MASTERING YOUR ATARI® THROUGH EIGHT BASIC PROJECTS

A Spectrum Book

Includes ready-to-run programs on disk, with spreadsheet, clock, music editor, programmable character generator, games, and more

```
T=
U=
C=
V=180-T-U
V=3.14159265*V/180
U=3.14159265*U/180
B=SIN(U)*C/SIN(V)
B?*


A=165*
B=148
C=173
```

by the staff of MICRO magazine

Mastering Your Atari®
Through Eight BASIC Projects

Tom Marshall
Editor

A SPECTRUM BOOK

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data
Main entry under title:

Mastering your Atari through 8 basic projects.

"A Spectrum Book."

1. Atari computer—Programming. 2. Basic (Computer program language) I. Computerist (Firm) II. Title: Mastering your Atari through eight basic projects.

QA76.8.A82M37 1984 001.64'2 84-3336
ISBN 0-13-559550-9

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-559550-9

Original programs: Phil Daley, Robert M. Tripp, and Loren Wright
Atari conversions and updates: Tom Marshall and John Hedderman
Edited: Emmalyn H. Bentley and Nancy Lapointe
Layout and production: Nancy Lapointe and Linda Fedas
Illustrations: Frank Wyman

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

Table of Contents

Introduction

Welcome to the world of Atari! You are about to embark on an incredible journey that will take you through the labyrinth of your microcomputer. You don't need to be a sophisticated programmer or have years of high-tech experience to enjoy this voyage; you just need an adventuresome spirit and a keen desire to learn. We hope you find this tour entertaining, stimulating, and useful.

Our itinerary includes explorations into many levels of computer programming. The absolute novice can simply 'RUN' and enjoy the programs, while the more experienced computerist can acquire new programming techniques. To get the most from this learning experience a rudimentary knowledge of BASIC programming would be helpful. This is neither a programming text nor a reference guide; teaching good programming by example is our intent. We have taken care to be consistent in our presentation of the material, including a list of concepts for each chapter, initial operating instructions, variable usage tables, and program listings in a structured format with adjacent descriptive text. We use figures, screen dumps, and short demonstration programs extensively. To become more knowledgeable about your Atari, investigate the programs in the book and on cassette. A synopsis of your journey follows.

If you have ever dreamed of becoming a *note-able* musician, you will enjoy ''Atari Player.'' This program provides you with a five-octave organ keyboard on which you can compose your own tunes or play some of your old favorites. LOAD and SAVE the songs to cassette then regale friends with your renditions of *La Paloma, Allouette,* or *Loch Lomond.* Edit your songs on the ''Song Editor'', a supplementary program to help you fully understand how to use the music capabilities of your Atari. A musical feast awaits you as you learn music programming and the use of cassette data files.

Whether you are traveling alone or with someone else you will enjoy ''MASTER,'' a simple guessing game for one or two players. Unlike the commercial version, this Atari game uses letters instead of colored pegs, but the rules are the same. We offer you three versions, and you can modify the program to play even more games. While trying to figure out your A's, B's, C's, and D's, you will have the opportunity to learn something about random numbers and flags.

And since there are always rules to every game, we have included ''Word Detective,'' another guessing game that lets you figure out what the rules are. The program introduces the concept of the computer handling letters and words, as well as numbers. A supplementary program, ''The Answer Machine,'' allows you to ask the computer questions, which it will answer with a yes, no, or maybe. If you are endowed with a good imagination, the narrative can become quite involved and you may rival even Scheherazade or Aesop.

No journey would be complete without a sporting event of some kind; we take you to a ball game. ''Breakup'' uses a paddle, joystick, or the keyboard to provide you with a simple bouncing ball game. At the same time you learn how animation is accomplished with PEEKs and POKEs to the screen memory.

If you have the time, perhaps you would like to display it in giant-size digital numbers on your Atari screen. ''Atari Clock'' proceeds, step by step, from the simplest uses of the Atari clock to a deluxe clock that displays the time in big numerals that you can read from across the room. At the same time learn a few simple techniques for programming the cursor and using string primitives.

While traveling you are bound to meet many characters. We provide you with quite a few in ''Programmable Characters,'' which redefines part of the Atari's character set. Demonstrations of this powerful technique include animated space invaders, a chameleon robot, and an abstract design that changes color at the touch of a key. Also included is a whole set of characters that lets you plot functions with extra resolution.

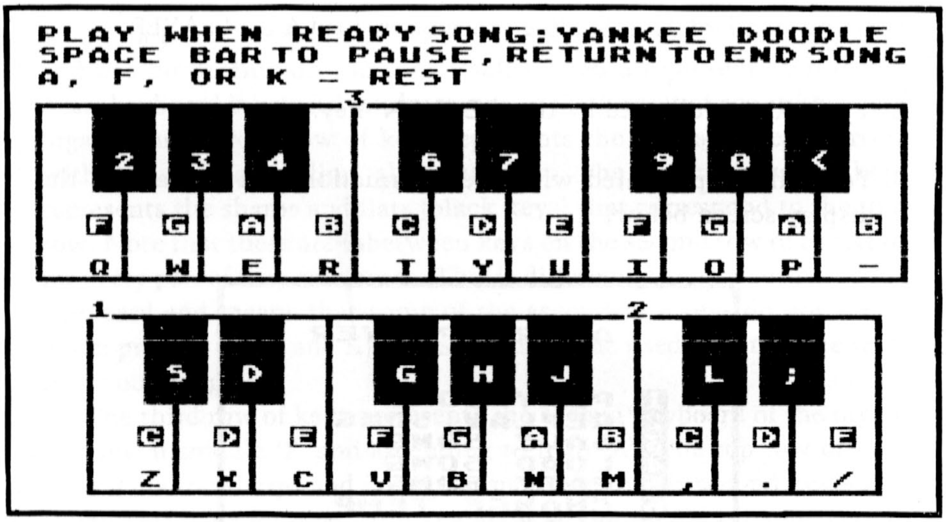
Explore the two programs in ''Sorting'' that demonstrate several methods of sorting data. You will enjoy watching the Atari's colorful character graphics sort bars of different lengths right in front of your eyes. The second program is a telephone directory that allows you to create new files, add, change, or delete old files, load existing files, save them to tape, and sort files in a feasible manner.

Finally, sail through "MICRO Calc," a miniature spread-sheet program that makes complex, repetitive calculations a breeze. LCAD the program from the diskette and your video screen will fill with ten yellow lines. A yellow diamond cursor indicates where you are on the screen. You can use this program to experiment with BASIC functions.

We sincerely hope that you find this venture exciting and educational and that it leads you to further exploration of your Atari. Now load your diskette and commence your journey. Bon voyage.

Atari Player

Making Beautiful Music



```
PLAY WHEN READY SONG:YANKEE DOODLE
SPACE BAR TO PAUSE,RETURN TO END SONG
A, F, OR K = REST
```

Atari Player enables you to play music on your Atari. It converts the Atari into a simulated organ with the organ keys being represented by the keys on the Atari keyboard, as shown above. Each note you play can be heard over one of the Atari voices. The keyboard spans three complete octaves, beginning with a low C and ending more than two octaves higher with a B. Each note that is played is stored in memory so that it can be instantaneously replayed and then saved on cassette tape or disk. Later you can load your song in again and replay it.

We have included a feature in Atari Player that allows you to stop playing, go back to correct mistakes, replay the song from the beginning to the current note, and then continue playing additional notes. Absolute perfection is possible using the editing option (number 5), which allows you complete control over every note that has been played. You can change the tone, octave, and/or duration of any of the notes; or, if you wish, you can add more notes or remove them altogether.

With the Atari Player installed in your computer you can make your Atari a musically instructive as well as entertaining device.

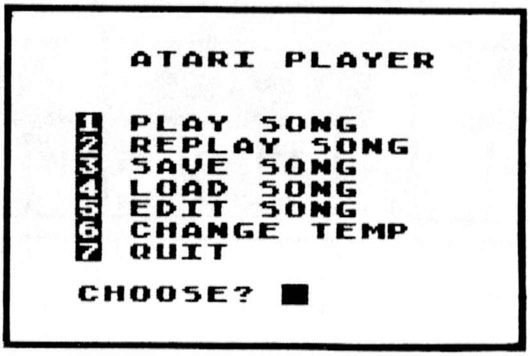
Operating Instructions for Atari Player Demonstration

The first time you use the Atari Player, you should listen to the song already provided on your disk to appreciate how much can be accomplished. To do this, follow the simple steps listed here.

1. Using standard Atari loading procedures, load Atari PLAYER.

2. Type 'RUN' followed by the 'RETURN' key.

3. You will be presented with the information that appears on the display screen below.



```
          ATARI  PLAYER

      1  PLAY  SONG
      2  REPLAY  SONG
      3  SAVE  SONG
      4  LOAD  SONG
      5  EDIT  SONG
      6  CHANGE  TEMP
      7  QUIT

      CHOOSE? █
```

4. Press the 4 key to select the 'LOAD SONG' option.

5. The display will ask for the file name under which the song is stored.

6. Type 'D:SONG' followed by the 'RETURN'. This will cause the song already provided in the file 'D:SONG' to be loaded into your Atari.

7. When the song has been loaded, Atari Player will tell you how many notes have been loaded, wait a few seconds, and then return to the original menu.

8. Press the 2 key to select the 'REPLAY SONG' option.

9. You will now be treated to a rousing song played automatically on your Atari computer.

Using the Atari Player

After choosing the 'PLAY SONG' option from the menu, the program will prompt you for the name of your song. (This is not the same thing as the file name that disk users have to supply when performing a save or a load. The name is saved with the song just for convenience and has no bearing on the song itself.) Then a representation of the Atari keyboard is printed on the screen in the format of a two-keyboard organ. The bottom row of keys represents the lowest notes, starting with 'C' and ascending alphabetically. The second row of keys represents the sharps and flats (black keys) that correspond to the first row. Note that there are inbetween keys on the second row of the Atari for every pair of first-row notes. This is different than the normal organ keyboard and means that some of the second-row keys do not sound when pressed (A, F, and K). These keys can be used to introduce rests into your song.

The third row of keys represents the second keyboard of the organ, starting at middle 'F' and ascending to high 'B'. The top row of keys represents the sharps and flats corresponding to this second keyboard. When you have mastered the keyboard, you are well on your way to com-posing your own music. Read on.

At the beginning, the program waits for you to start the song. This is one of the few times when a pause doesn't count. Once you start playing, the computer keeps track of every note and its length exactly as you play it. Practice a little bit to get the feel of the keyboard. It is not as simple as a piano, especially with the letters on the keys distracting you from what the true note is. The white keys on the display have the actual name of the note printed over the keyboard name of the key to help keep you oriented.

After you start a song, you may discover that you didn't mean to play a particular note. Fortunately there is a mistake-recovery method. As soon as you realize that you have made an error (sometimes the first note is an error), press the space bar to pause momentarily. You will be presented with several options.

1. CONTINUE allows you to start playing the song at exactly the point

where you stopped. This is a useful technique for the times when you become confused as to which note you want to play next; press the space bar to pause, regather your wits, and press 'C' to continue from where you stopped.

2. REPLAY will play the song up through the current note so that you can inspect your masterpiece as you input and make corrections if necessary. This option can be chosen as many times as you need it.

3. BACKUP is the option for which you've been waiting. This allows you to remove one note at a time from the current song, all the way back to the beginning if you want. When you make a mistake and press the space bar to pause, press the 'B' option and the note you are erasing will sound. Another 'B' will erase the next note, and so on. Then pressing 'C' will allow you to continue your song from the point to which you have backed up.

If, no matter how hard you try, you can't seem to get the song perfect, then the next step is to use the editor (option 5), which will allow you to manipulate the more obscure aspects of your song.

Using the Editor

The Editor option enables you to correct any minor (or major) mistakes that may creep into your performance. It allows complete control over every note in your song; the note, octave, and duration can all be changed to your specifications. And when using the insertion feature, you can start from scratch and construct your own song without ever playing a note on the Player keyboard! The Editor option is the perfect complement to Atari Player.

The first thing that the editor does is to list the notes currently in memory in groups of 30. These are notes that have been either entered from the keyboard or loaded in from a previously saved song. The 'Next' and 'Previous' commands allow you to page forward and backward through the note tables so that you can look for the note (or notes) that you wish to change. The 'Current' command simply redisplays the table starting at the same note. This option is usually used after an insert, delete, or change to see how the song looks.

The '#' option allows you to change the current notes in memory, one at a time. Note that the editor is not expecting the number sign itself; it is expecting a number. Also, it does not change any notes on the tape or disk, so don't worry about making a mistake. Experiment with changing notes and then play them together to see how they sound.

Pressing the return at the 'NEW LENGTH:' prompt will return you to the main menu. Pressing return at any other prompt will leave the rest of the parts of the note unchanged and skip ahead to the next note. This allows you to change the lengths without re-entering the remaining values that you don't want to change.

Pressing the '/' and return at any prompt preserves that one value and jumps to the next prompt. For example, if you don't want to change a value, such as length, but do want to change the note or octave, press '/' and it will skip over the values you want to save without your having to retype them.

The insert option moves all the notes (including the one whose number you entered) forward one position and then returns you to the 'Next, Previous, Current, or #' prompt. You might want to use the 'Current' option now to see how the song looks. Change the note that was at the point of insertion to whatever specifications you want and you have just entered a note.

Delete moves all the notes following the one you specified down one position, effectively erasing that note. As with the insert option, you will be returned to the 'Next, Previous, Current, or #' prompt. Again, you may want to use the 'Current' option to see how the song looks after the deletion.

When you are prompted for a new note, the only notes that will be accepted are 'C', 'C#', etc., as listed in the data statement (line 13010). A rest is indicated by '[ ]'. The program looks up your note in the note table and converts it into the number that represents that particular note. If it can't find that note, it will give you the '?NOTE NOT FOUND.' error and return to the 'NEW NOTE' prompt to give you another chance to change it.

Other Menu Options

There are five additional menu options that allow you to hear your song, load a song from or save a song to the tape player, change the tempo of the song, or quit.

Choosing option 2, 'REPLAY SONG,' will play the song currently in memory over the television speaker. The routine uses the current tempo for the speed at which to play the song. If there are no notes in the song, then the error '?NO SONG IN MEMORY' is printed and you are returned to the menu.

The 'SAVE' and 'LOAD' options are numbered 3 and 4, respectively.

To save the current song, choose option 3 and enter the file name of the song to be stored. If you are using a cassette you can specify only 'C:' as your file name. If you are using a disk, the file name must be a standard disk file name. Note that this file name and the name of the song you specify when you choose option 1 (PLAY SONG) are not the same. The name refers to the name that you wish the song to be titled — for the sake of convenience only. The file name is merely where you want the song to be stored and applies only to disk users. If you are using a cassette-based system you will be prompted to press PLAY and RECORD on the tape player. When the song is saved, the number of notes and the file in which they are saved are displayed and you are returned to the menu. If you are using the tape player, it will stop.

Option 4 is similar in operation to option 3. Remember that when you LOAD a new song, you will erase any song currently in memory; you must SAVE the current song (if you want to keep it for later use) before loading a new one. The file name prompt will now appear. (If you are using a cassette, there are no file names that you can specify. The tape will read from where it was left last.) When the song is loaded, as when saving, the number of notes and the file they were in are displayed and you are returned to the menu. Again, if applicable, the tape player will stop.

To change the current tempo setting, choose option 5. The minimum (fastest) setting allowed is '1'. There is no restriction on the maximum (slowest) setting, except for the fairly large internal limits of the Atari. However, even a setting of just 100 will result in extremely long notes.

When you are finished with Atari Player and want to return to BASIC, choose option 6. Always remember to save any song that you are currently working on before choosing this option. If you forget this rule, typing 'GOTO20' *might* enable you to return to the program without losing your song.

Programming Concepts

Using a Menu to Make a Choice

1. Input with a Single Keystroke

Normally input from devices is done with the INPUT statement. The only limitation of this statement is that it requires the input data to be followed by a carriage return. Thus, any input from the keyboard using this method would require at least two keystokes — one for the character being input and one for the return key. The way around this is illustrated in program lines 30 and 50.

Line 30 opens the keyboard for input. This means exactly what it seems: you can now input from the keyboard. The open statement is usually used for access to files on cassette or disk, but it can be used for any connected device, such as the keyboard and even the screen. The device specification for the keyboard is 'K:', which is the open statement's fourth parameter.

Because a single keystroke input is preferred, the keyboard input for the menu (line 50) is done with a GET. The GET statement waits for one character of data from the keyboard; i.e., the single keystroke. Thus no carriage return is required and an option may be selected by striking a single key. However, the data input is not a character, or a letter, but rather a number. This number is the numerical representation of the typed letter and is called an ASCII code.

2. ASCII Codes

Because computers cannot handle anything other than numbers, a method for handling characters such as punctuation marks, letters of the alphabet, or any other character, is needed. What has been devised is the ASCII codes (American Standard Code for Information Interchange), which are numerical representations of each character for the computer to handle internally. Most users never see these numbers for they are converted back to characters when displayed on the screen, but they are neccessary when dealing with such specific statements as the Atari GET.

3. Using a Menu to Make a Choice: Selection by Number

When you run Atari Player, the first display that you see is a list (or menu) that tells you what options are available. Each item on the menu is selected by pressing the number associated with it. The BASIC program steps required to evaluate your choice are in lines 50 and 60.

```
50 GET #1,A:IF A<49 OR A>55 THEN 50
60 A=A-48:GOSUB A*1000:GOTO 40
```

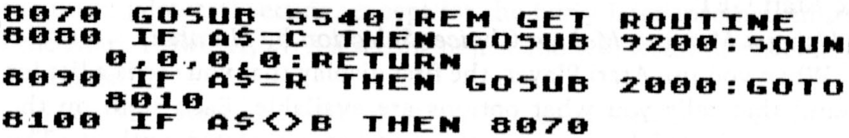
Line 50 gets the ASCII code for the keyboard character pressed and stores it in variable A. The following IF...THEN statement checks to see whether or not the character typed was a number. Note that, as stated above, the GET statement gets the internal code for the character typed rather than the character itself. This means that the program has to check for the characters 1-9 rather than the numbers 1-9. If A is less than 49 (ASCII 1) or greater than 55 (ASCII 7), then the character typed is ignored and the program returns to line 30 to get another choice from the keyboard.

Line 60 does the actual branching to the appropriate routine. Since the only options available are listed 1 through 7, and the program routines for each start on the line numbers that are multiples of a thousand (1000-7000), the branching to these routines can be done by formulae. All that is needed to branch to the appropriate routine is to multiply the option number chosen by 1000 and then perform a GOSUB to that result. Thus, 48 (ASCII 0) is subtracted from the code in A to convert it to one of the numbers 1-7. Then a GOSUB to the line number A * 1000 is performed. This is a useful space conservation technique. The only other way to do this would be to have a fairly large ON...GOSUB statement followed by the starting line number of every subroutine.

4. Selecting by First Letter

If the space bar is pressed during the 'PLAY SONG,' then another menu is displayed.

The first character of each item is displayed in reverse video to indicate which letter is to be pressed on the keyboard to select that choice. The routine that gets the characters from the keyboard is at line 5540. Note that this routine converts the number put in variable A into a string (A$) *via* the CHR$ command. This is done so that the routine will return a string holding the choice as opposed to returning the ASCII value of the choice. That is, this routine returns a letter, which is then serviced by its own IF...THEN statement.



```
8070 GOSUB 5540:REM GET ROUTINE
8080 IF A$=C THEN GOSUB 9200:SOUND
     0,0,0,0:RETURN
8090 IF A$=R THEN GOSUB 2000:GOTO
     8010
8100 IF A$<>B THEN 8070
```

If there are only a few choices, as in this example, then there is not a lot of code. If there were many choices, then the amount of code to service the letters could be significant.

5. Accessing the Keyboard for Real-Time Situations

The above method of selecting by first letter using the GET statement is fine for most situations involving input with a single keystroke, but it is severely limited in real-time applications. This is because the GET statement actually waits for a key to be struck. For instance, suppose you were writing a game that needed to keep track of the elapsed time between keystrokes. This would be impossible with the GET statement because the loop required to keep track of the time could never be executed. Such was the problem in writing the PLAY SONG option, which has to keep track of such things.

The Atari uses memory location 764 as the place to keep the last key hit. That is, whenever a key is struck (no matter what program is running) location 764 is filled with a number that represents the last key pressed. But this number is *not* ASCII. This location uses the actual hardwired codes for the keyboard — a different set of codes entirely. For the beginner, using this location is rather difficult as the codes follow no obvious pattern. For instance, a space is a 33, the letter A is 63, the B is 21, and so on. Therefore, the user must know which keys to expect ahead of time. The following short routine can be used to discover the various key codes. After typing RUN, simply hit a few keys; their corresponding codes will come up on the screen.

```
10 PRINT PEEK(764):GOTO 10
```

Lines 1030, 1060, and 1070 show how Atari Player uses this location. Note that it is POKEd with 255 after being read. This insures that the same key can be hit twice and registered as two keys. That is, because the location holds the last key hit, if you were to hit, say, the 'Z' key twice, and if the location was not reset after each read, it simply would not change from the first 'Z' to the next. This means that two notes would be read as one (with a rather long duration). Resetting location 764 also allows us to keep track of the duration in the first place (line 1030). By checking for the 255, we know that the key has not yet been hit and that the duration should then be incremented.

Space Saving with Arrays

Atari Player uses many arrays — for the storage of its notes, keyboard entries, and other such internal data. There are many ways of storing this information in the arrays. This section deals with the most economical way possible for storing the types of information contained in Atari Player.

1. Numeric Arrays

Atari Player could have stored all of its information in the form of numerical arrays as all of its information is, indeed, numbers. But numbers are very costly to store in BASIC arrays, especially on the Atari. Atari BASIC 'eats' up 15 bytes plus 6 bytes for every cell in the array. This is not including the space that is eaten up by the variable name and the DIM statement itself. For example, an array that was dimensioned for 100 cells would eat up 15 plus 6 times 100 — or 615 bytes! Therefore, the only array that we made numerical was W, and it was dimensioned for a whopping 501 cells. Over 3000 bytes is used to store it.

However, things could be worse. The W array has a little more information than normal crammed into it. In fact, that one array is made up of both the keystroke (for the note played) and the length of time that it is played. It was necessary to perform such a numerical combination to keep from wasting an exorbitant amount of memory. But before we get into such a space-saving trick, let's first examine how *not* to program the storage area.

As mentioned briefly before, each note that you play involves two pieces of information that must be saved by Atari Player: the number (keystroke) of the note and the duration of the note.

We could define a two-dimensional array that contains a number for both parts of each note: e.g., DIM W(500,1) would reserve space for 500 notes (the zero*th* element is not used) with two numbers per note. How much memory do you think this would take? It really is deceiving. Since 500 times 2 is 1000, is that the total number of bytes required? No! Each Atari BASIC array number requires six bytes of memory. Therefore, it would take 6 * 1000, or 6000, bytes of memory! On a 16K Atari (which, by the way, has only 13000 some bytes left after the operating system has taken its share) that leaves a little over 7K for the rest of the program.

You have to get a bit tricky to squeeze more out of the song space; but there is nothing wrong with getting tricky when writing programs. In fact, that can be half the fun! To really squeeze the memory in Atari Player, we took advantage of the size of the number that the Atari arrays can hold. The two values that we need to keep for each note played are the note number (0 to 38) and the duration (which is limited to 0 to 999 units). If only we could pack both of these individual values into a single number for storage and then unpack them when we needed to use them. Well, good news — we can!

The technique to pack the numbers is shown in line 5392.

```
5392  GOSUB 9500:REM PACK IT INTO W(V)
              .
              .
              .
9500  IF D>999 THEN D=999
9510  W(V)=INT(TP*1000+D):RETURN
```

This equation is not as difficult as it may at first appear.

W%(V) is the array of notes where V is the number of the note in the song

+TP * 1000 multiplies the keystroke number (0 to 63) by 1000

+D adds the duration value (0 to 999)

All we have done is multiply the keystroke value that we need to save by enough to make sure it does not overlap with the duration value. This insures that we will be able to unpack the separate parts later. The unpacking is a bit more difficult than the packing, but conceptually it is simple. All we need to do is reverse the packing process. This is accomplished in the following lines:

```
9200 TP=INT(W(V)/1000)
9210 D=W(V)-TP*1000:RETURN
```

Line 9200 restores the note number by dividing by 1000.

Line 9210 restores the duration by subtracting the note number multiplied by 1000. This simply returns the remainder that was sheared off by the INT 1000.

The above lines of program take the single integer value and convert it back into two separate parts. This method results in a 500-note song requiring only 501 array slots to store it, at six bytes per number, for a total storage of just over 3000 bytes — only half the amount that would be needed if we stored the two values separately!

2. Simulating String Arrays

NT$ is a string that holds all the possible note symbols. Normally these symbols would be entered separately in a string array, but the Atari does not have this capability. (It does dimension strings, but this refers only to the lengths of the strings and that doesn't help us much.) Instead, all string information must be concatenated, or 'strung together,' one after another in a single string. This means that the only way to access the different elements is to access the different character locations within the string. As long as the elements are of equal size this is fairly simple because the starting character position of any of the elements can be calculated using the index. Take for example the following lines:

```
10 DIM NAME$(30)
20 NAME$="TOM  DICK HARRYJIMBO
   RALPHBILLY"
100 INPUT INDEX: REM ---INDEX
   STARTS AT 0---
110 PRINT NAME$(INDEX*5+1,INDEX
   *5+5)
120 GOTO 100
```

This routine simulates a string array where the elements are first names. In Atari BASIC, to get part of a string, you specify the string name followed by the starting and ending character position of the substring you want. In line 110, what is printed is the substring that starts with character position INDEX*5+1 and ends with

INDEX*5 + 5. Thus, because the character position is calculated, every element must be the same length. If they aren't, and this is usually the case, then the smaller ones must be padded with spaces. Every element must start at every character position that is a multiple of some number in order for this particular technique to work.

The note symbols used by Atari Player are stored the same way. They are stored initially in line 13010 and then READ in as a single string (NT$) in the initialization routine at lines 12000 through 12050. Assuming Y is the note number (0-12), the following routine will set N$ equal to the corresponding note symbol:

```
10  N$ = NT$(Y*2 + 1,Y*2 + 2)
```

In fact, this very routine is found at the end of line 5062. Note that the elements in this case must be two characters long: a note, 'C' for instance, followed by either a space or a sharp sign, '#'.

There is another way that strings are used in Atari Player. It is a bit more complex but basically entails the same concepts. The difference lies in the fact that this method is used to store a series of numeric values such as would normally be done with numeric arrays. The savings are space and time and for the following reasons.

Strings use up only one byte of memory per character. A little more is used up in the variable name and such, but that is essentially it. So, if we were to somehow convert every number that is in a numeric array (six bytes) to a string character (one byte), we would have a savings of five bytes per element. Thus, a numeric array dimensioned for 100 elements that would normally eat up 600 bytes of storage could be made to fit in merely 100 bytes.

Loading in the initial values in a string is very fast, whereas loading in the initial values in an array is tediously slow. This is because string loading can be done with a single statement and all the elements can be loaded in at once; the elements in an array must be loaded in one after another in a loop. For arrays of ten elements or less, conversion to a string is hardly worth the effort because the loading is so short and, therefore, does not take up much time anyway. But for larger arrays, string conversion is definitely worthwhile because the loading time of even several minutes can be cut down to only a few seconds at most.

Lines 12042, 13000, and 13020 demonstrate how the strings are stored and read in.

```
12000 S=10:LN=500:DS=10:L=8:CF=752
12010 DIM W(LN),PITCH$(63),NT$(26)
     ,F$(14),B$(20),N$(3),A$(3),NTO
     CT$(63)
```

```
12020 RESTORE :B$="NONAME"
12042 READ PITCH$,NT$,NTOCT$
12050 RETURN
13000 DATA r◤f♥♥♥♥♥♥(♥#/♥—→♥6♥A♥♥▀Y
      SD♥L9♥♥♥U♥y♥1E♥C◆♥@♥H5♥<Q[*♥%2♥
      ♥!♥♥ L♥♥-f
13010 DATA C C#D D#E F F#G G#A A#B
      []
13020 DATA ▁◢♀▀ ♥▀ ♥♥!♥#←♥→%♥┐ ♥┤ ♥♥\ |
      ♥┬♥▄E♥♥┼♥‾♥▗ ▖♥▝┌♥┴♥▌ ↑♥└●— ♥''↓♥
      ♥$♥▀ ◢┘ ♥♥/├
```

There are a couple of problems when working with these strings. First, when addressing the element you must, in reality, address the individual characters of the string. That is, the index that you would normally use to access any of the elements in the numeric arrays is now the character position in the string. But element positions in arrays start at zero (0) while string positions start at one (1). Therefore, when accessing the data in strings, you must either ignore the zero*th* element entirely and start the element numbers off at one, or add one to the element number so that the element that was previously numbered zero will reside at string position one (1). Line 9800 demonstrates how this latter method is used in Atari Player.

```
9800 SOUND 0,ASC(PITCH$(TP+1,TP+1)),
   DS,L
```

where TP is the index to the string array PITCH$.

The second problem is a little tougher for the beginner to understand. Since each character in a string is essentially one byte of storage internally, the size of the numbers you can store are limited to what a byte can store. Because, as you may know, bytes are only eight bits long, the largest number that could possibly be stored in one is 11111111 binary (255 decimal). Also, the number must be a non-negative integer. To get the number in the string as an element it must be converted to a character *via* the CHR$ function. This function will turn the number specified into the character whose ASCII value is that number so that it may be stored in a string. When accessing that element, to get the number from the character, the ASC function must be used. This function converts the string specified (presumably an element in the string) to its ASCII equivalent. (See Programming Concepts 2: ASCII Codes.)

The Program

The Atari Player program contains seven major functions (which are selected from a menu), three minor functions used during the playing of a song from the keyboard, seven minor functions used during the editing of a song, and some support subroutines.

The main program (lines 10 - 50) calls subroutines to initialize the program, turn off the sound generators, and print the menu on the display. It waits for a character in the range of the menu (1 - 7) then goes to the appropriate subroutine to service the selected function.

The PLAY SONG subroutine (lines 1000 - 1120) gets the user's song name, calls a subroutine to print the music keyboard, gets input from the keyboard, calls a subroutine to pack information about the current note and store it in memory, and then, depending on which key you press, does one of the following things:

> A *Carriage Return* terminates the song by putting a 0 in the next song location and returning to the main menu
>
> A *Space* goes to the 'Continue, Replay, Backup' subroutine
>
> The *A*, *F*, and *K* keys are converted to a 'musical rest'
>
> A *defined key* (i.e., a 'music key') is converted to its sound generator.

The REPLAY SONG subroutine (lines 2000 - 2070) prints the message 'PLAYING' and the song title, if there is one. If there is no song in memory, it prints the error message '?NO SONG IN MEMORY', goes to a subroutine that produces a delay to enable the message to be read, and then returns to the main program.

If there is a song in memory, REPLAY calls a subroutine to unpack the keystroke corresponding to the appropriate note and to the duration of that note. It uses a subroutine to output the note to the sound generator for the specified duration. The sound generator is then turned off and the song pointer (variable V) is incremented to the next note. If the maximum song length possible (LN) has been exceeded, then a return is made to the main program; otherwise the keyboard is checked and if any key is pressed a return is made to the main program.

If no key is pressed, the next note of the song is unpacked and tested. If it is not the 'end of song' indicator (a zero value) the program continues playing the song. At the end of the song, REPLAY uses a subroutine to generate a brief delay and then goes to the main program.

The SAVE SONG subroutine (lines 3000 - 3030) uses a subroutine to print the message 'Enter filename...?' and inputs a file name. If 'C:' is specified it opens the cassette device for saving information, asks you whether or not the tape is ready, and outputs the song information one note at a time. If 'D:' is specified, it opens the specified disk file and outputs the song information. When it has output the last note it closes the output device, tells you how much was saved, and exits through the delay routine.

The LOAD SONG subroutine (lines 4000 - 4030) uses a subroutine to print the message 'Enter filename...?' and inputs a file name. It opens the appropriate device for loading information and inputs the song information one note at a time. When it detects an empty note it closes the device, tells you how much was loaded, and exits through the delay routine.

The EDIT SONG subroutine (lines 5000 - 5550) outputs the data of the first 30 notes in the song. It then asks you for the next, previous, or current group of 30 notes you wish displayed, or the number of the line you wish to edit. If just a carriage return is typed, the program returns to the main program (menu). If the ASCII value of the input response is not that of a number, then it goes to line 5430 where it branches to the appropriate routines. If the ASCII value is one of a number, then it asks you if you wish to change, delete, or insert a note at that point. If 'C' is chosen, it asks you for the values of the three parts to the note, and then it continues on with the next note until you hit a carriage return at the 'NEW LENGTH?' prompt. If 'I' or 'D' is chosen, then the program shuffles the entire array up or down one element, respectively, to either make room for the inserted note or obliterate the one already there.

The CHANGE TEMPO subroutine (lines 6000 - 6030) prints the current value of the tempo variable, then requests and inputs a new value. If the value is less than 1, it is ignored. When a valid value is input, the routine returns to the main program.

The QUIT routine (line 7000) enters graphics mode 0, thereby clearing the screen and setting all the colors back to normal, and then it executes an END to return to BASIC.

The PLAY SONG MENU routine (lines 8000 - 8120) provides three additional commands for use while playing a song. When called by pressing the space bar, it first uses a subroutine to turn off all sound generators and another to unpack the current note. It prints its own menu and waits for a keyboard selection. Upon receiving a 'C' it backs up the song pointer to the current note, unpacks the note, and turns off the sound generators before returning to the PLAY SONG routine.

The letter 'R' calls the REPLAY SONG subroutine, which plays the current song from the beginning to the current note, and then waits for another menu selection.

The letter 'B' causes the program to go to the subroutines to unpack the current note's values, output the note for the correct duration, and turn off the sound generators when the note is done. Then it backs up over the note played by subtracting one from the note pointer. If, however, the note pointer is at the beginning of the song, the entire routine does nothing. When it is done, it waits for another menu selection.

The support subroutines (lines 9000 - 13020) provide support for the main program and major subroutines.

Line 9000 provides a several-second delay to permit you to view messages that appear before clearing.

Line 9100 prints 'Enter filename...?' and accepts a name from the keyboard. If 'C:' is specified, you are prompted to hit return when the tape is ready.

Line 9200 unpacks the stored note information into its two components: the keystroke/note equivalent 'TP' and the duration of the note 'D'. See packing information in the section headed "Numeric Variables."

Line 9500 makes sure that the note duration is not greater than 999 and packs the two components of the note (the note/keystroke equivalent 'TP' and the duration of the note 'D') into a single integer value in the song array. See packing information in the section headed "Numeric Variables."

Line 9800 outputs a note by converting the keystroke number (TP) into a pitch for the SOUND statement. This is done by using TP as an index into the string array 'PITCH$'. Note that this is not the standard BASIC string array, as the Atari simply does not have that capability. (See Space Saving with Arrays 2: Simulating String Arrays.) After sounding out the note, the routine then waits for the duration of the note, which is calculated as the tempo 'S' times the note duration 'D' divided by 8 times the time it takes the BASIC FOR...NEXT loop to execute.

Line 10000 prints the main menu.

Line 12000 performs a series of initialization functions. It sets the tempo 'S' to 10 and the length of the song 'LN' to 500 notes; it sets the distortion 'DS' used by the SOUND statement to 10 (pure tones); it sets the loudness 'L' also used by the SOUND statement to 8, which is moderately loud; it dimensions a numeric array (W%(LN) to hold the song note information); it dimensions six strings (PITCH$ associates the keyboard characters (i.e., keystrokes) with the pitches needed by the SOUND statement; NT$ is a series of the 12 possible two-letter note symbols plus '[ ]' for rests; F$, B$, N$, A$ are just utility strings used by the program; and NTOCT$ associates the keystroke with a note and its octave). It also assigns B$ to the value of 'NONAME' as a song name default, reads in PITCH$, NT$, and NTOCT$, and then performs a RETURN.

Line 13000: This is the string value for PITCH$. It is a series of pitches for the Atari SOUND command that corresponds to the keystroke number, which is used as an index. That is, once the keystroke number is 'gotten' from the keyboard, it is used as a character position in the string. The ASCII value of the character at that position is the pitch for the SOUND statement.

Line 13020: This is the string value for NTOCT$. It holds the note number (0-12) and the octave. The accessing of its elements is done with character position indexes, the same way it is done in line 13000.

Line 14000: This is the routine that outputs the keyboard. The keyboard was drawn using the Atari graphics characters, and can be used on the normal mode 0 text screen. Earlier versions of Atari Player actually drew the keyboard in one of the graphics modes, but this method took up too much memory. Since the graphics characters can be used in mode 0, they take up extra memory.

You'll notice that in line 14000 a POKE is made to the location held in the constant CF. This is the cursor inhibit location and when it contains a non-zero value, the cursor display is suppressed. When it holds a 0, the cursor is displayed as the reverse of whatever character it is on. Since the cursor is usually on nothing other than blank screen, a solid white square (reverse-blank) is usually displayed while output to the screen is done. Since this is not wanted when the keyboard is being displayed, the cursor is disabled. In fact, during most of the displays in this program, the cursor is not wanted, thus providing a sharper and generally better-looking display. The user should experiment with how the displays look with and without the cursor.

Atari Player Variable Usage

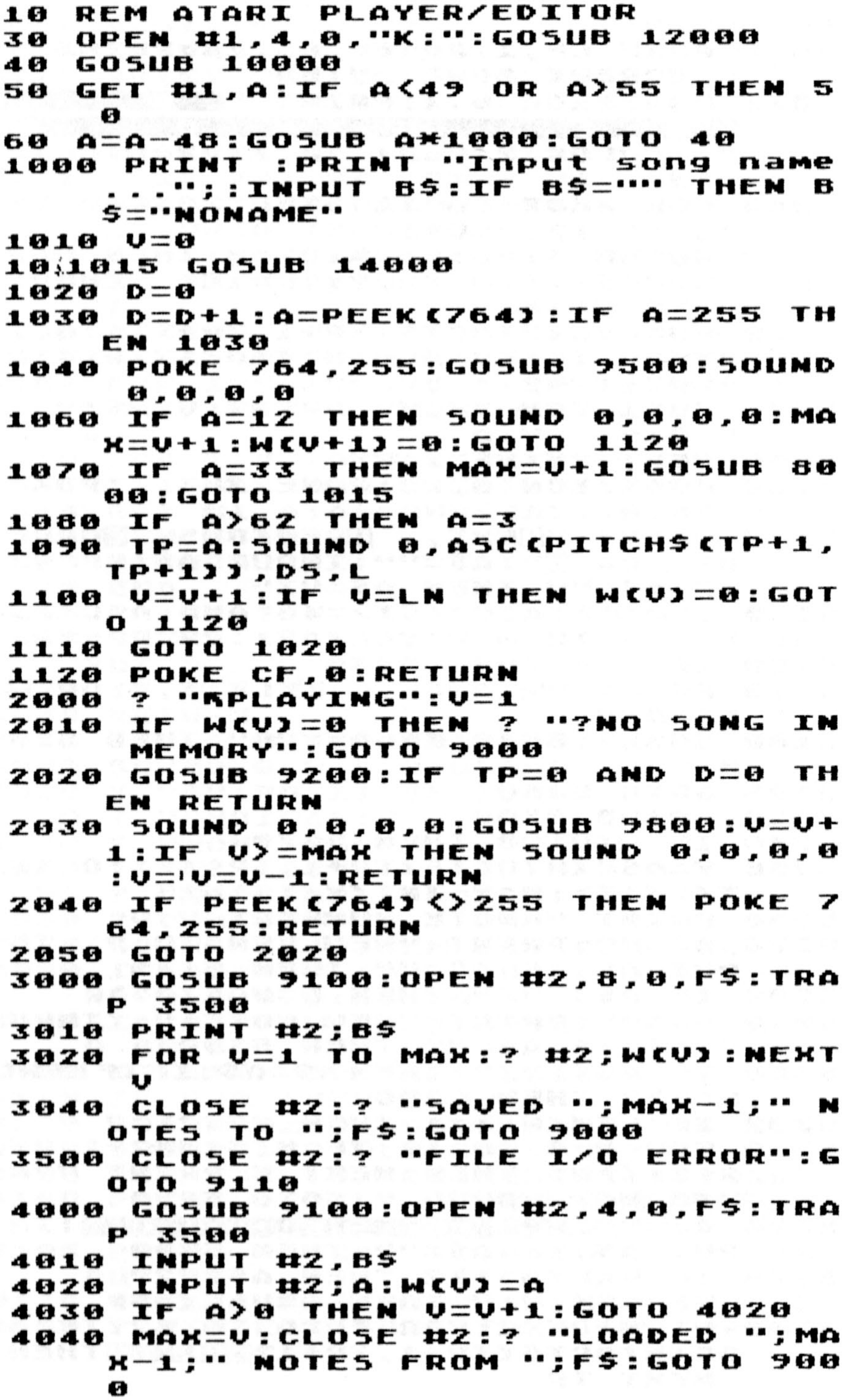
Constants

CF	POKE location for cursor inhibit
DS	Sound distortion
L	Sound loudness
LN	Last element available in W array
NT$	String 'array' of all note symbols
NTOCT$	Note/octave conversion from keystroke number
PITCH$	String 'array' of sound pitches for conversion from keystroke

Variables

A	ASCII value from keyboard
D	Duration for current note
I	Loop counter
MAX	Position of the zero element in the song (end of song)
N	Note number (0-12)
NT	Actual note (0-12) plus octave (0-2)
O	Note's octave
S	Tempo setting (speed of song)
TP	Keystroke/note equivalent
V	Note pointer for song
W( )	Array of notes and durations
XX	X position of editor's output
YY	Y position of editor's output
A$	Input string from keyboard
B$	Song name
F$	File name
N$	Note symbol

Listing 1: **Atari Player/Editor**



```
10 REM ATARI PLAYER/EDITOR
30 OPEN #1,4,0,"K:":GOSUB 12000
40 GOSUB 10000
50 GET #1,A:IF A<49 OR A>55 THEN 5
   0
60 A=A-48:GOSUB A*1000:GOTO 40
1000 PRINT :PRINT "Input song name
     ...";:INPUT B$:IF B$="" THEN B
     $="NONAME"
1010 V=0
10/1015 GOSUB 14000
1020 D=0
1030 D=D+1:A=PEEK(764):IF A=255 TH
     EN 1030
1040 POKE 764,255:GOSUB 9500:SOUND
      0,0,0,0
1060 IF A=12 THEN SOUND 0,0,0,0:MA
     X=V+1:W(V+1)=0:GOTO 1120
1070 IF A=33 THEN MAX=V+1:GOSUB 80
     00:GOTO 1015
1080 IF A>62 THEN A=3
1090 TP=A:SOUND 0,ASC(PITCH$(TP+1,
     TP+1)),DS,L
1100 V=V+1:IF V=LN THEN W(V)=0:GOT
     O 1120
1110 GOTO 1020
1120 POKE CF,0:RETURN
2000 ? "↖PLAYING":V=1
2010 IF W(V)=0 THEN ? "?NO SONG IN
      MEMORY":GOTO 9000
2020 GOSUB 9200:IF TP=0 AND D=0 TH
     EN RETURN
2030 SOUND 0,0,0,0:GOSUB 9800:V=V+
     1:IF V>=MAX THEN SOUND 0,0,0,0
     :V='V=V-1:RETURN
2040 IF PEEK(764)<>255 THEN POKE 7
     64,255:RETURN
2050 GOTO 2020
3000 GOSUB 9100:OPEN #2,8,0,F$:TRA
     P 3500
3010 PRINT #2;B$
3020 FOR V=1 TO MAX:? #2;W(V):NEXT
      V
3040 CLOSE #2:? "SAVED ";MAX-1;" N
     OTES TO ";F$:GOTO 9000
3500 CLOSE #2:? "FILE I/O ERROR":G
     OTO 9110
4000 GOSUB 9100:OPEN #2,4,0,F$:TRA
     P 3500
4010 INPUT #2,B$
4020 INPUT #2;A:W(V)=A
4030 IF A>0 THEN V=V+1:GOTO 4020
4040 MAX=V:CLOSE #2:? "LOADED ";MA
     X-1;" NOTES FROM ";F$:GOTO 900
     0
```

```
5000 POKE 201,5:POKE CF,1:POKE 82,
     0
5020 V=0
5030 POKE CF,1:POKE 703,24:PRINT "
     ↖CURRENT SONG: ";B$
5032 POSITION 0,1:PRINT "  #    N 
       O   L       #    N   O   L   
     ";:COLOR 2:PLOT 18,2:DRAWTO 18
     ,19
5040 FOR XX=2 TO 22 STEP 20:FOR YY
     =3 TO 17:V=V+1
5060 GOSUB 9200:IF W(V)=0 THEN MAX
     =V:POSITION XX,YY:PRINT "END":
     GOTO 5130
5062 NO=ASC(NTOCT$(TP+1,TP+1)):O=I
     NT(NO/13):Y=NO-INT(NO/13)*13:N
     $=NT$(Y*2+1,Y*2+2)
5080 POSITION XX,YY:PRINT V,N$;" 
     ";O;"    ";D
5090 NEXT YY:NEXT XX
5130 POSITION 0,20:POKE 703,4:POKE
      CF,0
5140 PRINT "NEXT, PREVIOUS, CURRE
     NT, OR X":A$="":INPUT A$:PP=V:
     IF A$="" THEN A$="Z"
5150 V=0:IF ASC(A$)>=48 AND ASC(A$
     )<=57 THEN V=VAL(A$):PP=PP-30
5160 IF V=0 THEN 5430
5170 PRINT "CHANGE, DELETE, OR I
     NSERT"
5180 GOSUB 5540:IF A$="C" THEN 520
     0
5190 GOTO 5460
5200 GOSUB 9200
5240 IF W(V)=0 THEN 5270
5250 Y=ASC(NTOCT$(TP+1,TP+1)):O=IN
     T(Y/13):N=Y-INT(Y/13)*13
5260 PRINT "↖NOTE NUMBER: ";V
5270 A$="":PRINT "NEW LENGTH:",:IN
     PUT A$:IF A$="" THEN 5140
5280 IF A$<>"/" THEN D=VAL(A$)
5290 A$="":PRINT "NEW NOTE:",:INPU
     T A$:IF A$="" THEN 5390
5300 IF A$<>"/" THENN$=A$:IFIFLEN(
     N$)>2 THEN 5290
5302 IF LEN(N$)=1 THEN N$(2)=" "
5310 FOR N=0 TO 12:IF NT$(N*2+1,N*
     2+2)<>N$ THEN NEXT N:PRINT "?N
     OTE NOT FOUND.":GOTO 5290
5320 A$="":PRINT "NEW OCTAVE:",:IN
     PUT A$:IF A$="" THEN 5390
5325 IF VAL(A$)>2 THEN A$="2"
5330 IF A$<>"/" THEN O=VAL(A$)
5390 NT=N+O*12:FOR TP=0 TO 62:IF A
     SC(NTOCT$(TP+1,TP+1))<>NT THEN
      NEXT TP
```

```
5392 GOSUB 9500:REM PACK IT INTO W
     (V)
5400 V=V+1:GOTO 5200
5415 IF V<0 THEN V=0
5420 PRINT "↖":GOTO 5030
5430 IF A$="P" THEN V=PP-60:GOTO 5
     415
5440 IF A$<>"N" THEN 5450
5442 POKE 703,24:PRINT "↖":IF PP+2
     9>MAX THEN PP=MAX-30
5443 IF PP<0 THEN PP=0
5444 V=PP:GOTO 5030
5450 IF A$="C" THEN V=PP-30:GOTO 5
     415
5458 RETURN
5460 IF A$="D" THEN 5510
5480 FOR I=MAX TO V STEP -1:W(I+1)
     =W(I):NEXT I:MAX=MAX+1:GOTO 51
     40
5510 FOR I=V TO MAX:W(I)=W(I+1):NE
     XT I:MAX=MAX-1:GOTO 5140
5540 GET #1,A:A$=CHR$(A):IF A$=""
     THEN 5540
5550 RETURN
5999 END
6000 ?? :? :?"           TEMPO = ";S
6010 ? "          NEW TEMPO = ";:IN
     PUT S
6020 IF S<1 THEN 6010
6030 RETURN
7000 CLOSE #1:GRAPHICS 0:END
8000 SOUND 0,0,0,0:GOSUB 9200
8010 ? "↖":POSITION 3,3
8020 ?? "CHOICES:":? :?
8030 ? "CONTINUE"
8040 ? "REPLAY"
8050 ? "BACK UP 1 NOTE":?
8060 ? "WHICH? ";
8070 GOSUB 5540:REM GET ROUTINE
8080 IF A$="C" THEN GOSUB 9200:SOU
     ND 0,0,0,0:RETURN
8090 IF A$="R" THEN GOSUB 2000:GOT
     O 8010
8100 IF A$<>"B" THEN 8070
8110 IF V=0 THEN 8070
8120 GOSUB 9200:GOSUB 9800:V=V-1
8130 SOUND 0,0,0,0:GOTO 8070
9000 FOR I=1 TO 300:NEXT I:RETURN
9100 PRINT :PRINT "C: for cassette
     , D:filename for   disk"
9105 PRINT "Enter filename...";:IN
     PUT F$
9110 V=1:IF F$<>"C:" THEN RETURN
9150 ? "INSERT SONG TAPE AMD PRESS
     RETURN"
```

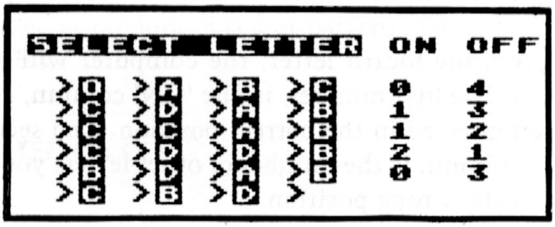
```
9160 GET #1,A:? :POKE CF,0:RETURN
9200 TP=INT(W(V)/1000)
9210 D=W(V)-TP*1000:RETURN
9500 IF D>999 THEN D=999
9510 W(V)=INT(TP*1000+D):RETURN
9800 SOUND 0,ASC(PITCH$(TP+1,TP+1)
    ),DS,L
9810 FOR I=1 TO S*D/8:NEXT I:RETUR
    N
10000 GRAPHICS 0:SOUND 0,0,0,0:POK
    E CF,1:SETCOLOR 1,0,0:SETCOLOR
     2,3,4:SETCOLOR 4,7,6
10030 ?"↖↓↓↓                ATARI PLAY
    ER":? :? :V=0
10040 ? "                   1 PLAY SONG"
10050 ? "                   2 REPLAY SONG"
10060 ? "                   3 SAVE SONG"
10070 ? "                   4 LOAD SONG"
10075 ? "                   5 EDIT SONG"
10080 ? "                   6 CHANGE TEMPO"
10090 ? "                   7 QUIT↓"
10100 POKE CF,0:? "            CHOOSE
    ? ";:RETURN
12000 S=10:LN=500:DS=10:L=8:CF=752
12010 DIM W(LN),PITCH$(63),NT$(26)
    ,F$(14),B$(20),N$(3),A$(3),NTO
    CT$(63)
12020 RESTORE :B$="NONAME"
12042 READ PITCH$,NT$,NTOCT$
12050 RETURN
13000 DATA r◤f♥♥♥♥♥♥(♥#/♥-→♥6♥A♥♥▀Y
    SD♥L9♥♥U♥y♥1▟♥▙♦♥@♥H5♥<Q[*♥%2♥
    ♥!♥♥ L♥♥▬f
13010 DATA C C#D D#E F F#G G#A A#B
     []
13020 DATA ▁◣♀▀ ♥▀ ♥♥!♥#←♥→%♥┐♥┤♥♥╲▕
    ♥┬♥▄E♥♥┼♥▔♥▖ ▖♥▝┌♥┴♥▌ ↑♥└●─ ♥"↓♥
    ♥$♥▀ ◢┘♥♥/├
14000 POKE 82,2:POKE CF,1:? "↖↓PLA
    Y WHEN READY    SONG:";B$:SETCO
    LOR 1,0,0:SETCOLOR 2,12,6
14010 ? "SPACE BAR TO PAUSE, RETUR
    N TO END SONG";
14020 ? "A, F, OR K = REST"
14030 ? "______________3______________
    ____________"
14040 ? "▌ ███ ███ ███ ▌ ███ ███ ▌ ██
    █ ███ ███ ▌"
14050 ? "▌ ███ ███ ███ ▌ ███ ███ ▌ ██
    █ ███ ███ ▌"
14060 ? "▌ █2█ █3█ █4█ ▌ █6█ █7█ ▌ █9
    █ █0█ █<█ ▌"
14070 ? "▌ ███ ███ ███ ▌ ███ ███ ▌ ██
    █ ███ ███ ▌"
```

```
14080 ? "|    |    |    |    |    |    |    |    |
     |    |    |"
14090 ? "|  F|  G|  A|  B|  C|  D|  E|  F|
    G|  A|  B|"
14100 ? "|    |    |    |    |    |    |    |    |
     |    |    |"
14110 ? "|  Q|  W|  E|  R|  T|  Y|  U|  I|
    O|  P|  -|"
14120 ? "___________________________________
______________"
14130 ? "  1____________________________2_
  ______________"
14140 ? "  |  ██ ██  |  ██ ██ ██  |
  ██ ██  |"
14150 ? "  |  ██ ██  |  ██ ██ ██  |
  ██ ██  |"
14160 ? "  |  █S█ █D█  |  █G█ █H█ █J█  |
  █L█ █;█  |"
14170 ? "  |  ██ ██  |  ██ ██ ██  |
  ██ ██  |"
14180 ? "  |    |    |    |    |    |    |    |
    |    |    |"
14190 ? "  |  C|  D|  E|  F|  G|  A|  B|
  C|  D|  E|"
14200 ? "  |    |    |    |    |    |    |    |
    |    |    |"
14210 ? "  |  Z|  X|  C|  V|  B|  N|  M|
   ,|   .|   /|"
14220 ? "  ______________________________
  ______________";:RETURN
```

ATARI

MASTER

A Simple Guessing Game



```
SELECT LETTER   ON  OFF

 >O  >A  >B  >C  0   4
 >C  >D  >A  >B  1   3
 >C  >C  >A  >B  1   2
 >C  >D  >D  >B  2   1
 >B  >D  >A  >B  0   3
 >C  >B  >D  >
```

MASTER is a simple guessing game for one or two players. The commercial version of this game involves colored pegs. One player constructs a pattern of four colored pegs behind a screen and it is up to the other player to guess the concealed pattern. The first player provides the second player with clues, telling him how many pegs have been guessed in the right position and how many pegs are the right color but in the wrong position. The second player continues to guess until he has discovered the colors and correct positions of all four pegs. The number of guesses is the score, and the player with the lower score wins. The Atari uses letters instead of pegs, but the rules are the same. In fact, the MASTER program offers you a choice of three different game versions, and you can modify the program to play even more games.

Operating Instructions for MASTER

1. LOAD the program 'MASTER' and RUN it.

2. In response to '1 OR 2 PLAYERS?' press '1'.

3. Select EASY game by pressing the '1' key again.

4. Read the rules of the game from the screen. You are trying to discover a secret pattern of four letters. In this EASY version of the game, each of the letters A, B, C, and D will appear exactly once in the pattern.

5. Type in a four-letter pattern using the letters A, B, C, and D. Your guess may use repeated letters, but the secret pattern in this version of the game does not.

6. Until you type the fourth letter, you may start your guess over again by pressing the 'DELETE' key.

7. After you press the fourth letter, the computer will respond with two numbers. The first number, in the 'ON' column, is the number of letters you guessed in the correct position. The second number, in the 'OFF' column, is the number of other letters you guessed correctly, but in the wrong position.

8. Use this information to determine the secret pattern. Repeat the steps from number 5 above until you have correctly guessed the secret pattern.

9. When you have guessed the pattern correctly, the Atari will congratulate you and tell you how many guesses you made.

Running the Program

Load the program 'MASTER' and RUN it. The screen will clear and the message '1 OR 2 PLAYERS?' will be displayed at the top. For now, select '1'. (The two-player game is described later.) Next you are offered a menu of game difficulty levels. Press '1', '2', or '3' to select a game. (If you want, you can change your choice for the next game.) The rules appropriate to the game you have selected are then displayed. The rules are printed here for reference.

> In the EASY game, only A, B, C, and D are allowed and no letter may be repeated in the secret pattern. Your guesses may include repeated letters, though.
>
> In the MID game, only A, B, C, and D are allowed, but these letters may be repeated in the pattern.
>
> In the HARD game, A, B, C, D, E, and F are allowed, and letters may be repeated.

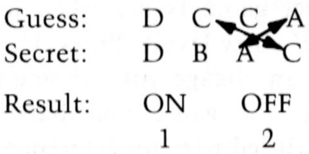
Press any key (except BREAK or RESET) to continue. The computer now generates, at random, a secret pattern. The screen will clear and appear as below:

```
SELECT LETTER ON OFF

   > ■
```

Only '?', 'DELETE', and the letters allowed in the game will be accepted from the keyboard. (The BREAK key does work, though!) Acceptable characters will be printed on the screen; unacceptable characters will have no effect. As soon as you enter the fourth letter in your guess pattern, the program will process it. Until you enter that fourth character, though, you may change your mind. Press 'DELETE' to restart your guess. If at any time you want to give up, the '?' key will print the secret pattern and let you start over with a new game and pattern.

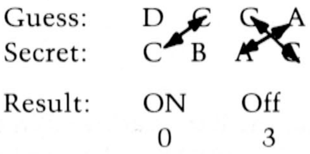
When you enter the fourth item in your guess pattern, the computer matches it against the secret pattern. In the 'ON' column is the number of letters guessed correctly and in the right position; in the 'OFF' column is the number guessed right but in the wrong position. Understanding the matching process will help you learn to play the game better. For instance, if you guessed 'D C C A' and the secret pattern is 'D B A C', then the computer will return a '1' in the 'ON' column and a '2' in the 'OFF' column.

Figure 1



The 'D' is in the correct position (indicated by the shading), but the 'C' and the 'A' (matches indicated with arrows), while they do exist in the secret pattern, were guessed in the wrong position. Only one of the C's in the guess is counted since there is only one C in the secret pattern. If the secret pattern were 'C B A C' instead, then the program would return '0' in the 'ON' column and '3' in the 'OFF' column. Both C's in the guess are now counted.

Figure 2



When you have guessed the secret pattern correctly, you will be congratulated and told the number of guesses you took. Then the program is restarted with selection of the game level.

As you play more and more games, you will begin to develop systems to help you guess the pattern as quickly as possible. One technique that is sometimes useful is substituting one character at a time.

Two – person Game

The two-person option allows a second player to input a secret pattern instead of having the computer provide one. The player who will be guessing should look away from the screen while the other player inputs a pattern. The program tests for the letters allowed, but it does not check for repetitions. Be sure to follow the repetition rule in effect. To go back to the one-person version press BREAK, type 'RUN', and press RETURN. This time answer '1' for the number of players.

Programming Techniques in MASTER

Random Numbers

In the one-player version of MASTER, the program presents a different secret pattern of letters each time the game is played. How is this done? The secret is random numbers. BASIC is able to generate random numbers using the RND function.

A random number is one that is obtained without any predictability or repeatability. Rolling a die, flipping a coin, and spinning a roulette wheel are all means of obtaining random numbers.

Many programming applications require a source of random numbers. For statistics programs they can provide sample data to test a model, and in physics they can be used for applications such as demonstrating the behavior of gas particles.

Many game programs require random numbers. These may be used in the form of playing cards, dice, or locations of hidden treasures. In the one-player version of MASTER, random numbers are used to generate the secret pattern of letters.

The BASIC function RND generates pseudo-random numbers in the range between 0 and 1. Pseudo-random means each succeeding number depends to some extent on the previous one. As a result, after many thousands of numbers, the sequence will start over. This makes statistics involving very large samples sometimes difficult, but it usually causes no problem in games, which use considerably fewer numbers.

Once we have a sequence of random numbers, how do we turn this sequence into the letter patterns for MASTER? Line 1020 does it all in one BASIC expression: RN=INT(RND(0)*N+1), where N is the number of letters allowed in the game. See figure 3 for a graphic illustration of how four random numbers are converted into the four letters of a secret MASTER pattern. RND(0) produces numbers in the range of 0 to 1, but this does not include 1 itself. First we multiply the number by the number of letters allowed in the game. If we allow four letters (N=4), then we multiply the random numbers by 4 to get numbers in the range 0 to 3.999.... Next we add 1 to make it 1 to 4.999.... Then we use the BASIC INT function to remove whatever is to the right of the decimal point, leaving us with 1, 2, 3, or 4. These numbers are never actually converted to letters. Instead, the letters the player types for a guess are converted to numbers.

Figure 3

	Element #			
BASIC Function	1	2	3	4
RND(1)	.555877482	.689094948	.828839479	.0619696133
*4	2.22350993	2.75637979	3.31535792	.247878453
1	3.22350993	3.75637979	4.31535792	1.24787845
INT( )	3	3	4	1
LETTER	C	C	D	A

The same technique can be used to get random numbers over any range. For dice, multiply by 6, add 1, and take the integer. For playing cards, multiply by 52, take the integer, and add 1. (Converting 1 to 52 into suits and ranks is another problem!)

You don't have to restrict yourself to a range of numbers or letters. Try the following short program. (You'll have to type NEW and press RETURN before you type it in. To use MASTER again, reLOAD it.)

```
10 DIM A$(24)
20 A$="APPLE PEAR  BANANAGRAPE "
30 X=INT(4*RND(0))*6+1
40 PRINT A$(X,X+5)
50 GOTO 30
```

A random sequence of fruits! Perhaps this is the beginning of a slot-machine program? The string defined in line 20 could be combinations of graphic characters and cursor controls that actually draw pictures (such as dice or playing cards) on the screen. See the "Atari Clock" chapter to learn about programming cursor controls into strings and positioning images on the screen.

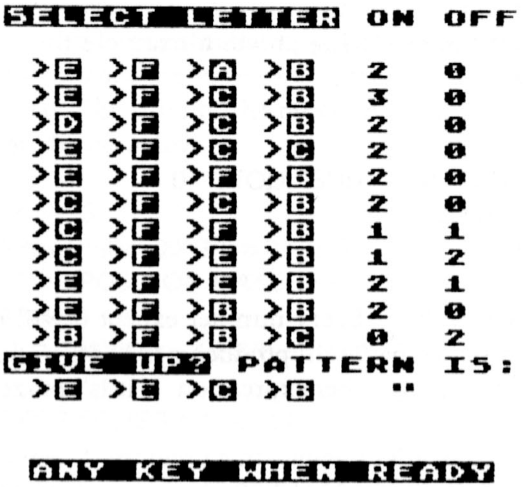
Flags and Logic

One of the most powerful features of a computer is its ability to make decisions. MASTER uses the Atari's decision-making ability throughout its program.

Every decision boils down to deciding whether an expression is true or false. The BASIC IF...THEN construction makes this decision. If the expression after the IF is true, then whatever appears on the line after the THEN is executed. If the expression is false, then the rest of the line is skipped and execution continues with the next line.

Figure 4

	Secret Pattern					Guess Pattern				PM	OM
I					J					0	0
1	D	B	A	C	1	D	C	C	A	1	0
2	~~D~~	$\boxed{B}$	A	C	2	~~D~~	$\boxed{C}$	C	A	1	0
3	~~D~~	B	$\boxed{A}$	C	3	~~D~~	C	$\boxed{C}$	A	1	0
4	~~D~~	B	A	$\boxed{C}$	4	~~D~~	C	C	A	1	0
2	~~D~~	$\boxed{B}$	A	C	2	~~D~~	$\boxed{C}$	C	A	1	0
2	~~D~~	$\boxed{B}$	A	C	3	~~D~~	C	$\boxed{C}$	A	1	0
2	~~D~~	$\boxed{B}$	A	C	4	~~D~~	C	C	$\boxed{A}$	1	0
3	~~D~~	B	$\boxed{A}$	C	2	~~D~~	$\boxed{C}$	C	A	1	0
3	~~D~~	B	$\boxed{A}$	C	3	~~D~~	C	$\boxed{C}$	A	1	0
3	~~D~~	B	A	C	4	~~D~~	C	C	A	1	1
4	~~D~~	B	~~A~~	C	2	~~D~~	C	C	~~A~~	1	2



```
SELECT LETTER  ON  OFF

 >E  >F  >A  >B   2   0
 >E  >F  >C  >B   3   0
 >D  >F  >C  >B   2   0
 >E  >F  >C  >C   2   0
 >E  >F  >F  >B   2   0
 >C  >F  >C  >B   2   0
 >C  >F  >F  >B   1   1
 >C  >F  >E  >B   1   2
 >E  >F  >E  >B   2   1
 >E  >F  >B  >B   2   0
 >B  >F  >B  >C   0   2
GIVE UP? PATTERN  IS:
 >E  >E  >C  >B    ..


  ANY KEY WHEN READY
```

Atari BASIC doesn't actually handle the words true and false. Instead, it assigns 1 to represent true and 0 to represent false. When evaluating expressions, any non-zero result is considered true. To see this in action try the following example:

```
10 PRINT "A=";:INPUT A
20 PRINT "B=";:INPUT B
30 IF A=B THEN PRINT "TRUE":GOTO 50
40 PRINT "FALSE"
50 PRINT A=B
60 GOTO 10
```

Run this program. Type in a value for A and press RETURN, type in a value for B and press RETURN. If the number you entered for A equals the number you typed for B, then 'TRUE' will be printed followed by a 1; Otherwise 'FALSE' is printed followed by a 0. The number 1 or 0 is the value BASIC assigned to the expression 'A=B'. Line 170 in MASTER checks to see whether or not the number of correct position matches (PM) is equal to the number of letters (NN) in the pattern. If so, the player has correctly guessed the pattern and the congratulation routine (6000) is executed by returning to line 100 before starting a new game.

Now enter the following program example that demonstrates the use of a flag.

```
10 INPUT A
20 IF A THEN PRINT "TRUE":GOTO 40
30 PRINT "FALSE"
40 GOTO 10
```

Try a few numbers. Every number except 0 will result in 'TRUE' being printed. Entering 0 will produce a 'FALSE'. The 'A' in line 20 is evaluated just like any other expression. If it is non-zero then it is considered true.

A flag is a convenient device in a program. It can be either set (true or 1) or clear (false or 0). MASTER uses several variables as flags: RP, RQ, and the arrays PF( ) and PG( ). RP is set or cleared in the game-selection routine (in line 7100, 7200, or 7300), depending on the game chosen. In line 1030, if RP is set (1) then lines 1040-1080, which prevent duplicate letters in the pattern, are skipped. RQ stays cleared unless a duplicate letter is found. If the flag is set, then the program returns to line 1020 to determine a new number. Each element of the secret pattern has an element in the flag array PF( ), and each element in the guessed pattern has an element in the flag array PG( ). See the discussion under "Processing a Guess" for details of how these flags are used.

Another interesting use of a flag is in the display of the congratulation message (6050-6140). A FOR...NEXT loop is used to alternate the variable I between 0 and 1. The flag I is tested in lines 6060 and 6110. If the flag is set, the character color is changed to red and then back to blue. If the flag is clear, no color changes are made. This produces the alternating color effect.

The program has to make decisions in a number of other places, evaluating an expression to determine what to do next. The IF...THEN statement is used most commonly for decision making, but ON...GOSUB and ON...GOTO are also used. ON...GOSUB is used in line 110 to decide whether to generate a random pattern in a 1-player game or to let a player input a pattern.

Processing a Guess

As explained earlier, the match count is determined by first checking for exact position matches and then going through to check for out-of-position matches. No element in either the secret or guess pattern may be used more than once in a match.

To avoid reusing pattern elements in matches, we need to program a way to cross off pattern elements that have been used in a match. In addition to the two arrays of the elements themselves, two corresponding flag arrays are used. The lines of the program involved in checking for matches are reproduced below.

```
3000 REM CLEAR FLAGS
3010 FOR I = 1 TO NN:PF(I) = 0:PG(I) = 0:NEXT I
3020 REM CHECK FOR POSITION MATCHES
3030 FOR I = 1 TO NN
3040 IF R(I) = GU(I) THEN PF(I) = 1:PG(I) = 1:PM = PM1
3050 NEXT I
3100 REM CHECK FOR OTHER MATCHES
3110 FOR I = 1 TO NN
3120 IF PF(I) THEN 3170
3130 FOR J = 1 TO NN
3140 IF PG(J) THEN 3160
3150 IF R(I) = GU(J) THEN OM = OM1:PF(I) = 1:PG(J) = 1:J = NN
3160 NEXT J
3170 NEXT I:RETURN
```

At the beginning of the matching process, all the flags are cleared or set to 0 (line 3010). As each match is detected, the flags corresponding to the matched elements are set (in lines 3040 and 3150). The flags are checked in lines 3120 and 3140. If the flag is set, then the matching process is skipped and the next element is checked. In addition, when a match is found in line 3150, the higher numbered elements in the guess pattern are skipped by setting the loop index J to its maximum value, NN. The NEXT J statement in line 3160 sees J equal to its maximum value and is fooled into thinking it's through with the specified repetitions. Control passes to the NEXT I statement in line 3170.

This process is graphically demonstrated in figure 4. I is the index into the secret pattern, while J is the index into the guess pattern. The boxes indicate the two elements currently being compared, PM is the number of position matches, and OM is the number of out-of-position matches. A shaded box indicates a match, and a diagonal line through an element indicates that it has been used in a match already. First, the position matches are checked. The result is 1, with the D's in the first position crossed off. In the program, the flags PF(1) and PG(1) are set to 1.

Next, the out-of-position matches are checked. Since the first elements in each pattern have already been used, the comparison begins with the second elements. No match is found for the B, so the search continues with the third element of the secret pattern and the second element of the guess pattern. When the match is found with the fourth guess element, these two are crossed off and the out-of-position match counter OM is incremented. A match is found immediately for the fourth secret pattern element, so the remaining two elements are skipped and the counter incremented again. *One* position match and *two* out-of-position matches are reported to the player.

If you are still confused about how this works, try a different pattern and construct a table similar to figure 4. You might also try running through the program lines with an example.

Scrolling the Graphics Screen

Atari BASIC handles nine different graphics modes, three of which are text modes. Mode 0 is the standard mode that you see when you first turn on the computer. Modes 1 and 2 are expanded text modes; the characters are larger and there are more colors available. MASTER uses graphics mode 1 to make use of the extra colors. Normally, modes 1-9 leave a four-line text window at the bottom of the screen. A full screen can be obtained by adding 16 to the mode number. Therefore, the statement GRAPHICS 17 is used in MASTER to obtain a full-screen mode 1.

Displaying text in graphics mode 1 has one drawback — scrolling is not handled automatically if a PRINT statement is accidentally made beyond the screen limits. Instead, the program will halt with a 'Cursor Out of Range' error. One way to get around this problem is to clear the screen when the cursor reaches the bottom of the screen and to start over at the top. But in MASTER we want to keep the previous guesses on the screen because they are important to the discovery of the secret pattern. Therefore, a machine-language routine was added to scroll all of the screen but the heading 'SELECT LETTER ON OFF'.

```
DEC                 New Monics         Comments
104                 PLA                ;# of arguments
104                 PLA                ;source
133 204             STA     204
104                 PLA
133 203             STA     203
104                 PLA                ;destination
133 206             STA     206
104                 PLA
133 205             STA     205
104                 PLA                ;byte count
133 208             STA     208
104                 PLA
133 207             STA     207
160 0               LDY     #0         ;zero index
177 203   loop      LDA     (203),Y
145 205             STA     (205),Y
230 203             INC     203        ;increment source
208 2               BNE     to
230 204             INC     204
230 205   to        INC     205        ;increment destination
208 2               BNE     count
230 206             INC     206
198 207   count     DEC     207
165 207             LDA     207
201 255             CMP     255
208 2               BNE     check
198 208             DEC     208        ;decrement byte count
165 207   check     LDA     207
  5 208             ORA     208
208 224             BNE     loop
96                  RTS
```

The simple block-move routine above is called from within the program by a USR function. The format for this function is X = USR (routine address, argument list). X is a dummy variable in our case; its value is never important, but it must be included in the function call all the same. Normally, a value will be calculated within the USR call, and it is returned in X. The USR function requires the starting address of the routine. This is supplied in line 510 by the ADR function, which returns the starting address of SC$. This is a string composed of characters whose numeric codes correspond to the machine code of the routine. Following the starting address of the routine is a list of three arguments that are passed to the routine. These specify the starting address of the block of memory to move, the address to which it is to be moved, and the number of bytes to move. In MASTER, we use SCR + 60 as the first argument. SCR is the beginning of the screen memory; adding 60 starts the scroll at the third line of the screen. SCR + 40 designates the address to move to or the second line of the screen. Finally, we want to move 440 bytes of screen memory, which corresponds to 22 screen lines.

The scroll routine is performed as needed by using a TRAP statement. All errors that occur within a program are reported by the system unless a TRAP has been set up. The line number after this statement indicates to the system where to go when an error is encountered. The first line of the initialization routine (line 8010) points a TRAP at line 500, designating it as an error-handling routine. Line 500 checks for errors other than 'Cursor Out of Range' errors; these are taken care of by lines 540-550, which print a simple error message and end the program. Line 510 performs a one-line scroll, and line 520 resets the TRAP to line 500 so that future cursor errors will also lead to a scroll. Then a return is made to the error-causing line, where the PRINT statement is this time successfully performed. Notice that in the areas of the program where scrolling might be needed (as in lines 9000-9100), PRINT statements are limited to one per line. If there are two or more per line, an infinite loop is created by the TRAP and the GOTO at the end of the TRAP routine, performing the first of the multiple PRINTs repeatedly.

Customizing Your MASTER Game

Adding an EXPERT Level

Because of the way MASTER is written it is easy to add your own version to the game. As an example of how to do this, let's add an EXPERT game to the three choices we have already. Add or substitute the following lines to the program supplied.

```
7045 PRINT #6:PRINT #6;"   4 EXPERT"
7060 T=T-48:IF T <1 OR T >4 THEN 7050
7070 ON T GOTO 7100,7200,7300,7600
7600 N=8:RP=1:G(1)=8:G(2)=12:G(3)=16:G(4)=20:
G(5)=25
7610 PRINT #6;"   EXPERT GAME:"
7620 GOSUB 7400
7630 PRINT #6;"   MORE THAN ONCE"
7640 RETURN
```

This version of the game allows the first eight letters of the alphabet. The operation of the game itself is controlled by the values of N and RP in line 7600. The rest of the program changes involve adding the game to the menu and displaying the rules. The value of N determines the number of letters allowed in the game. RP is a flag, which, if set (1), allows repeats of letters in the pattern (see the "Flags" section above). The array G( ) holds the cut-off numbers of guesses for each congratulation message. Adjust these values and program the appropriate messages as in the example above and you will be able to add your own game version.

Number of Elements in Pattern

The number of elements is four for all versions of the game described so far. This number can be changed to practically any number, the only limitations being the width of the display and the amount of memory in your Atari. The number of elements in the pattern is determined by the value of NN in line 8140 of the initialization routine. Change line 8140 to read:

```
8140 K=764:NN=3:G(0)=1
```

Now run the program. Notice that everything works as before, except only three letters are generated in the secret pattern and only three are expected in each guess.

Up to six elements can be accommodated without disturbing the rest of the display. However, with more than four elements allowed per guess, only the MID and the HARD games may be chosen because they contain more than four elements in their secret patterns. One solution for longer patterns is to print the clues on the next line:

```
180 POSITION 14,PEEK(CR+1):PRINT #6;PM;"  ";OM
```

MASTER Variable Usage

Constants

CR	Address of current row position
K	Address containing code of last key pressed
NN	Number of elements in pattern
SCR	Address of beginning of screen memory
CN$	'ANY KEY WHEN READY' message at bottom of screen
FD$	' ' pointer to GET next guess element
MS$	Congratulation messages
OB$	Colored blocks for first six letters
SC$	Machine-language scroll routine

Variables

GN	Guess number
I	Loop variable
J	Loop variable
MS	Index into congratulation-message string
N	Number of letters allowed
NP	Number of players
OM	Out-of-position match counter
PM	Position match counter
RN	Random number, temporary
RP	Flag (=1: repeats allowed)
RQ	Flag (=1: indicates number already used in pattern)
T	Keyboard character code
X	Dummy variable for USR call

Arrays

G( )	Cut-off numbers of guesses for congratulation messages
GU( )	Guess pattern
PF( )	Flags for crossing off secret elements
PG( )	Flags for crossing off guess elements
R( )	Secret pattern

Program Description

Initialization (20): Subroutine 8000 sets up a number of constants, and subroutine 7500 gets the number of players.

Program mainline (100-190): Subroutine 7000 gets the skill level for the game and displays the instructions for the game. Subroutine 5000 waits for a key to be pressed before continuing with the main program.

Line 110 uses the ON...GOSUB structure to determine whether to call subroutine 1000, which generates a random pattern, or subroutine 4000, which allows one player to input a pattern. NP can have only two values, 1 or 2. On 1, subroutine 1000 is called; on 2, subroutine 4000 is called.

GN is used to count the number of guesses. Line 120 calls subroutine 2000, which prints the header on the screen and receives the first guess. The second and subsequent guesses return to line 130, where the same subroutine is called at 2020 to avoid having the header reprinted for each guess.

In line 140, T contains the ASCII value of the last key the player pressed. A value of 63 corresponds to a '?', which indicates that the player has given up. A call is made to subroutine 9000, which prints out the secret pattern. GOTO 100 starts a new game.

Next the guess must be processed. Before each call to the processing routines, the match counters PM and OM are zeroed. Subroutine 3000 processes the guess, first checking for position matches and then for out-of-position matches. If PM (the number of position matches) equals NN (the number of elements in the pattern), then the player has guessed the pattern. Subroutine 6000 is the congratulations routine.

Line 180 prints out the results of the matching, with the position matches under the heading 'ON' and the out-of-position matches under the heading 'OFF'. The POSITION statement specifies the column and row at which to begin PRINTing; in this case the column is the 14th and the row is the current row, whatever that may be. CR is the memory location containing the current row position.

The guess counter GN is incremented and the program loops back to 130 for another guess.

Scroll Screen (500-510): This routine is called through an error TRAP. Line 500 checks ERR, the error-number location, to see whether or not the error that occurred was a 'Cursor Out of Range' error, number 141. If not, lines 540-550 print a simple error message and STOP the program. A call is made in line 510 to a machine-language USR subroutine that performs the scroll. SC$ contains the code for the routine, and ADR(SC$) supplies the USR function with its starting address. SCR is the beginning of screen memory; a block of this is moved to produce the scroll. SCR + 60 designates the third screen line, which is where the scroll will begin. SCR + 40 is the second line, to where the scroll is moving. Line 520 sets the cursor at the bottom row of the screen by POKEing 23 into CR, the current row. Then the TRAP is restored to line 500 so that the scroll routine is enabled again. Finally, a return is made to the program line that caused the screen error, and the PRINT statement that failed the first time is performed successfully. Memory locations 186 and 187 hold the number of the line in which the last error occurred; they are used in line 530 to return to the appropriate line in the program.

Generate Random Numbers (1000-1090): This routine is called at the beginning of each one-player game to generate the secret pattern. In the supplied version of the game NN is always 4, so four numbers are generated. Line 1020 returns in RN an integer between 1 and the number of letters allowed in the game (N). If RP is non-zero, then repeats are allowed in the pattern. Lines 1040-1080 are skipped and RN is copied into R(I), the current element of the pattern. If repeats are not allowed (RP = 0), then each RN must be checked against the previous elements in the pattern R( ). In line 1040, RQ is set to 0 to indicate that no element has been found so far to match RN. If I = 1, then there aren't any numbers in the pattern and we can skip to 1090 and accept this RN. The FOR...NEXT loop on J (lines 1050-1070) goes from 1 to the previous element (I – 1). If RN is found to match an existing element (RN = R(J)), then RQ is set to 1 to indicate a match has been found and J is set to I – 1 to terminate the FOR...NEXT loop. If no match is found, then the loop continues through all the previously assigned elements. RQ is tested in line 1080: if it is non-zero, then another RN must be calculated (return to 1020); if it is still zero, then we can accept the RN and install it in the current element R(I) of the pattern. The outside FOR...NEXT loop (1010 to 1090) continues until all of the elements required in the pattern have been calculated.

Process Guess (2000-2110): As discussed above under the program mainline, this routine is usually called at 2020, but the first time the call is made to 2000 to print the heading 'SELECT LETTER ON OFF'.

The routine consists of a big FOR...NEXT loop, in which I starts with a value of 1 and ends with the value NN, the number of elements in the pattern. Within this loop, characters from the keyboard are accepted or rejected. The GET function waits for a single character to be input from the keyboard. As soon as a key is pressed, the program continues at line 2060.

Now we test for two special characters, 'DELETE' and '?'. If the 'DELETE' key was pressed, T will have the value 126 (the numeric code for that key) and control will pass to line 2100 to erase the current guess. First, the cursor is POSITIONed at the beginning of the current row. Then a string of 19 spaces is printed to erase any characters in that row. The combination of printing only 19 spaces (rather than the possible 20 in the row) and including a semicolon at the end of the PRINT statement prevents the cursor from advancing a row. Thus, in line

2110, the value of CR will remain unchanged and the cursor will be repositioned at the beginning of the same row, ready for another guess. Then the loop is terminated by setting I to NN and executing a NEXT statement. The GOTO 2030 starts the loop over again. If we had failed to terminate the loop (by omitting the I = NN and NEXT statements) the user would be able to crash the program by repeatedly hitting the 'DELETE' key. BASIC keeps track of each FOR...NEXT loop in an area of memory called the stack. If we don't terminate a loop, that information continues to occupy space on the stack. Repeated calls to 2030 with the 'DELETE' key will continue to build up new FOR...NEXT information on the stack until there is no room left. At this point the program crashes with a 'Memory Insufficient' error. The '?' character (code 63) is similar. The FOR...NEXT loop is terminated and a RETURN is made to the program mainline.

Other characters are converted in line 2080 from their numeric codes into numbers beginning with 1. The code for the letter A is 65, so subtracting 64 makes the conversion. If T is less than 1 or greater than the number of letters allowed in the pattern, then the program branches to 2050 to GET another character. If the character is accepted, then the appropriate colored letter OB$(T,T) is printed and the number T is stored in the current element of the guess pattern GU(I). RETURN takes the flow back to the mainline.

Matching Routines (3000-3170): These routines are described in more detail in the main text under the section "Processing a Guess."

Line 3010 clears the flag arrays PG( ) and PF( ) by setting them to 0. Lines 3030-3050 advance, position by position, through the secret pattern R( ) and guess pattern GU( ) arrays checking for matches. If a match is found, the position match counter PM is incremented and the corresponding flags are set to 1.

Lines 3110-3170 check all the other possibilities for matches. The flags are used to cross off elements as they are matched. Some economy is achieved by skipping over crossed-off elements (lines 3120 and 3140) and by terminating the inside loop as soon as a match is found (J = NN at the end of line 3150).

Input Pattern with Two Players (4000-4130): After the instructions are displayed, this routine accepts letters one-by-one until the pattern is filled. It is similar to the guess-processing routine (2000-2110). Instead of filling the guess array, the secret pattern array R( ) is filled. See the description above for details.

ANY KEY WHEN READY (5000-5010): The string CN$ is a constant defined in the initialization routine. The result is to print the message 'ANY KEY WHEN READY' at the bottom of the screen. The GET statement waits for a key before the RETURN is made.

Congratulation Routine (6000-6140): Lines 6010-6030 use the number of guesses GN to determine the congratulation message. The array G( ) is set up for each version of the game in line 7100, 7200, or 7300. The messages in MS$ are set up during initialization. First, MS is cleared before any value is found for it. By comparing GN to each cutoff value G( ) with the < = (less than or equal to) operation, the offset MS for the correct substring of MS$ is determined. If MS is still 0 when the loop is finished, then GN was too big for even the last cutoff value; subsequently MS is set to 91, the offset for the 'TRY,TRY,TRY AGAIN!' message.

Line 6040 clears the graphics screen and clears the keyboard status location K by POKEing 255 into it. Lines 6050-6140 display the congratulation screen, alternating the character color of the message and the number of guesses between red and blue. The use of I as a flag is discussed above under "Flags." If I=1, then the character color is changed. I's value alternates between 0 and 1.

Line 6100 is a FOR...NEXT loop that does nothing between the FOR and the NEXT! By adjusting the number after the TO, you can achieve a delay in the program of nearly any time. Here it controls the rate of the flashing.

Lines 6120-6130 check to see if a key has been pressed on the keyboard. If the PEEK of location K equals 255, then no key was pressed and the congratulations display loop continues. Any number other than 255 in K means that a key has been pressed; line 6130 takes care of that case. First, location K is reset by POKEing a 255 into it, then the character color is returned to blue as it was originally. The FOR...NEXT loop is terminated by setting I=1 and performing a NEXT. Finally, a RETURN is made to the program mainline.

Select Game and Display Instructions (7000-7450): Lines 7010-7040 display a menu listing the different games available. Line 7050 awaits a key, which is converted to a number and tested against the range of the menu in line 7060. If the key is out of range, then the program branches back to 7050 for another key. The ON...GOTO instruction in line 7070 calls 7100 if T is 1, 7200 if T is 2, or 7300 if T is 3.

Each of these set-up-and-display routines establishes N (the number of letters allowed in the game), RP (the flag determining whether or not repeats are allowed), and G( ) (the array of guess number cutoff values for the congratulation messages). Then the name of the game is displayed. Next subroutine 7400, which displays parts of the instructions common to all games, is called. Finally, the rule regarding repeats is printed in the proper place.

Subroutine 7400 first prints the colored letters corresponding to the number of letters allowed (N). If the number allowed is four, the first four letters are printed. Then the portion of the directions common to all versions of the game is printed.

Get Number of Players (7500-7520): This subroutine is called once when the program is first run. After asking for the number of players, it GETs a key from the keyboard. Only 1 or 2 is accepted and the value is returned in NP.

Initialization (8000-8220): A TRAP is set up in line 8010 to send control to the scroll routine when a screen error occurs. Lines 8020-8030 dimension the strings and arrays that will be used in the program. Lines 8040-8060 set up graphics mode 1 without a text window (mode 17) and define the various colors to be used. Colors 0 through 4 are defined as blue, black, green, red, and white, respectively. Line 8070 opens the keyboard (specified by ''K'') as input device #1; the GET #1 statements within the program receive single keystrokes as input through device #1. Lines 8110-8120 set up the strings used in the program. The colored letters that are displayed for the guesses are set up in OB$. Lines 8130-8140 set up other constants and lines 8150-8200 define the congratulation messages. Line 8210 defines SC$, which contains the code for the machine-language scroll routine.

Print Pattern on Give-up (9000-9050): The secret pattern is printed out in the appropriate colored letters using the secret pattern array R( ).

***Listing 1:* Master**

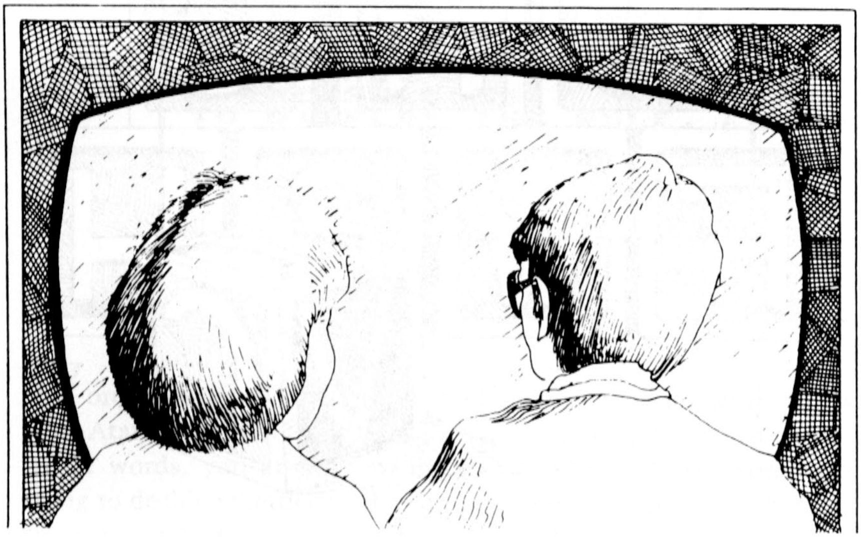
```
10 REM MASTER FOR THE ATARI
20 GOSUB 8000:GOSUB 7500
100 GOSUB 7000:GOSUB 5000
110 ON NP GOSUB 1000,4000
120 GN=1:GOSUB 2000:GOTO 140
130 GOSUB 2020
140 IF T=63 THEN GOSUB 9000:GOTO 1
    00
150 PM=0:OM=0
160 GOSUB 3000
170 IF PM=NN THEN GOSUB 6000:GOTO
    100
180 POSITION 14,PEEK(CR):PRINT #6;
    PM;"   ";OM
190 GN=GN+1:GOTO 130
500 IF PEEK(ERR)<>141 THEN 540
510 X=USR(ADR(SC$),SCR+60,SCR+40,4
    40)
520 POKE CR,23:TRAP 500
530 GOTO PEEK(ERL)+PEEK(ERL+1)*256
```

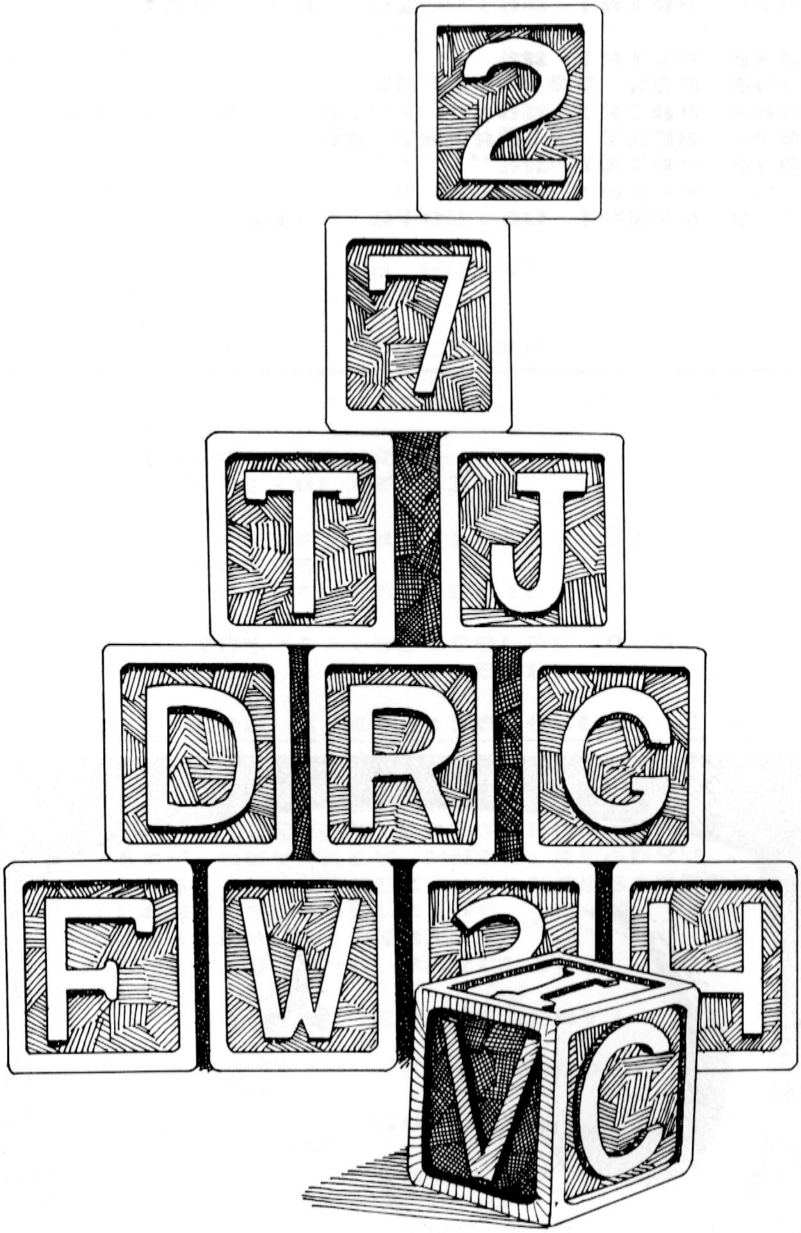
```
540 PRINT "ERROR ";PEEK(ERR);" AT
    LINE ";PEEK(ERL)+PEEK(ERL+1)*2
    56
550 CLOSE #1:STOP
1000 REM GENERATE RANDOM NUMBERS
1010 FOR I=1 TO NN
1020 RN=INT(RND(0)*N+1)
1030 IF RP THEN 1090
1040 RQ=0:IF I=1 THEN 1090
1050 FOR J=1 TO I-1
1060 IF RN=R(J) THEN RQ=1:J=I-1
1070 NEXT J
1080 IF RQ THEN 1020
1090 R(I)=RN:NEXT I:RETURN
2000 REM PRINT HEADER
2010 PRINT #6;"KSELECT LETTER ON O
    FF"
2020 REM PROCESS GUESS
2030 FOR I=1 TO NN
2040 PRINT #6;FD$;
2050 GET #1,T
2060 IF T=126 THEN 2100
2070 IF T=63 THEN I=NN:NEXT I:RETU
    RN
2080 T=T-64:IF T<1 OR T>N THEN 205
    0
2090 PRINTINT #6;OB$(T,T);:GU(I)=T:
    NEXT I:RETURN
2100 POSITION 0,PEEK(CR):PRINT #6;
    "                ";
2110 POSITION 0,PEEK(CR):I=NN:NEXT
    I:GOTO 2030
3000 REM CLEAR FLAGS
3010 FOR I=1 TO NN:PF(I)=0:PG(I)=0
    :NEXT I
3020 REM CHECK FOR POSITION MATCHE
    S
3030 FOR I=1 TO NN
3040 IF R(I)=GU(I) THEN PF(I)=1:PG
    (I)=1:PM=PM+1
3050 NEXT I
3100 REM CHECK FOR OTHER MATCHES
3110 FOR I=1 TO NN
3120 IF PF(I) THEN 3170
3130 FOR J=1 TO NN
3140 IF PG(J) THEN 3160
3150 IF R(I)=GU(J) THEN OM=OM+1:PF
    (I)=1:PG(J)=1:J=NN
3160 NEXT J
3170 NEXT I:RETURN
4000 REM INPUT PATTERN
4010 PRINT #6;"KONE PLAYER ENTERS"
4020 PRINT #6;" PATTERN WHILE"
4030 PRINT #6;"other player"
4040 PRINT #6;" LOOKS AWAY"
```

```
4050 PRINT #6:PRINT #6;"ENTER PATT
    ERN:      "
4060 FOR I=1 TO NN
4070 PRINT #6;FD$;
4080 GET #1,T
4090 IF T=126 THEN 4120
4100 T=T-64:IF T<1 OR T>N THEN 408
    0
4110 PRINT #6;OB$(T,T);:R(I)=T:NEX
    T I:RETURN
4120 POSITION 0,7:PRINT #6;"
                 "
4130 POSITION 0,7:I=NN:NEXT I:GOTO
     4060
5000 POSITION 0,22:PRINT #6;CN$
5010 GET #1,T:RETURN
6000 REM CONGRATULATIONS
6010 MS=0:FOR I=0 TO 5
6020 IF GN<=G(I) THEN MS=I*18+1:I=
    5
6030 NEXT I:IF MS=0 THEN MS=91
6040 PRINT #6;"↰";:POKE K,255
6050 FOR I=0 TO 1
6060 IF I THEN SETCOLOR 0,4,8
6070 POSITION 0,0:PRINT #6;MS$(MS,
    MS+17)
6080 PRINT #6:PRINT #6;"YOU TOOK "
    ;GN;" TRIES!"
6090 POSITION 0,22:PRINT #6;CN$
6100 FOR J=1 TO 50:NEXT J
6110 IF I THEN SETCOLOR 0,9,8
6120 IF PEEK(K)=255 THEN 6140
6130 POKE K,255:SETCOLOR 0,9,8:I=1
    :NEXT I:RETURN
6140 NEXT I:GOTO 6050
7000 REM PROCESS INITIAL CONDITION
    S
7010 PRINT #6;"↰ SELECT GAME:"
7020 PRINT #6:PRINT #6;"  1 EASY"
7030 PRINT #6:PRINT #6;"  2 MID"
7040 PRINT #6:PRINT #6;"  3 HARD"
7050 GET #1,T
7060 T=T-48:IF T<1 OR T>3 THEN 705
    0
7070 ON T GOTO 7100,7200,7300
7080 ON T GOTO 7100,7200,7300
7100 N=4:RP=0:G(1)=3:G(2)=5:G(3)=7
    :G(4)=10:G(5)=15
7110 PRINT #6;"↰EASY GAME:"
7120 GOSUB 7400
7130 PRINT #6;"  ONLY ONCE"
7140 RETURN
7200 N=4:RP=1:G(1)=4:G(2)=6:G(3)=8
    :G(4)=12:G(5)=18
7210 PRINT #6;"↰MID GAME:"
```

```
7220 GOSUB 7400
7230 PRINT #6;"  MORE THAN ONCE"
7240 RETURN
7300 N=6:RP=1:G(1)=5:G(2)=7:G(3)=1
    0:G(4)=15:G(5)=20
7310 PRINT #6;"↖HARD GAME:"
7320 GOSUB 7400
7330 PRINT #6;"  MORE THAN ONCE"
7340 RETURN
7400 POSITION 0,3:FOR I=1 TO N:PRI
    NT #6;OB$(I,I);" ";:NEXT I
7410 PRINT #6;'5;"ALLOWED":PRINT #6
7420 PRINT #6:PRINT #6;"  EACH MAY
     BE USED"
7430 POSITION 0,10:PRINT #6;"'DEL'
     TO CLEAR GUESS"
7440 PRINT #6;"  ?  TO GIVE UP"
7450 POSITION 0,7:RETURN
7500 PRINT #6;"↖ 1 OR 2 PLAYERS?"
7510 GET #1,T:IF T<49 OR T>50 THEN
     7510
7520 NP=T-48:RETURN
8000 REM INITIALIZATION
8010 TRAP 500
8020 DIM OB$(6),CN$(19),FD$(1),MS$
    (108),SC$(54)
8030 DIM R(6),GU(6),PF(6),PG(6),G(
    5)
8040 GRAPHICS 17:SETCOLOR 0,9,8
8050 SETCOLOR 1,0,2:SETCOLOR 2,12,
    10
8060 SETCOLOR 3,4,8:SETCOLOR 4,0,1
    4
8070 OPEN #1,4,0,"K:"
8100 REM SET-UP OF CONSTANTS
8110 OB$="abCDEf":FD$=CHR$(30)
8120 CN$=" ANY KEY WHEN READY"
8130 CR=84:SCR=PEEK(89)*256+PEEK(8
    8):ERR=195:ERL=186
8140 K=764:NN=4:G(0)=1
8150 MS$(1,18)="A PSYCHIC!
     "
8160 MS$(19,36)="EXCELLENT!
     "
8170 MS$(37,54)="VERY GOOD!
     "
8180 MS$(55,72)="GOOD
     "
8190 MS$(73,90)="FAIR
     "
8200 MS$(91,108)="TRY,TRY,TRY AGAI
    N!"
8210 SC$="hh▗Lh▗Kh▗Nh▗Mh▗Ph▗O ♥1K▗
    MfKP fL fMP fNF0%0I)P FP%0┐PP◆◆
    "
```

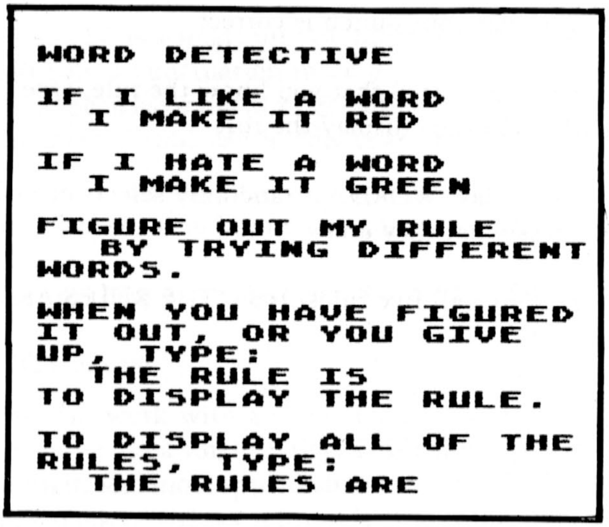
```
8220 RETURN
9000 REM PRINT PATTERN ON GIVE-UP
9010 PRINT #6
9020 PRINT #6
9030 PRINT #6;"GIVE UP? PATTERN IS
   "
9040 PRINT #6
9050 FOR I=1 TO NN
9060 PRINT #6;" >";OB$(R(I),R(I));
9070 NEXT I:PRINT #6
9080 PRINT #6
9090 PRINT #6
9100 PRINT #6:GOTO 5000
```




2
7
T J
D R G
F W 2 H
T
V C

Word Detective

A Word Guessing Game



```
WORD DETECTIVE

IF I LIKE A WORD
  I MAKE IT RED

IF I HATE A WORD
  I MAKE IT GREEN

FIGURE OUT MY RULE
   BY TRYING DIFFERENT
WORDS.

WHEN YOU HAVE FIGURED
IT OUT, OR YOU GIVE
UP, TYPE:
  THE RULE IS
TO DISPLAY THE RULE.

TO DISPLAY ALL OF THE
RULES, TYPE:
  THE RULES ARE
```

Word Detective is a game that challenges you to figure out why your Atari likes some words and hates other words. By typing in different words, you attempt to figure out what rule the computer is using to decide whether it likes or hates a word. The game introduces the concept of the computer handling letters and words, as well as numbers. Detailed listings and instructions show you how to modify the program to use rules that *you* create.

Operating Instructions

1. Load the program 'WORD' from disk.

2. Type 'RUN' and press RETURN.

3. You will be presented with the basic rules for Word Detective. Press any key to continue.

4. Type in a word for Word Detective to analyze.

5. When you press the RETURN at the end of the word, 'WORD' will turn the word GREEN if it hates the word or RED if it loves the word.

6. Continue giving 'WORD' one word at a time and try to guess the rule. When you think you know the rule, try some additional words to make sure that your hunch is correct.

7. When you are convinced that you know the rule, type 'THE RULE IS'. 'WORD' will then display the rule.

8. Each time you play 'WORD,' it randomly selects one of five rules. Can you guess what they are?

9. If you want to see all five rules, type 'THE RULES ARE'.

Programming Concepts

While we normally think of computers as devices for dealing with numbers (referring to them sometimes as 'number crunchers'), they can also deal effectively with alphabetic material. One of the major uses of microcomputers today is as a word processor — a machine that can replace the typewriter and offer many kinds of support to the writer. Word processors can provide editing features such as character insert, character delete, line insert, and line delete; they can move blocks of data, and more. Spelling-checking programs can eliminate most spelling errors. Text can be stored on cassette or disk, sent from one computer to another, and printed out in various formats. BASIC has a number of commands to assist in dealing with non-numeric information; the most important ones are demonstrated in Word Detective.

Processing ALPHABETIC Data

A CHARACTER is a single unit of information such as a letter (A d X), a digit (0 1 2), a punctuation mark (! ? '), a mathematical symbol (= > < +), a special symbol ($ # [@), or various graphic symbols and control characters. BASIC can handle characters in two ways:

1. It can refer to them directly, as in a PRINT string, by enclosing a string of characters in quotation marks: PRINT "THESE ARE CHARACTERS".
2. It can refer to them as a variable: C$ = "LETTER".

The STRING Concept

The Atari can work with a single letter, a word, a sentence, or a large portion of text as a single entity. This collection of letters (which can include digits and most special characters) is called a 'string.' Atari BASIC keeps track of strings separately from other types of information, such as floating-point numbers. It names them with up to 120 characters per name, like the numbers, but adds the dollar sign ($) to show that a string of characters is being referenced. Examples of string names include A$, Q23$, and HELLO$. A string may have *no* characters, in which case it is called a NULL string. A string may have a single character or may be very long, up to 32,767 characters in Atari BASIC.

In Word Detective we demonstrate several different ways to handle strings. An examination of some of these methods should provide a basic understanding of the string-processing capabilities of Atari BASIC.

```
130 PRINT #6;CHR$(30);
```

The CHR$ function determines the single-character string whose numeric ATASCII (Atari ASCII) code is specified. This string is not assigned a name, and it does not need to be DIMensioned in the program. In line 130 above, the character whose ATASCII code is 30 is printed on the graphics screen. Code 30 corresponds to a 'less than' sign (<) in Color Register 1, which is black in Word Detective. This character is used to tell the user that the program is waiting for him/her to type a word and to show the position where the next word will be placed.

```
194 IF LEN(Z$)=0 THEN 120
```

The LEN function determines the LENgth of the string that was input into Z$. In line 194 it is used with an IF statement to check for a NULL or empty string. If the length of Z$ = 0, then *no* word was typed (just a RETURN) and the program tries again to get a word. If any characters had been typed, then the length of Z$ would not be zero.

```
190 IF Z$="THE RULE IS" THEN GOSUB 3000:X=R:GOTO 110
```

The entire contents of the Z$ string may be tested by comparing it to a string that is specified by a pair of quotation marks. If the Z$ string matches the string "THE RULE IS" exactly, character by character, then the subroutine at 3000 will be executed. If there is not an exact match, then the next instruction will be executed.

In addition to dealing with the entire text string, as in the IF or PRINT statements, Atari BASIC can deal with portions of the total string. To work with part of the string, BASIC needs functions to isolate that part of the string that is of interest. A substring function is provided for this purpose and it may be used in two ways, with either one (1) or two (2) subscripts:

1. Z$(A), which will isolate the substring within Z$ beginning with the A*th* character and extending to the last character.
2. Z$(A,B), which will isolate the substring within Z$ beginning with the A*th* character and ending with the B*th* character.

'A' may be a number or an algebraic expression. Its value is used to determine the starting character of the substring. 'B' works in the same way — its value is used for the ending character of the substring.

Let us assume that the input to the Z$ string was the word 'SAMPLE'.

```
220 F$ = Z$(1,1)
```

In this line of the program, F$ is set equal to 'S', the first character in 'SAMPLE'. The substring function in this case specifies the string within Z$ beginning *and* ending at character number one (1).

```
230 L$ = Z$(L)
```

L$ is set equal to 'E', the last character in 'SAMPLE'. This substring function specifies the string within Z$ from the L*th* character to the end. There are only L total characters in Z$ anyway, so L$ returns with the single rightmost character of Z$.

```
300 V$ = Z$(K,K)
```

V$ is set equal to the character at position K in the string Z$: 'S' when K is 1, 'A' when K is 2, and so on up to 'E' when K is 6. The substring specified consists of one character in the K*th* position in the string Z$.

Word Detective Variable Usage

Constants

CR	Location of current cursor row position
ERR	Location holding last error number
ERL	Location holding line number at which error occurred
SCR	Beginning of screen memory
SC$	Machine-language scroll routine

Variables

ADD	Offset used to change words' colors
C	Number of consonants
COL	Code of character on screen
D	Double letter flag
K	Loop counter
L	Length of input string
R	Random number rule
V	Number of vowels
X	Previous rule
Z	Temporary storage
F$	First letter of Z$
L$	Last letter of Z$

V$	Middle letter of Z$
X$	'I hate...' part of rule
Y$	'I love...' part of rule
Z$	Input string

Examples of String Functions

Y$ = ''DIFFICULT''	The string Y$ contains the word ''DIFFICULT''.
LEN(Y$) = 9	The LENgth of the string Y$ is 9.
Y$(1,1) = ''D''	The leftmost character of string Y$ is ''D''.
Y$(LEN(Y$)) = ''T''	The rightmost character of string Y$ is ''T''.
Y$(3,4) = ''FF''	The middle two characters starting at position 3 and ending at position 4 are ''FF''.
ASC(Y$) = 68	The ASCII decimal value of the first character, ''D'', in the Y$ string, is 68.
CHR$(73) = ''I''	The CHaRacter represented by the ASCII decimal value 73 is the letter ''I''.

Program Description

At line 20, the program goes to subroutine 9000 to initialize. The initialization procedure consists of the following steps:

Line 9010 sets an error TRAP pointing to line 500. Any errors occurring in the program will thus be handled by a routine beginning at 500.

Lines 9020 to 9030 DIMension the strings that will be used within the program. No string may be used unless its length is reserved with a DIM statement.

Line 9040 OPENs the keyboard (specified by 'K:') as device number 1. The '4' designates an input-only device. GET #1 statements within the program will use this device, waiting until a key is pressed on the keyboard before the next statement is executed.

Lines 9050 to 9070 set up the graphics mode and its colors. Mode 17 is a full-screen version of mode 1, an expanded text mode, which normally leaves a four-line text window at the bottom of the graphics screen. Colors 0 through 4 are defined as blue, black, green, red, and white.

Line 9080 sets CR to the address containing the current cursor row position. It also sets SCR to the beginning of the area in memory, which contains the data printed on the graphics screen. SCR must be defined *after* the desired graphics mode is set up, because the different modes use different areas of memory to hold the screen data. ERR, the location containing the number of the error that last occurred, and ERL, the location containing the number of the line at which it occurred, are also defined.

Line 9090 defines SC$, a string containing a machine-language routine used to scroll the graphics screen.

Lines 9110 to 9270 are simply a series of PRINT statements that display the operating instructions for the program. Notice that some words and phrases are printed in lower-case and/or inverse text. Graphics modes 1 and 2 (and their full-screen counterparts 16 and 17) do not actually print these words in lower case and inverse; instead these combinations control the colors with which they will be displayed. For example, lower/inverse prints in color register 3, which is red in Word Detective, and upper/inverse prints in color 2, which is green.

The GET statement in line 9280 waits for a key to be pressed and stores its ASCII value in Z. This value will not actually be used later, but the variable is required in the format of the GET statement. The screen is then cleared and the heading 'TRY A WORD' is displayed.

Line 9290 RETURNs the program to the mainline code.

The mainline of the program, lines 100 to 360, uses the random-number generator to select the current 'RULE', accept the word from the user, analyze the word for a number of characteristics, and then evaluate the word relative to the rule.

Line 110 generates a random number and then makes sure it is not the same number as X, the previous rule. If the new rule number is the same as the old one, then another number is generated until a new rule is selected.

Line 120 clears Z$, the string that will hold the user's guess word, and L, which will keep track of its length. A blank line is printed to position the cursor for the next word to be typed.

Line 130 prints a black ' ', which corresponds to the ATASCII code 30. This signals that it is the user's turn to type.

Line 140 accepts one key from the keyboard with a GET statement. The character is not stored as a string variable, however; its ASCII code is stored in the variable Z.

Lines 150 to 154 handle the DELETE key. If the code in Z is not 126 (the code for DELETE), then lines 152 to 154 are skipped. They are also skipped if L = 0, which means there are no characters yet to be deleted. Otherwise, L is decremented to take one character off the length of Z$, and the cursor is POSITIONed over the last character of the word. Then a space is printed over that character to erase it and the cursor is rePOSITIONed appropriately. A GOTO 140 is then used to look for another character.

Line 160 checks for a carriage return (code 155), which signals the end of the word.

Line 170 makes sure that the word typed in does not exceed 18 characters. If its length is 18, the only keys that will be accepted are 'DELETE' and 'RETURN'. This check is needed to prevent PRINTing off the right-hand side of the graphics screen or in the last column of the screen, which would signal a 'Cursor Out of Range' error. This error is handled in Word Detective with a machine-language scroll; the only time we want it to occur is upon an attempt to PRINT *below* the screen, in which case the scroll is the appropriate solution. Without the pointer ' >' at the beginning of each line, the word length could be extended to 19 characters. If line 170 did not take care of word length, the screen would scroll needlessly on long words.

Lines 180 to 185 print the last character typed on the screen and add it onto the end of Z$. L is incremented to account for the new letter. A GOTO 140 returns for another character.

Line 190: If the user types 'THE RULE IS', then the GOSUB 3000 will cause the current rule to be displayed. On return from line 3000, the program will GOTO 110 to generate a new rule and continue.

Line 192: If the user types 'THE RULES ARE', then the routine starting at line 4000 will display all of the rules on the screen.

Line 194: If no word was typed (only a carriage return), preparations are made for a new word at line 120.

The program reaches line 200 on any regular word. Each of four variables is set to an initial value of 0. The variables are: J for Odd or Even, D for Double letter, V for the Vowel counter, and C for the Consonant counter.

Line 220 uses a substring function to get the leftmost character of the string Z$ into the string variable F$.

Line 230 uses a substring function to get the rightmost character of the string Z$ into the string variable L$. The substring function uses L, the length of Z$, to specify the character to isolate.

Line 240 first divides the length L by 2 and then subtracts the whole, or integer, part of it. If there is any remainder left, the length is an odd number. If there is none, the length is even. If the length is odd, J is set to 1.

Line 250 simply tests for a single-letter word that obviously could *not* have a double letter and skips the next few lines if there is only one letter in the word.

Line 260 sets up a FOR...NEXT loop to test for a double character. The Z$(K,K) portion of line 270 isolates one character in the word, the Z$(K + 1,K + 1) isolates the next character in the word, and the IF function tests to see if these two consecutive characters are identical. If they are, it sets the double flag D to 1 for later testing; if not, it leaves the flag alone.

Lines 290 to 350 are a FOR...NEXT loop to count vowels and consonants in the word. Line 300 sets the V$ string variable equal to the next character in the word. Lines 310 and 320 test for the vowels and line 340 adds one (1) to the V counter for each vowel found. Line 330 adds one (1) to the C counter for any character that is not a vowel.

All variables that are going to be used for the various RULE tests below now have been calculated:

F$ is the First character
L$ is the Last character
J = 0 for an even number of characters
= 1 for an odd number of characters
L = the number of characters
V = the number of vowels
C = the number of consonants
D = 0 for no double letters
= 1 for double letters

Lines 500 to 530 take care of scrolling the graphics screen as it is needed. This routine is reached through an error TRAP, which is initially set up in line 9010. When a PRINT statement is attempted below the bottom line of the screen in graphics mode 1, an error occurs and lines 500 on are called. Line 500 checks the error number to be sure it was indeed a 'Cursor Out of Range' error. If it wasn't, lines 540 and 550 display a simple error message and STOP the program; otherwise line

510 calls the scroll routine with a USR function. Then the cursor is set at the bottom row, number 23, and the TRAP is reset. Finally, a return is made to the line that caused the error, using a calculated GOTO. Locations 186 and 187 hold the number of this line. For more details on this scroll routine, see ''Scrolling the Graphics Screen'' in the MASTER chapter.

Line 1000 branches to a subroutine based on the number of the rule in R. It will go to subroutine 1100 on rule 1, 1200 on rule 2, ..., 1500 on rule 5.

Line 1100 tests for odd or even number of characters. If J = 1, which means that the word has an odd number of characters and is disliked, it goes to line 2000 to display the word in green; otherwise the word is liked and goes to line 2010 to be displayed in red.

Line 1200 tests for the length to be less than 6 and, if it is, goes to line 2000 to turn the word green; otherwise it goes to line 2010 to turn the word red.

Line 1300 compares the number of consonants and vowels. If there are as many or more vowels as there are consonants, then it goes to line 2000 to turn the word green.

Line 1400 tests for the first letter of the word to be earlier in the alphabet than the last letter. If so, it goes to line 2000 to turn the word green.

Line 1500 tests for double letters. If there are double letters, it turns the word green.

Lines 2000 and 2010 each define an offset that will be used to change the color of the word to red or green. In graphics mode 1, different numeric codes specify not only the character to be printed but also the color in which to plot them. Thus, adding an appropriate offset to the character's numeric code is a simple way of changing its color. If the word is disliked, this routine will enter through line 2000, which sets the offset ADD to 128 for green. If the word turns red because it is liked, line 2010 is called; it sets ADD to 160.

Lines 2020 to 2050 form a FOR...NEXT loop that turns the word, character by character, into the appropriate color. Line 2020 uses the LOCATE function to find the ASCII code of the next letter in the word. The column and row specified in this function (I and PEEK(CR) in this case), indicate the screen position to check; LOCATE finds the character in this position and returns its ASCII code in the variable COL.

Line 2030 adds the offset ADD to the numeric code COL, skipping spaces (code 32) because they need not be changed.

The COLOR function is used in line 2040 with the new code COL to redefine the current character. The PLOT statement prints the new red or green character over its old blue counterpart.

Line 2050 executes a NEXT. It RETURNs to line 1010 when the loop is completed.

Lines 3000 to 3120 handle the response when the user types 'THE RULE IS'. Line 3010 RESTOREs the DATA pointer for the following READ statement so that the READ will start at the first DATA statement.

Line 3020 uses R, the number of the current rule, to READ the appropriate pair of strings that make up that rule. X$ corresponds to the 'I hate words with...' part of the rule, and Y$ is its opposite, the 'I love words with...' part of the rule. If R is 1, the first two strings in the DATA statements are READ as X$ and Y$ in line 3020. If R is 2, then the second time through the loop the next pair of strings are assigned to X$ and Y$, writing over their previous values. The same process is repeated for values of R greater than 2. The loop is thus a simple means of READing through the DATA statements to the desired pair of strings.

Lines 3030 to 3110 print the rule on the screen with the 'I hate...' message in green and the 'I love...' message in red. The PRINT statements are limited to one per line because of the error TRAPping for the scroll. More than one per line would repeat the first PRINT in the line infinitely.

Line 3120 RETURNs to line 190 in the program mainline.

Lines 4000 to 4030 handle the response when the user types 'THE RULES ARE'. Line 4010 PRINTs a blank line to separate the rules from the previously typed words.

Line 4020 uses a FOR...NEXT loop to output each of the five rules. First it RESTOREs the DATA pointer to the first item. Then it READs the next two strings from the DATA statements, assigning them to X$ and Y$.

Line 4030 goes to the subroutine at line 3090 to output the "I love..." information only (using Y$ only) and returns to the NEXT statement. On values of R from 1 to 4 this statement will go back to line 4020. On R equals 5 (the last value specified in the FOR loop in line 4020) the program will go to the next sequential statement (the GOTO 110), which selects a new random number for the next game.

That's all there is to it. Word Detective is a simple game and a simple program. If you study the program and fully understand it, then you are well on your way to understanding BASIC programming on the Atari.

Random-Number Generation

A 'random number' is a number that has no relation to the numbers that precede it or that come after it. Basically the concept is simple. For example, tossing a coin is one method of generating a random event. With an honest coin, what happens on any toss, or series of tosses, gives you no information as to what will happen on the next toss. Each toss is independent of the other tosses. This simple concept is important to a wide range of computer-oriented processing including statistics, simulations, and games.

While the concept of a random number is simple, generating a truly random number is difficult — particularly on a computer whose main role in life is to act with great precision and accuracy, not randomly! Atari BASIC has a function whose sole purpose is to generate pseudo-random numbers. A computer, by its very nature, cannot generate truly random numbers; the best it can do is to generate a series of numbers that *appear* to be independent.

Using the RND(X) function is quite simple. The value of X is of no importance, but it must exist all the same. Simply calling the RND function with any value for X will produce a pseudo-random number. The tricky part about the RND(X) function is that it returns a number in the range greater than or equal to 0 and less than 1. Often the program needs numbers in a range other than 0 to 1. The RND(X)-generated value can be converted easily into any range required.

```
110 R=INT(RND(0)*5)+1:IF R=X THEN 110
```

The RND(0) function is used to create a random number in the range of 0 to 1. This number is multiplied by 5 (the number of choices we actually require) to put the numbers in the range of 0 to 5. The INT function is used to reduce this to a number 0, 1, 2, 3, or 4. One is added to make the range 1, 2, 3, 4, or 5.

Programming Projects

There are many ways you can change Word Detective. Let's look at a few.

Reversing the Rules

To reverse a rule (e.g., to make 'WORD' like EVEN words and hate ODD words), two parts of the program must be changed. First, the test for ODD or EVEN must be reversed. This can be done by changing line 1100.

```
1100 IF J=1 THEN 2000
```

When J equals 1, the word is ODD and, in the original version, goes to the 'hate word' routine at 2000. To reverse the rule simply test for J equals 0. This will then branch to the 'hate word' routine when the word is EVEN.

```
1100 IF J=0 THEN 2000
```

Second, the strings that will be displayed for 'THE RULE IS' and 'THE RULES ARE' must be reversed.

```
10000 DATA ODD NO. OF LETTERS,EVEN NO. OF LETTERS
```

should be changed to read

```
10000 DATA EVEN NO. OF LETTERS,ODD NO. OF LETTERS
```

Changing the Rules

Any rule can be eliminated and a new rule put in its place. The new rule can be based on the existing tests ODD/EVEN, LENGTH, VOWELS/CONSONANTS, FIRST/LAST LETTER, and DOUBLE LETTERS, or new tests can be added. Using the existing tests, the FIRST/ LAST LETTER can be modified so that the word is 'liked' only when the FIRST letter *equals* the LAST letter.

```
1400 IF F$<L$ THEN 2000
```

becomes

```
1400 IF F$< >L$ THEN 2000
```

An entirely new test may be added. For example, if only words with the letter 'A' were to be 'liked,' then insert a new test in the loop that goes through the entire string looking for vowels and consonants (lines 300 to 350). Add

```
305 IF V$="A" THEN A=1
```

Initialize the variable A to 0 by adding A=0 to the end of the existing line 210:

210 J=0:D=0:V=0:C=0:A=0

Now replace the Doubles test with the Letter A test. Change

1500 IF D=1 THEN 2000

to

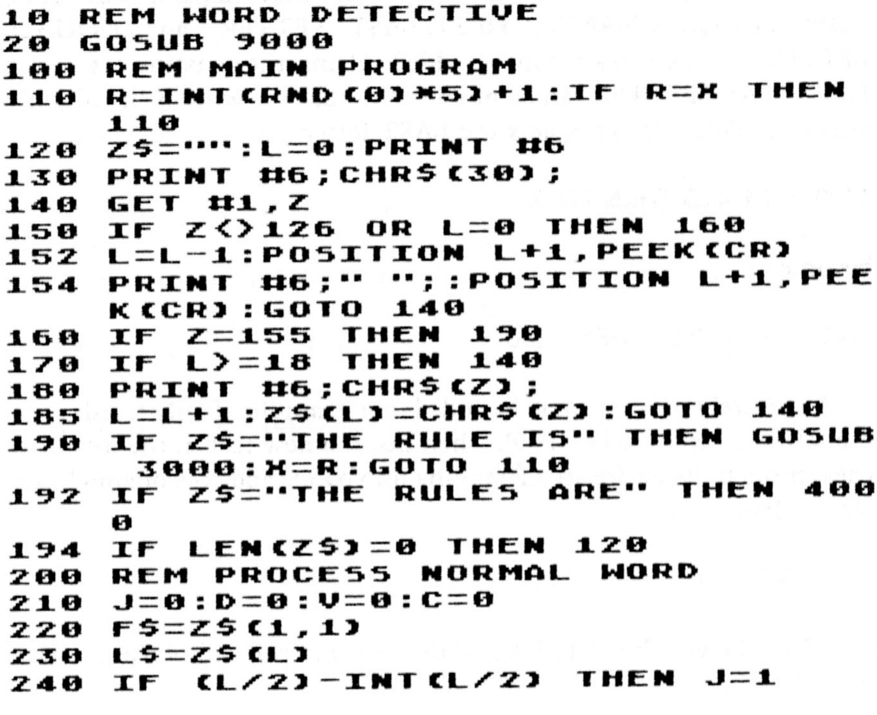
1500 IF A=0 THEN 2000

Finally, change the strings at 10040 to reflect the new rule:

10040 DATA LETTER A,NO LETTER A

Use your imagination! There are many rules, and you can even combine two rules to make a new one. For example, 'like' ODD letters in words that are seven characters or more in length, 'like' EVEN letters in words that are six characters or fewer in length, and 'hate' the other two possible combinations.

***Listing 1:* Word Detective**



```
10 REM WORD DETECTIVE
20 GOSUB 9000
100 REM MAIN PROGRAM
110 R=INT(RND(0)*5)+1:IF R=X THEN
    110
120 Z$="":L=0:PRINT #6
130 PRINT #6;CHR$(30);
140 GET #1,Z
150 IF Z<>126 OR L=0 THEN 160
152 L=L-1:POSITION L+1,PEEK(CR)
154 PRINT #6;" ";:POSITION L+1,PEE
    K(CR):GOTO 140
160 IF Z=155 THEN 190
170 IF L>=18 THEN 140
180 PRINT #6;CHR$(Z);
185 L=L+1:Z$(L)=CHR$(Z):GOTO 140
190 IF Z$="THE RULE IS" THEN GOSUB
     3000:X=R:GOTO 110
192 IF Z$="THE RULES ARE" THEN 400
    0
194 IF LEN(Z$)=0 THEN 120
200 REM PROCESS NORMAL WORD
210 J=0:D=0:V=0:C=0
220 F$=Z$(1,1)
230 L$=Z$(L)
240 IF (L/2)-INT(L/2) THEN J=1
```

```
250 IF L=1 THEN 290
260 FOR K=1 TO L-1
270 IF Z$(K,K)=Z$(K+1,K+1) THEN D=
    1
280 NEXT K
290 FOR K=1 TO L
300 V$=Z$(K,K)
310 IF V$="A" OR V$="E" OR V$="I"
    THEN 340
320 IF V$="O" OR V$="U" OR V$="Y"
    THEN 340
330 C=C+1:GOTO 350
340 V=V+1
350 NEXT K
360 GOTO 1000
500 IF PEEK(ERR)<>141 THEN 540
510 Z=USR(ADR(SC$),SCR+60,SCR+40,4
    40)
520 POKE CR,23:TRAP 500
530 GOTO PEEK(ERL)+PEEK(ERL+1)*256
540 PRINT "ERROR ";PEEK(ERR);" AT
    LINE ";PEEK(ERL)+PEEK(ERL+1)*2
    56
550 CLOSE #1:STOP
1000 REM SELECT TEST
1010 ON R GOSUB 1100,1200,1300,140
    0,1500:GOTO 120
1100 IF J=1 THEN 2000
1110 GOTO 2010
1200 IF L<6 THEN 2000
1210 GOTO 2010
1300 IF V>=C THEN 2000
1310 GOTO 2010
1400 IF F$<L$ THEN 2000
1410 GOTO 2010
1500 IF D=1 THEN 2000
1510 GOTO 2010
2000 ADD=128:GOTO 2020
2010 ADD=160
2020 FOR I=1 TO L:LOCATE I,PEEK(CR
    ),COL
2030 IF COL<>32 THEN COL=COL+ADD
2040 COLOR COL:PLOT I,PEEK(CR)
2050 NEXT I:RETURN
3000 REM THE RULE IS
3010 RESTORE
3020 FOR I=1 TO R:READ X$,Y$:NEXT
    I
3030 PRINT #6
3040 PRINT #6
3050 PRINT #6;"RULE #";R
3060 PRINT #6
3070 PRINT #6;"I HATE WORDS WITH"
3080 PRINT #6;X$
3090 PRINT #6
```

```
3100 PRINT #6;"i love words with"
3110 PRINT #6;Y$
3120 RETURN
4000 REM THE RULES ARE
4010 PRINT #6
4020 RESTORE :FOR R=1 TO 5:READ X$
    ,Y$
4030 GOSUB 3090:NEXT R:GOTO 110
9000 REM INITIALIZATION
9010 TRAP 500
9020 DIM Z$(20),F$(1),L$(1),V$(1)
9030 DIM X$(20),Y$(20),SC$(54)
9040 OPEN #1,4,0,"K:"
9050 GRAPHICS 17:SETCOLOR 0,9,8
9060 SETCOLOR 1,0,0:SETCOLOR 2,12,
    10
9070 SETCOLOR 3,4,8:SETCOLOR 4,0,1
    4
9080 CR=84:SCR=PEEK(89)*256+PEEK(8
    8):ERR=195:ERL=186
9090 SC$="hh▖Lh▖Kh▖Nh▖Mh▖Ph▖O ♥1K▖
    MfKP fL fMP fNFO%OI▶P FP%O⌐PP◆◆
    "
9100 REM GAME DESCRIPTION
9110 PRINT #6;"↰word detective
     "
9120 PRINT #6;"IF IF LIKE A WORD"
9130 PRINT #6;"  I MAKE IT red
    "
9140 PRINT #6;"IF I HATE A WORD"
9150 PRINT #6;"  I MAKE IT GREEN
    "
9160 PRINT #6;"FIGURE OUT MY RULE"
9170 PRINT #6;"BY TRYING DIFFERENT
    "
9180 PRINT #6;"WORDS.
    "
9190 PRINT #6;"WHEN YOU HAVE"
9200 PRINT #6;"FIGURED IT OUT, OR"
9210 PRINT #6;"YOU GIVE UP, TYPE:"
9220 PRINT #6;"  the rule is"
9230 PRINT #6;"TO DISPLAY THE RULE
    ."
9240 PRINT #6;"TO DISPLAY ALL OF"
9250 PRINT #6;"THE RULES, TYPE:"
9260 PRINT #6;"  the rules are
    "
9270 PR:PRINT #6;"ANY KEY TO CONTINUE
    "
9280 GET #1,Z:PRINT #6;"↰TRY A WOR
    D"
9290 RETURN
10000 DATA ODD NO. OF LETTERS,EVEN
      NO. OF LETTERS
```

```
10010 DATA 5 LETTERS OR FEWER,6 LE
    TTERS OR MORE
10020 DATA VOWELS=>CONSONANTS,VOWE
    LS < CONSONANTS
10030 DATA FIRST LETTER < LAST,FIR
    ST LETTER=>LAST
10040 DATA DOUBLE LETTERS,NO DOUBL
    E LETTERS
```

The Answer Machine

The Intelligent Computer

The Answer Machine allows you to have a discussion with your Atari, or to have your Atari tell you a story or answer your questions. All you have to do is type in a question that can be answered with a YES or NO; The Answer Machine will do the rest. To enter a question simply type it, ending with a question mark ('?'). The Answer Machine will provide you with an answer immediately. It may not always be right, but it is always fun!

Program Description

The Answer Machine may be one of the easiest programs you ever try! The routine uses only twelve BASIC statements and four BASIC functions. The BASIC statements are CLOSE, DIM, END, GET, GOTO, IF...THEN, LET, ON...GOSUB, OPEN, PRINT, REM and RETURN. The four BASIC functions are ASC, CHR$, INT, and RND.

Line 10 is a REMark statement.

Line 20 OPENs the keyboard as the input device. This allows the GET statement to be used in line 110. It DIMensions string variable **S$** to reserve space for the typed input, and **A$** for the input character.

Lines 30 to 80 simply PRINT the instructions for The Answer Machine. The **?** is equivalent to the statement **PRINT**.

Line 100 LETs the string variable **S$** be equal to the NULL string, LETs the numeric variables **C** and **T** be equal to 0, and then PRINTs a question mark. Where is the LET in line 100? It is implied. S$ = 0 is equivalent to LET S$ = 0. The LET does not have to be specified and is normally omitted.

Line 110 GETs a character from device 1 which was OPENed in line 20 as the keyboard into the variable A. It then PRINTs the value as a character, converting it to a displayable form using the CHR$(A) expression.

Line 120 adds the value of the character to the total variable T; adds 1 to the counter variable C; and places the character into the Cth location in the string variable S$.

Line 120 tests to see if the character just input was **not** a question mark. If it was not, then the program goes back to line 110 for the next character. If it was a question mark, then the program continues at the next line, line 130.

Line 140 ends the program if S$, the string input, is only a question mark. The cursor is placed on the next line. The keyboard device is then CLOSEd and the program executes an END to return to BASIC.

Line 150 converts T, which has the total of the ASCII values of all of the characters in the string S$, into a number in the range 1 to 10.

Line 160 uses the ON...GOSUB to branch to a YES, NO, or MAYBE. On return from the subroutine the program does a GOTO 100 to start the next question.

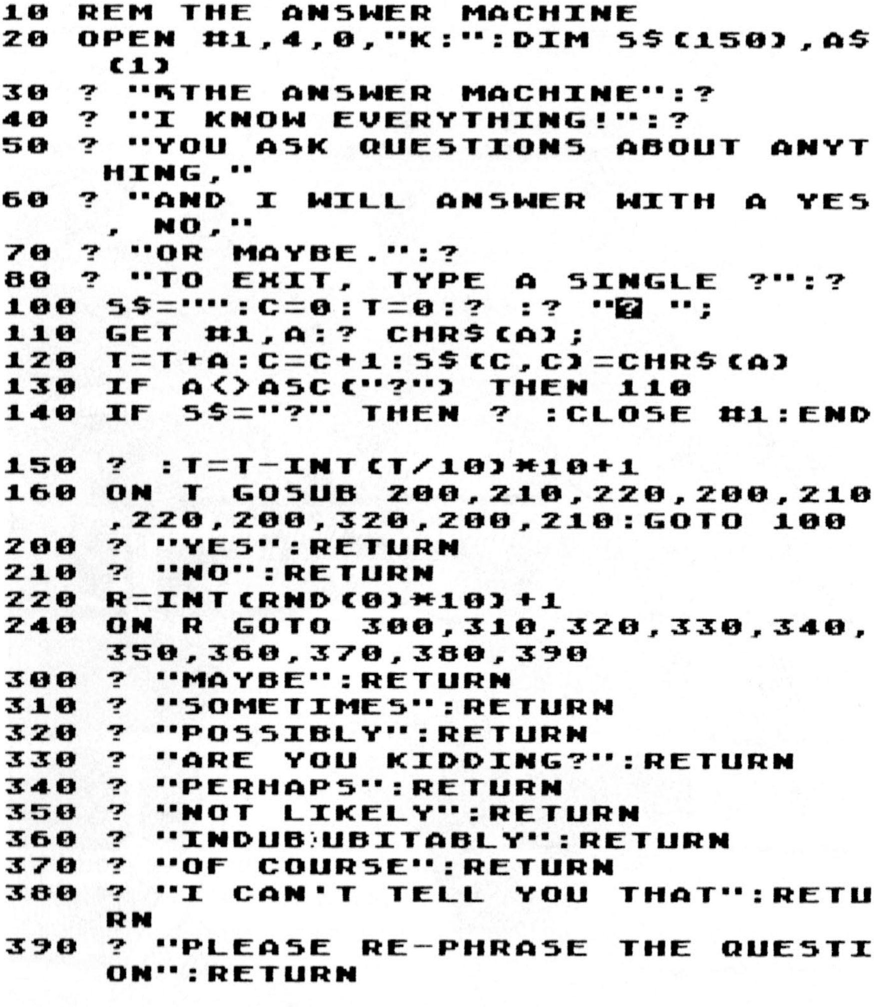
Line 200 PRINTs the YES, 210 PRINTs the NO, and 220 generates a random number to select among 10 choices for the MAYBE.

Line 240 uses the ON...GOTO to branch to one of 10 possible responses.

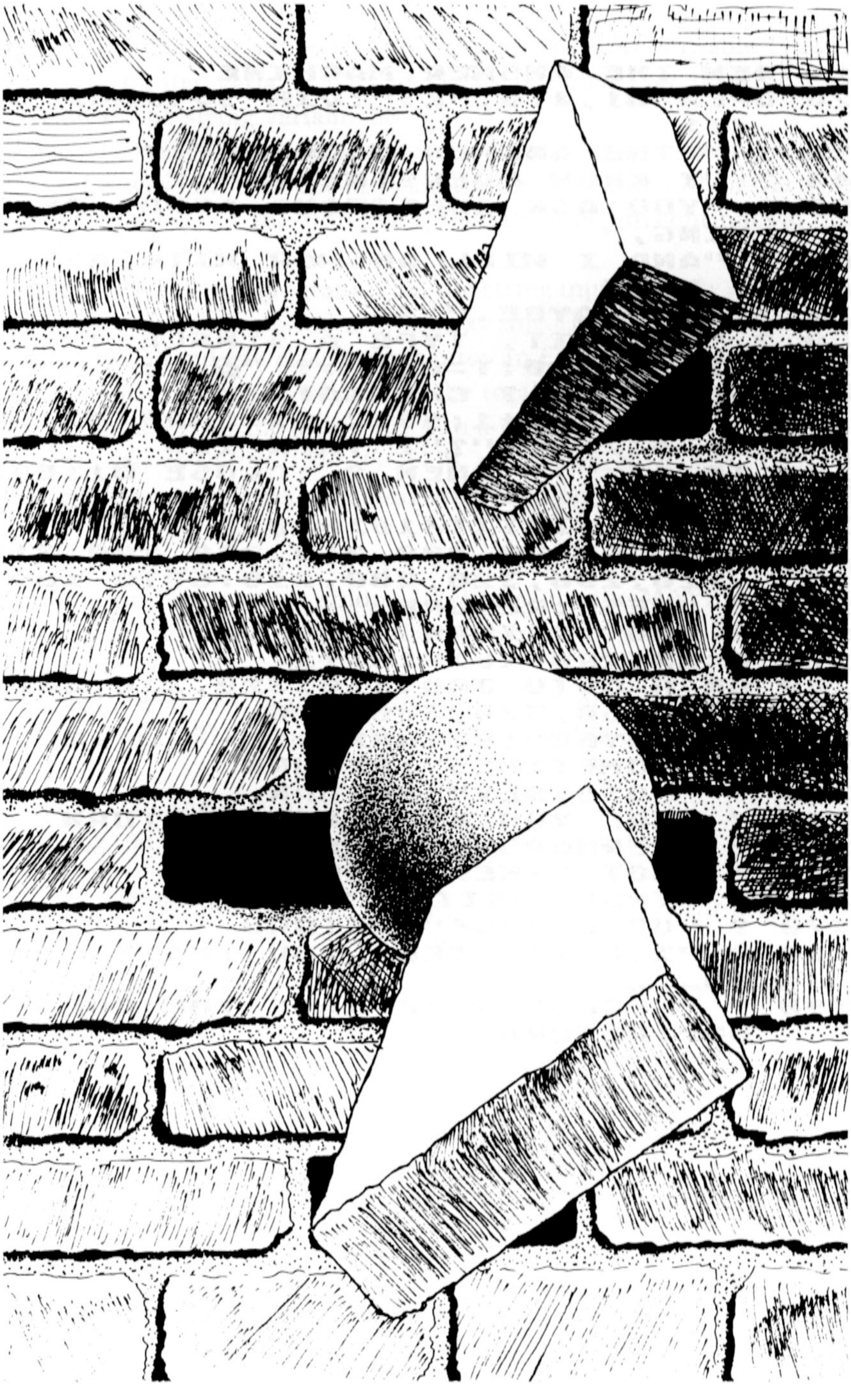
Lines 300 to 390 are the PRINTs for each of the responses. Each ends in a RETURN, which will return to line 160 at the GOTO 100.

That's all there is to it. You can change the responses easily by changing the text in lines 300 to 390, or change the number of YES and NO responses by changing the line numbers in line 180. This simple program shows how your computer can be made to *appear* intelligent without really having a whole lot of 'smarts'.

Listing 2: **The Answer Machine**

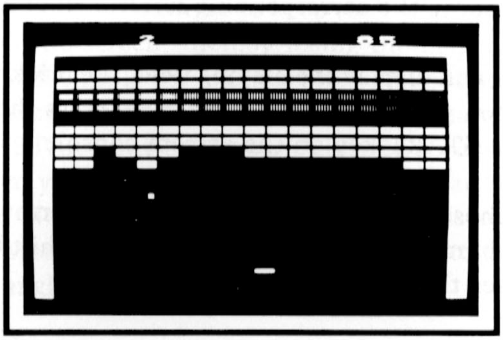


```
10 REM THE ANSWER MACHINE
20 OPEN #1,4,0,"K:":DIM S$(150),A$
   (1)
30 ? "↰THE ANSWER MACHINE":?
40 ? "I KNOW EVERYTHING!":?
50 ? "YOU ASK QUESTIONS ABOUT ANYT
   HING,"
60 ? "AND I WILL ANSWER WITH A YES
   , NO,"
70 ? "OR MAYBE.":?
80 ? "TO EXIT, TYPE A SINGLE ?":?
100 S$="":C=0:T=0:? :? "⍰ ";
110 GET #1,A:? CHR$(A);
120 T=T+A:C=C+1:S$(C,C)=CHR$(A)
130 IF A<>ASC("?") THEN 110
140 IF S$="?" THEN ? :CLOSE #1:END

150 ? :T=T-INT(T/10)*10+1
160 ON T GOSUB 200,210,220,200,210
   ,220,200,320,200,210:GOTO 100
200 ? "YES":RETURN
210 ? "NO":RETURN
220 R=INT(RND(0)*10)+1
240 ON R GOTO 300,310,320,330,340,
   350,360,370,380,390
300 ? "MAYBE":RETURN
310 ? "SOMETIMES":RETURN
320 ? "POSSIBLY":RETURN
330 ? "ARE YOU KIDDING?":RETURN
340 ? "PERHAPS":RETURN
350 ? "NOT LIKELY":RETURN
360 ? "INDUBIUBITABLY":RETURN
370 ? "OF COURSE":RETURN
380 ? "I CAN'T TELL YOU THAT":RETU
   RN
390 ? "PLEASE RE-PHRASE THE QUESTI
   ON":RETURN
```



Breakup

A Brick Wall Demonstration



Get ready to hit the bouncing ball with your bumper and knock a few bricks out of the wall. The farther away the brick is, the more points you will get for knocking it out. If you are dexterous enough to knock out the entire wall of bricks, don't become over-confident; the game will continue with a screen of bricks that are even more difficult to knock out.

Breakup is a simple graphics-display game that presents the principles of animation using player/missile graphics to move characters on the screen and test for collisions. It includes a "ball" that moves around the screen, rebounds from struck objects, and knocks bricks from a wall. It also includes a player-controlled "bumper" to keep the ball from going out of bounds and being lost, a defined playing field with three walls from which to bounce the ball, and eight rows of blocks. The amount of points received for hitting the blocks depends on their color and distance from your bumper.

The game keeps score by color; 5 points for the green blocks at the bottom, 15 points for the blue blocks in the middle, and 20 points for the yellow-orange blocks at the top. If you clear the entire screen, you are awarded an extra ball and the paddle shortens by one dot, moving closer to the blocks. This continues, screen after screen, until the bumper is as small and as close to the bricks as it can be. In addition, the points received for hitting the blocks are increased by three points. That is, when you are playing the second screen, the green blocks are worth eight points, and on the third screen they are worth eleven points, etc. Unlike the size of the bumper, the values for the bricks have no limit and increase in value for as long as you can play the game.

Operating Instructions

1. Load in 'BREAKUP' from your disk and RUN it.

2. First you are asked whether you will play with paddles or the keyboard. Choose the corresponding letter — P or K.

3. The program will display the playing field, the brick walls, and your bumper. When you are ready to start play, press the button on the paddle or the START key on the system console.

4. If you have chosen the keyboard, use the cursor arrow keys←and→ to move the bumper left and right. Holding the shift key at the same time increases the speed of the bumper.

5. If for some reason you halt program execution with the Break key, you must hit the SYSTEM/RESET key before reRUNing. (More on this later.)

The Program

The ball starts from a random position at the bottom of the screen and travels upwards, hitting a brick. This causes the brick to disappear, adds the appropriate amount of points to your score, and rebounds the ball towards the bottom. Here is the challenge: You must hit the ball back with your bumper to keep the ball from travelling out of bounds and off the screen, thereby losing the ball. If you are successful, the ball will simply hit another brick and bounce back. If you miss the ball, a buzzer will sound and the program halts until you hit the paddle or the START button. You are allowed six balls plus one extra for each screen you clear. Also, the angle and relative speed of the ball increase the closer you hit the ball to the ends of the bumper. Hitting the ball near the center of the bumper helps to restore the ball to a less radical angle.

Breakup's Animation:
The Idea of Player – Missile Graphics

The animation in Breakup is done with the Atari's player-missile graphics capabilities. I used PM graphics because of the very fast speed of the moving figures (players — such as the ball and paddle) on the screen. Also, PM graphics makes it easy to test for collisions, providing a faster and more challenging game. In fact, even machine-language versions of this game, which are generally fast due to the nature of machine language, employ PM graphics because it is easy to use.

A "player" is a zone on the screen that is eight pixels wide and extends vertically off both the top and bottom of the screen. A missile is generally a very thin player; it is only two pixels wide and, also, extends past the top and bottom of the screen. There are several locations (registers) that correspond to the characteristics of each of the players and missiles, such as color, pixel width, priority, collision detection, and horizontal position of each. The reason the players and missiles are so thin when compared to their height stems from the fact that there is no vertical position register for them — only a horizontal position register. This means that in order to move a player vertically (as needed by the ball, for example), we have to physically redraw the player either higher or lower in memory. Before we describe the locations of PM graphics, lets first discuss how the Atari handles PM graphics.

The Atari allows four separate players and four missiles on the screen — or five players if you combine all four missiles and treat them as a player. There are, in general, two types of players: those drawn in one-line resolution and those drawn in two-line resolution. One-line resolution is just that; the players are drawn out one scan line at a time. Two-line resolution is simply drawing the players out two scan lines at a time. One-line resolution produces better pictures and requires 2K of memory to store. Two-line resolution requires 1K of memory to store. Each player in one-line resolution takes 256 bytes to describe (one for each scan line from the top of the screen to the bottom). Each player in two-line resolution takes up to 128 bytes since each byte corresponds to two scan lines instead of one. Note that not only does the one-line resolution take up more room, but the memory used must start on a 2K boundary (the starting location must be divisible by 2048). The two-line resolution memory can start on a 1K boundary (the starting location must be divisible by 1024). Therefore, we should be somewhat careful in our placement of the player-missile memory.

The Atari finds this memory through its base address register at location 54279. That is, location 54279 tells the Atari where to go to find out how the players look. Since the location is only one byte in size (it is only one location) it has to hold the *page number* of the PM memory. Studious readers will remember from the Programmable Characters chapter that a page is 256 bytes. Therefore, a single byte, which can hold any number from 0 to 255, will be able to address any one of the 256 pages in the Atari. The paging method is simply a way for the Atari to find its way around with only one byte telling it where to go.

That's the Way the Ball Bounces

Another time-saving feature of PM graphics is its collision-detection capability. A collision occurs when any player or missile touches something other than the background. This capability allows the program to find out, with a single PEEK statement, if anything is hitting any one of the players or missiles or if they are touching anything. This makes the whole checking routine for the ball and paddle collisions very fast.

Collision detection works simply. There is a register for every possible PM collision. Now look at the player-to-playfield collisions register. This is the location that is read constantly to see if the ball (player 0) has hit something so that the appropriate ball-movement routine can be activated. Similarly, the player-to-player collision register is read to see if the paddle has hit the ball.

Additional Information

To make the colored bricks, I used redefined characters in graphics mode 2. I used characters simply because of their color capability and because they are easy to draw and erase. Remember that the characters in graphics mode 2 can be displayed as four different colors. I redefined the character '$' to a 7-dot × 5-dot brick. The different colors are achieved the same way the different colored space invaders are displayed in the chapter on Programmable Characters.

Look at figure 1 again. Note that the first 384 bytes of memory (in double-line mode) is always unused; and the first 512 bytes remain unused because this program does not enable the missiles (everything is done with the players). That means that we have 512 bytes sitting on a 1K boundary doing nothing — a perfect place for the graphics 2 character set. By using this space for the somewhat altered character set, we can store information that would normally require 1½K (1K for the PM storage and ½K for the character set) in only 1K.

The actual movement of the ball is calculated in BASIC and executed in machine language. This is because, as mentioned earlier, PM graphics moves figures quways around this, but using a machine-language routine is the easiest.

Breakup Variable Usage

Strings

BALL$	Holds the PM description for the ball
M$	Holds the machine-language block-move routine
M2$	Holds the machine-language ball-move routine

Arrays

A	Holds the possible ball angles resulting from a bumper hit
P	The number of points per line (for the bricks)
PAD	One-byte PM description for the paddle

Constants

CHBASE	Character-set base register
COLP0-COLP3	Color register locations for the four players
DMACTL	Direct memory access control register
HITCLR	Collision clear register
GRACTL	Graphics control register

P0-P3	Horizontal position register for four players
PMBASE	Player-missile base register
SIZEP0-SIZEP3	Size register for four players
P0PL	Player 0 to player collision register
P0PF	Player 0 to playfield collision register
P1PL	Player 1 to player collision register

Numerics

BALL	The starting location of the ball description
BALLXY	Starting location for the move-ball routine
BMOVE	Starting location for the block-move routine
BL	Number of balls left in the game
BPF	Variable holding the ball to character collisions
BPL	Variable holding the ball to wall collisions
C	The variable used in all character inputs
CTRL	Holds the line number of the bumper routine to be used
CN	Loop variable for character internal number loop
D	Dummy variable used for assigning values to arrays in loops
H,V	Horizontal and vertical displacement for the ball movement
I	Increment for the bumper when controlled by the keyboard
NB	Number of bricks left on the screen
P,P9	Value of the keyboard left arrow, right arrow, and shift key
PB	Variable holding the paddle to ball collisions
PP	Paddle position
PY	Paddle vertical position on the screen
Q,QQ	Loop variables
RY	Real Y position on the screen for the bricks
SC	Score
START	Starting location of the PM and character-set area
STP0	Start of the player 0 (ball) memory area minus 2 for ball movement
SZ	Index to determine the bumper's size, screen after screen
U	Dummy variable for the USR statements
X,Y	X and Y coordinates for the ball
ZERO	Used to either move the paddle up one dot or not move it at all, depending on the screen number and bumper size

Program Description

The routine to move the ball and paddle, test for collisions, and do anything else that involves animation is contained entirely in lines 100 to 190. Note that this routine is almost at the very top of the program; all initialization and other routines are done below it. This is a programming trick to speed up the game; the more lines that exist above a routine, the slower that routine will be. This is because when a GOTO is encountered, BASIC starts looking for the destination line number from the top; that is, it checks each one until it finds where it has to go. This is time consuming, especially if you have a lot of lines above the routine. Therefore, all routines that are not time-dependent, such as the intialization and score-keeping routines, appear below the movement routine. This way, no time is wasted.

Line 10 dimensions all the strings and arrays used by the program: M$ holds the block-move routine discussed in other chapters; M2$ holds the ball-movement routine; BALL$ holds the player-missile description for the ball (only twenty bytes worth); A holds the possible angles resulting from a collision with the bumper; P holds the points for each line of bricks on the screen; and PAD holds the byte that describes how the paddle looks from screen to screen. These features will be discussed more thoroughly below.

Line 20 calls the initialization routine at line 30000.

Line 30010 lowers the top-of-memory pointer by 1K (four pages) to make room for the player-missiles and new character set. Fortunately, location 106 points to a 4K boundary, so subtracting 1K from this location insures that the location will be on a 1K boundary (divisible by 1024). The graphics 1 screen is initialized right after the pointer is moved so that the computer can readjust the appropriate pointers after losing the 1K of memory.

In line 30012, START is assigned the address of the new memory area and the two machine-language routines are loaded in.

Line 30014 POKEs the starting location with a 0 and propagates it through the entire 1K by moving 1023 bytes from the starting location to the following location.

Line 30020 uses the block-move routine to move the standard character set from ROM to the new memory allocated just before the PM memory area. This allows us to redefine the few characters necessary and to keep the rest as they are.

Lines 30030 and 30040 make players 2 and 3 into the left and right walls of the game. These walls could have been a character, as is the top wall above the bricks, but they were made as players so that a single check could be made to determine whether or not the ball should bounce horizontally.

Line 30044 puts the description of a 7-dot-wide paddle into the player 1 area.

Lines 30050 to 30054 redefine the two characters whose internal value is 4 and 5 ('$' and '%', respectively) to the brick and solid block. The latter is used to draw the wall across the top of the screen.

Lines 30060 to 30066 define the values of all the constants in the program. The majority of these are the locations for characteristic changes in the player missiles.

Line 30070 opens the keyboard for later input. For convenience's sake, it will remain open during the entire program execution.

Line 30080 sets up all the game values. (See the Variable Usage table for details.)

Lines 30082 to 30090 load in the values for the A, P, and PAD arrays.

Lines 30100 to 30120 asks if you want to play with paddles or the keyboard. CTRL holds the line number of the appropriate bumper routine.

Lines 30200 to 30260 are a routine to initialize the screen. The PM graphics and character set are enabled and the bricks and walls are set up.

Line 50 stops the game until either the paddle or START button is pressed. This gives the user a moment to reconnoiter before the ball is released.

The entire game is controlled by lines 100 to 190.

In line 100, the horizontal and vertical displacements are added to the X and Y coordinates of the ball. Then the paddle is moved (CTRL is the line number of the appropriate routine). A machine-language routine that moves the ball within the player is then called. This routine is given the following values: x coordinate, y coordinate, the starting location of the ball description, the start of player 0 (where to put the ball), and how many bytes of the ball description to move. Player 0 is moved horizontally (only one location to change), player 0 is moved vertically, and the collision registers are cleared. The routine then waits for 1/60th of a second and returns to BASIC. The collision registers are cleared by the Atari internally whenever location 53278 is POKEd with any number. It takes 1/60th of a second for the collisions to register.

Line 110 assigns the needed collision registers to the following variables: BPF (ball to character collisions), BPL (ball to wall collisions), and PB (paddle to ball collisions). Y is then checked to see if the ball has been missed.

Line 150 turns off any sound that may have been turned on by a previous collision. BPF is then checked to see if it has hit playfield 0, 1, or 2 (one of the bricks). If a collision has occurred, then control is passed to the brick routine at line 200.

Line 160: If the ball hits playfield 3, then it reflects (negates) the vertical displacement and a sound is made.

Line 170: If the ball hits either wall, then it is horizontally reflected and a sound is made.

Line 180: If the paddle hits the ball then it is vertically reflected, H is assigned the appropriate angle of horizontal reflection, and a sound is made.

Line 190 returns control back to line 100 in the event that none of the above has occurred.

Lines 200 to 210 handle the brick-collision routine.

Line 200 prints a space over the brick, effectively erasing it, adds the appropriate amount of points to the score, vertically reflects the ball, makes a sound, and subtracts 1 from the number-of-bricks variables (NB).

Line 202 prints the score. If NB is 0, then control is passed to the new screen routine.

Line 210 passes control back to the main loop.

The value of CTRL is set in the routine at 30100 and is either a 300 or a 400. CTRL is the line number of the appropriate bumper routine. If the game is controlled by the paddles, then CTRL is 300; if it is controlled by the keyboard, then CTRL is 400. Line 300 assigns the variable PP with the paddle position negated and moved to the right a little. The paddle value is negated so that paddle movement will correspond to the bumper movement on the screen.

Lines 400 to 420 move the paddle left or right one pixel depending on whether the left or right arrow key is pressed. If the shift key is pressed, then the paddle is moved by fivew pixels in the direction specified. This allows the paddle to speed up if necessary.

Lines 500 to 550 are the missed-ball routine. If the number of balls left is greater than 0, then the game values are re-initialized, the number of balls left is decremented by one, and the game resumes at line 50. If the number of balls is 0, then the game is over and you are asked whether you wish to try again. If you specify 'N', then the top-of-memory pointer is reset to its original spot and the program halts. If you specify 'Y', the top-of-memory pointer is reset and the program is reRUN. If the program is stopped *via* the Break key and then rerun, the top of memory will be even lower than it was before. Continuing this will cause the computer to eventually run out of memory and crash. You should hit SYSTEM/RESET whenever you stop the program *via* the Break key.

Lines 600 to 690 handle the screen-cleared routine. One pixel is subtracted from the paddle, which is moved up three lines. This is done at line 610 by block moving the description bytes for the paddle up one byte three times. Between each move upward, a sound is made briefly and a delay occurs so that the paddle change is obvious. SZ is a flag telling the program that there is still room to move the paddle upward three lines and that the paddle can still be shortened. It is incremented every time the paddle is raised. If SZ reaches 7, then the paddle is no longer raised or shortened every time the screen is cleared. The points received for each brick struck is increased by 3 for each consecutive screen. When this routine occurs, the game values are reinitialized and the game resumes at line 50.

The DATA statements in lines 32010 and 32110 hold the two machine-language routines in string form. These are read into the appropriate strings during the initialization routine.

The rest of the DATA statements in lines 32210, 32220, 32310, 32410, and 32510 hold the values for the new characters in the character set, the paddle angles, the points received for the blocks per line, and the paddle sizes per new screen, respectively. They are also read into their appropriate variables during the initialization routine.

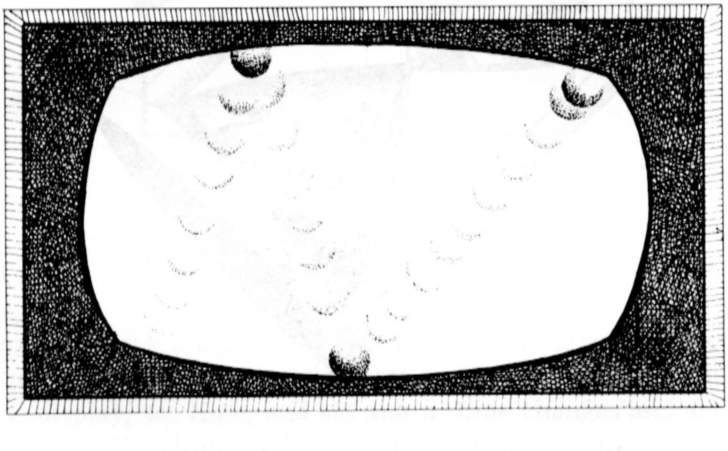
Listing 1: **Breakup**

```
10  DIM M$(54),M2$(99),BALL$(20),A(
    7),P(23),PAD(6)
20  GOSUB 30000
50  IF PTRIG(0) AND PEEK(53279)<>6
    THEN 50
100 X=X+H:Y=Y+V:GOSUB CTRL:POKE P1
    ,PP:U=USR(BALLXY,X,Y,BALL,STP0
    ,14)
110 BPF=PEEK(P0PF):BPL=PEEK(P0PL):
    PB=PEEK(P1PL):IF Y>111 THEN 50
    0
150 SOUND 0,0,0,0:IF BPF>0 AND BPF
    <8 THEN 200
160 IF BPF>7 THEN V=-V:SOUND 0,80,
    10,10
170 IF BPL>3 THEN H=-H:SOUND 0,80,
    10,10
180 IF PB/2<>INT(PB/2) THEN V=-V:H
    =A(X-PP+1)*(BPL<=3)+H*(BPL>3):
    SOUND 0,50,10,10:GOTO 100
190 GOTO 100
200 RY=INT((Y-16)/4):POSITION INT(
    (X-48)/8),RY:? #6;" ";:SC=SC+P
    (RY):V=-V:SOUND 0,100,10,10:NB
    =NB-1
```

```
202 POSITION 15,0:PRINT #6;SC:IF N
    B=0 THEN 600
210 GOTO 100
300 PP=250-PADDLE(0):RETURN
400 I=1:P=PEEK(764):P=P-64*(P>64):
    P9=PEEK(53775):IF P9<248 THEN
    I=4
410 IF P9<>255 THEN PP=PP-I:IF P=7
     THEN PP=PP+2*I
420 RETURN
500 POSITION 5,0:PRINT #6;BL:IF BL
    >0 THEN SOUND 0,200,12,14:FOR
    Q=1 TO 100:NEXT Q:SOUND 0,0,0,
    0:GOTO 550
502 FOR Q=200 TO 100 STEP -2:SOUND
     0,Q,10,10:SOUND 1,300-Q,10,10
    :NEXT Q
504 FOR Q=1 TO 100:NEXT Q:SOUND 0,
    0,0,0:SOUND 1,0,0,0
510 POSITION 0,5:PRINT #6;"
                        TRY again (Y/N)
    ?                             "
520 GET #1,C:IF CHR$(C)<>"Y" AND C
    HR$(C)<>"N" THEN 520
522 IF CHR$(C)="Y" THEN POKE 106,P
    EEK(106)+4:GRAPHICS 1:POKE GRA
    CTL,0:RUN
530 CLOSE #1:POKE 106,PEEK(106)+4:
    GRAPHICS 0:POKE GRACTL,0:END
550 BL=BL-1:POSITION 5,0:PRINT #6;
    BL:X=INT(144*RND(0)+56):Y=111:
    H=+2:V=-2:PP=124:GOTO 50
600 U=USR(BMOVE,START+512,START+51
    3,127)
602 FOR Q=200 TO 0 STEP -5:SOUND 0
    ,Q,10,14:SOUND 0,Q/2,10,10:NEX
    T Q:SOUND 0,0,0,0:IF PY=82 THE
    N PY=85:ZERO=1
610 FOR Q=1 TO 3:U=USR(BMOVE,START
    +641,START+640+ZERO,127)
612 SOUND 0,30,8,14:FOR QQ=1 TO 20
    :NEXT QQ:SOUND 0,0,0,0:FOR QQ=
    1 TO 20:NEXT QQ:NEXT Q
620 SZ=SZ+1:IF SZ=7 THEN SZ=6
630 POKE COLP1,15:SOUND 0,200,10,1
    4:PY=PY-3:POKE START+640+PY,PA
    D(SZ):SOUND 0,0,0,0:POKE COLP1
    ,78
634 FOR Q=0 TO 23:IF P(Q)>0 THEN P
    (Q)=P(Q)+3
636 NEXT Q
640 BL=BL+1:NB=144:GOSUB 30200:X=I
    NT(144*RND(0)+56):Y=111:H=-2:V
    =-2:PP=124
690 GOTO 50
```

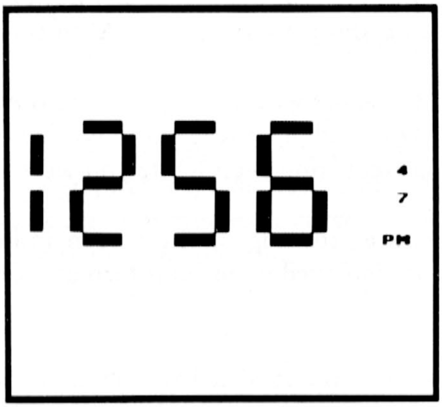
```
30000 REM ---INITIALIZATION---
30010 POKE 106,PEEK(106)-4:GRAPHIC
    S 17
30012 START=256*PEEK(106):READ M$,
    M2$:BMOVE=ADR(M$):BALLXY=ADR(M
    2$)
30014 POKE START,0:U=USR(BMOVE,STA
    RT,START+1,1023)
30020 U=USR(BMOVE,256*PEEK(756),ST
    ART,512)
30030 POKE START+788,255:U=USR(BMO
    VE,START+788,START+789,91):REM
     L WALL
30040 POKE START+916,255:U=USR(BMO
    VE,START+916,START+917,91):REM
     R WALL
30044 POKE START+740,254:REM PADDL
    E
30050 FOR CN=4 TO 5
30052 FOR Q=CN*8 TO CN*8+7:READ D:
    POKE START+Q,D:NEXT Q
30054 NEXT CN:REM NEW CHARS
30060 P0=53248:P1=53249:P2=53250:P
    3=53251:P0PF=53252:P0PL=53260:
    P1PL=53261:HITCLR=53278:DMACTL
    =559:GRACTL=53277
30062 SIZEP0=53256:SIZEP1=53257:SI
    ZEP2=53258:SIZEP3=53259:COLP0=
    704:COLP1=705:COLP2=706:COLP3=
    707
30064 PMBASE=54279:CHBASE=756:STP0
    =START+512-6
30066 BALL$="♥♥♥♥♥♥@@♥♥♥♥♥♥♥":BALL=
    ADR(BALL$)
30070 OPEN #1,4,0,"K:"
30080 X=INT(144*RND(0)+56):Y=111:H
    =+2:V=-2:BL=5:NB=144:PY=100:PP
    =124
30082 FOR Q=0 TO 7:READ D:A(Q)=D*2
    :NEXT Q:REM PADDLE ANGLES
30084 FOR Q=0 TO 23:READ D:P(Q)=D:
    NEXT Q:REM POINT VALUES
30090 FOR Q=0 TO 6:READ D:PAD(Q)=D
    :NEXT Q:REM PADDLE SIZES
30100 POSITION 0,5:PRINT #6;" pADD
    LES OR kEYBOARD";
30110 GET #1,C:IF CHR$(C)<>"P" AND
     CHR$(C)<>"K" THEN 30110
30120 CTRL=400:IF CHR$(C)="P" THEN
     CTRL=300
30200 POSITION 0,0:POKE PMBASE,PEE
    K(106):POKE CHBASE,PEEK(106)
30210 POSITION 0,1:PRINT #6;"▀▀▀▀▀
    ▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀"
30212 PRINT #6;" $$$$$$$$$$$$$$$$
```

```
     $   $$$$$$$$$$$$$$$$$$   $$$$$$$
     $$$$$$$$$$$  $$$$$$$$$$$$$$$$$
     $ "
30220 PRINT #6;" ┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤
     ┤  ┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤   ┤┤┤┤┤┤┤
     ┤┤┤┤┤┤┤┤┤┤┤  ┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤┤
     ┤"
30230 POKE P2,48:POKE P3,201:POKE
     COLP0,14:POKE COLP1,78:POKE CO
     LP2,70:POKE COLP3,70
30240 POKE SIZEP0,0:POKE SIZEP1,0:
     POKE SIZEP2,0:POKE SIZEP3,0
30250 POKE DMACTL,42:POKE GRACTL,2
30260 POSITION 5,0:PRINT #6;BL:POS
     ITION 15,0:PRINT #6;SC
30900 RETURN
32000 REM ---BLOCK MOVE ROUTINE---
32010 DATA hh▖Lh▖Kh▖Nh▖Mh▖Ph▖O ♥1K
     ▗MfKP fLfMP fNFO%OI▶P FP%O▖PP◆
     ◆
32100 REM ---BALL MOVE ROUTINE---
32110 DATA hhh▬♥Phh▖Mh▖Lh▖Kh▖Nh┴eM
     ▖M%Ni ♥▖Nhh(╱1K▗M@♥PW▬←P▬ ▜TI↑PY
     ▬ ▜TI▙ PY◆
32200 REM ---NEW CHARS ($,%)---
32210 DATA 0,0,0,127,127,127,127,1
     27
32220 DATA 255,255,255,255,255,255
     ,255,255
32300 REM ---PADDLE ANGLES---
32310 DATA -2,-1.5,-1,-.5,.5,1,1.5
     ,2
32400 REM ---POINTS PER LINE---
32410 DATA 0,0,0,20,20,15,15,0,5,5
     ,5,5,0,0,0,0,0,0,0,0,0,0
32500 REM ---PADDLE SIZES---
32510 DATA 0,126,124,60,56,24,16
```



ATARI Clock

A Digital Time Display



The Atari Clock program displays the time in large seven-segment digits on your TV screen. Features include AM/PM indication, seconds display, set time, and set alarm. Line changes and additions are given to add a chime and to improve the appearance of the digits.

Operating Instructions

1. LOAD the program CLOCK from the *Mastering Your Atari* disk and RUN it.

2. Set the time. The space bar toggles between hours and minutes. With 'HOURS' indicated in reverse, press the ' + ' key once. The displayed time will advance by one hour. If you hold the key down it will advance continuously. The ' – ' key works the same way to turn back the time. When you have the right hour, press the space bar to set the minutes. Be sure the AM/PM indication is correct.

3. Set the minutes using the ' + ' and ' – ' keys. This time will take effect as soon as you hit RETURN, so you might want to set the clock a minute ahead and wait until exactly the right moment to hit RETURN.

4. The clock will continue to display the correct time until you stop the program or turn off the computer.

5. To change the time or set the alarm, hit any key except 'A'. You will be presented with a short menu. Select 'A' to set the alarm.

6. The alarm is set the same way as the time. Set the alarm time using the ' + ' and ' – ' keys and press RETURN. Notice that the clock has maintained the correct time even while you were setting the alarm.

7. A message appears at the top of the screen indicating the alarm time. This time is displayed in 24-hour format. Subtract 1200 to get PM times.

8. When the alarm goes off it will beep about once a second for a minute. You can turn off the alarm by hitting the 'A' key, which acts as a toggle to set the alarm or to turn it off.

Atari Clock Information

Take a look at a digital clock or watch. You will see that the numbers are composed of seven linear segments. Each of the ten digits is produced by lighting the proper segments in this seven-segment display. Figure 1 shows how each of the ten digits is produced using only these seven segments.

Atari Clock simulates the operation and appearance of a digital clock. When you run the program, you will see that both the horizontal and vertical segments consist of three reversed spaces. Each of the segments is the result of printing a string of characters on the screen. What characters are included in these strings? The reversed spaces are obviously included. To produce a vertical segment we must print three spaces in a vertical line. Between spaces the cursor must be moved back and down. This is accomplished by including cursor control characters in the string. When a cursor control character is printed, the corresponding cursor movement is executed.

Each of the strings for the two kinds of segments has a corresponding string that does not include the reversed spaces. These are used to turn a segment off when the current digit does not include it. Each space removes the reversed space, making the segment appear to turn off. Thus there are a total of four strings to be printed — the 'on' and 'off' versions of both horizontal and vertical segments. These are combined into two final strings (H$ and V$) — the horizontal and vertical segment strings, respectively. Each of these two strings consists of two substrings of equal length, the first designating the 'off' segments and the second designating the 'on' segments. Each substring is shown in figure 2, along with a graphic respresentation of how it is printed on the screen. The black arrow indicates a cursor movement, with the arrow's head showing the cursor position at the end of each move. A solid circle indicates where the cursor begins, and an asterisk indicates where the cursor is positioned at the end.

There are two unseen strings other than the blank segments. These consist entirely of cursor movement characters. These strings, BK$ and UP$, are used to move the cursor across the digit for the next segment. BK$ moves the cursor from the end of a horizontal segment back to the position under the first space of the horizontal segment. We are now ready to print a vertical segment. BK$ is shown with its printed results in figure 3.

BK$ is also used to ready the cursor to print a horizontal segment as shown in figure 4.

Figure 1. How the ten digits look in seven segments.

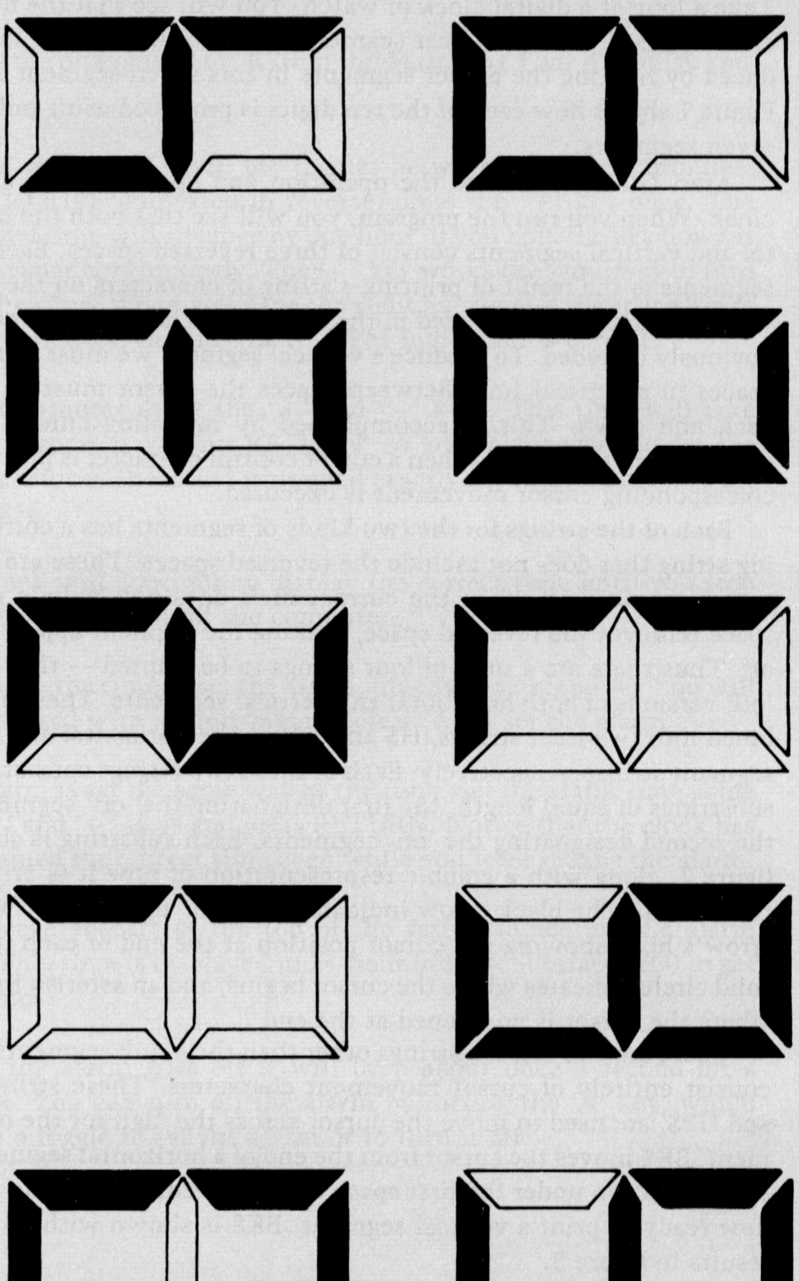


Figure 2: H$(1,5),H$(6,10),V$(1,7),V$(8,14) produce the off and on segments on the screen.

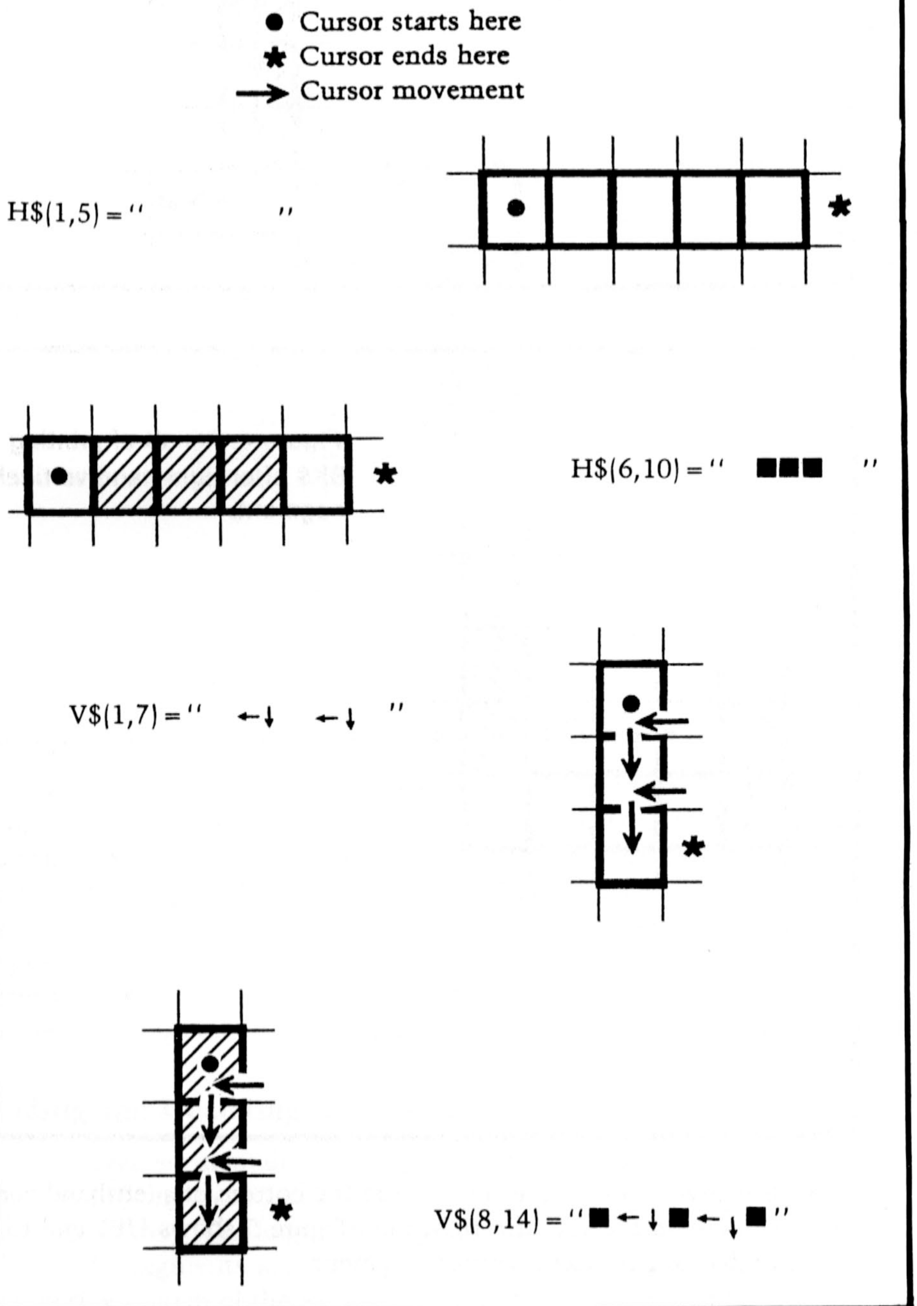
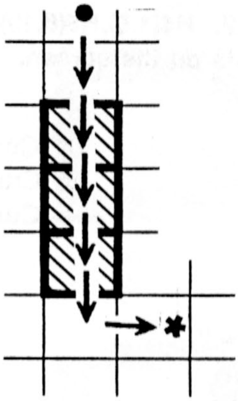


Figure 3: Result of printing BK$ after horizontal segment.

BK$ = "→→→→→↓"



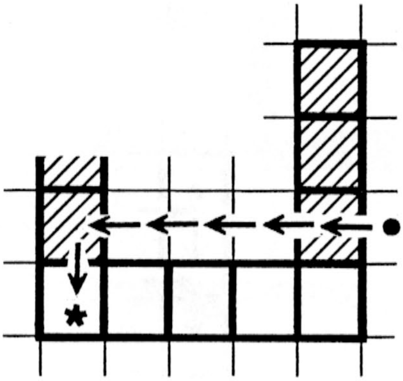
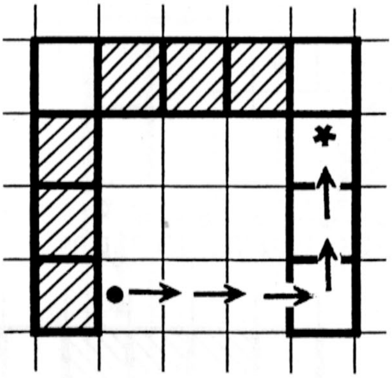


Figure 4: Result of printing BK$ after right-hand vertical segment.

UP$ is used to move the cursor from the bottom of a left-hand vertical segment to the top of a right one. Figure 5 shows UP$ and the result of printing it after a vertical segment.

Figure 5: Result of printing UP$ after left-hand vertical segment.



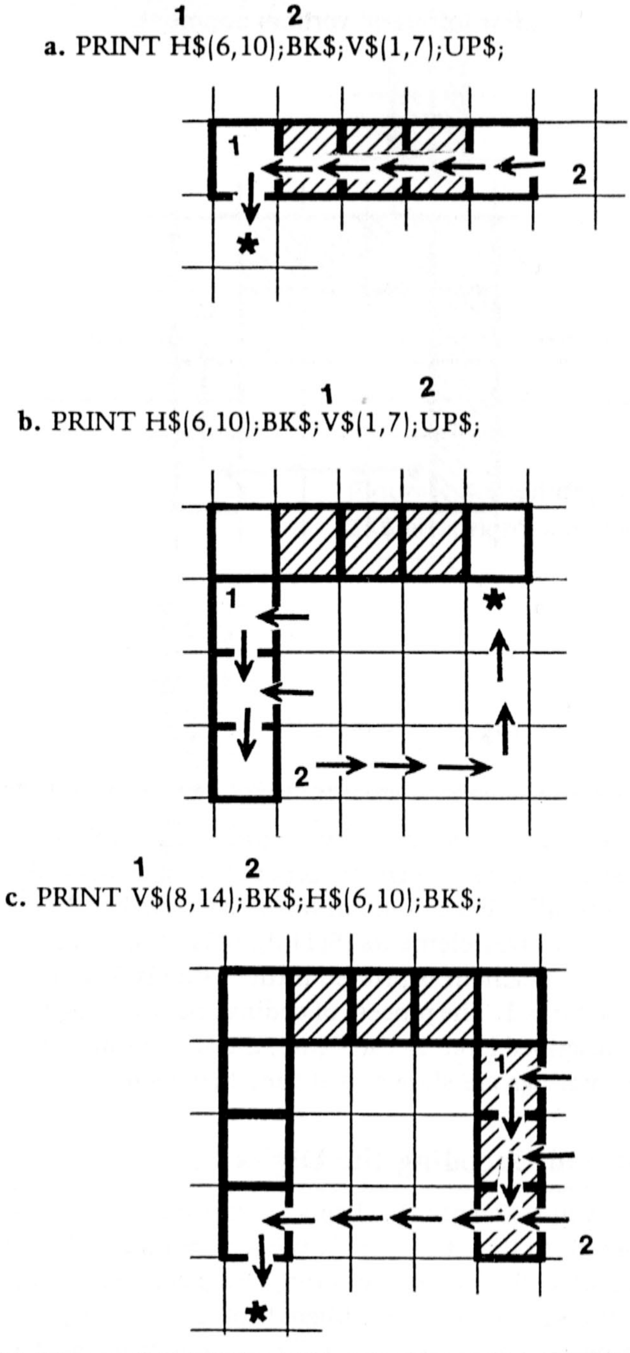
UP$ = "→→→↑↑"

Figure 6 shows, segment by segment, the process of printing the proper sequence of strings to produce a seven-segment '2' on the screen. Actually this is done in lines 1510-1570 of ATARI Clock. The values for the array elements (S(TI,0), S(TI,1), etc.) are always either 1 or 0. A blank segment is printed if this value is 0, and a filled segment is printed if it is 1. The process of coding and decoding the segment patterns is described later. In the figure, the starting and ending characters of the substrings are shown as if they had been calculated already.

Coding and Decoding the Digits

The seven-segment digit display is controlled by a two-dimensional array, S(I,J). In lines 8510-8520, the loop counter I designates the current digit (0 to 9). The loop counter J designates the segment number of a digit; the segments are numbered from 0 to 6. Figure 7 shows the numbering system of the Atari Clock display and the data that is read

Figure 6: Producing a seven-segment '2'.

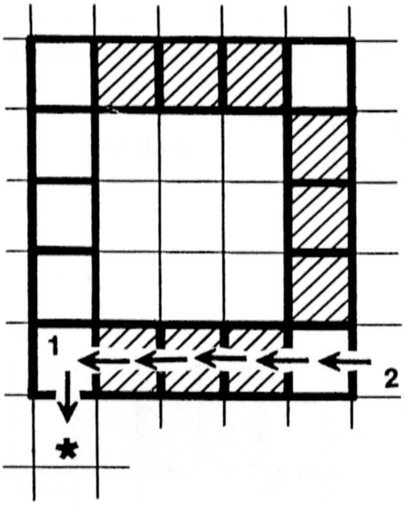


a. PRINT H$(6,10);BK$;V$(1,7);UP$;

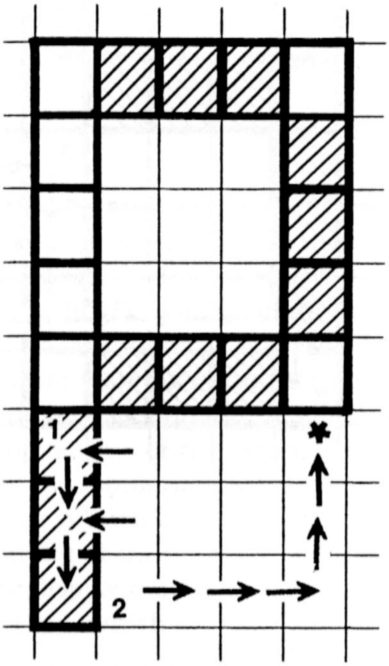
b. PRINT H$(6,10);BK$;V$(1,7);UP$;

c. PRINT V$(8,14);BK$;H$(6,10);BK$;

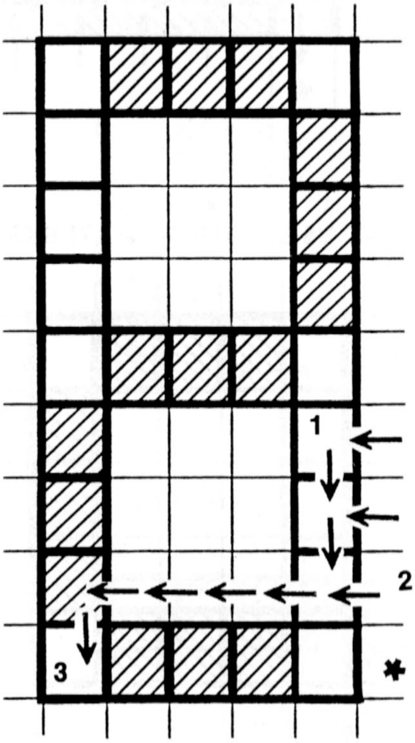
```
                 1         2
d. PRINT V$(8,14);BK$;H$(6,10);BK$;
```



```
        1            2
e. PRINT V$(8,14);UP$;V$(1,7);BK$;H$(6,10)
```



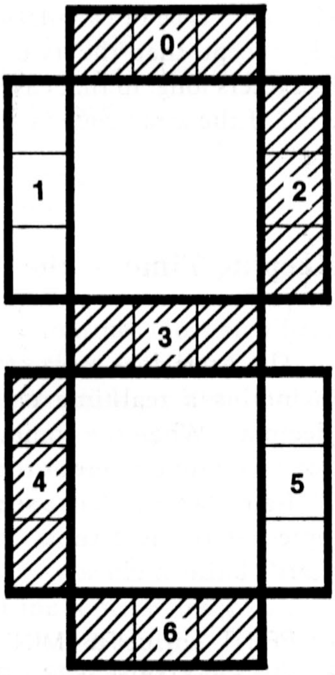
```
                      1       2   3
f. PRINT V$(8,14);UP$;V$(1,7);BK$;H$(6,10)
```



in for each digit. A '0' represents a segment that is turned off, and a '1' represents a segment that is turned on. For the digit '2' (I=2), segments 0, 2, 3, 4, and 6 are 'on'. Therefore, the array elements S(2,0), S(2,2), S(2,3), S(2,4), and S(2,6) are set to 1. S(2,1) and S(2,5) are set to 0 to designate the 'off' segments of the digit.

Figure 7: Coding of segment data.

	Segment							
	6	5	4	3	2	1	0	
Digit								
0	1	1	1	0	1	1	1	119
1	0	1	0	0	1	0	0	36
2	1	0	1	1	1	0	1	93
3	1	1	0	1	1	0	1	109
4	0	1	0	1	1	1	0	46
5	1	1	0	1	0	1	1	107
6	1	1	1	1	0	1	1	123
7	0	1	0	0	1	0	1	37
8	1	1	1	1	1	1	1	127
9	1	1	0	1	1	1	1	111



The decoding process consists of converting the array element 1's and 0's into segments. This is done within the digit-display routine (lines 1500-1570). Each PRINT statement within this routine prints a string corresponding to a horizontal or vertical segment, which is either off or on. Where do these strings come from? The horizontal segment strings are combined as one string, H$, and are actually substrings of it. The same is true of the vertical segment strings, which comprise V$. To print the appropriate segment string of H$, the PRINT statement must use the substring function of the format H$(S,E), with S designating the starting character of the substring and E the ending character. In the digit display routine, S and E are calculated using the values in the array S(I,J), which are either 0 or 1:

```
1510 PRINT H$(S(TI,0)*5 + 1,S(TI,0)*5 + 5);BK$;
```

In the program line above, TI is the current digit. S(TI,0) is therefore the code for the 0*th* segment of that digit, or the top horizontal segment. If the value of that code is 0, the values of S and E in the substring function are 1 and 5. If the code is 1, S and E are calculated as 6 and 10. The PRINT statement thus prints the first five characters of H$ (the 'off' segment substring) if the array code is 0, and it prints the last five characters of H$ (the 'on' segment substring) if the code is 1. The same type of calculation is used to print the correct substring of V$, except that the two vertical segment strings are each seven characters long. In that case the subscript arguments S and E result in 1 and 7 if the array code is a 0 or 8 and 14 if the code is a 1.

Keeping Time — the Atari Way

The Atari does most of the work when keeping time. The system maintains a real-time clock in memory locations 18, 19, and 20 (decimal). When the computer is first turned on, the three timer locations contain 0's. From that time on, the timer is incremented every TV frame (a sixtieth of a second). The time may be changed, though, to represent the real time. The following short program demonstrates Atari's built-in clock.

```
10 PRINT "      HRS  MIN  SEC":POKE 752,1
20 TI=PEEK(18)*65536+PEEK(19)*256+PEEK(20)
30 H=INT(TI/216000)
40 M=INT(TI/3600-H*60)
50 S=INT(TI/60-H*3600-M*60)
60 POSITION 5,2:PRINT H;"    "
70 POSITION 10,2:PRINT M;"    "
80 POSITION 15,2:PRINT S;"    "
90 GOTO 20
```

RUN the program. The time will be displayed continually at the top of the screen.

Press the BREAK key, wait a few seconds, then type RUN again. The time shown is current because the timer's value is maintained even when no program is running. The only time the Atari fails to keep proper time is when the disk drive or cassette recorder is used.

Atari Clock Variable Usage

Constants

K	Address containing code of last key pressed
AP$	AM or PM indication
BK$	Moves cursor ready to print horizontal segment or left vertical segment
CR$	Carriage-return character
H$	Off and on horizontal segment string
HM$	HOURS/MINUTES indication during time/alarm-set
SA$	SET TIME or SET ALARM message
UP$	Moves cursor ready for right vertical segment
V$	Off and on vertical segment string
ZR$	Six 0's, used to clear time strings

Variables

AF	Alarm flag (1 set, 0 off)
H	Real-time hours in read-time routine, most significant byte of time counter in write-time routine
I	Loop counter, indicates current digit position
J	Loop counter
M	Real-time minutes in read-time routine, middle byte of time counter in write-time routine
MF	Mode flag (1 set minutes, 0 set hours)
PM	AM/PM flag (1 PM, 0 AM)
S	Real-time seconds in read-time routine, least significant byte of time counter in write-time routine
TF	Flag used to determine if time has been changed in set-time routine
TH	Hours in AM/PM, used in AM/PM correction routine
TI	Value of current digit
TM	Hours in AM/PM, used in set-time routine
X	In main program, code of last key pressed. In read-time and write-time routines, value of time counter
T$	Character from keyboard
TA$	Alarm-set time string
TI$	Current time string, as last read by read-time routine
TM$	Temporary time string

Arrays

D( )	Last values displayed in static time-display routine
S(,)	Contains on/off data for each digit's segments
T( )	In time/alarm-set routine, used for hours and minutes portions of time. In main program, used for last values of each digit position

Atari Clock Program Description

Initialization sequence (lines 10-30): The program starts with a call to the initialization subroutine (8000), where constants are defined and the segment coding for the digits is set up. Next, the time-setting routine 5500 is called (with T$ set to 'T').

Program mainline (lines 100-220): Line 100 sets the array of position flags to all – 1's. The elements of this array normally contain the last value for each digit in the time display. Minus 1 is an impossible digit value, so this ensures that all digits are printed on the first run. Line 110 calls the read-time subroutine (500), which reads the current time and converts it into TI$. It then prints the two digits of the seconds, one over the other at the right side of the screen. Substring functions are used to extract one character each from the fifth and sixth positions of TI$.

Lines 120-190 go through the first four digits of TI$, displaying them as seven-segment digits on the screen. Because the fourth digit changes the most often, the FOR...NEXT loop counts down from 4 to 1. This is accomplished by using the extension 'STEP – 1' in line 120. Without this extension, the loop would execute only once. Line 120 also extracts the proper digit from TI$ using a substring function. This returns a string character, so we need to turn the character into a number in TI using the VAL function. Line 130 tests the keyboard status location (K), which contains a code for the last key pressed. If the code (X) is 255, then no key has been pressed and program control resumes at line 150. Any code other than 255 signals that a key has been pressed, in which case line 140 is executed. First, 255 is POKEd into the keyboard status location to reset it. Then a call is made to the mode-select subroutine (3000), after which the program returns to the beginning of the mainline. Since this occurs in the middle of a FOR...NEXT loop, we can't jump out of it without a little tidying up. Before the GOTO 100, we must terminate the FOR...NEXT loop by setting I to the ending value and performing a NEXT statement.

Because the time found in TI$ is in 24-hour format rather than AM/PM, the digits for the first two positions have to be adjusted for PM times. Subroutine 4000 performs this adjustment. Line 150 allows us to perform the full conversion (by starting at 4000) when the display reaches the second digit. However, when the display reaches the first digit, the conversion has already been performed and we need to extract only the first digit from TM$ (by starting at line 4070). Remember that the FOR...NEXT loop (lines 120-190) counts down from 4 to 1. When I is 3 or 4, the subroutine at 4000 is skipped.

Line 170 compares the current digit with the last digit printed in that position. If they are the same, there is no need to print the same thing over again and a branch is made to the NEXT statement (line 190). Line 180 readies the cursor for the next digit and prints the appropriate large digit in that position with a call to subroutine 1500. Finally, the 'previous digit' array element is set to the value of TI. The NEXT statement goes back to 120 for the next digit.

In line 200, either 'AM' or 'PM' is printed below the seconds display, depending on the value of PM. PM can have a value only of 0 (for AM) or 1 (for PM). A little formula is used within the PRINT statement to calculate the proper substring of AP$ to print. In line 210 the alarm-set flag is tested. If it is set (not 0), then the alarm-check routine at line 1000 is executed. The cursor is positioned above the first digit of the time, and the hours and minutes portion of TA$, the alarm-set time, is printed.

In line 220, the attract mode is overridden with a POKE to memory location 77 (decimal). A value of 7 in that location signals attract mode; any lower value means it is off. If this POKE is not made repeatedly, the attract mode may present a strange-looking display. Finally, GOTO 110 starts the mainline over again but without clearing the previous digit array.

Read – time routine (lines 500-560): The Atari timer, located in memory locations 18, 19, and 20, is copied into X in line 500. Then in line 510 the time X is split up into H, M, and S (hours, minutes, and seconds). The INT function of a positive number drops any fraction and returns with a whole number. Lines 520-550 initialize TI$ by setting it equal to ZR$, a string of 0's defined in line 8110, and copy the values of H, M, and S into it. If one or more of those numbers is less than the two-digit space it must take up, a '0' is already there to fill it out because TI$ was initially equal to ZR$.

Write – time routine (lines 600-620): Line 600 assembles the hours, minutes, and seconds of TM$ into X, the time to be set. Each part of the time must be multiplied by a factor of 60 to get a result in sixtieths of a second. Then, H, M, and S, are used to represent the most significant, middle and least significant bytes, respectively, of the three-byte timer. These are in turn POKEd into their appropriate memory locations, 18-20.

Alarm – check routine (lines 1000-1020): This routine consists of a simple comparison of the first four characters of TI$ and TA$ (the time for which the alarm is set). If the two strings are found to be equal, then the alarm routine is executed. As long as the alarm is set (AF does not equal 0) and the times are equal, the program flow will go from line

210 to line 1000 to line 2000 and back to the mainline at 210. If the alarm is turned off (by pressing 'A') or the time advances to the next minute, then the alarm will stop.

Print – digit routine (lines 1500-1570): This routine uses the array elements from S(,) to print a seven-segment digit with the appropriate segments on or off. Each element in the array is set to 1 or 0 by the set-up routine starting at line 8500. A 1 indicates that a segment is on, and a 0 indicates that a segment is off. Lines 1510-1570 print a series of strings, end to end. Each substring of H$ or V$ is calculated using the value of the appropriate array element S(TI,X), with TI representing the current digit and X representing the segment number. H$(6,10) and V$(8,14) are 'on' segments, while H$(1,5) and V$(1,7) are 'off' segments. For a more detailed description, see the main text and the accompanying figures.

Alarm routine (lines 2000-2060): The background color is turned blue in line 2010 by the SETCOLOR command. The numbers after this statement determine the color register to be changed, its hue, and its luminance. In this case, register 2 (background in Graphics Mode 0) is set to a light blue (color 8, luminance 10). While the screen is blue, the sound is turned on with the SOUND statement. The sound produced is in voice 0, with a high pitch (50), low distortion (10), and mid-to-high volume (10). The FOR...NEXT loop in 2020 determines the duration of the sound. Upon exiting the subroutine, the sound is turned off, and the background color is returned to the normal red color.

Mode - select routine (lines 3000-3100): If the code for the last key pressed was a 63 (an 'A'), then line 3010 toggles the value of AF between 1, which is read by an IF...THEN statement as true, and 0, which is read as false. The NOT instruction in Atari BASIC changes true conditions to false and false to true, so NOT(1) = 0 and NOT(0) = 1. If the last key pressed was not 'A', then the menu of options is printed. Line 3060 looks for a character from the keyboard. The three acceptable characters are tested in lines 3070-3090. If no acceptable character is received, then line 3100 branches back to 3060 for another character. A carriage return returns to the mainline without changing either the time or the alarm-set time. An 'A' causes the alarm-set subroutine (5000) to be executed and a 'T' causes the time-set (5500) subroutine to be executed. Each of these subroutine calls is made without a GOSUB...RETURN. Since the subroutines themselves end with RETURNs, it is more efficient to GOTO the beginning of the appropriate subroutine; the RETURN encountered sends program control back to the mainline.

PM correction routine (lines 4000-4080): The left two digits of TI$ (representing hours) are extracted with a substring function. Next, this two-character string is converted to a number in TH using the VAL function. At the same time, TM$ is cleared by setting it equal to ZR$. In line 4020, if TH is greater than 23, then 24 is subtracted from TH until it falls in the range of 0 to 23. If TH is greater than 11, TM is set to TH minus 12; otherwise TM is set to TH. This move assures us that TM is between 0 and 11 (AM/PM format), while TH is still between 0 and 23 (24-hour format). If TM is 0 (midnight) by this time, it becomes a 12 for the digit-display routine. Then the value of TM is copied into the first two characters of TM$. If TM$ contains only one digit, then a '0' is already there to fill out the string to two characters.

Line 4070 extracts the proper digit from TM$ to return to the mainline. I's value is set by the FOR...NEXT loop in the mainline. Here it can be only 1 or 2. One character, starting with the I*th* position, is extracted. This is returned in TI. Line 4070 also sets the AM/PM flag PM to 1 if TH (the hours in 24-hour format) is greater than or equal to 12. If TH is less than 12, then PM is set to 0.

Set alarm (lines 5000-5050): Line 5010 sets the alarm-set flag to 1 and changes the screen color to blue. The current alarm-set time TA$ is copied into the temporary string TM$ before the call to the alarm/time-set routine (line 6000). Then TM$, presumably changed, is copied back into TA$. Before returning, the screen color is changed back to red.

Set time (lines 5500-5550): This routine is similar to the alarm-set routine. The screen color is changed to green and the current time is read into TI$ using the read-time routine at 500. TF is a flag used to determine whether or not any change has been made in the time. If a change has been made, then the changed TM$ is copied into TI$ and, using a call to the write-time routine at 600, POKEd into the timer memory locations. If no change was made in the time, TI$ (which has been marching onward in value) is not affected. TF is set to 0 to start and changed to 1 only if the ' + ' or ' – ' key is hit in subroutine 6000. Therefore, if TF is still 0, then no change has been made. The screen color is returned to red before the return.

Alarm/time set (lines 6000-6530): This routine, used for both setting the time and for setting the alarm, starts with a clear-screen followed by the appropriate heading 'SET TIME' or 'SET ALARM'. The expression 'T$ = "T" ' evaluates to 1 if true and to 0 if false. This value is in turn used to determine the correct substring of SA$ to print. MF is set to 0, indicating that the hours will be set first. Line 6020 sets the array of digit display flags to all – 1's. Like the similar statement in line 100, this assures that every digit will be displayed the first time. The hours and minutes are extracted from TM$ using substring functions. The VAL function then converts the characters into numbers in T(1) and T(2). T(1) represents the hours in 24-hour format, while TM is used to represent the hours in AM/PM format. Lines 6040-6070 perform this adjustment. Line 6080 tests T(1) and sets PM to 1 if it is after noon. It also clears TM$ by setting it equal to ZR$.

Lines 6100 and 6110 form a new TM$ from TM and T(2). Line 6090 calls the static-display routine (line 7000) to display the time on the screen. Next the cursor is positioned under the time display, ready to print the 'HOURS/MINUTES' message. 'HOURS' or 'MINUTES' is printed in reversed characters, depending on the state of the flag MF.

Line 6200 looks for a keyboard character. The four acceptable characters are tested in lines 6210-6240. If no acceptable key is pressed, then line 6240 returns the flow to line 6200 for another key. A space toggles MF and branches back to line 6130 to update the 'HOURS/ MINUTES' display. A '+' or '−' adjusts the hours or minutes appropriately, depending on the value of MF. If MF is 0 then T(1) (hours) is adjusted and if MF is 1 then T(2) (minutes) is adjusted. Both conditions result in a branch to line 6300. If the return key is pressed then the flow is sent to 6250.

Lines 6300-6330 adjust for over- or under-flows on the hours and minutes. If T(1) is greater than 23, then it is set back to 0. If it is less than 0 then it is set to 23. The minutes, T(2), are set using the 'modulo' function in line 6320. This removes any multiples of 60 from T(2), leaving only the excess; so 60 becomes 0 and −1 becomes 59.

TM$ is put together in lines 6250-6270, as it was in lines 6080-6110. Because TM$ was set to ZR$ to begin with, the last two characters (seconds) are pre-set to '00' to make a full six-character string, as required for TI$.

Static−time display (lines 7000-7030): This routine is much simpler than the dynamic display routine in the mainline. The FOR...NEXT loop calls subroutine 1500 to print the digits from left to right. TI is the value of the current digit. If it is equal to the digit currently displayed (D(I)), then the display routine is skipped; otherwise the cursor is positioned and the digit is displayed. Then the corresponding 'currently displayed digit' element is updated by setting D(I) equal to TI.

Initialization (lines 8000-8530): Lines 8000-8030 dimension all of the strings and arrays that the program will use. Strings are dimensioned by reserving a specific length for each of them. Line 8040 opens the keyboard as input/output device number 1. The '4' after the OPEN statement declares the device as an input-only device. The 'K:'

specifies the keyboard. Line 8050 sets the constant K equal to the address of the keyboard status code; this is used in the program mainline. CR$ is defined as a carriage-return character, and the POKE 752,1 turns off the cursor so that it does not interfere with the display. Line 8060 sets the character color to black (color 0, luminance 0) and the background color to a light red (color 4, luminance 10). Finally, lines 8110-8170 define a number of string constants and initialize the alarm-set time TA$.

Set up digits (lines 8500-8530): This routine reads in the elements of the array S(I,J). The loop counter I ranges from 0 to 9, designating the current digit. Loop counter J designates the segments of digit I, which are numbered from 0 to 6. X is used as a dummy variable for the READ statement, which will not directly read a subscripted variable. The DATA for the array S(,) is contained in lines 9000-9040.

Expanding the ATARI Clock Program

More Attractive Digit Display

The appearance of the digits on the screen can be improved greatly by filling in the six segment intersections with solid or diagonal half characters. To accomplish this, we must define 11 different horizontal segment strings and change the coding to reflect these new possibilities. The analogy to a seven-segment display clock is no longer valid. Type in the following lines to improve the appearance of the digit display.

```
8010 DIM H$(55),V$(14),BK$(6),UP$(5)
8170 H$(1,20)= "□□□□■◢■■■◣◢■■■◤■■■■■ "
8180 H$(21,40)= "■■■■◣■■■■◤■□□□■◥■■■◣ "
8190 H$(41,55)= "◥■■■◤◥■■■■□■■■■ "
9000 DATA 1,1,1,6,1,1,8,0,0,1,0,0,1,0
9010 DATA 1,0,1,2,1,0,9,1,0,1,10,0,1,8
9020 DATA 6,1,1,3,0,1,0,3,1,0,7,0,1,5
9030 DATA 1,1,0,4,1,1,8,3,0,1,0,0,1,0
9040 DATA 1,1,1,3,1,1,8,1,1,1,9,0,1,0
```

Line 8010 reserves 55 characters for H$ to allow for 11 segment substrings that are each five characters long. Lines 8170-8190 define the 11 different horizontal segment strings. Note the use of the two diagonal fill characters and their reversed images. Lines 9000-9040 contain the new data for the segment digits. The digits are now coded with the numbers 0 through 10 instead of 0's and 1's.

The display routine (lines 1500 on) works exactly the same way. The PRINT statements still calculate the proper starting and ending locations of the substring (within either H$ or V$) to be printed, using the number codes in the array S(I,J) The only difference is that there are nine more horizontal segments that can be printed.

Adding a Chime

This function requires the following line changes:

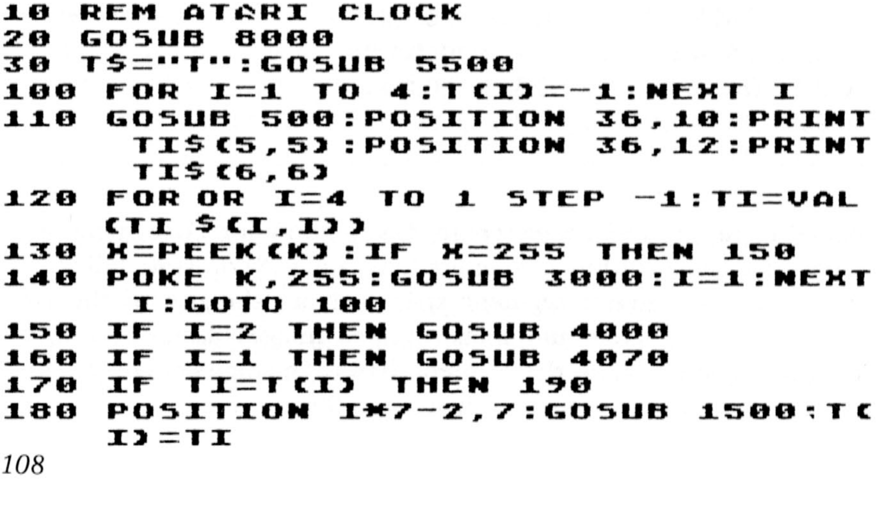
```
180 POSITION I*7 – 2,7:GOSUB 1500:IF I = 2 THEN CH = TI
185 T(I) = TI
205 IF CH THEN GOSUB 2000:CH = CH – 1
```

CH is used to count the proper number of chimes. In line 180 it is set to the current number of hours as corrected in subroutine 4000 for AM/PM. This line is reached only when the digit in a particular position changes. If I is 2, then the hour has just changed. Line 205 tests CH; if it is not 0, then a chime is sounded by a call to the alarm routine. CH is decremented and the main program continues.

You may want to make improvements in the chime. Try programming a mechanism to avoid chiming when the time has just been set. Also, you might try using different frequencies for the chime and alarm.

Other changes to consider in the program are running it on 24-hour time, playing a melody on the hour, or introducing a ticking sound. The 24-hour modification is a matter of deletion rather than of addition. The others require more effort and possibly more memory than provided in an unexpanded Atari.

Listing 1: **Atari Clock**

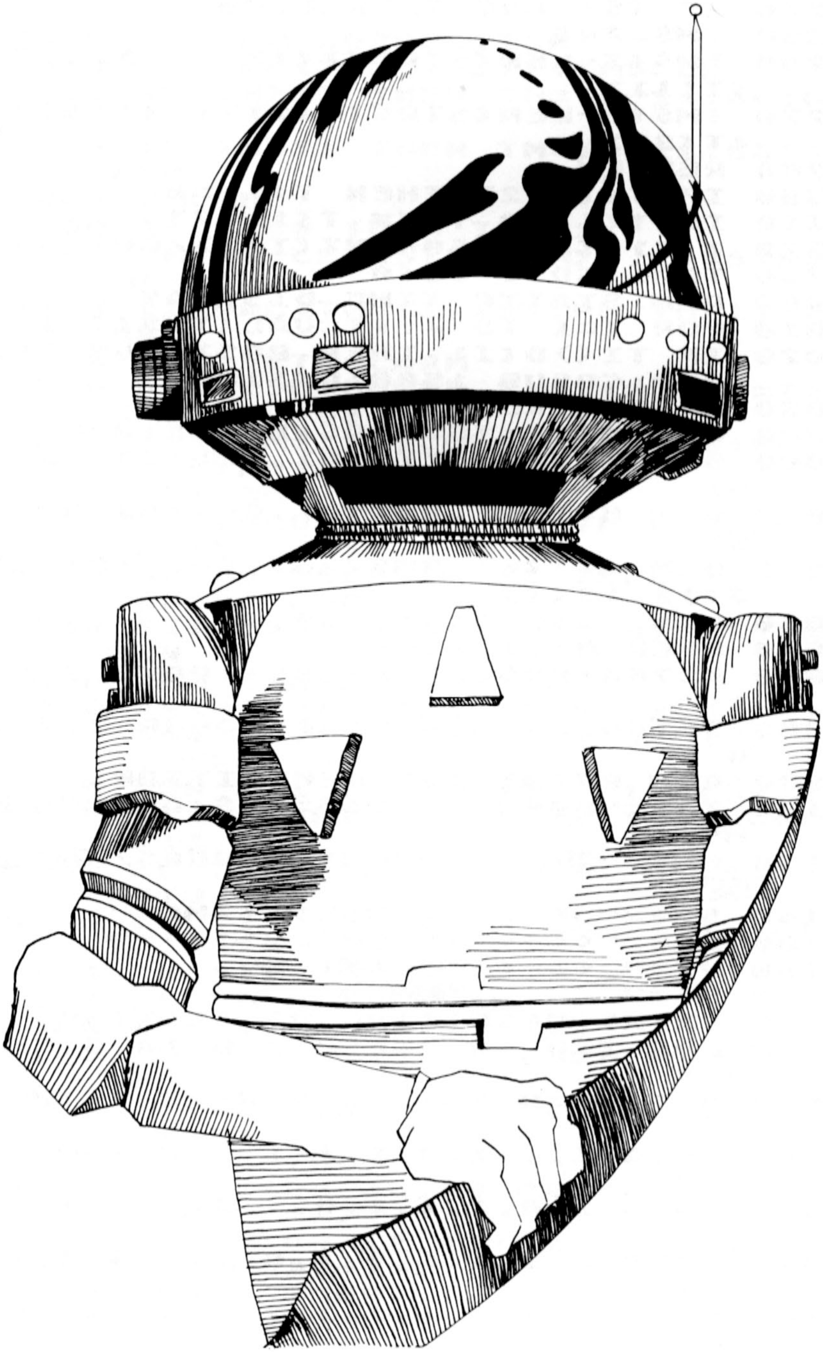


```
10 REM ATARI CLOCK
20 GOSUB 8000
30 T$="T":GOSUB 5500
100 FOR I=1 TO 4:T(I)=-1:NEXT I
110 GOSUB 500:POSITION 36,10:PRINT
      TI$(5,5):POSITION 36,12:PRINT
      TI$(6,6)
120 FOR OR I=4 TO 1 STEP -1:TI=VAL
     (TI $(I,I))
130 X=PEEK(K):IF X=255 THEN 150
140 POKE K,255:GOSUB 3000:I=1:NEXT
      I:GOTO 100
150 IF I=2 THEN GOSUB 4000
160 IF I=1 THEN GOSUB 4070
170 IF TI=T(I) THEN 190
180 POSITION I*7-2,7:GOSUB 1500:T(
     I)=TI
```

```
190 NEXT I
200 POSITION 35,15:PRINT AP$(PM*2+
    1,PM*2+2)
210 IF AF THEN GOSUB 1000:POSITION
     5,4:PRINT "ALARM: ";TA$(1,4)
220 POKE 77,0:GOTO 110
500 X=PEEK(18)*65536+PEEK(19)*256+
    PEEK(20)
510 H=INT(X/216000):M=INT(X/3600-H
    *60):S=INT(X/60-H*3600-M*60)
520 TI$=ZR$
530 TI$(3-LEN(STR$(H)),2)=STR$(H)
540 TI$(5-LEN(STR$(M)),4)=STR$(M)
550 TI$(7-LEN(STR$(S)),6)=STR$(S)
560 RETURN
600 X=VAL(TM$(1,2))*216000+VAL(TM$
    (3,4))*3600+VAL(TM$(5,6))*60
610 H=INT(X/65536):M=INT((X-H*6553
    6)/256):S=X-H*65536-M*256
620 POKE 18,H:POKE 19,M:POKE 20,S:
    RETURN
1000 REM CHECK ALARM
1010 IF TI$(1,4)=TA$(1,4) THEN GOS
    UB 2000
1020 RETURN
1500 REM PRINT DIGIT
1510 PRINT H$(S(TI,0)*5+1,S(TI,0)*
    5+5);BK$;
1520 PRINT V$(S(TI,1)*7+1,S(TI,1)*
    7+7);UP$;
1530 PRINT V$(S(TI,2)*7+1,S(TI,2)*
    7+7);BK$;
1540 PRINT H$(S(TI,3)*5+1,S(TI,3)*
    5+5);BK$;
1550 PRINT V$(S(TI,4)*7+1,S(TI,4)*
    7+7);UP$;
1560 PRINT V$(S(TI,5)*7+1,S(TI,5)*
    7+7);BK$;
1570 PRINT H$(S(TI,6)*5+1,S(TI,6)*
    5+5):RETURN
2000 REM ALARM
2010 SETCOLOR 2,8,10:SOUND 0,50,10
    ,10
2020 FOR I=1 TO 175:NEXT I
2030 SOUND 0,0,0,0:SETCOLOR 2,4,10
    :RETURN
3000 REM MODE SELECT
3010 IF X=63 THEN AF= NOT (AF):RET
    URN
3020 PRINT "↖SELECT MODE:"
3030 PRINT :PRINT "  SET ALARM"
3040 PRINT :PRINT "  SET TIME"
3050 PRINT :PRINT "  RETURN"
3060 GOSUB 7500
3070 IF T$=CR$ THEN PRINT "↖":RETU
    RN
```

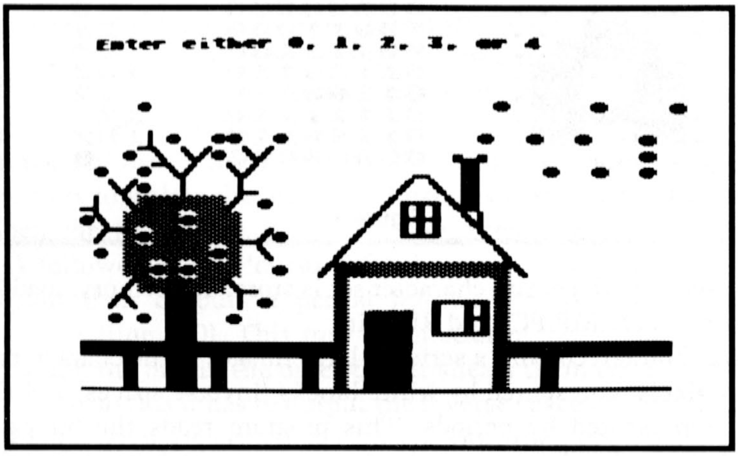
```
3080 IF T$="A" THEN 5000
3090 IF T$="T" THEN 5500
3100 GOTO 3060
4000 REM PM CORRECTION
4010 TH=VAL(TI$(1,2)):TM$=ZR$
4020 IF TH>23 THEN TH=TH-24:GOTO 4
     020
4030 IF TH>11 THEN TM=TH-12:GOTO 4
     050
4040 TM=TH
4050 IF TM=0 THEN TM=12
4060 TM$(3-LEN(STR$(TM)),2)=STR$(T
     M)
4070 TI=VAL(TM$(I,I)):PM=TH>11:RET
     URN
5000 REM SET ALARM
5010 AF=1:SETCOLOR 2,8,10:TM$=TA$:
     GOSUB 6000
5020 TA$=TM$:SETCOLOR 2,4,10:PRINT
     "↖":RETURN
5500 REM SET TIME
5510 SETCOLOR 2,12,10:TF=0:GOSUB 5
     00:TM$=TI$
5520 GOSUB 6000:IF TF THEN TI$=TM$
     :GOSUB 600
5530 SETCOLOR 2,4,10:PRINT "↖":RET
     URN
6000 REM ALARM/TIME SET
6010 PRINT "↖":POSITION 15,3:PRINT
      SA$((T$="T")*9+1,(T$="T")*9+9
     ):MF=0
6020 FOR I=1 TO 4:D(I)=-1:NEXT I
6030 T(1)=VAL(TM$(1,2)):T(2)=VAL(T
     M$(3,4))
6040 IF T(1)>23 THEN T(1)=T(1)-24:
     GOTO 6040
6050 IF T(1)>11 THEN TM=T(1)-12:GO
     TO 6070
6060 TM=T(1)
6070 IF TM=0 THEN TM=12
6080 PM=T(1)>11:TM$=ZR$
6100 TM$(3-LEN(STR$(TM)),2)=STR$(T
     M)
6110 TM$(5-LEN(STR$(T(2))),4)=STR$
     (T(2))
6120 GOSUB 7000
6130 POSITION 35,15:PRINT AP$(PM*2
     +1,PM*2+2)
6140 POSITION 12,19:PRINT HM$(MF*1
     3+1,MF*13+13)
6200 GOSUB 7500
6210 IF T$=" " THEN MF= NOT (MF):G
     OTO 6130
6220 IF T$="+" THEN T(MF+1)=T(MF+1
     )+1:GOTO 6300
```

```
6230 IF T$="-" THEN T(MF+1)=T(MF+1
    )-1:GOTO 6300
6240 IF T$<>CR$ THEN 6200
6250 TM$=ZR$
6260 TM$(3-LEN(STR$(T(1))),2)=STR$
    (T(1))
6270 TM$(5-LEN(STR$(T(2))),4)=STR$
    (T(2))
6280 RETURN
6300 IF T(1)>23 THEN T(1)=0
6310 IF T(1)<0 THEN T(1)=23
6320 T(2)=T(2)-60*INT(T(2)/60)
6330 TF=1:GOTO 6040
7000 REM STATIC TIME DISPLAY
7010 FOR I=1 TO 4:TI=VAL(TM$(I,I))
7020 IF TI<>D(I) THEN POSITION I*7
    -2,7:GOSUB 1500:D(I)=TI
7030 NEXT I:RETURN
7500 GET #1,X:T$=CHR$(X):RETURN
8000 DIM ZR$(6),TI$(6),TM$(6),TA$(
    6)
8010 DIM H$(10),V$(14),BK$(6),UP$(
    5)
8020 DIM AP$(4),HM$(26),SA$(18),CR
    $(1),T$(1)
8030 DIM T(4),D(4),S(9,6)
8040 OPEN #1,4,0,"K:"
8050 K=764:CR$=CHR$(155):POKE 752,
    1
8060 SETCOLOR 1,0,0:SETCOLOR 2,4,1
    0
8100 REM PRIMITIVE DEFINITIONS
8110 ZR$="000000":TA$=ZR$:AP$="AMP
    M"
8130 HM$="HOURS/MINUTESHOURS/MINUT
    ES"
8140 SA$="SET ALARMSET TIME "
8150 BK$="←←←←←↓":UP$="↑↑→→→"
8160 V$=" ←↓ ←↓ █←↓█←↓█"
8170 H$="      ███ "
8500 PRINT "KSETTING UP DIGITS"
8510 FOR I=0 TO 9:FOR J=0 TO 6:REA
    D X
8520 S(I,J)=X:NEXT J:NEXT I:RETURN

9000 DATA 1,1,1,0,1,1,1,0,0,1,0,0,
    1,0
9010 DATA 1,0,1,1,1,0,1,1,0,1,1,0,
    1,1
9020 DATA 0,1,1,1,0,1,0,1,1,0,1,0,
    1,1
9030 DATA 1,1,0,1,1,1,1,1,0,1,0,0,
    1,0
9040 DATA 1,1,1,1,1,1,1,1,1,1,1,0,
    1,1
```



Programmable Characters

Customize the Atari Character Set

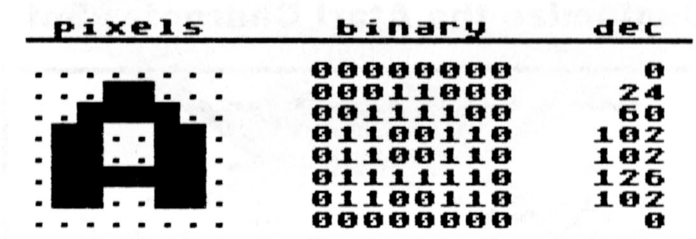


```
Enter either 0, 1, 2, 3, or 4
```

The Atari comes with a set of 128 characters. The reversed images of these characters add 128 more for a total of 256 characters. This set of characters is more than adequate for most situations. However, there are times when a few extra characters would be nice (for space invaders, for instance). Other times we may need to change the entire character set. The Atari provides this capability. This chapter shows you how to program custom characters on the Atari and gives you a few ideas on how to use them.

How Characters are Stored

Each Atari character is made up of an eight-by-eight array of little dots called pixels. Each pixel shows up on the screen as either the background or character color. Pixel patterns are stored in an area of Read-Only Memory (ROM) starting at location 57344. Each character description is stored as eight bytes, each representing a row of pixels on the screen. Within each byte, if a bit is set (1), then the corresponding pixel will show up in the character color; if the bit is off (0), then the pixel will show up in the background color. Figure 1 shows the Atari letter 'A' as it appears on the screen in pixels and again in memory as 1's and 0's.



```
pixels        binary       dec
              00000000       0
              00011000      24
              00111100      60
              01100110     102
              01100110     102
              01111110     126
              01100110     102
              00000000       0
```

Figure 1

To see how the entire character set is stored in memory, load in the program 'CSDUMP.PC' and RUN it.

You should recognize a series of large images of the characters with the on pixels represented by white blocks (inverse spaces) and the off pixels represented by periods. This program reads the bit patterns directly from the character set in memory, byte by byte. A description of how it works follows.

The location in memory that tells the Atari where to find its character set is 756. Such locations are called pointers because they literally point to where in memory information can be found. Thus, such pointers actually hold locations themselves. But since the pointer is only a single byte in size and, therefore, can hold a number only between 0 and 255 inclusive, this pointer can't hold the actual location of the character set. This is because locations themselves are two bytes in length and can be any number from 0 to 65535. Instead, this pointer holds the page number (a page is a group of 256 bytes) of the character set. To find the actual location, we must multiply the value found in 756 by 256. Line 100 does this and stores that location in ROMCS. In the Atari, the internal character set (ROM) lies at location 57344, as indicated by the REM statement. Therefore, the value in 756 should be 224 (what is 224 * 256?).

Line 200 initiates a loop that will point to the starting byte of each of the 128 characters. Because there are 128 characters and each is eight bytes' worth of description in length, this program will output exactly 1024 bytes of character description (0-1023). The STEP 8 modifier to the FOR...NEXT loop simply forces PTR to be the starting byte of the characters. Note that the inverse characters need not be stored. The Atari, when it needs one, merely reverses all the bits in the description so that it comes out with the on pixels off and the off pixels on.

Line 210 initiates a loop that will point to each of the eight bytes in the description. ROMCS (the starting location of the character set) is added to BYTE to get the actual locations in memory that hold the descriptions. VALUE is the value at the locations of each of the eight bytes in the character. The location is displayed followed by a number of spaces for the picture. STORE is then given the value of VALUE because VALUE will eventually be decremented to zero.

Line 215 sets a loop to pluck out each of the bits in the current descriptor byte (VALUE).

Line 220 uses a technique known by most machine-language programmers to isolate the least significant (bottommost) bit in the byte. It then checks that bit. If the bit is set, then it outputs a solid white block followed by two 'cursor-left' characters. If the bit is not set, the program then outputs a period followed by the two 'cursor-left' characters (line 230). This may seem strange, but what it is doing is displaying the bits in the order that it shears them off; i.e., from right to left. To do this it has to output the reverse space (or period) and then back up over it twice — once to position the cursor back over the displayed bit and once more to back it up to the position just before the displayed bit. Thus, the next bit to be displayed falls right before the last bit displayed. In short, the display process to the screen is temporarily reversed.

The technique used to test the bottommost bit involves subtracting the integer of VALUE divided by 2 and then multiplied by 2 from VALUE. Dividing by 2 effectively shifts all the bits in the number to the right 1 bit, thereby throwing what was the bottommost bit out past the decimal point. Taking the integer shears off this fraction. When VALUE is then multiplied by 2, all the bits are shifted left one space, bringing a 0 in at the bottom. So far all we have done is zeroed the bottom bit. To get what that bit was we subtract from the original number the value just derived. This procedure will not seem so long and complex once you have done it a few times.

Line 240 shifts VALUE to the right 1 bit so that we can test for the next bit.

Line 250 moves the cursor to the right 11 spaces and then outputs the original value. If the program is on the fifth byte displayed, it also outputs the internal code for the character being represented.

Line 252 forces a carriage return to the next line.

Line 270 goes to the next byte in the character description. At the end of the character it prints a blank line.

Line 280 goes to the next character.

Note that the characters are stored in the order of their internal codes, *not* their ASCII equivalents.

Defining Your Own Characters

Since, as previously mentioned, the pixel patterns are stored in ROM, an area of memory that can be only read from and not written to, we cannot change the character set there. Instead, we change where the Atari looks for its characters. This is done at location 756, which holds the page number where the character set resides. That is, the Atari looks at this location to find how to draw its characters. All we have to do is change 756 so that it points to an area of RAM (Random Access Memory) and put our new character set there.

However, it is not all that easy. The Atari can display its characters in two ways: in graphics mode 0 (text mode) and in either mode 1 or 2. When it is in one of the latter two modes, the Atari uses only the first 64 of its characters. Briefly, this is because when in these modes the Atari displays every group of 64 characters the same way it does its first 64 characters — only in a different color. That is, the character whose code is 33 (internal code for upper-case 'A' — group 1) looks the same but is a different color than the character whose code is 97 (internal code for lower-case 'a' — group 2). This means that for these two modes, the only descriptions needed are those for the first 64 characters; only the first 512 bytes of storage are used.

Graphics mode 0 is another story. This mode displays *all* 256 characters and therefore has to access the entire 128 in memory (the reverse images need not be stored). It uses the full 1024 bytes of description found through location 756.

Here is the kicker. There is a rule for the Atari that states that when using modes 1 and 2, the character set must reside on a ½K (512-byte) boundary; i.e., it must start on a location that is a multiple of 512. This means that location 756 must hold a number that is a multiple of 2. A similar rule applies to graphics mode 0. It states that its character set must reside on a full 1K (1024-byte) boundary. So when using the characters in this mode you must make sure that the value in 756 is a multiple of 4. The beginner may find this bothersome.

Load and run the next program, 'FILLIN.PC'.

The first thing FILLIN does is to move the character set pointer (location 756) to a blank part of memory. This causes all the characters to appear as spaces. Next it outputs a screenful of text (which will be totally blank), and then it loads the character set from ROM into the memory just cleared. This causes the character set to more or less 'ooze' onto the screen.

To get a batch of free memory, FILLIN uses the memory used by the string CS$. CS$ is dimensioned to a length of 2048, or some 1024 more than the amount needed. This may seem wasteful, but it is neccessary because the character set must be on a 1K boundary, and the actual location of the string could be anything. Dimensioning it for 2048 insures that there will be such a boundary somewhere within it. Line 60 then zeros that string, which really zeros the memory that holds it. We now have a batch of memory (somewhere) that is zeroed.

Lines 110 to 130 find which character position in the string corresponds to the 1K boundary. START holds the address of the string (found through the ADR function). PAGE holds the page number of the start of the boundary. This is found by locating the 1K boundary just before the string, adding 1 to it (effectively finding the boundary just INSIDE the string), and then multiplying by 4 to convert it from a batch of 1024 to a batch of 256. This gives us the page number of the boundary. CHRPOS (character position) is then assigned to the difference between the memory start of the string and the location of the 1K boundary within it. This yields the character position within the string where the character set (when it is loaded) is to be stored.

Line 140 gets the location of the ROM character set. Line 150 causes the new character set to come from the 1K boundary in the string CS$.

Lines 200 to 262 display the full screen of text.

Lines 310 to 330 fill in the character set, byte by byte and character by character, at the boundary position in the string. As each byte of description is loaded into the string, it is displayed on the screen.

Animation with Programmable Characters

Load in and run 'ANIMATE.PC'. Five space invaders will appear, marching noisily across the screen diagonally. Watch them move their arms, legs, and mouths.

As a rule, animation with programmable characters is almost always very fast. This is because the entire description of every occurrence of any particular character on the screen can be changed with only a few statements. As in the case of the moving space invaders, the movement of their arms, legs, and mouths is carried out with only one statement. Here is a description of how it works.

Lines 10 to 140 put the Atari into graphics mode 2 and set up CS$ in the same manner as FILLIN. The only difference is that, as previously mentioned, modes 1 and 2 require a definition for only the first 64 characters (512 bytes). Also it need be on only a ½K boundary, so the string can be even smaller.

Lines 210 to 230 get the ROM character set and put it into CS$.

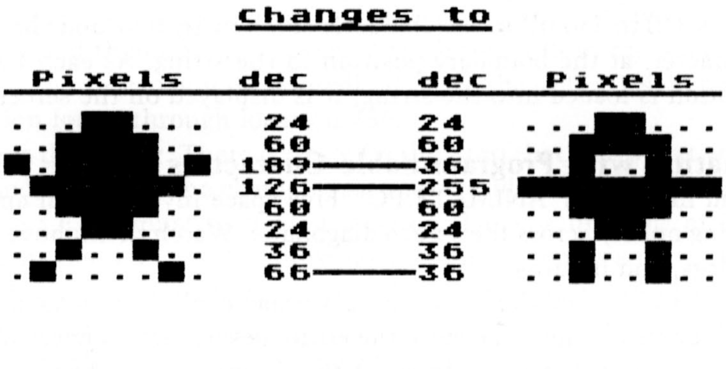
Lines 310 to 330 redefine the '$' to look like a space invader with his arms up.

Line 400 redirects the character set pointer to point to the string.

Line 510 sets up the invaders to appear on lines 1 through 10. X is incremented so that the invaders will move diagonally. Line 520 clears the screen and at position X,Y displays the invaders in each of the four colors. To get a different color for each, I added 64, then 128, and then 192 to the internal code for '$'. In mode 2 this displays the same character but in different colors.

Line 530 sets up a 100-count delay so that they won't move too fast.

Lines 540 and 550 make the sounds and then move the invaders' arms, legs, and mouths. This is done by changing the part of the string that defines from their mouths on down to another string. If the invaders are displayed on an even-numbered line they are forced to look different; if they are not, the invaders remain in their original condition. Figure 2 shows the original space-invader character and which bytes are altered to obtain the preferred movements.



changes to

Pixels	dec	dec	Pixels
	24	24	
	60	60	
	189	36	
	126	255	
	60	60	
	24	24	
	36	36	
	66	36	

Figure 2

Multi-Colored Characters

The Atari has a mode that allows any position on the screen to display a character comprised of more than one color. Up to four colors per character and five colors total per screen (including the background) are possible. The pixels that make up these characters come in two sizes: the size of the pixels involved in graphics mode 2 text (which is internal mode 5) and the size of the pixels involved in graphics mode 1 text (which is internal mode 4). Using mode 5 will cause a picture to appear as though it were drawn in graphics mode 7 (the pixels are the same size) while using less memory. And using mode 4 will result in a full-color picture twice the resolution of graphics mode 7!

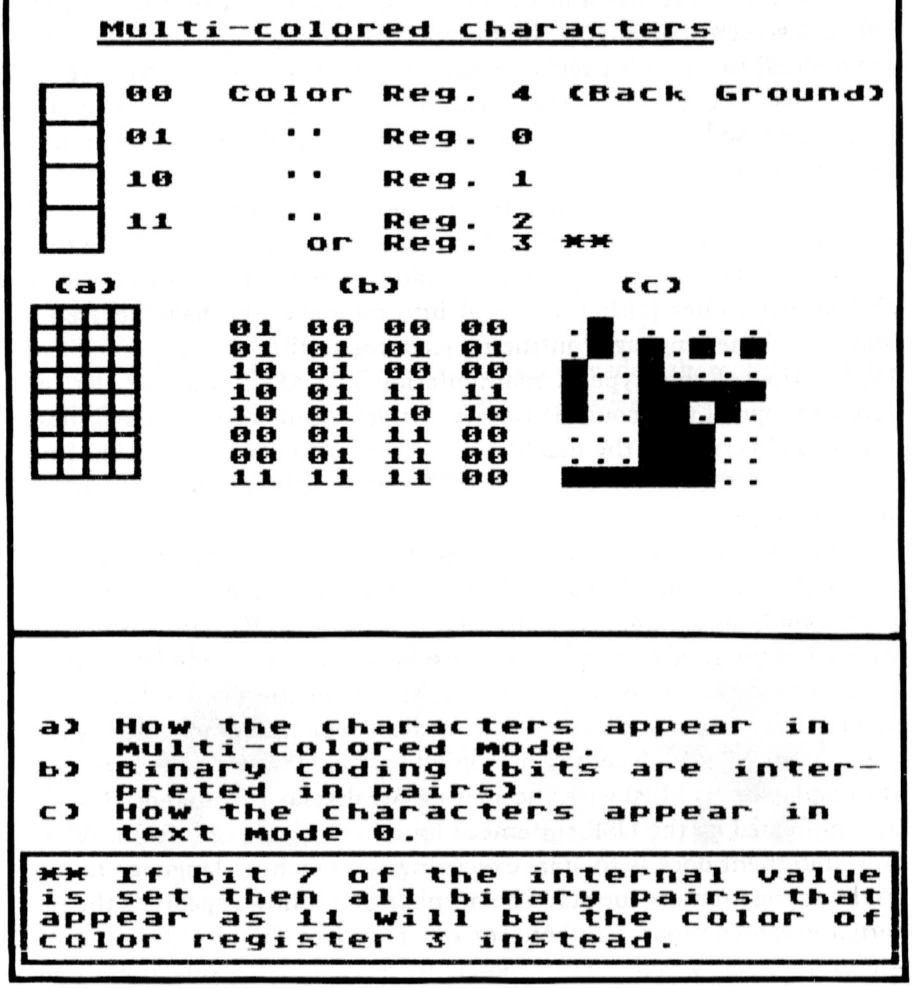
Load and run 'MCOLOR.PC'. After about a 20-second delay, a front yard should appear, tree and all. This is an example of using an internal mode 4 screen. Hit any of the numbers from 0 to 4. These numbers correspond to the color registers found at location 708. Hitting them will cause the value in the corresponding color register to be incremented, so holding one down will cause all the colors to appear, one after another.

These graphics modes cannot be accessed through an Atari BASIC command. Instead we must modify something called the display list. We modify the actual display mode, scan line by scan line, by POKEing the correct values (either a 4 or 5) into memory. We have devised a short machine-language routine to accomplish this. A comprehensive explanation of this type of manipulation in BASIC is above the intended scope of this book. However, for you more adventurous types, the BASIC version of the machine-language subroutine is listed at line 32700, and the procedure used is briefly outlined in the following paragraph.

The display list is a series of bytes in memory, the starting location of which can be found in locations 560 and 561. Each byte in the list corresponds to a graphics 'command' for the ANTIC chip inside the Atari. The possible commands are: display (a line of any mode), display blank lines, and jump (go to a different part of the display list). The display list is structured as a sort of endless loop in memory that is executed by ANTIC. To get the new graphics mode we must be sure that the display list is filled with the appropriate display commands. This is accomplished by the USR statement found at the end of line 20. What this statement does is to start executing the machine language found in the string memory for M$. The number that follows specifies the internal graphics mode number. For our purposes this number will be either a 4 or 5, where 4 corresponds to the smaller pixels and 5 corresponds to mode 7 pixels. If you wish to execute the BASIC version for some reason (it is much slower), simply replace the entire USR statement with a GOSUB 32700.

Each multicolored character uses eight bits of data, the same as with the other characters. But because it has to account for color storage, the eight bits are enough for only four pixels across. That is, the bits are grouped into pairs, and each pair is then interpreted as a color (see figure 3). The technical reason for this is that two binary digits can hold a number from 0 to 3; hence we have four colors. There is a way, believe it or not, to get a full five colors on the screen; this will be discussed later.

Figure 3



```
     Multi-colored characters

     00   Color Reg. 4 (Back Ground)
     01     ''  Reg. 0
     10     ''  Reg. 1
     11     ''  Reg. 2
           or   Reg. 3 **

 (a)          (b)              (c)

        01 00 00 00
        01 01 01 01
        10 01 00 00
        10 01 11 11
        10 01 10 10
        00 01 11 00
        00 01 11 00
        11 11 11 00

a) How the characters appear in
   multi-colored mode.
b) Binary coding (bits are inter-
   preted in pairs).
c) How the characters appear in
   text mode 0.

** If bit 7 of the internal value
is set then all binary pairs that
appear as 11 will be the color of
color register 3 instead.
```

Stop the program and type 'LIST 32700,32740'. Now change line 30030 so that the variable MODE holds a value of 5. RUN the program again. The top half of the earlier scene should appear — but twice as tall! This is an example of an internal mode 5 screen. Note that because of its new size, only the top half of the picture is visible; the bottom half is down near your knees!

This is how 'MCOLOR.PC' works.

Lines 15 and 20 set up the screen in the specified mode and perform most program initialization.

Lines 50 to 230 set up the new character set in CS$ as in the other demos. This character set is a full 128 characters (1024 bytes) long.

Lines 300 to 396 redefine the characters '#', '$', '%', and '&' to tiny checks, larger two-tone checks, and two solid blocks (different colors), respectively. Note that these are the decimal equivalents to the binary patterns used, so what these characters look like internally may not be obvious.

Lines 1000 to 1270 display the front-yard scene.

Lines 2000 and 2010 get a number from 0 to 4 from the keyboard, add 708 to it to get the location of its corresponding color register, and POKE it with what it previously held plus 2.

Line 30000 is the initialization routine, line 32000 holds the bytes in string form for the machine-language display-list alteration, and lines 32700 to 32740 hold the BASIC equivalent of that alteration routine.

Note that the picture is printed as though the Atari were in graphics mode 0. This is because it thought it was! The Atari was put into graphics mode 0 at line 20. It was our routine that took it out, but the Atari still operates as though it were in mode 0. To prove this, hit break and type LIST. What follows should be something in a program-like format. You won't be able to recognize anything because the string that holds the character set literally moves when you stop program execution. Now type POKE 756,224 (it will not look like what you typed). This will redirect the character-set pointer back to ROM. What should appear is the bottom part of the program listing, which looks something like graphics mode 0 text, if you are in mode 4. If you are in mode 5, the listing will be made of characters twice as tall. Also, you won't see what you are typing unless you clear the screen, because only the top half is displayed in this mode.

As mentioned before, in these two modes it is possible to display five colors. Recall that when in text mode 0, the computer simply displays all characters greater than 128 as the inverse of its lower description. But how does the computer display inverse color characters? What it does is simply to display all bit pairs within that character description that are coded binary 11 as the color found in color register 3 rather than register 2. This means that two characters with the same data can be displayed in different colors. For example, take a look at the apples in the tree and the smoke or the smokestack and the tree trunk. Note that in the display the characters responsible for these parts of the drawings are merely inverses of each other, but the computer displays them as different colors. This applies only to the characters made up of binary 11's; it won't work for them all.

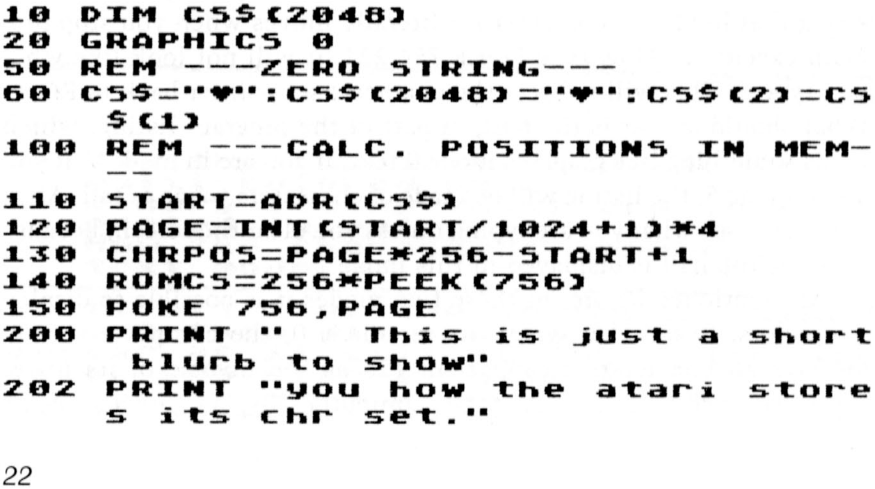
Combining Some Techniques

The most time-consuming part of the entire program is the part that loads the existing character set into CS$. Things can be speeded up in two major ways. We can make the Atari load in only the characters that it needs; i.e., the graphics characters and a few letters (this can become cumbersome and will still take several seconds). Or we can use the machine-language 'block-move' routine that was used to scroll the graphics mode 1 screen in the MASTER and Word Detective chapters. All that is needed is to perform a move about 1024 bytes from the character-set default location (57344) to the 1K boundary location within the string (PAGE * 256). The statement will replace all of the lines from 200 to 230 and should appear as follows:

```
U = USR(ADR(M2$),ROMCS,256*PAGE,1024)
```

Refer to the section titled "Scrolling the Graphics Screen" in the MASTER chapter for subroutine specifics.

Listing 1: **Fillin.PC**



```
10  DIM CS$(2048)
20  GRAPHICS 0
50  REM ---ZERO STRING---
60  CS$="♥":CS$(2048)="♥":CS$(2)=CS
    $(1)
100  REM ---CALC. POSITIONS IN MEM-
    --
110  START=ADR(CS$)
120  PAGE=INT(START/1024+1)*4
130  CHRPOS=PAGE*256-START+1
140  ROMCS=256*PEEK(756)
150  POKE 756,PAGE
200  PRINT "    This is just a short
      blurb to show"
202  PRINT "you how the atari store
    s its chr set."
```

```
204 PRINT "it also shows you how t
    he chr set is"
206 PRINT "loaded into strings. No
    te that the"
208 PRINT "loading of the characte
    rs is done in"
210 PRINT "the order of the intern
    al character"
212 PRINT "codes. (This is NOT ASC
    II.) This is"
214 PRINT "why the UPPER CASE lett
    ers are done"
216 PRINT "first, and the lower ca
    se letters are"
218 PRINT "done after."
220 PRINT "    NOTE THAT THE LOWER
    CASE LETTERS"
222 PRINT "MUST BE STORED SEPERATE
    LY AS, INDEED,"
224 PRINT "THEY ARE SEPERATE CHARA
    CTERS, LOOK"
226 PRINT "DIFFERENT, AND MUST THE
    REFORE HAVE"
228 PRINT "DIFFERENT DESCRIPTIONS
    IN MEMORY."
230 PRINT "   note also that REVER
    SE LETTERS"
232 PRINT "NEED NOT BE STORED AS T
    HAT WOULD BE"
234 PRINT "A TERIBLE WASTE OF MEMO
    RY WHEN THEY"
236 PRINT "ARE MERELY THE INVERSE
    OF CHARACTERS"
238 PRINT "ALREADY STORED."
240 PRINT "MORE CHARACTERS:"
250 FOR Q=0 TO 31:PRINT CHR$(27);C
    HR$(Q);:NEXT Q:PRINT
260 FOR Q=128 TO 159:IF Q=155 THEN
     PRINT "█";:GOTO 264
262 PRINT CHR$(27);CHR$(Q);
264 NEXT Q
300 REM ---GET ROM CHAR SET---
310 FOR PTR=0 TO 1023
320 CS$(CHRPOS+PTR,CHRPOS+PTR)=CHR
    $(PEEK(ROMCS+PTR))
330 NEXT PTR
9999 GOTO 9999
```

Listing 2: Animate.PC

```
10 DIM CS$(1024)
20 GRAPHICS 2+16:DEG
50 REM ---ZERO STRING---
60 CS$="♥":CS$(1024)="♥":CS$(2)=CS
   $(1)
100 REM ---CALC. POSITIONS IN MEM-
    --
```

```
110 START=ADR(CS$)
120 PAGE=INT(START/512+1)*2
130 CHRPOS=PAGE*256-START+1
140 ROMCS=256*PEEK(756)
200 REM ---GET ROM CHAR SET---
210 FOR PTR=0 TO 511
220 CS$(CHRPOS+PTR,CHRPOS+PTR)=CHR
    $(PEEK(ROMCS+PTR))
230 NEXT PTR
300 REM ---REDEFINE $---
310 FOR PTR=32 TO 39:READ D
320 CS$(CHRPOS+PTR,CHRPOS+PTR)=CHR
    $(D)
330 NEXT PTR
390 DATA 24,60,189,126,60,24,36,66
400 REM ---REDIRECT CS POINTER---
410 POKE 756,PAGE
500 REM ---DISPLAY ON SCREEN---
510 FOR Y=1 TO 10:X=X+1
520 PRINT #6;CHR$(125):POSITION X,
    Y:PRINT #6;"$ ⊟┤ ▐ $":SOUND 0
    ,0,0,0
530 FOR Q=1 TO 100:NEXT Q
540 IF Y/2=INT(Y/2) THEN SOUND 0,2
    40,10,10:CS$(CHRPOS+34,CHRPOS+
    39)="$◨<┴$$":GOTO 590
550 SOUND 0,255,10,10:CS$(CHRPOS+3
    4,CHRPOS+39)="▤◂<┴$B"
590 NEXT Y:X=0:GOTO 510
```

Listing 3: MColor.PC

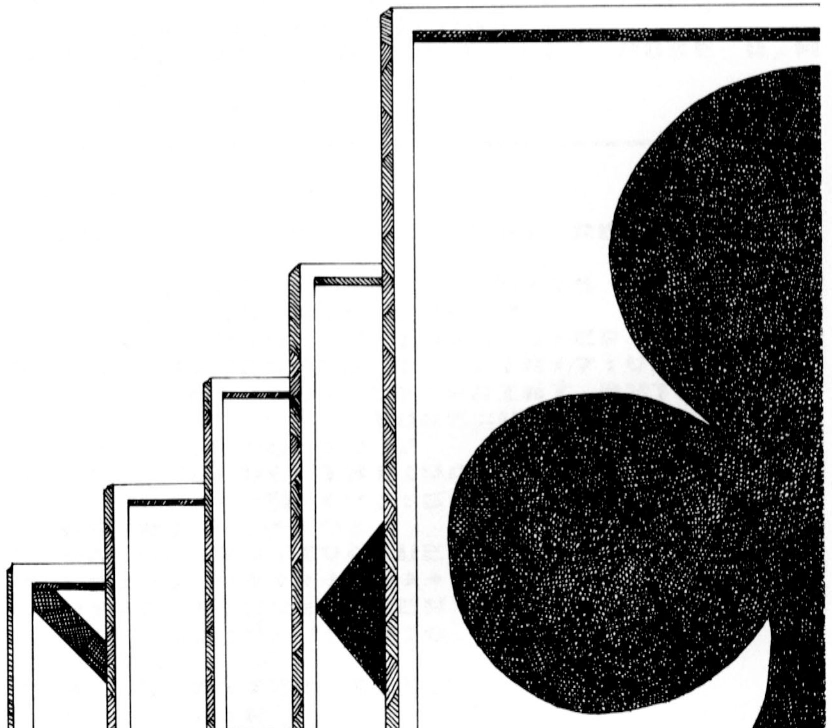
```
10 DIM M$(29),CS$(2048)
12 GOSUB 30000
20 GRAPHICS 0:U=USR(ADR(M$),MODE)
50 REM ---CLEAR STRING---
60 CS$="♥":CS$(2048)="♥":CS$(2)=CS
   $(1)
100 REM ---CALC. POSITIONS IN MEM-
    --
110 START=ADR(CS$)
120 PAGE=INT(START/1024+1)*4
130 CHRPOS=PAGE*256-START+1
140 ROMCS=256*PEEK(756)
150 POKE 756,PAGE
200 REM ---GET ROM CHAR SET---
210 FOR PTR=0 TO 1023
220 CS$:S$(CHRPOS+PTR,CHRPOS+PTR)
    =CHR $(PEEK(ROMCS+PTR))
230 NEXT PTR
300 REM ---REDEFINE #,$,%,&---
305 RESTORE 390
310 FOR PTR=24 TO 55
320 READ D
330 CS$(CHRPOS+PTR,CHRPOS+PTR)=CHR
    $(D)
```

```
340 NEXT PTR
390 DATA 68,17,68,17,68,17,68,17
392 DATA 165,165,165,165,90,90,90,
    90
394 DATA 255,255,255,255,255,255,2
    55,255
396 DATA 85,85,85,85,85,85,85,85
1000 POKE 82,1:PRINT "Enter either
     0, 1, 2, 3, or 4"
1010 PRINT :PRINT :PRINT
1100 ? "    □
    ●      □    ●"
1110 ? "    | □  □□  □
    ●  □  ●   □"
1120 ? "   □‾╲ ╲╱□ ╲╱           ▁▁
    ▪         □"
1130 ? " □  □‾╲╱  ╲╱          ▁   ▨
        ●  □  ●"
1140 ? "  ╲╱●  |   └         ╱ ╲ ▨
    "
1150 ? "┘  ╲######╱ □        ╱┌┬┐╲▨
    "
1160 ? "□╲ ####□###         ╱ ├─┤ ╲
    |"
1170 ? " ╱‾#●######_╱□     ╱   └┴┘
    ╲"
1180 ? " □ ######□# ╲    ╱
    ╲"
1190 ? "   ##□#●### □  ╱#########
    ###╲"
1200 ? "   ╱######╲_     #
     #"
1210 ? "  ╱╲  %%  | □    #     ┌─┬
    ┐#"
1220 ? "  □ □ %%  □      # $$$ | ├
    ┤#"
1230 ? "      %%         # $$$ └─┴
    ┘#"
1240 ? "&&&&&&%%&&&&&&&&&# $$$
     #&&&&&&&&&&&"
1250 ? "   &  %% &     & # $$$
     #  &"
1260 ? "   &  %% &     & # $$$
     #  &"
1270 ? "──────────────────────────
    ────────────"
2000 GET #1,A:IF A<48 OR A>52 THEN
     2000
2010 A=A-48:P=PEEK(708+A):P=P+2-25
    6*(P=254):POKE 708+A,P:GOTO 20
    00
30000 RESTORE 32000:READ M$
30020 OPEN #1,4,0,"K:"
30030 MODE=4
```

```
30090 RETURN
32000 DATA h█0 ┌K─1 ┌Lhh█/┌KH@↓0Ü▪
   @█┘┌K◆
32700 REM ---SET UP DL FOR IR4---
32710 DL=256*PEEK(561)+PEEK(560)
32720 POKE DL+3,64+MODE
32730 FOR Q=DL+6 TO DL+28:POKE Q,M
   ODE:NEXT Q
32740 RETURN
```

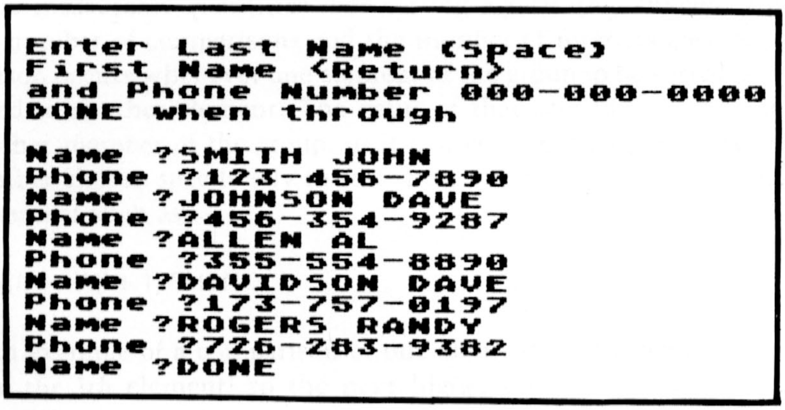
Listing 4: CSDump.PC

```
100 ROMCS=256*PEEK(756):REM IS 573
    44
190 ? :PRINT "MEM,   BITS SET, DEC
    IMAL, INTRNL CODE"
200 FOR PTR=0 TO 1023 STEP 8
210 FOR BYTE=PTR TO PTR+7:VALUE=PE
    EK(ROMCS+BYTE):PRINT BYTE+ROMC
    S,;"     ";:STORE=VALUE
215 FOR BIT=0 TO 7
220 IF (VALUE-INT(VALUE/2)*2) THEN
     PRINT "█←←";:GOTO 240
230 PRINT ".←←";
240 VALUE=INT(VALUE/2):NEXT BIT
250 PRINT "→→→→→→→→→→→→→";STORE,:IF
    BYTE-PTR=4 THEN PRINT "[";INT(
    PTR/8);"]":GOTO 270
252 PRINT
270 NEXT BYTE:PRINT
280 NEXT PTR
```



Sorting

Information on Five Sorting Methods



```
Enter Last Name (Space)
First Name <Return>
and Phone Number 000-000-0000
DONE when through

Name ?SMITH JOHN
Phone ?123-456-7890
Name ?JOHNSON DAVE
Phone ?456-354-9287
Name ?ALLEN AL
Phone ?355-554-8890
Name ?DAVIDSON DAVE
Phone ?173-757-0197
Name ?ROGERS RANDY
Phone ?726-283-9382
Name ?DONE
```

"A picture is worth a thousand words." Often quoted, but how often implemented? Here is a program that demonstrates, on the screen, behind-the-lines activity of a sorting algorithm — a much taught and well-documented topic.

Have you ever wondered how a computer keeps all of the information stored on it in proper order? Have you ever wanted to sort a long list into alphabetic or numerical order? Here are the answers to everything you need to know about computerized sorting and then some.

Operating Instructions

1. LOAD the program BARSORT and RUN it.

2. When the menu appears, press '1' for bubble sort.

3. Choose 'R' for random numbers and sit back and watch as your Atari puts the bars into descending order.

4. To stop the action, press the spacebar; to continue, press it again.

5. To watch the sample being sorted one step at a time, press 'X'. The program will do one step and wait for another 'X' or the spacebar to continue.

6. To become acquainted with the program, select the number for each of the other sorts, then read on.

Background Information

To begin, we present some background of sorting theory including information on five sorting methods: bubble sort, insertion sort, selection sort, Shell sort, and quick sort. A complete program is included for each method.

Apart from specific details for each of the algorithms, the theory connected with sorting has to do with efficiency. The sorts listed above are more or less in ascending order of efficiency, bubble sort being the slowest. Bubble sort, however, is the easiest to understand, so we will spend the most time on it.

When discussing sorting, it is necessary to define a few mathematical terms; please bear with us and we will make this as painless as possible. The number of items to be sorted (the total number of objects in a group with which the program deals) is called 'N'. Number of elements to sort = N.

To talk about the efficiency of any algorithm, you need to have some measurable quantities. For sorting algorithms, concentrate on the number of comparisons and the number of interchanges. A comparison occurs whenever one element of the group to be sorted is compared to another element. The value of that second element can be another member of the group, or it can be a very large or very small number chosen specifically to sort the group above or below. Thus, a statement such as

```
IF A(I) < A(I + 1) THEN ...
```

is an example of a comparison of one element in the array A (in this case the I*th* element) to the next higher one (I + 1). Also, a statement like

```
IF A(I) = MAX THEN ...
```

is a comparison of the element A(I) to a large number, MAX.

An interchange occurs whenever a member of the group is moved from one place in the computer's memory to another and, possibly, some other element takes its place. The classic interchange can be described by this sequence of statements:

```
TEMP = A(I)
A(I) = A(J)
A(J) = TEMP
```

assuming that I is not equal to J. Not all sorting algorithms use this form, but usually there is an easily identifiable interchange step whose repetition you can count.

Counting the number of comparisons and interchanges that take place during program execution helps you measure the efficiency of that algorithm. In addition, there is computer time spent in loop control and subroutine calls that is difficult to estimate except by empirical observation.

Now that you are armed with a few terminological weapons, we shall present some of the sorting buzz phrases. Do not be alarmed if they sound Greek. They are background information only and are not necessary to the operation of the program itself.

If you count the number of comparisons made during the sorting of a group by a particular algorithm, it may turn out that N^2 (N squared) comparisons are needed. This means that ''on the order of'' N times N comparisons are needed to sort an array of size N using that particular algorithm.

Another type of algorithm may use a different amount of comparisons called $N * (\log_2 N)$ — N log N. This is N multiplied by the log base 2 of N, a smaller number than the previous example, take our word for it. Why should you be interested in these numbers, and what is the significance of the difference between them?

Look at table 1. It shows values for N, N^2, $(\log_2 N)$ and $N * (\log_2 N)$. Assuming that the overhead from one algorithm to the next is relatively constant, you can see that there is an ever increasing difference between N^2 and N log N. To make the comparison more concrete, assume that a comparison costs .001¢, and that you need to sort an array containing 1,048,576 numbers. Using an N^2 sort will cost $10,995,116.27, while using an N log N sort will cost $209.72. Of course, a single comparison on today's big computers costs considerably less than .001¢. But even at .0000001¢ per comparison (a rate of 1¢ per 10,000,000 comparisons), the cost will be 2¢ for an N log N sort and $1,099.51 for an N^2 sort! With that kind of differential, you can see why a commercial sorting program would not use an N^2 approach.

Bubble Sort

This algorithm is probably the most widely known and easiest to understand of all sorting algorithms. Assume that N elements, which are denoted by A(1), A(2), A(3), ... A(N), are to be arranged in ascending order. This sort operates by sweeps through the array, causing various elements to ''bubble up'' in the process . For each pass, at least one element is guaranteed to be placed in its final sorted position. At the heart of each sweep is the idea of comparing two adjacent entries in the array:

A(I) > A(I + 1)

If A(I) is greater than A(I + 1), then the two elements are known to be out of correct order and must be swapped. This is accomplished by the classic interchange mentioned previously and illustrated here in BASIC (see figure 1).

```
100 IF A(I) < = A(I + 1) THEN 140
110 TEMP = A(I)
120 A(I) = A(I + 1)
130 A(I + 1) = TEMP
140 ...
```

Figure 1

Now consider the number of comparisons necessary to arrange the array into sorted order. First, you can make no assumptions about the original array with respect to the relative positions of the members. You must successively compare each element to the next higher one until the end is reached. That is, you compare A(1) to A(2) and A(2) to A(3), until you compare A(N – 1) to A(N). Positions of various elements will be swapped (using the code in figure 1); in particular, the largest element in the original array will end up in the A(N) position after the sweep is completed. To be convinced that this is true, ask yourself, "If the largest value is originally in A(J), then with what other array elements will it be swapped?"

After one sweep of the array, you have a picture such as that shown in figure 2. You have partially sorted the original array. In fact, the only sorted element is the last. On each subsequent sweep through the array, one additional element is sorted. The last sweep is when A(1) and A(2) are interchanged, if necessary, to complete the sort. Since two elements are arranged by this swap, and one by each previous sweep, it requires N – 1 sweeps through the array to ensure that all elements are sorted (see figure 3).

Since each sweep makes N – 1 comparisons, the total number of passes necessary to completely sort the array is $(N - 1)^2$. In the case of large numbers, this is relatively close to N^2. This inefficiency of the bubble sort is compensated for by its simplicity, especially in understanding how it works. Consequently, it is quite acceptable for sorting tasks involving only small values of N.

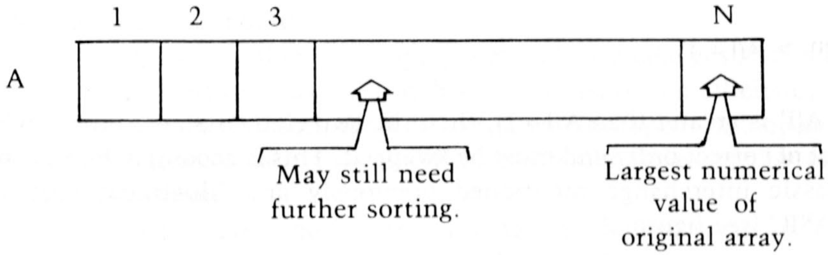

1
2
3
N
A
May still need further sorting.
Largest numerical value of original array.

Figure 2: Array after sweep of bubblesort

For those so inclined, there is a discussion of Quick Sort, another sorting algorithm presented in the program (see page 142).

```
10 FOR I = 1 TO N - 1
20 FOR J = 1 TO N - 1
30 IF A(J) < = A(J + 1) THEN 70
40 TEMP = A(J)
50 A(J) = A(J + 1)
60 A(J + 1) = TEMP
70 NEXT J
80 NEXT I
```

Figure 3

Sorting Implemented

The BASIC program in listing 1 provides implementations of visual sorts for the following five methods: bubble sort, straight insertion sort, selection sort, Shell sort, and quick sort. The visual display arranges the array to be sorted as a table of 20 different bar lengths, each bar length representing a number. The array table uses the random number generator from BASIC. If you wish no repeated numbers, the modification to the program is shown in figure 4.

```
FOR I = 0 TO LL : B(I) = N : NEXT I
FOR I = 0 TO N
L = RND (N + L) : IF B(L) < = 0 THEN REPEAT
B(L) = I : X = L : GOSUB DISPLAY
NEXT I
```

Figure 4

All values generated are positive and less than 19. This is due to horizontal space constraints of the display and does not reflect any inherent limitations in the algorithms themselves.

Each of the sorting algorithms are carried out in the program. As the array is sorted, the values displayed on the screen reflect the changes taking place internally. Various colors are used to highlight this. Each time a comparison is made, the bar being examined is displayed in blue. If it becomes a temporary storage number, it is switched to black. If it is swapped with another number, it changes to green. In any case, when the program is through with that number, the bar returns to red. Notice that some sorts hold temporary values longer than others and a bar may remain black for some time. You can get a good idea of how each algorithm does its job just by watching the pattern of colors on the screen. In addition, each sort prints on the top border of the display some additional information about what is happening.

The program begins with a menu and asks you for your choice of sort. After you have entered your selection, you may choose to have a new random pattern or repeat the previous random pattern. This is so that you can see the difference in the various sorts using the same pattern. You cannot repeat a pattern until you have sorted a new pattern.

To stop the program at any point, press the spacebar. To restart the sort, again press the spacebar. While the program is temporarily stopped, pressing either 'F' or 'S' to restart the program puts the sort into 'F(AST)' or 'S(LOW)' mode, respectively, to hurry the sort along or make it sort each element slowly so that you can better understand the steps involved in sorting an array. An X continues the sort for one step and stops again, waiting for another X to proceed one more step or for the spacebar to continue running.

A running total of comparisons and interchanges is kept at the top of the screen while the sort is in progress. This is printed on the menu screen for each sort while using the same random pattern. When a new pattern is selected, the old totals are cleared. To get a clearer idea of how long each sort takes, run the program through each type of sort, always specifying the same random order. After all five sorts are done, the menu page will contain a list of the running totals of comparisons and interchanges. Examine this list and you will get an accounting of how long it took each sort to complete its work. Timewise, an interchange is worth approximately two comparisons.

There is an additional sorting program supplied with BARSORT. Phone Book is an example of a bubble-sort program that sorts something useful — in this case a telephone log. The sort alphabetizes your personal telephone book. When run, the program prompts you to choose an option:

1. Create a new file (you can enter up to 20 names and telephone numbers).
2. Add, change, or delete names (you can alter a file that you have already created).
3. Load a file already saved on tape (you can load in your telephone book to look up numbers or make changes).
4. Save file to tape (you can put your tape recorder in record mode and save your telephone book on tape).
5. Sort file (you can sort the file into alphabetical order). If you want the file saved alphabetically, you must resave it at this time.

There are several points of interest in this program that do not occur in the other sort programs. The change-entry routine in lines 4000-4520 allows for a RETURN to indicate that the current entry is OK and does not need to be retyped. The entry is changed only if something new is typed in. Lines 7040 and 7050 set uppermost boundaries for the names and phone numbers so that there is sure to be an entry larger than any typed in. Lines 7130 and 7140 exchange both the name and the telephone number.

Telephone Book Variable Usage

Constants	
CURSOR	Location controlling cursor appearance
SIZE	Size of file
B$	String of 20 spaces
Variables	
A	Flag for visual or normal sort

AA	Flag for name or number sort
I	Number of names in file
J,K,L	Loop counters
Z	Input-character code
A$	Input character
F$	Name of file
N$	Name field
P$	Phone-number field
T$	Temporary input string
TEMP$	Temporary sort string

Bar Sort Variable Usage

Constants

LL	Maximum number of bars
B$	String of multi-colored dashes
C$	String of spaces
Q$	Names of sorts

Variables

A	Vertical printing coordinate
CO	Compare counter
F	Flag for bubble sort done
I,J,K,L,WW	Loop counters
M,SP,T,SP,V	Temporary storage
P,QQ	Partitions in quick sort
Q	Menu selection
QT	Delay counter
SW	Swap counter
X	Color code for printing bars
Z	GET character code
A$	Input character
CO$	Formatted CO for printing
SW$	Formatted SW for printing
X$	Single-step flag

Arrays

A( )	Random array for bar lengths
B( )	Current bar lengths
CO( )	Table of compares
S( )	Stack storage for quick sort
SP( )	Array for span lengths in Shell sort
SW( )	Table of swaps

The Bar Sort Program

The subroutine at line 1000 controls all the screen printing during the sorting operation. The variable A controls the vertical displacement of the print line; each line is printed A lines down from the top of the screen. Some sort routines may unknowingly send subroutine 1000 a value in A that is greater than LL, the last bar allowed. Line 1000 checks for this case and performs a RETURN if necessary. Line 1010 POSITIONs the cursor at the beginning of row A. It then prints a row of dashes, whose length is specified by B(A) and whose color is specified by X. B$ is a string of dashes that are used to make up the bar lines. The amount of line printed (B(A)) corresponds to the random number assigned to that position in the array. C$ is a line of spaces and is used to fill out the rest of the line, which erases any previous lines that might have been longer in length. Lines 1020-1030 convert the value CO, the number of compares, and SW, the number of swaps, into 3-character, right-justified strings. Line 1040 prints the compares and swaps at the top of the screen.

The input subroutine at line 2000 looks for certain key characters that tell the program how to proceed. An 'X' is the single-step character and causes the program to proceed only until the next print subroutine call. A space is the indicator to pause in the sort until another key is pressed. The 'F' character (for FAST) sets the delay timer QT to 1, the smallest possible delay. 'S' (for SLOW) sets the delay timer to 200. The subroutine is also called at line 2030 to read a key from the keyboard during menu choices.

The subroutine at line 3000 prints the compare and swap totals next to the menu names. The subroutine at line 4000 clears the compare and swap totals when changing random patterns. Line 6000 initializes the array with random numbers between 1 and 18.

Bubble sort is at line 7000. The compare is made at line 7020 and swap, if necessary, at line 7060. The flag 'F' is set every time there is a swap. If the sort goes through the entire loop without a swap, then it is finished and the flag returns unset and, in line 7040, sets the loop counter to LL, which finshes the sort.

The straight insertion-sort subroutine begins at line 8000. The compare is made in line 8020 and elements are swapped in line 8040 when necessary. T is used to hold each succeeding element while the compares are made back to the beginning of the array to see where the current T line is to be inserted.

The selection sort saves the largest line currently found in M. The compare is at line 9030 and swapping is done at line 9065. M is used to store the largest element currently found, which is moved to the bottom position at the end of each loop with the bottom position progressing toward the beginning of the array.

The subroutine at line 10000 sorts by the Shell-Metzner method, which searches the array by decreasingly smaller intervals. The SP( ) array contains the intervals that are used in the loop counters. The STEP increments of SP are 10, 6, 4, 2, and 1, which increasingly fine-tune the sort until completed. The compare is done in line 10040 and swapped, when necessary, at 10050.

Quick sort is located at line 11000 and is the most complicated sort (see shaded box). Compares are made not only of the elements, but also of the loop counters. In addition to swaps of the elements, one or more of the elements may be stored on a temporary stack until a suitable place in the array is found to put them. It is also the quickest of the sorts.

Telephone Book

A sample program to demonstrate how sorting could be used in a useful manner is included with BAR SORT. Load the program PHONE and type 'RUN'.

Since we don't have any files saved on tape, type '1' to create a new file. When asked for the name of the file, press return to accept the default name of 'D:PHONEFIL'. At this time you could enter up to 20 different people and their telephone numbers.

For this trial run put in your name, last name first as prompted. Do not put a comma after your last name as that would indicate to the Atari that you have two inputs on that line. After your first name, press return. Then enter your telephone number in the form 000-000-0000. Press return after that entry also. Enter one other person and number at this time. When prompted to enter name again, type 'DONE' in upper or lower case and press return. This will return you to the main menu.

You have created a short telephone file of two names, which is in the computer's memory but not yet saved. There are several options listed on the menu that you can perform now. Choosing the second menu selection will allow you to thumb through the file, one name at a time. After each name and number is displayed, you may DELETE the entry by pressing 'D', CHANGE all or part of the entry by pressing 'C', or CONTINUE through the names by pressing RETURN. During the CHANGE operation, pressing RETURN signifies that the old entry is acceptable; typing new information replaces the old entry with the new.

The LOAD and SAVE options work similarly to the Atari Player load and save. For the LOAD option, you must supply the name of the file. Cassette-saved files must include the 'C:' before the name, and disk-saved files must include the 'D:'. Choosing the SAVE option stores the file using the filename you supplied before creating it.

The SORT selection will perform a bubble sort on either the name field or the phone-number field. You also have the option of watching the sort or having the sort done more quickly and returning you to the menu when finished. When in the visual sort mode, you must press a key for each cycle through the sorting algorithm. This slows the sort down enough so that you can see the interchanges.

Exit CLOSEs the keyboard device and returns you to BASIC in graphics mode 0.

The Telephone Program

Line 10: SIZE may be set as large as memory allows to accommodate a larger number of names and phone numbers. Setting it to 21 here allows up to 20 names in a file. The other space is reserved for the sort routine; it is used to hold a string of 'Z's so that there is a name within the file that is alphabetically higher than any of the others. CURSOR is equal to memory location 752. If this location contains a 0, then the cursor is displayed. If it contains a 1, the cursor is invisible. Line 20 DIMensions the strings that will be used within the program. Line 30 sets up the character, background, and border colors. Line 40 OPENs the keyboard as input device number 1 for use with GET statements within the program. Lines 50 and 60 set up the strings N$ and P$, respectively, filling each with spaces. Line 70 creates B$ as a string of 30 spaces for padding names that are less than 30 characters long.

Line 1000 is the get-character subroutine. The program waits for the single character A$ to be typed on the keyboard before RETURNing.

Lines 2000-2090 display the main menu and request an option choice. Line 2100 first clears the variables A and AA, which are used in the sort routine. It then inputs a character through subroutine 1000, checking its value to make sure it is a valid menu number (between 1 and 6). Line 2110 performs a GOSUB to the appropriate subroutine, using the VALue of A$ to calculate the line number to which it should go. Then the menu is called again with a GOTO 2000.

Line 3000 is the routine to add names to the file. As in the LOAD routine, a filename must be supplied including the 'C:' or 'D:' that should precede the name. Lines 3040-3070 print a few lines of instruction for the input, and lines 3080-3120 print the input prompts and keep track of the number of entries.

Line 4000: The add, change, and delete routine lists the file names one at a time and checks to see if you want to make changes or deletions. If you type a 'C', you may change the name and/or the phone number of the entry just displayed. If you type a 'D', that entry will be deleted from the file and the next one will be displayed. If you just press RETURN, the next entry will appear. When the end of the file is reached, the routine checks to see if you want to add any more names, in which case it jumps to the add routine or else it returns to the menu.

Line 5000: As at the beginning of the file-creation routine, a filename must be supplied. Line 5020 checks for a RETURN as input, which assigns the default name of 'D:PHONEFIL' to the file. Once the name is supplied, the file is loaded into memory. The number of entries in the file is stored in the first 30 characters of N$. The message 'File loaded' is displayed on the screen for a few moments before the menu is called again. Lines 6000 and on save the current file using either the name it was given upon its creation or the one given as it was loaded from disk or tape.

Line 7000: A bubble sort is performed on either the names (N$ field) or the phone numbers (P$ field), AA = 0 for name sort and AA = 1 for number sort. The compares in lines 7070 and 7080 are executed according to AA. The subroutine at line 7130 does the interchanges and the one at line 7160 prints the display during the sort, indicating with an asterisk the entry being compared.

Line 8000 CLOSEs device number 1 (the keyboard), returns to graphics mode 0, and ENDs the program.

Programming Changes

The other sorts in the BARSORT demonstration could be implemented in this program by substituting the code from BARSORT in place of the bubble-sort code in the subroutine at line 7000. Make sure that I is set to the number of entries currently on file.

If you have additional memory, other fields could be set up for an address (including city, state, and zip fields) by adding the necessary prompts. It might be especially useful to have the sort on the zip-code field for mailing-list purposes.

Quick Sort

This sorting algorithm, invented by C.A.R. Hoare, is probably the most elegant of the sorting techniques yet devised. It is of the N log N class, based on a very simple idea and programmable in very few lines of code. The greatest difficulty in understanding how it works involves the details of applying the basic step that controls its operation. There is a tendency to ask, "You mean, that's all there is to it?" or "What do you mean to simply apply the same procedure to both halves?" Nonetheless, once appreciated, the algorithm is one you will never forget.

The basic idea underlying quick sort is to perform interchanges of non-adjacent array elements in hopes of bringing order to the array more quickly. (You have already seen the inefficiency of interchanging adjacent elements in bubble sort.) The idea is applied using the concept of *partitioning*. Since this is a difficult programming concept, first we will describe a quick sort in an analogy that everyone can grasp.

You have a pile of papers to alphabetize. The first step is to run through all the papers, dividing them into two groups — those preceding 'M' in the alphabet and those following 'M' in the alphabet, including 'M.' You can see that if the first pile, A-L, is alphabetized and the second pile, M-Z, is alphabetized, all that would remain to be done is place the A pile on top of the M pile. This is what quick sort does. Once the two subarrays are sorted, the entire array is sorted since each subarray contains only elements on its side of the partition.

In this case the partition chosen was 'M.' However, how do you alphabetize the two subarrays? Here is where you have to have a good imagination. Make a new partition in each sub-pile, then divide those subarrays. For example, take the A-L pile and divide it into two piles of A-E and F-L. When these are sorted into order, the A-L pile is sorted. Next, divide the A-E pile into A-C and D-E. This is continued until there is only one element in each partition, which means that there are no further partitions possible and that the sub-piles are alphabetically sorted. If this is done with each sub-pile, eventually all the sub-piles are sorted and the whole array is sorted alphabetically.

We will not attempt to prove that this is a much superior way to sort; you can see that for yourself when you run the sample sort programs from the tape. Time both the bubble sort and the quick sort to see which is faster.

Table 1 Powers and Logs of Numbers

N	N^2	Log_2 N	N Log_2 N
64	4,096	6	384
128	16,384	7	896
256	65,536	8	2,048
512	626,144	9	4,608
1,024	1,048,576	10	10,240
2,048	4,194,304	11	22,528
4,096	16,777,216	12	49,152
8,192	67,108,864	13	106,496
16,384	268,435,456	14	229,376
32,768	1,073,741,824	15	491,520
65,536	4,294,967,296	16	1,048,576
131,072	17,179,869,184	17	2,228,224
262,144	68,719,476,736	18	4,718,592
524,288	274,877,906,944	19	9,961,472
1,048,576	1,099,511,626,776	20	20,971,520

BAR SORT Listing

```
1 GOTO 12000
1000 IF A>LL THEN RETURN
1010 POSITION 0,A:PRINT #6;B$(X*19
    +1,X*19+B(A));C$(B(A)+1,19);
1020 CO$="   ":CO$(4-LEN(STR$(CO))
    ,3)=STR$(CO)
1030 SW$="   ":SW$(4-LEN(STR$(SW))
    ,3)=STR$(SW)
1040 POSITION 7,0:PRINT #6;"CMP";C
    O$;" SWP";SW$;
2000 IF X$="X" THEN X$="":GOTO 203
    0
2010 Z=PEEK(764):IF Z=33 THEN POKE
     764,255:GOTO 2030
2020 FOR WW=1 TO QT:NEXT WW:RETURN

2030 GET #1,Z:A$=CHR$(Z)
2040 IF A$="X" THEN X$="X"
2050 IF A$="F" THEN QT=1
2060 IF A$="S" THEN QT=200
2070 RETURN
3000 IF CO(I)=0 THEN PRINT #6:RETU
    RN
3010 POSITION 15-LEN(STR$(CO(I))),
    I*2+4:PRINT #6;STR$(CO(I));
3020 POSITION 19-LEN(STR$(SW(I))),
    I*2+4:PRINT #6;STR$(SW(I))
3030 RETURN
4000 FOR I=0 TO 5:CO(I)=0:NEXT I:R
    ETURN
6000 FOR I=1 TO LL:A(I)=INT(RND(0)
    *18+1):NEXT I:RETURN
7000 FOR J=1 TO LL:F=0:A=1:X=1:GOS
    UB 1000:FOR L=1 TO LL-J
7010 CO=CO+1:A=L+1:X=2:GOSUB 1000
7020 IF B(L)>B(L+1) THEN GOSUB 706
    0
7030 X=1:GOSUB 1000:A=L:X=0:GOSUB
    1000
7040 NEXT L:A=L:X=0:GOSUB 1000:IF
    F=0 THEN J=LL
7050 NEXT J:RETURN
7060 A=L:X=3:GOSUB 1000:T=B(L):B(L
    )=B(L+1):B(L+1)=T
7070 SW=SW+1:A=L+1:X=3:GOSUB 1000:
    F=1:A=L+1:RETURN
8000 FOR I=2 TO LL:T=B(I):A=I:X=1:
    GOSUB 1000
8010 FOR J=I-1 TO 1 STEP -1:CO=CO+
    1:A=J:X=2:GOSUB 1000
8020 IF T>=B(J) THEN A=J:X=0:GOSUB
     1000:GOTO 8060
```

```
8030 SW=SW+1:A=J+1:X=3:GOSUB 1000:
     A=J:GOSUB 1000
8040 B(J+1)=B(J):A=J+1:X=0:GOSUB 1
     000
8050 A=J:GOSUB 1000:NEXT J:A=I:GOS
     UB 1000
8060 B(J+1)=T:A=J+1:GOSUB 1000:NEX
     T I
8070 RETURN
9000 FOR I=0 TO LL-1
9010 M=1:A=M:X=1:GOSUB 1000
9020 FOR J=2 TO LL-I
9030 A=J:CO=CO+1:X=2:GOSUB 1000:X=
     0:GOSUB 1000:IF B(J)<=B(M) THE
     N 9050
9040 A=M:X=0:GOSUB 1000:A=J:X=1:GO
     SUB 1000:M=J
9050 NEXT J:IF I=LL-1 THEN 9070
9060 SW=SW+1:A=M:X=3:GOSUB 1000
9065 T=B(M):B(M)=B(LL-I):B(LL-I)=T
9070 A=LL-I:X=3:GOSUB 1000:X=0:GOS
     UB 1000:A=M:GOSUB 1000
9080 A=LL+1-I:IF A<LL+1 THEN X=0:G
     OSUB 1000
9090 NEXT I:A=1:GOSUB 1000:RETURN
10000 FOR I=1 TO 5:SP=SP(I):FOR J=
     SP TO LL
10010 T=B(J):A=J:X=1:GOSUB 1000
10020 FOR K=J-SP TO 1 STEP -SP:IF
     K=0 THEN 10040
10030 A=K:CO=CO+1:X=2:GOSUB 1000
10040 IF T>=B(K) THEN 10080
10050 SW=SW+1:A=K:X=3:GOSUB 1000:B
     (K+SP)=B(K)
10060 A=K+SP:X=3:GOSUB 1000:X=0:GO
     SUB 1000
10070 NEXT K
10080 X=0:GOSUB 1000:B(K+SP)=T
10090 A=K+SP:X=0:GOSUB 1000
10100 NEXT J:NEXT I:RETURN
11000 B(LL+1)=32767:P=1:QQ=LL:TP=0
11010 IF P>=QQ THEN 11190
11020 K=QQ+1:V=B(P):I=P:J=K:A=P:X=
     1:GOSUB 1000
11030 J=J-1:IF B(J)<=V THEN 11050
11040 A=J:CO=CO+1:X=2:GOSUB 1000:X
     =0:GOSUB 1000:GOTO 11030
11050 I=I+1:IF I>LL THEN 11070
11060 A=I:CO=CO+1:X=2:GOSUB 1000:X
     =0:GOSUB 1000
11070 IF B(I)>=V THEN 11090
11080 GOTO 11050
11090 IF J<=I THEN 11130
11100 A=I:SW=SW+1:X=3:GOSUB 1000:A
     =J:GOSUB 1000:X=0
```

```
11110 T=B(I):B(I)=B(J):B(J)=T
11120 A=I:GOSUB 1000:A=J:GOSUB 100
      0:GOTO 11030
11130 B(P)=B(J):B(J)=V:SW=SW+1
11140 A=P:GOSUB 1000:A=J:X=3:GOSUB
       1000:X=0:GOSUB 1000
11150 IF J-P<QQ-J THEN 11170
11160 S(TP+1)=P:S(TP+2)=J-1:P=J+1:
      GOTO 11180
11170 S(TP+1)=J+1:S(TP+2)=QQ:QQ=J-
      1
11180 TP=TP+2:GOTO 11010
11190 IF TP=0 THEN 11210
11200 QQ=S(TP):P=S(TP-1):TP=TP-2:G
      OTO 11010
11210 RETURN
12000 GRAPHICS 17:SETCOLOR 0,4,8
12010 SETCOLOR 1,0,0:SETCOLOR 2,9,
      8
12020 SETCOLOR 3,12,10:SETCOLOR 4,
      0,14
12025 OPEN #1,4,0,"K:"
12030 DIM A(21),B(21),SP(5),CO(5),
      SW(5),S(10):LS=20
12040 SP(1)=10:SP(2)=6:SP(3)=4:SP(
      4)=2:SP(5)=1
12050 DIM Q$(30),B$(76),C$(20),A$(
      1),X$(1),CO$(3),SW$(3):B(21)=3
      0
12060 Q$="BUBBLEINSERTSELECTSHELL
      QUICK "
12070 B$="------------------______
      ____________------------------
      ----                          "
12080 C$="                    "
12090 PRINT #6;"K":LL=20
12100 PRINT #6;" ATARI BAR SORTER
      "
12110 PRINT #6:PRINT #6:PRINT #6;"
      WHICH SORT? CMP SWP"
12120 FOR I=1 TO 5:PRINT #6:PRINT
      #6;I;" ";Q$(I*6-5,I*6);:GOSUB
      3000:NEXT I
12130 PRINT #6:PRINT #6;"6 QUIT"
12140 PRINT #6:PRINT #6:PRINT #6;"
      CHOOSE":CO=0:SW=0:GOSUB 2030
12150 Q=Z-48:IF Q<1 OR Q>6 THEN GO
      SUB 2030:GOTO 12150
12160 PRINT #6;"K":IF Q=6 THEN END

12170 POSITION 0,3:PRINT #6;"WHICH
      ?"
12180 POSITION 0,6:PRINT #6;"RANDO
      M BAR LENGTHS"
```

```
12190 PRINT #6:PRINT #6;"KEEP SAME
     LENGTHS"
12200 PRINT #6:PRINT #6:PRINT #6;"
    CHOOSE":GOSUB 2030
12210 IF A$="R" THEN GOSUB 6000:GO
     SUB 4000:GOTO 12240
12220 IF A$="K" AND B(1) THEN 1224
     0
12230 GOTO 12170
12240 FOR I=1 TO LL:B(I)=A(I):NEXT
     I
12250 PRINT #6;"↖";Q$(Q*6-5,Q*6);:
     X=0:FOR I=1 TO LL:A=I:GOSUB 10
     00:NEXT I
12260 POSITION 0,22:PRINT #6;" COM
    PARE temp swap";
12270 GOSUB 6000+Q*1000
12280 POSITION 0,22:PRINT #6;"    P
     RESS ANY KEY    ";
12290 GOSUB 2030:CO(Q)=CO:SW(Q)=SW
     :GOTO 12090
```

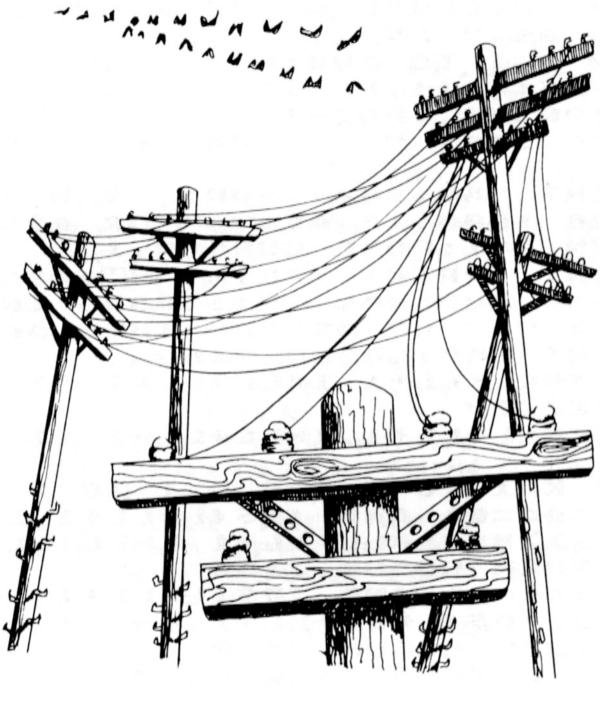
TELEPHONE BOOK Listing

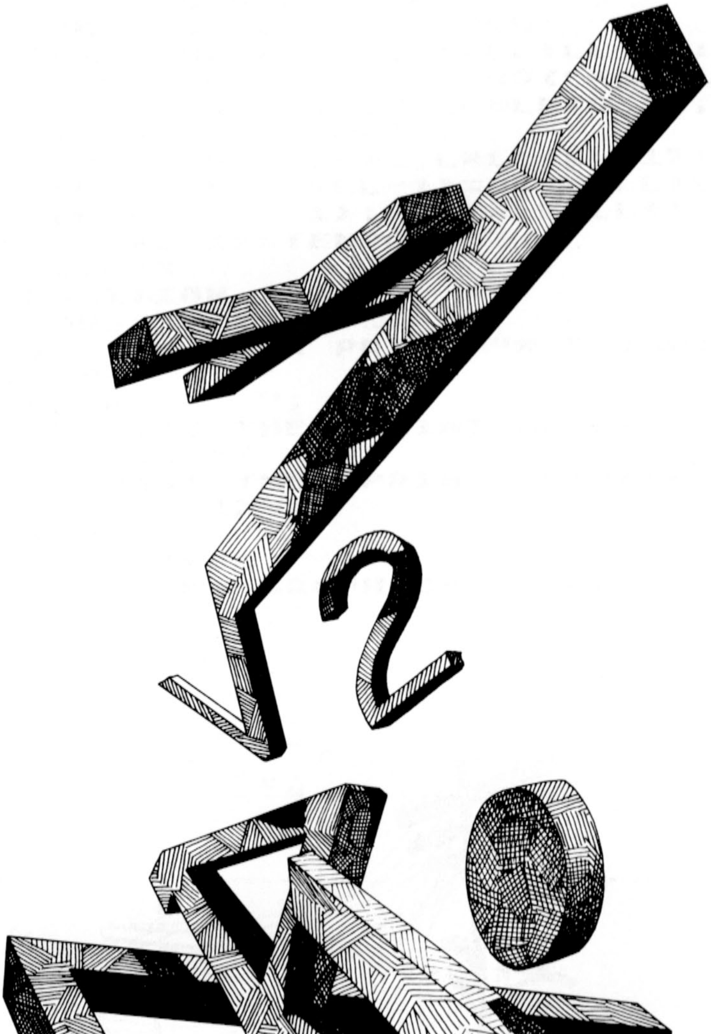
```
10 SIZE=22:LL=30:CURSOR=752
20 DIM N$(SIZE*LL),P$(SIZE*LL),A$(
    1),F$(15),T$(255),B$(LL),TEMP$
    (LL)
30 SETCOLOR 1,0,0:SETCOLOR 2,12,10
    :SETCOLOR 4,7,6
40 OPEN #1,4,0,"K:"
50 N$=" ":N$(SIZE*LL,SIZE*LL)=" ":
    N$(2)=N$
60 P$=" ":P$(SIZE*LL,SIZE*LL)=" ":
    P$(2)=P$
70 B$="
      "
80 GOTO 2000
1000 GET #1,Z:A$=CHR$(Z):RETURN
2000 PRINT "↖↓↓↓                Ata
    ri"
2010 PRINT "↓           Telephone 
    Log"
2020 PRINT "↓↓                  MENU"
2030 PR:PRINT "↓↓   1. Create new
     file"
2040 PRINT "      2. Add,Change,De
     lete names"
2050 PRINT "      3. Load existing
      file"
2060 PRINT "      4. Save file"
2070 PRINT "      5. Sort file"
```

```
2080 PRINT "          6. Exit"
2090 PRINT "↓↓          Your choice?";
2100 A=0:AA=0:GOSUB 1000:IF ASC(A$
    )<49 OR ASC(A$)>54 THEN 2100
2110 GOSUB VAL(A$)*1000+2000:GOTO
    2000
3000 PRINT "↰Name of file?"
3010 PRINT :PRINT "<Return> for de
    fault of D:PHONEFIL"
3020 PRINT :INPUT F$:IF LEN(F$)=0
    THEN F$="D:PHONEFIL"
3030 I=1
3040 PRINT "↰Enter Last Name (Spac
    e)"
3050 PRINT "First Name <Return>"
3060 PRINT "and Phone Number 000-0
    00-0000"
3070 PRINT "DONE when through":PRI
    NT
3080 PRINT "Name ";:INPUT T$
3090 IF T$="DONE" OR T$="Done" OR
    T$="done" THEN I=I-1:N$(1,LL-1
    )=STR$(I):RETURN
3095 N$(I*LL+1,I*LL+LL-1)=T$
3100 PRINT "Phone ";:INPUT T$:P$(I
    *LL+1,I*LL+LL-1)=T$
3110 IF I>SIZE-2 THEN RETURN
3120 I=I+1:GOTO 3080
4000 PRINT "↰";:J=0
4010 J=J+1:PRINT
4020 PRINT N$(J*LL+1,J*LL+LL-1)
4030 PRINT P$(J*LL+1,J*LL+LL-1)
4040 PRINT :PRINT "Chng or Del <Re
    turn> to continue";
4050 GOSUB 1000:PRINT
4060 IF A$=CHR$(155) THEN 4100
4070 IF A$="d" OR A$="D" THEN N$(J
    *LL+1)=N$(J*LL+LL+1):P$(J*LL+1
    )=P$(J*LL+LL+1):I=I-1:J=J-1:GO
    TO 4100
4080 IF A$<>"c" AND A$<>"C" THEN 4
    100
4090 PRINT "New name ";:INPUT T$:I
    F LEN(T$)>0 THEN N$(J*LL+1,J*L
    L+LL-1)=T$
4095 PRINT "New phone ";:INPUT T$:
    IF LEN(T$)>0 THEN P$(J*LL+1,J*
    LL+LL-1)=T$
4100 IF J<I THEN 4010
4110 PRINT :PRINT "Add more names?
     Y/N";:GOSUB 1000:PRINT
4120 IF A$="y" OR A$="Y" THEN I=I+
    1:PRINT :GOTO 3080
4130 RETURN
```

```
5000 PRINT "↰What is the name of t
     he existing file?"
5010 PRINT "C:  for cassette, D:fi
     lename  for disk"
5015 PRINT "<Return> for default o
     f D:PHONEFIL"
5020 PRINT :INPUT F$:IF LEN(F$)=0
     THEN F$="D:PHONEFIL"
5030 IF F$(1,2)<>"C:" THEN 5050
5040 PRINT "Insert file tape and p
     ress <Return> ";:GET #1,Z
5050 OPEN #2,4,0,F$:N$="":P$="":TR
     AP 5100:INPUT #2;SIZE
5060 INPUT #2;T$:N$(LEN(N$)+1)=T$:
     INPUT #2;T$:P$(LEN(P$)+1)=T$
5070 GOTO 5060
5100 CLOSE #2:I=VAL(N$(1,LL-1)):PR
     INT "File loaded"
5110 FOR J=1 TO 300:NEXT J:RETURN
6000 PRINT "↰Saving file"
6010 IF F$(1,2)<>"C:" THEN 6030
6020 PRINT "Insert file tape and p
     ress <Return> ";:GET #1,Z
6030 OPEN #2,8,0,F$:PRINT #2;SIZE
6040 FOR J=1 TO LEN(N$) STEP 200:K
     =J+199:IF K>LEN(N$) THEN K=LEN
     (N$)
6050 PRINT #2;N$(J,K):PRINT #2;P$(
     J,K)
6060 NEXT J:CLOSE #2:PRINT "File s
     aved as ";F$
6070 FOR J=1 TO 300:NEXT J:RETURN
7000 IF N$(LL+1,LL+1)=" " THEN PRI
     NT :PRINT :PRINT "        No Fi
     le to Sort ";:GOSUB 1000:RETUR
     N
7010 PRINT "↰Visual SORT  Y/N ?";:
     GOSUB 1000:IF A$="y" OR A$="Y"
      THEN A=1:POKE CURSOR,1
7020 PRINT :PRINT :PRINT "Names or
      Phone numbers ?";:GOSUB 1000:
     IF A$="p" OR A$="P" THEN AA=1
7030 PRINT "↰":GOSUB 7160
7040 N$(I*LL+LL+1,I*LL+LL*2-1)="ZZ
     ZZZZZZZZ"
7050 P$(I*LL+LL+1,I*LL+LL*2-1)="::
     :-:::-::::                 "
7060 FOR K=1 TO I:FOR L=1 TO I
7070 IF (AA=0) AND (N$(L*LL+1,L*LL
     +LL-1)>N$(L*LL+LL+1,L*LL+LL*2-
     1)) THEN GOSUB 7130
7080 IF (AA=1) AND (P$(L*LL+1,L*LL
     +LL-1)>P$(L*LL+LL+1,L*LL+LL*2-
     1)) THEN GOSUB 7130
7100 NEXT L:IF A THEN GOSUB 7160
```

```
7105 NEXT K
7110 PRINT "Press any key to conti
   nue ";:GOSUB 1000
7120 POKE CURSOR,0:RETURN
7130 TEMP$=N$(L*LL+1,L*LL+LL-1):N$
    (L*LL+1,L*LL+LL-1)=N$(L*LL+LL+
    1,L*LL+LL*2-1):N$(L*LL+LL+1,L*
    LL+LL*2-1)=TEMP$
7140 TEMP$=P$(L*LL+1,L*LL+LL-1):P$
    (L*LL+1,L*LL+LL-1)=P$(L*LL+LL+
    1,L*LL+LL*2-1):P$(L*LL+LL+1,L*
    LL+LL*2-1)=TEMP$
7150 RETURN
7160 POSITION 2,0
7170 FOR J=1 TO TO I
7180 IF J=K THEN PRINT "* ";:GOTO
    7190
7185 PRINT "  ";
7190 IF AA=0 THEN PRINT N$(J*LL+1,
    J*LL+LL-1)
7200 IF AA=1 THEN PRINT P$(J*LL+1,
    J*LL+LL-1)
7220 NEXT J
7230 GOTO 1000
8000 CLOSE #1:GRAPHICS 0:END
```

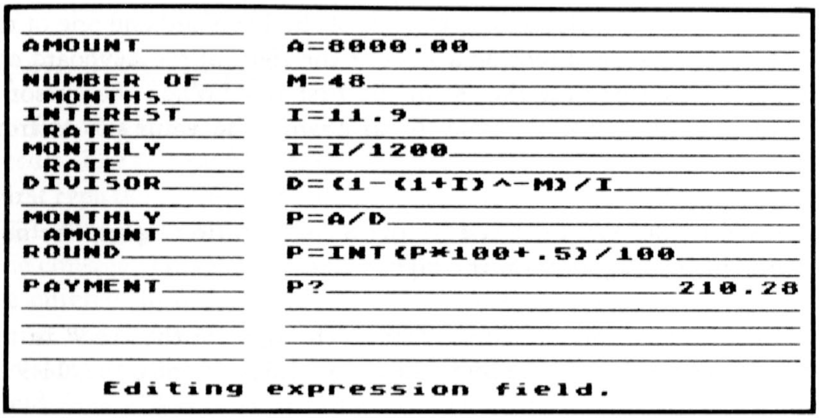




√2

MICRO Calc

A Miniature Spreadsheet



```
AMOUNT            A=8000.00
NUMBER OF         M=48
  MONTHS
INTEREST          I=11.9
  RATE
MONTHLY           I=I/1200
  RATE
DIVISOR           D=(1-(1+I)^-M)/I
MONTHLY           P=A/D
  AMOUNT
ROUND             P=INT(P*100+.5)/100
PAYMENT           P?                  210.28

        Editing expression field.
```

The "electronic spreadsheet" is one of the most popular types of business programs available. The first program of this kind, *VisiCalc* (sold by VisiCorp), made quite an impact on the business world and many offshoots of that original program are now on the market. A spreadsheet program lets you perform mathematical computations on the video display, and applications range from accounting to inventory to printing bar graphs. MICRO Calc is a miniaturized version of a spreadsheet program. With the touch of a key you can perform even very elaborate calculations. Once your sheet is defined for a particular application, you can save it and use it again and again for that application.

Operating Instructions

1. Load in the MICRO Calc program and RUN it. Hit the ESC key and then the console OPTION key. At the save/load/edit prompt type 'L', which signifies a load operation. At the filename prompt, type 'D:MSAMPLE'. This will cause a demonstration page to be loaded from disk and displayed as shown on the previous page.

2. You are now in the editing mode of the program. If you were to type any of the keys now, the information on the spreadsheet would change. Type ESC and at the following prompt hit the console START key. This executes the instructions on the screen and after a few seconds the number 210.28 should appear on the eighth line. You have just calculated the monthly payment on an $8000 car with a 48-month term and 11.9 annual percentage rate.

3. Hold down the CTRL key at the left of the keyboard and one of the appropriate arrow directional keys at the right of the keyboard and move the cursor over the '8' in the first line. When the cursor is right over the number, type '6' to change the value of 'A' from $8000 to $6000.

4. Press the ESC key followed by the START button again; the new payment now appears on the eighth line.

5. This time position the cursor just to the right of the '11.9' in the third line and hit the DELETE key four times to erase the '11.9'.

6. Type in a new value of '9.9', hit ESC, push START, and see how much difference a couple of percentage points makes.

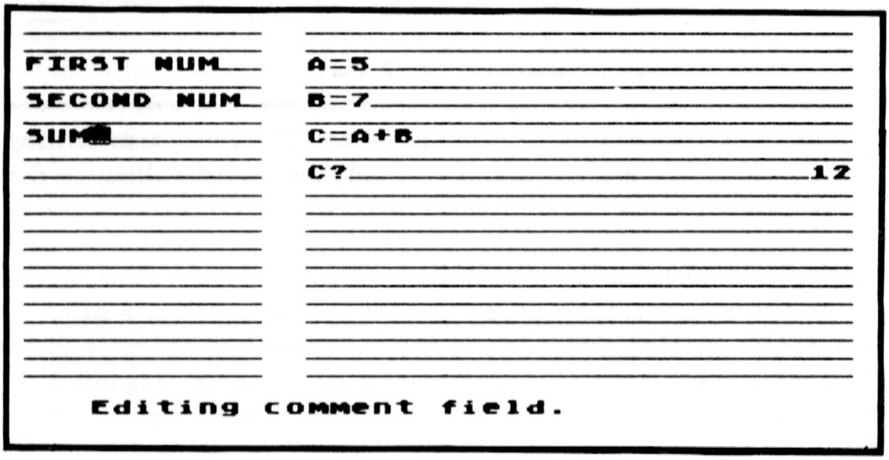
Using MICRO Calc

Load in and RUN MICRO Calc now. If you have MICRO Calc in memory already, then simply hold down the Shift key and the Clear key to clear the screen. After a few seconds the screen will fill with a series of lines arranged in two columns of twenty rows each. The longer lines at the right are for the numerical expressions. The smaller lines to the left are for comments. When the program is run, it defaults to editing the expression field as indicated by the message line at the bottom of the screen.

Notice the reverse underline (white block) at the corner of the expression field. This is the cursor and it simply shows where the next character typed will be placed. The cursor is moved the same way as in BASIC — through the use of the CTRL key and one of the four arrow keys. Hold down CTRL and one of the arrow keys. Note that you can't move the cursor out of the editing field. This insures that the screen stays neat. Later I will dicsuss an easy way to edit the comment field.

A Simple Example

For this first example we want to add two numbers and print the result. The two numbers are labeled A and B and the sum defined as C. First type the right-hand portion of each of the following lines, pressing RETURN after each. Hit the ESC key and then the SELECT button on the console. This switches the editing fields so that the comments can be entered. In other words, if the expression field were in use at the time, the SELECT key would enable you to edit the comment field. Pressing the SELECT key again will allow you to edit the expression field. The SELECT key simply toggles (switches back and forth) the editing fields.



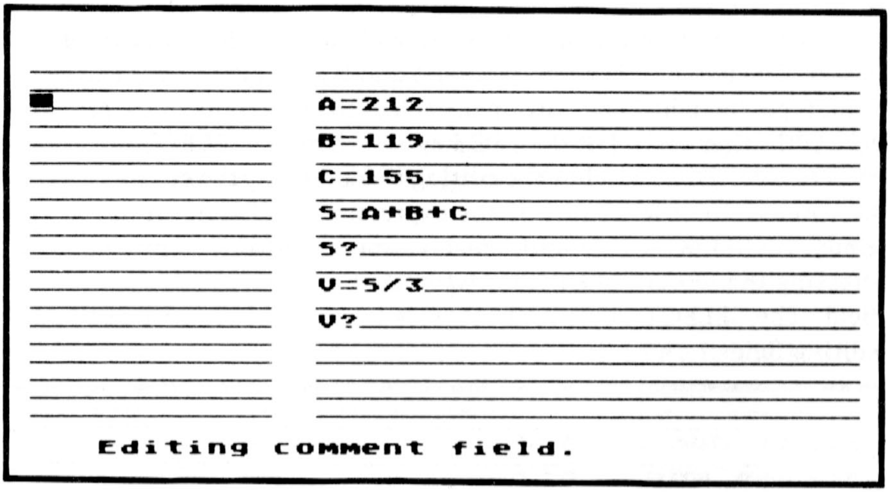
```
FIRST NUM        A=5
SECOND NUM       B=7
SUM              C=A+B
                 C?                      12

     Editing comment field.
```

Now press ESC followed by START. The cursor disappears for a second, the answer '12' (the sum of 5 and 7) is printed after the 'C?', and the cursor reappears where it was left before STARTing. To change the value of A to equal 2, move the cursor over to the '5', type '2', and then reSTART. The new answer (9) appears after the 'C?'.

The formula may also be changed. For example, to make 'C = A – B', move the cursor over to the ' + ' and type ' – '. When you reSTART, the new answer, ' – 5', appears.

Now let's try a more practical example — calculating averages. Use shifted CTRL/Clear key to clear the screen and type in the following.

START the program and the sum of the three scores, '486', will appear after the S? on the fifth line and the average of the scores, '162', after the V? on the seventh line. This procedure could be used to calculate bowling scores. Any three numbers you want to average could be used. Note that a more flexible method of averaging would be to put the averaging lines further down the screen so they could be adapted to handle more than three numbers. Do not worry about blank lines since they are ignored.



```
A=212
B=119
C=155
S=A+B+C
S?
V=S/3
V?
```

Editing comment field.

To show how similar programming MICRO Calc is to writing BASIC, here is a BASIC program that does the same thing as the averages sheet above.

```
10 INPUT"FIRST SCORE";A
20 INPUT"SECOND SCORE";B
30 INPUT"THIRD SCORE";C
40 S=A+B+C
50 PRINT S
60 V=S / 3
```

```
70 PRINT V
80 GOTO 10
```

The Rules

MICRO Calc, as you can see, is easy to use. However, it has some rules that must be followed to avoid errors.

1. Nothing should be typed after a '?'. That is where the program fills in its result.
2. After an '=', you must type either a number or a BASIC expression that evaluates to a number. Any BASIC function that yields a numerical result, such as TAN, SQR, RND, LOG, or ABS, may be used. BASIC string functions may not be used unless they yield a number.

Consult your *Atari BASIC Reference Manual* to learn about BASIC's built-in functions and numerical expressions, then use MICRO Calc to help you understand how they work.

3. Do not type in any BASIC commands or statements such as PRINT, RUN, NEW, CLOAD, etc., as *they will be executed at start time.* There are a few commands and statements that will work (such as SETCOLOR) but for safety's sake, the general rule-of-thumb is to avoid them.
4. *Never* use a variable name that ends with '0' as it could conflict with the variables used by MICRO Calc itself. Sometimes you will be able to get away with it, but results are unpredictable and total anihilation of the program is possible.

MICRO Calc allows you to use BASIC storage locations called "variables." As in BASIC, each variable is uniquely identified by a letter or series of letters of the alphabet. These letters may be followed by a digit or a series of digits provided that the digit is not '0'. A variable length is limited only to MICRO Calc's line length (which is 25). Therefore, all variable names must be somewhat shorter.

In our first example, when we typed 'A = 5', we assigned the value of 5 to the variable A, we typed 'B = 6' to assign the value 6 to B, and we typed 'C = A + B' to assign the sum of A and B to C. Until it is changed, 5 is substituted for A whenever it is used in an expression. Also, B and C retain their values until they are changed.

As in a BASIC program, MICRO Calc looks at the lines on the screen from the top down. Therefore, a 'C?' on a line before C is defined will give the wrong answer. Also, if you assign a variable a new value, the new value will be used in all subsequent calculations.

Editing

The MICRO Calc editing capabilities are a subset of the ATARI screen-editing capabilities. Nearly all of the editing functions of the ATARI were duplicated for MICRO Calc, from cursor movement to the repeat-key function, where holding down a key for more than half a second causes it to repeat. In fact, the only BASIC editing abilities that are not in this editor are the scroll line left, scroll line right, delete line, and insert line. These abilities have been omitted to insure that no accidents occur on the screen.

The screen editor supplied is cursor-oriented. This means that changing anything on the screen can be accomplished by moving the cursor over to that section and simply retyping it. Cursor movement is done by holding down the CTRL key while striking one of the arrow keys. Remember that you cannot move the cursor out of the editing field as the cursor merely 'wraps' around to the other side of the field.

There are three other keys that can be used to edit the screen. The return key simply positions the cursor at the start of the line below. The cursor will wrap to the top of the screen if it is at the bottom when the return key is pressed.

The clear key clears the screen of all text, effectively starting you off with a clean slate. This, as you might guess, is a potentially hazardous ability, so its use has been safeguarded in a couple of ways. First, the CTRL or SHIFT key must be held down when you hit the clear key; otherwise the computer will simply ignore it. Second, the computer will ask you if this is really what you want to do. If it is (you don't want what is on the screen) type 'Y'; otherwise type any other key.

The delete key deletes the character just to the left of the cursor and then moves the cursor over to that position. Also, to keep things neat, it removes the lines under the character it erases. If you are at the first character position of the line, the delete key simply wraps around to the end of that same line to delete the character there.

Avoiding Errors

To minimize typographical errors, MICRO Calc disallows characters that could cause trouble, such as lower- and upper-case letters and CTRL graphics characters. However, if you are not careful it is possible to make a RUN-time (or perhaps START-time) error. The program will stop and an error message, similar to those found in BASIC, will be printed. Because it uses the BASIC numbering system, the typical error message will look like '?Error 11 at line 1'. (You should keep a table of the Atari BASIC error messages in front of you.) Should one of these errors occur, the computer would simply inform you of it and place the cursor at the start of the troublesome line. You are then free to correct the problem and restart.

If you use parentheses to assign a variable, for example 'C = (A + B)/(A – B)', be sure there are as many right parentheses as left parentheses. Also remember that, unlike in algebra, multiplication here must be explicitly indicated with an asterisk. Use '10*B' and '5*(A + B)', not '10B' and '5(A + B)'. If you use a variable that has not been assigned in a previous line, its value is assummed to be zero. Division by an unassigned variable will result in a ?DIVISION BY ZERO ERROR.

More on ESC

So far we have used most of the ESC key's capabilities, though not all. Now, when we hit the ESC key we get a line asking us to hit the START, SELECT, or OPTION console key. As we have learned, START causes the screen expressions to execute, and SELECT allows us to select which screen we wish to edit.

Saving and Loading Screens

When you hit the OPTION button, the computer will display yet another line of information asking you to type an 'S', 'L', or 'E'. The 'E' is merely a failsafe enabling you to return to editing the screen. Typing an 'S' allows you to save your screen. As in some of the previous programs in this book, you will be asked for a filename and, as before, this filename is in the form of 'D:filename.ext' for disk users or 'C:' for cassette users. Both the comment field and the expression field are saved, so you will get back later exactly what you have typed. When MICRO Calc has saved your screen, it will redisplay the line asking for an 'S', 'L', or 'E'. At this point you can continue editing the screen, save the screen to another file (make a backup copy), or load another screen.

To Load in a screen simply specify the 'L' option. You will be asked for a filename the same way as when saving a screen. After the filename is specified and the return key is hit, the loading will commence. When MICRO Calc has loaded your screen, the newly loaded screen will be displayed and editing can begin in the upper left-hand corner of the expression field.

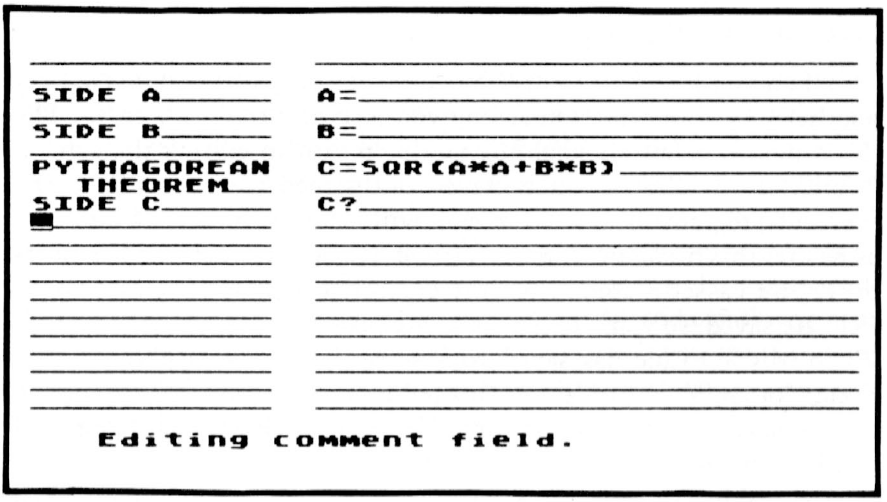
A Hint to the Efficient

We will give you a few pages of typical problems for MICRO Calc to solve. Saving each one as you type them in is fine as this allows you to keep the method for solving the problem for future use. But it is cumbersome to have to erase and retype all the numbers when you want to solve a different problem. Try saving the screens without the numbers typed in first, and then when you load them in later all you have to do is supply the numbers. For example, type in the averaging screen previously shown without any values for the variables A, B, or C; however, don't START it! Since the screen will not run without the numbers, you will have to supply them each time the screen is loaded in. This technique makes the screens more versatile for all conceivable situations.

More Examples

The Pythagorean Theorem

According to the Pythagorean Theorem, ''The hypotenuse of a right triangle is equal to the square root of the sum of the squares of the other two sides.'' Since this is, in effect, an expression just like any other, it can be handled by MICRO Calc. Enter the following lines, pressing RETURN after each line.



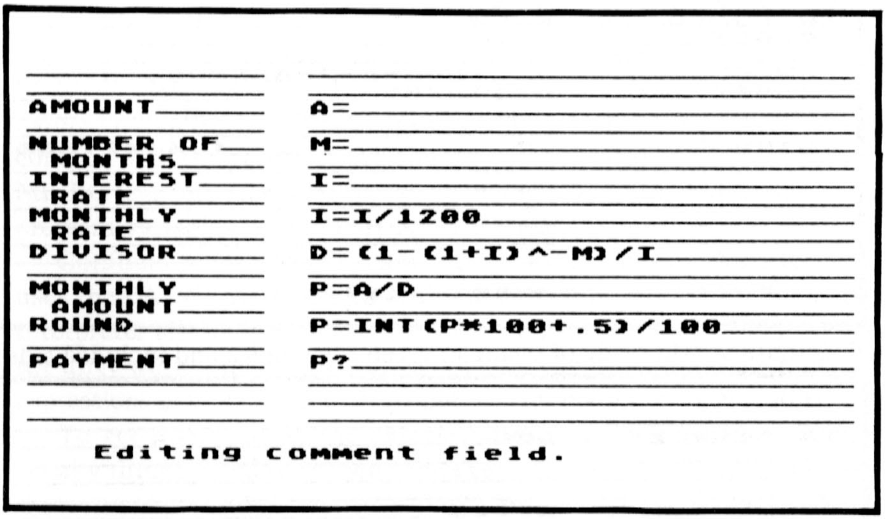
```
SIDE A              A=
SIDE B              B=
PYTHAGOREAN         C=SQR(A*A+B*B)
  THEOREM
SIDE C              C?

     Editing comment field.
```

Monthly Payments

You can calculate monthly payments of installment loans using the following formula:

$$D = \frac{1 - \frac{1}{(1+i)^m}}{i}$$

i is the interest rate per month and *m* is the number of months. The principal (the amount you are borrowing) is divided by *D* to get the monthly payment. Below is the screen you enter to perform these calculations with MICRO Calc.

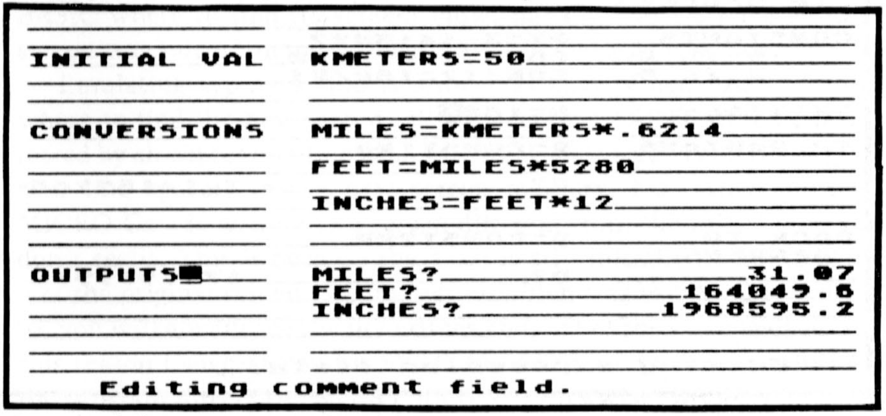


```
AMOUNT              A=
NUMBER OF           M=
  MONTHS
INTEREST            I=
  RATE
MONTHLY             I=I/1200
  RATE
DIVISOR             D=(1-(1+I)^-M)/I
MONTHLY             P=A/D
  AMOUNT
ROUND               P=INT(P*100+.5)/100
PAYMENT             P?

     Editing comment field.
```

Before you try it, SAVE the screen as described above. When it is saved, the program will return with the cursor where you left it. Enter values for A, M, and I, START it, and the result will appear on the eighth line. Now you can make a change (as you did with one of the first examples above) and see the change in the monthly payment.

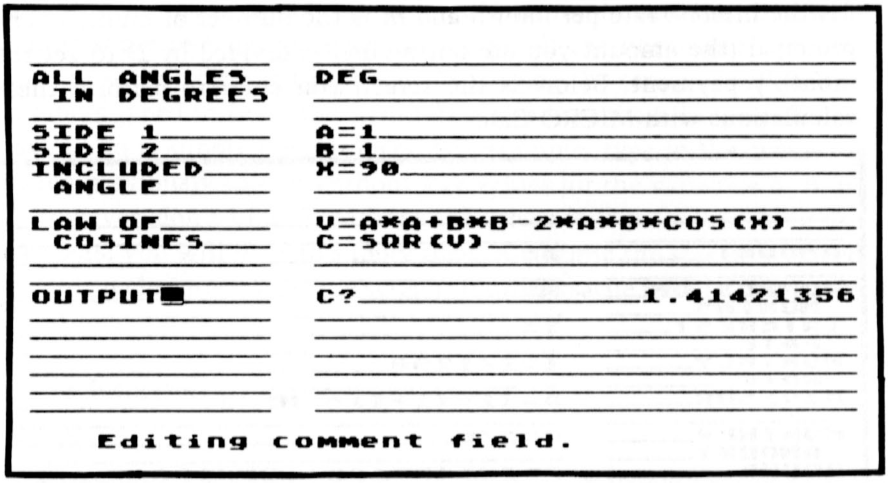
Once you understand MICRO Calc, you can use it to perform a wide variety of repetitive calculations. Be sure that you save the more elaborate screens, and you will develop a library of useful solutions to just about any problem.

Here are two graphics: one shows an empty screen and the other is a sample run. Enter your own data and try to calculate the answers.



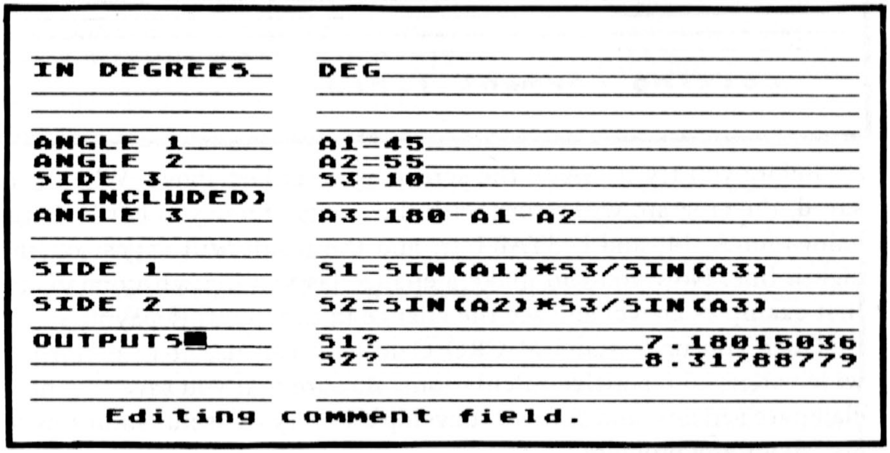
```
INITIAL VAL         KMETERS=50

CONVERSIONS         MILES=KMETERS*.6214
                    FEET=MILES*5280
                    INCHES=FEET*12

OUTPUTS█            MILES?____________31.07
                    FEET?___________164049.6
                    INCHES?________1968595.2

     Editing comment field.
```

Convert kilometers to miles, feet, and inches.



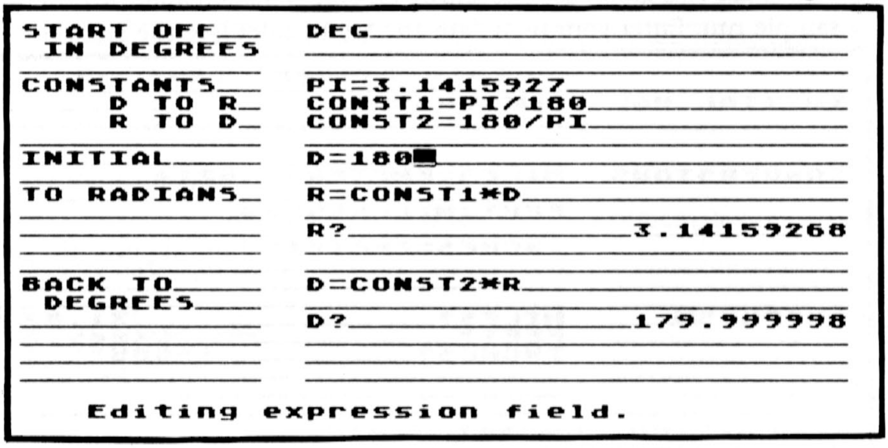
```
ALL ANGLES      DEG
 IN DEGREES

SIDE 1          A=1
SIDE 2          B=1
INCLUDED        X=90
 ANGLE

LAW OF          V=A*A+B*B-2*A*B*COS(X)
 COSINES        C=SQR(V)

OUTPUT█         C?                  1.41421356

     Editing comment field.
```

Calculate another side of triangle, given two angles and included side.



```
IN DEGREES      DEG


ANGLE 1         A1=45
ANGLE 2         A2=55
SIDE 3          S3=10
 (INCLUDED)
ANGLE 3         A3=180-A1-A2

SIDE 1          S1=SIN(A1)*S3/SIN(A3)

SIDE 2          S2=SIN(A2)*S3/SIN(A3)

OUTPUTS█        S1?                 7.18015036
                S2?                 8.31788779

     Editing comment field.
```

Calculate third side of triangle, given two sides and included angle.



```
START OFF       DEG
 IN DEGREES

CONSTANTS       PI=3.1415927
    D TO R      CONST1=PI/180
    R TO D      CONST2=180/PI

INITIAL         D=180█

TO RADIANS      R=CONST1*D

                R?                  3.14159268

BACK TO         D=CONST2*R
 DEGREES
                D?                  179.999998

     Editing expression field.
```

Convert angle from degrees to radians and back to degrees again.

How the Program Works

MICRO Calc is composed almost entirely of two major routines: a screen editor and an expression evaluator. The editor is constructed simply and requires no explanation beyond the program description that appears later. The expression evaluator, however, is the heart of the program and, therefore, requires some explaining.

If you take a look at the listing, you will notice that all the variable names end in '0'. This is so that the variables used in the screen do not conflict with the variables used by the program itself. The variables in MICRO Calc are ended with a '0' to insure that staying away from the program variables is no handicap to the user.

Conflicts with variable names is possible because MICRO Calc uses the BASIC interpreter to evaluate the expression lines. Using the interpreter allows the program to be much shorter and lets BASIC do the dirty work. Since BASIC already has a way to evaluate the numeric expressions in program lines, why not utilize this ability in MICRO Calc?

To get BASIC to do the expression evaluation, I had to do some sneaky things. I had to suspend the execution of MICRO Calc, making the computer think it was in BASIC's immediate mode, then enter a line, somehow 'hit' return over that line to enter it, and resume operation of MICRO Calc.

To make the computer act as if it were in immediate mode we have to duplicate the environment of immediate mode exactly. This means that we have to make another screen, type in a line, and hit return (somehow). I lowered the top of free memory by POKEing location 106 (see program description) with a smaller number than was in it. This allows us to have about 1K of protected storage that cannot be ruined accidentally by the computer. The bottom 960 bytes of this storage is the memory to hold our new screen. Then locations 88 and 89 are POKEd with the location of the new screen. These locations tell BASIC where to find its screen. Now, not only do we have another screen way out in memory, but BASIC also regards it as one.

Emulating typing a line is almost as easy. Once BASIC knows where to find the new screen, all we need is a PRINT statement followed by an expression line from MICRO Calc's screen and the expression line appears in the new screen memory. Note that although MICRO Calc is writing to a screen different from the initial one, you don't see it. This is because the Atari is still initially set up to *show* you the original screen. So, no matter which screen BASIC is writing to, you will always see the underlines and text of the screen editor. For this reason I suggest you hit SYSTEM RESET every time you BREAK the program during the START routine. Otherwise, BASIC may keep outputting to the new screen and everything could get irreparably and indescribably fouled up.

So far we have designed another screen to handle the tokenization of the expression line and have actually printed the expression to the screen. But how do we emulate the return key so that we can actually enter in the line? Luckily, there is a magic location (842) that can 'turn' the return key on and off. When this location holds the number 13, the return key is activated and returns occur everywhere. And whatever was written to the screen that happened to be 'returned' over, is entered as if you typed it in yourself. When this location is reset with the number 12, the returns are halted. Try it! Type 'POKE 842,13' at the BASIC 'READY' prompt and press return; the cursor should jump through everything below it (entering everything along the way) and continue down the screen, causing it to scroll repeatedly. Hit SYSTEM RESET to regain control.

Now all the program has to do is output the line, followed by a 'CONT' (continue statement) a few lines down, POSITION the cursor to the top of the screen, POKE location 842 with a 13, and then perform a STOP. The return key will enter everything in sight, beginning with the expression line. After it has entered the expression line (usually less than half a second) it then enters in the 'CONT' statement, and execution is continued at the line following the 'STOP'. When this is done, location 842 is reset to the value 12, and the computer redirects the pointer at locations 88 and 89 to point the initial (edited) screen to display the greater-than sign and START-time errors (if any).

If a question mark ('?') is discovered at the end of the expression line, the computer merely outputs that line (without the '?') preceeded by a 'A990'. This causes whatever that expression evaluates to be stored in variable A990. Then when execution is resumed at the line following the STOP, instead of immediately proceeding to the following line, MICRO Calc outputs the value of A990 right-justified (flush with the right-hand edge) on the same line as was requested.

Fortunately, handling errors is no problem. Setting a 'TRAP 1900' right before the outputting is done to the new screen causes the computer to go to line 1900 when it encounters an error. At this routine it outputs a similar-to-BASIC error message on the message line and then lets you in the editor — on the troubled line. The only problem with this is that syntax errors (errors where BASIC simply refuses to accept any of the line) are not TRAPable. Instead, look at the line following where the expression was entered (or attempted to be entered). If there is nothing there but blank spaces, then the line was entered correctly and no error occurred. If there is something there, then there is an error in syntax and execution goes to the special routine at line 1990.

MICRO Calc Variable Usage Table

Strings

F0$	Holds the Save/Load filename
M0$	Holds the machine-language block-move routine
ST0$	Storage string for the editing screen
TEMP0$	Temporary string
UL0$	Holds the underlines used in the editing screen
SPACE0$	Holds an entire screen line of spaces

Arrays

DX0,DY0	X and Y displacement for each of the four arrow key combinations
TBL0	Holds a value telling the printable status of each character
QM0	Holds the position of the question mark within each line

Numerics

A990	The variable to which the value of a 'question mark' expression line is assigned so that the value can be printed out.
C0	Holds the characters typed in during all editing and single-character input requests
CON0	Value of the console button location
DX0,DY0	Variables used to read in the value for the DX0 and DY0 arrays
L0,M0	Low byte and high byte of the starting address of the hidden screen memory
LL0	Line length of the currently enabled editing field
MAX0	Maximum screen position of the end of the editing field's lines
P0	Position within ST0$ of any particular character.
START0	The starting address of the hidden screen memory
Q0,QQ0	Loop variables
QL0,QM0	High and low byte of the current location of the normal screen

TY0	The type of the character just input, as derived from the TBL array
U0	Dummy variable used for USR statements
X0,Y0	X and Y coordinates of the cursor during screen editing

Program Description

Line 9 lowers the BASIC top-of-RAM pointer by 1024 bytes to allow a new screen to be protected in high memory. This POKE effectively drops the number of free bytes by 1K. The new screen is needed to allow MICRO Calc to use the BASIC interpreter without seeing it. Next, the computer is put into graphics mode 0 (text mode). This graphics statement has to appear after the POKE statement so that the new text screen can adjust itself to the new memory constraint. (For more information, see the description for lines 1000-1990.)

Line 10 dimensions the various strings and arrays used in MICRO Calc. ST0$ is most important as it is the storage area for the edited screen.

Line 20 calls the initialization routine at line 30000. When the routine is finished, execution then begins at line 300. (See the description for line 30000 for details on variable usage.)

Line 100 GETs a character from the keyboard in ASCII. TY0 is assigned a value from 0 to 7, depending on its ASCII code. If the value is a 0, then the character received is not allowed and the program goes back to 100.

There is a small routine to handle each of the seven classes of characters. Line 110 executes the appropriate routine according to the variable TY0.

Line 120 displays and stores a normal character. When it is done, as with all these character routines, control is passed back to line 100 so that another character can be input.

Lines 130 to 138 handle the arrow keys and the resulting cursor movement.

Line 140 handles the DELETE key.

Line 150 handles the RETURN key.

Lines 160 and 162 handle the question mark.

Lines 170 to 174 clear the screen. Since typing the CLEAR key is altogether too easy to do, the program prompts you to make sure you want to clear the screen. If you type 'Y', the screen will clear. If you type any other character, the screen does not change. When the screen is indeed to be erased, the storage area (ST0$) is filled with blanks.

Lines 200 to 226 display a one-line menu that appears on line 22 when the ESC key is pressed. If START, SELECT, or OPTION is pressed, control is passed to lines 1000, 250, or 2000, respectively.

Lines 250 and 260 switch the active editing fields.

Lines 300 to 320 display at line 22 which editing screen is active.

Lines 1000 to 1990 are the START routine.

Line 1000 saves the location of the normal screen in variables QL0 and QM0.

Line 1100 sets up a loop to point to the starting character of each of the 20 lines stored in ST0$.

Line 1102 displays the greater-than sign at the currently executing line.

Line 1103: If the line is blank the program proceeds to the next line.

Lines 1104 to 1106 clear from the end of the expression to the end of the line so the printed results don't remain forever.

Line 1108 sets up the new screen and clears it. Clearing is done by filling the memory that holds it with some 960 zeros. This is accomplished by using the move routine to move 959 bytes from the top location (previously POKEd with a 0) to the location just below it. The net effect is that the 0 is propagated throughout the 960 bytes.

Line 1110: If the expression ends with s '', then GOTO value-printing routine.

Lines 1120 to 1150 handle the rest of the expression tokenization routine. The line is output and entered using the technique discussed in the section titled ''How the Program Works.''

Lines 1200 to 1290 take care of outputting the value of the expression line.

Line 1300 deletes the remaining greater-than sign on the screen, re-enables the cursor display, and goes to line 300.

Lines 1900 to 1910 hold the error-handling routines (except the syntax error).

Lines 1950 and 1960 re-enable the old screen and set up the new screen, respectively.

Line 1990 takes care of the syntax errors.

Lines 2000 to 2060 are the screen-save and load-menu options.

Lines 2100 to 2180 load in a screen.

Lines 2200 to 2220 save a screen.

Lines 2900 to 2910 handle the possible errors in saving and loading.

Lines 30000 to 30090 initialize the program variables and memory:

Line 30010 initializes the starting location of the new screen.

Lines 30010 to 30028 set up a table with the numbers to describe each character to the editor. The table holds a number from 0 to 7, where 0 is a non-displayable character. The ASCII value of the character received from the keyoard is used as an index to this array so that the numbers actually pair up to the ASCII values. That is, TBL0(155) is a 6 (155 is the ASCII code for the RETURN key), which signals the editor to GOTO the RETURN key handler, just as TBL0(65) is a 1 (65 is the ASCII code for the letter 'A'), which tells the editor to print the character normally.

Line 30030 sets up the initial values for the editing field.

Line 30040 reads in the direction displacements for each of the arrow keys. The ASCII code of the arrow keys (28-31) is used as an index to the arrays DXJ0( ) and DY0( ) once 28 has been subtracted from it. The displacements are either a 1, 0, or negative 1, depending upon the direction.

Lines 30050 to 30054 set up the three strings used by the program.

Line 30060 puts the initial values into the question mark array QM0( ). These values later hold the position within the storage string (ST0$) of the question mark (if any). In this way, during the START routine, a quick check can be made to see if the lines indeed hold a ''.

Line 30080 OPENs the Atari keyboard for input. This is so the characters typed can be input one at a time.

Line 30088 sets up the screen initially with the underlines, etc.

Line 30090 POSITIONs the cursor at the current X0(X) and Y0(Y) coordinates on screen, then performs a RETURN.

Lines 32000 to 32010 hold the data for the arrow displacements. Each arrow (beginning with CHR$(28), the up arrow) needs a pair of these numbers.

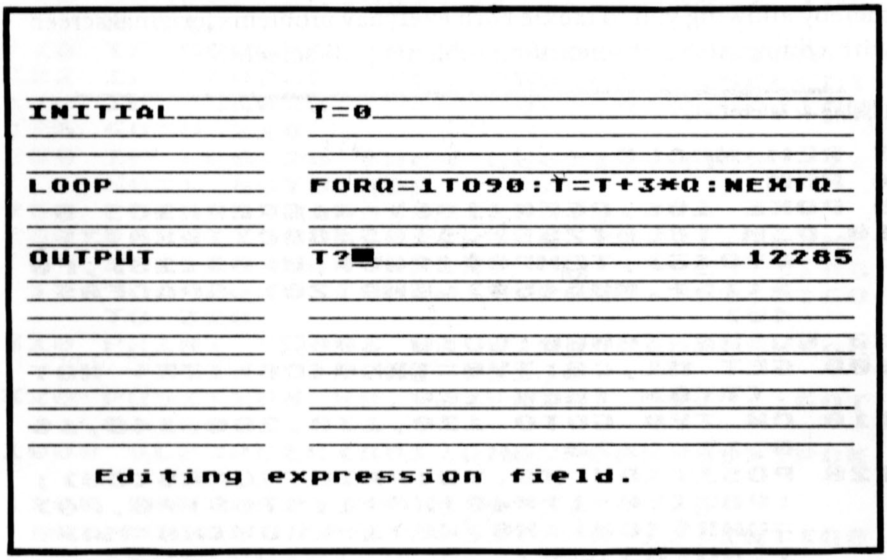
Lines 32100 to 32110 hold the machine-language block-move routine (used in previous programs) in string form. (See the section entitled ''Scrolling the Graphics Screen'' in the MASTER chapter of this book.)

Altering MICRO Calc

With just a little work MICRO Calc can be made to handle strings as well as BASIC numbers. There are only a few things that need to be changed. First, the method of assigning a string variable a value in itself works without any modifications due to the way MICRO Calc evaluates the lines. But we are not so lucky with the rest of the program. The value-printing routine would have to be modified to print strings. All you need to do is to check the screen line for a '$' after the variable name and right before the '?' and assign that line to be the value of 'A990$' instead of 'A990'. Then, all you have to do is output the string at the end of the line, using the same right-hand justification technique used for outputting the numbers.

There are a few aspects about using strings with MICRO Calc that are not so rosy. All strings on the Atari have to be previously dimensioned to a certain character length *via* the DIM statement. This means that they must be dimensioned either at the start of the MICRO Calc program itself (at line 10 perhaps) or on the screen. If you do it at the beginning of the program, you will be able to specify only a limited number of strings (because of memory limitations). For example, only the string variables A$, B$, and C$ would be allowed. If you are going to specify the DIM statement at the top of the screen, then you must be sure that after the initial STARTing of the screen, the DIM statements are no longer there. No matter what you try, short of clearing all the variables with a CLR statement (don't try this as it will clear MICRO Calc's variables as well), you can only dimension a string variable once. This may be more trouble than it is worth and it is for this reason that string handling is almost entirely omitted from MICRO Calc.

There are a few things that you can do with MICRO Calc that are not transparent at first. At the start of the program itself you might define a few constants such as PI = 3.14159265, AVAGNUM = 6.02 E23, MYWEIGHT = 450, or whatever you find necessary. Then you can use the variables in any screen you wish without having to predefine them. If you don't want to alter the BASIC code, you can create a screen file that is nothing but constant definitions. You can Load it in, START it up, and then erase the screen for your other screen work; the values will still be there for the following screens. Either way you can build a whole library of values for future efficiency.



```
INITIAL________   T=0___________________________

LOOP___________   FORQ=1TO90:T=T+3*Q:NEXTQ_

OUTPUT_________   T?█_____________________12285


      Editing expression field.
```

MICRO Calc even has looping capability similar to that in BASIC, provided the entire loop takes place on a single screen line. This means that the lines will be a little tight, but there is enough room for some FOR...NEXT loops. Clear the screen and type the short screen above that calculates the sum of the first 90 multiples of 3. Note that spaces were removed to conserve space in the line.

If you find that the 25 characters per expression line is not enough, you can change the line length to be any size (up to 40 characters). First you must change the screen initialization routines (so that the underlines are displayed.

Next, the values of MAX and LL must be changed to fit the new editing dimensions. MAX is the number of the last position on the screen (probably 38 or 39) and LL is the line length (probably 38 or 40). This enables the editor to wrap things around on the screen correctly.

If you no longer wish to have a comment field, you must disable the SELECT key by changing the line number from 250 to 300. Otherwise, changing the MAX and LL variables in the lines that handle the editing-field switching will be enough.

Next the START routine must be modified so that it checks, outputs, and keeps track of the different lines. This is complicated since it requires altering not only the print statements to the hidden screen (see the program descriptions) but also other less obvious things, such as positioning the greater-than sign (>) to the left of the executed expression line.

When you have finished the alterations, save the new program so that in the future you have a choice between the two MICRO Calcs, thereby allowing you to tackle both everyday problems (original screen with comments) and scientific problems (full screen).

***Listing 1:* MicroCalc**

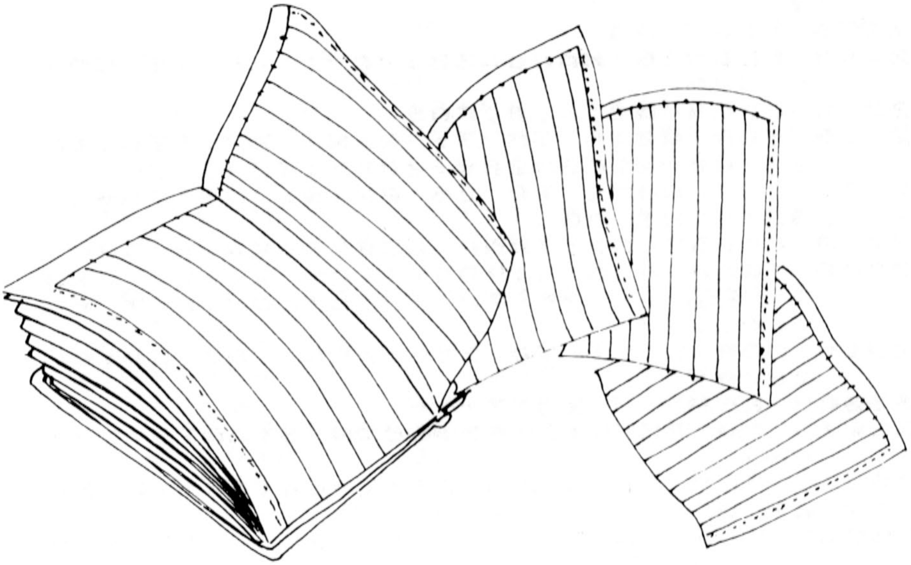
```
3 REM MCALC
4 REM
9 POKE 106,PEEK(106)-4:GRAPHICS 0
10 DIM TBL0(255),ST0$(800),DX0(3),
    DY0(3),TEMP0$(200),UL0$(25),F0
    $(15),M0$(54),QM0(20),SPACE0$(
    40)
20 GOSUB 30000:GOTO 300
100 GET #1,C0:TY0=TBL0(C0):IF  NOT
     (TY0) THEN 100
110 ON TY0 GOTO 120,130,200,140,16
    0,150,170
120 POSITION X0,Y0:PRINT CHR$(C0);
    :P0=(Y0-1)*40+X0+1:ST0$(P0,P0)
    =CHR$(C0):X0=X0+1-LL0*(X0=MAX0
    ):GOTO 100
```

```
130 X0=X0+DX0(C0-28):Y0=Y0+DY0(C0-
    28):IF X0<MAX0-LL0+1 THEN X0=M
    AX0
132 IF X0>MAX0 THEN X0=MAX0-LL0+1
134 IF Y0<1 THEN Y0=20
136 IF Y0>20 THEN Y0=1
138 POSITION X0,Y0:PRINT "→←";:GOT
    O 100
140 X0=X0-1+LL0*(X0=MAX0-LL0+1):PO
    SITION X0,Y0:PRINT "_←";:P0=(Y
    0-1)*40+X0+1:ST0$(P0,P0)=" ":G
    OTO 100
150 X0=MAX0-LL0+1:Y0=Y0+1-20*(Y0=2
    0):POSITION X0,Y0:PRINT "→←";:
    GOTO 100
160 POSITION X0,Y0:PRINT "?";:X0=X
    0+1-LL0*(X0=MAX0):IF X0<>MAX0
    THEN PRINT UL0$(1,MAX0-X0+1);
162 P0=(Y0-1)*40+X0:ST0$(P0,P0)="?
    ":QM0(Y0)=P0:POSITION X0,Y0:PR
    INT "→←";:GOTO 100
170 POSITION 0,22:PRINT "⇧  Clea
    r screen: Are you sure...?";:G
    ET #1,C0:IF CHR$(C0)<>"Y" THEN
     300
172 FOR Q0=1 TO 20:POSITION 1,Q0:P
    RINT UL0$(1,11);"  ";UL0$(1,25
    ):NEXT Q0
174 ST0$=" ":ST0$(800)=" ":ST0$(2)
    =ST0$:GOTO 300
200 POKE 752,1:POSITION 0,22:PRINT
     "⇧    ..SELECT, OPTION, or ST
    ART ..";:POKE 752,0
210 CON0=PEEK(53279):IF CON0=7 THE
    N 210
220 IF CON0=6 THEN 1000:REM START
222 IF CON0=5 THEN 250:REM .SELECT
224 IF CON0=3 THEN 2000:REM OPTION
226 GOTO 210
250 IF LL0=25 THEN LL0=11:MAX0=11:
    X0=1:GOTO 300
260 LL0=25:MAX0=38:X0=14
300 POSITION 0,22:POKE 752,1:IF MA
    X0=11 THEN PRINT "⇧   Editing
     comment field.":POKE 752,0:GO
    TO 320
310 PRINT "⇧    Editing expression
     field.":POKE 752,0
320 POSITION X0,Y0:PRINT "→←";:GOT
    O 100
1000 QL0=PEEK(88):QM0=PEEK(89):POK
    E 752,1
1100 FOR Q0=1 TO 800 STEP 40
1102 GOSUB 1950:POSITION 13,INT(Q0
    /40):PRINT " ↓←>↓← ";
```

```
1103 IF ST0$(Q0+14,Q0+38)=SPACE0$(
     1,25) THEN POSITION 14,INT(Q0/
     40)+1:PRINT UL0$(1,25):GOTO 12
     90
1104 FOR QQ0=38 TO 14 STEP -1:IF S
     T0$(Q0+QQ0,Q0+QQ0)=" " THEN NE
     XT QQ0:GOTO 1108
1105 IF QQ0=38 THEN 1108
1106 POSITION QQ0+1,INT(Q0/40)+1:P
     RINT UL0$(1,38-QQ0);
1108 GOSUB 1960:POKE START0,0:U0=U
     SR(ADR(M0$),START0,START0+1,95
     9)
1110 P0=QM0(INT(Q0/40)+1):IF ST0$(
     P0,P0)="?" THEN 1200
1120 POSITION 0,4:PRINT ST0$(Q0+14
     ,Q0+38):? :? :PRINT "CONT"
1130 POSITION 0,0:TRAP 1900:POKE 8
     42,13:STOP
1140 POKE 842,12:IF PEEK(START0+20
     0)<>0 THEN 1990
1150 GOTO 1290
1200 POSITION 0,4:PRINT "A990=";ST
     0$(Q0+14,QM0(INT(Q0/40)+1)-1):
     ? :? :PRINT "CONT"
1210 POSITION 0,0:TRAP 1900:POKE 8
     42,13:STOP
1215 POKE 842,12:IF PEEK(START0+20
     0)<>0 THEN 1990
1220 GOSUB 1950:POSITION 39-LEN(ST
     R$(A990)),INT(Q0/40)+1:PRINT A
     990;:GOSUB 1960
1290 NEXT Q0:POKE 752,0
1300 GOSUB 1950:POSITION 13,INT(Q0
     /40):PRINT NT " ";:POKE 752,0:
     GOT O 300
1900 GOSUB 1950
1902 POKE 842,12:POSITION 0,22:PRI
     NT "↑↖     ?Error ";PEEK(195);"
      at line ";INT(Q0/40)+1;".";:P
     OKE 752,0
1910 X0=MAX0-LL0+1:Y0=INT(Q0/40)+1
     :POSITION X0,Y0:PRINT "→←";:GO
     TO 100
1950 POKE 88,QL0:POKE 89,QM0:RETUR
     N
1960 POKE 88,L0:POKE 89,M0:RETURN
1990 GOSUB 1950:POSITION 0,22:PRIN
     T "↑↖    ?Syntax Error at line
      ";INT(Q0/40)+1;".";:POKE 752,
     0:GOTO 1910
2000 POKE 752,1:POSITION 0,22:PRIN
     T "↑..(S) to save, (L) to load
      (E) to edit..";:POKE 752,0
```

```
2010 GET #1,C0:IF CHR$(C0)<>"S" AN
     D CHR$(C0)<>"L" AND CHR$(C0)<>
     "E" THEN 2010
2030 IF CHR$(C0)="E" THEN 300
2050 POSITION 0,22:PRINT "⇧   Ent
     er filename...";:INPUT F0$:TRA
     P 2900
2060 IF CHR$(C0)="S" THEN 2200
2080 OPEN #2,4,0,F0$
2100 ST0$=" ":ST0$(800)=" ":ST0$(2
     )=ST0$:ST0$=""
2120 FOR Q0=1 TO 4:INPUT #2,TEMP0$
     :ST0$(LEN(ST0$)+1)=TEMP0$:NEXT
      Q0
2122 FOR Q0=1 TO 20:INPUT #2,QM0:Q
     M0(Q0)=QM0:NEXT Q0
2130 CLOSE #2:TRAP 65535
2150 FOR Q0=1 TO 800 STEP 40
2152 POSITION 1,INT(Q0/40)+1:FOR Q
     Q0=11 TO 1 STEP -1:IF ST0$(Q0+
     QQ0,Q0+QQ0)=" " THEN NEXT QQ0:
     ? UL0$(1,11);:GOTO 2160
2154 PRINT ST0$(Q0+1,Q0+QQ0);:IF Q
     Q0<11 THEN PRINT UL0$(1,11-QQ0
     );
2160 POSITION 14,INT(Q0/40)+1:FOR
     QQ0=38 TO 14 STEP -1:IF ST0$(Q
     0+QQ0,Q0+QQ0)=" " THEN NEXT QQ
     0:? UL0$(1,25);:GOTO 2170
2162 PRINT ST0$(Q0+14,Q0+QQ0);:IF
     QQ0<38 THEN PRINT UL0$(1,38-QQ
     0)
2164 IF Q0/40=INT(Q0/40) THEN PRIN
     T
2170 NEXT Q0
2180 LL0=25:MAX0=38:Y0=1:X0=14:GOT
     O 300
2200 OPEN #2,8,0,F0$
2210 FOR Q0=0 TO 3:PRINT #2;ST0$(Q
     0*200+1,Q0*200+200):NEXT Q0
2212 FOR Q0=1 TO 20:PRINT #2;QM0(Q
     0):NEXT Q0
2220 CLOSE #2:TRAP 65535:GOTO 300
2900 TRAP 65535:POSITION 0,22:PRIN
     T "⇧◪   ?File input/output er
     ror..."
2910 FOR Q0=1 TO 200:NEXT Q0:GOTO
     2000
30000 REM ---INIT---
30010 M0=PEEK(106):L0=0:START0=256
     *M0
30020 FOR Q0=0 TO 26:TBL0(Q0)=0:NE
     XT Q0
30022 FOR Q0=32 TO 94:TBL0(Q0)=1:N
     EXT Q0
```

```
30024 FOR Q0=95 TO 255:TBL0(Q0)=0:
    NEXT Q0
30026 FOR Q0=28 TO 31:TBL0(Q0)=2:N
    EXT Q0
30028 TBL0(27)=3:TBL0(126)=4:TBL0(
    63)=5:TBL0(155)=6:TBL0(125)=7
30030 LL0=25:MAX0=38:Y0=1:X0=14
30040 FOR Q0=0 TO 3:READ DX0,DY0:D
    X0(Q0)=DX0:DY0(Q0)=DY0:NEXT Q0
30042 READ M0$:POKE START0,0:U0=US
    R(ADR(M0$),START0,START0+1,959
    )
30050 UL0$="______________________
    ____"
30052 ST0$=" ":ST0$(800)=" ":ST0$(
    2)=ST0$
30054 SPACE0$="
                                 "
30060 FOR Q0=1 TO 20:QM0(Q0)=(Q0-1
    )*40+30:NEXT Q0
30080 OPEN #1,4,0,"K:"
30088 POKE 82,0:POKE 752,1:FOR Q0=
    1 TO 20:POSITION 0,Q0:PRINT "
    ";UL0$(1,11);" ";UL0$:NEXT Q0
30090 POKE 752,0:POSITION X0,Y0:?
    "→←";:RETURN
32000 REM ---ARROW DISPLACEMENTS--
32010 DATA 0,-1,0,1,-1,0,1,0
32100 REM ---6502 MOVE (FILL)---
32110 DATA hh▗Lh▗Kh▗Nh▗Mh▗Ph▗0 ♥1K
    ┌MfKP fL fMP fNFO%OI)P FP%O┐PP◆
    ◆
```



Using the Diskette

Boot the disk operating system from your **Atari System Disk** and then insert the **Mastering Your Atari disk.**

Type **RUN "D:PROGRAM [return] t**o run any desired program. Type the name exactly as given in the table below, and do not forget the **"D:** which must precede the program name.

Press **SYSTEM RESET** when you are done with one program before you RUN another.

Diskette Directory

Name	**Description**
PLAYER	Music playing program
SONG	Sample song for PLAYER - Do Not RUN it
MASTER	A Simple Guessing Game
WORD	A Word Guessing Game
ANSWER	An Intelligent Computer
BREAKUP	An Animated Game
CLOCK	A Digital Time Display
CSDUMP.PC	Display of Atari Character Set
FILLIN.PC	Demonstration of Character Set
ANIMATE.PC	Demonstration of Animated Characters
MCOLOR.PC	Demonstration of Multi-Color Characters
BARSORT	Comparison of Five Sorting Techniques
PHONE	Create and Sort by Name and Number
MCALC	Micro Calculator Program
MSAMPLE	Sample Screen for Micro Calc - Do not RUN it.

Note that **SONG** and **MSAMPLE** are data files used by **PLAYER** and **MCALC** and will not **LOAD** or **RUN**.

Your **Mastering Your Atari** disk is **WRITE protected**. Before using **PLAYER, PHONE** or **MCALC**, which permit you to create data files on disk, you should copy these programs to another disk that can then be run **Unprotected**.

COMPUTERS

Written and produced by the editors and programmers of MICRO magazine, this new book is your guide to complete mastery of Atari BASIC. Learn quickly how to write your own programs, modify them, and add a wide assortment of features—all on your own.

You'll learn by doing. Each of the eight BASIC projects listed below begins with a complete running program . . . a practical utility or highly entertaining and challenging game. Then, detailed, easy-to-follow instructions show you how to add features. You can actually create your own software for Atari!

Eight Basic Projects . . . Dozens of Programs

MICRO CALC—a miniature spreadsheet program that makes complex, repetitive calculations a breeze. Can be used to experiment with BASIC functions.

MASTER—a guessing game for one or two players. Teaches programming with random numbers and flags.

ATARI CLOCK—a deluxe clock that displays time in giant-size numerals on the Atari screen. Teaches ON. . .GOSUB function and character graphics.

WORD DETECTIVE—lets you discover the computer's rule for accepting or rejecting the words you type. Also teaches string manipulation functions.

ATARI PLAYER—shows you how to play musical tunes using the Atari's keyboard like an organ. Teaches you how to LOAD or SAVE files, how to program music, and how to use cassette data files.

SORTING—a "bubble" sort is demonstrated in two programs—one uses colorful character graphics, the other sorts a telephone directory. Other sorting methods are also described.

BREAKUP—use a paddle, a joystick, or the keyboard to play this exciting bouncing-ball game and learn how animation is accomplished with PEEKs and POKEs to screen memory.

PROGRAMMABLE CHARACTERS—learn how to redefine part of the Atari's character set to add extra plotting resolution while retaining most normal characters.

FOR ATARI 400, 800, AND XL SERIES

Cover design by Mike Freeland

PRENTICE-HALL, Inc.,
Englewood Cliffs, New Jersey 07632

0 21898 55955 5

3

ISBN 0-13-559550-9