=1
4020 NEXT J9
4090 RETURN
4998 REM * COMPUTER RESPONDS
5000 FOR I9=1 TO N0
5005 LAST=LEN(PP$)
5010 IF NAMES$(START(I9),START(I9))=PP$(LAST,
LAST) AND AVNAMES(I9)=1 THEN POP :GOTO 5050
5015 NEXT I9
5020 PRINT :PRINT " I have run out of names!"
5025 CP$="QUIT"
5030 GOTO 5090
5050 CP$=NAMES$(START(I9),FINISH(I9))
:AVNAMES(I9)=0
5060 PRINT "    I CHOOSE: ";CP$
```

```
5090 RETURN
5998 REM * PERSON GO
6000 PRINT
6010 PRINT "    YOUR TURN: ";
6011 INPUT PP$
6012 IF PP$="QUIT" THEN 6190
6015 IF LEN(PP$)>1 THEN 6030
6020 PRINT "NAME TOO SHORT ":GOTO 6010
6030 IF PP$(1,1)=CP$(LEN(CP$),LEN(CP$)) THEN
6040
6035 PRINT "NO MATCH":GOTO 6010
6040 FOR I9=1 TO N0
6045 IF PP$=NAMES$(START(I9),FINISH(I9)) THEN
6100
6050 NEXT I9
6055 IF N0<100 THEN 6065
6060 PRINT "NO MORE ROOM FOR MORE NAMES":GOTO
6010
6065 N0=N0+1
6067 LONGSIZE=LEN(NAMES$):SIZE=LEN(PP$)
6068 IF LONGSIZE+SIZE>2000 THEN 6060
6069 START(N0)=LONGSIZE+1
6071 FINISH(N0)=START(N0)+SIZE-1
6073 NAMES$(LEN(NAMES$)+1)=PP$:AVNAMES(N0)=0
6080 GOTO 6190
6098 REM * "FOUND NAME"
6100 IF AVNAMES(I9)=1 THEN 6150
6110 PRINT "USED ALREADY":GOTO 6010
6150 AVNAMES(I9)=0
6190 RETURN
6998 REM * COMPUTER BEGIN THE GAME
7000 X9=INT(RND(0)*(N0-1))+1
7028 CP$=NAMES$(START(X9),FINISH(X9))
7030 PRINT "FIRST PLACE : ";CP$:AVNAMES(X9)=0
7090 RETURN
7998 REM * READ PLACES FILE
8000 OPEN #3,4,0,"D:PLACES"
8005 INPUT #3;N0
8010 FOR I9=1 TO N0
8015 LONGSIZE=LEN(NAMES$)
8020 START(I9)=LONGSIZE+1
8025 INPUT #3;NAME$
8030 SIZE=LEN(NAME$)
8035 FINISH(I9)=LONGSIZE+SIZE
8040 NAMES$(LEN(NAMES$)+1)=NAME$
8045 NEXT I9
8050 CLOSE #3
8090 RETURN
8498 REM * UPDATE PLACES FILE
8500 CLOSE #3:OPEN #3,8,0,"D:PLACES"
```

```
8510 PRINT #3;NØ
8520 FOR I9=1 TO NØ
8530 PRINT #3;NAMES$(START(I9),FINISH(I9))
8540 NEXT I9
8550 CLOSE #3
8590 RETURN
8998 REM * INSTRUCTIONS
9000 PRINT ")This program will play a
geography"
9005 PRINT "game with you. You will take turns"
9010 PRINT "with the computer. Each of you
will"
9015 PRINT "be trying to think of names of
places"
9020 PRINT "such that the first letter of your"
9025 PRINT "name is the same as the last
letter"
9030 PRINT "of the previously used place name."
:PRINT
9035 PRINT "PLEASE USE CAPITAL LETTERS ONLY."
:PRINT
9040 PRINT "ARE YOU READY (Y/N)"
9045 INPUT A$
9050 IF A$(1,1)<>"Y" AND A$(1,1)<>"y" THEN 9045
9060 PRINT ")":REM ESC CTRL+CLEAR
9080 RETURN
```

Program 9-6. File-oriented Geography game.

Once again we have reaped tremendous benefits from good program organization and extensive use of subroutines. By segmenting our original Geography program we made it relatively simple to convert it to operate with a data file. We replaced one subroutine, added another, and made minor changes in the control routine. By making minor changes in a well-structured program we have made major changes in that program's behavior. It is important to realize that we have isolated all possible sources of error in small areas of the resulting program. If the first program had been badly put together, we would have found ourselves tinkering in numerous places to create the new program. The tinkered program would have contained far more potential error sources.

Problems for Section 9-2 . . . .

1. Write a program that lists the place names in the Geography game file.
2. Try as people will, somebody will misspell a name in a game of Geography. Write a program that enables us to edit place names.
3. Write a program that will enable you to eliminate a place name from the Geography name file.

4. The Geography game logic for the computer response scans the names array from item 1 every time. Modify the game so that the scan begins at some random point. Don't forget to come around to the beginning of the list after checking the last name.

5. The scan for the computer's turn in Geography covers the entire names array. That unnecessarily includes the names that have been added during the current game. Modify the program so that the scan for the computer's turn covers only those place names which came from the name file at the beginning of the current game.

9-3...Random-Access Files

Since entries in a file can vary in length, they can occupy varying amounts of space. Therefore there is no way of predicting just where the fifth or the 50th entry might begin. Even if we use blank spaces to make each item in the sequential file the same length, we can only read sequential files from the beginning. When we write to such a file, the safest way is to write or rewrite the entire file. As the file becomes larger and larger, all this takes more and more time.

All of that changes with random-access files. This new file structure makes it possible to read the 25th entry in a file, make changes, and rewrite it to the file without any risk of damaging any of the other entries. This is done by allocating a fixed amount of space for each entry. This means that in many applications there is some unused space in the file.

We use random-access files for all kinds of record keeping. The ability to access any data entry at will is ideal for applications where we will not be processing every entry every time we access the file. Contrast this with the Geography game, in which we clearly processed every entry in the file with every use of the program. Random-access files are used for mailing lists, every conceivable financial accounting function, and stock portfolio management. Recipes, home management data, and magazine article reference material are all appropriate for random-access files.

In many applications several files are linked together to form a system of files. An order entry might "point" off to a mailing-list file and an inventory file.

With sequential files the fundamental unit of storage is the character or byte. With random-access files the fundamental unit of storage is the record. A record is simply a collection of bytes. If 20 bytes are enough for the entries we plan to store, then we may organize our file into records that contain just 20 bytes. The record size is entirely up to us; we decide it according to our application. It is important to study each application thoroughly and plan effectively how we will organize files to manage the data required. It is devastating to lay out a file structure with records holding three strings in 40 bytes only to find out after three long programs have been written that we should have four strings in 48 bytes.

Often a group of programs will be used to handle a file or system of files—one program to enter and delete entries, another to edit entries, and perhaps a third to print a nicely formatted report to display all of the data in the file.

...OPEN

For random-access files we use the following form of the OPEN statement:

```
OPEN #N,12,0,"D:FILENAME.EXT"
```

The 12 indicates that we want to open the file for input and output and that we do not want the file to be erased. The "pointer" that tells the computer where it is in the file will be positioned at the start of the file. Note that this statement works only if a file with this name already exists on disk and if that file has been set up with records in the format that we want to use. Since there is always a chance that we put the wrong disk in the disk drive, it is good practice to use TRAP with the OPEN statement, just in case. We will use

```
OPEN #1,8,0,"D:FILENAME.EXT"
```

to open a channel for creating the file. We will then PRINT data to the file using blank strings to fill the file. If at some future time we want to expand the file we will have to go through the same procedure. We can use OPEN to append data (using 9) to do this and not waste too much disk space if we add a block of extra blank records whenever we go through this procedure.

...NOTE

Once we have created the file and it has been OPENed for random-access, we use NOTE to determine the location on the disk where the first piece of data is located. The disk is divided into 707 sectors, each containing 125 bytes. (There are actually 128 bytes in each sector, but DOS uses the last three for its own purposes, leaving us with 125 for data.) The NOTE statement takes the form

```
NOTE #N,A,B
```

where A is the sector and B is the byte of the current pointer position. The pointer will initially be positioned at the beginning of the file and so, in A and B, we will have the location of the start of file data. We can then PRINT data to the disk and follow this with NOTE to determine where the next piece of data goes. It we do this in a loop, we can go through the file and get the locations of the beginning of all the pieces of file data. If we store this information in an array, we will be able to access any individual piece of data without having to read the file from the beginning. We only need to go through this NOTE procedure once for a given file. This will also allow us to use files that are too big to fit in the computer's memory, since we will be able to selectively locate records we are interested in, without having to load the entire file into memory.

...POINT

Once we have used NOTE to locate where all the pieces of data start in a file, we can use POINT to place the pointer in position to read in any particular record of interest or write to that location. The POINT statement takes the form

```
POINT #N,A,B
```

As above, A and B refer to the sector and byte that mark the start of the record sought or the location where you want to write. We then use INPUT #N;X or INPUT #N;A$ to read one record or PRINT #N;X or PRINT #N;A$ to write records, just as we did with sequential files.

One very important point. When you write over data in a file, do *not* write more characters than were originally there, or else you will be writing into the next record. This is most easily handled by setting the file up originally with a series of dummy records, all the same length.

An example is certainly in order. We'll first create a file that has five pieces of data in it, each 15 characters long (plus one for the end-of-line character). Then we'll use NOTE to fill an array that tells the computer where each piece of data starts. Then we'll use POINT to access individual pieces of data and to overwrite some of the data with new data, without having to read through the entire file to find the data to be replaced. Here's the program:

```
90 REM * RANDOM ACCESS DEMONSTRATION
100 DIM SEC(5),BYT(5),A$(20)
110 A$="This is data #"
198 REM * CREATE A FILE AND USE NOTE
199 PRINT "THIS IS THE INITIAL DATA IN THE
FILE:"
200 OPEN #1,8,0,"D:TESTFILE"
210 FOR I=1 TO 5
215 NOTE #1,SECTOR,BYTE
216 SEC(I)=SECTOR
217 BYT(I)=BYTE
220 PRINT #1;A$;I
225 PRINT A$;I
230 NEXT I
240 CLOSE #1
250 PRINT
498 REM * USE POINT TO READ
499 PRINT "HERE IS SOME RANDOMLY ACCESSED
DATA:"
500 OPEN #1,12,0,"D:TESTFILE"
505 FOR I=1 TO 10
510 X=INT(RND(0)*5)+1
520 A=SEC(X)
530 B=BYT(X)
540 PRINT "RANDOM NUMBER = ";X
550 POINT #1,A,B
560 INPUT #1;A$
570 PRINT A$
575 PRINT
580 NEXT I
585 CLOSE #1
590 PRINT
598 REM * USE POINT TO OVERWRITE
```

```
600 OPEN #1,12,0,"D:TESTFILE"
610 A=SEC(3)
620 B=BYT(3)
630 POINT #1,A,B
640 PRINT #1;"NEW DATA #3      "
650 CLOSE #1
998 REM * READ THE MODIFIED FILE
999 PRINT "HERE IS THE MODIFIED FILE DATA:"
1000 OPEN #1,12,0,"D:TESTFILE"
1010 FOR I=1 TO 5
1020 A=SEC(I):B=BYT(I)
1030 POINT #1,A,B
1040 INPUT #1;A$
1050 PRINT A$
1060 NEXT I
1070 CLOSE #1
```

Program 9-7. Demonstration of random access using NOTE and POINT.

The individual parts of the program are indicated with REM statements. In line 640 we overwrite the third item in the file with another item, being careful that this new item is not longer than the one it is replacing (it's okay for it to be shorter). Failure to do this will inevitably lead to problems. In most cases, it is best to make all items in the random-access file the same length, even if this requires adding spaces.

```
RUN
THIS IS THE INITIAL DATA IN THE FILE:
This is data #1
This is data #2
This is data #3
This is data #4
This is data #5

HERE IS SOME RANDOMLY ACCESSED DATA:
RANDOM NUMBER = 1
This is data #1

RANDOM NUMBER = 5
This is data #5

RANDOM NUMBER = 5
This is data #5

RANDOM NUMBER = 4
This is data #4

RANDOM NUMBER = 2
This is data #2
```

```
RANDOM NUMBER = 2
This is data #2

RANDOM NUMBER = 1
This is data #1

RANDOM NUMBER = 2
This is data #2

RANDOM NUMBER = 2
This is data #2

RANDOM NUMBER = 1
This is data #1


HERE IS THE MODIFIED FILE DATA:
This is data #1
This is data #2
NEW DATA #3
This is data #4
This is data #5

READY
```

Figure 9-3. Execution of Program 9-7.

9-4...A Random-Access Mailing List

Let's develop a computerized name-and-address list. This is a common need for business and personal use. The idea here is to store all the names and addresses in a disk file. Then we may extract those we need for any particular situation. Names may be classified by a code. We might set up a personal family mailing list file using H, W, or C to designate friends of husband, wife, or children. A business might use B and S for billing and shipping addresses.

In business it is common practice to arrange these names alphabetically, by zip code, or by business volume. In order to achieve this we would not rearrange the name file itself; instead we would create a file that contains just a list of the records in the desired order. We might maintain several such lists of record numbers. Then we can easily write a program that will read a list of record numbers and print the corresponding name-and-address data from the data file in the desired order.

Let's organize a program to build the mailing-list data file. There are a number of major tasks involved. One part of the program needs to request all of the necessary data from the keyboard. Another will write the entry into the file. Another will have to determine where the new entry belongs.

We will have to organize the entry itself. We must decide what information belongs in an entry and how many characters to allow for each item and then calculate the necessary record size. Our program must include code to manage all these things. We need a routine that will write the entry in the data file. Probably the most important part of writing the program is deciding how to organize entries within the file.

When we sit down to enter the first name and address we know that the file is empty. After that we have no idea how many names are in the file. Therefore we have no idea where the next entry should go in the file. We could keep track of how many names there are on a piece of paper. Then we might just as well keep the names on paper too. The whole idea is to let the computer do the work. We need to develop a plan for keeping track of where things are. One scheme is to assign each entry its record number as an identification number and include that number as part of the data entry. The first name in the system is number 1, the second is number 2, and so on. Now we can have the next number to be assigned saved in the file itself. A good place to do this is in record 0. Lucky for us the record count beings at 0. So, a file with no names in it should have a 1 stored in record 0. We can easily write a little initialization program to do this.

Then after each new name is entered, the program adds 1 to that value in record 0. Next, we should be thinking about how we delete a name from a file, even though we are preparing to write the program to place new entries in the file.

Deleting names from a mailing list can be handled in one of several ways. We could replace the name with the word "DELETED." Or we could develop a concept that provides that the most recently deleted entry record becomes immediately available for use by the next new entry. We can make deleted records available for new entries by setting up a catalog of available space within the file itself. To do this we include the record number itself as an item of data, with the name and address. Then when an entry is deleted we store the number of the last deleted record in the deleted record and then store the number of the currently deleted record in record 0 along with the number of the next highest record in the file. This will leave a trail of deleted record numbers beginning with the number stored in record 0. Now we have two numbers stored in record 0: the next record at the end of the file and the most recently deleted record. When we start up a new file, the most recently deleted record will be 0.

This scheme provides a method for determining whether an entry has been deleted or not. Read the record. If the identification number equals the record number, then it is real data. If not, then the entry has been deleted, and the number is the record number of the previously deleted record. As an example of a file with some deleted records see Figure 9-4.

0	9 {on the end}, 8 {last deleted entry}
1	1 JONES JOHN . . .
2	2 SMITH WILLIAM . . .
3	3 HAYES MARY . . .
4	6 {deleted entry} . . .
5	5 BRADSHAW ELEANORE . . .
6	0 {deleted entry (first one)} . . .
7	7 HOUGH HUGH . . .
8	4 {deleted entry} . . .
9	{never used}

Figure 9-4. Layout of used and deleted records.

Let's trace the available-space catalog in Figure 9-4. The second number in record 0 is 8. Look at record 8. There we find a 4. Look at record 4. There we find a 6. Look at record 6. There we find a 0. Thus the deleted records are 8, 4, and 6. When we use record 6 for a new entry, the program should place a 0 in record 0 where the 8 is now.

The entry program will have to look at the two record numbers stored in record 0 and decide whether to place the new entry at the end of the file or on a record from which a name has been deleted. That is easy. If the deleted record number is 0 the new name goes on the end. Otherwise use the deleted record.

It is important to observe in all this that even though we are designing the program to enter data, it is necessary to think through the deleting process thoroughly. We must design the whole system before actually coding any part of it.

We have entering and deleting pretty well under control. Now how about changing an entry? As long as each name has an identification number we can easily read the corresponding record and display each item as it appears, giving ourselves the opportunity to make changes in each case. We will need periodically to print up a list of the names with the IDs. It should be relatively easy to write a program to scan the file from beginning to end, displaying the data in each undeleted record. That program can easily select various categories according to the code stored in the code item.

Let's now design the layout for a data record. See Table 9-1. Note the large value for telephone. That allows for the area code, the number, and a four-digit extension.

The total record comes to 120 characters (119 plus an end-of-line character). We will put all of these data items in one string and use the known length of each individual data item in the string to allow us to manipulate it in any way we choose. We have already manipulated strings in several other programs, so it is not such a formidable task.

DATA ITEM	LABEL	MAXIMUM NUMBER OF CHARACTERS
Identification number	ID#	4
Code	CODE	5
Last name	LAST	20
First name	FRST	20
Address	ADDR	30
City	CITY	16
State	STAT	2
Zip Code	ZIPC	5
Telephone	PHON	17
		119 + 1 = 120

Table 9-1. Record layout for mailing-list file.

We have thought through four functions of our mailing-list system: enter, delete, change, and display. We have also established what each data record will consist of. We can now proceed to create a file for our use. We will create a file containing 101 "blank" records, each 120 characters long, using NOTE to fill an array that tells us where each one starts. We'll save this array in a separate disk file that our main program will read into memory and use. After that, we won't need this initialization program any longer. This is not the only way we could have set up the file, but it is straightforward and easy to understand. Here's our initialization program.

```
90 REM * INITIALIZE MAILING LIST FILE
95 DIM SEC(100),BYT(100),FILLER$(120)
98 FOR I=1 TO 120:FILLER$(I,I)=" ":NEXT I
100 CLOSE #3:OPEN #3,8,0,"D:MAILLIST"
102 FOR MARK=0 TO 100
105 NOTE #3,SECTOR,BYTE
110 SEC(MARK)=SECTOR:BYT(MARK)=BYTE
115 PRINT #3;FILLER$
118 NEXT MARK
120 CLOSE #3
122 OPEN #3,12,0,"D:MAILLIST"
124 POINT #3,SEC(0),BYT(0)
126 PRINT #3;1
128 PRINT #3;0
130 CLOSE #3
140 OPEN #3,8,0,"D:FINDFILE"
150 FOR I=0 TO 100
160 PRINT #3;SEC(I):PRINT #3;BYT(I)
170 NEXT I
180 CLOSE #3
```

Program 9-8. Initialize mailing-list file.

Notice that we use OPEN-to-write (8) to create the file and then CLOSE and OPEN-to-read/write (12) to place two values in the zeroth record. The value 1 refers to the next available record (which is 1, since we are just initializing the file and do not yet have any records). Similarly, since we have not yet deleted anything, the value 0 refers to the most recently deleted record. We then saved all the sector and byte values in our arrays to a file named FINDFILE.

If we are careful about listing all of the considerations discussed so far, we will have the structure of the control routine for our name-and-address entry program. Once we have the control routine we may concentrate on a single subroutine at a time. The functions include:

1. Read data labels.
2. Read available space parameters.
3. Display next available ID and request data.
4. Terminate on null LAST name.
5. Prepare available space.
6. Write new entry.
7. Write available space information back to record 0.
8. Do it again.

Six of the eight tasks in our list are appropriate for subroutines. Some of those subroutines will also be used by the other programs that we will be writing for our name-and-address system. For number 4 to terminate on null LAST name we need to provide a way for the data requesting routine to send back a signal to quit. Number 8 will simply direct the program to repeat the procedures again, beginning with number 3.

We may arbitrarily select line numbers for the subroutines and for the control routine itself, and we will have our program substantially completed. See Program 9-9a.

```
200 GOSUB 1000:REM * READ DATA LABELS
210 GOSUB 900:REM READ AVAILABLE SPACE
PARAMETERS
220 GOSUB 800:REM * DISPLAY NEXT AVAILABLE ID
AND REQUEST DATA
230 IF E1=1 THEN END :REM * TERMINATE ON NULL
LAST NAME
240 GOSUB 700:REM * PREPARE AVAILABLE SPACE
250 GOSUB 600:REM * WRITE NEW ENTRY
260 GOSUB 500:REM * WRITE AVAILABLE SPACE INFO
 BACK TO RECORD ZERO
270 GOTO 220:REM * DO IT AGAIN
```

Program 9-9a. Control routine for mailing-list program.

We have six subroutines and two control statements in our main routine of Program 9-9a. Line 230 requires that the value of E1 be set to 1 if the operator desires to exit and set to any other value for any entry that is to be placed in the file.

Line 270 simply uses a GOTO statement to repeat the request for another new entry. We will now write the subroutines one at a time.

We read the data labels at 1000. If we give some more thought to how to design the routine to take data from the keyboard, we should be able to come up with a creative scheme. We could surely ask the eight questions in eight statements, using INPUT requests with prompts. For each of the eight inputs we could have a statement that checks to see if the entry is too long. Any changes in the file design will require changing that routine. Wouldn't it be a good idea to put the prompt labels and the maximum field sizes in DATA statements and read them into two arrays? Then major changes in the program can be made with simple changes in the DATA statements. Our DATA statements will come directly from the labels and character limits in Table 9-1. We can read the DATA into two arrays with a FOR . . . NEXT loop. See Program 9-9b.

```
 998 REM * READ DATA LABELS AND LIMITS
1000 READ NØ
1010 FOR X9=1 TO NØ
1020 START=(X9-1)*4+1:FINISH=START+3
1022 READ A$,X
1025 LA$(START,FINISH)=A$
1028 LE(X9)=X
1030 NEXT X9
1090 RETURN
1998 REM * DATA LABEL & LIMITS
2000 DATA 9
2005 DATA ID #,4
2010 DATA CODE,5
2015 DATA LAST,20
2020 DATA FRST,20
2025 DATA ADDR,30
2030 DATA CITY,16
2035 DATA STAT,2
2040 DATA ZIPC,5
2045 DATA PHON,17
```

Program 9-9b. Read the data labels for mailing-list program.

In Program 9-9b, N0 is the number of data items in an entry. The labels are stored in the LA$ string and the number of characters permitted for each label is stored in the LE array, both of which must be DIMensioned at the start of our full program.

The subroutine for reading the available space parameters is very simple. It just reverses the action of the initialization program. We need to select variables for the two available space values. See Program 9-9c.

```
 898 REM * READ AVAILABLE SPACE AND SECTOR/BYTE
     FILE
 900 CLOSE #3:OPEN #3,12,Ø,F$
 910 INPUT #3;NS
```

```
920  INPUT #3;DS
940  CLOSE #3
950  OPEN #3,4,0,"D:FINDFILE"
955  FOR I=0 TO 100
960  INPUT #3;SECTOR
965  SEC(I)=SECTOR
970  INPUT #3;BYTE
975  BYT(I)=BYTE
980  NEXT I
985  CLOSE #3
990  RETURN
```

Program 9-9c. Read available space in mailing-list program.

In Program 9-9c we have chosen to carry the new space in the variable NS and the deleted space value in DS. Notice that we did not need to use POINT to get these values, since they are in the first record; the OPEN statement in line 900 automatically positions the pointer at the start of the first record. Also, F$, the file name, is assigned for added flexibility. FINDFILE, referred to in line 950, is the file we created to initialize our random-access disk storage area.

Now it is time to display the next available ID and request data. We said we would do this at 800. Since we have planned carefully, this will be very straightforward. The first job here is to determine the next actual available space. We choose to first make it new space. Then if there is any deleted space we reassign DS to the ID. We handle the label display and the data request with a FOR . . . NEXT loop. See Program 9-9d.

```
798  REM * PROCESS DATA ENTRY FROM KEYBOARD
800  ID=NS:IF DS<>0 THEN ID=DS
803  PRINT
805  PRINT LA$(1,4);": ";ID
808  A$=STR$(ID)
809  IF LEN(A$)<4 THEN A$(LEN(A$)+1)=" ":GOTO
     809
810  DA$(1,4)=STR$(ID)
815  FOR I9=2 TO N0
818  START=(I9-1)*4+1:FINISH=START+3
820  PRINT LA$(START,FINISH);"? ";
825  INPUT A$
828  IF I9=3 THEN IF LEN(A$)=0 THEN E1=1:GOTO
     890
830  IF LEN(A$)<=LE(I9) THEN 840
835  PRINT "TOO LONG":PRINT "      : ";:GOTO 825
840  IF LEN(A$)<LE(I9) THEN A$(LEN(A$)+1)=" "
     :GOTO 830
845  DA$(LEN(DA$)+1)=A$
850  NEXT I9
855  E1=0
890  RETURN
```

Program 9-9d. Handle keyboard data entry for mailing-list program.

Note that in line 828 we set E1 to 1 if the response to the request for LAST name is of zero length. This will be the length if the program user simply presses the RETURN key without any preceding characters. We must include DA$( ) in the DIMension statement in the completed program.

Notice also that in line 840 we add spaces to each entry to bring it up to the maximum length for that item. In this way we ensure that each item is located in the correct position in our DA$ string. This will make it possible to modify records easily.

Next we must prepare available space. What we do here depends on whether we are going to replace a deleted entry or write a new record. If we are going to use a new record we simply add 1 to the new space variable and RETURN. If we are going to write this data to a previously deleted record then we must retrieve the record number that was written there when the deletion occurred. That number is essential for accurately maintaining the available space catalog. Remember this from Figure 9-4.

Next we'll write the subroutine that determines if we are to write to a new space or reuse the space of a deleted record. We check the value of DS to do this. If it is 0, then there are no deleted records. If it is not 0, we find the record with that identification and save that value to be used for our next write.

```
698 REM * IF THIS ENTRY REPLACES DELETED DATA
MAKE PREPARATIONS
700 IF DS=0 THEN 760
710 CLOSE #3:OPEN #3,12,0,F$
720 POINT #3,SEC(DS),BYT(DS)
730 INPUT #3;A$
735 DS=VAL(A$(1,4))
740 CLOSE #3
750 GOTO 790
760 NS=NS+1
790 RETURN
```

Program 9-9e. Prepare available space for mailing-list program.

Note that in this subroutine either new space changes or deleted space changes, but never both.

Once the available space situation is taken care of, we may actually write the entry to the file. This is a very short subroutine with no particular complications.

```
598 REM * WRITE ENTRY
600 CLOSE #3:OPEN #3,12,0,F$
610 POINT #3,SEC(ID),BYT(ID)
620 PRINT #3;DA$
650 CLOSE #3
660 DA$=" "
690 RETURN
```

Program 9-9f. Write data entry in the mailing-list program.

In line 610 we use POINT to indicate the location where we will write the record. After writing we clear the DA$ string to prepare it for the next entry (line 660).

Last but by no means least, we must provide the subroutine that writes the available space parameters to record 0. This is exactly like the initialization program, except that we must write NS and DS. This keeps track of the number of records in the file and the number of the last deleted record. See Program 9-9g.

```
498 REM * WRITE AVAILABLE SPACE DATA
500 CLOSE #3:OPEN #3,12,0,F$
510 PRINT #3;NS
520 PRINT #3;DS
530 CLOSE #3
590 RETURN
```

Program 9-9g. Write available space parameters in mailing-list program.

Finally, in order for all this to happen, we must DIMension the required arrays and strings and define F$.

```
10 REM * ID=>ENTRY IDENTIFICATION NO.
11 REM * NS=>NEW SPACE
12 REM * DS=>DELETED SPACE
15 RECLEN=120:ITEMS=9
20 DIM DA$(RECLEN),LE(ITEMS),LA$(4*ITEMS)
25 DIM F$(10),A$(30),SEC(100),BYT(100)
30 F$="D:MAILLIST"
```

Program 9-9h. Program parameters for mailing-list program.

Line 15 defines the record length (RECLEN) and the number of items in each record (ITEMS). These values are then used in line 20 to DIMension the DA$ string and the LE array. This makes it very easy to change the record size for another mailing-list application. Line 30 assigns the file name to F$. Again, this makes it easy to change the program to work with another name-and-address file.

```
10 REM * ID=>ENTRY IDENTIFICATION NO.
11 REM * NS=>NEW SPACE
12 REM * DS=>DELETED SPACE
15 RECLEN=120:ITEMS=9
20 DIM DA$(RECLEN),LE(ITEMS),LA$(4*ITEMS)
25 DIM F$(10),A$(30),SEC(100),BYT(100)
30 F$="D:MAILLIST"
200 GOSUB 1000:REM * READ DATA LABELS
210 GOSUB 900:REM * READ AVAILABLE SPACE
PARAMETERS
220 GOSUB 800:REM * DISPLAY NEXT AVAILABLE ID
AND REQUEST DATA
230 IF E1=1 THEN END :REM * TERMINATE ON NULL
LAST NAME
240 GOSUB 700:REM * PREPARE AVAILABLE SPACE
250 GOSUB 600:REM * WRITE NEW ENTRY
260 GOSUB 500:REM * WRITE AVAILABLE SPACE INFO
  BACK TO RECORD ZERO
```

```
270 GOTO 220:REM * DO IT AGAIN
498 REM * WRITE AVAILABLE SPACE DATA
500 CLOSE #3:OPEN #3,12,0,F$
510 PRINT #3;NS
520 PRINT #3;DS
530 CLOSE #3
590 RETURN
598 REM * WRITE ENTRY
600 CLOSE #3:OPEN #3,12,0,F$
610 POINT #3,SEC(ID),BYT(ID)
620 PRINT #3;DA$
650 CLOSE #3
660 DA$=" "
690 RETURN
698 REM * IF THIS ENTRY REPLACES DELETED DATA
MAKE PREPARATIONS
700 IF DS=0 THEN 760
710 CLOSE #3:OPEN #3,12,0,F$
720 POINT #3,SEC(DS),BYT(DS)
730 INPUT #3;A$
735 DS=VAL(A$(1,4))
740 CLOSE #3
750 GOTO 790
760 NS=NS+1
790 RETURN
798 REM * PROCESS DATA ENTRY FROM KEYBOARD
800 ID=NS:IF DS<>0 THEN ID=DS
803 PRINT
805 PRINT LA$(1,4);": ";ID
808 A$=STR$(ID)
809 IF LEN(A$)<4 THEN A$(LEN(A$)+1)=" ":GOTO
809
810 DA$(1,4)=STR$(ID)
815 FOR I9=2 TO N0
818 START=(I9-1)*4+1:FINISH=START+3
820 PRINT LA$(START,FINISH);"? ";
825 INPUT A$
828 IF I9=3 THEN IF LEN(A$)=0 THEN E1=1:GOTO
890
830 IF LEN(A$)<=LE(I9) THEN 840
835 PRINT "TOO LONG":PRINT "    : ";:GOTO 825
840 IF LEN(A$)<LE(I9) THEN A$(LEN(A$)+1)=" "
:GOTO 840
845 DA$(LEN(DA$)+1)=A$
850 NEXT I9
855 E1=0
890 RETURN
898 REM * READ AVAILABLE SPACE AND SECTOR/BYTE
 FILE
900 CLOSE #3:OPEN #3,12,0,F$
```

```
910  INPUT #3;NS
920  INPUT #3;DS
940  CLOSE #3
950  OPEN #3,4,0,"D:FINDFILE"
955  FOR I=0 TO 100
960  INPUT #3;SECTOR
965  SEC(I)=SECTOR
970  INPUT #3;BYTE
975  BYT(I)=BYTE
980  NEXT I
985  CLOSE #3
990  RETURN
998  REM * READ DATA LABELS AND LIMITS
1000 READ NØ
1010 FOR X9=1 TO NØ
1020 READ A$,X
1025 LE(X9)=X
1030 START=(X9-1)*4+1:FINISH=START+3
1035 LA$(START,FINISH)=A$
1040 NEXT X9
1090 RETURN
1998 REM * DATA LABEL & LIMITS
2000 DATA 9
2005 DATA ID #,4
2010 DATA CODE,5
2015 DATA LAST,20
2020 DATA FRST,20
2025 DATA ADDR,30
2030 DATA CITY,16
2035 DATA STAT,2
2040 DATA ZIPC,5
2045 DATA PHON,17
```

Program 9-9. Entering names in a mailing-list file.

Program 9-9 is intended to be a simple example of a workable mailing-list data entry program. Using the preceding discussion and some of the routines of this program you should be able to develop programs to delete entries, change entries, and print mailing labels.

There are many ways in which this program can be made more flexible. We might change the design a little to place the record size in record 0 of our file along with the available-space information already there. We might request the mailing-list file name from the program operator. We might eliminate the DATA statements from the program by placing the data in a companion file. The benefits of doing things this way are tremendous. With all of the information about the mailing list stored in a file, our one program can be used to process may different mailing lists. We can handle different numbers of items in a record, we can handle different sets of item size limits, we can handle different record sizes, and we can handle different labels, all with the same program. We will soon find that we have to write

a program to manage the companion file that contains all of this useful information. That is a small price to pay. When we can change the behavior of a program by changing data in a file, we approach data base management capabilities.

Programming for the delete and change functions can be handled either by writing separate programs or by including the new subroutines necessary right in Program 9-9. We could provide a menu that lets the user select which function is desired.

...SUMMARY

Using NOTE and POINT gives us the ability to locate the start of each record in a disk file. This offers a tremendous advantage over sequential files. We can read the 200th entry just as quickly as we can read the first.

To use random-access files we first create a file that has only blank records in it, using NOTE to establish an array that contains the beginning location of each record. When we want to read or write any particular record, we are then able to use POINT to access the start of that record. NOTE and POINT only work if the file is first opened with

```
OPEN #N,12,0,"D:FILENAME.EXT
```

where N is a number from 1 to 5. The form of NOTE and POINT are then

```
NOTE #N,A,B
```

```
POINT #N,A,B
```

where A is the sector and B is the byte of the record of interest.

Problems for Section 9-4. . . . .

1. Incorporate a delete routine in the name-and-address-entry program.
2. Write a program to edit data in the mailing-list file. Display each item and ask if the user wants to make a change.
3. Write a program to display all data from the file for names having a specified code.

PROGRAMMER'S CORNER 9

...DOS and DUP in Detail

Typing DOS and pressing the RETURN key gets you to the DOS menu. As we have previously warned, this also erases any file currently in the computer's memory. We have already used the I and H selections in the menu to format and write the DOS.SYS and DUP.SYS files onto disks. Now let's look at all the features available in more detail.

```
DISK OPERATING SYSTEM II  VERSION 2.0S

A. DISK DIRECTORY  I. FORMAT DISK
B. RUN CARTRIDGE   J. DUPLICATE DISK
C. COPY FILE       K. BINARY SAVE
D. DELETE FILE(S)  L. BINARY LOAD
E. RENAME FILE     M. RUN AT ADDRESS
F. LOCK FILE       N. CREATE MEM.SAV
G. UNLOCK FILE     O. DUPLICATE FILE
H. WRITE DOS FILES
```

Figure 9-5. The DOS II menu.

...Wild Cards (* and ?)

We'll first cover an item not on the menu—the use of "wild cards" in file names. Many of the menu selections permit the use of these wild cards. They permit file names to be selected without typing in the entire name and they permit us to select more than one file name by using just one name that contains wild cards.

There are two different wild cards, which are selected by using the asterisk (*) and question mark (?). The asterisk stands for any combination of characters and the question mark stands for a single character. Let's look at an example in which we have the following file names on a disk:

FILE1.BAS	FILE1
FILE2.BAS	FILE.LST
FILE11.BAS	FILE.1S
FILE12.BAS	FILE1

All of these are legal file names. We can, of course, select any of them by typing the full name. We can select all of them with *.*, where the asterisks mean that any legal combination of letters and numbers will be read. We can select all of those that end with .BAS by using *.BAS, which will not select any from the second column. When an asterisk is used as a wild card, all subsequent characters in the name or extension are ignored. The decimal point in the file name ends the effect of the asterisk as a wild card. Thus, *1.BAS and *.BAS are the same, as are FILE.*ST and FILE.*. The question mark as a wild card allows any character in its position. FILE11.BAS and FILE12.BAS will be chosen if FILE1?.BAS is specified.

You are likely to find many more uses for the asterisk wild card, particularly its ability to select all file names on a disk with *.*, and also its ability to fill in the end of a file name when you aren't sure what it is. Not all of the DOS menu selections support the use of wild cards.

DOS MENU OPTION	WILD CARDS PERMITTED
A. Disk Directory	Yes
B. Run Cartridge	No
C. Copy File	Yes
D. Delete File	Yes
E. Rename File	Yes
F. Lock File	Yes

G.	Unlock File	Yes
H.	Write DOS File	No
I.	Format Disk	No
J.	Duplicate Disk	No
K.	Binary Save	No
L.	Binary Load	No
M.	Run at Address	No
N.	Create MEM. SAV	No
O.	Duplicate File	Yes

Figure 9-6. DOS menu options that can and cannot use wild cards.

Now let's look at what happens when you select each of the menu items.

...A. Disk Directory

If you select A, you will get the next prompt

```
DIRECTORY--SEARCH SPEC,LIST FILE?
```

Respond by pressing the RETURN key to list on the screen all the files on the disk, how many sectors each occupies, and the number of sectors on the disk not yet used ("FREE SECTORS"). Remember that you can use CTRL and 1 to stop and restart the listing. You could use wild cards in the SEARCH SPEC to list only some of the files on the disk and you also can use P: for the LIST FILE if you have a printer that is turned on to obtain a printout.

...B. RUN CARTRIDGE

Selection B will transfer control back to your BASIC cartridge. If it is not installed, you will get a message to that effect. If you have a MEM.SAV file (see Section N below) on the disk, the disk will load this file before transferring control back to the cartridge.

...C. COPY FILE

The prompt for selection C is

```
COPY-FROM,TO
```

This is useful for copying files from one disk drive to another or for making a backup copy of a file on the same disk (with a different name). You cannot copy DOS.SYS files with this command. If MEM.SAV is on the disk, you will be asked if it is okay to use program area; that doing so will destroy MEM.SAV. To copy a file from disk drive 1 to 2 use

```
D1:FILENAME.EXT,D2:FILENAME.EXT
```

The "D1:" is optional. The second file name need not be the same as the first. You can make a backup on the same disk by responding

```
FILENAME.EXT,FILENAME.BAK
```

or whatever names you want to use.

...D. DELETE FILE

Selecting option D will result in the response

```
DELETE FILESPEC
```

to which you respond with the name of the file to be deleted (wild cards acceptable). The use of D1: is not necessary if you are using drive 1. As a failsafe measure, you will then be asked to verify that you want to delete that file. Adding /N to your choice will bypass this verification.

...E. RENAME FILE

Selection E yields the prompt

```
RENAME, GIVE OLD NAME, NEW
```

Again, the use of D1: is optional for drive 1, in which case you can type

```
OLDNAME.EXT,NEWNAME.EXT
```

...F. LOCK FILE

Selection F will write-protect a single file or, if wild cards are used, several files. Locked files can be transferred but cannot be renamed or written over. However, formatting a disk with the I command will override any write protection from locking files. Locked files are indicated in the disk directory (produced by option A) with an asterisk before the file name. This should not be confused with the asterisk used as a wild card. Attempting to write over a locked file will produce

```
ERROR- 167
```

...G. UNLOCK FILE

Option G will produce the prompt

```
WHAT FILE TO UNLOCK?
```

Respond with the file name, preceded by DN: if drive 1 is not being used. The asterisk in the disk directory listing will be removed.

...H. WRITE DOS FILE

We have already used this command. DOS.SYS and DUP.SYS can be written on disks only by using this option. You will be asked which drive to write DOS to and also to verify your choice.

...I. FORMAT DISKETTE

We have also used this command. You are asked which drive to use and also to verify your choice. It can't be said too strongly that formatting will irrevocably

erase all files from a disk. Such a loss can only be prevented by putting opaque tape over the notch on the edge of the diskette.

...J. DUPLICATE DISK

Option J will produce an exact copy of one disk on another. All files will be placed on the same sectors as they were on the original. This will work with one or two drives. With one drive you will be prompted when to insert the original and when to insert the copy. It is advisable to put opaque tape over the notch in the original to avoid accidentally writing in the wrong direction. Attempting to write to a disk protected in this way will result in ERROR— 144, and no writing will take place.

...K. BINARY SAVE
...L. BINARY LOAD
...M. RUN AT ADDRESS

These three commands are not of much interest to BASIC programmers. They are used with files in the form that the computer understands directly, called *machine language*. You may have occasion to use option L, to load machine language programs. All that is required is that the file name be typed. Binary programs sometimes appear in computer magazines. Nearly all commercially available games and many utility programs are written in this format.

...N. CREATE MEM.SAV

The prompt you will receive if you select option N is

```
TYPE "Y" TO CREATE MEM.SAV
```

Because DOS uses portions of memory that may contain your BASIC program, the ability to create a MEM.SAV file is included in the DOS menu. If your diskette has such a file, typing DOS from BASIC will cause the computer to save the common portions of memory onto a file called MEM.SAV before going to DOS and restoring this portion of memory when you leave DOS with the B option. The trade-off here is that MEM.SAV requires 45 sectors on the disk and it takes time to write and read the MEM.SAV file each time you transfer to or from DOS. Our suggestion is to generally avoid using MEM.SAV by always using

```
SAVE "D:FILENAME.EXT"
```

from BASIC to save the current program before going to DOS. Once you have this habit, you won't need MEM.SAV.

...O. DUPLICATE FILE

The prompt for option O is

```
NAME OF FILE TO MOVE?
```

with another prompt after you specify the file name if you have MEM.SAV on the disk. You will then be prompted when to insert the source and destination disks to allow the transfer to occur. This is the only method that allows a copy of one file or

several files from one disk to another when there is only one disk drive and you don't want to copy the entire disk.

Over a period of time you will become very comfortable with these DOS options and will rarely need any reference material to use them.

...Using DOS Commands from BASIC

Atari BASIC has an XIO command that can be used for many purposes. We will discuss its use for accessing some of the DOS functions from BASIC in this chapter. This command generally has the form

```
XIO CMD,#N,Ø,Ø,"DN:FILENAME.EXT"
```

where N refers, as usual, to the channel number. Note that these commands are an an exception to a general rule, in that OPEN and CLOSE are not used. The new parameter here is CMD, with the following values performing the indicated commands:

CMD	COMMAND
32	RENAME
33	DELETE
35	LOCK FILE
36	UNLOCK FILE
254	FORMAT

With the RENAME command, you would use

```
XIO 32,#1,Ø,Ø,"DN:OLDNAME,NEWNAME"
```

Notice that the "DN:" is used only once. With the FORMAT command the proper form is

```
XIO 254,#1,Ø,Ø,"D:"
```

The other three commands use normal, single file names.

XIO may also substitute for some other BASIC statements, such as OPEN, CLOSE, PUT, and GET. The BASIC names, however, are much more informative and their use is recommended.

...Disk Miscellany

There is a maximum to the number of files that you can store on one disk that is determined by the capacity of the disk directory. This maximum is 64 files, whether they be data files or programs. If you try to store information on a disk that contains 64 files, you will get an ERROR—169 message, indicating that the directory is full. The other maximum you have to contend with is the number of sectors that are available on a disk. An empty, formatted disk has 707 sectors available. If the disk contains DOS.SYS and DUP.SYS you then have 626 sectors available. Once a disk is full you will get ERROR—162 if you attempt to write to the disk.

...Other DOS Systems

DOS II is not the only Disk Operating System that is used on the Atari. DOS III, mentioned in the introduction to this chapter, supports the storage of more information on a diskette (approximately 1.5 times the information). DOS III has menus that are somewhat different from the DOS II menu discussed here. However, the commands it provides are nearly the same. As with DOS II, you can use the "HELP" screens provided to obtain more information.

There are a number of other disk drive manufacturers whose drives are capable of operating in "double density" mode, that is, they are capable of storing twice as much information as DOS II on a diskette. Each one has slightly different instructional menus and each must be studied to learn how they correspond with the DOS II options. It is important to note that all of these disk drives will support DOS II, which is the format used for all Atari disk-based computer programs through most of 1984, but the Atari disk drives manufactured prior to that will support *only* DOS II. The message is this: It is always safe to use single density for storage on diskettes (which is supported by all the various DOS formats); other DOS versions and the use of double density may not be transferrable to another brand of disk drive.

Chapter 10

Sound

Games demand sound. Crashes and whistles and laser sounds all add to the fun of a game. Music by itself or as an introduction to a program contributes to user enjoyment. Sound effects that indicate user choices are also effective. Beeps indicating that a choice has been made, buzzes for incorrect or illegal choices, musical chords for correct choices—the potential uses of sound go on and on. Sound can be as effective an attention grabber as graphics.

10-1...A Versatile Extra

One of the extras built into Atari computers is the ability to make sounds on command. You can instruct the computer to produce pure musical tones or a seemingly endless variety of sound effects. You can have up to four voices singing or rasping or whatever, all at the same time. This is all accomplished with the SOUND statement and only a few rules for using it. The number of possible combinations is tremendous and your experiments will be greatly rewarded.

...Sound from the Computer Speaker
If you have a model 400 or 800 Atari, you also have a small speaker built into the computer. It is from this speaker that the click comes when you press the keyboard on these models. Because of the multiple voices and effects available when you use the SOUND statement through the TV speaker, the computer speaker is of limited value. If you use POKE to enter a 0 into location 53279 with

```
POKE 53279,0
```

you'll hear a click. Send a series of clicks with a FOR . . . NEXT loop and you'll hear a tone. Put a variable pause in the FOR . . . NEXT loop and you can vary the tone.

...Sound from the TV Speaker

All the other sounds you can coax out of your Atari are generated by

```
SOUND A,B,C,D
```

where

A = The voice selected (0–3)
B = The pitch (0–255)
C = The type of tone or distortion (0–15)
D = The loudness (0–15)

All four variables must be included in each SOUND statement. Let's look at each of these variables in more detail.

...Voice

Once again, the computer prefers to start counting from 0. You have four voices to choose from: 0, 1, 2, and 3. You can select any combination of voices to play at any time. Each voice chosen needs a separate SOUND statement.

...Pitch

Values from 0 to 255 are permitted. Small pitch values produce high notes and large pitch values produce low notes. All the 256 pitch values do not correspond to musical notes, in the sense that they do not all correspond to the keys on a piano. Figure 10-1 shows which pitch values do have musical note equivalents and Figure 10-2 matches these values with the keys on a piano. When you're playing music or musical harmony, those are the only pitch values of interest. The entire range, however, is useful for sound effects.

...Distortion

Values for this variable are 0 to 15. Use a value of 10 or 14 for pure tones, that is, when your goal is to produce music rather than distorted sounds. Odd-numbered values are of limited use. There really are no hard-and-fast rules for obtaining specific sound effects. Experimentation is definitely in order!

...Loudness

Values from 0 to 15 produce a range from silence to the loudest volume. This depends, of course, on where you have set the volume control of your TV set. There is a smooth variation from softest to loudest. To keep from introducing unwanted distortions, keep the total volume for all the voices currently in use to no more than 32. So if you happen to be using four voices all at the same volume, the volume should not exceed 8 for each voice.
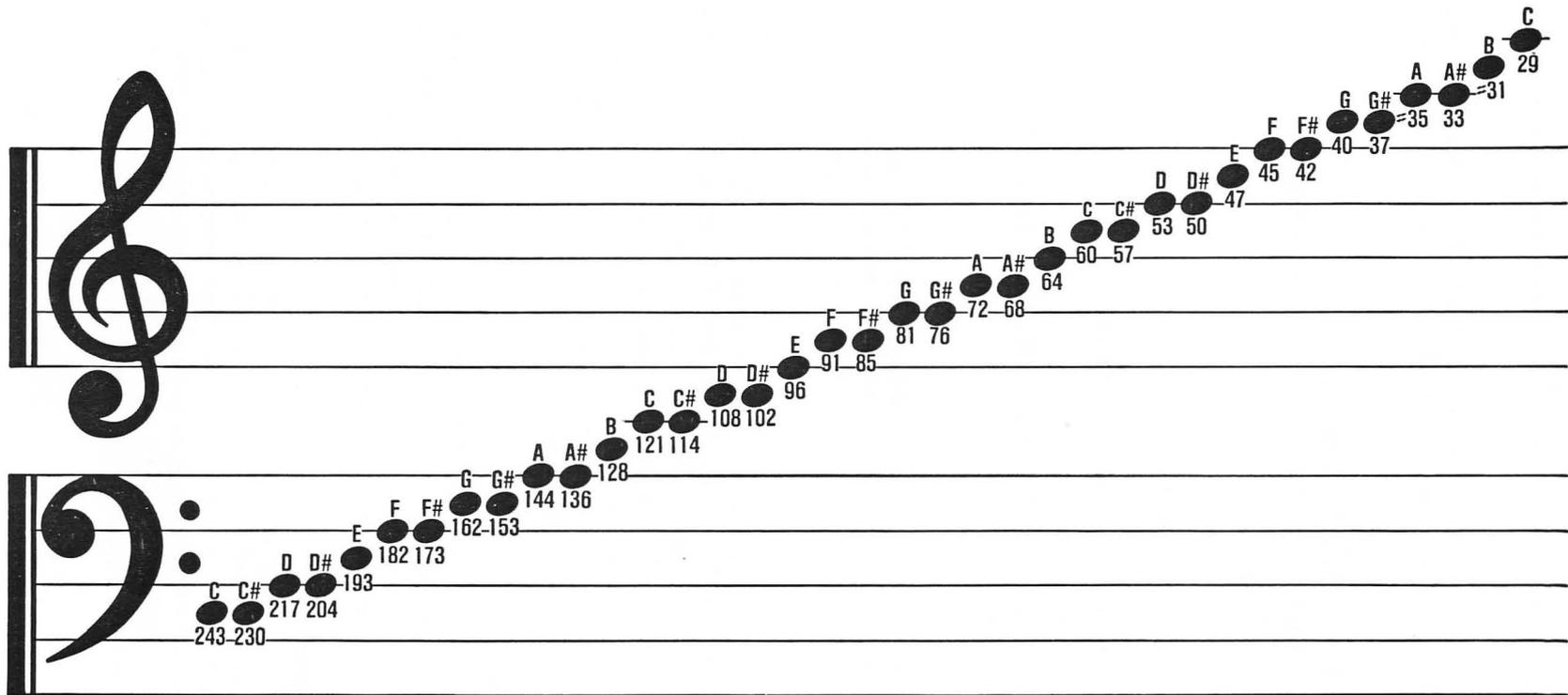
Figure 10-1. Pitch values that have musical equivalents

Figure 10-2. Matching pitch values with the keys on a piano.

10-2...Using SOUND

Let's start at the beginning. Type the following line and press RETURN.

```
SOUND Ø,121,1Ø,8
```

You're hearing middle C in a pure tone with a medium volume. Adjust your TV volume to a comfortable level. Notice that the sound doesn't stop on its own. It will

stop at the end of a program, when you RUN a program, or if you press SYSTEM RESET. It won't stop if you press BREAK. From within a program, you can turn off the sound from a particular voice with

```
SOUND VOICE,Ø,Ø,Ø
```

where VOICE = 0,1,2, or 3 for the voice of interest. In immediate mode, typing END or pressing SYSTEM RESET are the easiest ways to turn off a sound.

We can use a simple FOR . . . NEXT loop to hear some of the sounds that we can generate.

```
90 REM * LOOP TO HEAR ALL TONES
100 FOR I=Ø TO 255
110 SOUND Ø,I,1Ø,8
120 PRINT "PITCH = ";I
130 FOR DELAY=1 TO 25:NEXT DELAY
140 NEXT I
```

Program 10-1. Producing different sounds by varying pitch.

Notice how similar the sounds become as the pitch values increase (tone gets lower).

Let's write a program that allows us to better explore the sounds that can be generated on the Atari. An immediate thought is to use FOR . . . NEXT loops to vary the pitch and distortion. Let's also use another loop to introduce a pause between the first 70 pitch values for each distortion value so that we can hear them better, since we have seen that small changes in pitch produce big changes in sounds for these values.

```
90 REM * SOUND EXPLORER
100 PRINT " ":REM ESC CTRL+CLEAR
105 POKE 752,1
110 POKE 84,8:POKE 85,10
120 PRINT "Sound Explorer"
130 FOR DISTORTION=Ø TO 15 STEP 2
140 FOR PITCH=Ø TO 255
150 SOUND Ø,PITCH,DISTORTION,8
155 POKE 84,10:POKE 85,10
160 PRINT "      PITCH = ";PITCH
170 POKE 85,10
180 PRINT "DISTORTION = ";DISTORTION
190 X=DISTORTION
195 IF PITCH<70 THEN FOR DELAY=1 TO 50:NEXT
DELAY
200 POKE 84,10:POKE 85,10
210 PRINT "      PITCH = ";"    "
220 NEXT PITCH
230 NEXT DISTORTION
```

Program 10-2. Producing different sounds by varying pitch and distortion.

We have used two POKE statements to position our printing on the screen in this

program (see line 110). Notice how much nicer a display this is than having an endless string of numbers moving down the screen. We learned in Chapter 4 about using POKE 85,X to position the cursor horizontally on a line at position X. Here we also use POKE 84, Y to position the cursor vertically in row Y. In line 210 we also print enough blank spaces to erase what was previously printed. This takes care of the case in which, for example, we might have the value 254 on the screen and then want to print 15 starting where the 2 is located. If we don't erase the 254 we will end up with 154, with the 4 left over from the previous printing.

You can explore the sound possibilities of your Atari computer with programs such as this. Let's go on and look at using the musical pitch values. Since we know which of the pitch values correspond to musical notes (from Figure 10-1 and 10-2), we can put just these values into an array and then have a program play the musical notes of our choosing. We'll now put these musical notes into a program that also takes a look at the effect of changes in the loudness of notes on their sound. Variations in loudness (called "attack" for how a tone approaches its maximum volume and "decay" for how it decreases after reaching this maximum) are important in determining whether a sound resembles a musical instrument. We control attack and decay by using a FOR . . . NEXT loop to vary the loudness of a note. It looks like this:

```
100 FOR J=0 TO 10 STEP ATK
110 SOUND 0,PITCH,10,J
120 NEXT J
```

This loop will increase the volume at a rate inversely proportional to the value of ATK. That is, if ATK = 11 there will be one step from silence to maximum volume, and if ATK = 1 there will be 11 steps (0,1, . . . 10). The second case will cause a volume increase we can hear for each note played, resulting in a different sort of sound. A similar loop to decrease the volume is

```
200 FOR J=10 TO 0 STEP -DEC
210 SOUND 0,PITCH,10,J
220 NEXT J
```

Now look at Program 10-3.

```
90 REM * EFFECT OF ATTACK AND DECAY
92 GRAPHICS 0
95 DIM A(37)
100 FOR X=1 TO 37
110 READ Y
120 A(X)=Y
130 NEXT X
200 READ ATK,DEC
205 PRINT "INCREMENT OF ATTACK = ";ATK
206 PRINT "INCREMENT OF DECAY  = ";DEC
207 PRINT
210 IF ATK=-1 THEN 480
220 FOR I=1 TO 37
```

```
230 FOR K=Ø TO 1Ø STEP ATK
240 SOUND Ø,A(I),1Ø,K
250 NEXT K
300 FOR K=1Ø TO Ø STEP -DEC
310 SOUND Ø,A(I),1Ø,K
320 NEXT K
330 FOR DELAY=1 TO 25:NEXT DELAY
335 SOUND Ø,Ø,Ø,Ø
340 NEXT I
350 GOTO 2ØØ
480 SOUND Ø,Ø,Ø,Ø
550 END
1ØØØ DATA 243,23Ø,217,2Ø4,193,182,173
1Ø1Ø DATA 162,153,144,136,128,121
1Ø2Ø DATA 114,1Ø8,1Ø2,96,91,85,81,76
1Ø3Ø DATA 72,68,64,6Ø,57,53,5Ø,47,45
1Ø4Ø DATA 42,4Ø,37,35,33,31,29
11ØØ DATA 11,11,5,5,2,2,1,1,11,5,11,2,11,1,1,
11,1,5,1,2,-1,Ø
```

Program 10-3. Producing different sounds by varying attack and decay.

The data in lines 1000 to 1040 are the pitch values that are the musical notes read into the array A( ). The DATA in line 1100 is used for ATK and DEC to vary the sound of the tones. If ATK = −1 the program stops. We PRINT the values of ATK and DEC on the screen each time. We also have put a delay loop at line 330 so the notes don't go by too fast. You can easily try out different attack and decay rates by changing the DATA.

It should be noted that you will get variations in these sounds in a long program, depending on where the FOR . . . NEXT loop is located in the program. This is because Atari BASIC starts from the beginning of the program when looking for line numbers. Therefore, when timing is important, it is advisable to put loops like this at the start of your program and access them as subroutines when they are needed.

So far we have looked only at single sounds. Sound effects and music are generally combinations of sounds. We have four sounds to combine and so the variations are pretty much inexhaustible. A chord is merely a group of notes played together. In conventional music, chords sound best if the four notes chosen are separated on the piano keyboard so that the gaps between the four notes of the chord are three, two, and four notes. Again, we'll make use of our musical note array to select the pitch values that correspond to the notes of a chord. Using the spacing mentioned above, this means that a chord will consist of PITCH, PITCH + 4, PITCH + 7, and PITCH + 12 from our array.

For variety, we'll use input from the keyboard to determine whether the chord gets higher or lower in pitch. We'll use "+" and "*" to move down and up the scale. This is logical since these keys also have the left and right arrows on them. We'll also want to be careful that we don't try to exceed the number of "notes" in our array, leading to an error message and termination of the program. Since we know that the

last note of each chord is 13 notes past the first, we'll check to be sure that we stop playing chords before we get there. We'll also want to be sure we don't get negative numbers, for the same reason. Our program will READ in the notes from DATA statements and print some instructions on the screen. We'll also want to print the pitch number for the first note in the chord so we'll have some visual reference.

```
90 REM * MUSICAL TONES
92 GRAPHICS 0
95 DIM A(37)
100 FOR X=1 TO 37
110 READ Y
120 A(X)=Y
130 NEXT X
170 PITCH=20
176 PRINT "Press '*' (right arrow) to go higher"
177 PRINT "Press '+' (left arrow) to go lower"
178 PRINT "Press 'S' TO STOP"
180 CLOSE #1:OPEN #1,4,0,"K"
190 GET #1,X
195 IF X<>42 AND X<>43 AND CHR$(X)<>"S" AND CHR$
(X)<>"s" THEN 190
200 IF X=42 THEN PITCH=PITCH+1
210 IF PITCH>25 THEN PITCH=25
220 IF X=43 THEN PITCH=PITCH-1
230 IF PITCH<1 THEN PITCH=1
240 IF CHR$(X)="S" OR CHR$(X)="s" THEN 500
250 SOUND 0,A(PITCH),10,6
260 SOUND 1,A(PITCH+4),10,6
270 SOUND 2,A(PITCH+7),10,6
280 SOUND 3,A(PITCH+12),10,6
285 PRINT "First note of chord = ";A(PITCH)
290 GOTO 180
500 FOR I=0 TO 3
510 SOUND I,0,0,0
520 NEXT I
550 END
1000 DATA 243,230,217,204,193,182,173
1010 DATA 162,153,144,136,128,121
1020 DATA 114,108,102,96,91,85,81,76
1030 DATA 72,68,64,60,57,53,50,47,45
1040 DATA 42,40,37,35,33,31,29
```

Program 10-4. Producing musical chords.

If you type in this program and RUN it, you will be rewarded with some nice melodic chords by pressing the "+" and "*" keys. In lines 190 to 195 we check for which key is pressed and reject all keys except "*," "+," "S," and "s." Notice lines 250 to 280, where we select the notes to have the correct intervals between one another. Also notice the loop in lines 500 to 520, which is used to stop all four voices before ending the program when the user presses "S" or "s".

10-3...A Sound Generator Program

We can combine the SOUND statement with our prior knowledge to write a useful sound generator program. We'll use a joystick for our input and vary the pitch to higher values by moving the stick forward and lower values by moving it backward, which seems to be the natural way to do it. Because there are so many pitch values to cover, we'll also put in a provision that we'll move in pitch increments of 10 unless the button is pushed when the stick is moved forward or backward, in which case we'll move in single pitch steps. We'll move the joystick left to select which voice we are working with. Moving the joystick right will change the volume, and pressing the button and moving the stick right will turn off that particular voice. Of course, we'll want to display all of this information on the screen, so that when we find combinations of interest, we can make a note of the values. We'll make use of our ability to position the cursor by using POKE statements to make a nicely organized screen display. This all seems like a tall order to program, but we can easily break it up in to convenient subroutines to handle the various chores.

Let's look at a few of the routines we need. Here's our initialization section:

```
90 REM * JOYSTICK SOUND EDITOR
91 DIM BLANK$(11),LOUD(4),PITCH(4)
92 FOR I=1 TO 11:BLANK$(I,I)=" ":NEXT I
93 FOR I=0 TO 3:LOUD(I)=0:NEXT I
94 FOR I=0 TO 3:PITCH(I)=125:NEXT I
95 POKE 752,1:POKE 82,1
96 FWARD=14:BWARD=13:LEFT=11:RIGHT=7
97 DIS=10:REM DISTORTION=10
98 GOSUB 3000
```

Program 10-5a. Initialize arrays for joystick sound editor program.

We will be using BLANK$ to erase areas of the screen. The two arrays, LOUD and PITCH, will be used to keep track of loudness and pitch for each of the four voices. By using the zero subscript values of the arrays (since the computer starts counting with zero), we can match the voice number perfectly; and this will help in keeping all the values and displays straight. Thus, VOICE (0) will have PITCH (0) and LOUD (0) for its values. Notice that we have assigned useful variable names for the joystick directions to aid our understanding (line 96). Now let's look at the input routine.

```
100 X=STICK(0):T=STRIG(0)
105 IF PEEK(53279)=5 THEN 4000
110 IF X=FWARD AND T=0 THEN STEP=1:GOSUB 500
115 IF X=FWARD AND T=1 THEN STEP=10:GOSUB 500
120 IF X=BWARD AND T=0 THEN STEP=1:GOSUB 600
125 IF X=BWARD AND T=1 THEN STEP=10:GOSUB 600
130 IF X=LEFT THEN 700
140 IF X=RIGHT AND T=1 THEN GOSUB 800
```

196

```
142 IF X=RIGHT AND T=0 THEN LOUD(VOICE)=0
:GOSUB 830
145 GOSUB 2000
150 GOTO 100
```

Program 10-5b. Input routine for joystick sound editor program.

The X tells us the direction the joystick is moved. T tells us if the button has been pressed (T = 0 means button pressed). We have also provided a means of exiting from the program by using PEEK(53279) to check to see if the SELECT key has been pressed (line 105). A clean exit from a program is always a worthwhile feature.

In order to make for a useful, changeable screen display we will use some interesting programming. We want to display four lines of changing information (for the four voices), and we want the display to reflect all changes we make, without changing values that are left the same. We also need some indication of which voice is being worked on at the moment. We'll use a flashing arrow to mark the voice in use, and we'll make it flash by alternately printing the arrow and erasing it. For our arrow, we'll use the arrow key itself. We use this in a program by preceding it with the ESC key and we get that in a program by pressing ESC twice, so that the key sequence becomes ESC ESC ESC CTRL and arrow (see line 2030 below).

```
1998 REM * BLINKING ARROW SUBROUTINE
2000 POKE 84,8+VOICE:POKE 85,1
2010 PRINT " "
2015 FOR DELAY=1 TO 10:NEXT DELAY
2020 POKE 84,8+VOICE:POKE 85,1
2030 PRINT " ▆→"
2040 RETURN
```

Program 10-5c. Screen arrow display routine for joystick sound editor.

We have made use of variables in our POKE statements to position the cursor on line 8 + VOICE before printing the arrow. Thus, line 8 on the screen is for Voice 0, line 9 for Voice 1, etc. We then use POKE to enter the same values again to put the cursor back on the same line and print a blank to erase the arrow. This alternation will make a blinking arrow.

We'll look now at one of the typical subroutines that is called in response to the joystick position. Here's the routine to increase the pitch.

```
498 REM * INCREASE PITCH SUBROUTINE
500 PITCH(VOICE)=PITCH(VOICE)-STEP
502 IF PITCH(VOICE)<0 THEN PITCH(VOICE)=0
504 IF PITCH(VOICE)>255 THEN PITCH(VOICE)=255
510 SOUND VOICE,PITCH(VOICE),DIS,LOUD(VOICE)
520 HPOS=13
530 GOSUB 1000
540 PRINT "PITCH ";PITCH(VOICE)
550 RETURN
```

Program 10-5d. Pitch change routine for joystick sound editor program.

197

The value of STEP in line 500 depends on whether we pressed the button (STEP = 1) or not (STEP = 10). We check in lines 502 and 504 to be sure we only get legal values (0 to 255), then play that pitch in line 510. HPOS in line 520 is the horizontal position where we want to print the new pitch value on the screen. We first erase the printing previously at the location. We make this erasing routine a separate subroutine at line 1000, since we need to use it in several places. Here's that routine:

```
998  REM * ERASE ROUTINE
1000 POKE 84,8+VOICE:POKE 85,HPOS
1010 PRINT BLANK$(1,11)
1020 POKE 84,8+VOICE:POKE 85,HPOS
1030 RETURN
```

Program 10-5e. Erase screen routine for joystick sound editor program.

Here we use HPOS to position the cursor, print a blank to erase what was on the screen by printing BLANK$, which is just a string of 11 blank spaces, reposition the cursor, and then RETURN. Thus this is a general routine that only needs the value of HPOS to erase an old value and get ready to print a new one. We actually made it general by making all the printing the same length on the screen (11 characters long).

The other joystick response subroutines are constructed similarly to the routine for increasing pitch that we've looked at. Here now is the entire program, including a subroutine at line 3000 that prints instructions and sets up the initial display. A program that makes such varied used of the joystick requires some directions on the screen at all times to aid the user.

```
90  REM * JOYSTICK SOUND EDITOR
91  DIM BLANK$(11),LOUD(4),PITCH(4)
92  FOR I=1 TO 11:BLANK$(I,I)=" ":NEXT I
93  FOR I=0 TO 3:LOUD(I)=0:NEXT I
94  FOR I=0 TO 3:PITCH(I)=125:NEXT I
95  POKE 752,1:POKE 82,1
96  FWARD=14:BWARD=13:LEFT=11:RIGHT=7
97  DIS=10:REM * DISTORTION=10
98  GOSUB 3000
100 X=STICK(0):T=STRIG(0)
105 IF PEEK(53279)=5 THEN 4000
110 IF X=FWARD AND T=0 THEN STEP=1:GOSUB 500
115 IF X=FWARD AND T=1 THEN STEP=10:GOSUB 500
120 IF X=BWARD AND T=0 THEN STEP=1:GOSUB 600
125 IF X=BWARD AND T=1 THEN STEP=10:GOSUB 600
130 IF X=LEFT THEN GOSUB 700:GOTO 150
140 IF X=RIGHT AND T=1 THEN GOSUB 800
142 IF X=RIGHT AND T=0 THEN LOUD(VOICE)=0
    :GOSUB 830
145 GOSUB 2000
150 GOTO 100
498 REM * INCREASE PITCH SUBROUTINE
```

```
500 PITCH(VOICE)=PITCH(VOICE)-STEP
502 IF PITCH(VOICE)<0 THEN PITCH(VOICE)=0
504 IF PITCH(VOICE)>255 THEN PITCH(VOICE)=255
510 SOUND VOICE,PITCH(VOICE),DIS,LOUD(VOICE)
520 HPOS=13
530 GOSUB 1000
540 PRINT "PITCH ";PITCH(VOICE)
550 RETURN
598 REM * DECREASE PITCH SUBROUTINE
600 PITCH(VOICE)=PITCH(VOICE)+STEP
610 IF PITCH(VOICE)<0 THEN PITCH(VOICE)=0
620 IF PITCH(VOICE)>255 THEN PITCH(VOICE)=255
625 SOUND VOICE,PITCH(VOICE),DIS,LOUD(VOICE)
630 HPOS=13
640 GOSUB 1000
650 PRINT "PITCH ";PITCH(VOICE)
660 RETURN
698 REM * CHANGE VOICE SUBROUTINE
700 HPOS=1
710 GOSUB 1000
715 PRINT " VOICE ";VOICE
720 VOICE=VOICE-1
722 IF VOICE<0 THEN VOICE=3
725 HPOS=1
730 GOSUB 1000
740 PRINT " VOICE ";VOICE
750 RETURN
798 REM * CHANGE VOLUME SUBROUTINE
800 LOUD(VOICE)=LOUD(VOICE)+1
810 IF LOUD(VOICE)<0 THEN LOUD(VOICE)=0
820 IF LOUD(VOICE)>15 THEN LOUD(VOICE)=0
830 SOUND VOICE,PITCH(VOICE),DIS,LOUD(VOICE)
840 HPOS=24
850 GOSUB 1000
860 PRINT "LOUDNESS ";LOUD(VOICE)
870 RETURN
998 REM * ERASE ROUTINE
1000 POKE 84,8+VOICE:POKE 85,HPOS
1010 PRINT BLANK$(1,11)
1020 POKE 84,8+VOICE:POKE 85,HPOS
1030 RETURN
1998 REM * BLINKING ARROW SUBROUTINE
2000 POKE 84,8+VOICE:POKE 85,1
2010 PRINT " "
2015 FOR DELAY=1 TO 10:NEXT DELAY
2020 POKE 84,8+VOICE:POKE 85,1
2030 PRINT " ▌→ "
2040 RETURN
2998 REM * INITIAL SCREEN DISPLAY
3000 PRINT "♪":REM ESC CTRL+CLEAR
```

```
3010 PRINT "      JOYSTICK SOUND EDITOR"
3025 PRINT "UP INCREASES PITCH, DOWN DECREASES
  IT."
3030 PRINT "BUTTON + UP OR DOWN USES SMALL
STEPS."
3035 PRINT "LEFT CHANGES VOICES."
3037 PRINT "RIGHT CHANGES VOLUME."
3040 PRINT "RIGHT + BUTTON SILENCES THAT
VOICE."
3050 POKE 84,8:POKE 85,1
3060 FOR I=0 TO 3
3070 PRINT " VOICE ";I;"      PITCH 125
LOUDNESS 0"
3080 NEXT I
3090 POKE 84,15:POKE 85,12
3100 PRINT "DISTORTION = ";DIS
3200 PRINT
3210 POKE 85,10
3220 PRINT "PRESS SELECT TO QUIT"
3230 RETURN
3998 REM * EXIT PROGRAM
4000 FOR I=0 TO 3
4010 LOUD(I)=0
4020 NEXT I
4030 GRAPHICS 0
4040 END
```

Program 10-5. Joystick sound editor program.

The program is written to produce combinations of pure tones (note the DIS = 10 value in line 97). Simply by changing this value you can use this program to explore other, more unusual sounds.

10-4...Sound Effects

You've heard some of the strange sounds that can be generated by your Atari with distortion values other than 10. In this section we'll present some sound effects, showing only a few of the possibilities that go beyond simple sounds. Some use pure pitches and some invoke various distortions. All are available to be modified and built upon.

Program 10-6 uses two voices and a distortion value of 6 to reproduce the "hyperspace" sound of the Atari game Star Raiders. Lines 20 to 40 represent one loop that produces a sound that decreases in pitch, then line 50 holds the lowest pitch before the loop in lines 50 to 100 increase the pitch. Try different distortion values and vary the limits on the FOR . . . NEXT loop to change the effect.

```
10 REM * HYPERSPACE SOUND
20 FOR I=0 TO 250
30 SOUND 0,I,6,10:SOUND 1,I+3,6,10
```

```
40 NEXT I
50 FOR I=1 TO 500:NEXT I
70 FOR I=250 TO 0 STEP -1
90 SOUND 0,I,6,10:SOUND 1,I+3,6,10
100 NEXT I
```

Program 10-6. Hyperspace sound demonstration.

You can get a very pleasant tinkling sound from Program 10-7. This program uses the RND function in line 20 to select a pitch value and then sounds that value in line 30 before going back in line 40 to select another random number for the pitch. Changing the limit in line 20 will change the pitch range over which the tinkling is heard.

```
10 REM * FIRST COMPUTER SOUND
20 I=INT(RND(0)*20)
30 SOUND 0,I,10,10
40 GOTO 20
```

Program 10-7. Computer sound demonstration.

A police siren is the result of Program 10-8. The pitch fluctuates up and then down, using two FOR . . . NEXT loops (see lines 20–28 and lines 30–38). Varying the limits on these loops will change the siren sound. Within each pitch control loop is a delay; changing one or both of the delays will also change the siren effect.

```
10 REM * POLICE SIREN
20 FOR I=130 TO 60 STEP -1
22 FOR DELAY=1 TO 10:NEXT DELAY
26 SOUND 0,I,10,10
28 NEXT I
30 FOR I=60 TO 130
32 FOR DELAY=1 TO 10:NEXT DELAY
36 SOUND 0,I,10,10
38 NEXT I
40 GOTO 20
```

Program 10-8. Police siren sound demonstration.

Program 10-9 produces a gunshot sound by using a distortion value of 8 and a pitch value of 30. The decay of the sound is controlled by a simple FOR . . . NEXT loop. Notice that in line 20 the STEP value is -0.2. This is an interesting method of producing delays without using delay loops. This is so because the volume value that is controlled by this step can only vary in steps of 1 or more. Therefore, it will take five cycles through the loop to produce a step change of 1. During these five loops the volume will not vary. If we use a STEP value of -0.1 the change will be half as fast; a value of -0.4 will double the speed. You can thus use fractional STEP values in place of delay loops. Simply changing the STEP size will change the delay.

```
10 REM * GUNSHOT SOUND
20 FOR I=14 TO 0 STEP -0.2
30 SOUND 0,30,8,I
```

201

```
40 NEXT I
50 GOTO 20
```

Program 10-9. Gunshot sound demonstration.

A xylophone sound results from the use of Program 10-10. In line 20 the pitch is sounded at full loudness and then reduced in loudness at a rate that depends on the value of STEP (0.5 in our example). Again, different STEP values will change the character of the xylophone-like sound. This is much like using Program 10-3 in an earlier section of this chapter.

```
10 REM * XYLOPHONE SOUND
12 FOR Z=10 TO 100
20 FOR I=15 TO 0 STEP -0.5
30 SOUND 0,Z,10,I
40 NEXT I
45 NEXT Z
50 END
```

Program 10-10. Xylophone sound demonstration.

Overlaying three voices with a distortion value of 0 and three different pitch values results in an explosion sound in Program 10-11. You can go from a short explosion to a long, drawn out one by varying the STEP value in line 20. Fractional STEP values again will function as delays and lengthen the decay of the explosion sound.

```
10 REM * EXPLOSION SOUND
20 FOR I=15 TO 0 STEP -1
30 SOUND 0,6,0,I
32 SOUND 1,21,0,I
34 SOUND 2,27,0,I
40 FOR DELAY=1 TO 50:NEXT DELAY
60 NEXT I
70 END
```

Program 10-11. Explosion sound demonstration.

Finally, in Program 10-12 we produce a heartbeat sound by using two voices with two different pitch values and two distortion values. Notice how lines 80 and 90 have the effect of switching the values of K and L each time through the FOR . . . NEXT loop. Changing the values of K and L will greatly change the characteristics of the sound; changing the STEP value in line 40 will vary the "pulse" of the heartbeat.

```
10 REM * HEARTBEAT SOUND
20 K=1:L=2
30 FOR I=1 TO 20
40 FOR J=6 TO 0 STEP -0.6
50 SOUND 0,125,0,J/K
60 SOUND 1,253,10,J/L
70 NEXT J
```

```
8Ø IF K=1 THEN K=2:L=1:GOTO 1ØØ
9Ø K=1:L=2
1ØØ NEXT I
```

Program 10-12. Heartbeat sound demonstration.

We have only scratched the surface of the kinds of sound effects you can generate with the Atari SOUND statement. With time and practice you will surely be able to make far more intriguing sounds than we have presented in this section. Using SOUND statements and FOR . . . NEXT loops with numeric variables in them makes it easy to keep changing the sound until you hit upon one that is worth keeping. Then you just save that program (with an appropriate REM statement to annotate it) and you are on your way to a sound effect library of your own.

...SUMMARY
Sounds on the Atari are invoked by using

```
SOUND A,B,C,D
```

where A is the voice (0–3), B is the pitch (0–255), C is the distortion (0–15), and D is the loudness (0–15). Small pitch values correspond to high-pitched notes. Distortion values of 10 or 14 correspond to pure tones, while other values introduce variations from musical tones. Up to four voices may sound at the same time, permitting pleasant musical chords or raucous sound effects.

Problems for Chapter 10 . . . .

1. Write a program to play all pitch values in reverse order (255 to 0), using a distortion value of 10.

2. Modify Program 10-4 to play chords by playing the lowest note first, holding it for a short time, adding the second note, holding for a short time, and adding the last two notes in the same manner.

3. Write a program that plays all pitch values from 0 to 100, playing each note at all possible distortion values before going on to the next note.

4. Write a program that has one voice varying continuously in pitch while another, with a different pitch, sounds its pitch and then stops before going on to the next pitch value.

5. Modify Program 10-5 to use keyboard input, using keys whose characters bear some relationship to the input desired.

6. There is phenomenon in physics that occurs when two notes of similar pitch are played together. It is the result of interference between the two sound waves and is heard as a pulsating sound. The rate of the pulsations decreases as the two notes approach each other in pitch. Write a program that demonstrates this effect by playing one note continuously with a pitch value of 180 and varying the pitch value of another note, played simultaneously, from 170 to 190. Include a delay to give the listener time to hear the effect.

PROGRAMMER'S CORNER 10

...Turning Sounds into Music

Music consists fundamentally of notes and combinations of notes, played for predetermined periods of time. We already know which of the Atari pitch values correspond to musical notes and we know how to select these notes and play them in combinations. To duplicate a musical selection we can use FOR . . . NEXT loops to time how long notes will sound.

Music notation uses a series of notes of fixed length called whole, half, quarter, eighth, sixteenth, and thirty-second notes. Figure 10-3 shows how they are represented. Each one in this series plays for half the time of the one before it. Thus, a quarter note is sounded for half the time of a half note and twice the time of an eighth note. If we specify the notes we want played and how long each should play, we can create or recreate a musical selection. We'll put the notes and their duration in DATA statements. That way we can easily make changes and we can also repeat the music from within the program by using the RESTORE statement to go through the DATA statements again. We'll need a routine to READ the DATA and then assign the pitch and duration to SOUND statements. We'll also want to include the possibility of stopping any or all of the notes.

We'll demonstrate all of this by programming the Atari to play the Fantasie Impromptu by Chopin. Figure 10-4 contains the music in its normal form. We have shown only one part of the music, leaving out a lower-pitched harmony for simplicity. Also shown are the pitch values that correspond to these notes. These were found from Figure 10-1. We will indicate the duration of a particular note by using 1 for whole notes, 2 for half notes, 4 for quarter notes, and 8 for eighth notes. It will then be easy to fix the duration that notes play by using

```
DUR=DELAY*1/HOLD
```

where HOLD is the value for the type of note (1, 2, 4, or 8) and the value of DELAY will determine how fast the music plays. Notice that there are several places where two notes are played at the same time. We'll take care of this by having two sections, one to play single notes and one to play pairs of notes. We'll signal which section to use by putting in a PITCH value of −1 to signal the program to switch routines. We'll use a PITCH value of −2 to signal the end of the music. Program 10-13 includes all of these features.

```
90 REM * PLAY A TUNE
95 DELAY=500
100 GOSUB 2000
150 GOTO 600
200 FOR I=1 TO DUR
210 NEXT I
250 RETURN
598 REM * START HERE
```

```
600 READ PITCH,HOLD
605 IF PITCH=-1 THEN 700
608 DUR=DELAY*1/HOLD
610 SOUND 0,PITCH,10,6
620 GOSUB 200
630 GOTO 600
700 READ PITCH1,PITCH2,HOLD
710 IF PITCH1=-1 THEN SOUND 1,0,0,0:SOUND 0,0,
0,0:GOTO 600
715 IF PITCH1=-2 THEN 800
720 DUR=DELAY*1/HOLD
730 SOUND 0,PITCH1,10,6
740 SOUND 1,PITCH2,10,6
750 GOSUB 200
760 GOTO 700
800 SOUND 0,0,0,0
810 SOUND 1,0,0,0
820 LIST 95
825 POKE 752,0
830 END
998 REM * NOTE DATA FOR FANTASIE IMPROMPTU
1000 DATA 108,1,96,4,108,4,81,4,72,4
1005 DATA -1,0
1006 DATA 64,108,1
1007 DATA -1,0,0
1008 DATA 53,1,60,2,64,2,72,2,64,4,81,4
1010 DATA 108,1,108,1,96,1,91,4,96,4,72,4,64,4
1020 DATA -1,0
1030 DATA 60,96,2,64,108,2,72,121,2,64,108,2
1040 DATA -1,0,0
1050 DATA 81,2,114,8,108,8,96,8,108,8
1060 DATA -1,0
1070 DATA 64,114,2,72,121,2,81,128,1,40,64,1
1080 DATA -2,0,0,0
1998 REM * TITLE SCREEN
2000 PRINT "⅃":POKE 752,1
2020 POSITION 10,4:PRINT "FANTASIE"
2030 POSITION 14,7:PRINT "IMPROMPTU"
2040 POSITION 18,10:PRINT "by"
2050 POSITION 21,14:PRINT "Chopin"
2060 RETURN
```

Program 10-13. Chopin's Fantasie Impromptu.

We have put the delay loop that determines a note's duration at the beginning of the program so it will not be affected by the length of the program. We can speed up or slow down the rate that the music plays (called the tempo) by just changing the value of DELAY in line 95. Note lines 1005, 1007, 1020, 1040, and 1060, where the −1 signals a change from one voice to two and vice versa. This is just a beginning. We

could go on by adding more voices and get into shaping the tones and varying the loudness for dramatic effect.

In the music for our demonstration, whenever two notes play they both have the same duration. This is not always the case in music. One way to handle the more general case is to break the music into segments as long as the shortest note in the entire selection. This allows for handling the simultaneous playing of both short and long notes, by continuing the pitch of long notes for the required number of segments while varying the pitch of short notes.

The program has made it easy to change the speed at which the music is played. Pay particular attention to line 820. When the music is finished, the program turns off all the voices and then line 820 causes line 95 to be listed on the screen. It is then easy to use the CTRL and arrow keys to move the cursor to where this line is listed, change the value of DELAY, and press RETURN. Typing RUN will then cause the music to be played at the new speed. Many "what if" situations can be investigated by using this technique. We will make use of this in some of the graphics programs in the next chapter.
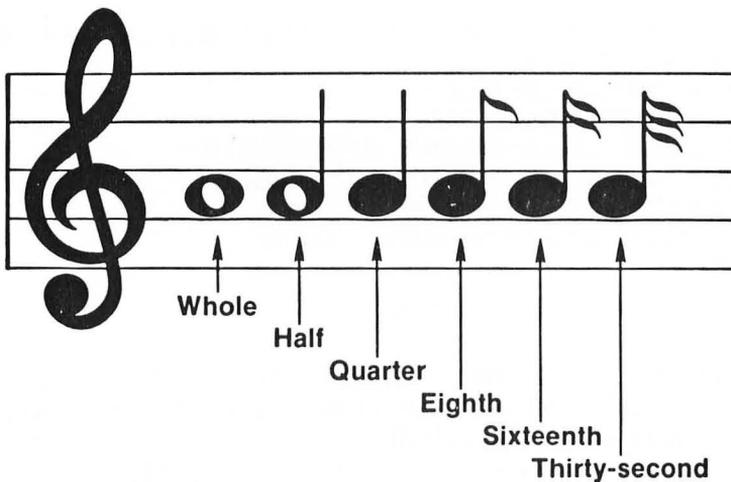
Figure 10-3. Representation of music notation.

FANTASIE IMPROMPTU

F. Chopin
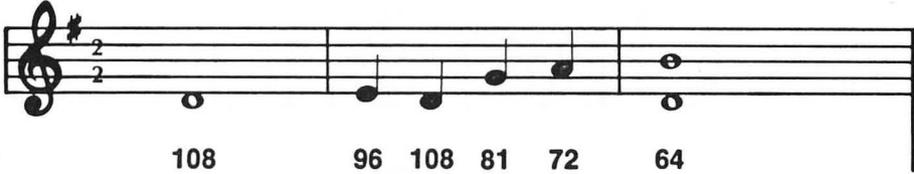
Pitch		108		96	108	81	72	64	
Values								108	
Duration		1		4	4	4	4	2	

Pitch	53		60	64		72	64	81	108
Values									
Duration	1		2	2		2	4	4	1

Pitch	108		96		91	96	72	64	60	64
Values									96	108
Duration	1		1		4	4	4	4	2	2

Pitch	72	64	81	114	108	96	108	64	72	81
Values	121	108						114	121	128
Duration	2	2	2	8	8	8	8	2	2	1
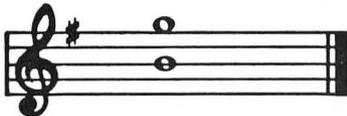
Pitch	40
Values	64
Duration	1

Figure 10-4. Music notation for Chopin's Fantasie Impromptu.

Chapter 11

Graphics
Graphics
Graphics

Atari computers have very powerful graphics capabilities. You have surely seen the dazzling games and colorful graphics demos in computer stores and magazines, and you probably have some of them in your own collection of programs. We will not be able to get to a level where we can recreate an arcade game in this book, but we can start you in that direction by showing what graphics capabilities you can access from Atari BASIC and how to use them. You'll be able to find many places in your own programs where graphics will add a welcome visual impact. Above all, don't let graphics intimidate you—they are certainly no harder to use than many of the capabilities we have already covered.

We'll begin with a discussion of text modes and then cover the commands common to the graphics modes. Then, after looking at these regular graphics modes in more detail, we'll look at the special graphics modes. Finally, we will present some advanced techniques that you can also master with practice. The key word is *practice*. It's just as true with graphics as it was with counting—comfort with concepts and ease of use are directly proportional to how much you practice, how much time you take for hands-on experience. A lot of the confusion concerning Atari graphics stems from the fact that the same commands produce somewhat differing results in different graphic modes. To keep things in better order, we will group similar families of modes together to make your job easier.

11-1...Text Modes

...GRAPHICS 0

We have used graphics mode 0 (GRAPHICS 0) extensively. It is the primary mode for displaying letters and words on the screen. In Chapter 5 we also learned how to redefine the characters to make them take on any shape we chose. To summarize GRAPHICS 0, there is one color available. This is chosen with

```
SETCOLOR 2,B,C
```

where, as usual, B is the color (0–15) and C is the brightness (0–14, even numbers only). The color of the border around the 40-by-24 text screen is chosen with

```
SETCOLOR 4,B,C
```

and the brightness of the text displayed can be varied with

```
SETCOLOR 1,B,C
```

where B has no effect. In GRAPHICS 0, the COLOR statement will select a character based on the internal character code (as discussed in Programmer's Corner 5) and "draw" with it. Look at the following program:

```
90 REM * "DRAWING" IN GR. 0
100 GRAPHICS 0
110 X=ASC("$")
120 COLOR X
130 PLOT 2,2
140 DRAWTO 37,18
150 END
```

Program 11-1. Drawing in GRAPHICS 0.

Running this program will result in

```
$$
  $$
    $$
      $$
        $$
          $$$
            $$
              $$
                $$
                  $$
                    $$$
                      $$
                        $$
                          $$
                            $
```

READY

Figure 11-1. Execution of Program 11-1.

...GRAPHICS 1 and GRAPHICS 2

These text modes produce text that is enlarged and colorful. This can make for very effective displays in educational programs where you want to display a list of the choices that a user has at that point in the program. GRAPHICS 1 produces text that is twice the width but the same height as GRAPHICS 0; GRAPHICS 2 gives text both twice the width and twice the height. Because of this, you are limited to fewer characters per line and per screen. Here is a comparison of the text modes.

GRAPHICS	SCREEN SIZE	
MODE	WITH TEXT WINDOW	WITHOUT TEXT WINDOW
0	...	40 by 24
1	20 by 20	20 by 24
2	20 by 10	20 by 12

With both of these large character modes you have a limited selection of characters (actually one-half the regular character set, and the inverse video characters are also not available). Normally you have at your disposal uppercase and punctuation characters. To access the other half of the character set (lowercase and graphics characters), use

```
POKE 756,226
```

If you use POKE to change this location to 224 you restore the regular set. When you attempt to use lowercase and graphics characters you will see that there is a problem. There is no space character—it is in the original set! The absence of a space character will make for very cluttered displays, since its place in this part of the character set is taken by the heart-shaped graphics character. We can fix this problem by using our ability to redefine characters to replace the heart with a blank space. This is left as a exercise.

In both GRAPHICS 1 and 2 we can have text displayed in four colors on a background that has a fifth color. This makes for very colorful text screens. Here's a short program that demonstrates how to do this.

```
90 REM * DEMONSTRATE GR.2 COLORED LETTERS
100 DIM A$(4)
110 A$="Ni C ":REM * C AND e are inverse video
120 XPOS=(20-LEN(A$))/2
130 YPOS=3
140 GRAPHICS 2
155 POKE 84,YPOS
160 POKE 85,XPOS
170 PRINT #6;A$
180 END
```

Program 11-2. Drawing in GRAPHICS 2.

The first thing to notice is that we needed a GRAPHICS 2 statement to get into the large text mode. You need a GRAPHICS N statement to get into any mode that you are not currently in. The second thing to notice is that we had to use PRINT #6;

rather than just PRINT to get the printing to go to the GRAPHICS 2 screen (the same holds true for GRAPHICS 1). A normal PRINT message will send the characters to the text window, if one exists at the bottom of the screen, or else force the display back to GRAPHICS 0 and then PRINT, leaving the display in GRAPHICS 0. Now notice line 110. The string A$ has just four letters, each in a different form. If this program is RUN, you will see the word "NICE" in the center of the screen, with each letter a capital and each one a different color. These modes can only display uppercase, standard letters (unless, as mentioned above, the alternate character set is chosen); the effect of using other forms is to select different color registers for the letters. See Table 11-1 for information on which form of the letter (uppercase, lowercase, regular video, or inverse video) indicates which color register. Finally, notice line 120. This provides a handy way to center text on a line to make a nice display (in any text mode). We take the total line length permitted in the text mode being used (20 in this case), subtract the length of the string to be printed (LEN(A$)) and divide by 2. The result is the value for the number of characters we need to move from the left edge of the screen before printing (XPOS).

...SUMMARY

GRAPHICS 0, 1, and 2 are the text modes. Program writing and jobs such as letter writing are done in GRAPHICS 0. This mode provides a 40-column screen with 24 lines per display. For bolder displays, as would be effective in title frames and educational programs, GRAPHICS 1 or GRAPHICS 2 are very useful. GRAPHICS 1 has characters twice as wide as GRAPHICS 0, but the same height. GRAPHICS 2 has characters both twice as wide and twice as tall. This means, of course, that fewer characters can be displayed per line and per screen.

Problems for Section 11-1 . . . .

1. Modify Program 11-2 to display a message in varying colors, using GRAPHICS 1.

2. Write a program to display the following message in GRAPHICS 1 in uppercase letters: "KEEP YOUR EYE ON THIS MESSAGE". Break the message into two lines after the word "EYE" and center both lines on the screen. Then, after a pause, have the program switch to the same message in lowercase characters. You should be able to make this change with one program line. Do not have a text window at the bottom of the screen. (Note: you will have hearts surrounding your lowercase message.)

3. Using GRAPHICS 2, write a program to print your first and last names on two different lines. Vary the background color through all its possible colors.

4. Write a program that uses the routine to redefine characters (from Programmer's Corner 5) to substitute a blank space for the heart character in the alternate character set for GRAPHICS 1 and 2. You will need to use POKE 756, TOP+2 to use the redefined set, where TOP= PEEK(106)−4.

11-2...The Graphic GRAPHICS Modes

We'll first discuss the instructions that are common to graphics modes 3 to 8 and then discuss the various groups of these modes in more detail.

...SETCOLOR

The SETCOLOR statement is used to put a specific color value in a specific color register. This is very much akin to filling a paint can before actually painting. The statement has the form

```
SETCOLOR A,B,C
```

where A is the color register (0–4), B is the color value (0–15), and C is the brightness (0–14, even numbers). Not all of these graphics modes have all color registers available, and not all registers are treated the same in the different graphics modes. This is the source of endless confusion. To help out this situation, we present Tables 11-1 and 11-2, which show all the relationships and also some other information that we will get to soon. These two tables present the same information in two different forms. You can choose the one that makes the relationships clearer to you. The correspondence between colors and their color values is given in Figure 3-1.

...Color

To actually draw on the screen, as with painting on paper, you first must put the paintbrush in the desired paint can. You do this with the statement

```
COLOR N
```

where N selects the color register. Unfortunately, there is no direct correspondence between N in the COLOR statement and A in the SETCOLOR statement. That is, COLOR 1 does not choose color register 1, and so on. The actual relationship is also shown in Tables 11-1 and 11-2, which you will want to refer to as the need arises in your graphics work. The computer manages an orderly translation between these sets of numbers, but that doesn't change the fact that you will have to work with the existing, somewhat cumbersome system. The best advice is to just use the table and concentrate your efforts on creating dazzling displays.

...PLOT and DRAWTO

The analogy to painting goes even further. Once the paintbrush is dipped in paint it must be placed on the paper at the point where you want to start to work and then you can proceed to paint. The statements to have the Atari do this on your TV screen are

```
PLOT X,Y
```

and

```
DRAWTO X,Y
```

GRAPHICS MODE	SETCOLOR VALUE	POKE LOCATION	EFFECT IS ON:
0	0	—	—
	1	709	Brightness of characters
	2	710	Screen color
	3	—	—
	4	712	Border
1, 2	0	708	PRINT #6; regular, uppercase text
	1	709	PRINT #6; regular, lowercase text; Brightness of text in text window
	2	710	PRINT #6; inverse, uppercase text; text window color
	3	711	PRINT #6; inverse, lowercase text
	4	712	Background
3, 5, 7	0	708	COLOR 1
	1	709	COLOR 2 and brightness of text in text window
	2	710	COLOR 3 and color of text window
	3	—	—
	4	712	Background
4, 6	0	708	COLOR 1
	1	—	Brightness of color in text window
	2	—	Color of text window
	3	—	—
	4	712	Background
8	0	—	—
	1	709	COLOR 1 brightness
	2	710	COLOR 1 (background)
	3	—	—
	4	712	Border

Table 11-1. Color relationships arranged by graphics mode.

In both of these statements, X is the horizontal distance across the screen, measured from the left side of the screen display, and Y is the vertical distance, measured from the top. In all graphics modes, the upper left corner is the point (0,0). The number of points across or down the screen varies with the graphics mode and determines the degree of detail that can be displayed. In the above, the point that you "draw to" would, of course, be different than the point where you start. To

SETCOLOR NUMBER	POKE NUMBER	GRAPHICS MODE	EFFECT IS ON:
4	712	0	Border
		1, 2	Background
		3, 5, 7	Background
		4, 6	Background
		8	Border
3	711	0	—
		1, 2	PRINT #6; inverse, lower-case text
		3, 5, 7	—
		4, 6	—
		8	—
2	710	0	Screen
		1, 2	Text window and PRINT #6; inverse, uppercase text
		3, 5, 7	Text window and COLOR 3
		4, 6	Text window
		8	Background color
1	709	0	Brightness of characters
		1, 2	Brightness of text in text window; PRINT #6; regular, lowercase text
		3, 5, 7	Brightness of text in text window; COLOR 2
		4, 6	Brightness of colors in text window
		8	COLOR 1 brightness (color is same as background)
0	708	0	—
		1, 2	PRINT #6; regular, upper-case text
		3, 5, 7	COLOR 1
		4, 6	COLOR 1
		8	—

Table 11-2. Color relationships arranged by SETCOLOR number.

simply plot individual points, you use a series of PLOT statements, with each specifying the location to be plotted. Following a DRAWTO statement with another PLOT statement has the effect of picking up the paintbrush and putting it down at a different place on the screen.

...**LOCATE**

The LOCATE statement can be used to examine the COLOR value (0–3) for any point on the screen. It has the form

```
LOCATE X,Y,N
```

where X and Y are the coordinates of the point of interest and N is the value from COLOR N for that point. This can be useful in games for determining when an object of a particular color passes a particular point on the screen.

11-3...**GRAPHICS 3, 4, 5, 6, 7, and 8**

...**GRAPHICS 3, 5, and 7**

In Chapter 3 we learned how to use modes 3, 5, and 7. These three modes form a family that all use the same statements to select colors to use and to draw on the TV screen. They differ only in the degree of detail (resolution) that they are capable of displaying. We have previously discussed the graphics statements SETCOLOR, COLOR, PLOT, DRAWTO, and LOCATE. Table 11-3 shows the screen display size and the memory requirements for these three modes.

GRAPHICS MODE	WITH TEXT WINDOW SCREEN SIZE	MEMORY REQUIRED	WITHOUT TEXT WINDOW SCREEN SIZE	MEMORY REQUIRED
3	40 by 20	434	40 by 24	432
5	80 by 40	1174	80 by 48	1176
7	160 by 80	4190	160 by 96	4200

Table 11-3. Screen size and memory requirements for graphics modes 3, 5, and 7.

In using Table 11-3, keep the following points in mind:

1. The upper left corner of the screen is considered by the computer to be the (0,0) location.

2. Since the computer starts counting from 0 in both X and Y, the maximum values that you are permitted to use for both X and Y are one less than the screen size given above.

3. Always update the COLOR statement before a PLOT or DRAWTO statement, or else you will be using the last color specified.

4. Use COLOR 0 to erase, since you will then be drawing with the background color. If you fail to pick a color at all, you'll be using COLOR 0 and your drawing will be invisible!

...**GRAPHICS 4 and 6**

GRAPHICS 4 has all the properties of GRAPHICS 5, with the exception that only one foreground color is permitted rather than four. Similarly, GRAPHICS 6 is like GRAPHICS 7, with the same exception. So why do these less colorful modes have

any value at all? Their value is that they require only about half as much memory as the corresponding four-color mode. Table 11-4 shows a comparison.

| | MEMORY NEEDED | |
GRAPHICS MODE	WITH TEXT WINDOW	WITHOUT TEXT WINDOW
4	694	696
5	1174	1176
6	2171	2184
7	4190	4200

Table 11-4. Memory requirements for graphics modes 4 and 6 compared with modes 5 and 7.

The message: if you've got the memory to spare, use the colorful modes. If not, either obtain more memory or else use modes 4 and 6.

...GRAPHICS 8

For the highest resolution, this is your mode. GRAPHICS 8 permits 320-by-196 displays and the capability of very detailed drawing. You are, however, limited to a one-color background and one brightness of the same color for the foreground. There are ways to increase the number of colors that we will be discussing in a later section. Graph paper to plan your GRAPHICS 8 drawing is a virtual necessity. Trial and error will only work with the simplest of GRAPHICS 8 displays. Beyond that, careful planning will pay big dividends.

...SUMMARY

GRAPHICS 3 to 8 are the regular graphics modes. Increased resolution is the reward of using the higher-numbered modes, while the need for more memory is the penalty. The same set of graphics commands applies to all of these display modes. In each graphics mode it is useful to picture the screen with a grid on its face. All positions on the screen are referenced to the upper left-hand corner. The number of plotting positions on the screen increases from 40 by 24 in GRAPHICS 3 to 320 by 196 in GRAPHICS 8. For colorful displays, GRAPHICS 3, 5, or 7 are preferred because they allow four different colors on the screen at the same time.

Problems for Section 11-3 . . . .

1. Write a program to make the screen a medium blue, then a dark orange, and then a bright green. Pause between color changes.

2. Draw a robot-shaped figure in GRAPHICS 3. Make its eyes blink and change color by changing the color of the eyes only.

3. Draw your first name on a GRAPHICS 8 screen in cursive. (Use graph paper!!)

4. Write a program using SETCOLOR to draw a 20-by-10 green block on a purple background in GRAPHICS 3. Then use the RND function to

test ten random points on the screen to see if they are inside or outside the green square. Plot a blue point at each test location after testing for its position. Display a message in the text window indicating if the point is inside or outside the green square.

11-4...Graphs from Formulas in GRAPHICS 8

Figures that can be described by using a formula are easy to graph. There are many examples from mathematics.

...Cartesian Coordinates

Let's develop a method for adjusting the X and Y values in the conventional Cartesian coordinate system for plotting on the Atari screen. We would like to move the (0,0) point nearer the center of the screen and alter the orientation for Y values so that they are increasing up instead of down. Suppose we specify that the point (160,80) on the screen shall represent the point (0,0) in a Cartesian system. The X conversion is easy. We simply want to move each plotted point to the right on the screen. The Y conversion requires that we turn the graph "upside down." So the point

$(X1,Y1)$

in the conventional Cartesian coordinate system becomes

$(160 + X1, 80 - Y1)$

on the GRAPHICS 8 screen.

It would be nice to plot the X and Y axes right on the screen. A very simple subroutine will do this for us. We can plot the vertical line two dots wide.

Plotting points that fit a formula is straightforward enough. For our first graphs we might do just functions. We need a subroutine that scans all possible values for X and determines if the Y value is on the screen. If it is, then the routine should do the plotting. If not, then the routine should simply try the next X value. All of this is done in Program 11-3.

```
90 REM * PLOT A FUNCTION
100 GRAPHICS 8
105 YS=40:XS=15
110 PI=3.141592
118 REM * DRAW BORDER
120 COLOR 1:GOSUB 600
128 REM * PLOT AXES
130 GOSUB 700
148 REM * DRAW THE GRAPH
150 GOSUB 200
160 LIST 220
170 END
198 REM * PLOT A FUNCTION
```

```
200 FOR X1=-150 TO 150
220 Y1=X1
230 X=160+X1
240 Y=80-Y1
250 IF Y<3 OR Y>156 THEN 270
260 PLOT X,Y:PLOT X+1,Y
270 NEXT X1
280 RETURN
600 PLOT 0,0:DRAWTO 319,0
610 DRAWTO 319,159:DRAWTO 0,159:DRAWTO 0,0
620 PLOT 1,0:DRAWTO 1,159
630 PLOT 318,0:DRAWTO 318,159
640 RETURN
698 REM * PLOT AXES FOR GRAPHING
700 PLOT 3,80:DRAWTO 316,80
710 PLOT 160,3:DRAWTO 160,156
720 PLOT 161,3:DRAWTO 161,156
730 RETURN
```

Program 11-3. Plot a function in GRAPHICS 8.

The program is structured to first draw the border (using the subroutine at line 600), then the axes (using the subroutine at line 700), and finally plot the function (using the subroutine at line 200). The program is intentionally done in GRAPHICS 8 rather than GRAPHICS 8+16, in order to keep the text window at the bottom of the screen. We use this window to print out the program line that contains the function plotted (line 160). This not only gives us a record of what we plotted, but also permits us to move the cursor to this line in the text window, change the function, and rerun the program without having to return to GRAPHICS 0, LIST the program, make the change, and then RUN the modified program. In addition, you might want to move the axes so that the point (0,0) is not in the exact center. This could be done by passing (X0,Y0) to the axes-plotting subroutine as the Atari coordinates of the (0,0) point for the Cartesian graph.

...Polar Graphs

Polar equations often produce interesting graphs. One of the reasons we don't draw many polar graphs by hand is that they take too much tedious calculation involving trigonometric functions. We can easily produce the graphs without the tedium by using GRAPHICS 8 and programming Atari BASIC to do the calculations.

We may use

$$R = 1 - 2 \cos (G)$$

as a sample equation. Using sines and cosines, we get the X and Y coordinates as follows:

$$X = R \cos (G)$$

and

$$Y = R \sin (G)$$

where G is the central angle in radians. To obtain a full graph the central angle must sweep through a full 360 degrees or 2π. That is about 6.29. We can get about 60 points by using STEP .1 in a FOR . . . NEXT loop. Since the point (0,0) is in the corner of the Atari screen we need to adjust the starting point to keep the figure in view.

To make our figures as large as possible we can use Graphics 8 + 16 to obtain full-screen graphics. After the plot is finished we will include a delay loop in our program so that the screen stays in this full-screen mode to allow us to admire the display. Now we have to think about adjusting the X and Y values on the conventional Cartesian coordinate system for plotting on the Atari screen. The X conversion is easy. We simply want to move each plotted point to the right on the screen. The Y conversion requires that we turn the graph "upside down." So the point

(X9,Y9)

in the conventional Cartesian coordinate system becomes

(X + X9, Y − Y9)

on the TV screen. Where the point (X,Y) defines the point on the Atari screen is where we want the Cartesian point (0,0) to be located.

It would be nice to display a polar axis right on the screen with the graph. We can easily plot a line beginning at the point (0,0) and extending to the right edge of the Atari screen. Placing the polar axis on the screen will clearly locate the graph for us.

Once we have a working program, it will be a simple matter to plug in other equations. In this way we can look at dozens of graphs in the time it would take to draw a single graph by hand. It is interesting to watch the figures as they are formed on the screen. Drawing a polar graph by hand (like typing a 100-page paper on a portable typewriter) is one of those things everybody ought to do once in his or her lifetime.

Our program divides nicely into three packages: the control routine, the polar axis plotting routine, and the graph plotting routine. Let's work on them in that order.

In the control routine we set up the graphics screen with GRAPHICS 8. Setting the color is easy. Next we define the X and Y axes and call the polar axis plotting subroutine. Polar graphs plotted true size are usually very small. So we should provide a scaling factor to produce a larger graph. We define the Radial Scale in RS. In the actual plotting subroutine we will be arranging for the central angle to range through a full rotation of 2π. But we might like to control the STep size in the control routine. Thus we set the value of ST here. Finally we call the plotting subroutine. That is all there is to it. See Program 11-4a.

```
100 GRAPHICS 8
110 COLOR 1
120 X=139:Y=80
130 GOSUB 1000:REM * PLOT POLAR AXIS
```

```
140 RS=25:ST=0.1
150 GOSUB 200:REM * PLOT THE GRAPH
160 LIST 210
190 END
```

Program 11-4a. Control routine for polar graphing.

In Program 11-4a line 120 sets the axes as close to the center of the screen as possible. Line 140 sets the radial scale at 25 and the step size at .1.

The easy one is the polar axis plotting routine. All we do is use PLOT statement to begin at (X,Y) and then DRAWTO to reach the right edge of the screen. This takes one statement. See Program 11-4b.

```
996 REM * PLOT POLAR AXIS
1000 PLOT X,Y:DRAWTO 279,Y
1030 RETURN
```

Program 11-4b. Draw a polar axis.

Now let's look at the actual plotting subroutine. We need to provide for the angle to sweep a full rotation. This is done with a FOR . . . NEXT loop ranging from 0 to 6.29. The number of points we want plotted may well depend on the size of the graph. We may want more points for larger graphs. So we let the calling routine establish the step size. A large step size will not give enough points on the graph, while too small a step size will take too long to plot. We can then experiment with each new equation until we get a nice graph. Again we plot two points next to each other to avoid spurious colors from artifacting (discussed in the next section).

```
198 REM * GRAPH POLAR GRAPH
200 FOR G=0 TO 6.29 STEP ST
210 R1=1-2*COS(G)
220 R9=RS*R1
230 X9=R9*COS(G):Y9=R9*SIN(G)
240 PLOT X+X9,Y-Y9
242 PLOT X+X9+1,Y-Y9
250 NEXT G
290 RETURN
```

Program 11-4c. Polar-graph-plotting subroutine.

In Program 11-4c, the polar equation is defined in line 210, the scaling factor is implemented in line 220, and the Cartesian X and Y values are calculated in line 230. It will be a simple matter to change the polar equation by changing line 210. We must be aware that other polar equations may contain points that are off the screen. We can test for out-of-range values and skip the plotting for those points. Further, we must be alert for equations that may cause BASIC to attempt to divide by 0. See Figure 11-2 for a trial run of this program.
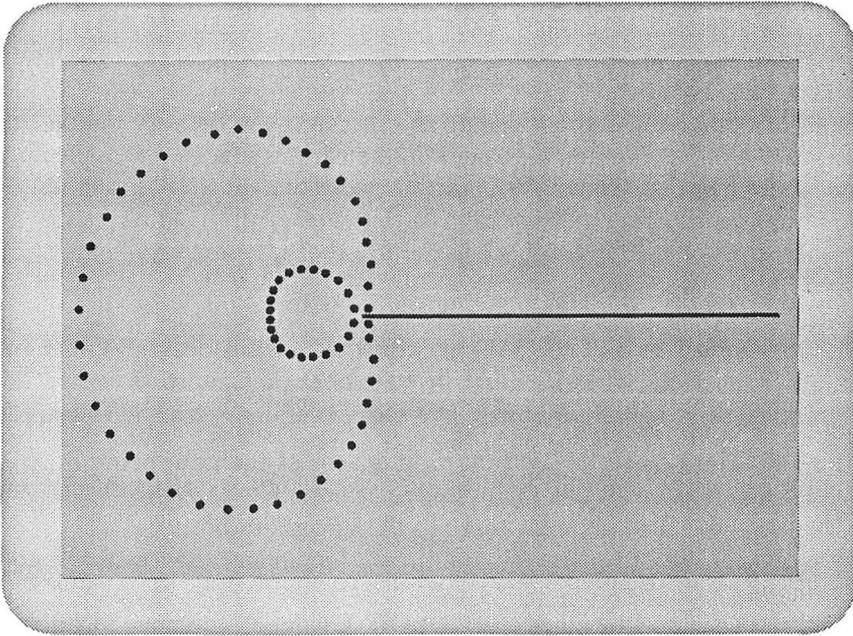
Figure 11-2. Execution of Program 11-4 a, b, c.

Problems for Section 11-4 . . . .

1. We can easily plot a circle with our polar equation plotting program using the polar equation $R = 1$. Do this.

2. There are lots of interesting polar graphs. Graph any of the following:

    a. $R = 1 + 2 \cos (G) - 3 \sin (G)^2$
    b. $R = 2 + \sin (3G)$
    c. $R = 2 + \sin (2G)$
    d. $R = \sin (G) + \cos (G)$

3. Many polar equations produce nice graphs, but they will cause our polar plotting program to fail. Some points will lie off the graphics screen. Some values of G will cause division by 0. We can easily test whether a point is on the screen between lines 230 and 240 of Program 11-4c. If a point is off the screen, don't plot it. If the formula we enter at line 210 has an indicated division then we can put in a test between lines 200 and 210. If the current value of G would cause such a 0 division, don't even execute line 210. Adding these features will enable you to draw graphs for any of the following:

    a. $R \cos (G) = 1$ d. $R = 2G$ (make the scale 1 and make G
    b. $R = 1 + R \cos (G)$ range from -50 to 50)
    c. $R = \tan (G)$ e. $R = 2/G$ (scale 25 and G from -10 to 10)

11-5...Miscellaneous Graphics Techniques

...Filling in Space

When you do complicated drawings on the screen it can be very tedious to use PLOT and DRAWTO statements to color in the various shapes that make up your drawing. To help in this, Atari BASIC includes a statement to fill in shapes. It has some limitations and certain rules need to be followed carefully, but it can save a lot of time and effort. To do a graphic fill you use the statement

```
XIO 18,#N,0,0,"S:"
```

The N ought to be 1 to 5. The "18" specifies that a fill is desired. Other values perform other functions with this versatile statement (in Chapter 9 we saw how it can be used for disk drive functions). The sequence of steps to be followed is:

1. Use PLOT to mark the bottom right corner of the item being drawn.
2. Use DRAWTO to reach the upper right corner.
3. Use DRAWTO to go from there to the upper left corner.
4. Use POSITION to place the paintbrush (cursor) at the lower left corner, without plotting.
5. Use POKE 765,X to place the COLOR X value in memory location 765.
6. Use XIO 18,#N,0,0,"S:".

This statement only works if the area inside the item drawn in steps 1 to 4 above is all the color of the background. If there are other colors in this area, the fill will stop when it gets to the first point not in the background color. This means that filling in a complicated shape can be difficult with this statement. To illustrate the use of the XIO statement, we'll write a program that draws the word "FILL!" in large letters using GRAPHICS 7 and then fills it in. We'll definitely want to use graph paper to plan it out. Figure 11-3 shows our goal. Looking at the drawing and then at the fill rules, it seems that we can do the "F" and the two "L's" in a straightforward manner, since they can be filled in one step. However, the "I" and the "!" are not quite as simple. Because of the shape of these characters, we'll have to fill the "I" and the "!" in three steps. For the fun of it, we'll use all the color registers as we draw to give the characters different colors. When we're done we'll rotate the colors to give a changing display. See Program 11-5.

```
90 REM * FILL THE WORD "FILL!"
95 GRAPHICS 7+16
98 REM * DO THE F
99 N=1:COLOR N
100 PLOT 20,70
110 DRAWTO 20,40:DRAWTO 25,40
120 DRAWTO 25,32:DRAWTO 20,32
130 DRAWTO 20,18:DRAWTO 30,18
140 DRAWTO 30,10:DRAWTO 10,10
150 POSITION 10,70
160 GOSUB 500
198 REM * DO THE I
```

Figure 11-3. Drawing of the word FILL using the XIO statement in GRAPHICS 7.

```
199 N=N+1:IF N=4 THEN N=1
200 COLOR N
210 PLOT 60,70
220 DRAWTO 60,62:DRAWTO 52,62
230 DRAWTO 52,18:DRAWTO 60,18
240 DRAWTO 60,10:DRAWTO 40,10
250 POSITION 40,18
260 GOSUB 500
265 DRAWTO 48,18
270 POSITION 48,62
275 GOSUB 500
280 DRAWTO 40,62
285 POSITION 40,70
290 GOSUB 500
298 REM * DO THE TWO Ls
305 FOR I=90 TO 120 STEP 30
307 N=N+1:IF N=4 THEN N=1
308 COLOR N
310 PLOT I,70
315 DRAWTO I,62:DRAWTO I-12,62
320 DRAWTO I-12,10:DRAWTO I-20,10
330 POSITION I-20,70
340 GOSUB 500
380 NEXT I
398 REM * DO THE !
399 N=N+1:IF N=4 THEN N=1
400 COLOR N
410 PLOT 145,70
420 DRAWTO 145,62:DRAWTO 135,62
430 POSITION 135,70
440 GOSUB 500
450 N=N+1:IF N=4 THEN N=1
455 COLOR N
470 PLOT 141,60
475 DRAWTO 150,34:DRAWTO 141,10
478 DRAWTO 139,10
480 POSITION 130,34
485 GOSUB 500
488 PLOT 130,34
490 POSITION 139,60
495 GOSUB 500
498 GOSUB 1000
499 END
500 REM * FILL ROUTINE
510 POKE 765,N
520 XIO 18,#1,0,0,"S:"
530 RETURN
1000 REM * ROTATE THE COLORS
1002 FOR J=1 TO 6
1010 X=0
```

```
1020 POKE 708,X:POKE 709,X+8:POKE 710,X+16
1025 FOR DELAY=1 TO 10:NEXT DELAY
1030 X=X+8:IF X+16<256 THEN GOTO 1020
1040 NEXT J
1045 FOR DELAY=1 TO 1000:NEXT DELAY
1050 RETURN
```
Program 11-5. Demonstrating the XIO statement in GRAPHICS 7.

Notice that since both "L's" are the same, we used a loop to draw them (lines 305–380), changing only their position on the screen. If you experiment with trying to fill a variety of shapes you will develop a feel for how to fill any shape.

...POKE vs SETCOLOR

When working with graphics, it is frequently important to have displays and colors changing rapidly. Here we are restricting our discussion to BASIC, which has definite speed limitations. Understanding the SETCOLOR concept is useful, but there will be times when using the POKE equivalent will be more convenient. The memory locations that store the five colors normally available (our color registers) are 708 to 712. Notice that Tables 11-1 and 11-2 contain both the SETCOLOR values for filling the color registers and their POKE equivalents. The value that you POKE into the appropriate memory location to replace a SETCOLOR statement is calculated as follows: if you want to use

```
SETCOLOR A,B,C
```

where A is the color register, B is the color (0–15), and C is the brightness (0–14, even numbers), then you use

```
POKE N,B*16+C
```

where N is the memory location for that SETCOLOR in the graphics mode you are using. As Tables 11-1 and 11-2 indicate, not all graphics modes have all of these colors, but the ones they do have are controlled by the registers shown. With color values from 0 to 15 and brightness values as even numbers from 0 to 14 there are 16*8 or 128 colors available. We can easily run through all of them by using POKE to put the values in the appropriate register. We know that SETCOLOR 4,B,C controls the background color in GRAPHICS 3, so we'll use that for our sample program.

```
90 REM * POKE FOR COLOR CHANGES
100 GRAPHICS 3
105 COLOR 0
110 FOR I=0 TO 128 STEP 2
120 POKE 712,I
130 NEXT I
131 PRINT "PRESS <START> TO REPEAT"
132 IF PEEK(53279)<>6 THEN 132
135 GOTO 100
```
Program 11-6. Changing colors using the POKE statement.

Notice that in line 110 we used STEP 2, since only even-numbered values have any meaning. We have used PEEK in line 132 to check whether the START key has been pressed; if it has we rerun the program. This was added because the 128 color changes occur so fast that it takes several looks to believe they occurred at all! Add the following line to slow things down so you can see the changes:

```
125 FOR DELAY=1 TO 250:NEXT DELAY
```

Using POKE is more than twice as fast as using SETCOLOR. The technique is available to add to your bag of tricks and use when appropriate.

...Color Artifacting

GRAPHICS 8 has high resolution but little color. We can get extra colors out of this mode by taking advantage of the limited resolution of the television screen. A color TV produces color by directing the electron beam that writes the screen to illuminate combinations of the three colored phosphors that make up each point on the screen. With our ability to have 320 points across the screen we can make use of this technique also. We will get different colors on a GRAPHICS 8 screen, depending on whether we illuminate even, odd, or both even and odd dots across the screen. This is most easily seen if we draw vertical lines on the screen. Let's look at the possibilities with Program 11-7.

```
90 REM * SIMPLE ARTIFACTING
95 GRAPHICS 8:COLOR 1
96 PRINT "ODD    EVEN    ODD AND EVEN"
100 FOR I=1 TO 50 STEP 2
110 PLOT I,1:DRAWTO I,150
120 NEXT I
130 FOR I=60 TO 110 STEP 2
140 PLOT I,1:DRAWTO I,150
150 NEXT I
160 FOR I=120 TO 170
170 PLOT I,1:DRAWTO I,150
180 NEXT I
```

Program 11-7. Color artifacting in GRAPHICS 8.

All we have done is to first write only in odd positions, then even positions, and then even and odd positions. The result is three different colors on the screen.

This can also be seen if we draw diagonal lines down the screen, where the bottom of the line is only slightly displaced from the top. The limited resolution of the display means that the line will "jog" as it goes down the screen. In the process, it will alternate colors as it moves from odd to even columns. How often the color change occurs will depend on the amount that the bottom of the line is displaced from the top. See Program 11-8.

```
90 REM * ARTIFACTING WITH DIAGONAL LINES
100 GRAPHICS 8:COLOR 1
110 FOR I=0 TO 50 STEP 2
```

```
120 PLOT I,1
130 DRAWTO I+1,150
140 NEXT I
150 FOR I=60 TO 110 STEP 2
160 PLOT I,1
170 DRAWTO I+2,150
180 NEXT I
190 FOR I=120 TO 170 STEP 2
200 PLOT I,1
210 DRAWTO I+3,150
220 NEXT I
230 FOR I=180 TO 230 STEP 2
240 PLOT I,1
250 DRAWTO I+4,150
260 NEXT I
270 FOR I=240 TO 290 STEP 2
280 PLOT I,1
290 DRAWTO I+5,150
300 NEXT I
```

Program 11-8. Color artifacting with diagonal lines.

Let's look at one more manifestation of this effect, called a *moire* pattern. By varying the angle of our diagonals it is possible to produce some striking and colorful patterns on a GRAPHICS 8 screen. Here is just one example of the effects that can be generated.

```
90 REM * GRAPHICS 8 MOIRE PATTERN
100 GRAPHICS 8+16
110 FOR I=0 TO 319 STEP 4
120 PLOT 0,0
130 DRAWTO I,159
140 NEXT I
150 FOR I=319 TO 0 STEP -4
160 PLOT 319,0
170 DRAWTO I,159
180 NEXT I
190 FOR I=0 TO 159 STEP 4
200 PLOT 159,0
210 DRAWTO I,I/2
220 NEXT I
230 FOR I=319 TO 159 STEP -4
240 PLOT 159,0
250 DRAWTO I,(319-I)/2
260 NEXT I
270 GOTO 270
```

Program 11-9. A moire pattern.

You can produce many dazzling high-resolution displays by starting the drawing from different places on the screen and drawing over existing patterns with new ones.

...SUMMARY

This section has presented three useful techniques for expanding the capabilities of your Atari. The ability to fill in shapes can greatly simplify the programming of complicated graphics displays. It is essential, however, to follow a prescribed set of steps in order to get the desired effect. Using POKE rather than SETCOLOR can greatly speed up color changes. This can be especially important in a long program, since BASIC can become slow in some operations if they are located near the end of a program. Color artifacting in GRAPHICS 8 can bring color to high-resolution displays that otherwise would not have it. Understanding how artifacting works also helps to prevent it when it is not desired. Some very hypnotic displays can be achieved by using artifacting in simple programs.

Problems for Section 11-5........................

1. Redo Problem 4 in Section 11-3 to use POKE rather than SETCOLOR to set the color and brightness.
2. Draw the outline of a box using GRAPHICS 5 and fill it in with a different color using XIO 18,#1,0,0,"S:".
3. Produce two modifications of Program 11-9 by varying the position that the lines originate from. Make large changes in the coordinates to see the variety of effects possible through artifacting.

11-6...GRAPHICS Modes 9,10,11,12,13,14,and 15

Type in the following two line program and RUN it.

```
10 GRAPHICS 9
20 GOTO 20
```

If your screen turned black, your Atari has a GTIA chip. If it stayed blue, you don't. The GTIA chip, which can be added to the computers that don't already have it, adds three new graphics modes to your repertoire. These three modes are unusual in their color and shading capabilities, and also in that they draw with a cursor that is four times as wide as it is high. This limits their usefulness for highly detailed plotting and drawing, but permits tremendous color displays that show off your Atari in ways unmatched by other personal computers. All three new graphics modes (GRAPHICS 9, 10, and 11) have no text window at the bottom of the screen and all display on a screen that is 80 columns across and 192 rows long. These modes are unusual in that they display fewer points across than down.

...GRAPHICS 9

GRAPHICS 9 allows 16 shades of any one color. This permits very realistic shading of shapes for a three-dimensional appearance. To select the color that this mode will draw with, use

```
SETCOLOR 4,B,0
```

where B is the color value (0–15). The brightness is then selected with

```
COLOR N
```

where N is 0 to 15; all 16 values are valid. Notice that in this mode COLOR does not select a color register for changing colors, but rather a brightness value. Also note that all 16 shades are now available, rather than just the even-numbered ones.

Here is a simple display of the shading that can be seen.

```
90 REM * GRAPHICS 9 DEMO #1
100 GRAPHICS 9
105 COL=0
110 SETCOLOR 4,COL,0
120 FOR I=0 TO 15
130 COLOR I
135 FOR J=I*4 TO I*4+4
140 PLOT J,10:DRAWTO J,140
150 NEXT J
170 NEXT I
180 FOR COL=0 TO 15
190 SETCOLOR 4,COL,0
200 FOR DELAY=1 TO 250:NEXT DELAY
210 NEXT COL
220 END
```

Program 11-10. GRAPHICS 9 demonstration 1.

The smooth gradation of shading will be clearly seen if you run this program. After you draw the vertical bars of different brightnesses with the loop in lines 135 to 150, the colors are changed to display the various shading possibilities. To better see the capabilities of GRAPHICS 9, we'll write a program to display a wall that alternatively gets taller and shorter to simulate the perspective of closer and further away. We'll continuously vary the shading in each section of the wall to enhance this depth effect. Finally, we'll cycle through the colors. Here's the program.

```
90 REM * GRAPHICS 9 DEMO #2
100 GRAPHICS 9
110 C=0:N=1:COL=0:COLOR 0
120 SETCOLOR 4,C,0
130 X=0:Y1=40:Y2=140
140 PLOT X,Y1:DRAWTO X,Y2
150 X=X+1:Y1=Y1-N:Y2=Y2+N
160 IF X/16=INT(X/16) THEN N=-N
170 COL=COL+1:IF COL>15 THEN COL=0
172 COLOR COL
175 IF X=79 THEN 190
180 GOTO 140
190 C=C+1:IF C>15 THEN END
195 SETCOLOR 4,C,0
200 FOR DELAY=1 TO 500:NEXT DELAY
210 GOTO 190
```

Program 11-11. GRAPHICS 9 demonstration 2.

Lines 130 to 180 draw the wall, varying the height in 16 steps by alternatively decreasing to a minimum and then reversing and increasing. Line 160 divides by 16 to check if it is time to reverse. After the wall is finished, lines 190 to 210 rotate the colors.

...GRAPHICS 10

This interesting graphics mode uses all the regular SETCOLOR registers (0–4) plus four more than can only be accessed with POKE. Since all of the registers can be controlled with the use of POKE, it is generally more convenient to do them all this way, rather than mixing SETCOLOR and POKE statements. Memory location 704 controls the background color; locations 705 to 712 are for foreground colors. The POKE value takes into account both the color and the brightness by using

```
POKE COLREG,B*16+C
```

where COLREG is the memory location (704–712), B is the color (0–15), and C is the brightness (0–14, even numbers). For example, you would use POKE 704,3*16+6 to make the background a medium red (3 for red and 6 for the brightness). Program 11-12, is a modification of Program 3-4, which drew squares in GRAPHICS 3, 5, and 7 using the three colors available. With GRAPHICS 10 we have many more colors to work with and so the display is much more dazzling.

```
90 REM * GRAPHICS 10 #1
95 GOSUB 3000
100 XMAX=79
105 X=INT(XMAX/2)
110 YMAX=191
112 Y=INT(YMAX/2)
120 GOSUB 1000
125 FOR DELAY=1 TO 500:NEXT DELAY
130 END
142 Y=INT(YMAX/2)
998 REM * OUR DRAWING SUBROUTINE
1000 GRAPHICS 10
1005 POKE 704,0
1010 L=2:M=4
1015 PLOT X+L,Y+M
1025 GOSUB 2000
1030 DRAWTO X-L,Y+M
1035 GOSUB 2000
1040 DRAWTO X-L,Y-M
1045 GOSUB 2000
1050 DRAWTO X+L,Y-M
1055 GOSUB 2000
1060 DRAWTO X+L,Y+M
1070 L=L+2:M=M+4
1080 IF L>=XMAX/2 THEN 1100
1090 GOTO 1015
1100 FOR I=1 TO 500:NEXT I
```

```
1105 FOR I=1 TO 100
1110 FOR COLREG=705 TO 712
1120 COL=INT(RND(0)*255)+1
1125 POKE COLREG,COL
1140 NEXT COLREG
1150 FOR J=1 TO 8
1160 COLOR J
1170 NEXT J
1180 NEXT I
1998 REM * ROUTINE TO VARY COLORS AND
BRIGHTNESS
2000 REM * PICK A COLOR TO USE
2010 N=INT(RND(0)*7+1)
2020 COLOR N
2030 RETURN
3000 REM * FILL THE REGISTERS TO START
3005 COL=4
3010 FOR COLREG=705 TO 712
3015 COL=COL+16
3020 POKE COLREG,COL
3025 COLOR N
3030 NEXT COLREG
3040 RETURN
```

Program 11-12. GRAPHICS 10 demonstration 1.

What we have done is remove all references to GRAPHICS 3, 5, and 7 and change the section from lines 1105 to 1180 to work with POKE rather than SETCOLOR. Because of the odd cursor size in Graphics 10, we also changed the value of M in line 1010 from 2 to 4, to better fill the screen with the pattern.

GRAPHICS 10 has fewer simultaneous colors than GRAPHICS 11. However, it allows the programmer to use POKE to insert colors into the color registers after the pattern is drawn. This can be used to produce very striking color motion on the screen. Program 11-13 demonstrates this.

```
90 REM * GRAPHICS 10 DEMO #2
100 GRAPHICS 10
120 COL=0
130 BRITE=2
140 GOSUB 500
150 BRITE=0
200 FOR I=0 TO 70 STEP 8
210 FOR J=1 TO 8
220 COLOR J
230 PLOT I+J,1:DRAWTO I+J,150
240 NEXT J
250 NEXT I
260 GOSUB 300
265 FOR LOOP=1 TO 15
270 GOSUB 500
```

```
275 GOSUB 300
280 NEXT LOOP
290 FOR DELAY=1 TO 500:NEXT DELAY
295 END
298 REM * ROUTINE TO ROTATE COLORS
300 FOR COUNTER=1 TO 25
310 POKE 712,PEEK(705)
320 FOR K=705 TO 711
330 POKE K,(PEEK(K+1))
340 NEXT K
350 NEXT COUNTER
360 RETURN
498 REM * ROUTINE TO SELECT COLORS
500 FOR K=712 TO 705 STEP -1
505 COL=INT(RND(0)*15+1)
507 BRITE=INT(RND(0)*8+2)
510 POKE K,COL*16+BRITE
530 NEXT K
540 RETURN
```

Program 11-13. GRAPHICS 10 demonstration 2.

The program draws vertical bands of color across the screen (lines 200–250). Then it alternately uses two subroutines, one to move the colors from one memory location to another (lines 300–360) and the second to put different colors in the color registers (lines 500–540). The first subroutine simply moves the color in 706 to 705, the color in 707 to 706, and so on. The effect is to have the colors move across the screen.

...GRAPHICS 11

This graphics mode is the inverse of GRAPHICS 9 in that it permits 16 colors (all the same shade) on the screen at the same time. The brightness used by all of the colors on the screen is selected with

```
SETCOLOR 4,0,B
```

where the value of B determines the brightness. The default value is 6. You select the color to use with

```
COLOR N
```

where N can be 0 to 15. Program 11-14, which has the same form as Program 11-10, displays vertical bands of the colors available.

```
90 REM * GRAPHICS 11 DEMO #1
100 GRAPHICS 11
110 SETCOLOR 4,0,4
120 FOR I=0 TO 15
130 COLOR I
135 FOR J=I*4 TO I*4+4
```

```
140 PLOT J,10:DRAWTO J,140
150 NEXT J
170 NEXT I
180 FOR DELAY=1 TO 2000:NEXT DELAY
```

Program 11-14. GRAPHICS 11 demonstration 1.

GRAPHICS 11 permits some very colorful displays. Let's write a program that produces some random, colorful "modern art" canvasses. We will plot points on the screen at random places and with random colors. The RND function will be used to select both. Single points will be too small, so we will draw blocks that are 2 to 6 columns wide in the X direction and 2 to 20 units long in the Y direction. We'll use the RND function to select these values and use the formula from Section 2-2 to ensure that we don't try to draw beyond the screen limits. We can draw each block with a loop that covers all the points that make up that block. Program 11-15 does it all.

```
90  REM GRAPHICS 11 DEMO #2
100 GRAPHICS 11
102 SETCOLOR 4,0,2
105 FOR L=1 TO 300
110 X=INT(RND(0)*(75-4+1))+4
120 Y=INT(RND(0)*(181-11+1))+11
135 P=INT(RND(0)*10+1)
140 Q=INT(RND(0)*3+1)
145 N=INT(RND(0)*15+1)
147 COLOR N
150 FOR K=-P TO +P
155 FOR M=-Q TO Q
160 PLOT X+M,Y+K
170 NEXT M
180 NEXT K
190 NEXT L
200 FOR DELAY=1 TO 2000:NEXT DELAY
```

Program 11-15. GRAPHICS 11 demonstration 2.

Lines 110 to 140 generate the random numbers for one point of the block and check that the full block will fit on the screen. The loop from lines 150 to 180 then plots the block, using the color selected in line 135. The outermost loop with the counter L (see line 105) determines how many blocks will be drawn on the screen. Every time you RUN the program it will produce a different drawing because of the use of RND in lines 110, 120, 135, and 140.

As you can see, graphics modes 9, 10, and 11 are "show-off" graphic modes that make it possible to produce displays of highly varied color and shading.

If you have an Atari 1200 or one of the Atari XL computers, then you also have access to four more graphics modes. The easiest way to find out if you can use these modes from BASIC is to just type GRAPHICS 12 and press RETURN. If you don't receive an ERROR—145 message, then you have these modes at your disposal.

...GRAPHICS 12 and 13

These two modes complement one another and are particularly useful for character graphics—that is, when characters are redefined and then used on the screen to represent different shapes. In fact, it is necessary to redefine the character set in order to use these graphic modes at all. Go ahead and type

```
GRAPHICS 13:PRINT #6;"STRANGE CHARACTERS"
```

You will be greeted with strange-looking letters on the screen. As with GRAPHICS 1 and 2, with GRAPHICS 12 and 13 you use PRINT #6; to put characters on the screen. Unlike those text modes, however, the usual 8-by-8 characters are interpreted very differently. The color for a particular part of a character depends on whether it is on odd or even columns of the 8-by-8 matrix and also on whether the adjacent columns in the same row are used. We'll illustrate this effect by redefining the "C", "H", and "c" characters to all be different forms of the letter "C". See Figures 11-3a, b, c.
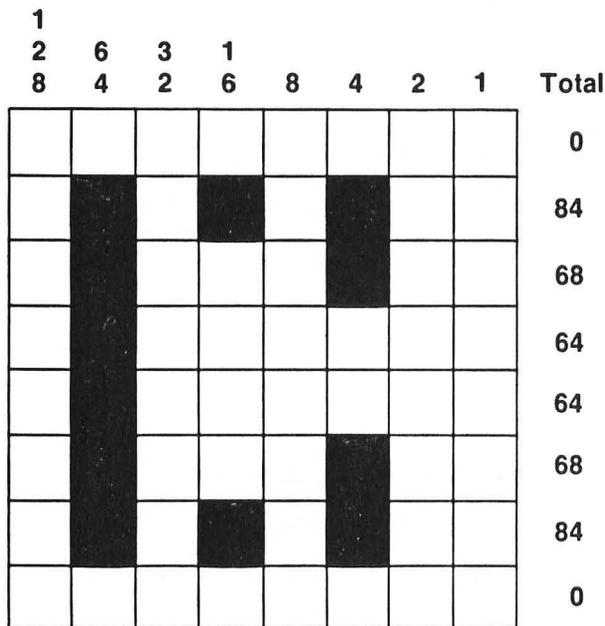
Figure 11-3a. The letter "C" in GRAPHICS 12 (odd columns).

We have used odd columns only, even columns only, and both in defining the three different versions of the letter "C." We will incorporate them in a program that first moves the character set to where it can be modified (see Programmer's Corner 5), then redefines the letters and prints samples of each in both regular and inverse video. Program 11-16 also allows you to use the joystick to see which SETCOLOR values effect which type of letter. Move the joystick left to change color registers, right to change the color in that register, and up to change the brightness. Certain

1 2 8	6 4	3 2	1 6	8	4	2	1	Total
								0
								168
								136
								128
								128
								136
								168
								0

Figure 11-3b. The letter "C" in GRAPHICS 12 (even columns).

1 2 8	6 4	3 2	1 6	8	4	2	1	Total
								0
								252
								204
								192
								192
								204
								192
								0

Figure 11-3c. The letter "C" in GRAPHICS 12 (odd and even columns).

brightnesses will match the background, causing the printing in the text window to disappear. When this happens, just change the brightness by moving the joystick forward to make the printing visible again.

```
10 REM *1200 GRAPHICS 13 DEMO*
20 POKE 106,PEEK(106)-4
30 GRAPHICS 13:N=0:COL=0:BRITE=4
35 PRINT "MOVING THE CHARACTER SET...."
40 CHRLOC=PEEK(106)*256
50 POKE 756,CHRLOC/256
60 FOR SETMOVE=0 TO 1023:POKE CHRLOC+SETMOVE,
PEEK(57344+SETMOVE):NEXT SETMOVE
65 FOR LOOP=0 TO 2
70 READ MEMVAL
80 ACTLOC=MEMVAL*8
90 FOR I=1 TO 8:READ CHARVAL:POKE CHRLOC+
ACTLOC+I,CHARVAL:NEXT I
95 NEXT LOOP
100 POSITION 15,1:? #6;"C C C C C"
105 POSITION 15,2:? #6;"C C C C"
106 POSITION 15,3:? #6;"H H H H H"
107 POSITION 15,4:? #6;"H H H H"
108 POSITION 15,5:? #6;"c  c  c  c  c"
109 POSITION 15,6:? #6;"c c c c"
110 REM C
120 DATA 35,0,168,136,128,128,136,168,0
130 REM H
140 DATA 40,0,84,68,64,64,68,84,0
290 REM c
300 DATA 99,0,252,204,192,192,204,252,0
500 REM
505 X=STICK(0)
510 IF X=11 THEN N=N+1:IF N=5 THEN N=0
520 IF X=7 THEN COL=COL+1:IF COL=16 THEN COL=0
522 IF X=14 THEN BRITE=BRITE+2:IF BRITE=16
THEN BRITE=0
525 SETCOLOR N,COL,BRITE
530 PRINT "SETCOLOR ";N;",";COL;",";BRITE
540 GOTO 500
```

Program 11-16. GRAPHICS 12 demonstration.

The rows of letters on the screen are as follows:

ROW ON SCREEN	HOW DRAWN	HOW PRINTED
1	Odd	Regular
2	Odd	Inverse
3	Even	Regular
4	Even	Inverse
5	Both	Regular
6	Both	Inverse

If you play with this program for awhile, you will come to learn the following relationships:

1. SETCOLOR 0,B,C controls the color of characters drawn in odd-numbered columns.

2. SETCOLOR 1,B,C controls the color of characters drawn in even-numbered columns.

3. SETCOLOR 2,B,C controls the color of characters drawn by using both even and odd columns and printed in regular video. It also controls the color in the text window.

4. SETCOLOR 3,B,C controls the color of characters drawn in odd and even columns and printed in inverse video.

5. SETCOLOR 4,B,C controls the color of the background.

Both of these modes are five-color modes. Unlike GRAPHICS 1 and 2, both permit use of the full character set. GRAPHICS 12 prints characters that are the same size as GRAPHICS 0 and GRAPHICS 13 prints characters that are double height.

Many of the arcade-style games available for the Atari use these modes. For example, a character may be redefined to be a tree and then printed to make a forest on the screen. The advantage of this method is that very little memory is required to display an entire screen, since there are only 40 * 24 = 960 maximum screen positions in GRAPHICS 12 and 40 * 12 = 480 in GRAPHICS 13. By redefining an entire character set, individual shapes and groups of shapes can be printed to portray anything desired.

...GRAPHICS 14 and GRAPHICS 15

These two modes both use a screen that is 160 by 192. GRAPHICS 14 has one color but four brightnesses, while GRAPHICS 15 has four colors with one brightness. GRAPHICS 15 is much like GRAPHICS 7, with the same resolution in the X direction. However, it has twice the resolution in the Y direction. We can make use of this resolution to make multicolored moire patterns. In an earlier section of this chapter, we looked at moire patterns by artifacting with GRAPHICS 8. Program 11-17 shows a similar program, but we have changed the color for each of the lines drawn to give quite a nice display.

```
90 REM * GRAPHICS 15 DEMO
100 GRAPHICS 15+16:COL=1:COLOR COL
105 XSTART=1:YSTART=96
110 FOR I=1 TO 159
120 PLOT XSTART,YSTART:DRAWTO I,0
122 COL=COL+1:IF COL=4 THEN COL=1
124 COLOR COL
130 NEXT I
140 FOR I=1 TO 191
150 PLOT XSTART,YSTART:DRAWTO 159,I
160 COL=COL+1:IF COL=4 THEN COL=1
170 COLOR COL
180 NEXT I
```

```
190 FOR I=159 TO 1 STEP -1
200 PLOT XSTART,YSTART:DRAWTO I,191
210 COL=COL+1:IF COL=4 THEN COL=1
220 COLOR COL
230 NEXT I
250 FOR I=191 TO 1 STEP -1
260 PLOT XSTART,YSTART:DRAWTO 1,I
270 COL=COL+1:IF COL=4 THEN COL=1
280 COLOR COL
290 NEXT I
300 FOR DELAY=1 TO 2500:NEXT DELAY
```

Program 11-17. GRAPHICS 15 demonstration.

By changing the values of XSTART and YSTART in line 105 you can produce different patterns. You might also investigate the effect of changing the colors once the pattern has been completed.

GRAPHICS 15 can also be used to plot functions, since it has fine enough resolution in both the X and Y directions. With its three-color foreground selection you can plot the function in a different color from the axes for a nice visual display. We'll leave this as an exercise.

...SUMMARY

The last two sections have presented information on using the "unusual" graphics modes that are available on some Atari computers. You must have the GTIA chip in your Atari to make use of GRAPHICS 9, 10, and 11. You must have the Atari 1200 or one of the XL series of computers to use GRAPHICS 12, 13, 14, and 15 from BASIC.

GRAPHICS 9, 10, and 11 are noteworthy for their ability to create realistic shading or multicolored displays. To achieve these effects, however, the resolution (compared to GRAPHICS 7 or 8) has been compromised. Nonetheless, they contribute to giving your Atari an outstanding graphics display.

GRAPHICS 12, 13, 14, and 15 can be especially useful to the game programmer who wants to produce colorful displays using a minimum of memory. By redefining characters, the programmer can have animation of colorful characters as easily as it is possible to move text characters around the screen.

Problems for Section 11-6 . . . .

1. Fill the entire screen with vertical bands showing the shading variations of GRAPHICS 9.

2. Write a program that draws horizontal stripes of all the possible shadings available in GRAPHICS 9.

3. Redo Program 11-3 to draw the axis with a different color than the function.

4. Draw a square in GRAPHICS 10 that is 35 units wide and has seven differently colored horizontal stripes, each 15 units high. Make all the stripes different colors and use the default colors.

5. Modify Program 11-17 to change the colors after the pattern is drawn, as suggested in the text.

11-7...An Important Digression

...The Screen Display

For the following section and other advanced techniques, it will be helpful to understand how your Atari manages the screen display. This is the same as asking where and how is the screen information stored in the computer's memory and how does the computer know what sort of screen (that is, graphics mode) it is supposed to display.

Let's begin with how the TV screen display works. If you look closely at your TV screen, you'll see a series of horizontal lines very closely spaced. These are called scan lines. The TV screen that you see is "written" on the picture tube by a single electron beam. It begins in the upper left corner, writes the first line, is turned off (no writing); goes back to the left end of the second line, writes the second line, is turned off; goes back to the left end of the third line, and so on down the screen. When it gets to the right end of the last line at the bottom of the screen, it is turned off and returns to the upper left corner again, ready to do it all over again. This process is repeated 60 times per second! The result is that you don't see the drawing happening at all.

When your TV is connected to your Atari, the computer takes responsibility for maintaining the screen display. All the information on the screen, in whatever graphics mode you are in, is stored in the computer's memory. In fact, it is stored exactly in the same order that the TV electron beam moves—upper left corner first, then scan line by scan line down the screen. All this information is stored like one long string, with scan lines following scan lines in consecutive memory locations.

Also stored in a different area of computer memory is the information that tells the computer what kind of screen it is displaying—GRAPHICS 0, GRAPHICS 8, or whatever. Now, of course, these different graphic screens are addressed by the user very differently. GRAPHICS 0 has text only, with 40 characters across and 24 characters down, while GRAPHICS 8 has 320 points on each of 192 scan lines. We learned in Programmer's Corner 5 that each text character is defined in a grid eight lines high (and eight columns wide). Thus 24 lines * 8 lines per character = 192. The message here is that all the modes use the same number of scan lines (192); they just treat them differently.

It is also important to understand that the computer does this by having a list of what each line to be displayed on the screen contains and a list of how it should be displayed. Note also that the basic "paintbrush" for all screen displays is our electron beam (one scan line wide) and that everything happens so fast that we see the whole picture all of the time.

...A Little Computer Math

It will be useful for our upcoming discussion of new techniques to see how the computer handles the numbers that represent the memory locations of interest in graphics.

We have seen on several occasions that each memory location can hold numbers from 0 to 255 only. What about bigger numbers? These are stored in pairs

of memory locations. The system works much as a digital clock. With the clock, the minute counter goes from 0 to 59. Instead of going to 60, the minute counter goes back to 0 and the hour counter increments by 1. If we want to know the total minutes elapsed since the clock started counting, we take the hour counter value, multiply it by 60 (for 60 minutes per hour) and add the minute counter value. The computer does essentially the same thing, except the counters count from 0 to 255 instead of from 0 to 59. A single memory location holds numbers up to 255. If it contains 255 and we add one more, the computer will reset this location to 0 and add 1 to the next memory location. Note that the first memory location holds the faster changing value. So, if we know that two particular memory locations contain a number we need, we can find that number by getting the value in the second location (called the *high byte*), multiply it by 256, and add the value in the first location (called the *low byte*).

Here are some examples:

NUMBER	HOW STORED		CALCULATION	
240	240	0	240 +	0*256
256	0	1	0 +	1*256
560	48	2	48 +	2*256
53279	31	208	31 +	208*256

Notice again that, contrary to our usual math, the smaller part of the number comes first in the computer's memory.

All of this is important because we want to be able to find where particular screen information is stored in the computer's memory by using the PEEK function. In that way we can change selected screen information (using POKE statements). For example, two memory locations, 88 and 89, hold the number of the start of the screen display in the computer's memory. This value depends on how much memory your computer has. We get this value by applying what we have just learned with

```
START=PEEK(88)+PEEK(89)*256
```

A word of caution is in order. If you POKE the wrong location or POKE the right location with the wrong value, anything can happen (except damage to the computer!). You may see nothing, but later parts of your program may be affected. You may also see the computer "lock up" and refuse to respond to any input (even SYSTEM RESET). If this happens you will have to turn the computer off and then on again, losing any program that was in memory. So be careful and check program statements before running a program. Also, be sure to save your program on tape or disk *before* you RUN it, so that if you do lose control of the computer you can still load the program back in (without retyping it) and then find where you went wrong and fix it.

...Text on a GRAPHICS 8 Screen

As the highest-resolution screen available, GRAPHICS 8 is great for plotting functions. It would be ideal for this purpose if we could put text on the screen to

label the plot. In fact, there are numerous applications where it would be nice to have text on a GRAPHICS 8 screen. With a further look at the screen display, we can do just that.

We already have all the basic information that we need. We know how to find the character set because we used this information to move and then redefine characters. We know that each character in the character set is defined on an eight-by-eight grid. We know where in the computer's memory the screen display starts and how it is stored. With this information we can put any character any place we want on a GRAPHICS 8 screen.

Whenever a character is displayed on the screen in GRAPHICS 0 the computer uses eight scan lines to display the eight lines of the grid that make up the character (see Figure 5-1 in Programmer's Corner 5 to see how the letter "B" is displayed). In GRAPHICS 0 the computer takes care of all of this. In GRAPHICS 8 we'll do exactly the same thing, but our program will be responsible for putting each piece of the character in the correct place on the screen.

This correct place is determined by looking at the TV screen as containing 40 characters across (as in GRAPHICS 0) and 192 lines down (as in GRAPHICS 8). We do this because each line of a character, which is stored as one number in the computer, takes care of eight points across the screen. Thus, eight points for 40 characters is 320 points across the screen (as GRAPHICS 8 requires). To print a character at a specific place on the screen we print the top line of that character there (eight points). Then we move 40 * 8 points further along the display, putting us directly beneath the first point we printed. We then print the second line of the character and repeat this six more times. Program 11-18 does this job.

```
90 REM * PUTTING TEXT IN GRAPHICS 8
92 GRAPHICS 8
95 GOSUB 500
100 DIM A$(15),O$(1)
110 A$="GRAPHICS 8 TEXT":X=5:Y=20
125 SCREENSTART=PEEK(88)+PEEK(89)*256
130 CHARPLACE=SCREENSTART+Y*40+X
140 FOR Z=1 TO LEN(A$)
150 O$=A$(Z,Z):GOSUB 200
160 CHARSTART=57344+X*8
170 FOR I=0 TO 7
180 POKE CHARPLACE+I*40,PEEK(CHARSTART+I)
190 NEXT I:CHARPLACE=CHARPLACE+1
195 NEXT Z
197 END
198 REM * FINDING THE INTERNAL CHARACTER VALUE
200 X=ASC(O$):IF X>127 THEN X=X-128
210 IF X>31 AND X<96 THEN X=X-32:GOTO 230
220 IF X<32 THEN X=X+64
230 RETURN
500 REM * SOME GR.8 DRAWING
510 COLOR 1
520 FOR I=10 TO 150 STEP 4
```

```
530 PLOT I,10:DRAWTO I,100
535 PLOT 10,I:DRAWTO 100,I
540 NEXT I
550 RETURN
```

Program 11-18. Text on GRAPHICS 8 screen.

The subroutine at line 500 draws some GRAPHICS 8 lines. Line 110 contains the message we want printed and where we want it to start (five characters from the left edge and 20 scan lines down from the top of the screen). Line 125 retrieves the memory location where the upper left corner of the screen display starts and line 130 calculates how many points past this we have to go to start printing the first character. The FOR . . . NEXT loop beginning at line 140 takes the characters to be printed one at a time and first uses the subroutine starting at line 200 to determine the internal code for the character. This value, X, permits us to find the eight numbers that comprise that character, since we know where the character set begins (memory location 57344). Finally, the FOR . . . NEXT loop in lines 170 to 190 uses POKE to put the correct line of the character at the correct place on the screen. Note that there is a multiplication by 40 in the POKE location in line 180 to align the parts of the letters properly. To see how the printing takes place, you can add a delay loop to slow down the action with

```
185 FOR DELAY=1 TO 100:NEXT DELAY
```

If you want to print the letters one under another rather than in a single line, change line 190 to read

```
190 NEXT I:CHARPLACE=CHARPLACE+8*40
```

This works because eight scan lines are needed for each character.

...SUMMARY

We have seen how the Atari manages the screen display and have used this information to put text on the GRAPHICS 8 screen. By knowing the meaning of just a few memory locations and how the screen was stored in the computer's memory, we were able to alter the screen information. We saw how the computer stores numbers that are too large to fit in one memory location (that is, greater than 255). We will make further use of this information in Programmer's Corner 11 when we actually alter the screen to display more than one graphics mode at one time.

Problems for Section 11-7 . . . .

1. Write a program that prints the message "TEXT IN GRAPHICS 8" in all four corners of a GRAPHICS 8 screen.

2. Modify Program 11-3 to horizontally label the X-axis with the text "X AXIS" and vertically label the Y-axis with "Y AXIS".

11-8...Player-Missile Graphics

There are many ways to animate graphics in a computer program. You can draw something, erase it, and redraw it at a slightly different position to give the illusion of motion. You can flip from one screen to another, each containing differing figures, much like motion pictures. However, there is a much better method that can be accessed from Atari BASIC, called *player-missile graphics*. The name comes directly from games where players shoot missiles at each other, with resulting sound effects and colorful explosions.

Effective use of player-missile graphics (which we'll call P-M graphics) requires a little understanding of how the images are stored in the computer's memory and a lot of attention to the details of POKE statements that bring about the graphic animation.

P-M graphics is totally independent of the graphics mode that you are working in. You also can color the "players" independently of the mode you are in, thus adding many colors to the display.

What is a player? It is useful to picture a player as a vertical strip on the TV screen that is eight points wide and 256 scan lines high. This extends off both the top and bottom of the screen, allowing the player to disappear off the screen. The player can be located anywhere along this strip, giving motion up and down. The entire strip can be moved left or right, giving motion in those directions. Motion can go off the screen display in these directions also. A player is defined in very much the same way as we defined characters. Instead of having an eight-by-eight grid to work with, however, we have an 8-by-256 grid, although to have room for movement we won't use all of the 256 lines possible.

What is a missile? A missile can most easily be described as a player that is two points wide rather than eight. Therefore, four missiles occupy the same memory space as one player. It is, in fact, possible to combine the four missiles to make a fifth player.

Figure 11-4 shows a diagram of how the players and missiles are stored in the computer's memory. The label P-MBASE at the top of that figure refers to where in memory we start to store the player-missile information. As with redefined characters, we need to reserve memory for these purposes. We do it just as we did then, by tricking the computer and shifting the place that the computer thinks is the top of the memory. There are differences in our table depending on whether you are using single or double-line resolution, which simply means whether each number that corresponds to a line in the player occupies one or two screen scan lines. We will work here with single-line resolution since this permits the most detailed figures (but consumes the most memory). The vertical position of a player on the screen is determined by where in the 256-byte strip that is reserved for that player the image is located. By moving the numbers that make up the player around in this memory area, we move the image up and down the screen. We move the image horizontally more easily. We merely need to use POKE to insert a value into a certain memory location and the player moves instantaneously.

(Single Resolution)					(Double Resolution)
P-M Base -		Unused			- P-M Base
+768		Missiles			+384
	0	1	2	3	
+1024					+512
		Player 0			
+1280					+640
		Player 1			
+1536					+768
		Player 2			
+1792					+896
		Player 3			
+2048					+1024

Figure 11-4. Memory locations for P-M information.

Let's now go over the memory locations that are important when you are using P-M graphics, and then define a player and make it do something on the screen.

1. Memory location 559: Put a 62 here for single-line resolution and a 46 here for double-line resolution.

2. Memory location 53277: Put a 3 here to enable P-M graphics and a 0 here to disable P-M graphics. By putting a 3 here we instruct the computer that we want it to begin to use this type of graphics.

3. Memory location 623: Establishes priorities for movement. You can use the values put here to create the illusion of objects passing in front of or behind other objects. We have already discussed players, and there can be four of them (numbered 0–3). There are also *playfields*; they are also numbered 0 to 3. There is nothing mysterious about playfields; they are merely anything drawn on the screen by using a color register of the same number. The number of playfields possible will then depend on the graphics mode that you are working in. Keep in mind that it is not the color of the playfield that matters (which is set by a SETCOLOR statement), but rather the COLOR value. You have your choice of the following priorities (and only these choices):

   a. A value of 1 gives all players priority over all playfields. Therefore, all players will appear to pass in front of anything else drawn on the screen.

   b. A value of 4 will give all playfields priority over all players, giving the appearance of players passing in back of anything drawn on the screen.

   c. A value of 2 gives the following order of priorities: Player 0 and Player 1, then all playfields, then Player 2 and Player 3.

> d. A value of 8 gives this order of priorities: playfield 0 and playfield 1, then all players, then playfield 2 and playfield 3.
> In cases c and d, players will appear to go behind some playfields (those with higher priority) and in front of others (those with lower priority).

4. Memory location 54279: We will use PEEK(106) to find the current top of memory and then subtract off an appropriate value before using POKE to insert this value into 54279 to inform the computer where our P-M table starts. The value we subtract off will be a multiple of four for double-line resolution and eight for single-line resolution. Given this restriction, the number actually subtracted will depend on the graphics mode that is being used for the display.

5. Memory locations 704 to 707: These are used to establish the color for players 0 to 3, with the color of player 0 in 704, and so on. Colors are handled as we have seen before, with the value of the color times 16 plus the brightness inserted into the appropriate location by a POKE statement.

6. Memory locations 53248 to 53251: These are used to hold the horizontal position (0 to 255) for players 0 to 3, with the horizontal position of player 0 in 53248, and so on. Note that the visible screen has values from 48 to 221, with values less than this being off the left edge and values greater than this being off the right edge.

7. Memory locations 53252 to 53255: These hold the horizontal position for missiles 0 to 3 and are handled exactly as for players.

8. Memory locations 53256 to 53259: These are locations that control the width of a player, with player 0 controlled by 53256, and so on. The following values are used:
  a. A value of 0 is for normal width. A value of 2 is treated the same.
  b. A value of 1 is for double width.
  c. A value of 3 is for quadruple width.
  What this means is that a player's width can be changed merely by placing the appropriate value in this memory location.

9. A number of memory locations serve multiple purposes. The computer can sort out the use intended by the values that it finds in the location. Thus there are several memory locations that serve to indicate the presence of collisions between missiles and players, players and players, missiles and playfields, and players and playfields. This very powerful feature takes away a lot of complications that could arise if you had to keep track of these things with your own programming. There is also one location, 53278, appropriately named HITCLR, that will clear all the collision registers if it is filled with any value other than 0. You can only use POKE to manipulate location 53278. Using PEEK to access this location will not give you any meaningful value. The collision registers themselves can only be accessed by using PEEK. The only way to alter their values is through HITCLR. Of course, if the appropriate colli-

sion occurs, the value in the correct collision register is changed automatically by the computer.

Here's a table showing which memory locations track which collisions. In the table, the column headed "WHICH ONE" refers to collisions between the first item and the second item. For example, REGISTER 53252 keeps track of collisions between player 0 and the various playfields.

TYPE	WHICH ONE	REGISTER
Player-player	0	53260
	1	53261
	2	53262
	3	53263
Missile-player	0	53256
	1	53257
	2	53258
	3	53259
Player-playfield	0	53252
	1	53253
	2	53254
	3	53255
Missile-playfield	0	53248
	1	53249
	2	53250
	3	53251

Table 11-5. Collision registers.

The values that you will find on using PEEK to access one of these locations will, of course, depend on which object your player or missile runs into. The values follow a pattern that is the same for all cases. Let's consider a collision of player 0 with playfields (memory location 53252). The value you get will be:

1 for a collision with playfield 0
2 for a collision with playfield 1
4 for a collision with playfield 2
8 for a collision with playfield 3

Recall that playfield 1 is merely anything on the screen drawn by using COLOR 1. A collision with playfields 1 and 2 will result in a value of 6, and so forth. In this way all possible collisions are accounted for with unique values.

...Building an Example

Let's put some of this information to use. We'll begin by defining a player and then give it some movement. Then we'll add more color, different sizes, a playfield, priorities, and collisions. We'll do it one step at a time to show how the process is developed.

...Our Player

A rocket ship makes a nice shape to move on the screen. Figure 11-5 shows the rocket shape we want and the numbers that define its shape. So far it is pretty much like defining any other character, except it is taller.

Figure 11-5. Our rocket-shaped "player."

...Adding Movement

Now let's get things moving on the screen. Let's go over Program 11-19a line by line to see how it's done.

```
90 REM * SIMPLE PLAYER-MISSILE GRAPHICS
100 GRAPHICS 3+16
110 SETCOLOR 4,0,0
120 POKE 559,62
130 POKE 53248,10
140 COL=84
150 POKE 704,COL
160 POKE 623,1
200 I=PEEK(106)-16
210 POKE 54279,I
220 POKE 53277,3
300 J=I*256+1024
310 FOR Y=J TO J+255
320 POKE Y,0
330 NEXT Y
350 FOR Y=J+120 TO J+134
360 READ Z
370 POKE Y,Z
380 NEXT Y
500 FOR X=10 TO 220
502 SOUND 1,230,6,4
505 POKE 53248,X
510 FOR DELAY=1 TO 10:NEXT DELAY
520 NEXT X
700 GOTO 500
2000 DATA 64,96,112,112,254,254,255,235
2010 DATA 255,254,254,112,112,96,64
```

Program 11-19a. Start-up routines for player-missile program.

LINE NUMBER	WHAT IT DOES
100	Establish our background mode
110	Determine the background color (black)
120	We want to use single-resolution graphics
130	Horizontal position for player 0 at start
140	The color we will use for player 0
150	Give player 0 this color
200	Where our P-M table will be located in memory
210	Inform the computer of this fact
220	Enable player-missile graphics
300–330	Be sure there's nothing but 0s in player 0 area
350–380	Put our player 0 shape near the middle of its space
500–520	Move player 0 across the screen
2000–2010	Our player 0 shape data

The program has a delay at line 510 to slow down the movement so we can appreciate it. Different delay values will result, of course, in different speeds across the screen. We have also used sound to give a low rocket-like rumble (line 502).

...Adding More Color

We can easily change the color of the rocket each time it crosses the screen by adding the following lines to our program

```
530 COL=COL+16
535 IF COL>=255 THEN COL=4
540 POKE 704,COL
```

Program 11-19b. Color routine for player-missile program.

All we have done is to add 16 to the COL value each time across the screen. Then we use POKE to place this new value into the player 0 color register (704).

...Changing Sizes

It is even easier to change the size of player 0 each time it goes across the screen. Recall that there are three sizes possible. Add the following lines to our program.

```
550 SIZE=SIZE+1
555 IF SIZE>=4 THEN SIZE=0
560 POKE 53256,SIZE
```

Program 11-19c. Size change routine for player-missile program.

We change the size value each time around, resetting it to 0 when it reaches a value of 4. Then we use POKE to place the value into the player 0 size register (53256). Add this change and RUN the program and you will see the rocket first as regular size, then double size, then regular size again, then quadruple size. Things are beginning to look impressive.

...Adding a Background

For a more colorful display we can add a background for the rocket to fly past. We are in GRAPHICS mode 3 so we can have three foreground colors. We'll add a routine to the program to draw six vertical stripes on the screen in three different colors. With more effort, these could be mountains or clouds or whatever. Add Program 11-19d.

```
110 GOSUB 1000
1000 START=0:N=1
1002 SETCOLOR 4,0,0
1005 FOR STRIP=1 TO 6
1010 COLOR N:START=START+5
1015 GOSUB 1100
1020 N=N+1:IF N=4 THEN N=1
1030 NEXT STRIP
1100 FOR I=START TO START+3
1110 PLOT I,0:DRAWTO I,19
1120 NEXT I
1130 RETURN
```

Program 11-19d. Background routine for player-missile program.

...Depth Effects with Priority

This is the easiest change of all. Adding one line does the job.

249

```
160 POKE 623,8
```

Program 11-19e. Adding priority to player-missile program.

This establishes that player 0 has priority over playfield 2, but not over playfields 0 and 1. Add this line and you will see that our rocket appears to pass behind stripes 1, 2, 4, and 5 and in front of stripes 3 and 6. Change the 8 to 1, 2, or 4 and the priorities change.

...Collisions with Playfields

For our demonstration, let's keep track of when our player 0 collides with playfield 1, which is stripes 2 and 5. Collisions of this type are monitored by memory location 53252 (see Table 11-5). We'll have the program make a different sort of sound as the rocket passes this stripe. This will give the program the effect of having the rocket pass behind stripes 1 and 4, through stripes 2 and 5 (with the sound accompaniment), and in front of stripes 3 and 6. Add Program 11-19f.

```
510 GOSUB 800
800 BUMP=PEEK(53252)
810 ON BUMP+1 GOTO 830,830,820,820,830,830,820
,820
820 SOUND 0,1,4,6:SOUND 2,5,4,6
825 FOR DELAY=1 TO 10:NEXT DELAY:GOTO 840
830 FOR DELAY=1 TO 10:NEXT DELAY
835 SOUND 0,0,0,0:SOUND 2,0,0,0
840 POKE 53278,1
850 RETURN
```

Program 11-19f. Sound effects for player-missile program.

We check the value in the register that checks on collisions of player 0 with playfields (53252). Then we make a sound or turn it off, depending on the value we find, using ON . . . GOTO in line 810. We want to make our drilling sound when any part of our player 0 is "in" the stripe that is playfield 1. Because our player has variable width we need to check not only on collisions with playfield 1, but also on multiple collisions that occur with this playfield and with those on either side of it as the rocket interacts with several stripes at the same time. For example, the value in the collision register will change when player 0 is part way out of playfield 1 and part way into playfield 2, so we have to check on this possibility also. The values we have to watch for are thus 2, 3, 6, and 7. BUMP in line 800 can have values 0 to 6, so in line 810 we add 1 to BUMP and use the ON . . . GOTO statement to direct the program to the correct line numbers, depending on which stripe(s) our rocket is currently touching.

Finally, we present the entire program, with REM statements indicating the various subsections, for your study. See Program 11-19.

```
90 REM * SIMPLE PLAYER-MISSILE GRAPHICS
100 GRAPHICS 3+16
110 GOSUB 1000
119 REM * INITIAL P-M SETUP
```

```
120 POKE 559,62
130 POKE 53248,10
140 COL=84
150 POKE 704,COL
160 POKE 623,8
200 I=PEEK(106)-16
210 POKE 54279,I
220 POKE 53277,3
300 J=I*256+1024
308 REM * CLEAR OUT PLAYER AREA
310 FOR Y=J TO J+255
320 POKE Y,0
330 NEXT Y
348 REM * POKE IN PLAYER DATA
350 FOR Y=J+120 TO J+134
360 READ Z
370 POKE Y,Z
380 NEXT Y
498 REM * THE MOVEMENT LOOP
500 FOR X=10 TO 220
502 SOUND 1,230,6,4
505 POKE 53248,X
510 GOSUB 800
520 NEXT X
528 REM * CHANGE COLOR
530 COL=COL+16
535 IF COL>=255 THEN COL=4
540 POKE 704,COL
548 REM * CHANGE SIZE
550 SIZE=SIZE+1
555 IF SIZE>=4 THEN SIZE=0
560 POKE 53256,SIZE
700 GOTO 500
798 REM * COLLISION ROUTINE WITH SOUND
800 BUMP=PEEK(53252)
810 ON BUMP+1 GOTO 830,830,820,820,830,830,820
,820
820 SOUND 0,1,4,6:SOUND 2,5,4,6
825 FOR DELAY=1 TO 10:NEXT DELAY:GOTO 840
830 FOR DELAY=1 TO 10:NEXT DELAY
835 SOUND 0,0,0,0:SOUND 2,0,0,0
840 POKE 53278,1
850 RETURN
998 REM * BACKGROUND STRIPE SETUP
1000 START=0:N=1
1002 SETCOLOR 4,0,0
1005 FOR STRIP=1 TO 6
1010 COLOR N:START=START+5
1015 GOSUB 1100
1020 N=N+1:IF N=4 THEN N=1
```

```
1030 NEXT STRIP
1098 REM * DRAW STRIPES
1100 FOR I=START TO START+3
1110 PLOT I,0:DRAWTO I,19
1120 NEXT I
1130 RETURN
2000 DATA 64,96,112,112,254,254,255,235
2010 DATA 255,254,254,112,112,96,64
```

Program 11-19. Player-missile program.

There is much more that can be done. We don't have any vertical movement at all. We also could add a "cannon" at the bottom that could be moved with a joystick and shoot missiles at the rockets. The rockets could explode when struck by a missile. The possibilities are really endless; the only limitation is your imagination.

...Vertical Player Movement

Moving a player horizontally involved merely using POKE statements to place a number into the horizontal position register to instantaneously erase the current image and move the player to the desired location. Vertical movement is much more complex. We have seen how the vertical position depends on where in the 256-byte-high strip the player information is placed. Moving vertically means moving the numbers that are the player image up or down this strip *and* erasing any numbers that are left behind. Failure to erase numbers will result in leaving parts of the image on the screen even after the player has moved. Let's demonstrate vertical and horizontal movement with a single player (a double arrow which is defined in the usual way by the DATA statement in line 2000) and have the movement controlled with the joystick.

```
90 REM * PLAYER-MISSILE GRAPHICS
100 GRAPHICS 3+16
108 REM * INITIAL P-M SETUP
110 SETCOLOR 4,0,0
120 POKE 559,62
130 POKE 53248,120
135 XPOS=120:YPOS=120
140 COL=84
150 POKE 704,COL
200 I=PEEK(106)-16
210 POKE 54279,I
220 POKE 53277,3
300 J=I*256+1024
310 FOR Y=J TO J+255
320 POKE Y,0
330 NEXT Y
350 FOR Y=J+120 TO J+128
360 READ Z
370 POKE Y,Z
380 NEXT Y
```

```
390 YPOS=J+120
498 REM * JOYSTICK ROUTINE
500 XX=STICK(0)
510 IF XX=14 THEN GOSUB 600
520 IF XX=13 THEN GOSUB 700
530 IF XX=11 THEN GOSUB 800
540 IF XX=7 THEN GOSUB 900
590 GOTO 500
598 REM * MOVE UP
600 FOR Y=YPOS-1 TO YPOS+8
610 POKE Y,PEEK(Y+1)
620 NEXT Y
630 YPOS=YPOS-1
640 RETURN
698 REM * MOVE DOWN
700 FOR Y=YPOS+9 TO YPOS STEP -1
710 POKE Y,PEEK(Y-1)
720 NEXT Y
730 YPOS=YPOS+1
740 RETURN
798 REM * MOVE LEFT
800 XPOS=XPOS-1
810 POKE 53248,XPOS
820 RETURN
898 REM * MOVE RIGHT
900 XPOS=XPOS+1
910 POKE 53248,XPOS
920 RETURN
2000 DATA 24,60,126,255,24,255,126,60,24
```

Program 11-20. Demonstration of vertical movement.

The initial part of the program is used to set up the player-missile graphics and is nearly the same as our initial player-missile demonstration. In line 135 we set the values for YPOS and XPOS, the vertical and horizontal positions of our player. In lines 350 to 380 we use POKE to enter the values of this player. In lines 500 to 540 we check the joystick position to determine the desired direction of movement and then go to the appropriate subroutine. The vertical movement routines at lines 600 and 700 work in similar ways. Let's look at moving up the screen (which means moving toward lower memory locations in the player stripe). We begin by moving the first value of the player image to one lower memory location. Then the second value is moved to where the first one was. We continue this until all nine values have been moved one location. At this point the image has moved one scan line on the screen, but the last value of the player is still on the screen in two places, where we moved it and where it originally was. We, therefore, repeat the procedure a tenth time to move a 0 into the location occupied by our duplicated value. This completes one cycle of the "up" vertical movement. Horizontal movement (subroutines at lines 800 and 900) merely requires a POKE statement altering one value.

If you RUN this program it will be painfully clear that vertical movement is

much more tedious and slow than horizontal movement. There are some techniques using strings that can make this faster, but they are somewhat complex mathematically. Instead, we will give you a short machine-language routine to provide rapid vertical movement. To use this routine there is a little more information that you need to have.

BASIC takes each program line and translates it into commands that the computer understands. Thus, it is an interpreter between English and machine language. Using machine language allows us to bypass the BASIC language interpreter and program directly in fundamental commands that the computer understands. This provides for very fast program execution, but is difficult to work with and totally unforgiving when it comes to errors. A detailed discussion of machine language is beyond the scope of this Atari BASIC book. We will simply introduce the BASIC statement that permits you to use a machine language routine from a BASIC program. Atari BASIC has a USR statement, which is called with

```
100 X=USR(LOC)
```

where LOC is the memory location where the routine is stored. The X can be any legal variable name, it doesn't matter what it is as long as it is there. In our example LOC will be 1536, since it turns out that there is an area of memory that is not used by BASIC (from 1536 to 1791) and that is therefore available for storing our routine. The routine is placed in this memory area by using READ . . . DATA statements to input the information needed and POKE statements to place it in the computer's memory. The routine is actually two routines: one for up and one for down motion. They do exactly what our BASIC vertical movement routine did, but they do it much faster.

To use the routine, you need to provide the program with three values: the length of the player (number of values used to make the player) and the two numbers that locate the player initially. This is done with

```
400 LO=INT(YPOS/256)
410 POKE 205,LO
420 POKE 204,YPOS-LO*256-1
430 POKE 206,9
```

where 9 is the player length in our example. Program 11-21 is identical to the previous program, except that the vertical movement is controlled by the machine-language subroutine. Notice how much more quickly the player now moves in the vertical direction.

```
90 REM * PLAYER-MISSILE GRAPHICS
95 REM * WITH M.L. VERTICAL MOVEMENT
100 GRAPHICS 3+16
105 GOSUB 2500
108 REM * INITIAL P-M SETUP
110 SETCOLOR 4,0,0
120 POKE 559,62
130 POKE 53248,120
135 XPOS=120:YPOS=120
```

```
140 COL=84
150 POKE 704,COL
200 I=PEEK(106)-16
210 POKE 54279,I
220 POKE 53277,3
300 J=I*256+1024
310 FOR Y=J TO J+255
320 POKE Y,0
330 NEXT Y
340 RESTORE 2000
350 FOR Y=J+120 TO J+128
360 READ Z
370 POKE Y,Z
380 NEXT Y
390 YPOS=J+120
400 LO=INT(YPOS/256)
410 POKE 205,LO
420 POKE 204,YPOS-LO*256-1
430 POKE 206,9
498 REM * JOYSTICK ROUTINE
500 XX=STICK(0)
510 IF XX=14 THEN GOSUB 600
520 IF XX=13 THEN GOSUB 700
530 IF XX=11 THEN GOSUB 800
540 IF XX=7 THEN GOSUB 900
590 GOTO 500
598 REM * MOVE UP
600 A=USR(1600)
640 RETURN
698 REM * MOVE DOWN
700 A=USR(1650)
740 RETURN
798 REM * MOVE LEFT
800 XPOS=XPOS-1
810 POKE 53248,XPOS
820 RETURN
898 REM * MOVE RIGHT
900 XPOS=XPOS+1
910 POKE 53248,XPOS
920 RETURN
2000 DATA 24,60,126,255,24,255,126,60,24
2500 REM * ML VERTICAL MOVE ROUTINE
2515 RESTORE 2560
2520 FOR I=1600 TO 1616
2530 READ DAT
2540 POKE I,DAT
2550 NEXT I
2560 FOR I=1650 TO 1664
2570 READ DAT
2580 POKE I,DAT
```

```
2585 NEXT I
2590 RETURN
3000 DATA 104,160,1,177,204,136,145,204,196,
206,200
3010 DATA 200,144,245,198,204,96,104,164,206,
177,204
3020 DATA 200,145,204,136,136,16,247,230,204,
96
```

Program 11-21. Vertical movement using machine language.

...SUMMARY

The use of players and missiles is tricky to master. There are many new concepts, such as position registers, collision registers, and priority. The animation that can be created, however, makes the effort worthwhile. You will have to struggle through many simple programs before you can confidently put together a complicated piece of colorful movement.

Problems for Section 11-8 ....

1. Design your own rocket or other shape to go with Program 11-19.
2. Modify Program 11-19 to bring the spaceship onto the screen at different vertical positions each time.
3. Make the "playfield" in the player-missile demonstration into something more interesting than simple stripes (a mountain? a series of clouds?); that is, modify 11-19 to make it more realistic. Make this a part of the overall program.

PROGRAMMER'S CORNER 11

...Multiple Screen Formats

With what we already know and one more concept we can create a TV screen with different graphics modes on display at the same time. Think of a display with bold, colorful GRAPHICS 2 on the top, some GRAPHICS 7 drawing in the center, a GRAPHICS 1 title below that, and more GRAPHICS 2 text on the bottom. This or nearly any other combination is possible once we understand the *display list*.

The display list is the name used for the list in the Atari's memory that indicates how a screen is to be displayed. It is actually a "program" for one of the electronic circuits in the Atari called ANTIC. The programming "language" that ANTIC understands is quite crude, but sufficient to provide the screen display. Whenever you issue a GRAPHICS N command or on initial power-up (when the computer goes into GRAPHICS 0), a display list is set up in memory. The start of this list is stored in memory locations 560 and 561. We retrieve this number by using

```
DLIST=PEEK(560)+256*PEEK(561)
```

Let's look at our GRAPHICS 0 display list with the following program.

```
90 REM * DISPLAY GR.0 DISPLAY LIST
100 DLIST=PEEK(560)+PEEK(561)*256
110 FOR I=DLIST TO DLIST+31
120 PRINT PEEK(I);"   ";
130 NEXT I
140 PRINT
150 END
```

Program 11-22. Display list for GRAPHICS 0.

RUN

```
112   112   112   66   64   156   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2   2
 2   2   2   2   2   2   2   65   32   156
```

READY

Figure 11-6. Execution of Program 11-22.

The first three 112 s are instructions to write eight blank scan lines each, for a total of 24 blank scan lines. This explains the blank area at the top of various screen displays. The next number actually describes the first display line and also tells the computer that the next two numbers will indicate where in memory the screen display starts. The 66 in our GRAPHICS 0 example is 64 + 2, with the 64 the "display location coming next" instruction and the 2 telling the computer that the first line is to be in GRAPHICS 0. A 2 for GRAPHICS 0? Yes, as Table 11-6 shows, our usual graphics modes 0 to 8 do not have the same designations in the display list.

ANTIC INSTRUCTION CODE	BASIC GRAPHICS MODE	BYTES PER LINE	SCAN LINES PER DISPLAY LINE
2	0	40	8
3	—	40	10
4	—	40	8
5	—	40	16
6	1	20	8
7	2	20	16
8	3	10	8
9	4	10	4
10	5	20	4
11	6	20	2
12	—	20	1
13	7	40	2
14	—	40	1
15	8	40	1

Table 11-6. Comparison of ANTIC display list and BASIC graphics modes.

Next comes the two-number indication of the start of screen display, which will depend on how much memory your computer has. In our case, we have 96 and 144, which means that the screen display stored in memory starts at $96 + 256 * 144 = 36960$. There are then 23 more 2's indicating, as expected, 23 more lines of GRAPHICS 0 display. Then comes a 65, indicating the end of the display list and two more numbers that tell the computer where to go next, when the computer gets back to displaying the screen after doing some other work.

Each 2 represents eight scan lines, since, as we learned earlier, the characters in GRAPHICS 0 are formed in an eight-dot-high box. Now, 24 lines (each eight scan lines high) require 192 scan lines. This is one number we want to keep in mind, since when we make changes to alter the display list, we'll need to set the total scan lines used less than or equal to 192 to avoid losing control over the display.
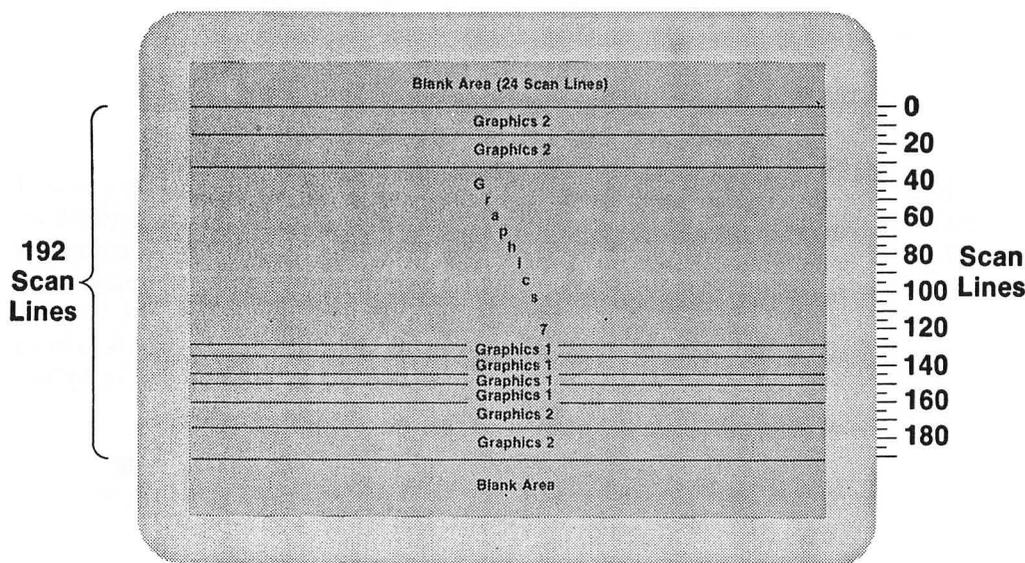
Figure 11-7. Layout of three graphics modes on one screen.

So let's get to it. Figure 11-7 shows what we want. We'll start as usual with the blank area at the top. Then we'll put four lines of GRAPHICS 2. We'll want two lines of GRAPHICS 2 at the bottom and four lines of GRAPHICS 1 just above them. The remainder of the screen will be in GRAPHICS 7. Here's the calculation that makes up the display:

MODE	LINES	SCAN LINES	LINE TOTAL
2	4	16	64
1	4	8	32
2	2	16	32

This leaves $192 - 128 = 64$ scan lines, which means 32 lines of GRAPHICS 7, since each line of GRAPHICS 7 requires two scan lines.

When we make our new display list, we'll want to start with the graphics mode that uses the most memory, which in our example is GRAPHICS 7. Its display list is similar in structure to the GRAPHICS 0 display list, but its third value will be 77 (64 + 13) and it will have 95 13s rather than 23 2s. Getting back to our new display list, it will look like this:

112 112 112 71 www xxx 7 7 7 6 6 6 6 13 13 . . . . . 13 13 7 7 65 yyy zzz

where there will be 31 13's and the values indicated as www, xxx, yyy, and zzz will be as in the original display list (yyy and zzz are the 103rd and 104th numbers in the regular GRAPHICS 7 display list). We can easily write a program with a FOR . . . NEXT loop to read these numbers in from DATA and then use POKE to place them in the required locations.

```
90 REM * MAKING A CUSTOM DISPLAY LIST
95 GRAPHICS 7+16
98 DIM A$(20),O$(1)
100 DLIST=PEEK(560)+PEEK(561)*256
110 FOR I=DLIST TO DLIST+3
120 READ X:POKE I,X
130 NEXT I
132 FOR I=DLIST+6 TO DLIST+8
134 READ X:POKE I,X
136 NEXT I
140 FOR I=DLIST+9 TO DLIST+9+31
150 POKE I,13
160 NEXT I
170 FOR I=DLIST+41 TO DLIST+41+5
180 READ X:POKE I,X
190 NEXT I
192 POKE DLIST+47,65
194 POKE DLIST+48,PEEK(DLIST+103)
196 POKE DLIST+49,PEEK(DLIST+104)
1998 REM * DISPLAY LIST DATA
2000 DATA 112,112,112,71
2010 DATA 7,7,7
2020 DATA 6,6,6,6
2030 DATA 7,7
```

Program 11-23a. Initialize memory for a custom display list.

At this point the screen has been modified but, with nothing printed on it, you can't tell. So next we'll PRINT some text on the areas that are text modes and draw something on the GRAPHICS 7 area. To do the printing, we need to know where each portion is stored in the computer memory and how to position the text properly. The start of screen display in memory locations 88 and 89 is found with

```
START=PEEK(88)+256* PEEK(89)
```

Our new display list has four lines of GRAPHICS 2 at its start; each line in that mode holds 20 characters. Therefore, by successively adding 20 to START we can get to

the start of lines 2, 3, and 4 on the screen. Adding 20 more gets us to the start of the GRAPHICS 7 area. Since there are 32 lines of GRAPHICS 7 this means 32 * 40 = 1280 eight-dot-wide characters in the GRAPHICS 7 space on the screen. Therefore, adding 20 + 1280 to the last value we assigned to START will place us at the start of the GRAPHICS 1 area of our display. To find the start of each subsequent line we again can add 20 to START for each line, since both GRAPHICS 1 and GRAPHICS 2 have 20 characters per line. In this way we can properly position our messages on the screen. You can also work it out by trial and error, but a little understanding can go a long way toward simplifying the process. Here's our entire program.

```
90 REM * MAKING A CUSTOM DISPLAY LIST
95 GRAPHICS 7+16
98 DIM A$(20),O$(1)
100 DLIST=PEEK(560)+PEEK(561)*256
110 FOR I=DLIST TO DLIST+3
120 READ X:POKE I,X
130 NEXT I
132 FOR I=DLIST+6 TO DLIST+8
134 READ X:POKE I,X
136 NEXT I
140 FOR I=DLIST+9 TO DLIST+9+31
150 POKE I,13
160 NEXT I
170 FOR I=DLIST+41 TO DLIST+41+5
180 READ X:POKE I,X
190 NEXT I
192 POKE DLIST+47,65
194 POKE DLIST+48,PEEK(DLIST+103)
196 POKE DLIST+49,PEEK(DLIST+104)
498 REM * PRINT AND PLOT ON NEW SCREEN
500 START=PEEK(88)+256*PEEK(89)
520 A$=" NEW display LIST":GOSUB 1000
530 A$="     FEATURING":START=START+20:GOSUB
1000
540 A$="    graphics two":START=START+20:GOSUB
1000
550 A$="     CHARACTERS":START=START+20:GOSUB
1000
560 A$="a gr.1 title line"
570 START=START+1300
580 GOSUB 1000
590 FOR COUNT=1 TO 3
600 A$="more GRAPHICS ONE"
610 START=START+20
620 GOSUB 1000
630 NEXT COUNT
640 START=START+20
650 A$="      AND MORE"
660 GOSUB 1000
```

```
670  START=START+20
680  A$="   graphics two"
690  GOSUB 1000
700  COLOR 2
710  FOR I=10 TO 150 STEP 4
720  PLOT I,2:DRAWTO I,30
730  NEXT I
740  COLOR 3
750  FOR I=2 TO 30 STEP 4
770  PLOT 10,I:DRAWTO 150,I
780  NEXT I
790  COLOR 1
800  FOR I=1 TO 100
810  X=INT(RND(0)*140+10)
820  Y=INT(RND(0)*28+2)
830  PLOT X,Y
840  NEXT I
850  GOTO 850
998  REM * INTERNAL CODE CONVERSION
1000 FOR I=1 TO LEN(A$)
1020 O$=A$(I,I)
1030 GOSUB 1100
1040 POKE START+I,X
1045 NEXT I
1050 RETURN
1098 REM * CONVERT TO INTERNAL CODE
1100 X=ASC(O$):IF X>127 THEN X=X-128
1110 IF X>31 AND X<96 THEN X=X-32:GOTO 1230
1120 IF X<32 THEN X=X+64
1230 RETURN
1998 REM * DISPLAY LIST DATA
2000 DATA 112,112,112,71
2010 DATA 7,7,7
2020 DATA 6,6,6,6
2030 DATA 7,7
```

Program 11-23. Making a custom display list.

Notice the subroutines at lines 1000 and 1100. We again use them to take care of finding the internal codes for the characters to be printed and for taking care of placing the characters on the screen, once we have provided a value for START.

We have put together an assortment of information to give you the tools to modify the display screen. Understanding a little about the display list has permitted us to produce an impressive visual display. In the process, we also learned information about the inner workings of your Atari computer that can be applied to other problems. That, in essence, is what this whole book has been about.

Appendix A

Useful Memory Locations for the BASIC Programmer

Throughout this book we have mentioned using the PEEK function to determine the value stored in a particular memory location and the POKE statement to change values in memory locations. For your convenience we here present a list of the PEEK and POKE locations used throughout the book along with a number of others that you may find useful in your BASIC programming.

As a refresher, recall that you use

```
X=PEEK(ABC)
```

to determine the value stored in memory location ABC, and

```
POKE ABC,X
```

to store the value X in the memory location ABC. Also recall that, for values greater than 255, the value is stored in two consecutive memory locations. (See Section 11-7 for a more thorough discussion.)

With this in mind, we present our PEEK and POKE list, arranged by function and by increasing memory location number within a section. The list is far from complete, containing only those locations that we have found to be the most useful for the BASIC programmer. For books containing a more complete list of memory locations, see Appendix D.

...The Keyboard
8: Simulating the BREAK key.
Using POKE to insert a value of 0 will cause a "warmstart," which is exactly what happens when you press the BREAK key. Your program is not erased. Normally this location contains a value of 255.

16: Protecting against accidentally pressing the BREAK key.
Use POKE to place a 16 into this location (and also into 53774) and you will disable the BREAK key; that is, pressing the BREAK key will have no effect on the program. Because this key is near the RETURN key, there is a good chance that it will be pressed if the user is requested to input data during the program. If this happens, of course, the execution of your program will terminate.

702: Forced shift lock.
You can use POKE to insert a value of 64 here to cause the next input to be in capital letters. This can be useful if your program is matching an input against an expected word and is looking for capital letters only. This protects against the user having shifted to lowercase. Similarly, a value of 0 will make the input lowercase.

755: Type of character displayed.
The values placed here with POKE statements have the following effect on the display: 0 will display inverse video as regular video characters; 1 will not display inverse video (you will get blank spaces instead); 2 has normal inverse video; 3 will display only inverse blank spaces where inverse video would be; 4 to 7 are the same as 0 to 3, but the text will be printed upside down. (This is not of great value, but fun to play around with.)

764: Last key pressed.
This location stores the internal code of the last key pressed. Using POKE to enter 255 here will clear out any old value before you make a check. "Last" really means last; that is, if you type ABC, the internal code for "C" will be stored. See Programmer's Corner 5 for discussion of the internal character code values (which are not the same as the ATASCII values).

53279: Console switches (OPTION, SELECT, START)
The value in this location indicates which of these keys or what combination of these keys was pressed. See Programmer's Corner 6 for a table of those values.

...The Screen
77: Disabling the "attract" mode.
If there is no input from the keyboard for a period of about nine minutes, the computer will begin to rotate the colors on the screen randomly. This is called the "attract" mode, from the idea that in an arcade this would attract attention. With the personal computer it does this to prevent possible damage to the TV tube. The assumption is made that no keyboard input for this length of time means that no one is using the computer. The problem is that you may be using joysticks or paddles for input and not the keyboard. You can prevent this from happening in your program by using POKE to insert a 0 into this location. This causes the nine-minute timer to

start over again. If you do this in some portion of your program that is accessed regularly, the attract mode will not be activated. If you want to force the computer into this mode, use POKE to insert a value of 255.

82: The left margin.
Normally the left margin is in column 2. You can use POKE to insert a value from 0 to 39 to change this. Keep in mind that the left margin needs to be kept to the left of the right margin.

83: The right margin.
This is normally set at column 39. You can use POKE to insert a value from 0 to 39 here also to put the right margin in that column. These two memory locations (82 and 83) can be useful to display information in varying formats on the screen.

84: Cursor position (row).
The value here will determine the row where the next print to the screen will occur.

85: Cursor position (column).
The value here determines the column where the next print to the screen will occur. Since a value as large as 255 can be stored in one memory location, only GRAPHICS 8, which has 320 possible print positions per line, cannot be handled with a single POKE to this location. Memory location 86 is used with location 85 to handle column values greater than 255. This is not generally needed, since it is usually the text modes where this capability is most useful. In Section 4-2 we used this memory location as a "tab" for printing to selected columns of the screen.

87: Fooling the operating system.
If you use POKE to place a value in this location after selecting a graphics mode to work in, the computer will think that it is in this new mode. You can create interesting effects by, for example, using POKE to insert a 7 after going into GRAPHICS 8, which will cause the top half of the screen to be in GRAPHICS 7 and the bottom half to be in GRAPHICS 8.

88 and 89: Start of screen memory.
These two memory locations store the value of where in computer memory the screen display begins (i.e., the upper left corner of the screen). This information was used in Programmer's Corner 11 to permit printing to selected areas of a screen that had more than one graphics mode displayed.

201: Print tab width.
Normally the tabs are set ten characters apart. You can change this by using POKE to enter other values.

559: Direct memory access.
The display that you see on the screen must be "refreshed" regularly to prevent it from fading away. This takes time away from the computer that could be spent doing other things, such as calculations. If your program doesn't need to use the screen, you can "turn off" the display and speed up other computer work. It is important to first use PEEK to access this location to determine what value is present before using POKE to insert a 0 to turn off the display. You do this so that

you can return the original value (using another POKE) to recover the screen display when it is needed.

752: Cursor off/cursor on.

There are numerous times in a program when the presence of the cursor detracts from the display. You can cause the cursor to become invisible by using POKE to enter any number other than 0 into this memory location. You can later recover the cursor by using POKE with a value of 0.

...PROGRAMMING

18, 19, 20: A built-in clock.

It is easy to have the Atari function as a clock by using these three memory locations. You can use POKE to insert a 0 into these three locations to start the clock and then use

```
PRINT 2^2
```

to measure time in seconds. See Programmer's Corner 6 for more details.

106: Top of memory.

This location contains the number of "pages" that are available for program use, where one page is equal to 256 bytes. Changing this value will reserve a memory area for such things as player-missile graphics. You should issue your GRAPHICS statement after doing this to correctly position the display list.

186 and 187: Where the program stopped.

PEEK (186) + 256*PEEK (187) will give you the line number that your program stopped at, whether it stopped because of an error or your hitting the BREAK key or encountering a TRAP or STOP statement.

195: Error codes.

PEEK(195) will display the error code. You will find memory locations 186, 187, and 195 useful in debugging programs.

623: Player-missile priority.

This memory location is manipulated to establish the priority of players and backgrounds to produce effective animation with player-missile graphics. See Section 11-8 for a use of this and other player-missile memory locations.

624-631: Paddle values.

These eight locations hold the values for paddles 0 to 7. Values range from 0 to 228 for each paddle. PEEKing these locations is equivalent to using the PADDLE(X) function.

632-635: Stick values.

Equivalent to STICK(0) to STICK(3) functions. See Programmer's Corner 4 for the possible values.

636-643: Paddle trigger values.

A 0 means that the trigger was pressed, a 1 means it was not. Identical to the PTRIG (X) function, with location 636 for paddle 0, 637 for paddle 1, and so on.

644-647: Stick trigger values.
Just like STRIG(0) to STRIG(3). Again, a 0 means that the trigger was pressed, a 1 that it was not.

...Graphics

704: Color of Player 0.
Also, in GRAPHICS 10, the background color.

705: Color of Player 1.

706: Color of Player 2.

707: Color of Player 3.

708: Color register 0.
Same as SETCOLOR 0. When you are in GRAPHICS 1 or 2, the color of uppercase letters.

709: Color register 1.
Same as SETCOLOR 1. When you are in GRAPHICS 1 or 2, the color of lowercase letters.

710: Color register 2.
Same as SETCOLOR 2. When you are in GRAPHICS 1 or 2, the color of inverse uppercase letters. Also the background color in GRAPHICS 0 and GRAPHICS 8.

711: Color register 3.
Same as SETCOLOR 3. When you are in GRAPHICS 1 or 2, the color of inverse lowercase letters.

712: Color register 4.
Same as SETCOLOR 4. Used for background and border colors in all graphics modes except GRAPHICS 10.

756: Character base.
A pointer to the start of portions of the character set. The default value is 224 for uppercase characters and numbers. In GRAPHICS 1 or 2 you get the lowercase characters and graphics characters by using POKE to insert a value of 226.

53248-278: Player-missile graphics.
These memory locations contain position, color, and brightness information for all the players and missiles. See Section 11-8 for more information on using these memory locations for animation.

Appendix B

Bugs in Atari BASIC

Any Atari BASIC cartridges manufactured before the third quarter of 1983 have some bugs in them. Most are very minor, but all of them can lead to problems in your programs if you are unaware of their existence. The Atari XL series of computers, with BASIC built into the machines, and all BASIC cartridges manufactured after the date above contain a version of BASIC that corrects these bugs. To determine which version of Atari BASIC you have, simply type

$$X=2 \wedge 2$$

and press the RETURN key. If you get a response of 4 you have a cartridge with the updated Atari BASIC. If you get any value other than 4 (most likely 3.999999996), read on to learn about the bugs and how to control them.

Bug 1. The "lock up" bug. When you do a lot of line editing you will sometimes see the computer all of a sudden fail to respond to any input from the keyboard. This is referred to as *lock up*. The only way you can regain control if this happens is to turn the computer off and back on again, losing any program that was in the computer's memory. This is, needless to say, a very frustrating experience. To guard against this possibility, save any program you are working on at several points along the way to its completion. By doing this, you cut down on the magnitude of the loss if this happens. Some people have claimed that the following sequence will restore your control without losing the program: press SYSTEM RESET, then type BYE (press RETURN), press SYSTEM RESET again, and then type LIST (press RETURN). If you have a system lock up you have nothing to lose by trying this.

Bug 2. The "calculation" bug. This manifests itself as incorrect values for some functions. For values of X from 0 to 1, LOG(X) and CLOG(X) are inaccurate. Also, exponentiation does not return integer values when they are expected. We used the

fact that 2^2 does not come out as 4 to check for which BASIC cartridge you have. The solution for noninteger values when integers are expected is to round off the answer with

```
X=INT (X+0.05)
```

which will force X to be the correct integer.

Bug 3. The "256" bug. BASIC has problems moving strings containing exactly 256 characters or multiples of 256 characters. The solution is to add or subtract one to the length of strings if these lengths are encountered.

Bug 4. The "extra digit" bug. Calculations are shown to nine decimal places, although they are only accurate to eight. This is unlikely to cause any problems.

Bug 5. The "NOT" bug. The statement PRINT A = NOT B will give unpredictable results. Avoid using NOT in print statements.

Bug 6. The "space" bug. A DIMension statement such as

```
DIM A (10)
```

with a space between the "A" and the "(" will result in an error message. Beware of extra spaces in DIM statements.

Appendix C

Error Codes and Their Meaning

As you are well aware, when errors are encountered in executing or typing in a program, your Atari computer reports back with a cryptic error number rather than a meaningful message that describes the problem. Leaving error messages out of Atari BASIC was one of the compromises made in order to make it fit in the 8K (8192 bytes) available. That's the bad news. The good news is that you do get instant error messages as each line of a program is typed. Many personal computers only give error messages when a program is RUN (and STOP when the first error is encountered). This means that you have to RUN the program time and again until all the errors are found and corrected.

Here is a list of the error codes most commonly encountered by the BASIC programmer, with comments on each of them. An uncommented, but complete, list is also available in your Atari BASIC Reference Manual, which came with the computer or the programmer's kit.

2: Memory Insufficient

Your program is too big for the available memory. This commonly comes up during execution of a program when there is a DIM statement that reserves more memory than is available. It also can occur (but is much less likely) if you have too many loops within loops.

3: Value Error

The program encountered a value too big or too small to handle or one that was negative when it should have been positive (for example, with the SQR function, where values must be positive).

4: Too Many Variables

You have 128 variables available in a program. Be aware that variables used once and later erased are counted toward this total. You can clear out old variables by LISTing your program to disk or tape, ENTERing it back in and then SAVEing it.

5: String Length Error

Your program calls for use of a string character value that is beyond the length that you have DIMensioned. This occurs particularly when string calculations incorrectly produce negative values or values greater than the DIM statement permits.

6: Out of DATA Error

Your loop that reads DATA statements ran out of DATA. The best solution is to use a TRAP statement to prevent the program from crashing if this occurs. If this error occurs without a line number, it means that you pressed RETURN when the cursor was on the same line that said READY.

7: Line Numbers Greater than 32767

Remember, line numbers may not exceed this value.

8: INPUT Statement Error

Most likely you INPUT letters, punctuation, or graphics characters when the program was looking for numbers only.

9: Array or String DIMension Error

You may have DIMensioned an array or string too large. You may have forgotten to DIMension an array. Remember particularly that all strings must be DIMensioned before the program encounters them. Alternatively, you may have had your program loop back and encounter the DIMension statement again. It is best to put all DIMension statements at the start of the program and take care not to go back to this area of the program during execution.

11: Numeric Overflow

Either division by zero occurred or a calculated value exceeded $1*10^{98}$.

12: Line Not Found

An IF . . . THEN, GOSUB, GOTO, ON . . . GOSUB, or ON . . . GOTO statement tried to transfer execution to a line number that was not in your program. This most commonly occurs when you make changes in a program and forget that you may need those line numbers. A fairly safe, but memory wasting method to prevent this is to replace any lines deleted with REM statements to keep the line number. Better yet is being careful in structuring your program.

13: NEXT Without FOR

Your program encountered a NEXT statement without having seen a FOR. Check for missing FOR statements or messed-up program structure.

16: RETURN Without GOSUB

A RETURN statement was reached without the required GOSUB statement. This can occur when subroutines are placed at the end of a program, but the program does not have an END statement. In this case the program may run past its normal

end and reach the subroutines without a GOSUB statement.

21: LOAD File Error

You tried to load either a program that was not a program (i.e., data for a program) or one that had been LISTed to disk or tape rather than SAVEd. Try ENTER (but remember to precede it with NEW unless you want to mix up what is currently in memory with the new program being ENTERed).

128: BREAK Abort

The BREAK key was used to halt the execution of a program.

129: IOCB Already Open

This is commonly encountered when you try to OPEN a channel that is already open. Having CLOSE statements in the correct places takes care of this. The best solution is to always precede an OPEN statement with a CLOSE statement for the same channel.

138: Device Timeout

Most likely you forgot to turn on the peripheral device that it being addressed, or else it is not on-line. Be especially aware of this error if you have a printer.

141: Cursor out of Range

You have attempted to code a PLOT or a DRAWTO statement using a point that does not exist in the graphics mode that you are using. This can happen if a value of less than zero or greater than the screen size is encountered. Be sensitive to the screen limits and remember that a screen that has 96 points available for graphics numbers these points from 0 to 95.

143: Serial Bus Data Frame Checksum

Most commonly encountered in loading tape programs. Try to load the program again. If the error occurs more than rarely, there is probably something wrong with your cassette recorder.

147: Insufficient RAM

Between the size of your program and the graphics mode you are working in, you have exceeded the available memory. Similar to Error 2.

162: Disk Full

You have filled the available disk space. Beware that the program will be on the disk; that is, the name will be there and as much of the program as could be fit on the disk. However, you will not be able to load the program. Immediately save the program on another disk, go to DOS, and erase the partial program to avoid later confusion.

164: File Number Mismatch

Most likely the disk file somehow got scrambled and the computer can no longer determine where to go on the disk to get the pieces of the program. Files on the disk are not saved in sequential order; instead the computer keeps track of where the parts are and the order they should have. This can sometimes get messed up; if it does, there is little the average user can do to fix it. The lesson here is to have backup copies of important programs to prevent this (or any other possible disk or tape error) from destroying your work.

169: Directory Full

You can put 64 files on a disk before this error message is encountered. This may still

leave lots of room on the disk, but no more files can be saved.

170: File Not Found
Either the file you are looking for is not on that disk or you misspelled the name. It is a good practice to have a label on each disk jacket containing the names of all the programs on that disk.

Appendix D

An Annotated Reading List

Where do you go from here? We are now at the end of our look at programming in Atari BASIC. We have gone from a program that uses nothing but a couple of PRINT statements for displaying a message on a TV screen to many-colored animations and complex data management programs. Along the way we have tried to show how to develop good BASIC programs. We have also explained the use of the features available through Atari BASIC. The information and programs presented in this book can serve as a valuable foundation for learning.

There are many applications at home, in school, and on the job where tasks can be made easier, more manageable, or just more fun by applying your Atari computer. The more you use it, the easier its use becomes. We hope that much of the mystery that shrouds computers has faded and been replaced with knowledge, enthusiasm, and excitement.

This book may well be all you'll need to understand your Atari and develop programs to use it. Or you may want to go further. There is much more that can be done, including:

1. Using machine language, which permits much faster program execution. This permits arcade-type action and very rapid data manipulation.
2. Learning more display techniques, so that you can display even more colors and shades at one time and show more realistic motion of many objects simultaneously.
3. Learning more applications of the Atari computer. We have just scratched the surface with the programs and problems presented here.

There are many commercially available programs that can be of use to you. In general, they are quite sophisticated in their programming and meet a specific

need. They rarely are useful for learning new programming methods. The programs are usually modified to protect the program coding from being seen by the user, which means you can't learn from the techniques that they use. Thus they are not valuable as learning tools. They do, however, provide some very powerful applications. You can, for example, use several of these programs to design complex graphics screens, virtually without knowing anything about how the display works. You can also perform very complex data base manipulations and formatting. We believe that you will always be better off if you have at least some fundamental understanding of how the process works before you apply these commercial programs. That, after all, is what this book is all about.

To help with your continued development, we'd like to present an annotated bibliography of books and magazines that may be of help to you.

...Magazines

ANALOG
565 Main Street
Cherry Valley, MA

An Atari only magazine. Good game programs and many useful articles.

Antic
600 18th Street
San Francisco, CA

An Atari-only magazine. Excellent for application programs.

COMPUTE!
P.O. Box 6502
Greensboro, NC

Has coverage of several computers in addition to some fine Atari articles. Good for brief application programs.

Creative Computing
39 E. Hanover Avenue
Morris Plains, NJ 07950

Not a lot of Atari coverage, but what there is is good.

Hi-Res
933 Lee Road, Suite 325
Orlando, FL

Atari computer and game machines. Good for beginners.

SoftSide
10 Northern Blvd.
Amherst, NH

Covers several computers. Has mainly BASIC game programs, which can be useful for learning programming techniques.

...Books
Your Atari Computer
Osborne/McGraw-Hill
A very thorough guide to Atari BASIC and the Atari peripherals. The most thorough presentation of the BASIC commands. A definite must for your Atari reference library.

First Book of Atari
Second Book of Atari
Third Book of Atari
First Book of Atari Graphics
Mapping the Atari
COMPUTE Books
A good assortment of books for the Atari. The *First Book of Atari* and the *Third Book of Atari* contain articles that appeared in issues of *COMPUTE* magazine. The *Second Book of Atari* contains all unpublished material and the *First Book of Atari Graphics* is a mixture of old and new material. All have articles and tips that are very useful for the BASIC programmer. *Mapping the Atari* is the best source of memory location information that exists. It also contains numerous short programs to demonstrate what you can do by using specific memory locations.

Inside Atari BASIC
Designs from Your Mind with Atari Graphics
Reston Publishing Company, Inc.
Inside Atari BASIC is a very light approach to BASIC. Good for the beginner and only for the beginner. *Designs from Your Mind with Atari Graphics* contains a nice, understandable presentation of hands-on usage of many of the graphic techniques discussed in this book.

Atari Programming with 55 Programs
Tab Books, Inc.
Teaches BASIC programming through a line-by-line discussion of 55 programs.

Atari BASIC
Prentice-Hall, Inc.
An informative presentation on Atari BASIC, emphasizing graphics.

I Speak BASIC to My Atari
Hayden Book Company, Inc.
A "viewgraph" presentation of the basics of Atari BASIC. Good as an introduction.

Appendix E

ATASCII Character Code

Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes
0	♥	CTRL ,	13		CTRL M	26		CTRL Z
1		CTRL A	14		CTRL N	27		ESC/ESC
2		CTRL B	15		CTRL O	28		ESC/CTRL -
3		CTRL C	16		CTRL P	29		ESC/CTRL =
4		CTRL D	17		CTRL Q	30		ESC/CTRL +
5		CTRL E	18		CTRL R	31		ESC/CTRL *
6		CTRL F	19		CTRL S	32		SPACE BAR
7		CTRL G	20		CTRL T	33	!	SHIFT 1
8		CTRL H	21		CTRL U	34	"	SHIFT 2
9		CTRL I	22		CTRL V	35	#	SHIFT 3
10		CTRL J	23		CTRL W	36	$	SHIFT 4
11		CTRL K	24		CTRL X	37	%	SHIFT 5
12		CTRL L	25		CTRL Y	38	&	SHIFT 6

Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes
39		SHIFT 7	61		=	83		S
40		SHIFT 9	62	>	>	84		T
41		SHIFT 0	63		SHIFT /	85		U
42		•	64		SHIFT 8	86		V
43		+	65	A	A	87		W
44		,	66	B	B	88	X	X
45		-	67	C	C	89	Y	Y
46		.	68	D	D	90	Z	Z
47	/	/	69	E	E	91		SHIFT ;
48	0	0	70	F	F	92	\	SHIFT ,
49	1	1	71	G	G	93		SHIFT +
50	2	2	72	H	H	94	^	SHIFT •
51	3	3	73	I	I	95		SHIFT -
52	4	4	74	J	J	96		CTRL
53	5	5	75	K	K	97		(LOWR) A
54	6	6	76	L	L	98	b	(LOWR) B
55	7	7	77	M	M	99	c	(LOWR) C
56	8	8	78	N	N	100	d	(LOWR) D
57	9	9	79	O	O	101	e	(LOWR) E
58		SHIFT ;	80	P	P	102	f	(LOWR) F
59		;	81	Q	Q	103	g	(LOWR) G
60	<	<	82	R	R	104	h	(LOWR) H

Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes	Decimal Code	ATASCII Character	Keystrokes
105	i	(LOWR) I	113	q	(LOWR) Q	121	y	(LOWR) Y
106	j	(LOWR) J	114	r	(LOWR) R	122	z	(LOWR) Z
107	k	(LOWR) K	115	s	(LOWR) S	123		CTRL ;
108	l	(LOWR) L	116	t	(LOWR) T	124		SHIFT =
109	M	(LOWR) M	117	u	(LOWR) U	125		ESC\CTRL < or ESC\SHIFT <
110	n	(LOWR) N	118	v	(LOWR) V			
111	o	(LOWR) O	119	w	(LOWR) W	126		ESC\BACK S
112	p	(LOWR) P	120	x	(LOWR) X	127		ESC\TAB

Appendix F

Specific Features of the XL Computers

There are a number of specific features that have been incorporated into the XL series of Atari computers. In this appendix, we will look at some of these features and give examples of how you can make use of them. Note that these features cannot be accessed with the Atari 400 or 800 computers.

...The HELP Key

The HELP key provides built-in tests of the memory, keyboard, and sound systems of the computer. You can use this key in your own programs by first checking on the contents of memory location 732. If it contains a 0, then the HELP key has not been pressed. If it contains the following values, the HELP key has been pressed.

VALUE	KEY(S) PRESSED
17	HELP
81	SHIFT + HELP
145	CTRL + HELP

What can you do with this key? Well, since it is labeled HELP anyway, you might use it to "help" your program. For example, in an educational program you could HELP the user by providing additional examples on request. In an adventure game, you could provide some clues, etc.

...More XL Features

Later in this appendix, we'll present a program that demonstrates some of the features built into the XL computers. First, however, let's discuss the features themselves.

...Fine Scrolling
The LISTing of a program occurs much too fast to read. We already know how to

use the CTRL key and 1 to start and stop the listing, but it would be helpful to be able to slow down the listing so that it is more readable. Memory location 622 controls this. A value of 0 here means normal, fast scrolling and a value greater than zero enables slow scrolling.

...Key Click
With the XL computers, the key click—the sound you hear when you press a key—comes through the television speaker. You can remove the sound by merely turning down the television sound. A more elegant way in which to remove the sound and retain the programmable option of turning it on or off is to use memory location 731. A value of 0 here means the click is on, and any value greater than zero turns the key click off.

...Key Repeat and Key Interval Delays
When you hold down any key on your XL keyboard, there will be a delay, and then the key will repeat and continue to repeat as long as you hold the key down. There are two delays involved—a delay between the initial pressing of the key and the first repeat and another delay between all subsequent repeats. The default values are 0.8 second before a key is repeated and 0.1 second between additional repeats. The first delay guards against getting multiple letters printed on the screen when you only wanted one. Memory location 729 controls the first delay and the default value in this location is 48. Since the values of these two locations are measured in sixtieths of a second, this means 48/60 or 0.8 second delay. Similarly, the default value in memory location 730 is 6 (6/60 or 0.1 second). However, 0.8 second is a rather long wait when you want to have a repeating key, as you do when, for example, you are moving the cursor around the screen with CTRL plus the arrow keys. You can control these delays by controlling the values that are in memory locations 729 and 730.

We'll write a program that will allow you to experiment with all those memory locations. The program will be a "set-up" program that will permit you to vary all of these values and then exit the program and try out the result. The program will present a menu and then, depending on the choices of the user, will turn the key click on or off, enable or disable the fine scrolling, and increase or decrease the key repeat and key interval delays.

Program F-1 demonstrates all of these features.

```
90 REM * ATARI XL FEATURE DEMONSTRATION
95 POKE 622,0:POKE 729,48:POKE 730,6:POKE 731,0
100 REM * MAIN MENU
105 OPEN #1,4,0,"K:"
110 GRAPHICS 0
115 POKE 752,1:POKE 84,0
120 POKE 85,12:PRINT " XL DEMONSTRATION "
130 PRINT
140 IF FLAG1=0 THEN PRINT " (1) KEY CLICK NOW ON "
145 IF FLAG1=1 THEN PRINT " (1) KEY CLICK NOW OFF"
```

```
150 PRINT
155 REPEAT=PEEK(729)/60
160 PRINT " (2) FASTER REPEAT: ";REPEAT;" SEC.         "
170 PRINT
180 PRINT " (3) SLOWER REPEAT:"
190 PRINT
195 DELAY=INT(PEEK(730)/60*100)/100
200 PRINT " (4) LESS DELAY: ";DELAY;" SEC.          "
210 PRINT
220 PRINT " (5) MORE DELAY:"
230 PRINT
239 REM * CHECK THE KEY PRESSED AND RESPOND
240 IF FLAG2=0 THEN PRINT " (6) SLOW SCROLLING OFF"
245 IF FLAG2=1 THEN PRINT " (6) SLOW SCROLLING ON "
250 PRINT
260 PRINT " (7) EXIT PROGRAM AND TRY OUT"
270 GET #1,SELECTION:X=SELECTION-48
300 IF X=1 AND FLAG1=0 THEN FLAG1=1:POKE 731,255:
GOTO 340
310 IF X=1 AND FLAG1=1 THEN FLAG1=0:POKE 731,0
320 IF X=2 AND PEEK(729)>6 THEN POKE 729,PEEK(729)-6
322 IF X=3 THEN POKE 729,PEEK(729)+6
325 IF X=4 AND PEEK(730)>1 THEN POKE 730,PEEK(730)-1
327 IF X=5 THEN POKE 730,PEEK(730)+1
330 IF X=6 AND FLAG2=0 THEN FLAG2=1:POKE 622,1:
GOTO 340
335 IF X=6 AND FLAG2=1 THEN FLAG2=0:POKE 622,0
337 IF X=7 THEN 400
340 GOTO 115
400 PRINT :PRINT "TRY LISTING THIS PROGRAM AND ALSO"
410 PRINT "TRY PRESSING AND HOLDING KEYS TO"
420 PRINT "TEST OUT THE CHANGES YOU HAVE MADE."
430 POKE 752,0
```

Program F-1. Set-up program for XL computer features.

In line 115 we use POKE 84,0 to position the cursor on the top line of the screen. This is important because as we make various selections from the screen, we will want to be sure to overwrite the correct lines and indicate our choices. The cursor will serve as a reference because it always starts at the top of the screen. In lines 160 and 200 we print the new values for the repeat and delay values, taking care to print some extra spaces after the text to ensure that we will erase what appeared previously on the screen. This takes care of the instance when we need to print one digit in a place where previously there were three. In line 105 we OPEN a channel for keyboard input and then, in line 270, we GET one character from the keyboard. We subtract 48 in order to convert to the actual number selected. Then, lines 300 to 337 select the action to be taken depending on the number chosen. Line 340 sends the program back to update the screen and await more input. When we exit the program we get the message in lines 400 to 420 and the cursor is turned back on in line 430. You are then free to try your modified keyboard.

Appendix G
Index of Programs

Solution Programs for Even-Numbered Problems

Each two-page spread should be read from top to bottom as one individual page.

Chapter 1

Problem No. 2

```
10  INPUT A,B,C,D,E
20  PRINT A+B+C+D+E


?124.3,657,801.45,-9,81
1654.75

READY
```

Problem No. 4

```
10  LET X=1/3
20  PRINT "     X = ";X
30  PRINT "   X*3 = ";X*3
```

Problem No. 10

```
10   REM * DIAGONAL MESSAGE
20   PRINT "H"
30   PRINT " A"
40   PRINT "  P"
50   PRINT "   P"
60   PRINT "    Y"
65   PRINT "     "
70   PRINT "      B"
80   PRINT "       I"
90   PRINT "        R"
100  PRINT "         T"
110  PRINT "          H"
120  PRINT "           D"
130  PRINT "            A"
140  PRINT "             Y"
150  END
```

```
40 PRINT "X+X+X = ";X+X+X
```

```
RUN
    X  =  0.3333333333
  X*3  =  0.9999999999
X+X+X  =  0.9999999999
```

READY

Problem No. 6

```
10  NUM=1/2+1/3
20  DENOM=1/3-1/4
30  PRINT NUM/DENOM
```

```
RUN
10
```

READY

Problem No. 8

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

```
RUN
3628800
```

READY

```
RUN
H
 A
  P
   P
    Y

     B
      I
       R
        T
         H
          D
           A
            Y
```

READY

Chapter 2

Section 2-1

Problem No. 2

```
100 REM * CHECK AVERAGE PACKAGE
WEIGHT FOR 180 GRAMS
200  TOTAL=0
210  COUNTER=1
220  PRINT "WEIGHT ";COUNTER;
230  INPUT WEIGHT
232  IF WEIGHT=0 THEN GOTO 999
235  TOTAL=TOTAL+WEIGHT
240  COUNTER=COUNTER+1
250  IF COUNTER<=5 THEN GOTO 220
```

Section 2-1
Problem No. 2
(continued)

```
260 AVERAGE=TOTAL/5
270 IF AVERAGE<180 THEN GOTO 290
275 PRINT "ACCEPT THIS LOT"
280 GOTO 295
290 PRINT "REJECT THIS LOT"
295 PRINT
297 PRINT
300 GOTO 200
999 END
```

```
RUN
WEIGHT 1?179
WEIGHT 2?182
WEIGHT 3?181
WEIGHT 4?180
WEIGHT 5?179
ACCEPT THIS LOT


WEIGHT 1?0

READY
```

Problem No. 4

```
100 READ A,B,C,D
110 PRINT " SCORES: ";A;" ";B;"
";C;" ";D
120 PRINT "AVERAGE: ";(A+B+C+D)/4
900 DATA 100,86,71,92
```

```
150 DAY=1
160 WAGES=0.01
170 TOTAL=0
198 REM
200 TOTAL=TOTAL+WAGES
210 LASTWAGE=WAGES
220 WAGES=WAGES*2
230 DAY=DAY+1
290 IF DAY<=30 THEN GOTO 200
300 PRINT "$ ";LASTWAGE;"   $ ";TOTAL
```

```
RUN
$ 5368709.12   $ 10737418.23

READY
```

Problem No. 10

```
100 REM * CALCULATE THE AMOUNT OF AN
ORDER
200 BOOKS=4*10.95*(1-0.25)
220 RECORDS=3*4.98*(1-0.15)
240 PLAYER=59.95
290 TOTAL=BOOKS+RECORDS+PLAYER
300 BILL=TOTAL*(1-0.02)
400 PRINT "AMOUNT $ ";BILL
```

```
RUN
AMOUNT $ 103.38902

READY
```

```
RUN
  SCORES:  100 86 71 92
AVERAGE:  87.25

READY
```

Problem No. 6

```
100 REM * COUNT AND SUM INTEGERS FROM
1001 TO 2213 DIVISIBLE BY ELEVEN
150 COUNTER=0
160 NUMBER=1001
170 SUM=0
210 COUNTER=COUNTER+1
220 SUM=SUM+NUMBER
280 NUMBER=NUMBER+11
290 IF NUMBER<=2213 THEN GOTO 210
310 PRINT COUNTER;"    NUMBERS"
320 PRINT SUM;" SUM"
```

```
RUN
111     NUMBERS
178266 SUM

READY
```

Problem No. 8

```
100 REM * DOUBLE WAGES EACH DAY
FOR 30 DAYS
```

Problem No. 12

```
90 REM * 12 DAYS OF CHRISTMAS-
RUNNING TOTAL OF GIFTS
150 GIFTS=0
160 DAY=1
170 PRINT "DAY","GIFTS SO FAR"
200 DAYGIFT=0
210 DAYGIFT=DAYGIFT+1
220 GIFTS=GIFTS+DAYGIFT
230 IF DAYGIFT=DAY THEN GOTO 300
240 GOTO 210
300 PRINT DAYGIFT,GIFTS
305 DAY=DAY+1
310 IF DAY<=12 THEN GOTO 200
400 PRINT "TOTAL NUMBER OF GIFTS IS:
;GIFTS
```

```
RUN
DAY          GIFTS SO FAR
1            1
2            4
3            10
4            20
5            35
6            56
7            84
8            120
9            165
10           220
11           286
12           364
TOTAL NUMBER OF GIFTS IS: 364
READY
```

Section 2-2

Problem No. 2

```
198 REM * FLIP A COIN 38 TIMES
199 TAILS=0
200 FLIPS=1
230 IF RND(0)<0.5 THEN 270
250 PRINT "T";
255 TAILS=TAILS+1
260 GOTO 280
270 PRINT "H";
280 FLIPS=FLIPS+1
290 IF FLIPS<=38 THEN 230
295 PRINT
300 PRINT TAILS;" TAILS"
999 END


RUN
HHHTTTHHHTHHTTTHTHTHTHHHHHTHHTTT
HHHTHH
15 TAILS

READY
```

Problem No. 4

```
90 REM * ROLLING TWO DICE TEN TIMES
150 COUNTER=1
160 PRINT "DICE#1","DICE#2"
200 DICE1=INT(RND(0)*6+1)
210 DICE2=INT(RND(0)*6+1)
220 PRINT DICE1,DICE2
```

```
RUN
ODD INTEGERS FROM 5 TO 1191 = 594

READY
```

Problem No. 4

```
90 REM * CALCULATE WAGES FOR DOUBLING
EACH DAY FOR 30 DAYS
180 WAGES=0.01
190 TOTAL=0.01
200 FOR DAY=2 TO 30
220 WAGES=WAGES*2
230 TOTAL=TOTAL+WAGES
290 NEXT DAY
300 PRINT "$ ";WAGES;" WAGE ON 30TH DAY"
310 PRINT "$ ";TOTAL;" TOTAL ACCUMULATED"


RUN
$ 5368709.12 WAGE ON 30TH DAY
$ 10737418.23 TOTAL ACCUMULATED

READY
```

Problem No. 6

```
90 REM * 12 DAYS OF CHRISTMAS TOTAL
150 GIFTS=0
160 FOR DAY=1 TO 12
190 DAYGIFTS=0
200 FOR COUNT=1 TO DAY
```

290.

```
250 COUNTER=COUNTER+1
290 IF COUNTER<=10 THEN 200
999 END


RUN
DICE#1      DICE#2
5           2
3           1
5           1
3           4
5           2
2           2
5           2
6           4
4           2
3           6

READY
```

Section 2-3

Problem No. 2

```
90 REM * COUNT ODD INTEGERS FROM 5
TO 1191
190 COUNTER=0
200 FOR INTEGER=5 TO 1191 STEP 2
210 COUNTER=COUNTER+1
290 NEXT INTEGER
300 PRINT "ODD INTEGERS FROM 5 TO
1191 = ";COUNTER
```

```
220 DAYGIFTS=DAYGIFTS+COUNT
230 NEXT COUNT
240 GIFTS=GIFTS+DAYGIFTS
250 NEXT DAY
400 PRINT "Total Number of Gifts is: "
;GIFTS


RUN
Total Number of Gifts is: 364

READY
```

Problem No. 8

```
198 REM * FLIP A COIN 30 TIMES - DO FIVE
SERIES
200 FLIPS=1
210 FOR TURN=1 TO 5
220 FLIPS=1
230 IF RND(0)<0.5 THEN GOTO 270
250 PRINT "T";
260 GOTO 280
270 PRINT "H";
280 FLIPS=FLIPS+1
290 IF FLIPS<=30 THEN GOTO 230
295 PRINT
296 PRINT
300 NEXT TURN
999 END

RUN
```

```
HHHHTTTTTTHHTHTTHTHTHTHHTHTHHHHTHHHHTTTH

HHTTHHHTHHTHTHTHHTTHTHTHTTHTHTHHTTTTHTHT

THTHTTHHHHHTHHHHHHTHTTHTHHHHTHHHHTTTHT

HHHTTHTTTHHTHTTTHTTTTHHTHHHHHHHTHTHHHHT

TTTHHHTHHTHTTHHHTTTTTHHHHHHHTTHTHHTHTH
```

READY

```
90 REM * USING A FOR...NEXT
95 REM * FLIP A COIN 1000 TIMES
150 TAILS=0
200 FOR FLIP=1 TO 1000
230 IF RND(0)<0.5 THEN 270
250 REM
255 TAILS=TAILS+1
260 GOTO 280
270 REM
280 NEXT FLIP
295 PRINT
300 PRINT TAILS;" TAILS IN 1000 FLIPS"
999 END

RUN
```

```
500 PRINT
510 PRINT "YOU GOT ";RIGHT;" CORRECT
OUT OF ";NUMBER
999 END


RUN
Let's do some ADDITION drill

How many problems do you want?5
Enter range of numbers desired (low
first)?10,41


28 + 11 = ?38
NO, that would be 39

10 + 19 = ?29
RIGHT!

18 + 22 = ?40
RIGHT!

30 + 29 = ?39
NO, that would be 59

14 + 33 = ?47
RIGHT!

YOU GOT 3 CORRECT OUT OF 5

READY
```

READY

Problem No. 12

```
90 REM * ADDITION DRILL
200 PRINT "Let's do some ADDITION
drill"
205 PRINT
210 PRINT "How many problems do you
want";
220 INPUT NUMBER
250 PRINT
260 PRINT "Enter range of numbers
desired (low    first)";
270 INPUT LO,HI
290 RIGHT=0
295 PRINT
298 REM * DRILL BEGINS HERE
300 FOR PROBLEM=1 TO NUMBER
310 N1=INT(RND(0)*(HI-LO+1)+LO)
320 N2=INT(RND(0)*(HI-LO+1)+LO)
330 SUM=N1+N2
340 PRINT
350 PRINT N1;" + ";N2;" = ";
360 INPUT ANSWER
400 IF ANSWER=SUM THEN 450
410 PRINT "NO, that would be ";SUM
420 GOTO 480
450 PRINT "RIGHT!"
460 RIGHT=RIGHT+1
480 NEXT PROBLEM
```

Chapter 3

Section 3-1

Problem No. 2

```
98 REM * TWO DICE: A "1" AND A "3"
100 GRAPHICS 3
110 COLOR 1
120 FOR I=1 TO 7
130 PLOT 1,I
135 DRAWTO 7,I
140 PLOT 1+10,I
145 DRAWTO 7+10,I
150 NEXT I
160 COLOR 3
170 PLOT 4,4
175 PLOT 2+10,2
180 PLOT 4+10,4
186 PLOT 6+10,6
190 END
```


Problem No. 4

```
90 REM * BARGRAPH
100 GRAPHICS 3
110 COLOR 1
115 FOR I=1 TO 9
120 READ A
130 PLOT I*2,20
140 DRAWTO I*2,20-A
150 NEXT I
160 PRINT "1 2 3 4 5 6 7 8 9 WEEKS"
180 PRINT "DOLLAR SALES IN
```

Section 3-1

Problem No. 4

(continued)

```
THOUSANDS";
190 END
300 DATA 17,12,15
305 DATA 6,12,18
310 DATA 15,14,15
```

Section 3-2

Problem No. 2

```
98 REM * DISPLAY A RANDOM DIE IN THE
UPPER LEFT CORNER
100 GRAPHICS 3
110 COLOR 1
120 X=0
130 Y=0
140 GOSUB 1000
150 COLOR 3
200 R=INT(RND(0)*6+1)
910 IF R=1 THEN GOSUB 1100
920 IF R=2 THEN GOSUB 1200
930 IF R=3 THEN GOSUB 1300
940 IF R=4 THEN GOSUB 1400
950 IF R=5 THEN GOSUB 1500
960 IF R=6 THEN GOSUB 1600
990 END
998 REM * DISPLAY THE DIE BACKGROUND
1000 FOR I=1 TO 7
1010 PLOT X+1,Y+I
1020 DRAWTO X+7,Y+I
1030 NEXT I
```

294

Problem No. 4

```
98 REM * DISPLAY TWO RANDOM DICE IN
THE LOWER LEFT CORNER
100 GRAPHICS 3
108 REM * FIRST DIE
110 COLOR 1
120 X=0
125 Y=12
130 GOSUB 1000
135 COLOR 3
140 R=INT(RND(0)*6+1)
150 GOSUB 910
208 REM * SECOND DIE
210 COLOR 1
220 X=10
230 GOSUB 1000
235 COLOR 3
240 R=INT(RND(0)*6+1)
250 GOSUB 910
900 END
910 IF R=1 THEN GOSUB 1100
920 IF R=2 THEN GOSUB 1200
930 IF R=3 THEN GOSUB 1300
940 IF R=4 THEN GOSUB 1400
950 IF R=5 THEN GOSUB 1500
960 IF R=6 THEN GOSUB 1600
990 RETURN
998 REM * DISPLAY THE DIE BACKGROUND
1000 FOR I=1 TO 7
1010 PLOT X+1,Y+I
1020 DRAWTO X+7,Y+I
1030 NEXT I
1090 RETURN
```

```
1090 RETURN
1098 REM * PLOT A 'ONE'
1100 PLOT X+4,Y+4
1190 RETURN
1198 REM * PLOT A 'TWO'
1200 PLOT X+2,Y+2
1210 PLOT X+6,Y+6
1290 RETURN
1298 REM * PLOT A 'THREE'
1300 PLOT X+2,Y+2
1310 PLOT X+4,Y+4
1320 PLOT X+6,Y+6
1390 RETURN
1398 REM * PLOT A 'FOUR'
1400 PLOT X+2,Y+2
1410 PLOT X+6,Y+6
1420 PLOT X+2,Y+6
1430 PLOT X+6,Y+2
1490 RETURN
1498 REM * PLOT A 'FIVE'
1500 PLOT X+2,Y+2
1510 PLOT X+6,Y+2
1520 PLOT X+2,Y+6
1530 PLOT X+6,Y+6
1540 PLOT X+4,Y+4
1590 RETURN
1598 REM * PLOT A 'SIX'
1600 PLOT X+2,Y+2
1610 PLOT X+6,Y+2
1620 PLOT X+2,Y+6
1630 PLOT X+6,Y+6
1640 PLOT X+2,Y+4
1650 PLOT X+6,Y+4
1690 RETURN
```

```
1098 REM * PLOT A 'ONE'
1100 PLOT X+4,Y+4
1190 RETURN
1198 REM * PLOT A 'TWO'
1200 PLOT X+2,Y+2
1210 PLOT X+6,Y+6
1290 RETURN
1298 REM * PLOT A 'THREE'
1300 PLOT X+2,Y+2
1310 PLOT X+4,Y+4
1320 PLOT X+6,Y+6
1390 RETURN
1398 REM * PLOT A 'FOUR'
1400 PLOT X+2,Y+2
1410 PLOT X+6,Y+6
1420 PLOT X+2,Y+6
1430 PLOT X+6,Y+2
1490 RETURN
1498 REM * PLOT A 'FIVE'
1500 PLOT X+2,Y+2
1510 PLOT X+6,Y+2
1520 PLOT X+2,Y+6
1530 PLOT X+6,Y+6
1540 PLOT X+4,Y+4
1590 RETURN
1598 REM * PLOT A 'SIX'
1600 PLOT X+2,Y+2
1610 PLOT X+6,Y+2
1620 PLOT X+2,Y+6
1630 PLOT X+6,Y+6
1640 PLOT X+2,Y+4
1650 PLOT X+6,Y+4
1690 RETURN
```

Problem No. 6

```
98 REM * SIMULATING ROLLING DICE
100 GRAPHICS 3
110 FOR D=1 TO 20
120 X=INT(RND(0)*32+1)
125 Y=INT(RND(0)*12+1)
130 R=INT(RND(0)*6+1)
140 COLOR 1
150 GOSUB 1000
160 COLOR 3
170 GOSUB 910
175 FOR DELAY=1 TO 100
180 NEXT DELAY
185 COLOR 0
190 GOSUB 1000
200 NEXT D
300 R=INT(RND(0)*6+1)
310 X=0
320 Y=12
330 COLOR 1
340 GOSUB 1000
350 COLOR 3
360 GOSUB 910
400 R=INT(RND(0)*6+1)
410 X=15
430 COLOR 1
440 GOSUB 1000
450 COLOR 3
460 GOSUB 910
900 END
910 IF R=1 THEN GOSUB 1100
920 IF R=2 THEN GOSUB 1200
930 IF R=3 THEN GOSUB 1300
```

```
1540 PLOT X+4,Y+4
1590 RETURN
1598 REM * PLOT A 'SIX'
1600 PLOT X+2,Y+2
1610 PLOT X+6,Y+2
1620 PLOT X+2,Y+6
1630 PLOT X+6,Y+6
1640 PLOT X+2,Y+4
1650 PLOT X+6,Y+4
1690 RETURN
```

Problem No. 8

```
98 REM * CONTROL DIE DISPLAY WITH
VARYING INTENSITY
100 GRAPHICS 3
110 COLOR 1
120 GOSUB 1000
200 FOR BRITE=0 TO 14 STEP 2
210 SETCOLOR 0,2,BRITE
220 FOR DELAY=1 TO 500
230 NEXT DELAY
240 NEXT BRITE
990 END
998 REM * DISPLAY A "1" DIE
1000 FOR I=0 TO 6
1010 PLOT 0,I:DRAWTO 6,I
1020 NEXT I
1030 COLOR 0
1040 PLOT 3,3
1050 RETURN
```

```
940 IF R=4 THEN GOSUB 1400
950 IF R=5 THEN GOSUB 1500
960 IF R=6 THEN GOSUB 1600
990 RETURN
998 REM * DISPLAY THE DIE BACKGROUND
1000 FOR I=1 TO 7
1010 PLOT X+1,Y+I
1020 DRAWTO X+7,Y+I
1030 NEXT I
1090 RETURN
1098 REM * PLOT A 'ONE'
1100 PLOT X+4,Y+4
1190 RETURN
1198 REM * PLOT A 'TWO'
1200 PLOT X+2,Y+2
1210 PLOT X+6,Y+6
1290 RETURN
1298 REM * PLOT A 'THREE'
1300 PLOT X+2,Y+2
1310 PLOT X+4,Y+4
1320 PLOT X+6,Y+6
1390 RETURN
1398 REM * PLOT A 'FOUR'
1400 PLOT X+2,Y+2
1410 PLOT X+6,Y+6
1420 PLOT X+2,Y+6
1430 PLOT X+6,Y+2
1490 RETURN
1498 REM * PLOT A 'FIVE'
1500 PLOT X+2,Y+2
1510 PLOT X+6,Y+2
1520 PLOT X+2,Y+6
1530 PLOT X+6,Y+6
```

Chapter 4

Section 4-1

Problem No. 2

```
90 REM * COMPARE 360 DAYS WITH 365
FOR COMPOUND INTEREST
100 PRINCIPAL=10000
150 PRINT "DAYS","RATE % ","AMOUNT"
200 READ DAYS,INTEREST
210 IF DAYS=0 THEN END
220 DAYINT=(INTEREST/100)/DAYS
240 AMOUNT=PRINCIPAL*(1+DAYINT)^DAYS
260 PRINT DAYS,INTEREST,AMOUNT
290 GOTO 200
900 DATA 360,5.5,360,12.5
910 DATA 365,5.5,360,12.5
990 DATA 0,0
```

```
RUN
```

DAYS	RATE %	AMOUNT
360	5.5	10565.3702
360	12.5	11331.2703
365	5.5	10565.3819
360	12.5	11331.2703

```
READY
```

Problem No. 4

```
100 REM * CALCULATE DAILY COMPOUNDED
INTEREST
```

Section 4-1
Problem No. 4
(continued)

```
110 PRINT "INPUT PRINCIPAL ($)"
120 INPUT PRINCIPAL
130 PRINT "INPUT INTEREST RATE (%
PER YEAR)"
140 INPUT INTRATE
145 RATE=INTRATE/100
150 PRINT "INPUT INTEREST PERIODS
PER YEAR"
160 INPUT PERIODS
170 INTEREST=RATE/PERIODS
200 AMOUNT=PRINCIPAL*(1+INTEREST)
^PERIODS
205 PRINT
210 PRINT AMOUNT
220 PRINT
230 GOTO 110


RUN
INPUT PRINCIPAL ($)
?10000
INPUT INTEREST RATE (% PER YEAR)
?15
INPUT INTEREST PERIODS PER YEAR
?12

11607.5465

INPUT PRINCIPAL ($)
?10000.
INPUT INTEREST RATE (% PER YEAR)
```

```
300 PRINT
310 PRINT "ALPHABETICALLY FIRST:";E$
890 END
900 DATA PENNSYLVANIA,NEW JERSEY
910 DATA CALIFORNIA,TEXAS
920 DATA VIRGINIA,FLORIDA
930 DATA STOP


RUN
PENNSYLVANIA
NEW JERSEY
CALIFORNIA
TEXAS
VIRGINIA
FLORIDA

ALPHABETICALLY FIRST:CALIFORNIA

READY
```

Problem No. 4

```
90 REM * YES/NO SUBROUTINE
95 DIM ANSWER$(3)
100 PRINT "QUESTION ";
110 GOSUB 9000
120 PRINT "VALUE = ";F1
900 END
8998 REM * YES/NO PROCESSOR
9000 INPUT ANSWER$
9010 IF ANSWER$="YES" THEN F1=1:GOTO 9090
9020 IF ANSWER$="NO" THEN F1=0:GOTO 9090
```

```
?10
INPUT INTEREST PERIODS PER YEAR
?12

11047.1315

INPUT PRINCIPAL ($)
?10000
INPUT INTEREST RATE (% PER YEAR)
?15
INPUT INTEREST PERIODS PER YEAR
?365

11617.989

INPUT PRINCIPAL ($)
?0
INPUT INTEREST RATE (% PER YEAR)
```

Chapter 5

Section 5-1

Problem No. 2

```
90 REM * FIND EARLIEST ITEM IN DATA
95 DIM E$(20),A$(20)
100 READ E$
110 PRINT E$
150 READ A$
155 IF A$="STOP" THEN 300
160 PRINT A$
170 IF A$<E$ THEN E$=A$
180 GOTO 150
```

```
9040 PRINT "'YES' OR 'NO' PLEASE"
9045 PRINT
9050 GOTO 9000
9090 RETURN


RUN
QUESTION ?OK
'YES' OR 'NO' PLEASE

?YES
VALUE = 1

READY
```

Section 5-2

Problem No. 2

```
90 REM * TEST FORMATTER
100 DIM X$(10),D$(10)
120 PRINT "TEST VALUE";
130 INPUT M1
135 IF M1=-9999 THEN END
140 GOSUB 1000
150 PRINT M1;" = $";D$
160 PRINT
170 GOTO 120
999 REM * FORMAT DOLLARS AND CENTS
1000 M9=INT(M1*100+0.5)
1010 X$=STR$(M9)
1020 D9=LEN(X$)-2
1030 D$=X$(1,D9)
1040 D$(LEN(D$)+1)="."
```

299

Section 5-2
Problem No. 2
(continued)

```
1050 D$(LEN(D$)+1)=X$(D9+1,LEN(X$))
1060 RETURN


RUN
TEST VALUE?31.9
31.9 = $31.90

TEST VALUE?-9999

READY
```

Problem No. 4

```
80 REM * HANDLE ZERO AMOUNT
90 REM * TEST FORMATTER
100 DIM X$(10),D$(10)
120 PRINT "TEST VALUE";
130 INPUT M1
135 IF M1=-9999 THEN END
138 IF M1=0 THEN D$="0.00":GOTO 150
140 GOSUB 1000
150 PRINT M1;" = ";D$
160 PRINT
170 GOTO 120
999 REM * FORMAT DOLLARS AND CENTS
1000 M9=INT(M1*100+0.5)
1010 X$=STR$(M9)
1020 D9=LEN(X$)-2
1030 D$=X$(1,D9)
1040 D$(LEN(D$)+1)="."
```

```
165 J=J+1
180 NEXT I
200 PRINT D$;" BECOMES ";E$
205 D$=""
206 E$=""
210 PRINT
220 GOTO 110


RUN
INPUT TEST STRING?<$1,234.51>
<$1,234.51> BECOMES -1234.51

INPUT TEST STRING?<1234>
<1234> BECOMES -1234

INPUT TEST STRING?23.41
23.41 BECOMES 23.41

INPUT TEST STRING?STOP

READY
```

Problem No. 8

```
90 REM * SIMPLE SCROLLING MESSAGE
95 PRINT "⌡":REM ESC+CTRL CLEAR TO CLEAR
SCREEN
98 POKE 752,1
100 DIM A$(60)
120 READ A$:IF A$="DONE" THEN 999
130 ALEN=LEN(A$)
140 L=ALEN+39
220 DIM B$(L)
230 B$(1,39)="                   ":REM 39 SPACES
```

```
1050 D$(LEN(D$)+1)=X$(D9+1,LEN(X$))
1060 RETURN

RUN
TEST VALUE?31.495
31.495 = 31.50

TEST VALUE?0
0 = 0.00

TEST VALUE?92.1
92.1 = 92.10

TEST VALUE?-9999

READY
```

Problem No. 6

```
90 REM * CONVERT FROM <$1234.56> TO
-1234.56
100 DIM D$(20),E$(20)
110 PRINT "INPUT TEST STRING";
120 INPUT D$
125 IF D$="STOP" THEN END
127 J=1
130 FOR I=1 TO LEN(D$)
140 IF D$(I,I)="<" THEN E$(J,J)="-":
J=J+1:GOTO 180
145 IF D$(I,I)=">" THEN 180
147 IF D$(I,I)="," THEN 180
150 IF D$(I,I)="$" THEN E$(I,I)=" ":
GOTO 180
160 E$(J,J)=D$(I,I)
```

```
240 B$(LEN(B$)+1)=A$
250 POSITION 0,10
260 PRINT B$(1,40)
270 B$(1,L-1)=B$(2,L)
280 B$(L,L)=B$(1,1)
290 FOR DELAY=1 TO 25:NEXT DELAY
300 GOTO 250
900 DATA THIS IS THE MESSAGE THAT SCROLLS
ACROSS THE SCREEN
990 DATA DONE
999 END
```

Chapter 6

Section 6-1

Problem No. 2

```
90 REM * ENTER THE TEMPERATURES IN
ARRAY WEEK
95 DIM WEEK(7)
100 FOR J=1 TO 7
110 READ TEMP
115 WEEK(J)=TEMP
120 NEXT J
145 REM * ST UP INITIAL CONDITIONS
150 TOTAL=WEEK(1)
160 HI=WEEK(1):LO=WEEK(1)
170 DAYHI=1:DAYLO=1
190 REM
200 FOR J=2 TO 7
210 TOTAL=TOTAL+WEEK(J)
230 IF WEEK(J)>HI THEN HI=WEEK(J):DAYHI=J
240 IF WEEK(J)<LO THEN LO=WEEK(J):DAYLO=J
```

Section 6-1
Problem No. 2
(continued)

```
280 NEXT J
290 REM
300 PRINT "AVERAGE TEMP: ";TOTAL/7
320 PRINT "HIGHEST TEMP: ";HI;"
ON DAY "; DAYHI
330 PRINT " LOWEST TEMP: ";LO;"
ON DAY "; DAYLO
890 REM
900 DATA 72,78,76,79,85,85,71
990 END


RUN
AVERAGE TEMP: 78
HIGHEST TEMP: 85 ON DAY 5
 LOWEST TEMP: 71 ON DAY 7

READY
```

Problem No. 4

```
90 REM * DRAWING TEN NUMBERS AT
RANDOM FROM AMONG TEN
92 REM * COUNTING UNUSED DRAWS
95 DIM A(10)
100 FOR J=1 TO 10
110 A(J)=1
120 NEXT J
180 UNUSED=0
190 REM
200 FOR J=1 TO 10
```

```
ORDER"
310 FOR J=20 TO 1 STEP -1
320 PRINT AR(J);" ";
330 NEXT J
900 END


RUN

DISPLAY IN ORDER
2 4 6 8 10 12 14 16 18 20 22 24 26 28
30 32 34 36 38 40

DISPLAY IN REVERSE ORDER
40 38 36 34 32 30 28 26 24 22 20 18 16
14 12 10 8 6 4 2

READY
```

Problem No. 8

```
90 REM * DISPLAY COMPOSITE OF TWO ARRAYS
95 DIM A1(15),A2(15),A3(30)
100 READ N1
110 FOR J=1 TO N1
120 READ X
125 A1(J)=X
130 NEXT J
200 READ N2
210 FOR J=1 TO N2
220 READ X
225 A2(J)=X
230 NEXT J
```

```
210 R=INT(RND(Ø)*1Ø+1)
250 IF A(R)=Ø THEN UNUSED=UNUSED+1
:GOTO21Ø
26Ø PRINT " ";R;
27Ø A(R)=Ø
28Ø NEXT J
29Ø PRINT
295 PRINT UNUSED;" UNUSED NUMBERS"
3ØØ END
```

```
RUN
 4 6 1 1Ø 7 8 2 9 5 3
14 UNUSED NUMBERS
```

READY

Problem No. 6

```
9Ø REM * DISPLAY ELEMENTS IN ORDER
AND IN REVERSE ORDER
1ØØ DIM AR(2Ø)
11Ø FOR J=1 TO 2Ø
12Ø AR(J)=2*J
13Ø NEXT J
19Ø PRINT "⌡":REM ESC CTRL+CLEAR
2ØØ PRINT "DISPLAY IN ORDER"
21Ø FOR J=1 TO 2Ø
22Ø PRINT AR(J);" ";
23Ø NEXT J
24Ø PRINT :PRINT
3ØØ PRINT "DISPLAY IN REVERSE
```

```
3ØØ FOR J=1 TO N1
3Ø5 J3=J
31Ø A3(J3)=A1(J)
315 NEXT J
325 FOR J=1 TO N2
33Ø FOR K=1 TO J3
335 IF A3(K)=A2(J) THEN 365
34Ø NEXT K
345 J3=J3+1
35Ø A3(J3)=A2(J)
355 GOTO 365
36Ø NEXT K
365 NEXT J
4ØØ PRINT "THE COMPOSITE ARRAY:"
41Ø FOR J=1 TO J3
42Ø PRINT A3(J);" ";
43Ø NEXT J
8ØØ END
9ØØ DATA 3,6,3,9
91Ø DATA 4,2,8,6,5
```

```
RUN
THE COMPOSITE ARRAY:
 6 3 9 2 8 5
```

READY

Section 6-2

Problem No. 2

```
9Ø REM * FIND MAXIMUM TEMP
95 DIM TEMP(7,4)
```

Section 6-2
Problem No. 2
(continued)

```
100 FOR DAY=1 TO 7
110 FOR READING=1 TO 3
120 READ TEMP
125 TEMP(DAY,READING)=TEMP
130 NEXT READING
140 NEXT DAY
150 GOSUB 2000
175 REM
180 PRINT " MAXIMUM"
190 PRINT "Day Temp"
200 FOR DAY=1 TO 7
210 PRINT DAY;"     ";TEMP(DAY,4)
220 NEXT DAY
230 PRINT
900 END
980 REM
1000 DATA 76,79,75,72,77,76
1020 DATA 74,79,81,75,80,83
1040 DATA 80,77,70,68,65,65
1060 DATA 65,67,76
1998 REM * FIND MAXIMUM TEMPERATURE
HERE
2000 FOR DAY=1 TO 7
2010 TEMP(DAY,4)=TEMP(DAY,1)
2020 NEXT DAY
2050 FOR DAY=1 TO 7
2060 FOR READING=1 TO 3
2070 IF TEMP(DAY,READING)>TEMP(DAY,4)
THEN TEMP(DAY,4)=TEMP(DAY,READING)
2080 NEXT READING
```

```
5006 NEXT I9
5010 FOR I9=N0 TO BEGIN
5015 IF NAMES$(START(I9),START(I9))=PP$
(LAST,LAST) AND AVNAMES(I9)=1 THEN POP
:GOTO 5050
5016 NEXT I9
```

Problem No. 4

```
90 REM * FIND THE ALPHABETICALLY FIRST NAME
95 DIM WEEK$(63),DAY$(9),FIRST$(9)
100 GOSUB 900
198 REM * FIND THE FIRST ONE
200 FIRST$=WEEK$(1,9):NUM=1
210 FOR I9=2 TO 7
220 START=(I9-1)*9+1
225 FINISH=START+8
230 IF WEEK$(START,FINISH)<FIRST$ THEN
FIRST$=WEEK$(START,FINISH):NUM=I9
240 NEXT I9
250 PRINT "ALPHABETICALLY FIRST IS ";FIRST$
260 PRINT "IN POSITION NUMBER: ";NUM
290 END
898 REM * READ WEEKDAY NAMES
900 FOR I9=1 TO 7
910 READ DAY$
915 IF LEN(DAY$)<9 THEN DAY$(LEN(DAY$)+1)="
":GOTO 915
920 WEEK$(LEN(WEEK$)+1)=DAY$
930 NEXT I9
990 RETURN
996 REM * DAY DATA
```

```
2090 NEXT DAY
2095 RETURN


RUN
 MAXIMUM
Day Temp
1     79
2     77
3     81
4     83
5     80
6     68
7     76
```

```
READY
```

Section 6-3

Problem No. 2

```
1 REM * CHANGES IN COMPUTER RESPONSE
SUBROUTINE FOR MORE RANDOM SELECTION
2 REM
4 REM * FIRST DELETE LINES 5000 AND
5015
6 REM * THEN ENTER THE FOLLOWING LINES
5000 BEGIN=INT(RND(0)*N0+1)
5002 LAST=LEN(PP$)
5003 FOR I9=BEGIN TO N0
5005 IF NAMES$(START(I9),START(I9))=PP$
(LAST,LAST) AND AVNAMES(I9)=1 THEN POP
:GOTO 5050
```

```
1000 DATA SUNDAY,MONDAY,TUESDAY
1010 DATA WEDNESDAY,THURSDAY,FRIDAY
1020 DATA SATURDAY


RUN
ALPHABETICALLY FIRST IS FRIDAY
IN POSITION NUMBER: 6

READY
```

Chapter 7

Section 7-1

Problem No. 2

```
100 PRINT "INPUT AN INTEGER";
110 INPUT N
120 IF N=0 THEN END
130 FOR E=8 TO 0 STEP -1
140 T=10^E
150 I=INT(N/T)
160 PRINT I;"  ";
170 R=INT(N-I*T+0.5)
180 N=R
190 NEXT E
200 PRINT :PRINT
210 GOTO 100


RUN
INPUT AN INTEGER?31368
0   0   0   0   3   1   3   6   8
```

INPUT AN INTEGER?0

READY

Problem No. 4

```
90 REM * PROGRAM TO FIND ALL THREE
DIGIT INTEGERS
92 REM * WHERE THE REVERSE IS ALSO
A PRIME NUMBER
94 REM * DUPLICATES ELIMINATED
95 POKE 82,0
98 DIM A(43):ARRAYNUM=1
99 FOR I=1 TO 43:A(I)=0:NEXT I
100 FOR X=100 TO 900 STEP 200
110 FOR Y=1 TO 99 STEP 2
120 N=X+Y:NU=N
130 FOR Z=3 TO SQR(N)
140 QUOT=N/Z
150 IF QUOT=INT(QUOT) THEN POP
:GOTO 370
160 NEXT Z
170 GOSUB 900
200 FOR Z=3 TO SQR(RE)
210 QUOT=RE/Z
220 IF QUOT=INT(QUOT) THEN POP
:GOTO 370
230 NEXT Z
240 FOR I=1 TO ARRAYNUM
```

```
200 PRINT "ENTER AN INTEGER";
205 INPUT I
210 IF I<=0 THEN 999
220 IF I<655536 THEN 300
230 PRINT "TOO LARGE":PRINT :GOTO
200
298 REM * LOAD THE ARRAY
300 FOR J=16 TO 1 STEP -1
310 IF I/2=INT(I/2) THEN A(J)=0
320 IF I/2<>INT(I/2) THEN A(J)=1
340 I=INT(I/2)
360 NEXT J
398 REM * DISPLAY RESULTS
399 FLAG=0
400 FOR J=1 TO 16
405 IF A(J)=1 THEN FLAG=1
410 IF FLAG=1 THEN PRINT A(J);
420 NEXT J
455 PRINT :PRINT
460 GOTO 200
999 END
```

```
RUN
ENTER AN INTEGER?129
10000001

ENTER AN INTEGER?32689
111111110110001

ENTER AN INTEGER?765432
TOO LARGE
```

```
250 IF RE=A(I) THEN POP :GOTO 370
260 NEXT I
270 A(ARRAYNUM)=NU
280 ARRAYNUM=ARRAYNUM+1
300 PRINT NU,
370 NEXT Y
390 NEXT X
500 END
896 REM
898 REM * REVERSE HERE
900 DIGIT3=INT(N/100)
920 N=N-DIGIT3*100
930 DIGIT2=INT(N/10)
940 N=N-DIGIT2*10
950 DIGIT1=N
960 RE=DIGIT1*100+DIGIT2*10+DIGIT3
970 RETURN
```

```
RUN
101        107        113        131
149        151        157        167
179        181        191        199
313        337        347        353
359        373        383        389
709        727        739        757
769        787        797        919
929

READY
```

Section 7-2

Problem No. 2

```
100 REM * CONVERT DECIMAL TO BINARY
110 DIM A(16)
```

```
ENTER AN INTEGER?-1

READY
```

Section 7-3: General-Interest Problems

Problem No. 2

```
90 REM * NUMBER GUESSING GAME
95 PRINT ")":REM ESC CTRL+CLEAR
100 PRINT "I will pick of a number between"
105 PRINT "1 and whatever you want."
110 PRINT "How large would you like me
to go "
120 INPUT MAX
140 IF MAX<1 THEN PRINT "TOO SMALL":GOTO
110
150 NUMBER=INT(RND(0)*MAX)+1
160 TRIES=0
200 PRINT
205 PRINT "Your Guess: ";
207 INPUT GUESS
208 TRIES=TRIES+1
210 IF GUESS<1 THEN 450
220 IF GUESS>MAX THEN 400
230 IF GUESS=NUMBER THEN 500
240 IF GUESS<NUMBER THEN 300
250 PRINT "Try LOWER"
260 GOTO 200
300 PRINT "Try HIGHER"
310 GOTO 200
400 PRINT "That's bigger than your limit"
410 GOTO 200
450 PRINT "Too SMALL"
460 GOTO 200
```

```
500 PRINT "*** YOU GOT IT! ***"
510 PRINT "IT TOOK YOU ";TRIES;"
GUESSES."
```

RUN

```
I will pick of a number between
1 and whatever you want.
How large would you like me to go
?100

Your Guess: ?50
Try LOWER

Your Guess: ?25
Try LOWER

Your Guess: ?12
Try LOWER

Your Guess: ?6
Try LOWER

Your Guess: ?3
Try HIGHER

Your Guess: ?4
*** YOU GOT IT! ***
IT TOOK YOU 6 GUESSES.
```

READY

```
1130 PRINT :GOTO 1100
1140 MONTHLYRATE=ANNRATE/12
1150 INTEREST=MONTHLYRATE/100
1188 REM * EXIT WITH MONTHLY RATE IN
INTEREST
1190 RETURN
1198 REM * GET PRINCIPAL
1200 PRINT "ENTER PRINCIPAL ($) ";
1210 INPUT PRINCIPAL
1290 RETURN
1298 REM * GET NUMBER OF YEARS
1300 PRINT "ENTER NUMBER OF YEARS ";
1310 INPUT YEARS
1320 PAYMENTS=YEARS*12
1388 REM * EXIT WITH NUMBER OF
PAYMENTS IN PAYMENTS
1390 RETURN
1398 REM COMPUTE MONTHLY PAYMENT
1400 X=(1+INTEREST)^PAYMENTS
1410 PA=(PRINCIPAL*INTEREST*X)/(X-1)
1490 RETURN
1498 REM * FORMAT PAYMENT
1500 X=INT(PA*100+0.5)
1510 P$=STR$(X)
1520 Y=LEN(P$)-2
1530 VALUE$=P$(1,Y)
1533 VALUE$(Y+1)="."
1536 VALUE$(Y+2)=P$(Y+1,Y+2)
1590 RETURN
```

RUN

```
ANNUAL INTEREST (%) ?15.5
```

Problem No. 4

```
90 REM * COMPUTE MONTHLY MORTAGE
PAYMENT
95 DIM P$(15),VALUE$(15)
100 PRINT "ͻ":REM ESC CTRL+CLEAR
105 TRAP 500
110 GOSUB 1100
120 GOSUB 1200
130 GOSUB 1300
140 GOSUB 1400
150 GOSUB 1500
205 PRINT :PRINT "MONTHLY PAYMENT =
$ ";VALUE$
210 PA=VAL(VALUE$)*PAYMENTS
220 GOSUB 1500
230 PRINT " TOTAL PAYMENTS = $ "
;VALUE$
240 PA=VAL(VALUE$)-PRINCIPAL
250 GOSUB 1500
260 PRINT " TOTAL INTEREST = $ "
;VALUE$
400 END
500 TRAP 40000
510 PRINT "BAD INPUT VALUE. PLEASE
START
OVER."
515 FOR I=1 TO 500:NEXT I
520 GOTO 100
1098 REM * GET INTEREST RATE
1100 PRINT "ANNUAL INTEREST (%) ";
1110 INPUT ANNRATE
1120 IF ANNRATE>1 THEN 1140
1125 PRINT "PERCENTAGE PLEASE"
```

```
ENTER PRINCIPAL ($) ?90000
ENTER NUMBER OF YEARS ?30

MONTHLY PAYMENT = $ 1174.07
  TOTAL PAYMENTS = $ 422665.20
  TOTAL INTEREST = $ 332665.20

READY
```

Section 7-3: Math-Oriented Problems

Problem No. 2

```
90 REM * EUCLID'S ALGORITHM
95 TRAP 500
100 PRINT "FIRST NUMBER";
110 INPUT NUMBER1
120 IF NUMBER1=0 THEN END
130 PRINT "SECOND NUMBER";
140 INPUT NUMBER2
150 QUOTIENT=INT(NUMBER1/NUMBER2)
160 DIFFERENCE=NUMBER1-NUMBER2*
QUOTIENT
170 IF DIFFERENCE=0 THEN 200
175 NUMBER1=NUMBER2
180 NUMBER2=DIFFERENCE
190 GOTO 150
200 PRINT "GREATEST COMMON FACTOR:
";NUMBER2
210 PRINT
220 GOTO 100
500 TRAP 40000
510 PRINT "BAD INPUT. TRY AGAIN."
520 FOR I=1 TO 500:NEXT I
```

Section 7-3
Problem No. 2
(continued)
```
530 GOTO 100


RUN
FIRST NUMBER?1001
SECOND NUMBER?1300
GREATEST COMMON FACTOR: 13

FIRST NUMBER?0

READY
```

Problem No. 4

```
90 REM * FIND PERFECT NUMBERS
100 DIM FACTOR(50)
110 COUNTER=0
200 FOR NUMBER=2 TO 32767 STEP 2
210 SUM=1
220 F1=1
230 FACTOR(F1)=1
250 FOR VALUE=2 TO SQR(NUMBER)
260 X=NUMBER/VALUE
270 IF X<>INT(X) THEN 330
280 FACTOR(F1+1)=VALUE
290 FACTOR(F1+2)=X
300 F1=F1+2
310 SUM=SUM+VALUE+X
320 IF SUM>NUMBER THEN 420
330 NEXT VALUE
```

```
98 PRINT "    PYTHAGOREAN TRIPLES"
99 PRINT "   SIDE1 SIDE2      SIDE3"
100 FOR SIDE1=1 TO 25
120 FOR SIDE2=SIDE1+1 TO 25
130 X=SIDE1*SIDE1+SIDE2*SIDE2
140 FOR SIDE3=SIDE2+1 TO 50
150 X1=SIDE3*SIDE3
155 IF X1>X THEN 180
160 IF X1<X THEN 170
165 PRINT "    ";SIDE1,SIDE2,SIDE3
170 NEXT SIDE3
180 NEXT SIDE2
190 NEXT SIDE1


RUN
```

```
      PYTHAGOREAN TRIPLES
   SIDE1  SIDE2      SIDE3
     3      4          5
     5     12         13
     6      8         10
     7     24         25
     8     15         17
     9     12         15
    10     24         26
    12     16         20
    15     20         25
    18     24         30
    20     21         29

READY
```

```
340 IF SUM<>NUMBER THEN 420
350 PRINT NUMBER;" IS PERFECT AND ITS
FACTORS ARE:"
360 FOR I9=1 TO F1
370 PRINT FACTOR(I9);" ";
380 NEXT I9
390 PRINT :PRINT
400 COUNTER=COUNTER+1
410 IF COUNTER=4 THEN END
420 NEXT NUMBER
900 END


RUN
6 IS PERFECT AND ITS FACTORS ARE:
1 2 3

28 IS PERFECT AND ITS FACTORS ARE:
1 2 14 4 7

496 IS PERFECT AND ITS FACTORS ARE:
1 2 248 4 124 8 62 16 31

8128 IS PERFECT AND ITS FACTORS ARE:
1 2 4064 4 2032 8 1016 16 508 32 254
64 127

READY
```

Problem No. 6

```
90 REM * FIND PYTHAGOREAN TRIPLES
95 PRINT "⌡":REM ESC CTRL+CLEAR
```

Problem No. 8

```
90 REM * FIND PI FROM A SEQUENCE
95 PRINT "⌡":REM ESC CTRL+CLEAR
98 PRINT "Have patience....I'm working
on it.."
99 PRINT
100 PI=2
110 FOR I9=1 TO 1500 STEP 4
120 PI=PI+16/((I9)*(I9+2)*(I9+4))
140 NEXT I9
150 PRINT "APPROXIMATE VALUE OF PI = ";PI
900 END


RUN
Have patience....I'm working on it..

APPROXIMATE VALUE OF PI = 3.14158981

READY
```

Chapter 9

Section 9-2

Problem No. 2

```
5 REM * EDIT SPELLING FOR GEOGRAPHY
10 DIM NAME$(20),NAMES$(2000),START(100)
,FINISH(100),X$(20),Y$(20)
21 NAME$=""
25 GOSUB 8000:REM * READ NAMES ARRAY
```

```
30 GOSUB 12000:REM * EDIT NAMES
SPELLING
35 GOSUB 8500:REM * REWRITE THE NAMES
FILE
110 END
7998 REM * READ PLACES FILE
8000 OPEN #3,4,0,"D:PLACES"
8005 INPUT #3;N0
8010 FOR I9=1 TO N0
8015 LONGSIZE=LEN(NAMES$)
8020 START(I9)=LONGSIZE+1
8025 INPUT #3;NAME$
8030 SIZE=LEN(NAME$)
8035 FINISH(I9)=LONGSIZE+SIZE
8040 NAMES$(LEN(NAMES$)+1)=NAME$
8045 NEXT I9
8050 CLOSE #3
8090 RETURN
8498 REM * UPDATE PLACES FILE
8500 CLOSE #3:OPEN #3,8,0,"D:PLACES"
8510 PRINT #3;N0
8520 FOR I9=1 TO N0
8530 PRINT #4;NAMES$(START(I9),
FINISH(I9))
8540 NEXT I9
8550 CLOSE #3
8590 RETURN
11998 REM * EDIT PLACE NAMES SPELLING
12000 PRINT "EDITING PLACE NAMES."
12005 PRINT
12010 PRINT "NAME TO FIX";:INPUT Y$
```

312

```
2 REM * TO PRODUCE MORE RANDOM SELECTION
3 REM
4 REM * FIRST DELETE LINES 5000 TO 5015
6 REM * THEN ENTER THE FOLLOWING LINES
5000 ST=INT(RND(0)*N0+1)
5002 FOR I9=ST TO N0
5004 IF NAMES$(START(I9),START(I9))=PP$
(LAST,LAST) AND AVNAMES(I9)
=1 THEN POP :GOTO 5050
5006 NEXT I9
5010 FOR I9=1 TO ST
5012 IF NAMES$(START(I9),START(I9))=PP$
(LAST,LAST) AND AVNAMES(I9)
=1 THEN POP :GOTO 5050
5014 NEXT I9
```

Chapter 9

Section 4

Problem No. 2

```
10 REM * ID=>ENTRY IDENTIFICATION NO.
11 REM * NS=>NEW SPACE
12 REM * DS=>DELETED SPACE
20 DIM F$(10),DA$(120),LE(9),LA$(4*9)
,A$(30)
21 DIM SEC(100),BYT(100),YN$(1)
30 F$="D:MAILLIST"
198 REM * MAILING LIST ENTRY EDITOR
200 GOSUB 1000:REM * READ DATA LABELS
210 GOSUB 900:REM READ AVAILABLE SPACE
PARAMETERS
220 GOSUB 3000:REM * REQUEST ID
```

```
12015 IF X$="DONE" THEN 12090
12020 FOR I9=1 TO NØ
12025 IF NAMES$(START(I9),FINISH
(I9))=Y$ THEN NAMENUM=I9:GOTO 1204
Ø
12030 NEXT I9
12035 PRINT "NOT FOUND":GOTO 12005
12040 PRINT "NEW PLACE";:INPUT X$
12045 FOR J9=1 TO NØ
12050 IF NAMES$(START(J9),FINISH
(J9))=X$ THEN 12200
12055 NEXT J9
12060 TEMPNAME$=NAMES$(1,FINISH
(NAMENUM-1))
12065 TEMPNAME$(LEN(TEMPNAME$)+1)=X$
12070 TEMPNAME$(LEN(TEMPNAME$)+1)
=NAMES$(START(NUMENUM+1),FINISH(N
Ø))
12075 NAMES$=TEMPNAME$
12080 DIFF=LEN(X$)-LEN(Y$)
12085 FINISH(NAMENUM)=FINISH
(NAMENUM)-DIFF
12090 FOR I9=NAMENUM+1 TO NØ
12100 START(I9)=START(I9)-DIFF:
FINISH(I9)=FINISH(I9)-DIFF
12110 NEXT I9
12120 GOTO 12005
12200 PRINT "DUPLICATE NAME-REENTER"
12210 GOTO 12005
12290 RETURN
```

Problem No. 4

```
1 REM * CHANGES IN COMPUTER RESPONSE
  SUBROUTINE
```

```
225 IF ID=Ø THEN END
230 GOSUB 3100:REM * READ AND  EDIT
ENTRY
250 GOSUB 600:REM * WRITE NEW ENTRY
270 GOTO 220:REM * DO IT AGAIN
598 REM * WRITE ENTRY
600 CLOSE #3:OPEN #3,12,0,F$
610 FOINT #3,SEC(ID),BYT(ID)
620 PRINT #3;DA$
650 CLOSE #3
660 DA$=""
690 RETURN
898 REM * READ AVAILABLE SPACE AND
SECTOR/BYTE FILE
900 CLOSE #3:OPEN #3,12,0,F$
910 INPUT #3;NS
920 INPUT #3;DS
940 CLOSE #3
950 OPEN #3,4,0,"D:FINDFILE"
955 FOR I=Ø TO 100
960 INPUT #3;SECTOR
965 SEC(I)=SECTOR
970 INPUT #3;BYTE
975 BYT(I)=BYTE
980 NEXT I
985 CLOSE #3
990 RETURN
998 REM * READ DATA LABELS AND LIMITS
1000 READ NØ
1010 FOR X9=1 TO NØ
1020 START=(X9-1)*4+1:FINISH=START+3
1022 READ A$,X
1025 LA$(START,FINISH)=A$
1028 LE(X9)=X
1030 NEXT X9
```

```
1090 RETURN
1998 REM * DATA LABEL & LIMITS
2000 DATA 9
2005 DATA ID #,4
2010 DATA CODE,5
2015 DATA LAST,20
2020 DATA FRST,20
2025 DATA ADDR,30
2030 DATA CITY,16
2035 DATA STAT,2
2040 DATA ZIPC,5
2045 DATA PHON,17
2998 REM REQUEST ID
3000 PRINT
3010 PRINT "ID #:";:INPUT ID
3020 IF ID<NS AND ID>=0 THEN 3090
3030 PRINT "NON-EXISTENT ID":GOTO 3000
3090 RETURN
3098 REM * READ THE ENTRY IF IT IS
REAL
3100 CLOSE #3:OPEN #3,12,0,F$
3120 POINT #3,SEC(ID),BYT(ID)
3125 INPUT #3;DA$:IF ID=VAL(DA$(1,4))
THEN 3140
3130 PRINT ID;" HAS BEEN DELETED"
3135 E1=0:GOTO 3280
3140 CLOSE #3
3142 PLACE=2
3145 FOR I9=2 TO N0
3150 START=(I9-1)*4+1:FINISH=START+3
3155 PRINT LA$(START,FINISH);":";
3160 PRINT DA$(PLACE,PLACE+LE(I9)-1)
```

```
190 GET #1,X
195 IF X<>42 AND X<>43 AND CHR$(X)<>"S"
AND CHR$(X)<>"s" THEN 190
197 GOSUB 3000
200 IF X=42 THEN PITCH=PITCH+1
210 IF PITCH>25 THEN PITCH=25
220 IF X=43 THEN PITCH=PITCH-1
230 IF PITCH<1 THEN PITCH=1
240 IF CHR$(X)="S" OR CHR$(X)="s" THEN 500
250 SOUND 0,A(PITCH),10,6
255 GOSUB 2000
260 SOUND 1,A(PITCH+4),10,6
265 GOSUB 2000
270 SOUND 2,A(PITCH+7),10,6
275 GOSUB 2000
280 SOUND 3,A(PITCH+12),10,6
285 PRINT "First note of chord = ";A(PITCH)
290 GOTO 180
500 FOR I=0 TO 3
510 SOUND I,0,0,0
520 NEXT I
550 END
1000 DATA 243,230,217,204,193,182,173
1010 DATA 162,153,144,136,128,121
1020 DATA 114,108,102,96,91,85,81,76
1030 DATA 72,68,64,60,57,53,50,47,45
1040 DATA 42,40,37,35,33,31,29
2000 REM * DELAY SUBROUTINE
2010 FOR I=1 TO 50:NEXT I
2020 RETURN
3000 REM * TURN SOUNDS OFF
3010 SOUND 0,0,0,0:SOUND 1,0,0,0
3020 SOUND 2,0,0,0:SOUND 3,0,0,0
3030 RETURN
```

```
3170 PRINT "IS THIS OK (Y/N)";:INPUT
YN$
3175 IF YN$="Y" THEN 3255
3180 IF YN$="N" THEN 3200
3190 PRINT "Y OR N ONLY":GOTO 3170
3200 PRINT "    :";:INPUT A$
3210 IF LEN(A$)<=LE(I9) THEN 3230
3220 PRINT "TOO LONG":GOTO 3200
3230 IF LEN(A$)<LE(I9) THEN A$(LEN
(A$)+1)=" ":GOTO 3230
3240 DA$(PLACE,PLACE+LE(I9))=A$
3255 PLACE=PLACE+LE(I9)
3260 NEXT I9:GOTO 3290
3280 CLOSE #3
3290 RETURN
```

Chapter 10

Problem No. 2

```
90 REM * MUSICAL TONES
92 GRAPHICS 0
95 DIM A(37)
100 FOR X=1 TO 37
110 READ Y
120 A(X)=Y
130 NEXT X
170 PITCH=20
176 PRINT "Press '*' (right arrow)
to go higher"
177 PRINT "Press '+'  (left arrow)
to go lower"
178 PRINT "Press 'S' TO STOP"
180 CLOSE #1:OPEN #1,4,0,"K"
```

Problem No. 4

```
90 REM * TWO VARYING VOICES
100 FOR I=0 TO 240
110 SOUND 0,I,10,10
120 SOUND 1,I+7,10,10
130 FOR DELAY=1 TO 20:NEXT DELAY
140 SOUND 1,0,0,0
150 FOR DELAY=1 TO 20:NEXT DELAY
160 NEXT I
```

Problem No. 6

```
90 REM * INVESTIGATING "BEATING"
100 A=180
120 FOR B=170 TO 190
130 SOUND 0,A,10,10
140 SOUND 1,B,10,10
150 PRINT "A=";A,"B=";B
160 FOR DELAY=1 TO 1000:NEXT DELAY
170 NEXT B
180 SOUND 0,0,0,0
190 SOUND 1,0,0,0
200 END
```

Chapter 11

Section 11-1

Problem No. 2

```
90 REM * UPPPERCASE THEN LOWERCASE
MESSAGE
95 DIM MESSAGE$(20)
```

Section 1

Problem No. 2
(continued)

```
100 GRAPHICS 1+16
110 MESSAGE$="KEEP YOUR EYE"
115 YPOS=10:GOSUB 500
120 PRINT #6;MESSAGE$
122 MESSAGE$="ON THIS MESSAGE"
124 YPOS=12:GOSUB 500
125 PRINT #6;MESSAGE$
130 FOR DELAY=1 TO 2000:NEXT DELAY
140 POKE 756,226
150 FOR DELAY=1 TO 2000:NEXT DELAY
190 END
498 REM * DETERMINE POSITION FOR
MESSAGE
500 XPOS=(20-LEN(MESSAGE$))/2
510 POKE 84,YPOS
520 POKE 85,XPOS
530 RETURN
```

Problem No. 4

```
90 REM * REMOVE HEARTS FROM GR.2
CHARACTER SET
105 TOP=PEEK(106)-4
108 POKE 106,TOP
110 GRAPHICS 2
198 REM * MOVE CHARACTERS AND
REDEFINE THE "HEART"
230 CHARGO=TOP*256
232 CHARFROM=PEEK(756)*256
235 PRINT "Moving the character set.
Hold on.."
```

```
150 COLOR 3
160 PLOT 25,11:DRAWTO 28,11:DRAWTO 28,7
170 PLOT 15,11:DRAWTO 12,11:DRAWTO 12,7
180 PLOT 18,15:DRAWTO 18,18:DRAWTO 17,18
190 PLOT 22,15:DRAWTO 22,18:DRAWTO 23,18
200 FOR I=1 TO 15
210 SETCOLOR 1,I,6
220 FOR DELAY=1 TO 100:NEXT DELAY
230 SETCOLOR 1,0,0
240 FOR DELAY=1 TO 100:NEXT DELAY
250 NEXT I
260 SETCOLOR 1,5,6
```

Problem No. 4

```
90 REM * TESTING FOR LOCATION IN
GRAPHICS 3
100 GRAPHICS 3
103 SETCOLOR 4,5,4:SETCOLOR 0,12,6:
SETCOLOR 1,8,6
106 COLOR 1
110 FOR I=10 TO 30
120 PLOT I,5:DRAWTO I,15
130 NEXT I
190 FOR I=1 TO 10
200 X=INT(RND(0)*39)+1
210 Y=INT(RND(0)*19)+1
220 LOCATE X,Y,N
230 IF N=1 THEN PRINT "YOU ARE INSIDE
THE SQUARE":GOTO 245
240 PRINT "YOU ARE OUTSIDE THE
SQUARE"
245 COLOR 2:PLOT X,Y
```

```
240 FOR PLACE=0 TO 1023
250 POKE CHARGO+PLACE,PEEK(CHARFROM+
PLACE)
260 NEXT PLACE
280 FOR I=0 TO 7
282 POKE CHARGO+I+(64*8),0
284 NEXT I
290 POSITION 3,4
295 POKE 756,226
300 PRINT #6;"WHAT A MESS"
310 PRINT "5THE OLD CHARACTER SET
HAS HEART"
320 FOR DELAY=1 TO 1000:NEXT DELAY
330 POKE 756,TOP+2
335 POSITION 1,4
337 PRINT #6;"WHERE IS THE MESS"
340 PRINT "5THE NEW CHARACTER SET
HAS NO HEART"
```

Section 11-3

Problem No. 2

```
100 GRAPHICS 3
105 COLOR 1
110 PLOT 17,2:DRAWTO 23,2:DRAWTO 23,
7:DRAWTO 17,7:DRAWTO 17,2
115 COLOR 2
120 PLOT 19,4:PLOT 21,4
125 COLOR 1
130 PLOT 19,8:DRAWTO 21,8
140 PLOT 16,9:DRAWTO 24,9:DRAWTO 24,
14:DRAWTO 16,14:DRAWTO 16,9
```

```
250 FOR DELAY=1 TO 500:NEXT DELAY
260 NEXT I
```

Section 11-4

Problem No. 2

```
50 REM * SIMPLY ENTER PROGRAM 11-4A,B,C
51 REM * AND TYPE LINE 210 ACORDING TO
THE FORMULAS
```

Section 11-5

Problem No. 2

```
90 REM * USING FILL IN GRAPHICS 5
100 GRAPHICS 5
110 COLOR 1
115 PLOT 30,30
120 DRAWTO 50,30:DRAWTO 50,10
130 DRAWTO 30,10:POSITION 30,30
140 POKE 765,2
150 XIO 18,#1,0,0,"S:"
```

Section 11-6

Problem No. 2

```
90 REM * HORIZONTAL STRIPES IN
GRAPHICS 9
100 GRAPHICS 9
110 COL=0
120 FOR I=1 TO 10
```

Problem No. 2
(continued)

```
125 COLOR COL
130 PLOT 10,I+Y:DRAWTO 70,I+Y
140 NEXT I
150 COL=COL+1
160 Y=Y+10
170 IF COL=15 THEN GOTO 200
180 GOTO 120
200 GOTO 200
```

318.

Problem No. 4

```
90 REM * GRAPHICS 10 HORIZONTAL
STRIPED SQUARE
100 GRAPHICS 10
120 COL=1
140 Y=10
210 FOR J=1 TO 15
220 COLOR COL
230 PLOT 20,J+Y:DRAWTO 55,J+Y
240 NEXT J
250 COL=COL+1:Y=Y+15:IF COL=10
THEN 300
260 GOTO 210
300 GOTO 300
```

```
DRAWTO 0,0
620 PLOT 1,0:DRAWTO 1,159
630 PLOT 318,0:DRAWTO 318,159
640 RETURN
698 REM * PLOT AXES FOR GRAPHING
700 PLOT 3,80:DRAWTO 316,80
710 PLOT 160,3:DRAWTO 160,156
720 PLOT 161,3:DRAWTO 161,156
730 RETURN
1000 REM
1025 SCREENSTART=PEEK(88)+PEEK(89)*256
1030 CHARPLACE=SCREENSTART+Y*40+X
1040 FOR Z=1 TO LEN(A$)
1050 O$=A$(Z,Z):GOSUB 2000
1060 CHARSTART=57344+X*8
1070 FOR I=0 TO 7
1080 POKE CHARPLACE+I*40,PEEK
(CHARSTART+I)
1085 IF FLAG=1 THEN 1092
1090 NEXT I:CHARPLACE=CHARPLACE+
1:GOTO 1095
1092 NEXT I:CHARPLACE=CHARPLACE+8*40
1095 NEXT Z
1097 RETURN
2000 X=ASC(O$):IF X>127 THEN X=X-128
2010 IF X>31 AND X<96 THEN X=X-32
:GOTO 2030
2020 IF X<32 THEN X=X+64
2030 RETURN
```

Section 11-8

Problem No. 2
```
90 REM * SIMPLE PLAYER-MISSILE GRAPHICS
```

```
90 REM * PLOT A FUNCTION
100 GRAPHICS 8
102 DIM A$(15),O$(1)
105 YS=40:XS=15
110 PI=3.141592
118 REM * DRAW BORDER
120 COLOR 1:GOSUB 600
128 REM * PLOT AXES
130 GOSUB 700
148 REM * DRAW THE GRAPH
150 GOSUB 200
152 A$="X AXIS":X=25:Y=81
154 GOSUB 1000
156 A$="Y AXIS":X=19:Y=20:FLAG=1
158 GOSUB 1000
160 LIST 220
170 END
198 REM * PLOT A FUNCTION
200 FOR X1=-150 TO 150
220 Y1=X1
230 X=160+X1
240 Y=80-Y1
250 IF Y<3 OR Y>156 THEN 270
260 PLOT X,Y:PLOT X+1,Y
270 NEXT X1
280 RETURN
600 PLOT 0,0:DRAWTO 319,0
610 DRAWTO 319,159:DRAWTO 0,159:
```

```
95 REM * WITH RANDOM INITIAL VERTICAL
POSITION
100 GRAPHICS 3+16
110 SETCOLOR 4,0,0
120 POKE 559,62
130 POKE 53248,10
140 COL=84
150 POKE 704,COL
160 POKE 623,1
170 N=140
200 I=PEEK(106)-16
210 POKE 54279,I
220 POKE 53277,3
300 J=I*256+1024
310 FOR Y=J TO J+255
320 POKE Y,0
330 NEXT Y
340 RESTORE 2000
350 FOR Y=J+N TO J+N+14
360 READ Z
370 POKE Y,Z
380 NEXT Y
500 FOR X=10 TO 220
502 SOUND 1,230,6,4
505 POKE 53248,X
510 FOR DELAY=1 TO 10:NEXT DELAY
520 NEXT X
710 N=INT(RND(0)*(160-80))+80
720 GOTO 310
2000 DATA 64,96,112,112,254,254,255,235
2010 DATA 255,254,254,112,112,96,64
```

Index

Basic ATARI® BASIC

JAMES S. COAN AND RICHARD KUSHNER

This book is a complete guide to Atari BASIC and takes the reader from beginning concepts, such as entering data and obtaining output, to more advanced topics, such as numeric and string arrays, sequential and random-access files, sound generation, and the use of either tape or disk. Both low-resolution and high-resolution graphics are discussed.

The approach of this book is simple and direct. Start with short computer programs, master them quickly, add a new command, and watch as the desired effect on the program is created and illustrated. Move on to another capability. Test your knowledge with the various problem sections.

Programmer's Corner sections at the end of each chapter focus on special Atari features or advanced programming ideas. The book also includes a handy program index and solutions to all even-numbered problems.

Another Book of Interest...

Stimulating Simulations, Atari Version, **Second Edition**
C.W. Engel

Here is an exciting handbook containing twelve BASIC "simulation programs," which are actually game programs. Each of the programs is presented with a listing, sample run, instructions, and program documentation, including a flowchart and ideas for variations. "This book is a good starting point for the computer hobbyist who wishes to explore the use of the small computer in simulating real events." *Computer Notes*. #5201-4, paper, 128 pages.