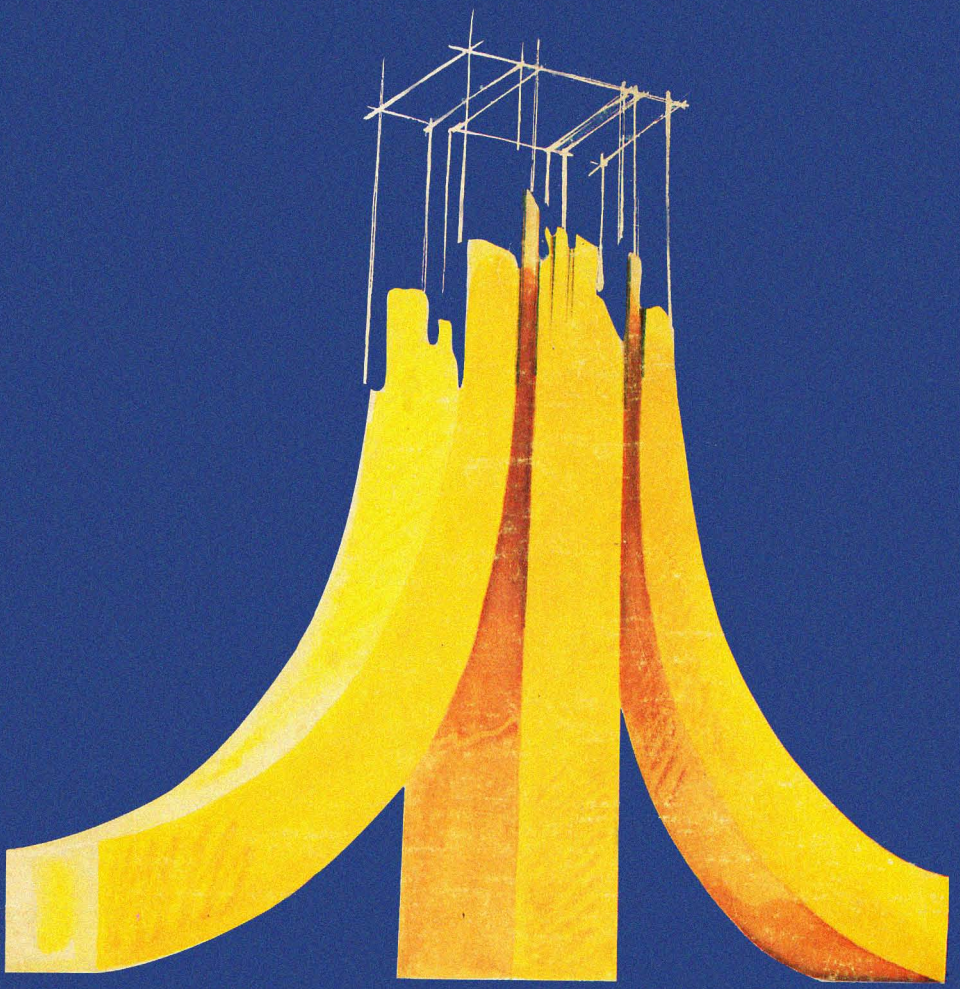
COMPUTE!'s FIRST BOOK OF ATARI®



From The Editors of COMPUTE! Magazine

COMPUTE!'s FIRST BOOK OF ATARI®

Published by **COMPUTE!** Books,
A Division of Small System Services, Inc.,
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-00-0

10 9 8 7 6 5 4 3 2 1

Table of Contents

INTRODUCTION

Robert Lock, Editor/Publisher, **COMPUTE!** Magazine

In the fall of 1979, **COMPUTE!** Magazine began with the initial vision of providing a resource and applications magazine to owners and users of various personal computers. We made the decision, at that time, to support the new personal computers from Atari, Inc.

Our first "Atari Gazette," a monthly part of **COMPUTE!**, was a total of three pages long . . . Frequently we struggled, during those early issues, to seek out good editorial support. Now, every issue of **COMPUTE!** routinely carries 40-50 pages of material for the Atari. And we're still maintaining the same standards of quality. The Atari reader base is growing faster than ever, and we've never doubted our decision to support it.

At the time of this writing, mid-November, 1981, Atari, Inc. is shipping more personal computers each month, than they did in all of 1980!

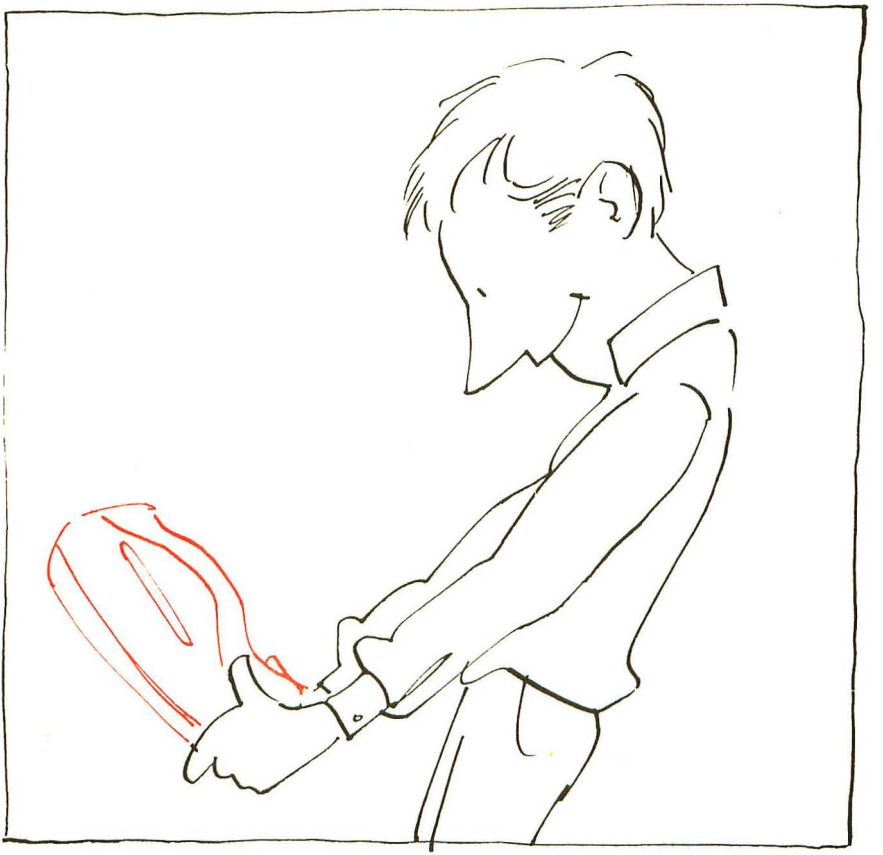
On the pages which follow, you'll find some of the best of the ATARI Personal Computer® material to appear in **COMPUTE!** Magazine during the year 1980.

We've organized the material and designed the book so that it will be easy to use. If you have any comments or suggestions regarding this book, or future books you'd like to see from us, please let us know.

Our special thanks to Charles Brannon and Richard Mansfield of the Editorial staff at **COMPUTE!**; Kate Taylor, Dai Rees, and De Potter of the Production staff; Georgia Papadopoulos, Art Director; and Harry Blair, our illustrator.

COMPUTE! Books is a division of Small System Services, Inc.,
publishers of **COMPUTE!** Magazine.
Editorial offices are located at
625 Fulton Street, Greensboro, NC 27403 USA. (919) 275-9809.

CHAPTER ONE: Getting To Know Your Atari



Atari's Marketing Vice President Profiles the Personal Computer Market

Michael S. Tomczyk

Atari's corporate character and projected company goals. The inside word.

Conrad Jutson

Atari doesn't especially like my nickname for their 400/800 personal computer — "the pop-top computer" — but it's a fact the computer has a "pop top" where the plug-in RAM/ROM cartridges fit, part of their innovative user-proof system which also includes interchangeable cards for the computer's various peripherals. Atari also has a growing array of educational and game software, including the most sophisticated real-time simulation game (STAR RAIDERS) in the galaxy . . . a long way from "Pong," the game that started it all.

Atari's competitors in the personal computer market chuckle at what they see as the company's attempt to develop the "home" computer market, in the face of extensive market research that says the home market won't "happen" for another 4-5 years. Does that mean Atari is wasting its resources? Are they really going after the home market? Or are they laying the groundwork for a broader marketing program?

To answer some of these questions, I interviewed Atari's new Vice President-Sales & Marketing for Personal Computers — he's Conrad Jutson, who came to Atari in November 1979 with a scant background in computers but over 20 years experience in consumer electronics at G. E. (12 yrs.), Toshiba (6 yrs.) and Texas Instruments (3 yrs.).

Jutson began by describing what he sees as the outlook for the personal computer market: "Small business in the short run will account for fifty percent of the personal computer business, dollar wise," he predicted, defining small businesses as those with less than $1 million in annual gross revenues, employing 10-15 people, and usually involved in manufacturing or a service-oriented industry. Typically, they do their bookkeeping by hand through a full or part time employee, or have it done by a local service. The key to

reaching this market, Jutson explained, is being able to show them that a microcomputer will increase their productivity and make the investment worthwhile.

The second broad market segment is the **consumer market** which, he said, consists of hundreds of subsets.

"If we were to profile the personal computer buyer in the early 80's, it would be a male or female head of household, most likely in a managerial, administrative or professional position, typically earning over $25,000 per year and falling into the 25 to 50 age bracket. Most likely, this person is already familiar with what a computer can do and can, in the home environment, identify a need for computing to address various problems and functions.

"There are several millions of these households in the U. S. that fit into the demographics I've described," he continued. "I don't believe personal computers will ever be an 'impulse item' off the shelf, partly because of the expense. So the logical question becomes, 'Why should I buy a personal computer and what will it do for me?' "

Jutson's answer to that question — what will a computer do for me — provided an interesting way of categorizing the personal computer market in terms of **function.** His list of personal computer uses included . . .

1) Planning and Record Keeping:

"I believe this type of managerial/administrative consumer does not pay enough attention to his own finances — this is confirmed by the rapid growth of financial-planning services. With the rapid inflation of the past few years, projected to continue through the 1980's, many consumers have found themselves in higher tax brackets with a higher cost of living that has made their lives more and more complex and difficult to manage. They've had to cope with budget planning, financial investments, mortgages, loan payments, credit unions, payroll stock plans, taxes, and pensions. In this new, complex environment, consumers have to organize their home record systems like they do at work — on a daily, year-round basis instead of just once a year at tax time. They have to look at their gross income, their investment tradeoffs, and I believe this type of consumer can justify the purchase of a personal computer with the appropriate software to meet these various needs . . . given that the typical first purchase of a personal computer is around $2000-$2200."

2) Home Education:

The next category of purchase that adds value to the computer is

home education. Jutson noted that a majority of schools and colleges are requiring some hands-on computer experience and more and more schools are bring computers into the classroom as instructional aids. There is already an enormous investment in home education being made by the American family — cutting across all demographic strata — in home courseware from encyclopedias to books. As a supplement to classroom education, this home courseware can be made much more exciting and "fun" through visual display and interaction with a computer, Jutson explained.

3) Personal Development & Interest:
There is also, he said, a huge market in how-to-books, all the way from how to fix your appliances to learning foreign languages. Literally hundreds of topics are addressed. Personal computers provide for active hands-on demonstration for all age brackets and interests, and speed the learning process.

4) Interactive Entertainment:
Having purchased a personal computer, we're all challenged by interactive entertainment, he said, whether the entertainment is one of skill or of strategy. The sale of strategic board games (chess, backgammon) never seems to let up and, in the skill area, the video arcades are doing extremely well. So entertainment accounts for a good deal of software sales.

5) Home Information/Communications:
If we move away from computation and hook up an interface and telephone modem, we have the capability to hook up to a timesharing service. Using the computer as a terminal provides a capability for dialing up and subscribing to a variety of evolving services. Some, like Micronet and The Source already have a fairly long menu. Atari has defined an informatin and communications strategy — obviously it will leverage our installed base of hardware to help our users gain access and may involve a wholly owned subsidiary like Warner Amex Cable. Some of the future uses of this home information system which we can envision include news, stock data and other services which will cut down driving time, mailing time, and minimize the hassle of shopping and bill paying. It's a question now of "getting the players together," he said, and making it happen.

6) Home Monitor & Control:
The decade of the 1980's will witness a growth of consumer electronic products deriving in large part from introduction of smart

electronics into the home. The personal computer is the "leading edge" of these products. By the mid-1980's, he expects to see dedicated smart electronics — CPU devices which interact with the electronic environment — in the home. It's unlikely that we'll see one massive all-purpose CPU controlling everything in the home. It will happen step by step, beginning with stand alone appliances containing their own microprocessors and other smart electronics. These, then, are some of the major uses which Jutson foresees for personal computers.

He goes on to say that the Atari product was designed to be easy to use by consumers, easy to access, easily loaded (cartridges), and easily connected (modular cords).

"Does the end user care about the architecture of the machine?" he asked rhetorically. "The answer is no. 'What will it do for me?' That's his major concern. We in the consumer electronics business are concerned with leveraging technology and bringing that technology to the consumer for his or her benefit, so why try to scare the consumer off by making it so he or she has to have a double E or be a computer programmer to utilize the full capabilities of a personal computer?"

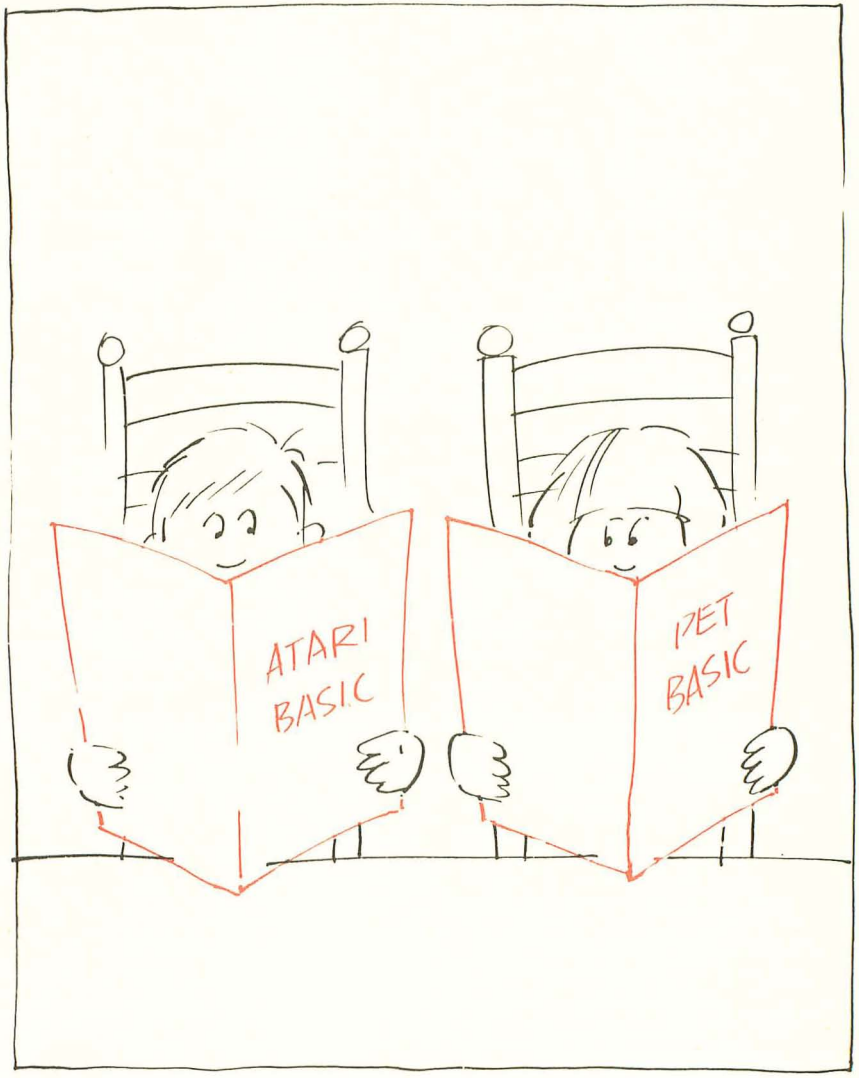
He drew a parallel between the personal computer industry and the home stereo industry, pointing out that 15 years ago there were 1500 hi-fi salons in the United States and now there are about 15,000 outlets in the U. S. He feels that computer stores will become to the computer market what hi-fi specialty shops were originally to the hi-fi industry, and predicted that a number of stores will proliferate and become strong chains. A parallel development, he said, is the entry of general merchandisers such as J. C. Penney Department Stores into the personal computer distribution scheme.

He emphasized that Atari only started shipping late in the fourth quarter of 1979 and is just getting into the market with its 400/800 computers. Heavy advertising is planned for the second and third quarters of 1980, including a full dealer support program.

"Having just come out of the gate we have to and will continue to have, a lot of things to do to strengthen our position in the industry," he said. "Atari is a young company that has already, in a few years, achieved significant growth in consumer electronics products. We have a vertically integrated manufacturing capability, a marketing staff that understands marketing, distribution, sales, and sales promotion; and a large blend of research and development and engineering expertise.

"We believe that the Atari computers are different because

from word one they were developed to take away whatever apprehensions a first time user might have and help him or her feel good about interfacing with our product. With Atari computers, you don't have to stop and think before you use them. Of course, more and more of the younger generation are learning to program and work with more sophisticated applications, and they will have the capability of doing so with our product."



Atari BASIC and PET Microsoft BASIC. A BASIC Comparison.

Joretta Klepfer

An important item to consider when shopping for a computer is the language that you will use to communicate. You need to decide what features are important for your application and examine the language accordingly. The brand new Atari computers offer yet another version of BASIC to tempt programmers and soon-to-be programmers. The following table is a comparison of the Atari BASIC (not Microsoft) language and the PET (Microsoft) BASIC language. I have indicated various features of each and then commented about the PET and Atari treatment or lack of treatment of that feature.

The table is not an exhaustive treatment of either language, but should assist you in learning the "basics" about both languages. The references used to determine the contents of the table are listed at the end of this article. You will also want to consult the manuals provided with the various computer peripherals to learn more about communication with these devices.

Two sources of information for the Atari BASIC language are provided with the computers: *Atari BASIC* by Albrecht, Finkel, and Brown and *BASIC REFERENCE MANUAL* (400-800) by Shaw and Brewster. I would like to share some thoughts with you about each one. Let's start with *Atari BASIC*.

The message on the binding indicates that *Atari BASIC* is "A Self-Teaching Guide" and the design of the book is well suited to accomplish that goal. The format uses proven teaching techniques. Each chapter begins with the instructional goals for that section and indicates what your skill levels should be when you finish it. The material is organized into numbered sections called frames, each of which presents information and then quizzes you about it. An important part of the learning process is the active participation on your part in answering the questions (without peeking at the answers) and writing the programs that are requested. By all means, turn your Atari on and use it in conjunction with the book. Another nice feature is the self-test at the end of each chapter and

at the end of the book. Answers are given to all the questions, but you will learn more if you take the tests without referring to them. This book is designed to teach BASIC to a novice and, if used properly, will accomplish this task very well.

Atari BASIC is not a reference book however, and BASIC programmers will grow frustrated trying to use it to learn about the Atari brand of BASIC. A welcome addition to the book sould be a categorized appendix which lists the Atari BASIC commands, statements, arithmetic and logical operators, special symbols, and variable naming conventions. (The built-in functions are already listed in the appendix, along with the ASCII character codes and error messages.) This type of "quick reference" section would also assist those who use this book to learn BASIC as they may need to refresh their memory from time to time.

The authors indicate in their message "To The Readers" that the BASIC in your new Atari computer may be more advanced than the 8K Atari BASIC they used in writing this book. This comment is an important one and means that you should read carefully all the manuals you receive with your unit to determine what refinements, if any, have been made. I am aware of at least one: *Atari BASIC* indicates that a variable name may be a single letter or a letter and a number, whereas the BASIC REFERENCE MANUAL gives you the freedom to create variable names of any length up to 120 characters as long as they begin with a letter. This difference should not create a lack of confidence in *Atari BASIC*, for the variable naming conventions given by Albrecht and company are probably best for beginners and are obviously still valid.

Atari BASIC does not include advanced programming techniques and applications such as creating and manipulating data files. You wll also not find information on saving and loading programs on cassette or disk; refer to the special operator's manuals for I/O information on these peripherals.

If you would like to learn Atari BASIC, *Atari BASIC* is an excellent place to start and I highly recommend it. If you already know BASIC and want to learn the idiosyncrasies of the Atari brand, read on!

I have been reading a preliminary draft of the new *BASIC REFERENCE MANUAL* which will be shipped with the Atari computers upon its completion. This book is designed in a more traditional manner, presenting information interspersed with examples. Be sure to start by reading the preface and the flowchart of the program for using the manual. Chapter 1 gives a general

introduction to the manual and its terminology and notation conventions. A lengthy list of abbreviations is given which you'll refer to frequently as you read through the manual.

The book is written in a friendly, non-threatening manner using a style that explains the BASIC language features in a very "readable," straight-forward way. One very nice feature of the style of text presentation is that the general format of a statement is presented first and then an example is given. For the most part, liberal use of visual aids such as flowcharts, diagrams, tables, and examples will assist you in your search for facts.

I believe that one or two sections will cause some difficulty for the beginning programmer, however. One of these is the section on Input/Output Operations. Dealing with the general format of the OPEN statement is not a trivial exercise and, since the book is aimed at all levels of readers, a different treatment of this complex subject would be easier for the newer computerist to grasp. The section on game controller functions has no examples longer than one line and very little information about the use of these functions. We are told that the "imaginative programmer will think of many uses" for these functions. Help! Atari — I'm not very imaginative and others might not be also; in the final manual please give us some ideas on how to use these unique functions.

I was pleased to find so many useful items in the appendices. There are several user programs and sample routines listed. A directory of BASIC keywords gives not only the keyword and a brief summary, but also gives the chapter number if you need further reference. A necessary listing is included of error messages and their corresponding numbers. Utility listings of Decimal to Hexadecimal conversion tables, and the ATASCII character set as well as PEEK and POKE information assist the serious programmer. A listing of trigonometric functions derived from the built-in functions should interest the scientific programmer. The section on the keyboard and editing features is a good introduction to this input device. It was an excellent idea to include, as an appendice, the glossary and chapter index of the words in the glossary, however I feel this addition should in no way replace a regular index. Hopefully, one will be included in the final edition.

Let me restate that all the comments I have made about the *BASIC REFERENCE MANUAL* came from examining a rough draft of the document. I look forward to reading the final copy. I have confidence that this manual will provide new Atari owners with ready access to their brand of BASIC.

Variable names

The first two alphanumeric characters form the unique variable name. However, for ease of reading, the name could be as long as you wish. Integer variables are created by adding % to the name. String variables are created by adding $ to the name.

Variable names may be any length, given memory limitations, and must start with a letter. 128 different variables are allowed in a single program. Each letter (rather than just the first two) is significant.

Subscripted variables

Three subscripts (i.e. three dimensional variables) are allowed. Original ROM PETs are limited to 255 elements in an array; on the newer models 8/16/32K PETs there is no limit on the number of elements except for memory limitations.

Two subscripts (i.e. two-dimensional variables) are allowed. Subscripts are numbered from 0.

String variables

Character strings may contain up to 255 characters even though the input buffer is limited to 80 characters. The concatenation operator + may be used to create longer strings (within the 255 limit). Non-subscripted string variables need not be dimensioned. Subscripted string variables must be dimensioned if the numer of elements in the array is over 10. The " symbol is used to designate characters strings.

All strings must be dimensioned. There is no limit on the length of strings; however, a limit of 99 is imposed for input of strings. String arays are not allowed. The + cannot be used for concatenation.

Integer variables

Integer variables may contain values of –32767 to 32767.

Not available

Dimensioning variables

No DIMension statement is necessary for arrays, single or multiple, which have subscripts with values of 10 or less. The dimension definition may be a constant, a variable, or an expression.

All character string and numeric arrays must be dimensioned.

Significant digits

Numeric values may contain nine significant digits and are

rounded if necessary.

Numeric values may contain nine significant digits and are truncated if necessary.

Scientific notation

Scientific notation is accepted for input as well as used to output numbers greater than 9 digits.

Same as PET

Arithmetic operators

+ addition − subtraction * multiplication
/ division • exponentiation
Same as PET except • for exponentiation.

Physical line

40 characters

38 characters

Logical line

80 characters

114 characters

Multiple Statements/line

Multiple statements are allowed and are separated by a : symbol.

Same as PET

Program Documenting

REM statements allow commenting in the body of your program.

Same as PET

Assignment

Keyword for an assignment statement is LET, but is not required. Assignment operator is the = symbol.

Same as PET

Looping

Looping may be accomplished by using the FOR-NEXT-STEP statements. The STEP value may be an integer or fraction, positive or negative (therefore allowing the value of the index to ascend or descend). Whatever the beginning and ending values of the index are, the loop will be executed at least once. A single NEXT statement may be used to match multiple FOR statements. For example: NEXT X,Y,Z.

Same as PET except the same NEXT may not be used for

multiple FOR statements. Also, NEXT must be followed by its variable. PET allows NEXT with an implied variable.

Input

The INPUT statement may have a prompt message included which will be presented to the user before the ?. This statement may be used with all types of variables.

You may not include a prompt message in the INPUT statement. INPUT may not be used with a subscripted variable.

The READ statement may be used for input with corresponding DATA statements. Data may be reused if the RESTORE statement is included appropriately. This type of input may be used with all type of variables.

READ . . . DATA also may not be used with a subscripted variable.

The GET statement may be used to input a single byte.

GET statement: same as PET, except that it waits for a keystroke.

Branching

Unconditional branching may be accomplished by using the GOTO statement with the statement number of the target statement. Conditional branching options include ON . . . GOTO and ON . . . GOSUB statements.

The argument of a GOTO or GOSUB may be a variable or an expression.

Subroutines

Subroutines are accessed by the GOSUB statement and need a RETURN statement to indicate the end of the routine.

The command POP can be used to cancel a GOSUB.

Decision-making

The IF – THEN statement uses the conditional operators , , = , =, =, and the Boolean operators AND, OR, NOT, for comparing both numeric and string variable values. Multiple statements following THEN will be executed if the condition is true.

AND, OR, and NOT do not operate on the binary level.

Output

The PRINT statement may include variables, arithmetic expressions, character strings, and constants. There are four default print positions which are used if items in the list are

separated by commas. A semi-colon between items causes closer printing with character strings being concatenated and numbers separated by one space. Cursor movement may be included in strings in the PRINT list. Output is automatically sent to the screen; a special OPEN statement is needed to cause printing on the printer. The ? symbol may be used to represent PRINT when keying in a program. The interpreter will insert the full word for you.

Same as PET except: the semicolon causes concatenation of numbers too. LPRINT is used to send output to the printer. The ? is not spelled out in a program.

Program termination

STOP and END will cause the program to cease execution. There does not have to be and END statement as the last statement in the program. The CONT command will allow you to resume execution after and END or STOP has been encountered.

CONT continues on the next line, not necessarily the next statement.

User-defined functions

DEF FN will allow you to create your own function in BASIC.

User-defined functions are not available in BASIC.

Built-in functions

Standard trigonometric and arithmetic functions are available as well as special purpose functions to do the following: PEEK at memory locations, TAB the cursor to a specified column, SPaCe the cursor the specified number of spaces, indicate the POSition of the cursor, give the number of FREe bytes left in memory, pass a parameter to a USeR machine language program, and communicate with the PET clock. Functions may be nested.

Standard trigonometric and arithmetic functions, FRE, and PEEK are the same as PET. In addition there are CLOG for base 10 logs, ADR to return decimal memory address of specified string, DEG, RAD to specify either degrees or radians for this function. Tab operations are accomplished by keystroke combination, POSITION, or POKE.

Standard string functions are available, as well as special functions to designate substrings. The + symbol is used as the concatenation operator.

Same as PET but no functions for substringing. Substrings are formed by using subscripts with the string variable name to indicate characters in the string. The + is not used for concatenation.

Graphics capabilities

Graphics symbols are accessed by pressing the shift key and the appropriate key (printed on the front of the keys on the PETs with graphic-style keyboards). These symbols may be used in PRINT statements to create displays on the screen. Graphic displays may also be created by using the POKE statement to insert graphic symbols into the screen memory.

The Atari also has special characters, and provides special keywords to make creating graphic displays much easier, such as PLOT, DRAWTO, POSITION, FILL (X10 18), POKE, and GRAPHICS. There are nine different graphic modes: three for text only, giving normal, double wide, and double size characters; three modes with split screen & four colors; two with split screen and only two colors; and one high resolution mode.

Color capabilities

No color capability

Special keywords are provided to create color displays, such as COLOR which selects one of four color registers, and SETCOLOR to specify the hue and luminance of each color register. By using a combination of 16 hues and 18 luminance settings 128 colors can be created.

Sound capabilities

Sound is achieved by using the POKE statement to cause signals to be sent to the parallel user port to which is attached an external device to produce sound. Rhythm is controlled by using timing loops. Non-Commodore products are available for the PET to produce four-voice music similar to the Atari.

Atari provides a SOUND statement which allows specification of voice, pitch, distortion, and volume. Four voices can be played at the same time. Control of distrotion creates interesting sound effects. Rhythm is controlled by timing loops. The sound is heard through a speaker in the TV monitor.

Game I/O

No special statements or functions are available to aid in game

interaction.

Four functions are provided for ease in programming paddle and joystick control. They are PADDLE and STICK to control movement, and PTRIG and STRIG to control the trigger button.

Files

Files must be OPENed before use with parameters specifying logical file number, device number, secondary address (permits intelligent peripherals to operate in any number of modes), and file name (for tapes, name may be up to 128 characters. Only 16 characters are used by the system, though). The CLOSE statement is used to close a file and needs only the logical file number as a parameter. PRINT#, INPUT#, GET# are used with tape or disk file I/O. Tape files are recorded twice to aid in checking for errors.

Files must be OPENed before use with parameters specifying logical file number, type of operation (read, write, both), file name (8 characters or less) and device type. PRINT#, INPUT#, PUT#, GET#, and X10 may be used for I/O operations. NOTE and POINT are functions provided to facilitate creation of random access files.

Commands

In addition to the standard commands of NEW, LIST, RUN, CONT, LOAD, SAVE, and POKE, the PET has a VERIFY command to allow tape files to be verified before erasing memory, and a CMD command to keep the IEEE-488 Bus actively listening. The LOAD and SAVE commands may include a file name.

Atari commands are the same as PET except that Atari has no VERIFY or CMD commands and file names may not be used with the CLOAD and CSAVE commands. Program files can be located on the tape by means of the counter on the cassette.

Error correction & editing

You may erase characters or an entire line while typing. Later editing of programs is possible by cursor control and line deletion, by typing the line number and RETURN. Duplication of lines is possible by first LISTing the line, changing the line number, and pressing RETURN.

Same as PET, with the addition of three editing functions: insert line, delete line, and backspace (not the same as delete).

Error messages

For syntax errors, the line number is given, but not the cause of the error. For execution errors, the error message and line number are printed on the screen.

Syntax errors are indicated by pringing the line and showing the error in reverse video. Execution errors will cause a message to appear on the screen giving you an error message to look up in your manual.

REFERENCES:

1. Bob Albrecht, Leroy Finkel, Jerald R. Brown, **Atari BASIC.** John Wiley & Sons, Inc., New York (1979)
2. Carol Shaw, Keith Brewster. **BASIC REFERENCE MANUAL.** draft, Atari, Inc., Sunnyvale, CA (1979)
3. **CBM User Manual,** First Edition. Commodore Business Machines, Santa Clara, CA (1979)
4. **Atari 400 Operators Manual.** Atari, Inc., Sunnyvale, CA (1979)
5. **Atari 800 Operators Manual.** Atari, Inc., Sunnyvale, CA (1979)

The Ouch in Atari BASIC

Glenn Fisher and Ron Jeffries

Atari does have some flaws — not catastrophes, but flaws nonetheless. Note that the LET operator does permit any name to be used as a variable name.

After using the Atari 800 for a couple of months, we have found its version of BASIC to be less than perfect. Please don't misunderstand; we think that the Atari is a great machine, and is very usable in spite of these faults. (Other computers will have an equally long list of defects, they will just be different defects.)

Essentially, there are no character strings in Atari BASIC. Instead, you have arrays of characters, which ain't the same thing! (On the good side, however, you are not limited to 255 character strings as in Microsoft BASIC.)

Would you believe there are no error messages? Well, unless you consider ERROR 9 to be an error message . . . (it means "Subscript out of range").

There is not a DELETE command. True, few of the competing BASICS have this essential feature, either. But hope springs eternal.

Atari doesn't have user-defined functions (such as DEF FNA(X))). This is one of those things you don't miss until you need it, but when you need it you really need it!

Would you believe — there is not a TAB function? This is essential when you need to produce neatly formatted output.

AND and OR do not allow you to get at individual bits of a number. (We see you yawning! but this is more important than you might suspect, especially when dealing with PEEKs and POKEs.)

Unlike some of Atari's competitors, the Atari does not, repeat NOT, have any "typeahead." Typeahead allows you to give commands before previous commands finish, which is very nice when you want to quickly give a series of commands.

As best we can tell, there is no way to verify a saved file to see that it got saved properly. Of all the things to omit . . .

The GET statement has an interesting "feature": it waits until there is a character available. It would be far more convenient if it returned a special "no data yet" value.

There is a clock in the Atari, but you, Dear Reader, don't get easy access to it. There are BASICs that give you clock values in two flavors: as "ticks" since the machine was turned on, and as time of day measured from when the machine was turned on.

Although you can have long, meaningful variable names (all of whose characters are significant, as opposed to lesser BASICs that only use the first two characters), there is a problem! Variable names cannot contain keywords. For example, POINTS and SCORE are both illegal. (This from a company known for its games!)

You can't list an open-ended line range. So, you have to say LIST 500,32767 when what you want to do is list everything from line 500 on. Sigh!

The INPUT statement doesn't allow a prompt string. You have to first PRINT the prompt, then do the INPUT. Sure, you can live with it, but it's a pain.

Here's one for the books: in Atari BASIC you can't READ or INPUT a value into an array element! (You guessed it: you first READ into an ordinary variable, then assign that variable to the array element. I hope that somebody on the design team at least has a guilty conscience.)

You can only have four colors on the screen at once. (The Apple has a minimum of six.)

The BREAK key should turn off sound. (It is nice that typing END will do it, however.)

Obviously, this list represents what we know as of November, 1979, when this was written. To the best of our knowledge, all of the problems are real. We won't be surprised if some of these flaws are corrected by Atari. (We may also have misunderstood the preliminary manuals.)

Finally, if you feel that we are really "down" on the Atari, please realize that none of the problems mentioned here are serious enough to keep us from publishing our Atari software product. Despite its flaws, the Atari is a very useful and flexible personal computer.

Atari BASIC Part II

John Victor

There is no question that the Atari graphics and other machine features make it superior to its predecessors as a personal computer. But these great features would be worth little if programmers could not readily take advantage of them. Atari BASIC makes the use of color graphics and the generation of sound incredibly easy.

A good example of what can be done with Atari BASIC can be found in the December issue of INTERFACE AGE. Al Baker of the Image Producers wrote a short version of the game of SIMON for the Atari 400 (using about 80 instructions). The game used color graphics and musical chords. The player attempts to duplicate a series of notes and colors made by the computer in ever increasing lengths, and his or her entries are made by pushing a joystick. All of the versions of this game that I have seen on other computers have involved some machine language kluges to make them work, but this Atari program is done entirely in BASIC. The only thing here that might give the novice programmer some difficulties is the mathematical relationships of musical notes in a chord. Otherwise, the program is a model of simplicity.

Although Atari BASIC is not Microsoft BASIC, it is pretty much like the BASICs found on Apple, PET, and the TRS-80. The BASIC interpreter resides in a 10K ROM cartridge that plugs into a slot in the front of the Atari 400 or 800. (Both computers use the same BASIC.) Its floating point software computes to 9 place accuracy, it supports multiple statement lines, and it contains the usual compliment of library routines. Its execution speed appears to be a bit slower than Applesoft's, but it seems to be better than TRS-80 Level II. If the BASIC has any deficiencies, it is in the area of string handling logic. It does not support string arrays.

In some ways the BASIC resembles Apple's integer BASIC. This is particularly noticeable to Apple programmers when the computers enters the graphics modes and finds an area at the bottom of the screen with 4 lines of text. Atari BASIC also allows the programmer to use variables in GOTO and GOSUB statements (GOTO A). In addition, the variables can be words, (GOSUB ERRORROUTINE, GOTO CHOICE, etc.), where CHOICE, for example, has a line number as a value.

There is one incredible innovation here that makes Atari BASIC unique. ANY WORD CAN BE USED AS A VARIABLE

— EVEN SO-CALLED 'FORBIDDEN' WORDS! The programmer could use the word END or LIST as a variable. This is definitely not allowed in any other version of BASIC. LIST can also be used as a program statement to make the listing of the resident program print out during the running of the program.

Here are some examples of the use of words as variables in Atari BASIC. Note that if a program command is goin to be used as a variable, the word LET must precede it when setting its value.

```
10 LET LISTING = 1000 :
LET ERRORCOUNT = 2000 :
LET CHOICE = 3000
. .
. .
120 IF ANSWER$ = CORRECT$
THEN RETURN
130 GOTO CHOICE
. .
. .
```

Variable graphics modes can be entered by giving the graphics instruction along with a number. Most of these will have an area at the bottom of the screen for four lines of text. The programmer can eliminate this area by adding 16 to the number of the graphics mode. For example, GRAPHICS 3 has four lines of text, but GRAPHICS 3+16 does not. The graphics instruction will clear the screen. This can also be deactivated by adding 32 to the graphics mode number (i.e. GRAPHICS 35 enters GRAPHICS 3 without clearing the screen.)

The following is a brief description of some of the GRAPHICS modes.

Graphics 0

This is the regular text mode for BASIC. The user gets 24 lines of 40 characters, where the characters can be upper or lower case, regular or reversed. In addition, the user can access, by pressing the CONTROL key, a set of pseudo graphics from the keyboard. These special characters can be used to draw pictures (very much like the special characters found on the PET).

The user has the ability to change the background color using the SETCOLOR instruction. The user can change the color designated by color register 2 (which controls background color) with the following instruction: SETCOLOR 2,4,14. The screen will turn light pink, since color register 2 contains the number 4 for red and the number 14 for the luminescence (0 for darkest to 14 for

lightest).

In GRAPHICS 0 the user cannot mix the color of the type, which can only be a darker or lighter version of the background color. By setting color register 1 with a luminescence of 0, we get a dark type against a light background. SETCOLOR 1,0,14 plus SETCOLOR 2,0,0 will produce a dark grey background with light characters. Using the luminescences, the user has a choice of about 120 different shades of colors.

Graphics 1 And Graphics 2

There are the "large type" modes, with GRAPHICS 2 producing the largest type. In this mode the characters can be put on the screen in a variety of ways — they can be PLOTed on like graphics, or PRINTed on. Different color characters can be made by defining the characters as upper case, lower case, or reversed characters. When the type appears on the screen, it appears as all capitals, but the color of the characters is different. A word printed as lower case may appear on the screen as upper case red characters, while a word printed as reverse capitals may be blue. For example, PRINT #6; "BLUE green" produces 2 "all-capital" words in two different colors.

Graphics 3 To Graphics 11

These are the real graphics modes where the computer PLOTs points at a given screen location. GRAPHICS 3 has the largest points, and the size goes down as the mode number increases. GRAPHICS 11 is a high resolution mode. The color of the points is taken from the color register indicated by the user. COLOR 3 tells the computer to make the point the same color as specified in color register 3.

To make plotting easier, the graphics modes use a DRAWTO instruction which will automatically plot a line from any given point to any other point on the screen, even if the line is a diagonal. There is also a technique to fill in a predetermined area of the screen to make a square of a specific color.

Sound

The user has a choice of four sound generators which can be used to produce sounds or musical tones. The sound generators can also be used simultaneously to make chords. Once turned on, each sound generator stays on until the program reaches and END statement or the program shuts it off. SOUND 0, 121, 10, 8 plays middle C on sound register 0.

Control Characters

Screen and cursor control functions can be put in a BASIC program in PRINT statements as control characters. If the user wants to clear the screen he or she can press the Clear Screen key. This can also be done in the program by making a PRINT statement and then pressing the ESCAPE key. When the user hits the Clear Screen key, a special control character is printed. When the program is run and the PRINT statement is executed, the screen will be cleared. The statement will appear like this: PRINT "↑".

Editing And Error Messages

The screen editor on the Atari is the best I've seen. On the Apple, for example, the user cannot move type around the editor field, but on the Atari this can be done with simple keyboard inputs. The user does not need to worry about hidden errors, or relisting since all changes are immediately visible. If the user is making a line too long, a bell rings a warning (just as it does on a typewriter).

If a syntax error is made while entering or editing a line, the BASIC interpreter gives an immediate error message at the carriage return. This saves quite a bit of debugging time when entering a program. Unfortunately, for errors encountered during a program run, the user gets a numbered error message that must be checked in the manual. There are several of these messages, so they are not going to be easily memorized.

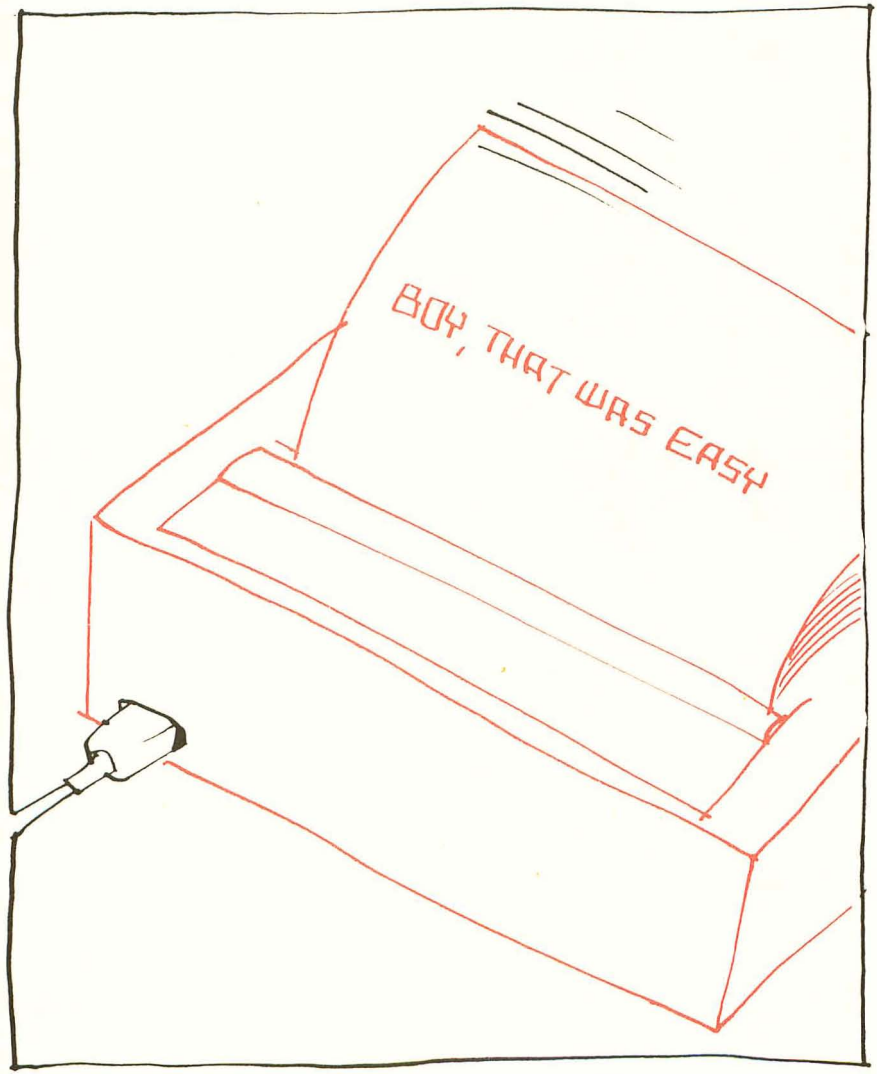
Computer I/O

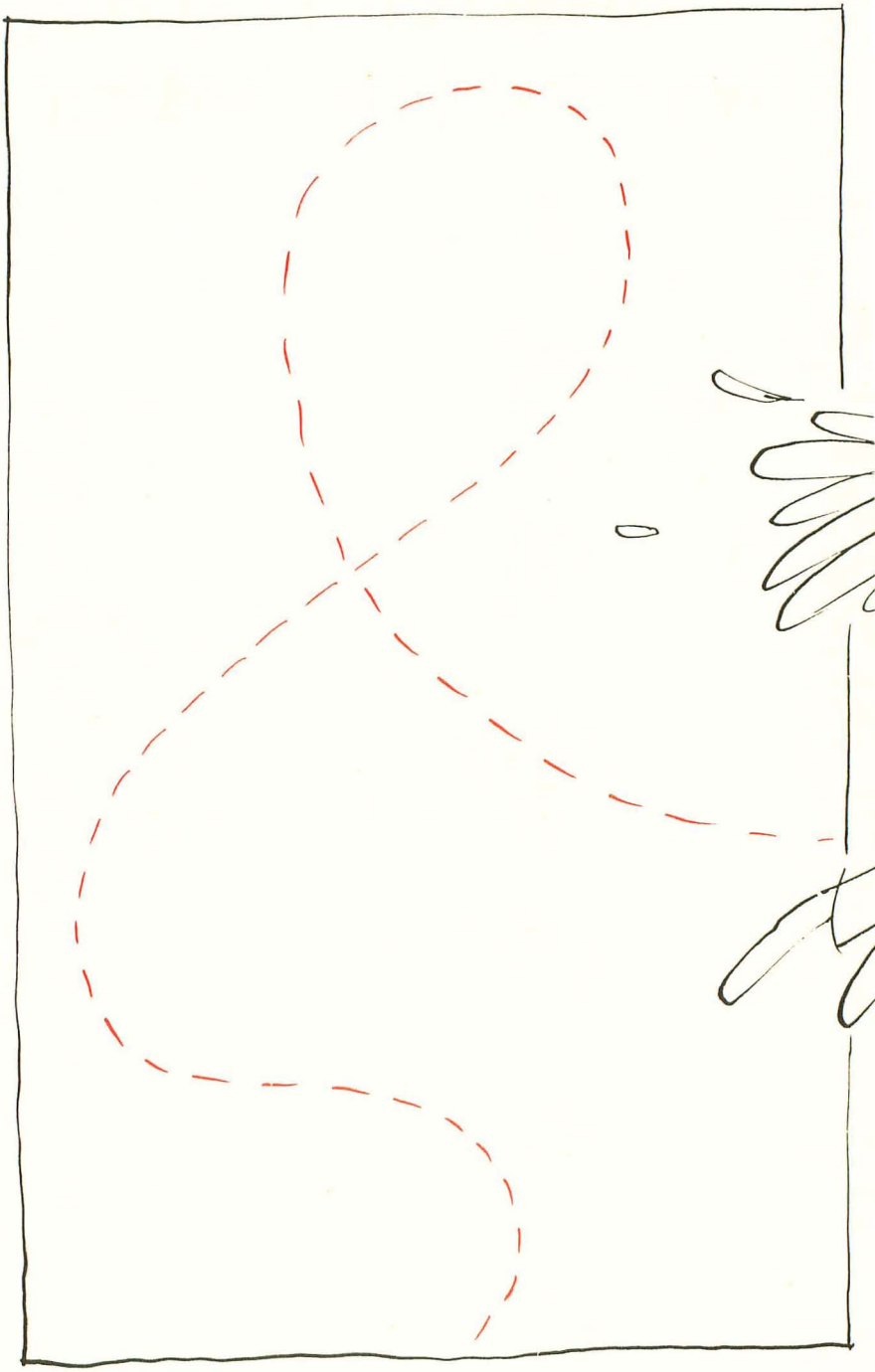
In order to get FCC approval for the computer (so it could be plugged into a regular TV set) Atari had to get approval for all of its peripheral devices at the same time. So the computer and its peripherals were designed as one package. This is reflected in the ease of access to peripherals from the BASIC. There are specific instructions to access disk, joysticks, printers and the cassette machine directly from BASIC. In addition to these, the user can define peripherals using an OPEN instruction. For example: OPEN #2, 8, 0, "C:" opens the cassette machine for special operations. The cassette player is now specified by #2. PUT #2, A outputs the value of A to the cassette player. The user can use INPUT, PRINT, GET, PUT, etc. as I/O instructions to peripherals.

BASIC can also treat the video screen and the keyboard as I/O devices for certain kinds of operations.

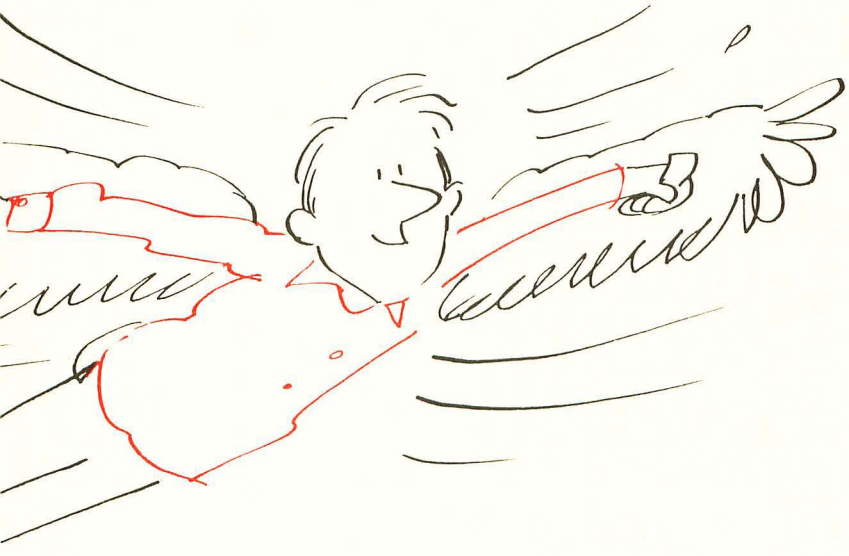
Atari BASIC, like any other version of BASIC, suffers from some deficiencies when considering it for some special application.

However, in the area of graphics and manipulation of text displays, this version of BASIC is, in my opinion, hands down superior to Apple, PET or TRS-80 BASIC. Its functions are complex, but the user will find the BASIC relatively easy to use compared to some other forms of BASIC.





CHAPTER TWO:
Beyond The Basics



Inside Atari BASIC

Larry Isaacs

For those who want to experiment with the machine, write utility aids, or just tinker around . . .

This article will present informatin on how ATARI BASIC stores programs in memory. If you are new to the field of microcomputer programming, this information should help increase your awareness of what your ATARI is doing.

The following information is based solely on what I have been able to observe while working with an ATARI 800. I believe the information to be accurate. However, it is hard to know how complete the information is.

Also, for those new to microcomputer programming, the next section gives some preliminary information which should help make the rest of the article more understandable.

Preliminary Information

One very important term in the field of microcomputing is the term **"byte."** For purposes of this article, it can be considered a number which can have a value ranging from 0 to 255. The memory in your ATARI consists of groups of bytes, each byte of which can be referenced by a unique address. The part of memory which is changeable, called RAM, starts with a byte at address 0 and continues with bytes at increasing sequential addresses until the top of RAM is reached. The top of RAM is determined by the type and number of memory modules you have in your ATARI.

Bytes, or combinations of bytes, can be used to represent anything you want. Some common uses for bytes include representing memory addresses, characters, numbers, and instructions for the CPU in your ATARI. You will be exposed to several different uses for bytes in this article. Some of these uses will make reference to *two byte binary numbers*. This is where two bytes are used to represent a number whose value ranges from 0 to 65535. The decimal value of a two byte binary number can be computed using the formula: **FIRST BYTE+(SECOND BYTE *256).**

Also in this article, reference will be made to *page zero*. Page zero simply is the first 256 bytes of memory, i.e. addresses 0 through 255. This part of memory differs from the rest of memory in that these bytes can be referenced using a single byte address. The rest of memory requires two byte addresses.

The Conversion

After typing in a BASIC line, hitting RETURN causes the line to be passed to the programs found in the ATARI BASIC cartridge. Here the line will undergo a certain amount of conversion before it is stored in memory. One part of this conversion involves converting all of the BASIC reserved words and symbols to a one byte number called a *token*.

Another part of the conversion involves replacing each variable name in the line with an assigned number which will range from 128 to 255. If a variable name has been previously used, it will be replaced by the number previously assigned. If it hasn't been used before, it will be assigned the lowest unused number, starting with 128 for the first variable name. Also, numbers in the BASIC line must be converted into the form which the ATARI BASIC uses before they can be stored in memory.

After the conversion is finished, the line is stored in memory. If the BASIC line does not have a line number, it will be stored after the last statement of your BASIC program, and executed immediately. If it does contain a line number, the converted line will be inserted in the proper place in your program. After the line has been executed or stored, your ATARI will wait for you to type in another line. Even though the line undergoes this conversion, the order in which the reserved words, variables, and symbols occur in the line isn't changed when it is stored in memory.

The Memory Format For A Basic Line

Let's begin with the general format of how a BASIC line is stored. Once a BASIC line has been converted and stored, the line number is found in the first two bytes of the memory containing the BASIC line. These bytes form a two byte binary number which has the value of the line number. The value of this number can range from 0 to 32767.

The third byte contains the total number of bytes in this BASIC line. This means you can find the first byte of the next line using the following formula: **ADDRESS OF FIRST BYTE OF NEXT LINE = ADDRESS OF FIRST BYTE OF CURRENT LINE + NUMBER IN THIRD BYTE OF CURRENT LINE.**

The fourth byte contains the number of bytes in the first statement in the line, including the first four bytes. If the BASIC line contained only one statement, the third and fourth bytes will contain the same value. If the line had more than one statement, these bytes will be different.

Next come the bytes which represent the first statement in the line. If there is more than one statement, the next byte following the first statement contains the number of bytes in the first two statements. Naturally, if there is another statement after the second one, the first byte after the end of the second statement contains the number of bytes in the first three statements, etc.

This completes the format of a BASIC line as it is found in memory. Before going on, let's put this information to use in a short program which lists out its own line numbers along with the beginning address of each line. To do this we must first find out where the first byte of the first line is found. It turns out there is a two byte binary number found in page zero which contains the beginning address of the first line. This number is contained in bytes 136 and 137. Also, we will know when we've reached the end of the program when we find a line number of 32768, which is one more than the maximum allowed by ATARI BASIC. The program to print the line numbers and their beginning addresses is shown in Listing 1.

Tokens

In order to conserve memory, all of the BASIC reserved words, operators, and various punctuation symbols are converted into a one byte number called a token. This conversion also makes execution simpler and faster. The tokens can be divided into two groups. One group contains the tokens which occur only at the beginning of a BASIC statement and the other group contains the tokens which occur elsewhere in a BASIC statement.

Let's first take a look at the tokens which occur at the beginning of a BASIC statement. It turns out that all statements will begin with one of these tokens. After some investigation, I found that these tokens will range in value from 0 to 54.

The procedure for listing the tokens is fairly simple, though the actual implementation is a bit more involved than the brief explanation which follows. The idea is to put "1 REM" as the first statement of the program. Then use POKEs to change the line number and token of this REM statement. By setting the line number and token to the same number, listing the line will print the token and corresponding BASIC reserved word. Fortunately the programs in the BASIC cartridge which do the listing tolerate the incomplete BASIC statements. The program for displaying these tokens is shown in Listing 2. Notice when you run this program, no reserved word is printed for token 54. This is the invisible LET.

token which is used for assignment statements which don't begin with LET.

A similar procedure can be used to list the other tokens as well. The main differences are to make the first statement "1 REM A", POKE 54 (the invisible LET token) into the first byte of the statement, and make the changes for the token to the second byte of the statement. The values for the tokens which occur after the beginning of a statement range from 20 to 84. The program for printing these tokens is given in Listing 3.

After running this program, you will notice there is no reserved word or symbol printed for token 22. Token 22 is the terminator token found at the end of each BASIC line, except those whose last statement is a REM or DATA statement. Also, tokens 56 and 57 didn't print a reserved word or symbol. Both of these tokens represent the "(" symbol. The "(" doesn't print because these two tokens are associated with array names, and the "(" symbol is kept with the associated variable name, as will be seen in the next section.

Of course you noticed that most of the symbols occur more than once. There is a different token for each of the different uses of the symbol. For example, the word "=" has four different tokens. Token 45 calls for an arithmetic assignment operation as in A=A+1. Token 46 calls for a string assignment as in A$="ABC". Token 34 is used in arithmetic testing as in IF A=1 THEN STOP. And finally, token 52 is the same as token 34 except that it's for testing strings.

One more token, found after the ones listed in the previous program: token 14, which indicates a constant is stored in a following six-byte grouping.

Variable Names And Constants

As each new variable is encountered, it is assigned a number. These numbers begin with 128 and are assigned sequentially up to 255. Notice these numbers will fit into one byte. Also, as each new variable is encountered, the variable name is added to a variable name list, and 8 bytes of memory are reserved for that variable. In the case of undimensioned variables, these 8 bytes will contain the value of the variable. For strings and arrays, these 8 bytes will contain parameters, with the actual values and characters stored elsewhere.

This method of handling variables has some advantages. One advantage is that it keeps usage to a minimum. The variable name is

only stored once, and each time that name is referenced in a BASIC statement, it occupies only one byte in the stored program. Another advantage is that the address where the value for a variable is stored can be computed from the assigned number. This isn't true of the BASIC found in some other microcomputers where values must be searched for.

There are also some disadvantages as well. First, it limits you to 128 different variable names. However, the great majority of programs won't need more than 128 variable names. One other disadvantage is that, should a variable name be no longer needed, or accidentally entered due to a typo, there is no quick way to remove that variable from the variable name list and reuse the 8 bytes reserved for it.

Apparently, the only way to get rid of unwanted variables is to LIST the program to cassette or disk. For example, LIST "C" will list the program to cassette. Once the program is saved, use the NEW command to clear the old program. Then use the ENTER command to reload the program. For cassette this would be ENTER "C." Using the LIST command saves the program in character form. ENTERing the program then causes each line to be converted again as was done when you first typed it in. Now only the variables found in the program will be placed in the variable name list, and space reserved for their value. Using CSAVE and CLOAD won't do this because these save and load a copy of the memory where the program is stored. Unwanted variables are saved and loaded with the rest of the program.

Constants are stored in the BASIC statements along with the rest of the line. The constant will be preceded by a "14" token as mentioned previously. Explaining how ATARI BASIC represents the numbers used as constants and as variable values will require some explanation about BCD (Binary Coded Decimal) numbers. I will save this information for a later article.

To give an example of using the information in this section, let's take a look at the variable name list. Fortunately bytes 130 and 131 contain the address of the beginning of the variable name list. The list will consist of a string of characters, each character occupying one byte of memory. To indicate the last character of a name, ATARI BASIC adds 128 to the value representing that character. Since the values representing the characters won't exceed 127, the new value will still fit into one byte. To indicate the end of the list, a 0 is placed in the byte following the last character of the last name. The program which prints the variable name list is given

in Listing 4. Notice, when you run this program, that the "(" is saved as part of an array name, and the "$" as part of a string name.

Memory Organization

Finally, let's look at how the memory is organized for a BASIC program. The order in which the various parts of a program are found in memory is shown in Figure 1. The only part whose beginning is fixed is the variable name list which begins at address 2048. The beginning of the other parts will move appropriately, as the program grows. There are addresses in page zero which can be used to find each of the parts shown in Figure 1. These addresses, usually called pointers, are shown in Table I. This table includes the two pointers which were used in the previous programs.

Figure 1. MEMORY ORGANIZATION

```
Increasing
Addresses
????   End of Array Storage Area

  .
????   Beginning of Array Storage Area
????   End of Program
  .
????   Beginning of Program
????   End of Variable Storage Area
  .
????   Beginning of Variable Storage Area
????   End of Variable Name List
  .
2048   Beginning of Variable Name List
```

TABLE 1

ADDRESSES	NAME	CONTENTS POINT TO
130 & 131	BON	Beginning Of variable Names list
132 & 133	EON	End Of variable Name list
134 & 135	BOV	Beginning Of Variable storage area
136 & 137	BOP	Beginning Of Program
138 & 139	CEL	Beginning Of Currently Executing Line
140 & 141	BOA	Beginning Of Array storage area
142 & 143	EOA	End of Array storage area

Application

For those who are interested in putting this information to use, I will present one example here. I will try to give more examples in future issues of COMPUTE!.

At some time you may find it useful to be able to "undimension" some arrays of strings and reuse the memory for some other arrays and strings. It turns out that the CLR function only clears the variables found between the BOV (Beginning Of Variables) pointer and the BOP (Beginning Of Program) pointer. By temporarily changing the BOP pointer, we can keep some of the variables from being cleared. The array storage area is cleared by setting the EOA (End Of Arrays) pointer equal to the BOA (Beginning Of Arrays) pointer. We can save some of the array storage area by temporarily changing the BOA pointer.

The listing for this UNDIMENSION routine is shown in Listing 5. The listing also includes a small demo program to illustrate its use. Note that all of the names of variables which are to be cleared should occur in the program prior to any of the names of variables which are to be saved. This puts the storage for the variables to be cleared at the beginning of the variable storage area. Also note that a dummy string which can be cleared is needed by the UNDIMENSION routine. In your main program, this dummy string should be dimensioned just before dimensioning the strings and arrays that you will later clear, as was done in statements 120 and 150. This allows the use of the ADR functions to find the end of the array area to be saved.

The reason the UNDIMENSION routine is not executed by a GOSUB is that the return line number is lost in the clearing process. Loop parameters will also be lost, so the routine shouldn't be executed while in a FOR..NEXT loop.

Conclusion

I hope that you found the information in this article understandable and will find it useful at some point in the future. The information does show that ATARI BASIC is fairly efficient at using memory to store programs. Also, there is very little penalty in memory usage when using long variable names. If you have any questions please send them to **COMPUTE!**.

```
10 REM PROGRAM TO PRINT LINE NUMBERS
20 REM AND THEIR ADDRESSES
30 REM
40 REM Get address of first line
50 ADDRESS=PEEK(136)+PEEK(137)*256
60 REM Get the line number
```

```
70 LNUM=PEEK(ADDRESS)+PEEK(ADDRESS+1)*25
6
80 REM Test for end of program
90 IF LNUM=32768 THEN END
100 REM Print line number and address
110 ? "LINE #";LNUM;
120 ? " STARTS AT ADDRESS ";ADDRESS
130 REM Get address of next line
140 ADDRESS=ADDRESS+PEEK(ADDRESS+2)
150 GOTO 70
```

```
1 REM
100 REM PROGRAM TO PRINT THE TOKENS
110 REM WHICH BEGIN BASIC STATEMENTS
120 REM Get the beginning of program
130 BASE=PEEK(136)+PEEK(137)*256
140 REM Change statement terminator
150 POKE BASE+5,22
160 ? CHR$(125):REM CLEAR SCREEN
170 REM PRINT TOKENS
180 FOR I=0 TO 54
190 REM Change line number and token
200 POKE BASE,I:POKE BASE+4,I
210 LIST I:REM Print token
220 REM Undo line feed if needed
230 IF I>1 THEN ? CHR$(28);
240 REM Change left margin for columns
250 IF I=19 THEN POKE 82,12:POSITION 12,
1
260 IF I=39 THEN POKE 82,24:POSITION 24,
1
270 NEXT I
280 REM Put program back to normal
290 POKE BASE,1:POKE BASE+4,0
300 POKE BASE+5,155
310 POKE 82,2:POSITION 2,22
```

```
1 REM A
100 BASE=PEEK(136)+PEEK(137)*256
```

```
110 REM Change beginning token
120 POKE BASE+4,54:POKE BASE+6,22
130 REM Print operator and function toke
ns
140 PRINT CHR$(125)
150 FOR I=20 TO 84
160 POKE BASE,I:POKE BASE+5,I
170 LIST I
180 REM Undo line feeds
190 ? CHR$(28);:IF I=22 THEN ? CHR$(28);

200 IF I=39 THEN POKE 82,11:POSITION 11,
1
210 IF I=59 THEN POKE 82,13:POSITION 19,
1
220 IF I=79 THEN POKE 82,28:POSITION 28,
1
230 NEXT I
240 POKE BASE,1:POKE BASE+4,0
250 POKE BASE+5,65:POKE BASE+6,155
260 POKE 82,2:POSITION 2,22
```

```
100 REM PROGRAM TO PRINT THE VARIABLE NA
ME LIST
110 DIM ARRAYNAME(1),STRINGNAME$(1)
120 REM GET THE BEGINNING OF THE LIST
130 ADDRESS=PEEK(130)+PEEK(131)*256
140 ? CHR$(125);"VARIABLE NAME LIST"
150 REM GET CHARACTER AND TEST FOR END
160 A=PEEK(ADDRESS):IF A=0 THEN END
170 REM PRINT CHARACTER
180 IF A<128 THEN ? CHR$(A);:GOTO 210
190 ? CHR$(A-128)
200 REM GET NEXT ADDRESS AND REPEAT
210 ADDRESS=ADDRESS+1:GOTO 160
```

```
1 REM DIMENSION THE DUMMY STRING
2 DIM DUMMY$(1)
3 REM DIMENSION THE ARRAYS AND STRINGS
```

```
4 REM WHICH WILL NEED CLEARING
5 DIM A1(1),A2(1)
6 CLR :REM CLEAR THE VARIABLES
7 N=3:REM # OF VARIABLES JUST DIMENSIONE
D
8 REM INCLUDING DUMMY$
9 REM YOUR PROGRAM MAY BEGIN HERE
100 REM HERE IS AN EXAMPLE OF HOW TO
110 REM USE THE UNDIMENSION ROUTINE
120 DIM TEST$(20):TEST$="I'M STILL HERE"

130 DIM DUMMY$(1),A1(50,10)
140 A1(50,10)=1:? A1(50,10),TEST$
150 REM EXECUTE UNDIMENSION ROUTINE
160 LINE=170:GOTO 1020
170 DIM DUMMY$(1),A2(500)
180 A2(500)=2:? A2(500),TEST$
190 END
200 REM
1000 REM UNDIMENSION ROUTINE
1010 REM SAVE CURRENT POINTER VALUES
1020 S136=PEEK(136):S137=PEEK(137)
1030 S140=PEEK(140):S141=PEEK(141)
1040 REM MOVE END OF VARIABLES
1050 T1=PEEK(134)+8*N:T2=PEEK(135)
1060 IF T1>255 THEN T2=T2+1:T1=T1-256
1070 POKE 136,T1:POKE 137,T2
1080 REM MOVE BEGINNING OF ARRAYS
1090 T2=INT(ADR(DUMMY$)/256)
1100 T1=ADR(DUMMY$)-T2*256
1110 POKE 140,T1:POKE 141,T2
1120 CLR :REM CLEAR THE ARRAYS
1130 REM RESTORE POINTERS AND RETURN
1140 POKE 136,S136:POKE 137,S137
1150 POKE 140,S140:POKE 141,S141
1160 GOTO LINE
```

Atari BASIC Structure

W. A. Bell

By now you probably have had your Atari® Computer for a few months, and have had a chance to put in some fairly large programs and tinker with and embellish them. You may have even written some programs of that type. If so, then you have undoubtedly wished for a renumber command. In fact, if you have used BASIC on other systems, then you have probably roundly cursed those programmers who left that facility out. Or you may have wanted to change the name of a variable to make it more self-documenting, but didn't know everywhere it occurred. This article will explore, in tutorial fashion, the structure of Atari BASIC programs as they are stored in memory. It will provide you some tools for doing more of your own exploring, and then show how you can put this type of information to use.

To begin our exploration inside BASIC, the program shown in Listing 1 is useful. It lets us peek around in memory to find things that are of interest. It will search memory from a specified starting address and tell you where it finds a string of characters or data you have specified, or it will find address pointers to a specified memory location. It will also let you dump memory in two formats, decimal or hexadecimal, and character. If your Atari is plugged in, it may help your understanding to follow along on your keyboard.

Do the following steps in direct mode:

```
NEW
TESTVAR1=999
TESTVAR2=123456
TESTVAR3=98765432
```

Now enter the memory analysis utility program in Listing 1 (you may want to save it for future investigations). As an initial objective, let's try to find the following:

- Where the BASIC statements are stored
- Where variable names are stored
- Where variable values are stored

Let's start our search by seeing if we can find where the actual lines of the program are stored in memory. To do that, we *RUN* the memory analysis utility program, and request that it find the character string in the first REM statement (Line 10). To do that specify "S" for function required and enter the characer search mode by responding with a "C". Then enter the character string

"MEMORY ANALYSIS UTILITY." Be sure to request the dump in decimal this time. After the appropriate pause, a match should be found at address 2264 and you should see the first lines of comment.

At this point it should be explained that the article assumes throughout that you have a system without disk. For those of you with disk systems most of the addresses will be different, and there may be some variation in some of the commands, but the fundamental concepts remain the same. If you have trouble reproducing these results with a cassette system, it probably is because of differences in the sequence in which the program was entered, or errors in variable names. To resolve this you can do a *LIST "C*, a *NEW*, enter the variables again in direct mode, and do an *ENTER "C*.

Examining this more carefully, you will note that there are a few bytes in between the comments of each of the REM lines. After some study, you may note that the line numbers appear to start five bytes before each comment, at addresses 2259, 2288, etc. At this point you may wish to request another search, again with a decimal dump, looking for the character string "A DUMMY LINE" as listed in Line 256. The search will find a match at address 2847, and you will find that the value at address 2342 is now zero, but the next byte now has a value of one, where it previously was always zero. In fact the line number occupies two bytes, with the low order byte containing the low order bits, and the higher byte containing the high order eight bits. Thus the line number is 256 times the second byte plus the first byte, or 256*1+0=256. All binary 16-bit numbers in the Atari (and most 6502 processors) are stored in this fashion, including addresses. You may want to study lines 650 through 700 of Listing 1 to see how this type of number is manipulated.

To understand a little more of how this structure is laid out, try adding the following line to Listing 1.

```
1 REM
```

Now request the dump function starting at address 2259. You will see that we now have Line Number one, followed by five bytes, and then Line Number 10. Looking at the Line one dump, we see the first two bytes represent the line number, while the next two bytes contain the value six. Byte number five contains a zero, and byte number six contains a 155, which from Apendix C of the *Atari Basic Reference Manual* is a RETURN or EOL character. You will

note that the rest of the REM statements follow a similar format.

In fact we can now deduce that the third byte gives the length of the lines in bytes and by adding that to the address of the present line, we can find the next line. (Let's reserve study of the fourth byte until later). Similarly we can deduce that the fifth byte contains the equivalent of an opcode for the REM statement, while the EOL character signifies the end of the character string following the REM. This also conforms to the information in Chapter 11 of the *BASIC Reference Manual* under Item 2, where it states that each logical line requires six bytes of overhead.

With these facts in hand, let's leave the subject of BASIC statements for a moment, and see what we can observe about the other things we want to find.

Note that the second and third items are alluded to in the *BASIC Reference Manual* in Chapter 11, Item 3. The statement is made that a variable takes eight bytes plus the number of characters in the variable name the first time it is used, but that each subsequent reference takes only one byte. Thus the variable name and value cannot be stored in the BASIC statement.

Let's start the search for variable names by looking for the variable TESTVAR1 that we entered before we keyed in Listing 1. After typing *RUN*, specify a string search for the characters "TESTVAR." With an appropriate wait for the computer to find it, it should respond with an address of 2048 (decimal), and a dump of the surrounding area.

Examining the dump received, you will see the characters TESTVAR1 starting at the indicated address. However, note that the last character is in inverse video, or more precisely, that the high bit of the last character in the name has been set to a one. Following TESTVAR1, you will see the variable names TESTVAR2 and TESTVAR3, each with the last character in inverse video. You will also see the variables used in the program displayed in the same manner, each with the last character in inverse video.

Now specify an address pointer search for the address where the variable name table was found (2048). In this case several will probably be found, but the one of interest is the one found on memory Page 0 at address 130 and 131. (For those of you not familiar with the 6502 architecture and the significance of Page 0, you may want to refer to one of the excellent references on this subject.) One more problem with the variable name table remains. Since it is of variable length, depending on how many variables have been defined, and the length of each variable name, how do

we know where the table ends?

A little deductive reasoning is in order. Remember that variables can only contain alphanumeric characters. Thus any non-alphanumeric character could be used as a flag for the end of the variable table. Looking at a dump starting at 2048, sure enough after the variable BYTE0 we see the value 0 (address 2122). Now doing an address pointer search for address 2122, we find such a pointer at 132 and 133 on memory Page 0. We can also do a search for an address pointer to the beginning of the program lines by specifying a search for an address pointer to address 2259 where we found the first line of the program. Again a reference will be found on Page 0, this time at address 136 and 137.

Let's review what we found so far. We have a variable name table stored from address 2048 to 2122, with a pointer to the beginning of the table stored at addresses 130 and 131, and a pointer to the end of the table at 132 and 133. We also have the program lines stored beginning at address 2259, and an address pointer at 136 and 137. So what do you suppose is stored in between the end of the variable name table and the beginning of the program lines?

To find out, let's do a dump starting with the byte after the end of the variable name table, or address 2123, in decimal. After doing so, nothing much jumps out at you — right! So let's try a dump in hex starting at the same address. This time, with some study you will find in order the hex characters 09 99, 12 34 56, and 98 76 54 32 interspersed with other data. Looks like we may have found the variable value table, doesn't it?

Let's study this dump a little closer. Looking at the other bytes, and remembering what Chapter 11 said about 8 bytes per variable, study the value of TESTVAR1. What you should see is:

00 00 41 09 99 00 00 00

Similarly for TESTVAR2 and TESTVAR3 we see:

00 01 42 12 34 56 00 00 and
00 02 43 98 76 54 32 00

Thus the structure of the variable value is such that it is stored in binary coded decimal (BCD) as a floating point number. The digits are stored left-justified in bytes four through eight of the 8-byte block, with the exponent stored in byte three. The exponent is defined such that for numbers greater than one, the exponent is from hex 40 to hex 7F, while for numbers less than one it will have a value from 00 to 3F. For negative numbers the high order bit will

be set to one, or the exponent will range from 80 to FF. At this point you may want to end the dump program, change line 50 to assign a different set of values to the three variables, and then run a dump of this same area to see the changes.

Now that you have convinced yourself of the way numbers are stored, we still have a mystery or two to solve. What about byte two? Suppose that might be the variable number? Remember the statement in Chapter 11 about how additional references of a variable only take one byte. Seems that the only way to do that would be to assign a variable number. Also note that you are allowed a maximum of 127 different variables in a given BASIC program (see Chapter 1 of the *Reference Manual).* So the deduction that byte two of the 8-byte block is the variable number seems logical. Furthermore it gives a method of finding the variable name for such purposes as listing the program or operating in the direct mode.

Let's leave the use of the high order bit of byte two and the use of byte one of the 8-byte block to your investigation, with a couple of hints. Try examining the variables A$, B$ and HEX$. You may also want to define a numeric array in the direct mode and assign a set of values to it, and then dump its 8-byte block. One final step in this investigation is to try to find an address pointer to the variable value table. Specify a pointer to the address 2123, and we find that such an address pointer exists at 134 and 135 on Page 0 of memory.

Let's stop and summarize what we have learned at this point. FIGURE 1 is a visual depiction of the layout in memory of the address pointers on memory Page 0, the variable name table, the variable value table, and the program storage area.

At this point let's set our objective to create a full featured renumber utility. We have the fundamental information regarding memory layout and usage. The only additional data needed is to determine how line numbers are used in a program line. To investigate this, LISTING 2 has been developed. You can enter it at this point, either clearing the old program out, or leaving it at your option (if you have adequate memory).

The program in Listing 2 has been designed to let us dump a specific BASIC line. It will give us a decimal, hex, and character dump of any line we want. To digress for a moment, what we will get is a picture of the tokenized version of the BASIC line. This is the form used to store a program in the save mode. The list mode on the other hand stores the program just as you see it when you do

a list to the screen or printer. Also note that a save operation will save the variable name table and the variable value table as well.

The intention is to decipher the internal structure of a BASIC line; since we want to generate a renumber utility, more specifically we want to see what those lines with line number references look like. Let's start with one of the most common line referencing statements, the GOTO. When the program in Listing 2 has been entered, add the line

```
10 GOTO 10
```

Then in direct mode type

```
GOTO 20000
```

Now request that the program find and dump Line 10. What you will see as a dump is:

DEC	10	0	13	13	10	14	64	16	0	0
HEX	0A	00	0D	0D	0A	0E	40	10	00	00
DEC	0	0	22							
HEX	00	00	16							

Now change Line 10 to read

```
10 GOTO 123456
```

and with another GOTO 20000, the dump will read:

DEC	10	0	13	13	10	14	66	18	52	86
HEX	0A	00	0D	0D	0A	0E	42	12	34	56
DEC	0	0	22							
HEX	00	00	16							

From the change that takes place, it is obvious that the referenced line number is stored in bytes seven through 12 of the line. Not only that, but also it is stored in exactly the same format as variable values are stored. You may want to try a few other values for the referenced line number to convince yourself.

We can also speculate that the opcode for the GOTO must be either byte five or byte six, or a combination of the two. Now let's see how BASIC lines with multiple statements are formatted. Again modify Line 10 as follows:

```
10 GOTO 999:GOTO 999:GOTO 999
```

and doing a GOTO 20000 we get the following dump:

DEC	10	0	33	13	10	14	65	9	153	0
HEX	0A	00	21	0D	0A	0E	41	09	99	00
DEC	0	0	20	23	10	14	65	9	153	0
HEX	00	00	14	17	0A	0E	41	09	99	00

DEC	0	0	20	33	10	14	65	9	153	0
HEX	00	00	14	21	0A	0E	41	09	99	00
DEC	0	0	22							
HEX	00	00	16							

From this we can conclude that bytes four, 13 and 23 are used to describe the length of a given statement in the line. More precisely, they are used to give the offset from the address of the line number to the next statement, and the last of these in a multi-statement line will always be the same as byte three of the line.

At this point we need to establish what statements use line number references. After studying the BASIC Reference Manual, the following types of statements can have a line number reference:

GOTO GOSUB ON ( ) GOTO ON ( ) GOSUB TRAP
LIST RESTORE IF ( ) THEN IF ( ) THEN GOTO
IF ( ) THEN GOSUB

Taking each of these statements in order (entering the line number as shown, and then dumping it) we get the following results:

1 GOTO 999

DEC	1	0	13	13	10	14	65	9	153	0
HEX	01	00	0D	0D	0A	0E	41	09	00	00
DEC	0	0	22							
HEX	00	00	16							

2 GOSUB 999

DEC	2	0	13	13	12	13	65	9	153	0
HEX	02	00	0D	0D	0C	0E	41	09	99	00
DEC	0	0	22							
HEX	00	00	16							

3 ON Z GOTO 997, 998, 999

DEC	3	0	31	31	30	133	23	14	65	9
HEX	03	00	1F	1F	1E	85	17	0E	41	09
DEC	151	0	0	0	18	14	65	9	152	0
HEX	97	00	00	00	12	0E	41	09	98	00
DEC	0	0	18	14	65	9	153	0	0	0
HEX	00	00	12	0E	41	09	99	00	00	00

4 ON Z GOSUB 997, 998, 999

DEC	4	0	31	31	30	133	24	14	65	9
HEX	04	00	1F	1F	1E	85	18	0E	41	09
DEC	151	0	0	0	18	14	65	9	152	0
HEX	97	00	00	00	12	0E	41	09	98	00
DEC	0	0	18	14	65	9	153	0	0	0
HEX	00	00	12	0E	41	09	99	00	00	00

5 TRAP 999

DEC	5	0	13	13	13	14	65	9	153	0
HEX	05	00	0D	0D	0D	0E	41	09	99	00
DEC	0	0	22							
HEX	00	00	16							

6 LIST 999										
DEC	6	0	13	13	4	14	65	9	153	0
HEX	06	00	0D	0D	04	0E	41	09	99	00
DEC	0	0	22							
HEX	00	00	16							
7 RESTORE 999										
DEC	7	0	13	13	35	14	65	9	153	0
HEX	06	00	0D	0D	04	0E	41	09	99	00
DEC	0	0	22							
HEX	00	00	16							
8 IF Z THEN 999										
DEC	8	0	15	15	7	133	27	14	65	9
HEX	08	00	0F	0F	07	85	1B	0E	41	09
DEC	153	0	0	0	22					
HEX	99	00	00	00	16					
9 IF Z THEN GOTO 999										
DEC	9	0	17	7	7	133	27	17	10	14
HEX	09	00	11	07	07	85	1B	11	0A	0E
DEC	65	9	153	0	0	0	22			
HEX	41	09	99	00	00	00	16			
10 IF Z THEN GOSUB 999										
DEC	10	0	17	7	7	133	27	17	12	14
HEX	0A	00	11	07	07	85	1B	11	0A	0E
DEC	65	9	153	0	0	0	22			
HEX	41	09	99	00	00	00	16			

From these dumps we now deduce that all line number references are preceded by a byte having the decimal value 14. Furthermore, the byte preceding the byte with a value of 14 will have one of the following values if a line number reference follows:

OPCODE	STATEMENT
4	LIST
10	GOTO
12	GOSUB
13	TRAP
18	ON ( ) 2nd, 3rd, etc. line references
23	ON ( ) 1st line reference
24	ON ( ) GOSUB 1st line reference
27	IF ( ) THEN
35	RESTORE

In fact, it appears that the actual usage of the value 14 in a BASIC statement is to indicate that a BCD floating point constant follows. To see this, you may want to reload the program in Listing 1 and search for the decimal value 14. You should find that any occurrences in the program storage area, aside from line or statement lengths, precede a numeric constant.

With this information in hand, we now know enough to construct a Renumber utility. The basic algorithm is as follows:

1 — Find each line number reference

2 — Find the line that is referenced, and count the number of lines from the beginning
3 — Compute what the new line number will be
4 — Store that value as the new referenced line number
5 — When all line references have been set to their new value then do the actual renumbering of lines.

There remains a sticky implementation problem, since line numbers are stored as floating point numbers. (Why this approach was chosen by Atari remains a mystery — a binary format would have required two bytes instead of six, and no internal conversion.) Listing 3 demonstrates one technique for solving this problem, using the variable value table we found earlier. In this case, the location of the value for a specific variable (REFLINE) is established. That varible is used to store the new referenced line number when it is computed. Then that value is POKEd into the location for the line number reference.

Other more elegant solutions, requiring fewer statements, are possible, but they generally require some additional exploration of the structure of BASIC. At this point you will probably want to study Listing 3 along with its comments, and then enter it into your Atari. You should also note that this implementation of a renumber utility is not capable of renumbering itself. One other limitation is that the program will not deal with situations where variables are used as the line number reference. In such cases, you will have to follow the computational routines used to set the value of the line number reference, and either alter them appropriately, or else restore those line numbers to their original value after renumber has done its thing.

So how is such a program used? After the program has been entered, ready the tape recorder and, in the direct mode, type;

```
LIST "C
```

This will store the renumber utility on tape in a form so that it can be merged with other programs already in memory. (A CSAVE would be advisable, just for backup purposes.) First CLOAD a program you want to test the utility on. When that has finished, position the tape at the location where you started the List "C, and type:

```
ENTER "C
```

When the renumber utility has been loaded, a list command will show that it has been merged in at the end of the program previously loaded.

Now type GOTO 3200 and watch the results. One more step of course, is saving the program once it has been renumbered. If you simply do a CSAVE, you will also store the renumber utility with your original program. To avoid doing that, (gobbling up all that precious memory, not to mention space on your tape) do the following:

LIST "C1",0,31999

Rewind the tape to where the list started and

ENTER "C

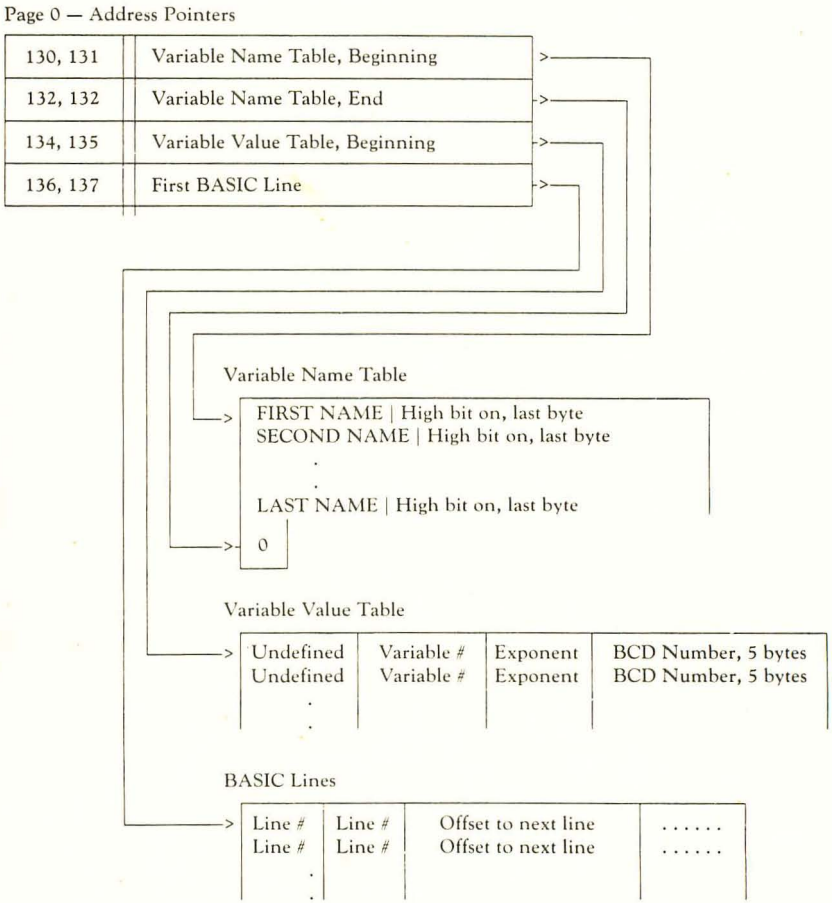
You now have just the original program in its renumbered form, and it can be CSAVEd in the conventional manner.

We have been able to develop a utility to renumber BASIC programs using the information we have uncovered. We have also found several techniques for conserving memory, such as not using the IF THEN GOTO statement, as it uses two more bytes than IF THEN. Using a variable will also save over using a constant if it is used more than twice. And, of course, every statement put into a multiple statement line saves three bytes. There are several other functions that could be implemented: such as changing variable names; finding all references to a given variable; the deletion of blocks of lines; and renumbering selected lines of a program. Some of these ideas require additional digging to find all of the data necessary; others can be implemented with the things we know at this point.

Two problems exist at this point. The first is that utilizes such as that in Listing 3 require a good deal of memory — a precious commodity for most of us. The second is that, for programs of any significant size, the use of such a utility will take a considerable period of time. A future article will take what has been developed to date and convert some of the more complex functions to machine language subroutines. These subroutines will be general purpose in nature, so that they can also be used in implementing some of the functions in the previous paragraph. Happy PEEKing!

FIGURE 1

Memory Layout for
Atari Basic Tables



Program 1

```
10 REM MEMORY ANALYSIS UTILITY
20 REM by W. A. Bell  May 1981
30 REM * Englewood, Colorado *
40 DIM A$(100),B$(1),HEX$(16)
50 HEX$="0123456789ABCDEF"
60 TESTVAR1=999:TESTVAR2=123456:TESTVAR3=98765432
70 PRINT CHR$(125)
90 PRINT "          MEMORY ANALYSIS UTILITY"
100 PRINT "ENTER  S  FOR DATA SEARCH"
110 PRINT "       D  FOR MEMORY DUMP"
120 PRINT "       A  FOR ADDRESS POINTER SEARCH"
130 PRINT "       E  TO END"
140 INPUT B$
150 IF B$="S" THEN 210
160 IF B$="D" THEN 770
170 IF B$="A" THEN 630
180 IF B$="E" THEN END
190 GOSUB 1270:GOTO 100
210 PRINT "ENTER  C  FOR CHARACTER DATA"
220 PRINT "       D  FOR DECIMAL DATA"
230 INPUT B$
240 IF B$="C" THEN 360
250 IF B$="D" THEN 270
256 REM A DUMMY LINE
260 GOSUB 1270:GOTO 210
270 PRINT "ENTER DECIMAL DATA TO SEARCH FOR"
280 PRINT "IN THE FORM  D1,D2,...,Dn"
290 PRINT "END WITH A VALUE OF 999"
300 ALENGTH=0
310 INPUT I
320 IF I>255 THEN 390
330 ALENGTH=ALENGTH+1
340 A$(ALENGTH,ALENGTH)=CHR$(I)
350 GOTO 310
360 PRINT "ENTER CHAR STRING TO SEARCH FOR"
370 INPUT A$
380 ALENGTH=LEN(A$)
390 GOSUB 1200
400 GOSUB 1170
410 POKE 1408,0
420 FOR I=0 TO 4000
430 IF PEEK(I)<>ASC(A$(1,1)) THEN 590
440 IF ALENGTH<2 THEN 490
450 FOR K=2 TO ALENGTH
460 IF PEEK(I+K-1)<>ASC(A$(K,K)) THEN 590
470 NEXT K
490 PRINT CHR$(125);"MATCH AT ADDRESS = ";I
500 POSITION 28,0:PRINT CHR$(138);CHR$(136)
510 PRINT "DUMP STARTS AT ";I-7
520 FOR K=I-7 TO I+83 STEP 10
530 GOSUB 920
540 NEXT K
550 PRINT "ENTER  C  TO CONTINUE SEARCH"
560 PRINT "       RETURN  TO QUIT";
570 INPUT B$
580 IF B$<>"C" THEN 90
590 NEXT I
```

```
600 PRINT "__ DATA NOT FOUND __"
610 GOTO 90
630 PRINT "ENTER ADDRESS POINTER TO SEARCH FOR"
640 INPUT ADDRESS
650 K=0:BYTE1=INT(ADDRESS/256)
660 BYTE0=ADDRESS-256*BYTE1
670 GOSUB 1170
680 FOR I=0 TO 4000
690 IF PEEK(I)<>BYTE0 THEN 730
700 IF PEEK(I+1)<>BYTE1 TEHN 730
710 PRINT "POINTER MATCH AT ";I;",";I+1
720 PRINT "LOOKING FOR OTHERS"
730 NEXT I
740 PRINT "NONE FOUND"
750 GOTO 90
770 PRINT "ENTER STARTING ADDRESS FOR DUMP"
780 INPUT ADDRESS
790 GOSUB 1200
800 PRINT CHR$(125);"DUMP STARTS AT ";ADDRESS
810 FOR K=ADDRESS TO ADDRESS+90 STEP 10
820 GOSUB 920
830 NEXT K
840 PRINT "ENTER  C  TO CONTINUE DUMP"
850 PRINT "       RETURN  TO QUIT";
860 INPUT B$
870 IF B$<>"C" THEN 90
880 ADDRESS=ADDRESS+91
890 GOTO 800
900 END
920 IF DUMP=0 THEN PRINT "DEC  ";:GOTO 940
930 PRINT "HEX  ";
940 FOR J=0 TO 9
950 DEC=PEEK(K+J)
960 IF DUMP=0 THEN 1020
980 HEX1=INT(DEC/16):HEX0=DEC-16*HEX1
990 PRINT HEX$(HEX1+1,HEX1+1);HEX$(HEX0+1,HEX0+1);" ";
1000 GOTO 1050
1020 IF DEC<10 THEN PRINT DEC;"  ";:GOTO 1050
1030 IF DEC<100 THEN PRINT DEC;" ";:GOTO 1050
1040 PRINT DEC;
1050 NEXT J
1060 PRINT :PRINT "CHAR ";
1070 FOR J=0 TO 9
1080 DEC=PEEK(K+J)
1100 IF (DEC>26 AND DEC<32) OR (DEC>124 AND DEC<128)
     THEN PRINT "_  ";:GOTO 1130
1110 IF (DEC>154 AND DEC<160) OR DEC>252
     THEN PRINT "_  ";:GOTO 1130
1120 PRINT CHR$(DEC);"  ";
1130 NEXT J
1140 PRINT
1150 RETURN
1170 PRINT "patience - this may take a while"
1180 RETURN
1200 PRINT "ENTER  H  FOR HEX DUMP"
1210 PRINT "       D  FOR DECIMAL DUMP"
1220 INPUT B$
1230 IF B$="H" THEN DUMP=1:RETURN
1240 IF B$="D" THEN DUMP=0:RETURN
1250 GOSUB 1270:GOTO 1200
```

```
1270 PRINT " *** INPUT ERROR *** "
1280 RETURN
```

Comments for Program 1.

General	The underscore (_) is used to indicate that characters are to be entered in inverse video

Lines	Comments
60	Required since a RUN command resets all variables to zero
90-190	Determine the function to be performed
210-610	Search memory for specified data
210-260	Determine if data input as character or decimal
270-350	Input of decimal data
360-380	Input of character data
410	Required to prevent match on BASIC input buffer
420-590	Actual search of memory
490-540	Match was found, dump memory at that point
630-750	Search for an address pointer
650-660	Convert to internal address format
680-730	Conduct the search, noting that addresses are stored low order byte, then high order byte
770-890	Dump specified area of memory
810-830	Dump a full screen of memory
920-1280	Subroutines
920-1150	Subroutine to dump memory
950-1050	Dump one line (10 bytes) in hex or decimal
980-1000	Hex dump after converting to hex
1020-1040	Decimal dump with appropriate spacing
1050-1130	One line of character dump for same memory
1100-1110	Check for cursor control characters and substitute inverse video space
1170-1180	Subroutine to print patience message
1200-1250	Subroutine to determine if dump is in hex or decimal
1270-1280	Subroutine for input error

Program 2.

```
20000 REM BASIC LINE DUMP UTILITY
20100 REM by W. A. Bell  May 1981
20200 REM * Englewood, Colorado *
20300 CLR :DIM HEX$(16),Q$(1)
20400 HEX$="0123456789ABCDEF"
20500 LINEADR=PEEK(136)+256*PEEK(137)
20600 PRINT "ENTER LINE NUMBER TO BE DUMPED";:INPUT LINENUM
20700 THISLINE=PEEK(LINEADR)+256*PEEK(LINEADR+1)
20800 IF THISLINE>LINENUM THEN PRINT "LINE DOESN'T EXIST -
      TRY ANOTHER":GOTO 20500
20900 IF THISLINE=LINENUM THEN 21300
21000 LINEADR=LINEADR+PEEK(LINEADR+2)
21100 GOTO 20700
21300 PRINT CHR$(125);"LINE # ";THISLINE;" START ADDRESS = "
      ;LINEADR
21400 LIST LINENUM
21500 Z=10:Y=5:MAXADR=LINEADR+PEEK(LINEADR+2)
21600 IF LINEADR+Z>MAXADR THEN Z=MAXADR-LINEADR
21700 FOR I=0 TO Z-1
```

```
21800 POSITION 2,Y
21900 PRINT "DEC"
22000 PRINT "HEX"
22100 PRINT "CHAR"
22400 Q=PEEK(LINEADR+I):X=3*I+8
22500 Q$=CHR$(Q)
22600 HEX1=INT(Q/16):HEX0=Q-16*HEX1
22700 POSITION X,Y
22800 PRINT Q
22900 POSITION X,Y+1
23000 PRINT HEX$(HEX1+1,HEX1+1);HEX$(HEX0+1,HEX0+1)
23100 POSITION X,Y+2
23200 PRINT Q$
23300 NEXT I
23400 LINEADR=LINEADR+Z:Y=Y+3
23500 IF Y<21 THEN 23900
23600 PRINT "ENTER  RETURN  FOR NEXT PAGE";
23700 INPUT Q$
23800 PRINT CHR$(125);"DUMP CONTINUES AT ADDRESS ";
      LINEADR:Y=2
23900 IF LINEADR<MAXADR THEN 21600
24000 PRINT "ENTER  C  TO DUMP MORE LINES"
24100 PRINT "       RETURN  TO QUIT";
24200 INPUT Q$
24300 IF Q$="C" THEN 20500
24400 END
```

Comments for Program 2.

General The underscore (_) is used to indicate that characters are to be entered in inverse video

Lines	Comments
20400	Constants used in hex conversion
20500-21100	Find the line the dump was requested for
20500	Find starting address of first line
20700	Compute line number of current line
21000	Compute address of next line
21300-21400	Set up to dump line
21500-23500	Dump one screen of memory
21500	Z is how many bytes to dump on this line Y is vertical position on screen MAXADR is start of next line
21700-23300	Dump Z bytes of memory
22700-22800	Dump byte in decimal
22900-23000	Dump byte in hex
23100-23200	Print character representation of byte - using POSITION avoids most of the problems with cursor movement except clear screen (Q=125)
23500	Test for full screen of dump
23600-23800	For lines that exceed a full screen
23900	Check for end of line

Program 3.

```
32000 CLR :PRINT CHR$(125);"        RENUMBER UTILITY "
32005 REM by W. A. Bell  May 1981
```

```
32010 REM * Englewood, Colorado *
32015 DIM OPCODE(10),REFNAME$(7)
32025 REFNAME$="REFLINE"
32030 REFADR=PEEK(130)+256*PEEK(131)
32035 REFCOUNT=0
32045 FOR I=1 TO 7
32050 IF PEEK(REFADR+I-1)<>ASC(REFNAME$(I,I)) THEN 32090
32055 NEXT I
32070 REFADR=PEEK(134)+256*PEEK(135)+8*REFCOUNT
32080 GOTO 32120
32090 REFCOUNT=REFCOUNT+1
32095 IF REFCOUNT>127 THEN PRINT "FATAL PROGRAM ERROR":END
32100 REFADR=REFADR+1
32105 IF PEEK(REFADR-1)>127 THEN 32045
32110 GOTO 32100
32120 RESTORE 32125
32125 DATA 10,12,23,24,13,4,35,27,18
32130 FOR I=1 TO 9
32135 READ X
32140 OPCODE(I)=X
32145 NEXT I
32150 STARTADR=PEEK(136)+256*PEEK(137)
32160 LINEADR=STARTADR:REFLINE=0
32165 OLDLINE=-1:LINECOUNT=0
32170 NULINE=PEEK(LINEADR)+256*PEEK(LINEADR+1)
32175 IF NULINE>31999 THEN 32220
32180 IF OLDLINE<NULINE THEN 32200
32185 PRINT "SEQUENCE ERROR AFTER ";OLDLINE
32190 LIST OLDLINE-1,OLDLINE+10
32195 END
32200 LINEADR=LINEADR+PEEK(LINEADR+2)
32205 LINECOUNT=LINECOUNT+1
32210 OLDLINE=NULINE
32215 GOTO 32170
32220 PRINT "LINE SEQUENCE VALID"
32225 PRINT LINECOUNT;" LINES"
32235 PRINT "ENTER START, INCREMENT";:INPUT BASE,INCR
32240 IF BASE+INCR*LINECOUNT<32000 THEN 32255
32245 PRINT "INPUT ERROR - WILL EXCEED MAXIMUM LINE
      NUMBER ALLOWED":END
32255 LINEADR=STARTADR
32260 NULINE=PEEK(LINEADR)+256*PEEK(LINEADR+1)
32265 IF NULINE>31999 THEN 32470
32270 LINEEND=LINEADR+PEEK(LINEADR+2)
32275 STMTSTART=LINEADR+4
32280 STMTEND=LINEADR+PEEK(LINEADR+3)
32285 FOR I=STMTSTART TO STMTEND-1
32290 IF PEEK(I)<>14 THEN 32430
32300 FOR J=1 TO 9
32305 IF PEEK(I-1)=OPCODE(J) THEN 32325
32310 NEXT J
32315 GOTO 32430
32325 FOR K=1 TO 6
32330 POKE REFADR+K+1,PEEK(I+K)
32335 NEXT K
32340 PRINT "LINE ";NULINE;" REFERENCES LINE ";REFLINE
32345 IF REFLINE<32000 AND REFLINE>-1 AND REFLINE=INT(REFLINE)
      THEN 32355
32350 PRINT "GARBAGE LINE NUMBER":GOTO 32430
32355 OLDADR=STARTADR:REFCOUNT=0
```

```
32360 OLDLINE=PEEK(OLDADR)+256*PEEK(OLDADR+1)
32365 IF OLDLINE=REFLINE THEN 32410
32370 IF OLDLINE>REFLINE THEN 32390
32375 OLDADR=OLDADR+PEEK(OLDADR+2)
32380 REFCOUNT=REFCOUNT+1
32385 GOTO 32360
32390 PRINT "ERROR - REFERENCED LINE DOESN'T EXIST"
32395 LIST NULINE
32400 GOTO 32430
32410 REFLINE=BASE+INCR*REFCOUNT
32415 FOR K=1 TO 6
32420 POKE I+K,PEEK(REFADR+K+1)
32425 NEXT K
32430 NEXT I
32435 STMTSTART=STMTEND+1
32440 IF STMTSTART>LINEEND THEN 32455
32445 STMTEND=LINEADR+PEEK(STMTEND)
32450 GOTO 32285
32455 LINEADR=LINEADR+PEEK(LINEADR+2)
32460 GOTO 32260
32470 LINEADR=STARTADR
32475 FOR I=1 TO LINECOUNT
32480 BASE1=INT(BASE/256):BASE0=BASE-256*BASE1
32485 POKE LINEADR,BASE0
32490 POKE LINEADR+1,BASE1
32495 BASE=BASE+INCR
32500 LINEADR=LINEADR+PEEK(LINEADR+2)
32505 NEXT I
32510 PRINT "*** RENUMBER COMPLETE ***":END
```

Comments for Program 3.

General — The underscore (_) is used to indicate that characters are to be entered in inverse video

The program requires 2319 bytes of memory in this form. To conserve memory, a number of lines could be deleted, eliminating some displays and error checking. These lines should be considered: 32095, 32180 through 32195, 32220, 32225, 32240, 32245, 32340 through 32350, and 32510. Smaller gains can also be made by converting the computation of line addresses and line numbers to subroutines, and by using shorter variable names.

Lines	Comments
32025-32110	Find the address of the variable REFLINE, used to store the referenced line number
32030	Beginning of the variable name table
32045-32055	Is this the correct variable?
32070-32080	Yes, compute the address in the variable value table
32090-32110	No, search for the end of this variable (inverse video) and increment the variable number
32120-32165	Initialize other variables
32120-32145	Set up the array of opcodes which use line numbers
32170-32225	Count the number of lines and check to make sure they are in ascending order
32235-32245	Input the renumber parameters and check see if they will exceed the first line number of this program
32260-32460	Find each line number reference, and replace with the new line number

32260-32280	Compute address of line, line number, address of end of line, start of statement and end of statement
32285-32430	Process each BASIC statement in the line
32290	Test for a BCD constant
32300-32310	Check for line referencing opcode
32325-32335	Store referenced line number in variable REFLINE
32345	Check for nonsense line numbers (just in case)
32355-32385	Scan program to locate referenced line
32410-32425	Referenced line found so compute what the new line number will be and store in line
32435-32460	Check for end of line and update address pointers accordingly
32470-32505	Now compute the new line number for each line and store in the first two bytes of the line

Input/Output on the Atari

Larry Isaacs

Here is much that you will want to know about dealing with files. There is also an explanation of the XIO commands.

In this article, I will try to explain how to use the various BASIC commands at your disposal to communicate with the peripheral devices in your system. These peripheral devices include the Screen Editor (E:), keyboard (K:), and TV Monitor (S:), all of which are part of your machine. External devices which are currently available include disk drives (D1: through D4:), printer (P:), and cassette (C:). The I/O (Input/Output) commands we will be discussing are the PUT, GET, PRINT, INPUT, XIO5, XIO7, XIO9, and XIO11 commands. Also, the discussion will be limited to the use of these commands as it relates to logical files.

Before we get into details, there are two important facts to remember. The first one is that these I/O commands result in the transfer of one or more bytes of data, and that, often, these bytes will be ATASCII characters. The second fact is that the byte or bytes which get transferred will be the same regardless of the device with which you are communicating.

Open And Close

Before you can communicate with a peripheral device, it must first be "opened," and, in the case of the disk, a file name provided. The syntax of the open command is as follows:

```
OPEN #iocb,mode,0,"device:name.ext"
```

iocb - I/O Control Block through which BASIC will send its requests to the I/O software.

mode - This should be an arithmetic expression which evaluates to 4, 6, 8, 9, or 12. For now we will just be using 4, 8, and 12. Their meaning is as follows:

4 = open for reading
8 = open for writing
12 = open for reading and writing

device - This should be a letter which identifies which device to associate with the I/O Control Block specified previously.

name - This should be a name of up to eight alphanumeric characters, the first of which must be a letter.

ext - This is an extension to the name which is usually used to indicate the type of file, BASIC program, data, etc. It may include up to three alphanumeric characters. The name plus extension form the file name which is needed when communicating with the disk. Once you have opened a device, you communicate with that device using the "iocb" number. To close a device or file, you use the CLOSE command. The syntax for this command is as follows:

CLOSE #iocb

Only one device can be associated with an IOCB at a time. If you wish to associate a new drive with an IOCB that is currently in use, you must close the old device first. In the case of the disk, cassette, and the printer, a CLOSE command may be required for proper operation. For example, the disk can only write groups of 128 bytes, called sectors, which are written once enough data has been received to fill the sector. The CLOSE command is required to cause the last sector of a file, which is only partially filled with data, to be written to the disk. The cassette also needs a CLOSE command to write the last group of bytes. And since the printer doesn't print a line until an EOL (End of Line) character is received, a CLOSE may be needed to print out the last line.

If a program terminates without error, or via an END statement, all open devices and files will be closed automatically. If the program terminates because of an error, a STOP statement, or the BREAK key being struck, the devices and files will be left open. If you aren't able to continue the program, you may close the devices and files by entering the necessary CLOSE commands directly, i.e. without line numbers. Also, executing the RUN command will close any open devices or files.

Now we will begin our discussion of the I/O commands. Many of the examples make use of the disk. If you wish to use cassette instead, simply change the file specification in the OPEN commands to the cassette device. Just place a blank cassette in the cassette player. Then whenever you hear two beeps, rewind the tape, press PLAY and RECORD on the player, then hit RETURN on the ATARI. Whenever you near one beep, rewind the tape, press PLAY on the player, then hit RETURN on the ATARI.

Put And Get

Let's look first at the PUT and GET commands, which are the most basic of the I/O commands. These two commands result in the

transfer of a single byte, with the PUT command sending a byte, and the GET command receiving a byte. Here is the syntax for the commands:

GET #fn, variable where "fn" is a file number, and "variable" is a simple variable, not an array or string variable

PUT #fn, expression where "expression" is an arithmetic expression

Listing 1 provides an example for using the GET and PUT commands. In this program the Screen device is opened for reading and writing. This open command will also cause the screen to be cleared. The letters from "A" to "Z" are sent to the Screen using the PUT command and, after the cursor is repositioned, the letters are fetched back from the Screen using the GET command.

Program 1

```
10 DIM T$(30)
20 OPEN #1,12,0,"S:":REM OPEN FOR R/W
30 FOR I=ASC("A") TO ASC("Z")
40 PUT #1,I:NEXT I
50 POSITION PEEK(82),0
60 FOR I=1 TO 26:GET #1,CHARACTER
70 T$(I,I)=CHR$(CHARACTER)
80 NEXT I
90 GET #1,I:REM MOVE CURSOR PAST THE Z
100 CLOSE #1
110 PRINT :PRINT T$
```

Listing 2 provides a similar example which communicates with a disk. Note, if you run this program a second time, opening the file for writing will cause the old file to be deleted. Also, if you try to get more bytes than were written to the file, an ERROR 136 (End of File encountered) will be given. Changing the 26 to 27 in line 60 will illustrate this.

Program 2

```
10 OPEN #1,8,0,"D:TEST.DAT"
20 FOR I=ASC("A") TO ASC("Z")
30 PUT #1,I:NEXT I
```

```
40 CLOSE #1
50 OPEN #1,4,0,"D:TEST.DAT"
60 FOR I=1 TO 26
70 GET #1,CHARACTER
80 PRINT ;CHR$(CHARACTER);
90 NEXT I
100 CLOSE #1
```

The advantage of using GET and PUT is that you are controlling the transfer of individual bytes. If this isn't necessary, you will likely find it simpler and faster to use one of the following I/O commands. Each of these commands involves the transfer of a string of bytes.

Print And Input

The PRINT and INPUT commands are used to transfer a string of characters. The syntax of these commands is as follows:

PRINT #iocb; list where the "list" is a list of expressions separated by commas or semicolons. The expressions may be numbers, strings, simple variables or string variables. If a semicolon is used prior to an expression, the characters for this expression will be sent immediately following any previous characters. If a comma is used instead of a semicolon, including the one shown in the syntax, tabbing will occur before characters from the expression are sent. If the list doesn't end with a comma or semicolon, an EOL character will be sent at the end of the list. If you wish, the list need not contain any expressions.

INPUT #iocb, list where the "list" is a list of expressions separated by commas. The expressions may be simple variables or string variables.

When printing strings, naturally the characters in the string are sent. However, when you print a number, the number is converted to a string of digits and sent as ATASCII characters. When you input a string, characters will be fetched until an EOL character is received. These characters will be stored in the string's reserved memory until that is filled or the EOL character is received. When you input a number, characters will be fetched until an EOL character or a comma is received. At this point, assuming all the characters were valid digits, the string is converted back to a number.

Listing 3 provides an example of using PRINT and INPUT

with the Editor device. Like the Screen device, the Editor will print and fetch characters from the screen memory. However, when printing to the Editor, control characters will perform the associated function instead of printing a character. When you input from the Editor, RETURN must be hit before the Editor will begin sending characters. Also, the Editor remembers the line and column of the cursor when the input request is made. As long as you don't hit a cursor-up or cursor-down, the fetching of characters will begin with the first character of the new line which the cursor occupies. The fetching of characters will continue until the last nonblank character of the line occupied by the cursor when RETURN was hit. You can explore the operation of the Editor further by making changes to Listing 3, and finding out what happens.

Program 3

```
10 DIM T$(80)
20 OPEN #1,12,0,"E:"
30 PRINT #1;"123,CHARACTERS"
40 PRINT #1;CHR$(28);:REM AN UP-CURSOR
50 INPUT #1,T$
60 PRINT #1;CHR$(28);
70 INPUT #1,NUMBER
80 CLOSE #1
90 PRINT "T$=";T$
100 PRINT "NUMBER=";NUMBER
110 REM JUST HIT RETURN WHEN EACH
120 REM OF THE INPUT STATEMENTS
130 REM EXECUTES
```

Listing 4 gives an example of using PRINT with the disk. The program reads back the characters using the GET command so you can see what was sent to the disk by the PRINT command. Again, you can experiment with changes to this program to improve your understanding of how these commands operate.

Program 4

```
10 DIM T$(10)
20 T$="ABCDEFGH"
30 OPEN #1,8,0,"D:TEST.DAT"
```

```
40 PRINT #1;T$
50 CLOSE #1
60 OPEN #1,4,0,"D:TEST.DAT"
70 FOR I=0 TO 8
80 GET #1,A
90 ? A,CHR$(A)
100 NEXT I
110 CLOSE #1
```

XIO9 And XIO5

The XIO9 and XIO5 commands, like the PRINT and INPUT commands, send and receive a string of characters. The syntax for these commands is as follows:

XIO cmdn,#icob,mode,0,exp

"cmdn" is the XIO command number.
9 = PUT RECORD
5 = GET RECORD
"iocb" and "mode" have the same function as in the OPEN statement.
"exp" may be a string or string variable when writing, or a string variable when reading.
The XIO9, or PUT RECORD command will write characters from the specified string until an EOL character is written. If the string contains an EOL character, the XIO9 terminates at this point, and the rest of the string isn't written. If the string does not contain an EOL character, one is appended. This differs from the PRINT command where the entire string is written regardless of content. the program in Listing 5 illustrates this difference.

Program 5

```
10 DIM T$(10)
20 T$="ABCDEFGHIJ"
30 T$(5,5)=CHR$(155)
40 OPEN #1,8,0,"E:"
45 ? "PRINT DOES THIS"
50 PRINT #1;T$
55 ? :? "XIO DOES THIS"
60 XIO 9,#1,8,0,T$
70 CLOSE #1
```

The XIO5 command, like the INPUT command, will fetch one string and store it in memory. But where the INPUT command stops (when the memory reserved for the string variable is filled), the XIO5 command keeps going. This means that the XIO5 command can load more than one string variable. A second difference is that the INPUT command doesn't store the EOL character, where the XIO5 command does. And one last difference, the INPUT command will change the length of the string variable to the number of characters stored, where the XIO5 command doesn't change the length of any string variable. Before you can make productive use of the XIO5 command, there is one more necessary fact. Once the XIO5 command fills the first string variable to its current length, the next character fetched is apparently discarded and the next memory location is left unchanged. This applies only to the string variable specified in the command statement. The program in Listing 6 illustrates the preceding discussion.

Listing 6

```
10 DIM D$(1),T$(4),T1$(4),T2$(10)
20 OPEN #1,8,0,"D:TEST.DAT"
30 XIO 9,#1,8,0," ABCDEFGHIJK"
40 CLOSE #1
50 T1$="XXXXXXXXXX":REM RESET T1$
60 T1$="YYYYY":REM MAKE LENGTH 5
70 OPEN #1,4,0,"D:TEST.DAT"
80 INPUT #1,T2$
90 CLOSE #1
100 ? "INPUT DOES THIS"
110 ? T2$,LEN(T2$)
120 T2$="XXXXXXXXXX":REM RESET T2$
130 T2$="YYYYY":REM MAKE LENGTH 5
140 OPEN #1,4,0,"D:TEST.DAT"
150 XIO 5,#1,4,0,T2$
160 CLOSE #1
170 ? :? "XIO5 DOES THIS"
180 ? T2$,,LEN(T2$)
190 T2$(10,10)="Z"
200 ? T2$,LEN(T2$)
210 ? "NOTICE THE X ISN'T WRITTEM OVER"
220 T$="XXXX":T1$="XXXX":T2$="XXX"
```

```
230 OPEN #1,4,0,"D:TEST.DAT"
240 XIO 5,#1,4,0,D$
250 CLOSE #1
260 ? :? "OR XIO5 CAN DO THIS"
270 ? T$:? T1$:? T2$
```

XIO11 And XIO7

The XIO11 and XIO7 commands are used to write and read blocks of 255 bytes, respectively. The syntax for these commands is the same as for XIO9 and XIO5 except for the command number. The commands transfer bytes beginning with the reserved memory of the string variable specified in the command. Since you are transferring bytes, their content has no effect on the operation of the command. As with the XIO5 command, once the XIO7 command fills the current length of the first string variable, the next byte fetched isn't stored in memory.

Naturally the XIO11 and XIO7 commands could be used for handling strings of characters. However, if we knew where the address of a string's reserved memory was kept, we could make changes to it, and use these commands to save and restore any portion of memory we want. Fortunately this isn't too difficult. Each string variable will have an entry in the variable storage area, which contains 8 bytes of parameters for the variable. The third and fourth bytes of the parameters contain the displacement from the beginning of the array storage area to the reserved memory for that string. If we dimension a string variable in the first statement of a program, then this displacement can be found by PEEK(134) + (PEEK(135)*256) +2. Also, the address of the reserved memory for this string will be at the beginning of the array storage area. For more detail about this, see INSIDE ATARI BASIC and ATARI TAPE DATA FILES in COMPUTE # 4.

Listing 7 shows how to save an array to disk and then read the data from disk into a different array. In this program we direct the XIO11 command to save the desired portion of memory by POKEing the required displacement into the parameters of D$. We then read the data into a different array, which could have been in a different program, by again POKEing the necessary displacement into the parameters of D$. Note the use of the MARK strings and the ADR function to find where the arrays are in memory. Another application might be to add some machine language routines to a program by reading them from disk or cassette and storing them in

the required location in memory.

Listing 7

```
10 DIM D$(1):REM DUMMY STRING
15 REM FIND ADDRESS OF DISPLACEMENT
20 ADDR=(PEEK(134)+PEEK(135)%256)+2
25 REM FIND BEGINNING OF ARRAY STORAGE
30 BOA=ADR(D$)
40 DIM MARK1$(1),ARRAY1(80),X1$(23)
45 REM WITH 6 BYTES/ARRAY NUMBER,
46 REM THIS DIMENSIONS 510 BYTES
47 REM OR 2%255
48 REM NOW FILL THE ARRAY
50 FOR I=0 TO 80:ARRAY1(I)=I:NEXT I
60 OPEN #1,8,0,"D:TEST.DAT"
65 REM NOW WRITE THE ARRAY IN 2 BLOCKS
70 FOR N=0 TO 1
80 TMP=ADR(MARK1$)
90 GOSUB 1000
110 XIO 11,#1,8,0,D$
120 NEXT N
130 CLOSE #1
140 DIM MARK2$(1),ARRAY2(80),X2$(23)
150 OPEN #1,4,0,"D:TEST.DAT"
155 REM NOW READ THE ARRAY
160 FOR N=0 TO 1
170 TMP=ADR(MARK2$)
180 GOSUB 1000
190 XIO 7,#1,4,0,D$
200 NEXT N
210 CLOSE #1
220 FOR I=0 TO 80 STEP 10
230 ? ARRAY2(I)
240 NEXT I
250 END
900 REM SUBROUTINE TO FIX THE
905 REM DISPLACEMENT- N=BLOCK NUMBER
1000 TMP=TMP-BOA+(N%255)
1010 POKE ADDR,TMP-INT(TMP/256)%256
1020 POKE ADDR+1,INT(TMP/256)
1030 RETURN
```

This concludes the explanations of the various I/O commands. I hope I have explained them well enough for you to put them to productive use. Some of the explanations are fairly brief, so to find out more, or to better understand their operation, I highly recommend that you do some experimenting of your own. This is the best way to find out what the commands will do in specific situations.

Why Machine Language?

Jim Butterfield

Here is an overview of several important aspects of machine language.

BASIC programmers soon discover that their machines have an "inner code." Somewhere inside, there seems to be another language that is very fast, powerful, and compact. Yet there seems to be no easy way to gain access to this feature; it's not easy to learn, and seems to be bound up with a special kind of machine jargon.

BASIC people often stand in awe of the machine language "gurus." They might be surprised to find that many machine language programmers find BASIC an intimidating language. Such people often find BASIC to be complex, poorly defined, and riddled with obscure syntax rules. Many KIM, SYM and AIM owners are quite nervous when they first meet BASIC — it's such a departure from the precise and (to them) simple machine language that they have learned.

Each language has its own advantages and disadvantages; neither is "better." BASIC is particularly good for scientific and business calculations, especially where a program is changed frequently. Machine language is used where speed is vital; it tends to be used in mechanical environments, especially for hardware interfaces. BASIC programmers tend to be data-oriented, and concentrate their efforts on getting information in and out. Machine language programmers like to work with the innards of the machine, and spend much of their time tinkering with the mechanics. There's room for both types of activity.

Let's compare BASIC and Machine Language to get an idea where each has advantages. Nothing in the following list is absolute: sometimes BASIC can be as fast as machine language; sometimes machine language can be as fast to code as BASIC. But the comparisons are generally valid.

BASIC is easier to write and get working. You have a freedom to change a line, insert new coding, and check out a program that can't be matched in Machine Language.

BASIC is easer to read. Its English-like syntax makes it relatively easy to pick up a program and see what it does. A dozen lines of BASIC might require a hundred lines of machine language (or more) to do the same job.

BASIC has splendid built-in capabilities. INPUT and PRINT are very powerful; in machine language you'd need to program the same capabilities the hard way. Other features, such as the way BASIC handles variables, strings, and arrays call for a lot of machine language coding.

BASIC usually uses less memory space. Surprise! Except for very small programs, machine language will gobble up more memory.

Machine language is fast. It's not uncommon for machine language programs to run ten or more times as fast as similar BASIC programs. Keep in mind, of course, that input and output of data will be geared to the speed of the external device you are working with; machine language won't get input from the keyboard any faster than BASIC.

Machine language can get at inner mechanisms that BASIC can't reach. BASIC is much more portable between different machines.

So what do these comparisons tell us?

First, if BASIC can do a job, and can do it fast enough, always use the BASIC. You'll write the program faster, and it will be easier to change in the future.

But if you have a speed problem, or if there's something you need to do that's beyond the capability of BASIC, then use machine language. Remember that with machine language you will lose flexibility and portability. But if that's what you need to do the job, use it.

There are other reasons why it's good to know machine language. It gives you a glimpse of the inner secrets of your computer. Even BASIC itself is just a huge machine language program stored in ROM. Each BASIC statement is executed by dozens of tiny machine language instructions which decide what is wanted and then perform the task. If you wanted to know precisely how a BASIC statement worked, you would ultimately have to trace through the machine language that did the job.

It's probably best to think of BASIC and machine language as complementary tools. You can and should use them together. BASIC can call in a machine language program when it needs it by using a SYS command or a USR function. The machine language code can return to BASIC when it's finished the job by using the RTS code. Data can be passed back and forth between the two languages.

The result: you can have the best of both worlds. The main part of your program will be in BASIC so that you can code it quickly and easily. The tricky bits, where you need speed or special functions, will be in relatively short machine language programs.

Machine language is picky and exacting. It doesn't allow you many mistakes. If you're an impetuous programmer, you might be happier to stay with BASIC, which is very lenient towards your mistakes. But if you're ready to take the time, and plot, scheme, plan, code, check, test and review — you can do some remarkable things with machine language.

It takes precision and patience. But there's nothing to compare with the rush you get when your machine language program finally works the way you planned it.

POKin' Around

Charles Brannon

Perhaps one of the most useful commands in BASIC is POKE. Why? Because POKE allows you to do some things that cannot be done as easily in BASIC. I recall the earlier days of the PET, where every time a nifty memory location was discovered, it was published with glee — indeed, they were real "tidbits." Nowadays, however, there are several very good memory maps that document the inner workings of the PET quite well.

In the Atari Basic Reference Manual, there is an appendix entitled "Memory Locations" (Appendix I). Although it is not a true memory map since it is incomplete, it does list some very interesting locations.

During the execution of a program, the cursor does not disappear. Rather, it moves with the print statements and sometimes is left behind, cluttering up the screen with little white squares. Fortunately, the visibility of the cursor can be zeroed out with a simple statement: POKE 752,1. To restore the cursor, press the BREAK key or POKE 752,0. The well-known problem of the non-standard behavior of the Atari's GET statement has led to the discovery of memory location 764. Here is stored the code representing the last key pressed. This is not in ATASCII, but is a code used in the scanning of the keyboard. If no key has been pressed, a value of 255 will be found here. I first found this technique right here in COMPUTE!. In "Adding a Voice Track to Atari Programs," the author suggested using a subroutine like this to check if a key has been pressed:

```
lineno IF PEEK (764)=255 THEN lineno (same lineno)
lineno POKE 764,255: RETURN
```

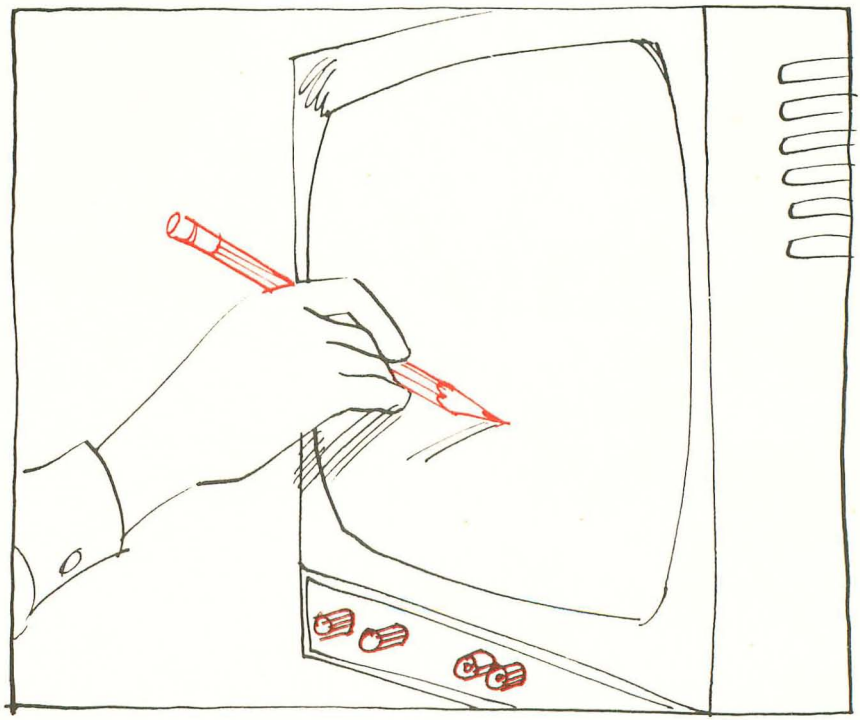
The first statement waits for a key to be pressed; the second discards that keypress by making BASIC think no key was pressed so that the keystroke would not be printed accidentally.

An example of how POKEing can be easier to use than a BASIC equivalent is in directly controlling the five color registers. After all, they too are only mere memory locations. Locations 708-712 correspond to SETCOLOR color registers 0-4. Using the notation *SETCOLOR aexp,aexp,aexp* where *aexp* is an arithmetical expression, the first number is from 0-4, so use the appropriate memory location. Then multiply *aexp* number two by 16 and add the third number to it. This gives you an integer in the range 0-255.

Now just enter POKE COLR, NUMBER where COLR is the memory location of the color register and NUMBER is that number you obtained. Figuring out what color is already being displayed is done in the reverse fashion. Get the contents of the color register with PEEK(COLR), and assign it to some variable, say X. (e.g. X=PEEK(COLR)). Divide X by sixteen, throw away the fraction using Y=INT(X), then find the luminance (aexp#3) with L=X-16*Y. Now you can set the color by basic with SETCOLOR COLR-708, Y,L or you can just store the numbers so you can meditate on them at a later date.

Have fun with these memory locations, you hackers! You beginners — step right up and add several new functions to your repertoire!

I want to leave you one more thing to try — POKE 755,6. It's weird! (You can get it back to normal with POKE 755,2 or by pressing RESET.



Printing to the Screen from Machine Language on the Atari

Larry Isaacs

If you use machine language you'll find this useful. There are two techniques presented here: screen output and a relocating loader.

If you are machine language programming on the ATARI, it can be very advantageous to know where some of the operating system subroutines can be found. I can provide you with only one at this time, but it's one of the handier ones. This is the output subroutine for the Editor device. It accepts the full ATASCII character set, printing the displayable character on the screen, or executing the control characters. To use the routine, simply load the character into the accumulator and execute a JSR $F6A4 instruction. The only other fact needed is that the X and Y registers aren't preserved by this subroutine.

To illustrate the use of this subroutine, the DUMP program is provided. This program also illustrates one way of using machine language with BASIC. The program asks for starting and ending addresses, which should be given in hex. Then the requested memory is dumped on the screen by a machine language program executed by the USR command.

Naturally, before the machine language can be executed, it must be placed in memory. This is done by the BASIC subroutine in statements 10200-10430. This subroutine loads machine code found in DATA statements, which begin at line 20000 in this program. The first thing the subroutine does is read the number of bytes in the machine language program. It then dimensions DMY$ to length 1 and an array called STORAGE of sufficient size to hold the machine code.

The subroutine then starts reading the data as strings and POKEing the appropriate code. If the string read doesn't start with a special character (".","*","+","=", or "!") then the string is assumed to be two hex characters which are stored in the next available byte. If the string begins with a ".", then the string is assumed to be a comment and is ignored. If it begins with an "*", the subroutine

assumes the rest of the string is four hex characters which form a two byte address. This address is POKE'd low byte first, then the high byte. If the string begins with a "+", the rest of the string is assumed to be four hex characters which form a two byte displacement from the beginning location of the code. This displacement is added to the beginning location of the code to form a two byte address. This address is also POKE'd low byte first, followed by high byte. If the first character is an "=", then the rest of the string is assumed to be a displacement as with "*". However, once the address is computed, the current poke location plus one is subtracted from this address to form a one byte displacement which is POKE'd into the next location. Finally, if the first character of the string is an "!", the subroutine stops loading machine code. The rest of the string is assumed to be a two byte displacement as with the "*", and the computed address is checked with the current poke location to see if it matches. If they don't match, it's likely that you've miscounted some bytes and that some of the displacements given by strings starting with the "*" or "=" character are in error.

This may seem somewhat complicated, but it really makes it fairly simple to write relocatable code. This relocatability is necessary because you don't know where the code will be loaded until the program is running. Relative addressed used by branch instructions may be given as a hex byte or as an "=" followed by the displacement from the beginning of the program. Internal absolute addresses should be given with a "+" followed by the displacement. And finally, external addresses can be specified by giving two hex bytes, or by an "*" followed by the address.

Once the code is loaded, ADR(DMY$) gives the first location. This also happens to be the entry point of the machine language dump program. Now the dump routine can be executed by calling for the USR function to be executed with ADR(DMY$) as its address. This is done on line 80 of the BASIC program.

It is important to note that the dump routine can only be executed while the BASIC program is running. Trying to execute it by a direct command will not work because the direct command gets inserted in between the end of the program and where the machine code has been poked. This will cause the machine code to be moved. Since it contained some internal absolute addressing, it will not execute properly any more. If the code contains no internal absolute addressing, it can be executed by a direct command.

The machine code is fairly simple, so you should be able to understand what it is doing. Upon entry, the machine code first

checks to see if the right number of parameters are present. If not, the parameters are pulled off the stack and the program returns to BASIC. If the correct number (2) is present, the machine code will dump the requested memory, printing 8 bytes per line.

I hope you will find some of the techniques used in this program useful, as well as the program itself.

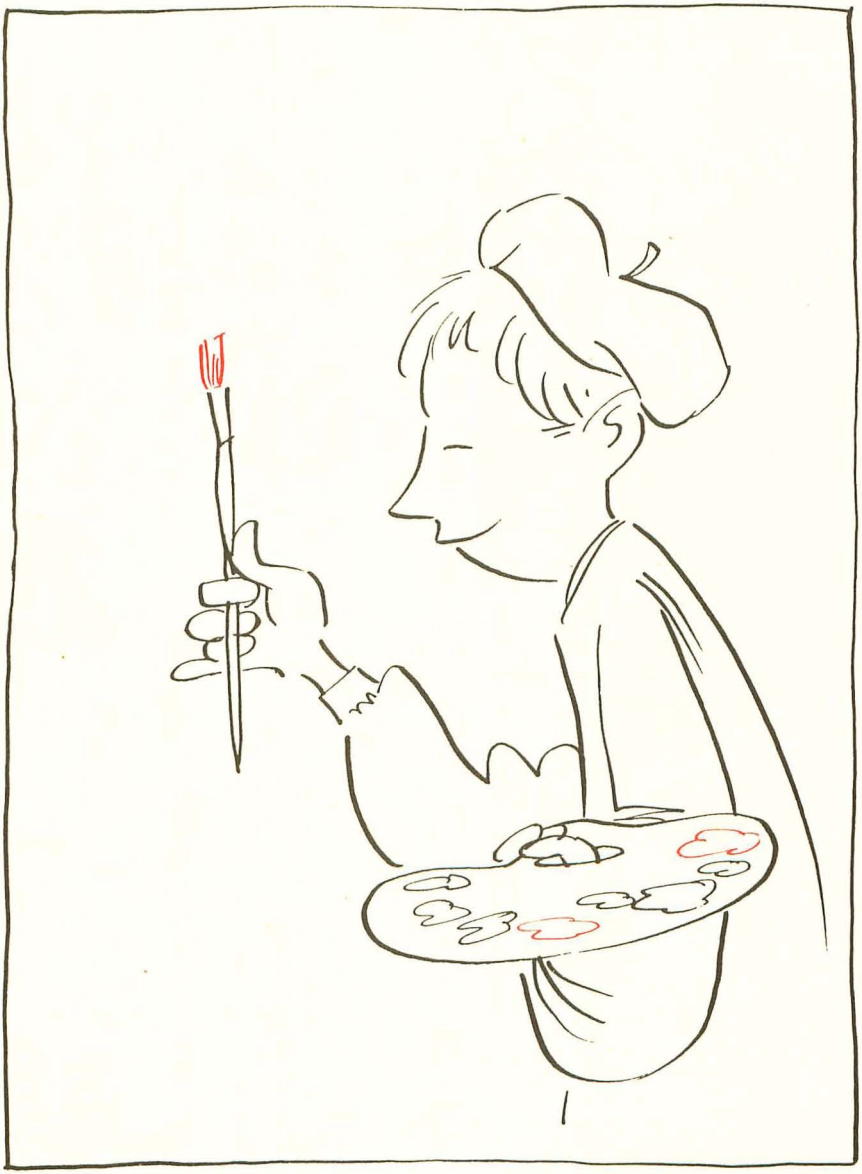
```
1 DIM SA$(4),EA$(4)
10 GOSUB 10200
20 PRINT "INPUT STARTING ADDRESS";
25 INPUT SA$
30 PRINT "INPUT ENDING ADDRESS";
35 INPUT EA$
40 WORD$=SA$:GOSUB 10100
50 SA=NWORD
60 WORD$=EA$:GOSUB 10100
70 EA=NWORD
80 DUMMY=USR(ADR(DMY$),SA,EA)
90 GOTO 20
10000 REM COMPUTE NBYTE FROM HEX$
10010 I=1:GOSUB 10040:NBYTE=X*16
10020 I=2:GOSUB 10040:NBYTE=NBYTE+X
10030 RETURN
10040 X=ASC(HEX$(I,I))-ASC("0")
10050 IF "0"<=HEX$(I,I) AND HEX$(I,I)<="
9" THEN RETURN
10060 IF "A"<=HEX$(I,I) AND HEX$(I,I)<="
F" THEN X=X-7:RETURN
10070 STOP :REM ERROR
10100 REM COMPUTE NWORD FROM WORD$
10110 HEX$=WORD$(1,2):GOSUB 10000:NWORD=
NBYTE*256
10120 HEX$=WORD$(3,4):GOSUB 10000:NWORD=
NWORD+NBYTE
10130 RETURN
10200 REM PUT THE CODE
10210 READ N:REM NUMBER OF BYTES
10220 DIM CODE$(40),HEX$(2),WORD$(4),DMY
$(1),STORAGE(N/6+1)
10230 PC=ADR(DMY$)
10240 READ CODE$
10245 IF CODE$(1,1)="." THEN GOTO 10240
```

```
10250 IF CODE$(1,1)="*" THEN GOTO 10300
10260 IF CODE$(1,1)="+" THEN GOTO 10310
10265 IF CODE$(1,1)="=" THEN GOTO 10350
10270 IF CODE$(1,1)="!" THEN GOTO 10410
10280 HEX$=CODE$(1,2):GOSUB 10000
10290 POKE PC,NBYTE:PC=PC+1:GOTO 10240
10300 WORD$=CODE$(2,5):GOSUB 10100:GOTO
10320
10310 WORD$=CODE$(2,5):GOSUB 10100:NWORD
=NWORD+ADR(DMY$)
10320 NBYTE=INT(NWORD/256)
10330 POKE PC,NWORD-NBYTE*256
10340 PC=PC+1:GOTO 10290
10350 WORD$=CODE$(2,5):GOSUB 10100
10360 NBYTE=ADR(DMY$)+NWORD-(PC+1)
10370 IF NBYTE>127 THEN STOP
10380 IF NBYTE<-128 THEN STOP
10390 IF NBYTE<0 THEN NBYTE=NBYTE+256
10400 GOTO 10290
10410 WORD$=CODE$(2,5):GOSUB 10100
10420 IF NWORD=PC-ADR(DMY$) THEN RETURN
10430 STOP :REM ERROR
20000 DATA 137
20010 DATA .0000,4C,+0030,.JMP START
20020 REM INCPNTR
20030 DATA .0003,E6,D4,.    INC PNTR
20040 DATA .0005,D0,=0009,.BNE @1
20050 DATA .0007,E6,D5,.    INC PNTR+1
20060 REM @1
20070 DATA .0009,60,.       RTS
20080 REM PRBYTE
20090 DATA .000A,48,.       PHA
20100 DATA .000B,4A,.       LSR A
20110 DATA .000C,4A,.       LSR A
20120 DATA .000D,4A,.       LSR A
20130 DATA .000E,4A,.       LSR A
20140 DATA .000F,20,+0015,.JSR PRNYBLE
20150 DATA .0012,68,.       PLA
20160 DATA .0013,29,0F,.    AND #$0F
20170 REM PRNYBLE
20180 DATA .0015,C9,0A,.    CMP #$0A
20190 DATA .0017,30,=001B,.BMI @2
```

```
20200 DATA .0013,69,06,.    ADC #$06
20210 REM @2
20220 DATA .001B,69,30,.    ADC #$30
20230 DATA .001D,20,%F6A4,. JSR OUTCHR
20240 DATA .0020,60,.       RTS
20250 REM TSTPNTR
20260 DATA .0021,38,.       SEC
20270 DATA .0022,AD,+002D,. LDA EA
20280 DATA .0025,E5,D4,.    SBC PNTR
20290 DATA .0027,AD,+002E,. LDA EA+1
20300 DATA .002A,E5,D5,.    SBC PNTR+1
20310 DATA .002C,60,.       RTS
20320 REM EA
20330 DATA .002D,00,00,.    .WORD
20340 REM COUNT
20350 DATA .002F,00,.       .BYTE
20360 REM START
20370 DATA .0030,68,.       PLA
20380 DATA .0031,F0,=0009,. BEQ @1
20390 DATA .0033,C9,02,.    CMP #$02
20400 DATA .0035,F0,=003E,. BEQ CONTINUE
20410 DATA .0037,AA,.       TAX
20420 REM @3
20430 DATA .0038,68,.       PLA
20440 DATA .0039,68,.       PLA
20450 DATA .003A,CA,.       DEX
20460 DATA .003B,D0,=0038,. BNE @3
20465 DATA .003D,60,.       RTS
20470 REM CONTINUE
20480 DATA .003E,68,.       PLA
20490 DATA .003F,85,D5,.    STA PNTR+1
20500 DATA .0041,68,.       PLA
20510 DATA .0042,85,D4,.    STA PNTR
20520 DATA .0044,68,.       PLA
20530 DATA .0045,8D,+002E,. STA EA+1
20540 DATA .0048,68,.       PLA
20550 DATA .0049,8D,+002D,. STA EA
20560 REM DUMP
20570 DATA .004C,A9,9B,.    LDA #EOL
20580 DATA .004E,20,%F6A4,. JSR OUTCHR
20590 DATA .0051,A9,24,.    LDA #'$
20600 DATA .0053,20,%F6A4,. JSR OUTCHR
```

```
20610 DATA .0056,A5,D5,.   LDA PNTR+1
20620 DATA .0058,20,+000A,.JSR PRBYTE
20630 DATA .005B,A5,D4,.   LDA PNTR
20640 DATA .005D,20,+000A,.JSR PRBYTE
20650 DATA .0060,A9,20,.   LDA #'
20660 DATA .0062,20,%F6A4,.JSR OUTCHR
20670 DATA .0065,A9,08,.   LDA #$08
20680 DATA .0067,8D,+002F,.STA COUNT
20690 REM LOOP
20700 DATA .006A,A9,20,.   LDA #'
20710 DATA .006C,20,%F6A4,.JSR OUTCHR
20720 DATA .006F,A0,00,.   LDY #$00
20730 DATA .0071,B1,D4,.   LDA (PNTR)Y
20740 DATA .0073,20,+000A,.JSR PRBYTE
20750 DATA .0076,20,+0003,.JSR INCPNTR
20760 DATA .0079,CE,+002F,.DEC COUNT
20770 DATA .007C,D0,=0069,.BNE LOOP
20780 DATA .007E,20,+0021,.JSR TSTPNTR
20790 DATA .0081,10,=004B,.BPL DUMP
20800 DATA .0083,A9,9B,.   LDA #EOL
20810 DATA .0085,20,%F6A4,.JSR OUTCHR
20820 DATA .0088,60,.      RTS
20830 DATA !0089
```

CHAPTER THREE: Graphics



Made In The Shade: An Introduction To "Three-Dimensional" Graphics on the Atari Computers

David D. Thornburg

If you know anything at all about the Atari 400 and 800, you probably know that these machines give you access to 128 colors. What you may *not* realize is that these colors are specified with two independent parameters which allow you to create "three-dimensional" objects on the display.

The two parameters of interest are *hue* and *luminosity*. Atari gives you access to sixteen colors (the hues), each of which can be darkened or lightened by setting the luminance to one of eight levels. Traditionally, computers that offer limited colors (sixteen total, for example) pre-select different hues *and* luminosities for each color so that inter-color contrast is always apparent, even when the picture is shown on a black and white display. If two colors have the same luminosity, you will not see any difference between the colors when they are shown on a black and white display — a phenomenon you should demonstrate to yourself sometime.

The beauty of the Atari scheme is that the wide range of available colors leads to the ability to create some pretty pictures, even though only four colors can be displayed at one time. The program presented here illustrates a common graphics task — the representation of a solid three-dimensional object through shading.

Since we can display three colors plus the background in a moderate resolution graphics mode, this lets us represent a shaded cube. After all, you can only see a maximum of three faces of a cube at any given time. The function of the program, then, is to create a two-dimensional representation of a cube in which the "realism" results from the control of the shading on each visible face.

In case you are not familiar with Atari graphics, a short digression is in order. Displayed colors are established by the

SETCOLOR command which takes the form SETCOLOR A, B, C in which A is the color register (0-4), B is the hue (0-15) and C is the luminosity (an even number from 0 to 14). (I don't know why luminosity isn't set with numbers between 0 and 7, but the use of even numbers doesn't present too much of a problem, as you will see.) The hues (see Table I) are the various basic colors you can use to draw your pictures, and the luminosities control the brightness from 0 (very dark) to 14 (almost white). Once you have set the color registers, you need to indicate which register should be used for the various plotting commands. This function is performed with the COLOR statement. This statement has the form COLOR D in which D refers to the color register where the desired color is located. Now, for reasons that I don't understand, the value of D is generally larger than the color register number by one. In other words, D = A + 1.

Program Listing

```
10 REM ** SHADING DEMO
20 GRAPHICS 23
30 OPEN #1,4,0,"K:"
40 FOR I=0 TO 4:SETCOLOR I,9,4:NEXT I
50 X0=48:Y0=36
60 COLOR 1
70 FOR I=0 TO 40
80 PLOT X0,Y0+I:DRAWTO X0+40,Y0+I
90 NEXT I
100 COLOR 2
110 FOR I=1 TO 24
120 PLOT X0+I,Y0-I:DRAWTO X0+I+40,Y0-I
130 NEXT I
140 COLOR 3
150 FOR I=1 TO 24
160 PLOT X0+40+I,Y0-I:DRAWTO X0+40+I,Y0+
40-I
170 NEXT I
180 FOR I=0 TO 2
190 GET #1,A
200 IF A<48 THEN A=48
210 SETCOLOR I,1,2*(A-48)
220 NEXT I
230 GOTO 180
```

Now that these tips on Atari color have been described, it is time to try out the program.

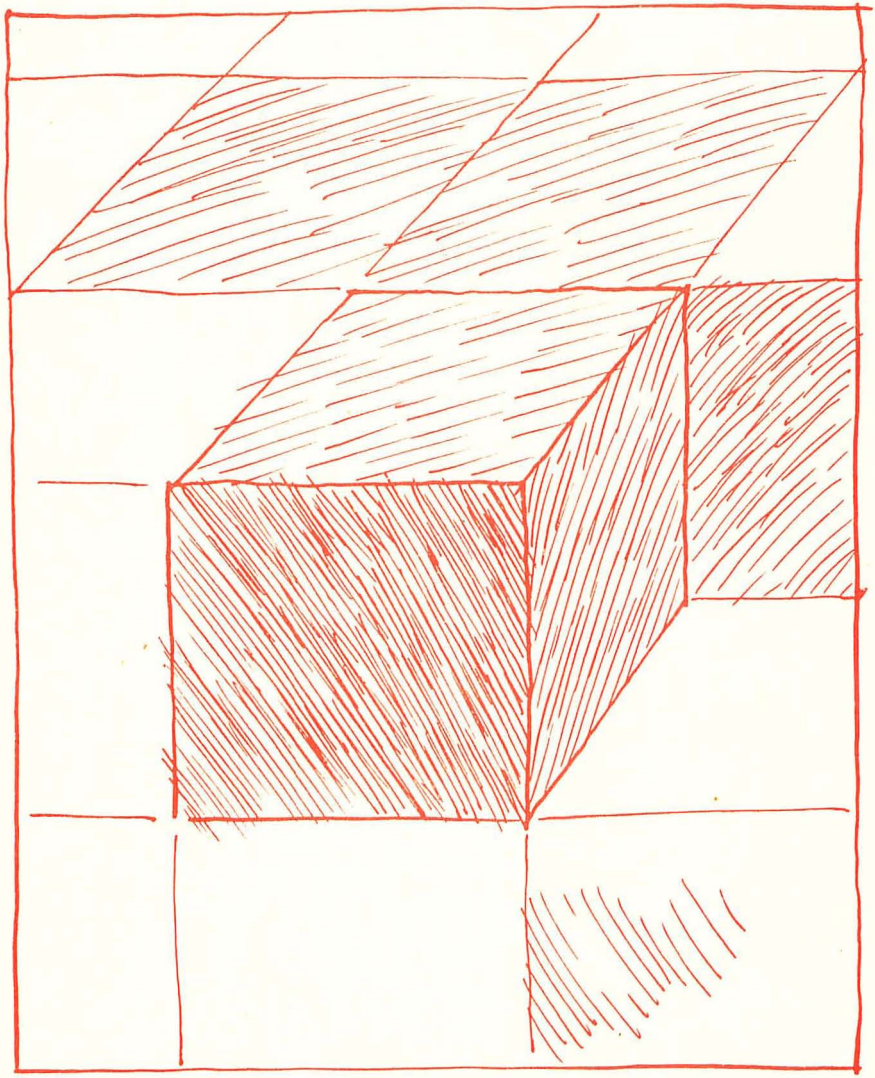
The listing starts out by setting a moderately high resolution full-frame graphics mode in statement **20.** This mode allows the display of four colors and contains 160 x 96 picture elements — plenty for our needs. The **OPEN** statement lets us use a **GET** statement to receive data from the keyboard without having to press **RETURN.** Note that the Atari version of **GET** is *very* different from the version you may be accustomed to from Microsoft BASIC. Next, the color registers are all set at the same color value so that when the cube is first drawn you cannot see it. The front face of the cube is drawn in **COLOR 1** (from **SETCOLOR** register **0**) in lines **70-90,** and the other two faces are drawn in **COLOR 2** and **COLOR 3** in lines **110-130** and **150-170** respectively. At this point, the computer waits in line **190** until a key is typed. (Note that in Microsoft BASIC the program would not stay at a **GET** command, but would look once and be on its way.) Since I expect to be **GET**-ting a keystroke from keys 0 through 7 (which have the Atari-ASCII values 48 through 55), lines **200** and **210** convert the keystroke to an even number between 09 and 14 for use in the **SETCOLOR** command. This program looks for three keystrokes — one for each face of the cube. As each key is typed (try 5, 6 and 7, for example) a cube face will become visible. The result is that a "three-dimensional" representation of a cube is now nicely displayed on your screen.

If you want to change the shadings, type three more numbers between 0 and 7 and see what happens. Next, for some more excitement, type J, K and L. Once again you will see the shaded cube, but the color will have changed from gold to more of a magenta. As you can see, luminance values greater than 14 cause the hue to change.

Now that you know about shading, you should be able to make some truly beautiful pictures with the Atari.

Table 1. Hue Values For The Atari Computers

COLOR	HUE VALUE
GRAY	0
LIGHT ORANGE	1
ORANGE	2
RED-ORANGE	3
PINK	4
PURPLE	5
PURPLE-BLUE	6
BLUE	7



The Fluid Brush

Al Baker

Picasso would have liked this one! Computers won't be replacing canvas and paint immediately, but nobody is placing bets on the limits of computer graphics.

This month I got carried away. I had so much fun changing and improving this program that it incorporates several hints on how to use the Atari plus another way to use the joysticks. Before digging into the program, though, let us see what it does.

Type in the program and run it. All but the bottom four lines of the screen turn black. Near the center of the black area is a white dot. Move joystick 0 and you can paint with the dot, just as if it were a brush dipped in white paint. The motion of the dot on the screen is quite slow. This is intentional. If the dot moved too fast, it would be hard to control. As your skill increases, you can speed the dot up.

Hold down the joystick button and move the joystick. Now the dot moves much faster, but it doesn't paint. It erases. You have a paint brush which moves quickly from one area of the screen to another, and yet paints slowly enough to give you complete control!

Unless you are Tom Sawyer, painting with a white brush can be quite boring. Let's change colors. As you probably know, you have three color registers available in graphics mode 7. When the program starts, you are painting with register 1 set to white. To pick another register, press either the 1, 2, or 3 key. You are now using this register. But before you can paint, you must choose a color. Type in a color number between 0, for white, and 15, for light gold. Press RETURN. Table 1 lists the 16 color possibilities. Now, as you move the joystick, you are painting with this new color.

What's Going On

The program is initialized between lines 1000 and 2030. The beginning location of the dot is position (X = 90, Y = 48); its color, C, is 0; its color register, R, is 1; and its brightness, L, is 10.

Line 1090 opens the keyboard for input. This statment is necessary if you want to read single ASCII characters from the keyboard without using the INPUT statement. The number 1 is my choice for the file number. I could have chosen anything between 1 and 5. The 4 means input, the 0 is required, and the "K:" means the input is from the keyboard.

Look at Diagram 1. The joystick returns numbers between 5 and 15 depending on its position. The actual number is ued as the subscript of arrays XD and YD to determine how the X and Y positions of the dot on the screen are to be changed. For example, if the joystick is pushed away from the user and to the right, the number is 6. XD(6) is equal to 1 and YD(6) equals -1. The dot would move one position right (+1) and one position up (-1) on the screen. The arrays XD and YD are initialized in lines 1110 to 2030.

The main program loop starts at line 150. Look closely at line 160. This statement determines the speed of the dot on the screen. If the button on joystick zero is pushed, STRIG(0) = 0 and S will be 0. If the button isn't pressed, STRIG = 1 and S = 100. Line 170 uses the variable S as the alarm on the delay timer. Finally, lines 150 and 180 cause the dot to blink. Line 200 makes sure that if the button is pressed, then the dot erases, or leaves a black spot, when the joystick is moved.

The rest of the program loop is between lines 250 and 310. Line 250 gets the value of the joystick. This value is used to modify the X and Y positions of the dot as previously discussed. Once these values are computed, line 290 places the dot in its new location.

The statement on line 280 keeps the dot from running off the TV screen. If the PLOT statement tries to put the dot off the screen, an error results. The trap on line 280 branches the computer to the routine at line 3000. That routine adds one to Y if Y is above the screen (Y<0) or subtracts one from Y if Y is below the screen (Y>79). Likewise, it adjusts X if X is to the left (X<0) or right (X>159) of the screen. Finally, the routine jumps back to line 280 to set the trap again and plot.

The user can type in a new register number and color. Line 300 scans the keyboard each time the program loops to see if the artist is ready to change colors. If location 764 isn't equal to 255, then a key has been pressed. The routine beginning on line 4000 is called on to respond to the artist's request.

The first thing done by the keyboard routine is get the ASCII value of the key pressed. The GET statement must use the same number as the open statement on line 1090 and it puts the value of the key in the variable R. The ASCII value for a one is 49 and for a three is 51. If the key is not between these two, it is ignored and another is required. Once a proper key has been pressed, line 4020 converts it into the numbers 1, 2, or 3 and line 4030 sets plotting to that color register.

The POSITION statement does not control the location of

print statements in the text window when graphics mode 1 through 8 are chosen. This is done by poking values into memory locations 656 and 657. Poking a number between 0 and 3 into location 656 will position a statement vertically on the bottom four text lines. Poking a number between 0 and 39 into location 657 will position a print statement between columns 0 and 39 horizontally on the screen. Line 4040 positions the next print statement on the third line of the text area at the bottom of the screen.

Since the print statement on line 4050 is always printed in the same location, it is necessary to erase any previous answers. This is done by including four spaces followed by four back-arrows after the word COLOR. To insert a back-arrow, or any arrow, in a print statement, press the ESC key before typing the arrow key.

Conclusion

I would like to thank Dick Ainsworth for his idea about using two different speeds on the joystick to control different functions, and I'd also like to thank William Bailey for his idea on using arrays to simplify the conversion of joystick values into directions. If you would like to share your ideas with other readers, send them in. If I use them, you will also be acknowledged.

Al Baker, Programming Director, The Image Producers, Inc.

Table 1: The Atari Colors

NUMBER	COLOR	NUMBER	COLOR
0	Gray	8	Blue
1	Gold	9	Gray blue
2	Orange	10	Turquoise
3	Red	11	Olive Green
4	Pink	12	Green
5	Violet	13	Yellow green
6	Purple	14	Brown
7	Light blue	15	Light gold

Diagram 1: The Joystick Control Arrays

CHANGE IN X — XD

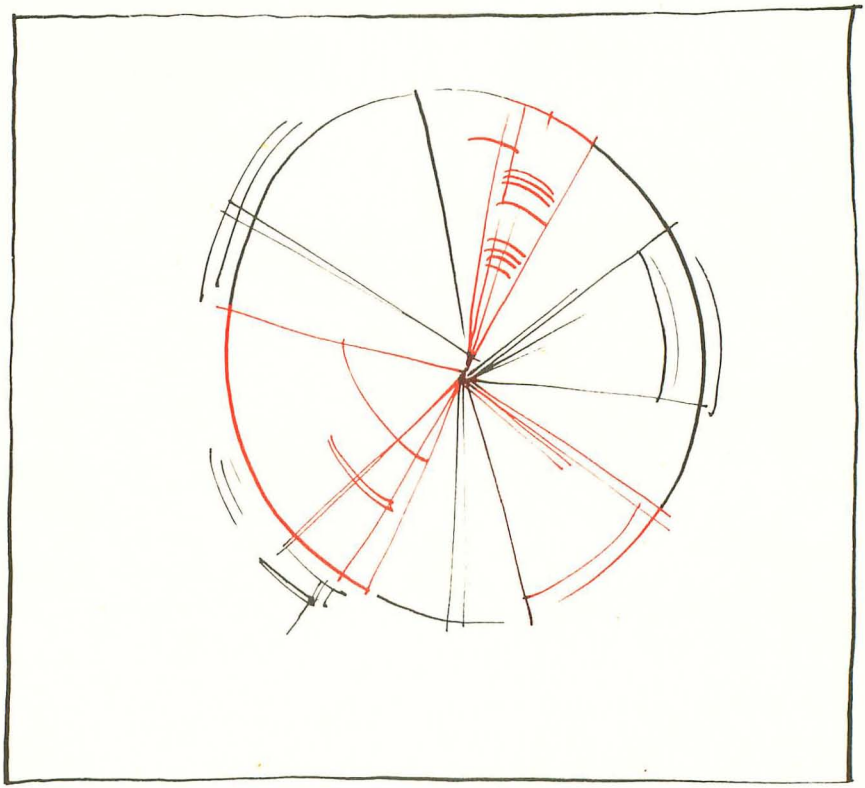
XD(10) = -1 XD(14) = 0 XD(6) = 1
XD(11) = -1 XD(15) = 0 XD(7) = 1
XD(9) = -1 XD(13) = 0 XD(5) = 1

CHANGE IN Y — YD

YD(10 = -1 YD(14) = -1 YD(6) = -1
YD(11) = 0 YD(15) = 0 YD(7) = 0
YD(9) = 1 YD(13) = 1 YD(5) = 1

```
10 REM THE FLUID BRUSH
20 REM
30 REM
40 REM GO SET UP CONDITIONS
50 REM
60 GOSUB 1000
120 REM
130 REM GET BUTTON FOR SPEED
140 REM
150 COLOR 4:PLOT X,Y
160 S=100*STRIG(0)
170 FOR I=1 TO S:NEXT I
180 COLOR R:PLOT X,Y
190 REM
200 IF S=0 THEN COLOR 4:PLOT X,Y:COLOR R
220 REM
230 REM MOVE DOT IF JOYSTICK MOVED
240 REM
250 J=STICK(0)
260 Y=Y+YD(J)
270 X=X+XD(J)
280 TRAP 3000
290 PLOT X,Y
300 IF PEEK(764)<>255 THEN GOSUB 4000
310 GOTO 150
970 REM
980 REM SET UP CONDITIONS
990 REM
1000 GRAPHICS 7
1010 X=90
1020 Y=48
1030 C=0
1040 L=10
1050 R=1
1060 SETCOLOR R-1,C,L
1070 COLOR 1
1080 PLOT X,Y
1090 OPEN #1,4,0,"K:"
1100 DIM XD(15),YD(15)
1110 FOR I=1 TO 15
1120 READ N:XD(I)=N
1130 READ N:YD(I)=N
1140 NEXT I
1150 RETURN
2000 DATA 0,0,0,0,0,0,0,0
2010 DATA 1,1,1,-1,1,0,0,0
2020 DATA -1,1,-1,-1,-1,0,0,0
2030 DATA 0,1,0,-1,0,0
```

```
2970 REM
2980 REM CATCH MOTION OFF THE SCREEN
2990 REM
3000 Y=Y+(Y<0)-(Y>79)
3010 X=X+(X<0)-(X>159)
3020 GOTO 280
3970 REM
3980 REM CHANGE COLOR
3990 REM
4000 GET #1,R
4010 IF (R<49)+(R>51) THEN 4000
4020 R=R-48
4030 COLOR R
4040 POKE 656,2
4050 PRINT "COLOR     <<<<<";
4060 INPUT C
4070 SETCOLOR R-1,C,L
4080 RETURN
```



Color Wheel for the Atari

Neil Harris

This program shows how easy it is to get "pretty pictures" with a minimum of coding. You might want to try the following changes:

110 GRAPHICS 7+16
*145 COLOR INT(RND(1)*3)+1*

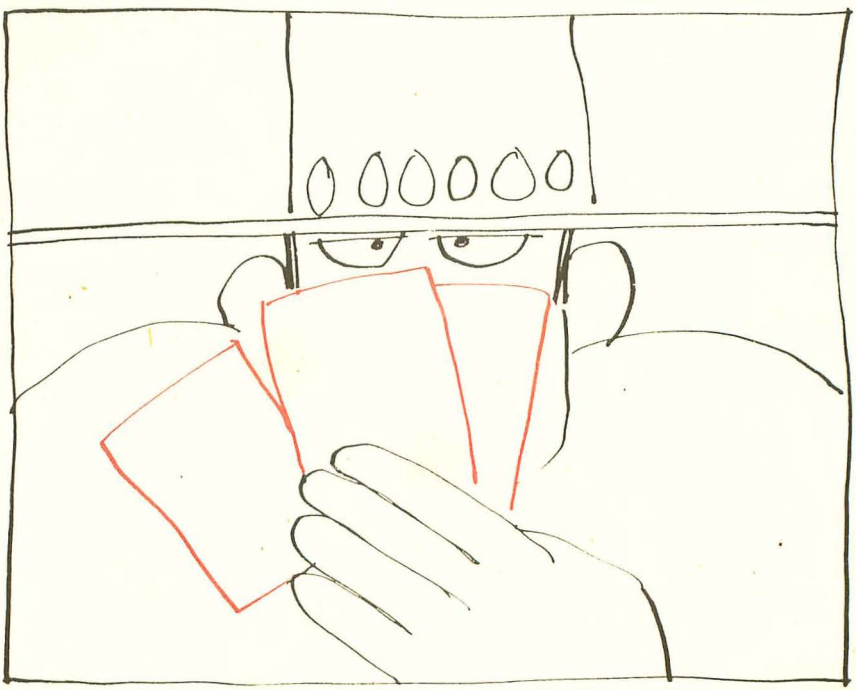
The Color Wheel program was written to experiment with some of the Atari's color graphic capabilities. The screen clears and a series of lines radiate from the center of the screen in random colors, forming a shape with the outline of an ellipse. As the color bands sweep the screen, the colors shift in intensity and hue, forming a constantly changing set of contrasts and shapes.

The program itself is quite simple, thanks to the easy Atari BASIC graphics commands. Graphics mode 7 features 160 by 80 points of resolution in four colors, which are set up in registers. One of the things that made this program possible was that you can change a color register value, which causes all points on the screen associated with that register to change color instantly.

Line 100 selects degree mode for trigonometric functions, which in this case leads to less messy numbers in the FOR-NEXT loop in line 140. Lines 120 and 130 select values for DX and DY, which determine the shape of the ellipse for that cycle. The STEP in line 140 was added because the smaller ellipses otherwise took the same time to draw as bigger ones. Line 145 randomly selects the color register for the current line (an interesting variation is to move this line to line 135, making each ellipse a solid color). Line 150 plots a point at the center of the screen. The formula in the DRAWTO in line 160 was arrived at by using simple trigonometry to determine the point on the ellipse at any given angle around the center. The SETCOLOR statement in line 180 changes a random color register on the screen to a random hue and intensity, and is selected 30% of the time by line 170. Line 190 completes the loop, and 200 allows the program to select a new ellipse shape and keep drawing. I usually put some PRINT statements between lines 110 and 120 for a message in the text window.

This program allows the Atari to show off its nice range of colors, and the plotting routine has been reduced to its bare essentials.

```
100 DEG
110 GRAPHICS 7
120 DX=INT(RND(1)*80)
130 DY=INT(RND(1)*40)
140 FOR L=0 TO 360 STEP (140-DX-DY)/20
145 COLOR INT(RND(1)*5)
150 PLOT 80,40
160 DRAWTO 80+DX*SIN(L),40+DY*COS(L)
170 IF RND(1)>0.3 THEN 190
180 SETCOLOR INT(RND(1)*4),INT(RND(1)*16
),INT(RND(1)*8)*2
190 NEXT L
200 GOTO 120
```



Card Games In Graphic Modes 1 and 2

William D. Seivert

With this subroutine, you can mix the four suit symbols with letters and numbers in graphics modes 1 or 2, [=heart,]=club, /=diamond, and ^=spade.

Have you ever wanted to design a card game to play in Graphics Mode 1 or 2, only to find that you couldn't get the suit characters (heart, spade, diamond, and club) to appear on the screen at the same time as the characters A, K, Q, J, and the digits 0 through 9?

Graphics modes 1 and 2 use the character base pointer (CHBAS, location 756) to point to the table defining the character sets. When location 756 contains 224, you get uppercase letters and the digits and normal punctuation. When you set it to 226, you get small letters and the graphics characters, including the characters for the suits. Since only 64 characters are available in these modes, you can't have both at the same time!

Try this in Direct Mode:

```
GRAPHICS 2: PUT #6,ASC("]"):POKE 756,226
```

When the POKE takes effect, the right bracket changes to its graphics equivalent. (So does the rest of the graphics window!) The table to look at is in the BASIC Reference Manual, Table 9.6.

The 224 or 226 that you POKE into location 756 is the Most Significant Byte (MSB) of the start address of the character set table. Since these tables are in ROM, they cannot be changed directly. Also, since only the MSB of the address is used, the table must begin on a page boundary.

It takes a lot of work and space in BASIC to hold the table and ensure that it is on a page boundary. However, there is an easier way!

The following BASIC subroutine will do the job.

Now I'll explain what this does by line number.

25000,25001 Just some documentation (Remember that GOSUB 25000 will work; BASIC will skip the REMs).

25010 Location 106 contains RAMTOP, the number of pages of RAM. Subtracting 8 leaves enough room for graphics modes 0, 1, and 2, and allows space for the new character set table. Thus, J is

the address where the table will start.

25020 Locations 144 and 145 contain MEMTOP which is BASIC's current top of memory. If, at the time the subroutine is called, the program is already too big to allow for the new table, we won't do it and leave. This implies that all arrays should be DIMensioned and variables defined before calling the subroutine.

25030,25040 This loop moves the original table (57344 = 224*256) from ROM to the location in RAM.

25050 Each character uses 8 bytes (1 byte per TV scan line) to define which pixels should be on for the given character. Adding 472 (=59*8) to the starting address gives the address of the left bracket ([) character.

25060 The TRAP is used so that if the subroutine is called more than once in a run, we won't get ERROR 9 (String DIM Error). We need 32 bytes for string ST$ (4 characters times 8 bytes per character). Then we cancel the TRAP so other errors don't come to this routine.

25070 Now we define the bytes for the four suit characters. The keying sequence after ST$ =" is: CTRL comma, 6, ESC TAB, ESC TAB, greater-than ESC CTRL minus, CTRL H, CTRL comma, CTRL comma, CTRL X, less-than, ESC BACK-S, ESC BACK-S, less-than, CTRL-X, CTRL comma, CTRL comma, ESC CTRL minus, ESC CTRL minus, lowercase W, lowercase W, CTRL H, ESC CTRL minus, CTRL comma, CTRL comma, CTRL X, less-than, ESC BACK-S, ESC BACK-S, CTRL X, less-than, CTRL comma, and the closing double quotes.

25080 Start the loop to put the bytes.

25090 Convert one character at a time to its ATASCII value and POKE it in the appropriate location.

25100 Finish the loop.

25110 Put the address of the new table in CHBAS (location 756).

25120 Return to the caller.

That's all there is to it! Of course this method will work for any characters you want to redefine. All you have to do is decide which characters you can do without, and the bit patterns of the characters you want.

With the above routine as it is, if you want a heart, use the left bracket, etc. Use PUTs to the screen for the characters you want. Remember that you can use inverse-video and/or add values to change colors.

For example, without using any SETCOLOR statements, try

```
GRAPHICS 2: GOSUB 25000
PUT #6,ASC("inverse-video of right bracket")
```

to get a blue Club, or

```
PUT #6,ASC("inverse-video of left bracket') + 32
```

to get a red Heart.

A Few Words of Warning

Every time you change graphics modes (even GRAPHICS n + 32 which doesn't change the screen), the Operating System resets location 756 to 224, pointing to the normal character set. If you want the suit characters back again, just GOSUB 25110.

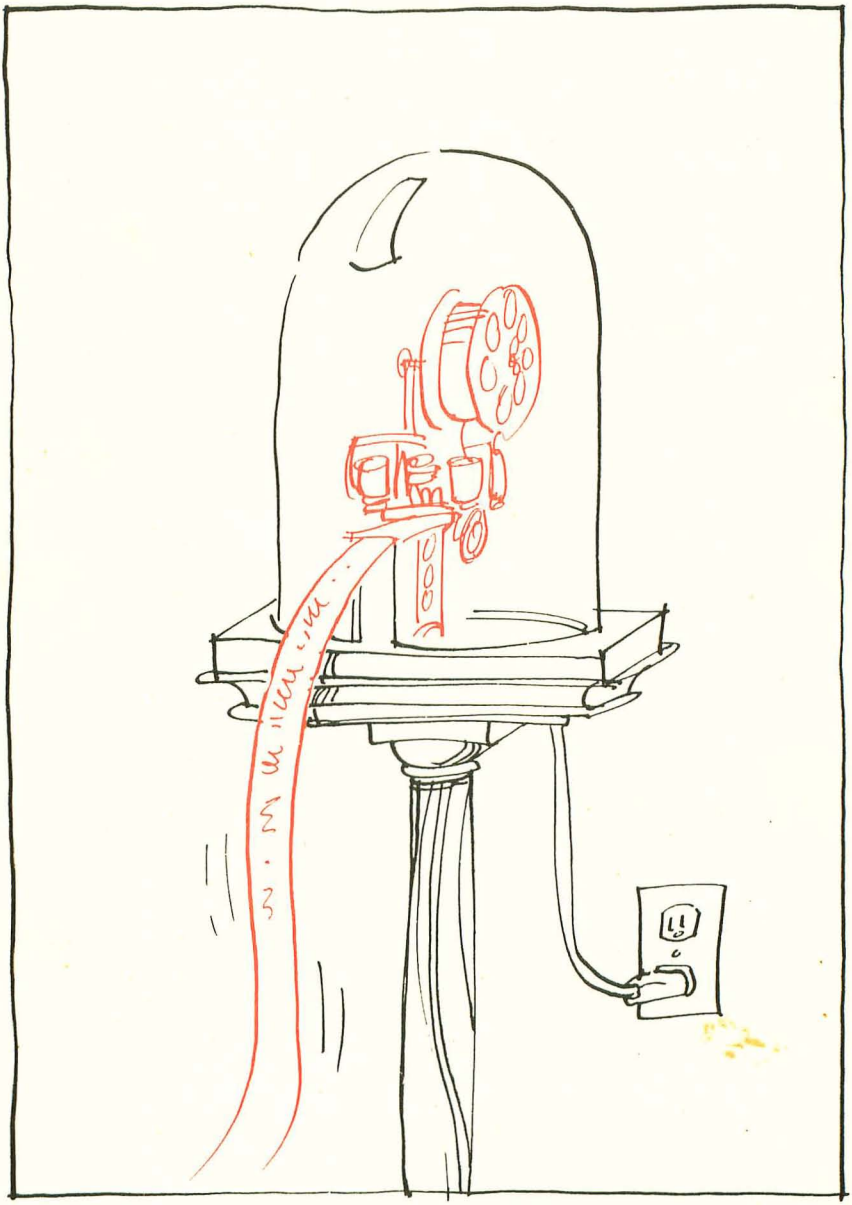
Also, if you use a graphics mode greater than 2, you might destroy the table. So you will probably want to GOSUB 25000 after coming out of graphics mode 3 or above.

Of course you do not have to use the same line numbers, and you might want to remove the memory overlap check at line 25020, but that's up to you.

Try it! You'll like it!

```
25000 REM REDEFINE CHARACTER SET AND REP
LACE / WITH [,]
25001 REM DESTROYS TRAP, USES STRING ST$
 AND VARIABLES I AND J
25010 J=(PEEK(106)-8)*256
25020 IF J<=PEEK(144)+256*PEEK(145) THEN
 ? "PROGRAM TOO LARGE TO REDEFINE CHARS"
:GOTO 25120
25030 FOR I=0 TO 1023
25040 POKE J+I,PEEK(57344+I):NEXT I
25050 J=J+472
25060 TRAP 25070:DIM ST$(32):TRAP 40000
25065 REM (FOLLOWING LINE IS PER IRIDIS
CONVENTION --ED.)
25070 ST$="(,)6(TAB TAB)>(UP)(H)(,,)(X)<
(BACK BACK)<(X)(,,)(UP UP)>>>(H)(UP)(,,)(
X)<(BACK BACK)(X)<(,)"
25080 FOR I=1 TO 32
25090 POKE J+I-1,ASC(ST$(I,I))
```

```
25100 NEXT I
25110 POKE 756,PEEK(106)-8
25120 RETURN
```



Ticker Tape Messages

Eric Martell and Chris Murdock

The large text modes, [GR. 1, GR.2] are very convenient. With text like this available, the Atari can become a useful and eye-catching message presentation device. The following program makes use of some simple string manipulations, to move text across the screen in a manner reminiscent of ticker tape or a marquee sign. The actual text movement is done by line 50 in the following manner:

The first 19 characters of the message string [A$] are printed at position 1.5 [the vertical center of the screen]. A temporary string [C$) is set equal to the second through the 20th characters in A$. Then A$ is added [concatenated] to C$. Since C$ and A$ are dimensioned to be the same length, this has the effect of attaching the first character in A$ to the end of C$. A$ is then set equal to C$ and printed once again.

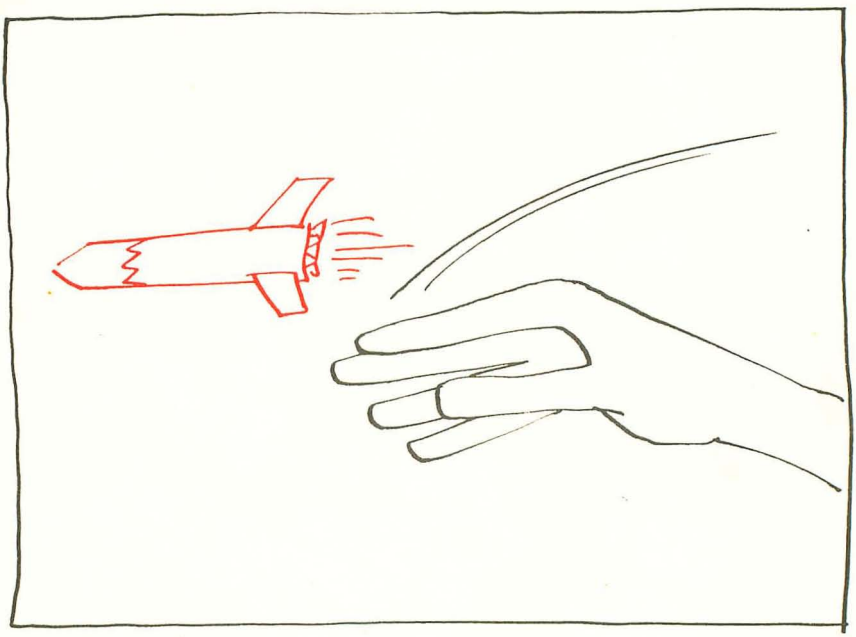
The variable K is set up to check for any key being pressed. This action will terminate the program in line 55. A delay loop is inserted in line 55 to increase readability, since the string manipulation is so fast that the letters become blurred unless slowed down.

The rest of the program contains enough remarks to be self-explanatory.

```
0 REM MOVING MESSAGE PROGRAM FOR THE ATA
RI
1 PRINT "{CLEAR}":REM CLEAR SCREEN BEFOR
E GOING ON
2 REM Dimension strings
10 DIM X$(1000),B$(1),W$(20),P$(20),Y$(2
0),Z$(20)
15 W$="* * * * * * * * * * ":REM BORDER
16 Y$=W$
19 REM Clear strings and set B$=blank fo
r clearing the remainder of X$
20 X$="":B$=" "
24 REM Input your text here
25 ? :? "Enter your message":INPUT X$
29 REM CLEAR THE REST OF X$ IF SHORTER T
HAN SCREEN WIDTH (19)
30 IF LEN(X$)<20 THEN FOR C=1 TO 20-LEN(
```

```
X$):X$(LEN(X$)+1)=B$:NEXT C:X$(LEN(X$)+1
)=B$
35 DIM A$(LEN(X$)),C$(LEN(X$)):A$=X$
39 REM GOTO GRAPHICS MODE 2+16 AND PRINT
 STRINGS
40 GRAPHICS 18
45 REM Move borders of stars
46 POSITION 1,3:? #6;Y$(1,19):P$=W$(2):P
$(LEN(P$)+1)=W$:W$=P$
47 POSITION 1,7:? #6;Y$(1,19):Z$=Y$(2):Z
$(LEN(Z$)+1)=Y$:Y$=Z$
49 REM Move message strins and check loc
ation 764 to see if a key was struck
50 POSITION 1,5:? #6;A$(1,19):C$=A$(2):C
$(LEN(C$)+1)=A$:A$=C$:K=PEEK(764)
54 REM Pause to increase readability, se
t color randomly, and reset attract flas

55 FOR TI=1 TO 50:NEXT TI:POKE 77,0:SETC
OLOR INT(RND(0)%4),INT(RND(0)%15),8:IF K
=255 THEN 46
```



Player/Missile Graphics With the Atari Personal Computer Systems

Chris Crawford

Some think that this is among the very best ideas printed about the Atari to date. Study it, experiment, and the techniques here will considerably amplify your programming skills.

Anybody who has seen ATARI's Star Raiders™ knows that the Atari Personal Computer System has vastly greater graphics capabilities than any other personal computer. Owners of these computers might wonder if they can get their machines to do the fabulous things that Star Raiders does. The good news is that you can indeed write programs with graphics and animation every bit as good as Star Raiders. In fact, I think it's possible to do better. The bad news is that all this video wizardry isn't as easy to use as BASIC. The Atari computer is a very complex machine; mastering it takes a lot of work. In this article I will explain just one element of the graphics capabilities of the Atari Personal Computer System: player-missile graphics.

Player-missile graphics were designed to meet a common need in personal computing, the need for animation. To understand player-missile graphics you need to understand the old ways of doing animation on machines like the Apple. These machines use what we call pure playfield graphics, in which bits in RAM are directly mapped onto the television screen. You move an image across the screen by moving a pattern of bits through RAM. The procedure you must use is as follows: calculate the current addresses of the bit pattern, erase the bit pattern from these addresses, calculate the new addresses of the bit pattern, and write the bit pattern into the new addresses.

This can be a terribly slow and cumbersome process, particularly when you are moving lots of bits (large objects or many objects) or when the motion is complex. Consequently, most animation on computers like the Apple is limited to pure horizontal motion, pure vertical motion, small objects, or slow motion. Animation like you get in Star Raiders is utterly impossible.

To understand the solution to this problem you must understand its fundamental cause. The screen image is a two-dimensional entity, but the RAM that holds the screen image is a one-dimensional entity. Images that are contiguous on the screen do not necessarily occupy contiguous RAM locations (see Figure 1). To move an image you must perform messy calculations to figure out where it will end up in RAM. Those calculations eat up lots of time. We need to eliminate these calculations by shortcutting past the 2d-to-1d transformation logjam. What we need is an image that is effectively one-dimensional on the screen and one-dimensional in RAM.

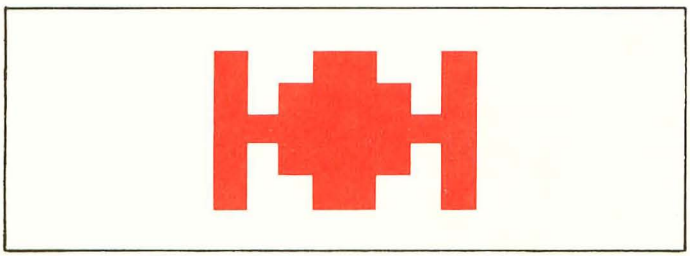
Let's set aside a table in RAM for this one-dimensional image. We'll call this table and its associated image a player. We'll have the hardware map this image directly onto the screen, on top of the regular playfield graphics. The first byte in the table will go onto the top line of the screen. The second byte will go onto the second line of the screen, and so on down to the bottom of the screen. Although I'm calling the image one-dimensional, it's actually 8 bits wide, because there are 8 bits in a byte. It's a straight bit-map; if a bit in the byte is turned on, then the corresponding pixel on the screen will be lit up. If the bit in the byte is turned off, then the corresponding pixel has nothing in it.

We can draw a picture with this scheme by turning the appropriate bits on or off. The picture we can draw is somewhat limited; it is tall and skinny, only 8 bits wide but stretching from the top of the screen to the bottom. Let's say we want to draw a picture of a little spaceship. We do this by storing zeros into most of the player RAM. We put the bits that form the spaceship into the middle of the player RAM so that it appears in the middle of the screen. See Figure 2 for a depiction of this process.

So far we don't have much: just a dinky image of a little spaceship. How do we get animation? We move it vertically with the same technique that other computers use. First we must erase the old image from RAM, then we draw in the new image. This time, however, the problem is much simpler. We move the image down by moving its bit pattern one byte forward in RAM. We move the image up by moving its bit pattern one byte backwards in RAM. We use no crazy two-dimensional calculations, just a simple one-dimensional move routine. It's trivial in BASIC and easy in assembly language. Horizontal motion is even easier. We have a hardware register for the player called the horizontal position register. When we put a number into the horizontal position

register, the player is immediately moved to that horizontal position on the screen. Put a big number in and POW! — the player is on the right side of the screen. Put a little number in and POW! — the player is on the left side of the screen. Horizontal motion is achieved by changing the number you put into the horizontal position register. The two techniques for horizontal and vertical motion can be mixed in any way to produce any complex motion you desire.

This is the two-dimensional screen image



Here are the corresponding bytes in RAM (hexadecimal)

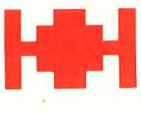
```
0  0  0
0  0  0
0 99  0
0 Bd  0
0 FF  0
0 Bd  0
0 99  0
0  0  0
0  0  0
```

This is how the bytes would be placed in one-dimensional RAM. Note how the bytes that make up the spaceship are scattered through the RAM. What a headache!

```
0
0
0
0
0
0
0
99
0
0
Bd
0
0
FF
```

```
0
0
Bd
0
0
Bd
0
0
99
0
0
0
0
0
0
0
```

Figure 2
How to draw in binary



graphical representation: one byte 8 bits	binary representation	hexadecimal representation	decimal representation
	1 0 0 1 1 0 0 1	99	153
	1 0 1 1 1 1 0 1	Bd	189
	1 1 1 1 1 1 1 1	FF	255
	1 0 1 1 1 1 0 1	Bd	189
	1 0 0 1 1 0 0 1	99	153

The capabilities I have described so far are nice, but taken alone they don't give you much. That's why Atari added a long list of embellishments to this basic system which enormously extend its power. The first embellishment is that you have not just one, not two, not three, but FOUR (count 'em, FOUR) players available. This means that you can have four little spaceships flying around on the screen. They are all independent and so can move independently. The next embellishment is that each player has its own color register. Thus, you can set each player to a different color, completely independent of the colors in the playfield. This gives you the capability of putting up to nine colors onto the screen, depending on your graphics mode. Next, you have the capability of making a player double or quadruple width. This doesn't change the eight-bit resolution of the player, but it does allow you to make him fatter or skinnier as you please. Next, you can select the vertical

resolution of the player to either single line resolution (each byte occupies one scan line on the screen) or double line resolution (each byte occupies two scan lines on the screen. Next, you can select the image priorities of players versus playfield. Since both players and playfield will be imaged onto the same location of the screen you have to decide who has priority in the event of a conflict. You can set players to have higher priority than playfield, playfield to have higher priority than players, or several mixtures of player-playfield priority. This allows you to have players disappear behind playfield or vice-versa. Finally, you have tiny two-bit players called missiles. Each player has one missile associated with him. The missile takes the same color as the player but can move independently of the player. This allows bullets or other small graphics items. If you want, you can group the four missiles together to form a fifth player. They then get a separate color.

How do you use all of these fantastic capabilities? You think that it would be terribly difficult to put all of this together into a program, but it isn't. Listing 1 shows a program that puts a player onto the screen and moves it around with the joystick. As you can see, the program is ridiculously short. Here's how it works:

Line 10 sets the background color to black (the better to see the player by). It also sets up our starting positions, X being the horizontal position and Y being the vertical position.

Program 1. Program to demonstrate player-missile graphics.

```
10 SETCOLOR 2,0,0:X=120:Y=48:REM set bac
kground color and player position
20 A=PEEK(106)-8:POKE 54279,A:PMBASE=256
*A:REM Set player-missile address
30 POKE 559,46:POKE 53277,3:REM Enable P
M graphics with 2-line resolution
40 POKE 53248,X:REM Set horizontal posit
ion
50 FOR I=PMBASE+512 TO PMBASE+640:POKE I
,0:NEXT I:REM Clear out player first
60 POKE 704,216:REM Set color to green
70 FOR I=PMBASE+512+Y TO PMBASE+516+Y:RE
AD A:POKE I,A:NEXT I:REM Draw player
80 DATA 153,189,255,189,153
```

```
90 REM Now comes the motion routine
100 A=STICK(0):IF A=15 THEN GOTO 100
110 IF A=11 THEN X=X-1:POKE 53248,X
120 IF A=7 THEN X=X+1:POKE 53248,X
130 IF A=13 THEN FOR I=6 TO 0 STEP -1:PO
KE PMBASE+512+Y+I,PEEK(PMBASE+511+Y+I):N
EXT I:Y=Y+1
140 IF A=14 THEN FOR I=0 TO 6:POKE PMBAS
E+511+Y+I,PEEK(PMBASE+512+Y+I):NEXT I:Y=
Y-1
150 GOTO 100
```

Line 20 finds the top of RAM and steps back eight pages to reserve space for the player-missile RAM. It then pokes the resultant page number into a special hardware register. This tells the computer where it will find the player-missile data. The players are arranged in memory as shown in Figure 3. Finally, line 20 keeps track of where the player memory is through the variable PMBASE. Because of this arrangement, this program will work on any Atari Personal Computer System, regardless of the amount of RAM in place. The number of pages by which you must step back (8 in this case) depends on how much memory your graphics mode consumes and whether you are in single-line resolution or double-line resolution. In any event, the number of pages you step back must be a multiple of 4 for double-line resolution and a multiple of eight for single-line resolution.

Line 30 first informs the computer that this program will use double-line resolution. Poking a 62 into location 559 would give single-line resolution. The next instruction enables player-missile graphics; that is, it authorizes the computer to begin displaying player-missile graphics. Poking a 0 into location 53277 revokes authorization and turns off the player-missile graphics.

Line 40 sets the horizontal position of the player.

Line 50 is a loop that pokes 0's into the player 0 RAM area. This clears the player and eliminates any loose garbage that was in the player RAM area when the program started.

Line 60 sets the player's color to green. You can use any color you want here. The colors here correspond exactly to the colors you get from the SETCOLOR command. Take the hue value from the SETCOLOR command, multiply by 16, and add the luminosity value. The result is the value you poke into the color register.

Line 70 reads data bytes out of line 80 and pokes them into the player RAM area. The bytes in line 80 define the shape of the player. I calculated them with the process shown in Figure 2. Here you have lots of room for creativity. You can make any shape that you desire, as long as it fits into eight bits. You want more bits? Use four players shoulder to shoulder and you have 32 bits. You can make the look longer to give more vertical height to your player.

These seven lines are sufficient to put a player onto the screen. If you only put in this much of the program, and ran it, it would show the player on the screen. The next lines are for moving the player with the joystick plugged into port 0.

Line 100 reads the joystick.

Line 110 checks to see if the joystick has been moved to the left. If so, it decrements the horizontal position counter and pokes the horizontal position into the horizontal position register. The line does not protect against bad values of the horizontal position (X<1 or X>255).

Line 120 checks to see if the joystick is pressed to the right. If so, it increments the horizontal position counter and pokes the horizontal position into the horizontal position register.

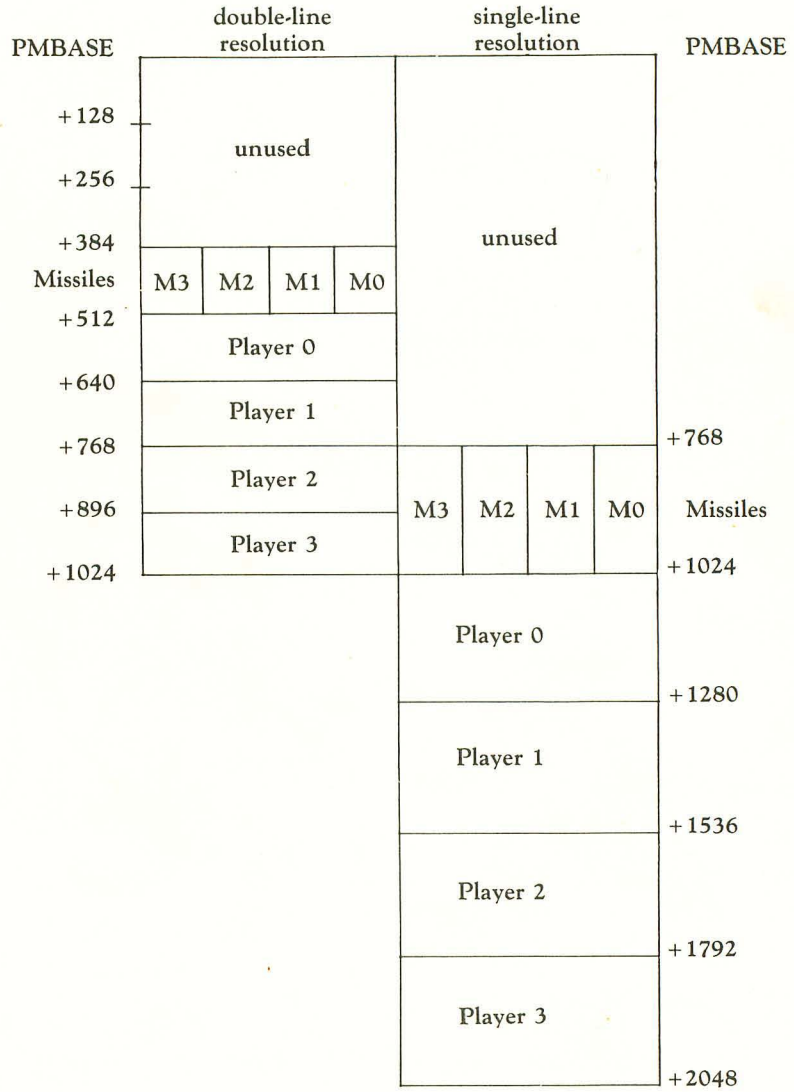
Line 130 checks to see if the joystick is pressed down. If so, it moves the player image in RAM forward by one byte. There are six bytes in the player image that must be moved. When it has moved them, it increments the vertical position counter.

Line 140 performs the same function for upward motion.

Line 150 starts the joystick poll loop over again.

This program was written to help you understand the principles of player-missile graphics; as such it has many weaknesses. It also has much potential for improvement. You might want to soup it up in a variety of ways. For example, you could speed it up with tighter code or an assembly language subroutine. You might add more players; perhaps each could be controlled by a separate joystick. You could change the graphics shapes. You could make the colors change with time or position or how much fuel they have left, or whatever. You could add missiles for them to shoot with. You could change width to give the impression of 3D motion that Star Raiders gives. You could add playfield priorities so they could move behind some objects, but in front of others. The possibilities are almost limitless.

Figure 3
Player-missile graphics RAM positioning
PMBASE must be on 1K boundary for double-line resolution,
2K boundary for single-line resolution



Useful addresses
(all values in decimal)

559 put a 62 here for a single line, a 46 for double line resolution
623 sets player/playfield priorities (only one bit on!)
1: all players have priority over all playfield registers
4: all playfield registers have priority over all players
2: mixed. P0 & P1, then all playfield, then P2 & P3
8: mixed. PF0 & PF1, then all players, then PF2 & PF3
704 color of player-missile 0
705 color of player-missile 1
706 color of player-missile 2
707 color of player-missile 3
53248 horizontal position of player 0
53249 horizontal position of player 1
53250 horizontal position of player 2
53251 horizontal position of player 3
53252 horizontal position of missile 0
53253 horizontal position of missile 1
53254 horizontal position of missile 2
53255 horizontal position of missile 3
53256 size of player 0 (0=normal, 1=double, 3=quadruple)
53257 size of player 1 (0=normal, 1=double, 3=quadruple)
53258 size of player 2 (0=normal, 1=double, 3=quadruple)
53259 size of player 3 (0=normal, 1=double, 3=quadruple)
53277 A 3 here enables player-missile graphics, a 0 disables them.
54279 put high byte of PMBASE here

The Basics of Using POKE in Atari Graphics

Charles G. Fortner

Did you ever wonder how the Atari can store 61,440 pixel in less than 8,000 bytes? With the information in this article, you'll have the background to create graphics in machine language, high-speed screen save/recall, mix text and graphics, and lots else.

In order to use the poke statement in Atari graphics, we must first know two things:

1) Where to poke
2) What to poke

To display where to poke, we must look at the display list for each graphics mode. This display list is found by PEEK (560) + PEEK (561) *256. The display list determines how the memory is displayed on the screen. The 5th and 6th byte of the display list hold the addresses of the first byte to be displayed. Table 1-1 gives the starting address for each graphics mode plus other information.

Determining what to poke involved trial and error with the following results:

1) Graphics Modes 3, 5, 7, 19, 21, 23
These modes are four color modes which display only four pixels for each eight bit byte of memory displayed. Bits 7 and 6, numbered as 7-6-5-4-3-2-1-0, determine the color of the first (left-most) pixel; bits 5 and 4 the second; 3 and 2 the third; and 1 and 0 the fourth. The two control bits act as a **"COLOR"** statement for each pixel. If the hex value of the two control bits equals 0 it corresponds to a **"COLOR 0"** statement; if they equal 1, they correspond to a **"COLOR 1"** statement, etc.

2) Graphics Modes 4, 6, 20, 22
These modes are two color modes which display eight pixels for each eight bit byte of memory. Each bit acts as a **"COLOR"** statement for an individual pixel. A one in a location corresponds to a **"COLOR 1"** statement and a zero corresponds to a **"COLOR 0"** statement.

3) Graphics Mode 8, ~~254~~ 24
These modes are high resolution modes with only one color. They display eight pixels per memory byte with a "1" bit displaying a

TABLE 1-1

GRAPHICS MODE	DISPLAY DATA ADDR.	# OF ROWS	# OF COLUMNS	BYTES PER ROW	BITS DISPLAYED PER BYTE	#OF COLORS AVAILABLE
3	24176	20	40	10	4	4
4	23936	40	80	10	8	2
5	23456	40	80	20	4	4
6	22496	80	160	20	8	2
7	20576	80	160	40	4	4
8	NOTE 1	160	320	40	8	1
19	24176	24	40	10	4	4
20	23936	48	80	10	8	2
21	23456	48	80	20	4	4
22	22496	96	160	20	8	2
23	20576	96	160	40	4	4
24	16720	192	160	40	8	1

NOTE #1: Graphics Mode 8 has two addresses — 16720 is the starting address for the first 80 lines and 20480 is the starting address for the second 80 lines.

pixel of the same color as the background but with a higher luminane. A "0" bit displays a pixel of the same color and luminance as the background.

The "COLOR" statements mentioned in the above explanations indirectly control the color of each pixel by determining which color register is active for an individual pixel. The exact manner in which a "COLOR" statement chooses this register is explained in Table 9.5 of the **Atari-Basic Reference Manual.**

Here's an interesting program to get started in graphics:

```
10 GRAPHICS 5
20 ADDR=PEEK(560)+PEEK(561)*256
30 ADDR=PEEK(ADDR+4)+PEEK(ADDR+5)*256
40 B=INT(RND(0)*800):REM -
 PICK A RANDOM BYTE IN DISPLAY
50 A=INT(RND(0)*255):REM -
 PICK RANDOM VALUE BETWEEN 0 AND 255
60 POKE ADDR+B,A:REM -
 POKE RANDOM VALUE INTO RANDOM BYTE
70 GO TO 40
```

References: "Atari 400/800 Basic Reference Manual," Copyright 1980, Atari, Inc.

A note on using the basics of POKE . . .

Larry rewrote the original program that Charles sent in so it will adjust itself to your machine's memory. After you try the program in the article, take a look at these. I expanded them to randomly alter the SETCOLOR parameters . . . you'll discover some of the versatility of your machine after you let the program run for five minutes or so.

```
10 GRAPHICS 23
20 ADDR=PEEK(560)+PEEK(561)*256
30 ADDR=PEEK(ADDR+4)+PEEK(ADDR+5)*256
35 I=INT(RND(0)*16)
36 J=INT(RND(0)*16)
37 K=INT(RND(0)*5)
38 SETCOLOR K,J,I
40 B=INT(RND(0)*3840):REM -
 PICK A RANDOM BYTE IN DISPLAY
50 A=INT(RND(0)*255):REM -
 PICK RANDOM VALUE BETWEEN 0 AND 255
60 POKE ADDR+B,A:REM -
 POKE RANDOM VALUE INTO RANDOM BYTE
70 GOTO 35
```

```
10 GRAPHICS 7
20 ADDR=PEEK(560)+PEEK(561)*256
30 ADDR=PEEK(ADDR+4)+PEEK(ADDR+5)*256
35 I=INT(RND(0)*16)
36 J=INT(RND(0)*16)
37 K=INT(RND(0)*5)
38 SETCOLOR K,J,I
40 B=INT(RND(0)*3200):REM -
 PICK A RANDOM BYTE IN DISPLAY
50 A=INT(RND(0)*255):REM -
 PICK RANDOM VALUE BETWEEN 0 AND 255
60 POKE ADDR+B,A:REM -
 POKE RANDOM VALUE INTO RANDOM BYTE
70 GOTO 35
```

Designing Your Own Atari Graphics Modes

Craig Patchett

This one is on the list of "things you 'gotta know" about the Atari. Get set for some video magic.

The graphics modes that Atari supplies with their 400 and 800 computers are nice, but what if you want a little more? For example, how about a large-type heading, with a smaller-type sub-heading below it, all over a graphics display? Terrific, you say, but you're not an Atari engineer? Don't worry about a thing. With this article, a little concentration, and some time in front of the keyboard, you'll have Atari graphics modes performing at the snap of your fingers.

First, a simple explanation of what we'll be doing. In a series of memory locations deep inside your Atari rests a special list of numbers that tell the computer which graphics mode it's in. Each time you change graphics modes, this list also changes. But wait a minute. Why a list of numbers instead of just one? Because there is one number for each graphics row on the screen. For example, in graphics mode 2+16 (no text window) there are twelve graphics rows, so there would be twelve numbers in the list. For graphics mode 7+16, there would be 96 rows, or 96 numbers. The table labeled *Modes and Screen Formats* in your Atari BASIC reference manual shows the number of rows in each graphics mode. We'll be referring to it again later.

As I said before, when you change graphics modes, using the GRAPHICS command, the list changes. It may become longer or shorter, depending on the mode, and the numbers in it will change. But the numbers will all be the same. Obviously, since they stand for the graphics mode of each row on the screen, if half of them were one number and the other half another, then half of the screen would be one mode and the other half another. This is not how Atari BASIC was designed. It is, however, what we want. So what we're going to be doing is changing the numbers in the list to make the screen behave the way we want it to. Let's take a look at exactly how it's done.

How Much Of Each Mode Should I Have?

The first thing we have to do is figure out exactly how we want the

screen to look. Let's take the example from the beginning of the article — a large-type heading (mode 2), with a smaller-type subheading below it (mode 1), all over a graphics display (mode 3). Unfortunately, we can't just decide to have, for instance, four rows of mode 2, two rows of mode 1, and nine rows of mode 3. There's a simple rule we have to follow in deciding how many rows of each mode we're going to have.

You may already know that your television picture is made up of hundreds of little lines going across the screen from top to bottom (if you don't, you know now!) If you look closely at the screen, you can probably see them. These lines are formed by a single beam of light that scans the screen very quickly (sixty times a second) to make the picture, so we'll call them scan lines. The part of the screen that your Atari lets you use for graphics has 192 of these lines.

Each graphics row is a certain number of scan lines "high." In mode 1, for example, each row is eight scan lines high. If you look at the **Table of Modes and Screen Formats** that I have mentioned before, you'll see that there are 24 rows in mode 1 (remember, we're only interested in "full screen.") Surprise! Twenty-four rows, each eight scan lines high, means $8 \times 24 = 192$ scan lines in all. To figure out how many scan lines high the rows in other modes are, just look at the table and divide 192 by the number of rows in a full screen.

The reason we need to know all this is because we must make our new mode so that it has a total of 192 scan lines. No more, no less. This means you have to do a little bit of juggling around with the different modes you want to use, but it's really not all that difficult. I'll demonstrate with our example. Let's suppose we need three rows of mode 2 and two rows of mode 1. All we need to do is figure out how many rows of mode 3 we should have to make a total of 192 scan lines. We look at the table and figure out that in mode 2, each row is sixteen (192 scan lines/12 rows) scan lines high. Since we want three rows of mode 2, that makes forty-eight scan lines so far. Similarly, we want two rows of mode 1, which uses eight (192 scan lines/24 rows) scan lines for each row. So that makes another sixteen scan lines, or sixty-four all together, which leaves us $192 - 64 = 128$ scan lines still left over. We'll use these for mode 3. We look at the table again and see that mode 3 uses eight scan lines for each row also, so how many rows do we need? 128 leftover scan lines/8 scan lines per row of mode $3 = 16$ rows of mode 3.

So now we know that our graphics mode is going to have three rows of mode 2, two rows of mode 1, and sixteen rows of mode 3.

Let's tell the computer.

How Do I Tell The Computer?

We have to start by telling the Atari in a graphics mode it understands. Of course, we can't use just any mode, but this time the rule is a lot easier. Out of the modes you're going to be using, take the one that uses the most memory (look at the table under "RAM required"). In our example, mode 1 uses the most memory, so the first line in our program is:

```
10 GRAPHICS 1
```

The next step is to find out where the list of numbers begins. Since it isn't always in exactly the same place, we must PEEK into the computer's memory at two locations that tell us where it is. Since we'll need to use the number that tells us where the list begins later, we'll give it a name:

```
20 BEGIN=PEEK(560)+PEEK(561)*256+4
```

This line will always be the same no matter what modes you are going to be mixing.

The third step can be ignored if the mode you want at the top of the screen is the same as the one that uses the most memory. If not, as in our example (mode 2 is at the top of the screen, mode 1 uses the most memory), then we have to change the number in the memory location right before the beginning of the list. The table below shows what number to use for the mode at the top of the screen.

MODE	0	1	2	3	4	5	6	7	8
NUMBER	66	70	71	72	73	74	75	77	79

So, for example, we would need:

```
25 POKE BEGIN-1,71
```

Remember, only do this step if the first graphics row is *not* the same mode as the one that uses the most memory.

Now we just have to go down the list and change the numbers that need to be changed. The numbers for the graphics mode with the most memory are already correct, since we start in that mode. Therefore, all we have to change are the other numbers. In our example, that would be the numbers for mode 2 and mode 3. To make the necessary changes, we simply POKE BEGIN + row number with the correct number for the mode we want in that row. What are the correct numbers? Just subtract sixty-four from the numbers in the table I gave above. That would mean, for example,

seven for mode 2, and eight for mode 3. So we have:

```
30 POKE BEGIN +2,7:POKE BEGIN +3,7
```

which takes care of mode 2. Note that we didn't POKE BEGIN+1. This was automatically taken care of when we POKEd BEGIN–1 in line 25. Remember that we also don't have to worry about the numbers for mode 1, since they are already correct. Therefore, all that's left is to change the numbers for mode 3. Since we want sixteen rows of mode 3, which means changing sixteen numbers, we'll use a FOR/NEXT loop to make life easier:

```
40 FOR ROW =6 TO 21:POKE BEGIN+ROW,
8:NEXT ROW
```

Now the list has the correct mode numbers in it. There's still one more thing we must do. Since there may be a fewer number of rows now than there were in the mode we told the computer to start with, we have to tell the computer where the new end of the list is. We do this by POKEing the number 65 into the row number right after the last one we used. This tells the Atari to go back to the beginning of the list. We also tell it where the beginning is. For our example:

```
50 POKE BEGIN+22,65:POKE BEGIN+23,
PEEK(560):POKE BEGIN +24, PEEK(561)
```

And now we're done. Note that the only changes that you would need to make in line 50 when designing your own modes is in the numbers 22, 23, and 24. These are just the three row numbers after the last one you use on the screen.

How Often Do I Have To Do All This?

This whole procedure must be repeated whenever you want to use a specially designed graphics mode. You can't skip any of the steps except for the third one, and then only under the condition I already described.

So Now What Do I Do?

The last thing I'm going to cover is how to print and draw in your new mode. This only applies if the row you want to print or plot on is within the normal range for whatever mode it is. In simpler terms, if we had put the sixteen rows of mode 3 at the top of the screen, and mode 2 at the bottom, then mode 2 would have been in rows, 19, 20, and 21. But mode 2 usually only has twelve rows, so if you tried to print on line 19 you would get an error message. Now, there is a way around this, but it's somewhat complicated so I'm going to leave it for a future article. For now, however, you can use the

following rules as long as you stay within the normal range of the mode you're working with.

The first thing you have to do is POKE location eighty-seven with the number of the graphics mode for the row you want to PRINT or PLOT in. Next, POSITION the cursor and PRINT, or PLOT and DRAWTO. When you tell the Atari to POSITION X, Y or PLOT X, Y, the X value is still the number of spaces in from the left that you want to go. The Y value is still the number of rows down from the top that you want to go, but you may have to experiment with different values to get it exactly where you want it. Just make sure that you remember to POKE 87 with the mode number you're going to PRINT or PLOT in.

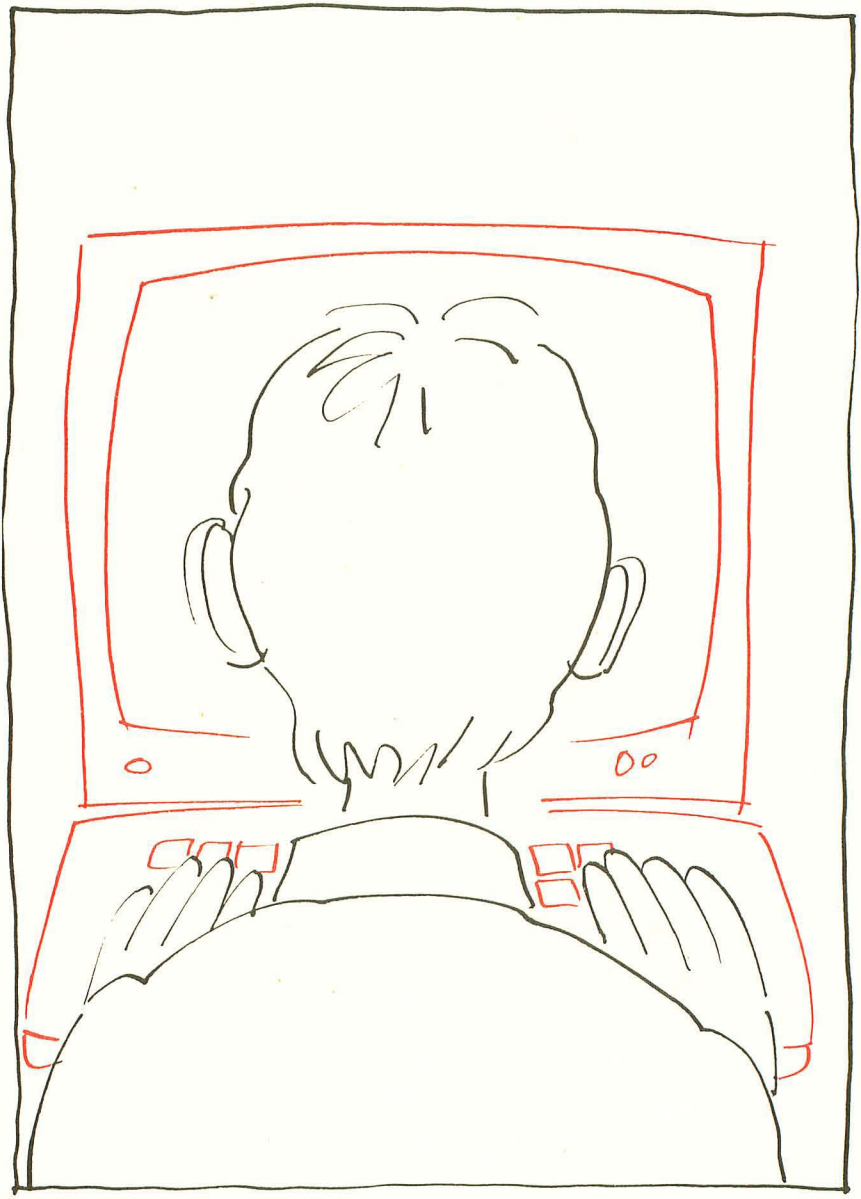
To help you understand what I just said, and to show off the example mode we've been working on, try entering these lines, as well as the other ones that are included throughout the article. When you've entered them in, just RUN the program, and BREAK in when you're done. Notice that the commands for colors are the same in the new mode; that is, you can still print different color letters and use the COLOR command for graphics points, etc. The one difficulty that might rise is when you mix mode 0 with other modes. Since mode 0 has a different background color (blue) than the other modes (black) you will have to use the SETCOLOR command to make the mode 0 rows invisible. Otherwise, you should have no problems whatsoever.

```
60 SETCOLOR 4,4,2:REM BACKGROUND
70 POKE 87,2:POSITION 6,0:PRINT #6;"THIS
 IS":POSITION 3,1:PRINT #6;"GRAPHICS MOD
E":POSITION 8,2:PRINT #6;"TWO"
80 POKE 87,1:POSITION 6,3:PRINT #6;"this
 is":POSITION 1,4:PRINT #6;"graphics mod
e one"
90 POKE 87,3:COLOR 3:FOR LINE=1 TO 3:PLO
T 15,LINE*5+8:DRAWTO 22,LINE*5+8:NEXT LI
NE:PLOT 22,13:DRAWTO 22,23
100 GOTO 100:REM KEEP GRAPHICS ON SCREEN
```

Look, Ma, New Modes!

That's all there is to making your own graphics modes on your Atari computer. The easiest way to make sense of everything I've covered here is to **experiment.** Start off by changing the example

program and watching what happens, and then try designing your own modes. Just a little practice and in no time you'll be an expert. Above all, have fun doing it; after all, the Atari works for you, not the other way around.

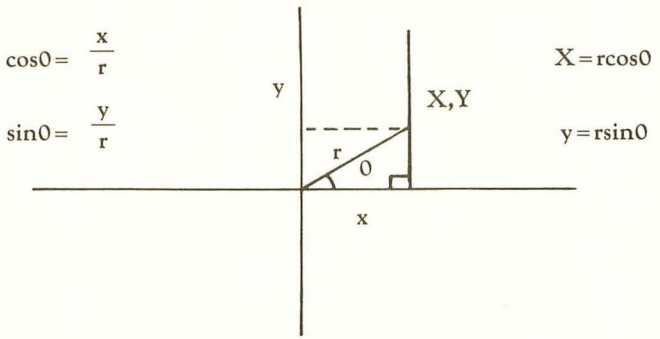


Graphics of Polar Functions

Henrique Veludo

One interesting type of program allows you to explore relationships between numbers and their visual analog. The routine here plots polar functions — this might not drive you wild until you realize that this means spirals and roses. A more seductive introduction to the beauty of math is difficult to imagine.

This program will plot polar functions such as roses, spirals, polygons, on the high screen of the ATARI 800, with input from the programmer. The general equations for converting the polar coordinates to rectangular coordinates are as follows:



First, the program will display a function menu (line 100), then ask the user to input which function to display, together with its parameters, INCR(element) and SC(ale). The INCR(element) is the interval in degrees that the computer uses to "increment" the angle T from 0° to 360°. One must decide whether the speed of execution or accuracy in plotting is preferable. A small INCR(element), e.g. 0.1, will draw a very accurate graph very slowly. A larger INCR(element), e.g. 5.0, will draw much faster and less accurately. An INCR of 1.0 is a good compromise. The SC(ale) is included to allow the graph to fill most of the screen. Without it, some functions will appear too small, others will be too large to plot. A SC(ale) between 10 and 100 should do for most functions. Lines 220 to 226 check for a 0 input that might confuse the program and display an

error message. Line 230 asks if the x-y axes are to be displayed and lines 390-395 display them. Lines 300-370 will select random colors and intensities (with enough separation to be visible). Lines 400-690 contain the calculation and plotting routines for x,y. In line 410 the variable U is included for use with the spiral function and dictates how many revolutions the spiral will have; it can be changed at line 222. Line 420 converts degrees to radians (in this context the program seems to work better with radians, but it could be converted to degrees, with the DEG function, and changing the values of the functions). Line 430 will direct the program to the proper function chosen in the input. Lines 610-620 calculate the x,y coordinates. Line 630 will check for an out-of-range cursor, stop the drawing, and avoid an error message. Line 670 will activate the buzzer to signal that the plotting is over. Lines 680-690 wait for a key to be pressed to clear the screen and return to the menu. If the buzzer sounds without anything being plotted, it means that the function is too large to plot. (Decrease the SC(ale) value to continue.) I chose to use randomly-selected colors. They could be chosen by the user in an input statement as well (where you input the parameters after the menu display).

Here are some values for the functions that work beautifully:

R=Q:SC=4:INCR=60
R=2(I-SIN(Q):SC=20
R=COS(2 SIN(6 (Q))):SC=90
R=SIN(COS(100 Q)):SC=90
R=COS(2 SIN(2 Q)):SC=90
R=I:INCR=45:SC=60 polygon

R=2(I+COS(Q)):SC=20
R=SIN(3(Q)):SC=80
R=SIN(4COS(2Q)):SC=90
R=COS(3SIN(Q)):SC=90
R=COS(SIN(100 Q)):SC=90
R=I:INCR=I20:SC=80 triangle

```
10 REM PROGRAM TO PLOT POLAR FUNCTIONS
20 REM BY HENRIQUE VELUDO FOR ATARI 800
80 DIM A$(1)
90 ? ")"
100 POSITION 7,1:? "GRAPHS OF POLAR FUNC
TIONS"
110 POSITION 2,3:? "FUNCTION MENU:":?
```

```
120 ? "      1)R=B*Q                    SPIRAL
"
130 ? "      2)R=A*(1+COS(Q))           CARDIO
ID"
140 ? "      3)R=A*(1-SIN(Q))"
150 ? "      4)R=A*SIN(B*Q)             ROSE"
160 ? "      5)R=A*COS(B*Q)"
170 ? "      6)R=COS(A*SIN(B*Q))"
180 ? "      7)R=SIN(A*COS(B*Q))"
190 ? "      8)R=A                      POLYGO
N"
200 ? :? :? "INPUT:":?
210 ? "FUNCTION #,A,B,INCR.,SC.";:INPUT
N,A,B,INCR,SC
220 IF N=0 THEN N=1
222 IF N=1 THEN U=4
224 IF A=0 THEN A=1
226 IF B=0 THEN B=1
230 ? :? :? "DO YOU WANT THE X-Y AXES DI
SPLAYED";
240 INPUT A$:IF A$(1,1)="Y" THEN W=1
300 COLOR 1:GRAPHICS 24
310 I=INT(RND(1)*16)
320 L1=INT(RND(1)*8)*2
330 L2=INT(RND(1)*8)*2
340 IF ABS(L1-L2)<4 THEN 320
350 SETCOLOR 4,I,L1
360 SETCOLOR 2,I,L1
370 SETCOLOR 1,I,L2
380 IF W<>1 THEN 410:REM --DISPLAY AXES?
390 FOR I=0 TO 319 STEP 4:PLOT I,96:NEXT
 I
395 FOR I=0 TO 191 STEP 3:PLOT 160,I:NEX
T I
400 REM ----PLOTTING CALCULATION
410 FOR T=0 TO 360*U STEP INCR
420 Q=T/57.3
430 ON N GOTO 510,520,530,540,550,560,57
0,580
500 REM ----EQUATIONS FOR R
510 R=B*Q:GOTO 610
```

```
520 R=A*(1+COS(Q)):GOTO 610
530 R=A*(1-SIN(Q)):GOTO 610
540 R=A*SIN(B*Q):GOTO 610
550 R=A*COS(B*Q):GOTO 610
560 R=COS(A*SIN(B*Q)):GOTO 610
570 R=SIN(A*COS(B*Q)):GOTO 610
580 R=A:GOTO 610
600 REM PLOTTING X,Y
610 X=INT((R*COS(Q))*SC)
620 Y=INT((R*SIN(Q))*SC)
630 IF ABS(X)>159 OR ABS(Y)>95 THEN 670
640 IF T=0 THEN PLOT 160+X,96-Y
650 DRAWTO 160+X,96-Y
660 NEXT T
670 FOR I=1 TO 75:POKE 53279,0:NEXT I
675 W=0
680 W=1:OPEN #1,4,0,"K:":GET #1,X:CLOSE
#1
690 PUT #6,125:GOTO 90
```

CHAPTER FOUR: Programming Hints

Reading the Atari Keyboard on the Fly

James L. Bruun

For most programs, the normal method of using the INPUT statement to get keyboard characters into a program is perfectly satisfactory. There are times, however, when we need to get a keystroke without stopping the program to wait for a key to be struck.

The ATARI computer has all the features needed to enable the programmer to check the keyboard without waiting for an INPUT statement to get the character. Memory location 764 retains a key code for the last key pressed. Further, when the RUN command is executed, that cell is set to 255 to indicate that no key has been pressed. During the running of a program, that location can be POKEd with a 255 to indicate that we've checked it since the last key was pressed.

Cell 764 is POKEd with 255 only if it isn't already finding it 255, then having a key pressed while the POKE 764, 255 instruction is being interpreted. This would cause the keystroke to be lost. In a long program the keystroke isn't often lost, but in a short program it happens quite often.

The following program illustrates the use of these features in a subroutine. First, initialize an I/O buffer and string variable.

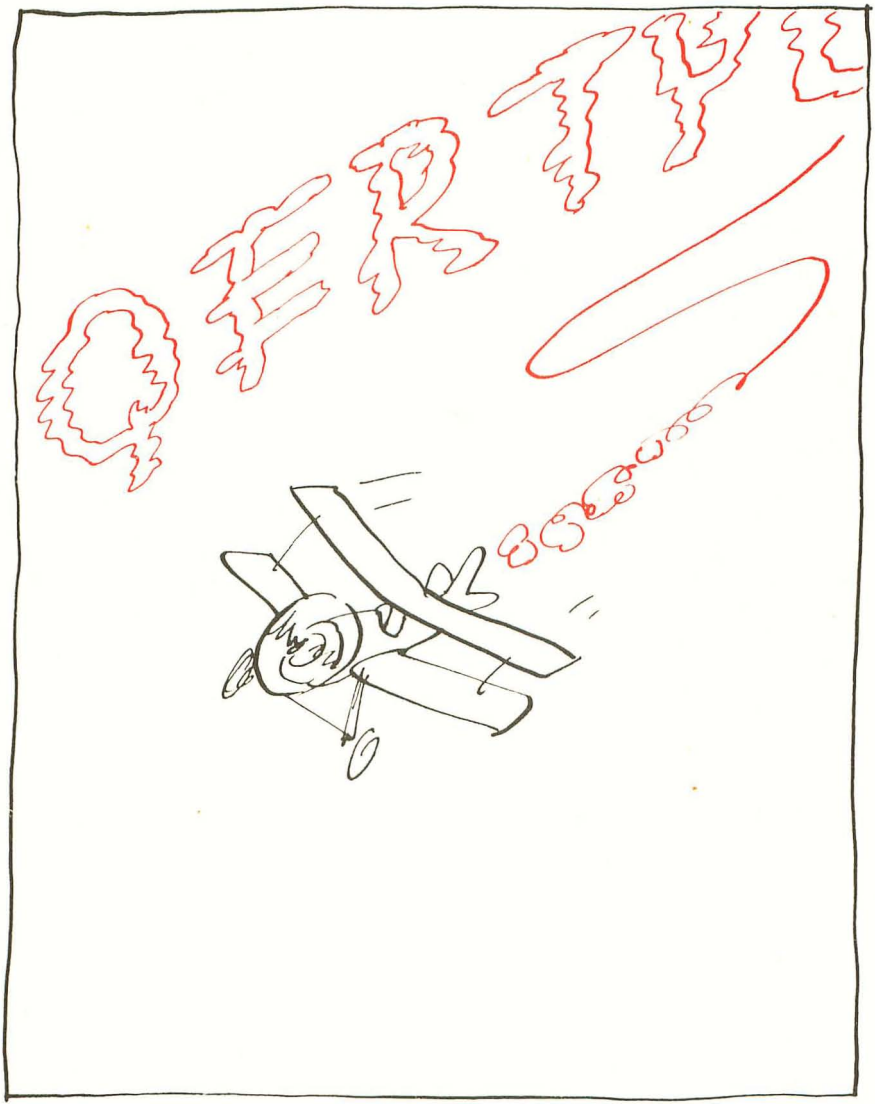
```
10 OPEN #1,4,0,"K:"
20 DIM CHAR$(1)
```

Then build the subroutine. Always precede your block of subroutines with an END statement to prevent accidental execution.

```
30 PRINT "{CLEAR}"
40 POKE 752,1
50 GOSUB 5000
60 IF CHAR=0 THEN 50
70 POSITION 5,5
80 PRINT "CHARACTER=(";CHAR$;")"
90 GOTO 50
4999 END
5000 CHAR=0
```

```
5010 IF PEEK(764)<>255 THEN GET #1,CHAR:
CHAR$=CHR$(CHAR)
5020 RETURN
```

Most programs that would need this feature would perhaps be doing complex things if the keystroke has not occurred, but in this one we have chosen to "do nothing" until a key is pressed.



Atari Sounds Tutorial

Jerry White

This program was designed to help you discover some of the amazing sounds of Atari. You will enjoy experimenting with this program and learn at the same time. Here's how it works:

We will use two FOR-NEXT loops to alter the volume and pitch variables of the SOUND command. You will be prompted to type the required data. The program will then execute using your data and you will hear the sound you created. Here is sample data for you to use to get the feel of the program. Respond to the prompts with Dist 10, Pitch 20, L1 from 15, L1 to 0, L1 step -0.5, L2 from 3, L2 to 0, L2 step -1. Notice how the sound seems to vibrate as it fades. If you want to hear it again, just hit the option key.

You may want to use that sound in a program you write. At this point you will notice a Basic subroutine is displayed near the top of the screen. Make note of it and any other interesting sounds you come up with. Start a library of subroutines. When you're ready to try a new sound, hit the START key.

There are a few other useful routines in this program you may want to study. Lines 12 and 14 will show you how to use random color. You will find extensive error trapping of input routines. The loop from line 410 to 440 shows how to make use of the OPTION and START keys. To see if the SELECT key has been pressed, PEEK at 53279 and see if it equals 5.

When you type in line 340, type those messages using inverse video. The routine from line 300 to line 380 will cause these messages to flash. To further dress up your display, I suggest you also use inverse video for the messages at lines 10, 130, and 6000.

After you've used and studied this program for a while, you will begin to realize that the variety of possible sounds is almost endless. Now consider this. You have been using only one of the four voices available. The four voices can be used at the same time. You control the volume, pitch, and distortion of each voice. Take it away, imagination!

```
0 REM SOUNDS PROGRAM BY JERRY WHITE 8/28
/80
1 GRAPHICS 0:DIM X$(1),BL$(20):BL$="
                  "
3 ? :? "PITCH=ANY NUMBER FROM 0 THRU 255
```

```
.":? "WE WILL MOVE THE PITCH IN LOOP 2."

4 ? :? "L1=OUTER LOOP 1 VOLUME.":? "TYPE
 ANY NUMBER FROM 0 THRU 15 AT PROMPT FOR
M, TO, AND STEP."
5 ? :? "L2=INNER LOOP 2 PITCH.":? "TYPE
ANY NUMBER FORM 0 THRU 255":? "AT PROMPT
 FROM, TO, AND STEP"
7 ? :? "HIT RETURN TO BEGIN";:INPUT X$
10 GRAPHICS 0:? :? "  SOUND TEST  "
12 POKE 752,1:C=RND(0)*16:REPEAT=0
14 SETCOLOR 1,C,2:SETCOLOR 2,C,8:SETCOLO
R 4,C,2
30 POSITION 2,3:? "TYPE DIST    ";:TRAP
34:INPUT D:TRAP 40000
32 IF D=0 OR D=2 OR D=4 OR D=8 OR D=10 O
R D=12 OR D=14 THEN 36
34 POSITION 2,3:? BL$:GOTO 30
36 POSITION 2,5:? "TYPE PITCH    ";:TRAP
 40:INPUT P:TRAP 40000
38 IF P<255 THEN 42
40 POSITION 2,5:? BL$:GOTO 36
42 POSITION 2,7:? "TYPE L1 FROM  ";:TRAP
 46:INPUT F1:TRAP 40000
44 IF F1<33 THEN 48
46 POSITION 2,7:? BL$:GOTO 36
48 POSITION 2,9:? "TYPE L1 TO    ";:TRAP
 52:INPUT T1:TRAP 40000
50 IF T1<33 THEN 54
52 POSITION 2,9:? BL$:GOTO 48
54 POSITION 2,11:? "TYPE L1 STEP ";:TRAP
 58:INPUT S1:TRAP 40000
56 IF S1<33 THEN 60
58 POSITION 2,11:? BL$:GOTO 54
60 POSITION 2,13:? "TYPE L2 FROM ";:TRAP
 64:INPUT F2:TRAP 40000
62 IF F2<256 THEN 70
64 POSITION 2,13:? BL$:GOTO 60
70 POSITION 2,15:? "TYPE L2 TO   ";:TRAP
 74:INPUT T2:TRAP 40000
72 IF T2<256 THEN 80
74 POSITION 2,15:? BL$:GOTO 70
```

```
80 POSITION 2,17:? "TYPE L2 STEP ";:TRAP
 84:INPUT S2:TRAP 40000
82 IF S2<256 THEN 100
84 POSITION 2,17:? BL$:GOTO 80
100 IF REPEAT>0 THEN GOSUB 5000:GOTO 400

120 GOSUB 5000:SOUND 0,0,0,0:? CHR$(125)

130 ? :? "  YOUR SOUND SUBROUTINE: "
140 ? :? "100 FOR L1=";F1;" TO ";T1;" ST
EP ";S1
160 ? "110 FOR L2=";F2;" TO ";T2;" STEP
";S2
180 ? "120 SOUND 0,";P;"-L2,";D;",L1"
200 ? "130 NEXT L2":? "140 NEXT L1":? "1
50 RETURN"
280 FOR DELAY=1 TO 500:NEXT DELAY
300 FOR TIME=1 TO 5:POSITION 2,20:? "  H
IT START TO RESTART":? "  HIT OPTION TO
REPEAT "
320 FOR DELAY=1 TO 10:NEXT DELAY
340 POSITION 2,20:? "  HIT START TO REST
ART":? "  HIT OPTION TO REPEAT "
360 FOR DELAY=1 TO 10:NEXT DELAY
380 NEXT TIME
400 SOUND 0,0,0,0
410 IF PEEK(53279)=6 THEN 10
420 IF PEEK(53279)=3 THEN 500
440 GOTO 410
500 REPEAT=REPEAT+1:GOTO 100
5000 FOR L1=F1 TO T1 STEP S1
5100 FOR L2=F2 TO T2 STEP S2
5200 TRAP 6000:SOUND 0,P-L2,D,L1:TRAP 40
000
5300 NEXT L2:NEXT L1:RETURN
6000 ? :? "    INVALID SOUND, TRY AGAIN.
    ":SOUND 0,0,0,0
6100 FOR DELAY=1 TO 250:NEXT DELAY
6110 GOTO 10
```

Al Baker's Programming Hints for Atari/Apple

Al Baker

Exploring joysticks . . .

Programming is the most complex and least organized of human endeavors. Well, maybe after the U.S. Government and raising children. Many people have tried to bring order out of this chaos. In this column, I will join that noble company. With your help, we just might pull it off.

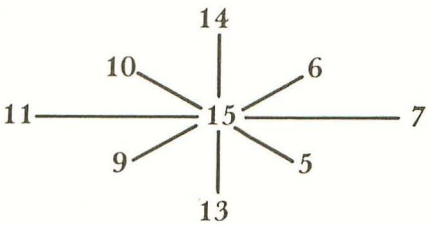
Two of my favorite computers are the Apple and Atari. They are superbly designed machines. (It's not that I don't like the PET. I do. I guess I'm just hopelessly addicted to sound and color, joysticks and paddles.) In this column, I am going to help you use the sound, color, and attachments of these two computers.

In each issue, I will show you one or two short routines fully utilizing some feature of an Atari or Apple. I'll put the routines to work and leave you with a chance to work on a programming exercise, answered in the next issue.

I said I needed your help. Send me any routines you have and would like to share. If I use them, you'll be given credit as the source.

The Atari Joystick

This month, we are going to explore the Atari joystick. The position of a joystick is read with the function STICK. The joysticks are numbered from 0 to 3. Thus, the position of the second joystick is STICK(1). This function returns the number 15 when the joystick is centered. Here are the results of STICK for the other joystick positions.



The button on the joystick is read with the function STRIG. STRIG(0) reads the button of the first joystick. The function is zero if the button is pushed and one if the button is not pushed.

Most programs use the joystick to move objects around on the screen. As soon as the program needs a yes or no response, however, the players must use the keyboard. This is inconvenient, especially when there are several players, none sitting close to the keyboard. Why not use the joystick to make the selection?

Two Entry Menu Selection

Look at the first listing. This is a routine which uses the joystick to get a yes or no response from a player. Line 45 turns off the cursor on the screen. Lines 60 through 140 set the default answer and display the options, YES NO, on the screen. The default answer, in this case YES, is highlighted in reverse video.

The routine assumes that the word YES is to the left of the word NO. If the joystick is moved to the left, then lines 180 through 240 set the answer to Y and highlight the word YES on the screen. If the joystick is moved to the right, then lines 280 to 340 set the answer to N and highlight the word NO. Pushing the button ends the routine. This is handled in line 380. The IF statement is true if STRIG (PLAYER-1) is 1. Remember that this means the button is not pushed. The program loops back to line 180 as long as the button is not pushed.

Lots of lines and REM statements take up memory and slow the program down. Look at the second listing. Here is a short program which uses the menu selection routine. The routine has been compressed into lines 1000 through 1050. The program needs no explanation. Play it and get some feel for the convenience of using the joystick instead of the keyboard.

At the tone the number is . . .

Listing 3 is another joystick input routine. This time we are using the joystick to input a number. It is similar to the first routine. Lines 60 through 140 set the default input number and print it on the screen. Notice that line 120 prints a blank after the number. This prevents garbage from appearing on the screen if 'A' goes from 2 digits to 1 digit.

Look at lines 130 through 140. This generates a muted bell sound, very similar to striking a xylophone. The SOUND statement has four parameters. The first is the sound register. This can be any number from 0 to 3. Up to four sounds can be created at one time. The second parameter is the pitch of the sound. The higher the

number, the lower the pitch. Using 100-A gives a pitch that goes up as A gets bigger and goes down as A gets smaller.

The third parameter is the sound quality. A 10 gives a clear tone. The fourth parameter is the loudness of the note. The NOTE goes from 15−0 = 15 or loud to 15−15 = 0 or quiet. This creates the bell effect.

Lines 180 through 240 decrease the input number as long as the joystick is pushed to the left. Lines 280 through 340 increase the input number as long as the joystick is pushed to the right. Line 380 ends the routine if the button is pushed.

Conclusion

Next time we will compress the number input routine and use it in a program. Try it yourself and let's see who does a better job at compressing it! We'll also try our hand at using the Apple paddles to do a menu select.

Al Baker is Programming Director of The Image Producers, Inc., 615 Academy Dr., Northbrook, IL 60062.

Program 1. Two Entry Menu Select

The words in the boxes are typed using the Atari key to put them in reverse video.

```
10 REM ... TWO ENTRY MENU SELECT ...
20 REM          FROM JOYSTICK
30 REM
40 REM
43 REM TURN OFF CURSOR
45 POKE 752,1
47 REM
50 REM DEFAULT ANSWER:
60 A$="Y"
70 REM
80 REM  DISPLAY MENU
90 REM  XY,YY IS POSITION OF YES
100 REM XN,YN IS POSITION OF NO
105 REM
110 POSITION XY,YY
120 PRINT "YES";
130 POSITION XN,YN
140 PRINT "NO";
150 REM
```

```
160 REM SCAN JOYSTICK FOR YES
170 REM
180 IF STICK(PLAYER-1)<>11 THEN 280
190 A$="Y"
200 POSITION XY,YY
210 PRINT "YES";
220 POSITION XN,YN
230 PRINT "NO";
240 GOTO 180
250 REM
260 REM SCAN JOYSTICK FOR NO
270 REM
280 IF STICK(PLAYER-1)<>7 THEN 380
290 A$="N"
300 POSITION XY,YY
310 PRINT "YES";
320 POSITION XN,YN
330 PRINT "NO";
340 GOTO 180
350 REM
360 REM SCAN TRIGGER FOR CHOICE
370 REM
380 IF STRIG(PLAYER-1) THEN 180
390 REM
400 REM WE HAVE ANSWER
410 REM
420 PRINT A$
```

Program 2. Do You Love Me?

```
10 REM ... DO YOU LOVE ME ...
20 REM
30 REM
40 REM DECLARE STRINGS AND CONSTANTS
50 DIM A$(1)
60 C1=1
70 REM
80 REM ASK MY OWNER IF HE LOVES ME
90 REM
95 GRAPHICS 0
100 XY=12:YY=18
```

```
110 XN=24:YN=18
120 PLAYER=1
130 POSITION 12,10
140 PRINT "DO YOU LOVE ME?"
150 GOSUB 1000
160 REM
170 REM RESPOND TO ANSWER
180 REM
190 POSITION 3,22
200 IF A$="Y" THEN PRINT "     SHUCKS, I
 LOVE YOU TOO."
210 IF A$="N" THEN PRINT "WELL, I LOVE Y
OU ANYWAY. (SNIFFLE)"
220 FOR DELAY=1 TO 1000
230 NEXT DELAY
240 GOTO 95
960 REM
970 REM JOYSTICK ROUTINE
980 REM (DISCUSSED ELSEWHERE)
990 REM
1000 POKE 752,C1:A$="Y"
1010 POSITION XY,YY:? "YES";:POSITION XN
,YN:? "NO";
1020 IF STICK(PLAYER-C1)=11 THEN 1000
1030 IF STICK(PLAYER-C1)=7 THEN A$="N":P
OSITION XY,YY:? "YES";:POSITION XN,YN:?
"NO";:GOTO 1020
1040 IF STRIG(PLAYER-C1) THEN 1020
1050 RETURN
```

Program 3. Number Select

```
10 REM      ... NUMBER SELECT ...
20 REM          FROM JOYSTICK
30 REM
40 REM
43 REM TURN OFF CURSOR
45 POKE 752,1
47 REM
50 REM DEFAULT ANSWER:
60 A=10
```

```
70 REM
80 REM  DISPLAY NUMBER:
90 REM  X,Y IS POSITION OF NUMBER
105 REM
110 POSITION X,Y
120 PRINT A;" ";
130 FOR SND=0 TO 15
135 SOUND 0,100-A,10,15-SND
140 NEXT SND
150 REM
160 REM SCAN JOYSTICK FOR SUBTRACT
165 REM DON'T GO BELOW LOW LIMIT
170 REM
180 IF STICK(PLAYER-1)<>11 THEN 280
185 IF A=LOW THEN 180
190 A=A-1
240 GOTO 110
250 REM
260 REM SCAN JOYSTICK FOR NO
265 REM DON'T GO ABOVE HIGH LIMIT
270 REM
280 IF STICK(PLAYER-1)<>7 THEN 380
285 IF A=HIGH THEN 180
290 A=A+1
340 GOTO 110
350 REM
360 REM SCAN TRIGGER FOR CHOICE
370 REM
380 IF STRIG(PLAYER-1) THEN 180
390 REM
400 REM WE HAVE ANSWER
410 REM
420 PRINT A
```

Al Baker's Programming Hints: Apple and Atari

Al Baker

During games it is often easier to use a joystick than to play musical chairs trying to share the console between two or more people. The subroutine is from line 1000 up is useful in such applications.

Last Issue: Atari

I left the Atari readers with a problem last time: condense the number selection routine as much as possible and use it in a program. If you'd like to share your solution with the rest of us, send me a listing. My solution is in Listing 1. The program is the old favorite "Guessing Game."

The routine is condensed into lines 1000 to 1050. I made a few changes in it to accommodate the game. The main change was to remove the setup of the variable "A". The rest of the program is the standard number guessing program. Lines 7 through 23 initialize the variables, including "A", and lines 30 through 80 pick out a random number and ask the player to guess it.

Line 90 calls the joystick number selection routine. If the player makes a correct guess, then lines 200 to 220 tell him so and loop back for another game. Otherwise lines 117 to 140 give him a Bronx cheer, tell him how he was wrong, and loop back for another guess.

```
1 REM          GUESS A NUMBER
2 REM
3 REM
5 REM      SET UP THE JOYSTICK DATA
6 REM
7 A=10
10 LOW=1
20 HIGH=20
21 X=17
22 Y=12
23 PLAYER=1
27 REM
28 REM        PLAY THE GAME
```

```
29 REM
30 GRAPHICS 0
40 POSITION 2,5
50 ? "I AM THINKING OF A NUMBER BETWEEN"

60 ? LOW;" AND ";HIGH;".  ";
70 ? "WHAT IS YOUR GUESS:"
80 GUESS=INT(RND(0)*20)+1
82 REM
84 REM      GET THE PLAYER'S ANSWER
86 REM
90 GOSUB 1000
100 POSITION 14,20
110 IF A=GUESS THEN 200
112 REM
114 REM    WRONG GUESS
116 REM
117 SOUND 0,200,10,15
118 FOR I=1 TO 50:NEXT I
119 SOUND 0,0,0,0
120 IF A<GUESS THEN ? "TRY HIGHER"
130 IF A>GUESS THEN ? "TRY LOWER "
140 GOTO 90
170 REM
180 REM       CORRECT GUESS
190 REM
200 ? "YOU GOT IT"
210 FOR I=1 TO 500:NEXT I
220 GOTO 30
970 REM      JOYSTICK NUMBER SELECT
980 REM       (DISCUSSED LAST ISSUE)
990 REM
1000 POKE 752,1
1010 POSITION X,Y:? A;" ";:FOR SND=0 TO
15:SOUND 0,100-A,10,15-SND:NEXT SND
1020 IF (STICK(PLAYER-1)=11)*(A>LOW) THE
N A=A-1:GOTO 1010
1030 IF (STICK(PLAYER-1)=7)*(A<HIGH) THE
N A=A+1:GOTO 1010
1040 IF STRIG(PLAYER-1) THEN 1020
1050 RETURN
```

Error Reporting System for the Atari

Len Lindsay

One of the disappointing aspects of the Atari Computer System is its lack of user-oriented messages. Particularly disturbing is the error message, or should I say error number? It stops and tells you

ERROR 138

What? Where did I put my manual? You then search through your desk, find the manual, flip pages until you hit the error messages, and look up number 138. If you have a disk system, the following program will do all the work for you, as well as offer you several options for continuing program execution. (Non-disk users will also find several aspects of the program suitable for use without a disk).

Here is what the program does for you each time an error is encountered:

1) It reports to you that an error was encountered and gives you the error number and the line number where the error was encountered.

2) If you have an error messages diskette in drive 1 it will next print out an error message in plain English, telling you what went wrong and possibly how to correct it. (Without a disk you won't get this message but all the rest of the program works fine).

3) It offers you the choice of ending program execution or of continuing in one of three ways:
a) continue with the line on which the error was encountered.
b) continue with the line immediately following the error line.
c) continue with the LINK line (equivalent to the TRAP function).

That is the system in a nutshell. It is structured to be of general use and should be modified to your particular needs. To aid in this, I will explain how the program works.

Program Explanation

LINE 0 is the required DIM statements for string variables used in the system.

LINE 1 sets the TRAP to 32500 — the start of the reporting system.

NOTE: The TRAP command cannot be used in your program. Instead, simply set the variable LINK to the line you normally would have used for TRAP. Example:

```
250 TRAP 5000
should be entered as:
250 LINK = 5000
```

LINE 32500 finds the line number in which the error occurred. It also finds the error number.

LINE 32510 prints the error number and the line at which it occurred.

LINES 32520-32530 assigns a file name to be used to recover the appropriate error message from disk.

LINE 32540 sets a TRAP to report a default message if an error occurs while retrieving the error message (for instance, if your disk is turned off, or if you have no disk).

LINE 32550 opens the appropriate disk file and, if successful, skips over the default message.

LINE 32570 gets the error message from disk.

LINE 32580 jumps to the subroutine to find what the next line after the error line is. It also resets the TRAP for future operation.

LINES 32581-32587 print your options.

LINE 32588 ask for your choice.

LINE 32589 clears the screen.

LINE 32590 turns off the TRAP and ENDs if you hit "S" (for STOP).

LINES 32591-32593 check for other legal choices and go to the appropriate line.

LINES 32599 jumps back to print your options once again if an illegal entry is detected.

LINE 32600 starts the routine to find the next line number after the error line. The variable NXLINE is initialized.

LINE 32610 finds the first line number in the program.

LINES 32620-32660 finds the line number by starting at the first line and checking one line at a time until it hits the error line. The next line is then used for the next line number.

LINE 32699 Returns back to the line calling this routine.

That's it!

```
0 DIM ERNUM$(5),ERFILE$(12),XA$(100)
1 TRAP 32500:REM TO ERROR REPORT ROUTINE
2 REM %%% ERROR REPORT SYSTEM by
3 REM %%% LEN LINDSAY (C) 1980
4 REM YOUR PROGRAM GOES HERE
5 REM SET VARIABLE  LINK  TO THE
6 REM BEGINNING LINE OF YOUR MODULES
7 REM - NEEDS A DISKETTE IN DRIVE 1
8 REM WITH ERROR FILES CREATED WITH
9 REM THE ERROR FILE WRITING PROGRAM
10 REM THANK YOU TO COMPUTE, IRIDIS, AND
 ATARI FOR INFO USED IN THIS
32500 ERLINE=256*PEEK(187)+PEEK(186):ERN
UM$=STR$(PEEK(195)):REM ERROR REPORT SYS
TEM
32501 REM %%% NEEDS: DIM ERNUM$(5)
32502 REM %%%         DIM ERFILE$(12)
32503 REM %%%         DIM XA$(100)
32504 REM %%% USES SUBROUTINE 32600 TO F
IND NEXT LINE
32510 PRINT ")ERROR NUMBER ";ERNUM$;" IN
 LINE ";ERLINE
32520 ERFILE$="D:ERROR"
32530 ERFILE$(LEN(ERFILE$)+1)=ERNUM$
32540 TRAP 32560
32550 OPEN #5,4,0,ERFILE$:GOTO 32570
32560 PRINT "ERROR NUMBER ";ERNUM$;" IS
NOT ON FILE":GOTO 32580
32570 INPUT #5;XA$:PRINT XA$:CLOSE #5
32580 GOSUB 32600:TRAP 32500
32581 PRINT " SHALL I :"
32582 PRINT " STOP"
32583 PRINT "    OR "
32584 PRINT " CONTINUE WITH :"
32585 PRINT " ERROR LINE ";ERLINE
32586 PRINT " NEXT LINE ";NXLINE
32587 PRINT " LINK LINE ";LINK
32588 PRINT "  WHICH CHOICE";:INPUT XA$
32589 PRINT ")":REM CLEAR SCREEN
32590 IF XA$="S" THEN TRAP 34567:STOP
32591 IF XA$="E" THEN GOTO ERLINE
```

```
32592 IF XA$="N" THEN GOTO NXLINE
32593 IF XA$="L" THEN GOTO LINK
32599 GOTO 32581:REM INVALID RESPONSE
32600 NXLINE=0:REM FIND NEXT LINE NUMBER
32601 REM *** ERLINE   IS INPUT TO THIS
ROUTINE AS THE LINE NUMBER
32602 REM *** NXLINE  IS RETURNED AS THE
 NEXT LINE NUMBER
32605 REM *** BASED ON COMPUTE #4 PAGE 3
2 PROGRAM LISTING
32610 ADDRESS=PEEK(136)+PEEK(137)*256:RE
M GET THE FIRST LINE NUMBER
32620 LINE=PEEK(ADDRESS)+PEEK(ADDRESS+1)
*256
32630 IF NXLINE=1 THEN NXLINE=LINE:GOTO
32699
32640 IF LINE=ERLINE THEN NXLINE=1
32650 ADDRESS=ADDRESS+PEEK(ADDRESS+2)
32660 GOTO 32620
32699 RETURN
```

In order to fully use the Error Report System you must have a diskette with all the error messages correctly recorded on it. The following program can be used to create your own custom-made error messages master diskette. It simply asks you for an error number and its matching message. The message is then written to disk under the appropriate error number file.

```
0 REM *** ERROR REPORT WRITER
1 REM *** (C) 1980
2 REM *** LEN LINDSAY
3 REM *** PUTS ERROR INFO TO DISK
10 DIM ERNUM$(5),ERFILE$(12),XA$(100)
90 PRINT "{CLEAR}":REM CLEAR SCREEN
100 PRINT "WRITE ERROR MEANINGS TO DISK"

110 PRINT " GET OUT YOUR ERROR LIST - LE
TS GO-"
120 TRAP 120:PRINT "  WHAT IS THE NEXT E
RROR NUMBER ";:INPUT ERNUM$
125 E=VAL(ERNUM$):TRAP 34567
```

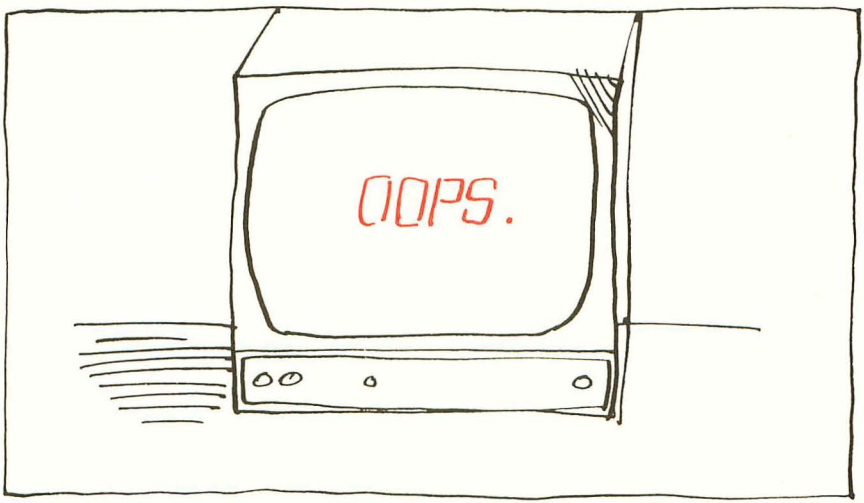
```
130 ERFILE$="D:ERROR"
140 ERFILE$(LEN(ERFILE$)+1)=ERNUM1$
150 PRINT "  PLEASE TYPE IN ITS MEANING
& HINTS":INPUT XA$
160 OPEN #1,8,0,ERFILE$
170 PRINT "}NOW WRITING ERROR NUMBER ";E
RNUM$
180 PRINT #1;XA$:CLOSE #1
190 GOTO 120
```

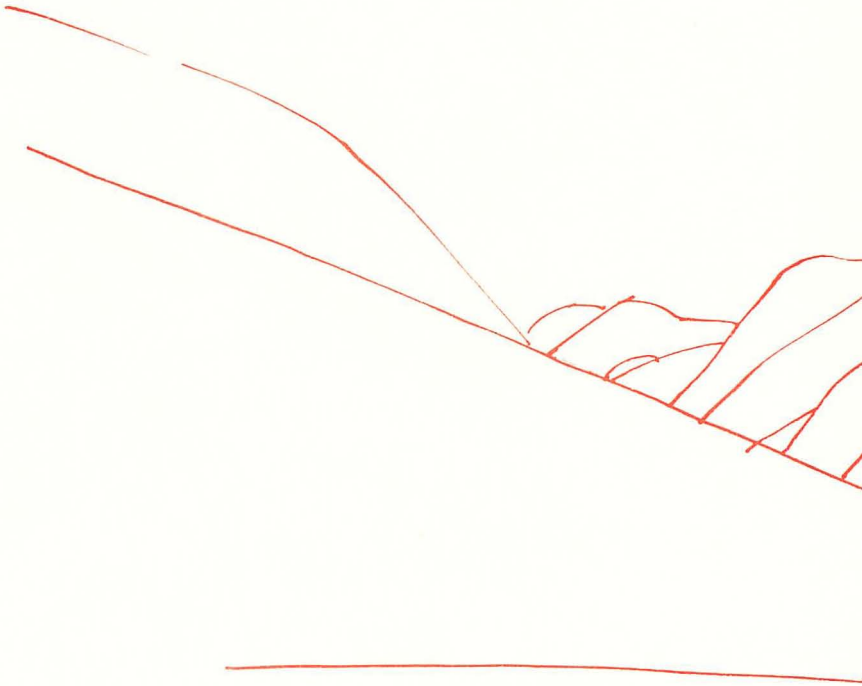
Possible System Uses or Modifications

The error reporting system can be used while developing your programs, providing you with messages during your next run as well as with several restart options. The system is presently under manual control after an error is encountered. This of course can be automated to provide error trapping AND error correction.

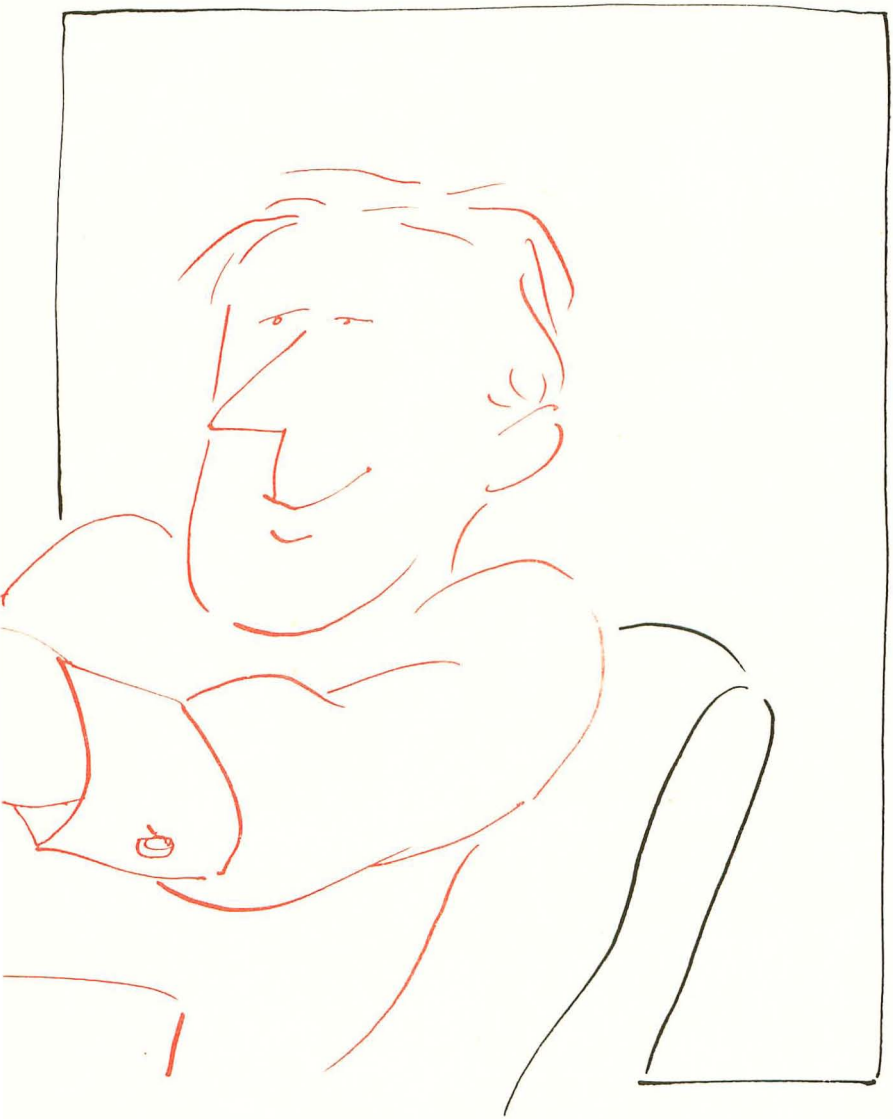
For example, your program may provide a hardcopy printout of the program results. If an error #138 is encountered, you may wish to print a message on the screen such as "Please turn on the printer" and then go back to the offending line. Print a cursor-up after the message and you can loop until the printer is turned on, after which the program immediately continues executing.

You may also be able to use pieces of this system in your own programs. For example, lines 32520-32530 show how your program can dynamically create its own disk file name, based on the value of variables.





CHAPTER FIVE: Applications



Atari Tape Data Files: A Consumer Oriented Approach

Al Baker

This complements Larry Isaacs' article "Inside Atari BASIC." The technique presented here is very useful for cassette users.

Introduction

This article is based on a major axiom of consumer computing:

> **Easier is Better**

The specific corollary when writing a program which saves data between program runs is:

> *Use only one tape. Program and data should be on the same tape. They should, in fact, be the same thing.*

A consumer should be able to load his program, run it to update his checkbook and balance his budget, and then save the program on tape when done. The next day, he can load his program and all data changes from the previous day should be there.

"Impossible," you say? Well, perhaps. It is certainly impossible on some of the computers on the market. But it is not impossible on the Atari. The trick is to fool Atari Basic into saving all dimensioned variables when a program is saved to tape. We won't try to save the simple variables. Since I am not a revered expert, I won't make the mistake of saying this is impossible. (But, I think it's impossible.) Saving the dimensioned variables with a program is relatively easy.

Write Your Program

Listing 1 is a simple program. Nothing tricky. But notice that I print the dimensioned variables in Lines 70-130 and then assign values to them in Lines 140-190. I am assuming the variables have valid contents before changing them! The only important restriction here is to type the line containing the DIM statement first. It doesn't have to be the first line in the program. Just make sure it is the first line *typed*.

The Atari Basic variable symbol table is constructed when each line is typed in, not when the program is run. Later we will need to find the locations of the string variables in the table. This is easier if they are the first variables present. For a more complex discussion of the symbol table, see the text in the box.

```
50 DIM A$(10),B(2,3)
70 ? A$
80 FOR I=0 TO 2
90 FOR J=0 TO 3
100 PRINT B(I,J),
110 NEXT J
120 PRINT
130 NEXT I
140 ? "STRING=";:INPUT A$
150 ? "I=";:INPUT I
160 IF I=9 THEN 200
170 ? "J=";:INPUT A:B(I,J)=A
190 GOTO 150
200 END
```

Suppose the program is already written and you didn't type the DIM statement first. Write your program to tape using the command LIST"C". Type NEW. Now type the DIM statement from your program with the string variables first. Finally, reload the program from tape with the command ENTER "C". Now the string variables are at the beginning of the variable tables.

Protect The Dimensioned Variables

The next step is to fool Basic into treating the dimensioned variables as part of the program. Also, you have to add the code to let the program save itself to tape. In an application, saving the program to tape will be the final program option selected by the user. In Listing 2 this is added to the program in lines 200 through 230.

```
50 DIM A$(10),B(2,3)
70 ? A$
80 FOR I=0 TO 2
90 FOR J=0 TO 3
100 PRINT B(I,J),
110 NEXT J
120 PRINT
130 NEXT I
140 ? "STRING=";:INPUT A$
150 ? "I=";:INPUT I
160 IF I=9 THEN 200
170 ? "J=";:INPUT A:B(I,J)=A
190 GOTO 150
200 A=PEEK(140)+PEEK(141)*256
210 A=A+82
220 POKE 141,INT(A/256):POKE 140,A-PEEK(141)*256
230 CSAVE
```

Locations 140 and 141 contain the address of the end of the computer program. Program line 200 places this address in the variable A. In line 210 we add the size of the dimensioned variables. Each string variable contains as many bytes as its dimension. Each numeric array contains 6 times the number of members of the array. The B array is 6x(2+1)x(3+1)=6x3x4=72 bytes. Thus we had to add 10+72 or 82 to the end of the program in the example.

Now run the program and let the internal CSAVE create a tape. Turn the computer off and then on. Now reload the newly created program from tape. For some reason this step is important. (I don't know why.) If you do not use the new tape, this procedure won't work.

Finish The Program

We now have a program in memory which has an invalid program-end pointer. See the third listing. Add lines 10 through 40 to your program. Make sure that you use the correct number instead of "-82" in line 10. Remember that this number is the size of your dimensioned variables.

Refer to Table 1. Locations 140 and 141 form the program-end address. Locations 142 and 143 form the stack address and locations 144 and 145 form the pointer to the end of memory used by the program. The RUN command sets all of them equal to the incorrect end-of-program pointer. Lines 10 through 40 correct them.

Here comes the only hard part. You are going to have to PEEK around in memory. The RUN command sets the length of all

Table 1.

These two byte addresses point to important areas used by Atari Basic.

Use this	To get the location of this
PEEK(130)+PEEK(131)*256	Variable name table
PEEK(134)+PEEK(135)*256	Variable value table
PEEK(136)+PEEK(137)*256	Beginning of program

Use these only when program running

PEEK(140)+PEEK(141)*256
End of program and beginning of dimensioned variables

PEEK(142)+PEEK(143)*256
End of dimensioned variables and beginning of stack

PEEK(144)+PEEK(145)*256
End of memory used by program

strings to zero. You must repair their lengths if you want to save string data.

Table 2.

The variable name table: Entry lengths are different. Box symbolizes that 128 is added to ACSII value of last character to show the name's end.

Variable	Variable name	
AB1	AB [1]	3 character number name
AR(3,4)	AR [(]	2 character array name
CDOG(17)	CDOG [(]	4 character array name
ALPHA$ (10)	ALPHA [$]	6 character string name
E	[E]	1 character number name
FIG	FI [G]	3 character number name

Note: Variable names can be up to 120 characters long and are completely unique. Variable ABC is different from variable ABCD. Variable names DO NOT appear in the program in memory. Only a 1 byte pointer to the variable name in the variable name table appears.

Look at Table 3. The third entry in the variable value table is the string ALPHA$. Its current length is 5+0*256 or 5. These two bytes must be set to the correct length of the string. Type command: PRINT PEEK(134)+PEEK(135)*256. Now you know where the variable value table is. If you have been writing the program in the listings, you should get the answer 2056. Assume the string is the first entry in the table. The location of the length is 2060 and 2061. Since the length of the string of data being saved in the example is 10, I set location 2060 to a 10 in line 60 of the program.

Try it out

The program is complete. SAVE it. Now RUN it. You will probably get garbage in the printout. Put a 10 character string in the string variable. Now put numbers in various entries in the B array. Typing a 9 for the I subscript will end the program with a CSAVE. Do this CSAVE onto a new tape. Turn the computer off and on. Now load this new copy of the program and RUN it. Viola! The data is still there! Now just imagine that this was your budget information, address book or other files. You have a no-hassle, one-tape system.

Conclusion

I have provided more information about the internals of the Atari

Table 3

The variable value table: Each entry is eight bytes.

Variable	Contents	Table Entry 1 2 3 4 5 6 7 8	Meaning
ABI	5	0/ 0/64/5,0, 0,0, 0	First byte is 0: this is a number. Second byte is 0: this is the first entry. 64 is the exponent. 5 is the binary coded decimal value.
AR(3,4)	doesn't matter	64+1/1/0, 0/4, 0/5,0	64 makes this an array. +1 means that it has been dimensioned. "+0*256 is the displacement into the array area. 4+0*256 is the size of the first dimension and 5+0*256 is the size of the second dimension.
CDOG(17)	doesn't matter	64+1/2/120,0/18,0/1, 0	This array is displaced 120 bytes into the array area, and it is dimensioned 18+0*256 by 1+0*256.
ALPHA$(10)	"12345"	128+1/3/228,0/5, 0/10, 0	128 makes this a string. +1 means that it has been dimensioned. It starts 228+0*256 bytes into the array area. The current length of the string is 5+0*256. The maximum size of the string is 10+0*256.
E	.05	0/ 4/63/5,0, 0,0, 0	This is a number. The exponent is now 63 so the number is only 1/100 of its integer value, or .05.
FIG	-5	0/ 5/64+128/5, 0,0, 0	This is a minus number (+128 on exponent).

than is really necessary to solve this problem. If you are interested in this kind of information, study it. If not, skip it. If you have any questions, I would be glad to answer them. One warning. Do not press break while the program is running and then type RUN. Always use the CONT command after pressing BREAK. Otherwise the statements in lines 10-40 will destroy the program data. This can be prevented if you know what the correct value of A should be in line 10. Replace line 10 with 10A=n, where n is this number. Do this for your finished product.

```
10 A=PEEK(140)+PEEK(141)*256-82
20 POKE 141,INT(A/256):POKE 140,A-PEEK(141)*256
30 POKE 143,INT(A/256):POKE 142,A-PEEK(143)*256
40 POKE 145,INT(A/256):POKE 144,A-PEEK(145)*256
50 DIM A$(10),B(2,3)
70 ? A$
80 FOR I=0 TO 2
90 FOR J=0 TO 3
100 PRINT B(I,J),
110 NEXT J
120 PRINT
130 NEXT I
140 ? "STRING=";:INPUT A$
150 ? "I=";:INPUT I
160 IF I=9 THEN 200
170 ? "J=";:INPUT A:B(I,J)=A
190 GOTO 150
200 A=PEEK(140)+PEEK(141)*256
210 A=A+82
220 POKE 141,INT(A/256):POKE 140,A-PEEK(141)*256
230 CSAVE
```

Changing Atari programs to save the dimensioned variables:

- Get the program working.
- Place the string variables at the beginning of the variable table.
- Change the program so that it internally points the program-end address past the dimensioned variables and then saves itself to tape.
- Run the program, creating a copy on tape.
- Turn the computer off, on, and then reload the program.
- Add the statements to the beginning of the program to correct the program-end pointer, stack pointer, and end-of-memory pointer.
- Add the code to restore the actual string variable lengths to the variable value table.
- Save your finished program to tape.

The ATARI BASIC
Symbol Table

Most BASIC interpreters assign values to the symbol table as the program is run. Not true with the Atari. New variables are placed in the symbol table when the program line they are contained in is first typed.

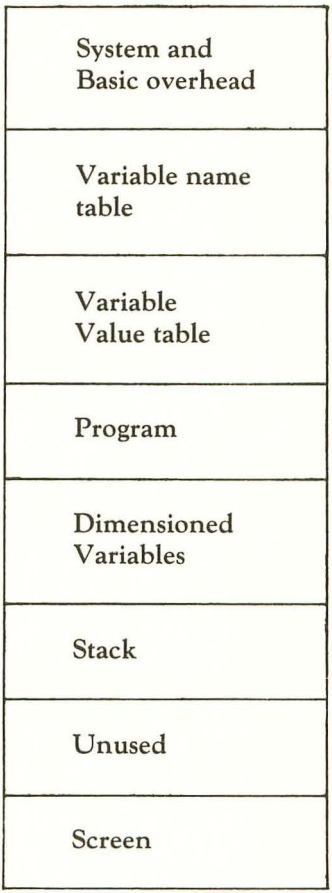
If you later change variable names, the old variable names are not removed from the table. They stay forever! Even the CLR command does not remove them.The continue to take up room. How much room? Eight bytes plus the length of the name. Add another byte if the variable is an array.

Fortunately, it is possible to clean up the variable table. Write the program to cassette using the command LIST "C", type NEW, and then reload the program from tape with the command ENTER "C".

A program can often be made to run faster by placing selected variables at the beginning of the variable table. This decreases the time it takes to find variables which are used in time-critical routines.

To place these variables at the beginning of the variable table, write the program to cassette using the command LIST "C" and then type NEW. Now use those variables. For example, if the variable A must be the first variable in the table, type A=0. If the string B$ must be used, type DIM B$(1). You are "ordering" the variable table. When you have finished placing as many variables in their correct order as you want, load the program you saved to tape with the command ENTER "C". This does not interfere with the contents of the variable table.

Figure 1:
BASIC Program Memory Layout



An Atari BASIC Tutorial: Monthly Bar Graph Program

Jerry White

Atari sound and graphics are great for game programs. In this monthly graph program, you will see how they can also be used to display data.

Data is often processed and compared on a monthly basis. Reports are generated to monitor things like cash flow or production. Sometimes it is much more meaningful to see totals in bar graph form rather than trying to compare a list of numbers. Using this program, the user types in the monthly totals and the program converts these figures into a beautiful graphic display.

For those who like to know how programs work: I'll break this one down and explain what each section is doing. For those who don't care: just key in the program and input your totals next to the appropriate month. The program will do the rest.

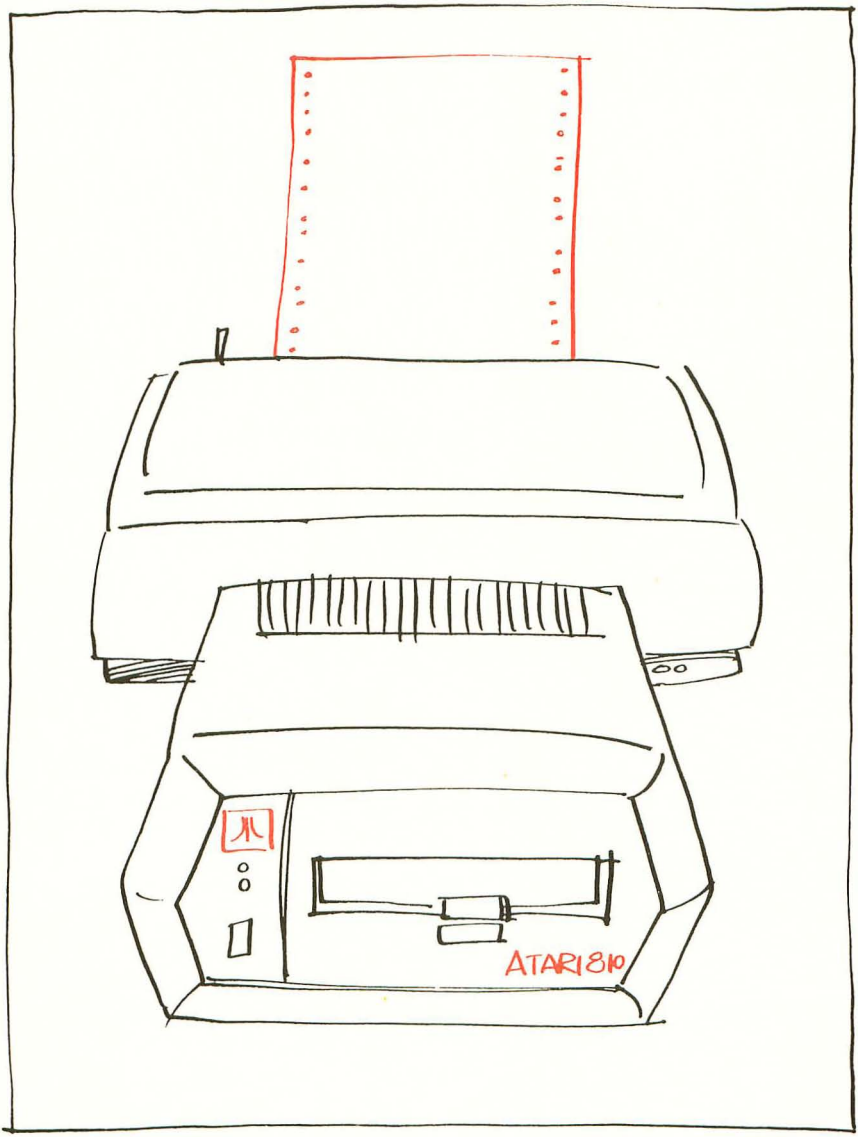
We begin by dimensioning A$ for use as a work string and two numeric arrays to hold 12 items. We go to the subroutine at 2000 and get our monthly totals and return to line 4. Here we get into graphics mode 6 with the text window at the bottom. We position our graphics window X and Y coordinates using PX and PY and put our heading into A$. Now we're off to the subroutine at 20. We will use this routine to convert our scratch string so that we can put text in the graphics window. Returning to line 8 — we use color 1 and draw a large rectangle. This is where we will draw our data bars. At line 100 we determine the highest amount (HAMT) so that we can base our key on that figure. The key will give meaning to the lengths of the bars. We set J1 = HAMT divided by 65 which is the length of the longest bar that fits into our rectangle. At line 130 we determine the top position of each bar. Then we make the top key figure (K) into a one or two position number and compute the numbers that will appear along the left side of the graph. At line 240 we begin to position and place our key of the screen. Then we set the screen margins as wide as possible and put the abbreviations for each month in the text window directly below the bar it represents. At line 310 we begin to draw our bars.

Being quite fond of sound, I couldn't resist adding line 360 as a finishing touch. This loop creates a tone as each bar is completed. Our purpose was to display data. Why not let the user use his ears as well as his eyes? Before we exit — we set the screen margins back to normal and loop at line 500. You could replace 500 with an end or exit routine. If you remove the first "? :" from line 300 there will be one line left in the text window for a message.

```
3 REM MONGRAPH REV.2 JERRY WHITE
1 REM FOR COMPUTE TUTORIAL
2 DIM A$(20),AMT(12),JW(12):GOSUB 2000
4 GRAPHICS 6:SETCOLOR 2,4,4:SETCOLOR 4,4,4:Z=1:SETCOLOR 0,1,10
6 PX=4:PY=0:A$="MONTHLY GRAPH":GOSUB 20
8 COLOR Z:PLOT 18,9:DRAWTO 158,9:DRAWTO 158,75:DRAWTO 18,75:
    DRAWTO 18,9
10 GOTO 100
20 DL=PEEK(560)+PEEK(561)*256:D1=PEEK(DL+4)+PEEK(DL+5)*256
22 FOR U=Z TO LEN(A$):D2=57344+((ASC(A$(U,U))-32)*8):
   D3=D1+PY*20+PX+U-Z:FOR JZ=0 TO 7
24 POKE D3+JZ*20,PEEK(D2+JZ):NEXT JZ:NEXT U:RETURN
100 FOR MON=Z TO 12:IF AMT(MON)>HAMT THEN HAMT=AMT(MON)
110 NEXT MON
120 J1=HAMT/65
130 FOR MON=Z TO 12:TAMT=75-(AMT(MON)/J1):JW(MON)=INT(TAMT):
    NEXT MON
140 IF HAMT>=10000 THEN K=INT(HAMT/1000):GOTO 200
150 IF HAMT>=1000 THEN K=INT(HAMT/100):GOTO 200
160 IF HAMT>=100 THEN K=INT(HAMT/10):GOTO 200
170 K=INT(HAMT)
200 KD=K/5:K2=INT(K-KD):K3=INT(K-(KD*2))
220 K4=INT(K-(KD*3)):K5=INT(K-(KD*4))
222 A$=STR$(K):PX=2-LEN(A$):PY=10:GOSUB 20
224 IF K<5 OR K>99 THEN 280
240 A$=STR$(K2):PX=2-LEN(A$):PY=24:GOSUB 20
250 A$=STR$(K3):PX=2-LEN(A$):PY=38:GOSUB 20
260 A$=STR$(K4):PX=2-LEN(A$):PY=52:GOSUB 20
270 A$=STR$(K5):PX=2-LEN(A$):PY=66:GOSUB 20
280 POKE 82,0:POKE 83,40:POKE 752,Z
300 ? :? "  K  J  F  M  A  M  J  J  A  S  O  N  D"
302 ? "  E  A  E  A  P  A  U  U  U  E  C  O  E"
304 ? "  Y  N  B  R  R  Y  N  L  G  P  T  V  C"
310 FOR MON=Z TO 12:JY=MON-Z
312 PLOT 18+(JY*12),JW(MON)
314 DRAWTO 25+(JY*12),JW(MON)
320 DRAWTO 25+(JY*12),75
330 DRAWTO 18+(JY*12),75
340 POSITION 18+(JY*12),JW(MON)
350 POKE 765,3:XIO 18,#6,0,0,"S:"
360 FOR VOL=10 TO 0 STEP -1:SOUND 0,JW(MON),10,VOL:NEXT VOL:
    NEXT MON
400 POKE 82,2:POKE 83,39
500 GOTO 500
2000 GRAPHICS 0:SETCOLOR 2,0,0:SETCOLOR 1,0,10:SETCOLOR 4,0,0:
    POKE 752,1
2080 ? :? ," MONTHLY GRAPH "
2100 ? :? "   TYPE AMOUNTS FOR EACH MONTH: ":?
```

```
2120 ? "   DO NOT USE NEGATIVE AMOUNTS  ":?
2200 TRAP 2200:? ,"JAN=";:INPUT JAN:AMT(1)=JAN:TRAP 40000
2210 TRAP 2210:? ,"FEB=";:INPUT FEB:AMT(2)=FEB:TRAP 40000
2220 TRAP 2220:? ,"MAR=";:INPUT MAR:AMT(3)=MAR:TRAP 40000
2230 TRAP 2230:? ,"APR=";:INPUT APR:AMT(4)=APR:TRAP 40000
2240 TRAP 2240:? ,"MAY=";:INPUT MAY:AMT(5)=MAY:TRAP 40000
2250 TRAP 2250:? ,"JUN=";:INPUT JUN:AMT(6)=JUN:TRAP 40000
2260 TRAP 2260:? ,"JUL=";:INPUT JUL:AMT(7)=JUL:TRAP 40000
2270 TRAP 2270:? ,"AUG=";:INPUT AUG:AMT(8)=AUG:TRAP 40000
2280 TRAP 2280:? ,"SEP=";:INPUT SEP:AMT(9)=SEP:TRAP 40000
2290 TRAP 2290:? ,"OCT=";:INPUT OCT:AMT(10)=OCT:TRAP 40000
2300 TRAP 2300:? ,"NOV=";:INPUT NOV:AMT(11)=NOV:TRAP 40000
2310 TRAP 2310:? ,"DEC=";:INPUT DEC:AMT(12)=DEC:TRAP 40000
2400 RETURN
```

CHAPTER SIX: Peripheral Information



Adding a Voice Track to Atari Programs

John Victor

This technique shows you how to let your Atari talk — that is, play audio cassettes totally under the control of your program.

We recently had a chance to see the latest in audio-visual technology — a video tape machine being controlled by an Apple computer. The student was shown selected film sequences on the video tape. Then the video tape would stop and the student would be asked questions by the computer.

This demonstration had some impressive features, but the most important was the integration of voice with the computer question and answer technique. The same effect can be generated on an Atari 400 by combining text, graphics, animation, and color with a sound track recorded on an audio cassette. And the Atari 400 is significantly cheaper and easier to program than the combination video tape player/computer.

There are several ways that a software designer/programmer can sync a cassette voice track to visuals on the computer screen. The cassette player that plugs into the Atari computer records and plays in a stereo format. The right track on the tape records and plays digital information (such as programs or data files), while the left track plays audio recordings. The "Talk and Teach" ROM and tapes supplied with Atari computers use both tracks simultaneously. As the voice explains material, ASCII characters are read off the digital track and shown on the screen. The two are coordinated in the manufacturing process so that they are always synchronized.

The problem with the "Talk and Teach" system for the average Atari owner is that the development of the Talk and Teach cassettes requires different hardware than is supplied with the Atari system. In fact, the system may be developed and run on non-Atari equipment — we have seen the cassettes run on a modified TRS-80 computer.

The simplest and most practical method for Atari users to sync voice with their own educational programs is to use a "timed-BASIC" method. The visuals are programmed into a BASIC program and run simultaneously with an audio tape cassette. The

program would then start and stop the Atari cassette player and change the visuals on the TV screen based on timing routines built into the program. The key to making this system work is that the audio tape must start at the same point each time it is used.

The computer course designer-programmer first writes a script as though he or she were producing a sound/filmstrip presentation. The spoken words, music, etc. would be specified along with a detailed description of what is to appear on the TV screen. The designer-programmer then writes a BASIC program that will produce the desired visual effects.

The next step is to coordinate the voice with the visuals. The best way to do this is to have a preliminary routine within the computer "freeze" each screen display until the programmer hits the 'RETURN' key. This can be done by sending the program to an INPUT subroutine, but this has the undesired side effect of printing an extraneous question mark on the screen. We prefer using the subroutine shown below since it prints nothing at all on the screen:

```
5000 IF PEEK(764)<>255 THEN POKE
764,255:RETURN
5010 GOTO 5000
```

Memory location 764 indicates whether a key has been pressed. If no key has been pressed, the number 255 will be stored there. When a key has been pressed, the routine sets the value back to 255 (to keep the computer from printing the key press) and the program returns from the subroutine.

The designer-programmer should read through the script and manually check the screen changes to see that the BASIC program and the script match up and produce the desired results.

The third step is to place timing routines into the program so that the visuals will be in sync with the recorded voice. We do **NOT** recommend using FOR . . . NEXT timing loops for these routines. FOR . . . NEXT loop timing is not linear on the Atari. This means that the Atari might take one second to count from 1 to 300 in one loop, and less time to do the same count in another loop of different length. In addition, the length of the program and position of the subroutine also affects the count.

Fortunately, the programmer can utilize a built-in clock used by the Atari computer to count the scan lines in the TV display, which is stored in memory locations 18, 19 and 20. Location 20 counts in "jiffies" or 1/60 second. Each 1/60 increases the value stored in location 20 by 1. When the count reaches 256, the value is cleared to 0 and location 19 is incremented by 1. It takes the

computer about 4.27 seconds to count from 1 to 256 in location 20, and about 18.2 minutes to count from 1 to 256 in location 19. You can watch this process with the following program:

```
10 PRINT PEEK(20), PEEK(19), PEEK(18):
GOTO 10
```

The results from these PEEKs could be converted to seconds, but we prefer to work in jiffies, which requires less math on the computer's part.

```
SECONDS = (PEEK(19)*256 + PEEK(20))/60
JIFFIES = PEEK(19)*256+ PEEK(20)
```

We recommend that at this point the designer-programmer makes the final audio cassette that is to go with the computer program. The program can be timed to this cassette, and, if all copies of the cassette can be made to use the same starting point, then the program will work with all copies as well.

The task now is to figure out the timing for each change so that the changes will be made in sync with the audio cassette. Figure 1 shows a program that we developed to automatically make these measurements for an audio tape. The user puts the audio tape in the Atari cassette player and rewinds it to the very beginning. With the play button depressed, the user runs the program. Line 20 starts the cassette player, and the program begins timing. At each point where the user wished the computer to change the visual (in conjunction with the voice), the user hits the 'RETURN' key. At the end of the program the cassette is shut off, and the user is given the times between each point on the voice track where the computer is to change the visual.

The user should note that memory locations 19 and 20 are set back to 0 after each timing, and that line 55 looks specifically for an input from the 'RETURN' key. This program counts up to 15 changes, but this number can be increased by increasing COUNT in lines 40 and 100.

The last step is to insert the time values into the computer program and to check to see that the voice cassette works in sync with the program. Figure 2 shows a program that we wrote to illustrate how timing values can be coordinated with a teaching program and audio tape. Line 50 of the program defines the subroutines, of which there are three: one to print questions on the screen, one to time the visual so that the voice on the tape can read the question, and one to shut off the tape so that the student can answer the question just asked.

The QUESTIONASK subroutine in lines 4000-4030 gets its

information for each question from a DATA line, which includes question number, screen color, answer to the question, number of lines to be read, and the lines of text making up the question. After printing the question, the program sets the time value for the voice, and goes to the clock subroutine at 5000-5020. When the correct time elapses, the program goes to the QUESTIONANSWER routine. Here the tape is shut off, and the user is required to answer the question. Upon answering, the tape is turned back on.

The time values in this program are based on our own personal reading of the questions.

While it is possible to record both programs and audio on the same cassette and still utilize the method we have described here, the best way is to record programs and audio separately. Ideally, the programs would be stored on disk and the voice on cassette.

It is possible that with very long audio cassettes the computer and tape will get out of sync due to small variations of the cassette player. The designer-programmer can correct for this by occasionally having the student press 'RETURN' when he or she hears a beep on the audio track, This gives a frame of reference for the program timing to match up to the tape. The least obvious way of doing this is to have the student press 'RETURN' before answering a question.

Figure 1

```
5 REM TIMING PROGRAM BY JOHN VICTOR
6 REM FOR ATARI COMPUTER VOICE TRACK
10 DIM TIME(15),A$(1)
20 POKE 54018,52:REM TURN ON CASSETTE
30 GRAPHICS 0:POSITION 2,6
35 PRINT "START COUNTING..."
40 FOR COUNT=1 TO 15:SETCOLOR 2,INT(RND(
1)*15),4
50 POKE 19,0:POKE 20,0
55 IF PEEK(764)<>12 THEN 55
60 JIFFY=256*PEEK(19)+PEEK(20):TIME(COUN
T)=JIFFY:PRINT "CHANGE #";COUNT
73 POKE 764,255
75 NEXT COUNT
78 POKE 54018,60:REM SHUT OFF CASSETTE
80 PRINT :PRINT "PRESS RETURN TO SEE TIM
E VALUES IN JIFFIES"
90 INPUT A$
```

```
100 FOR COUNT=1 TO 15:PRINT "CHANGE #";C
OUNT;"=";TIME(COUNT):NEXT COUNT
200 END
```

Figure 2

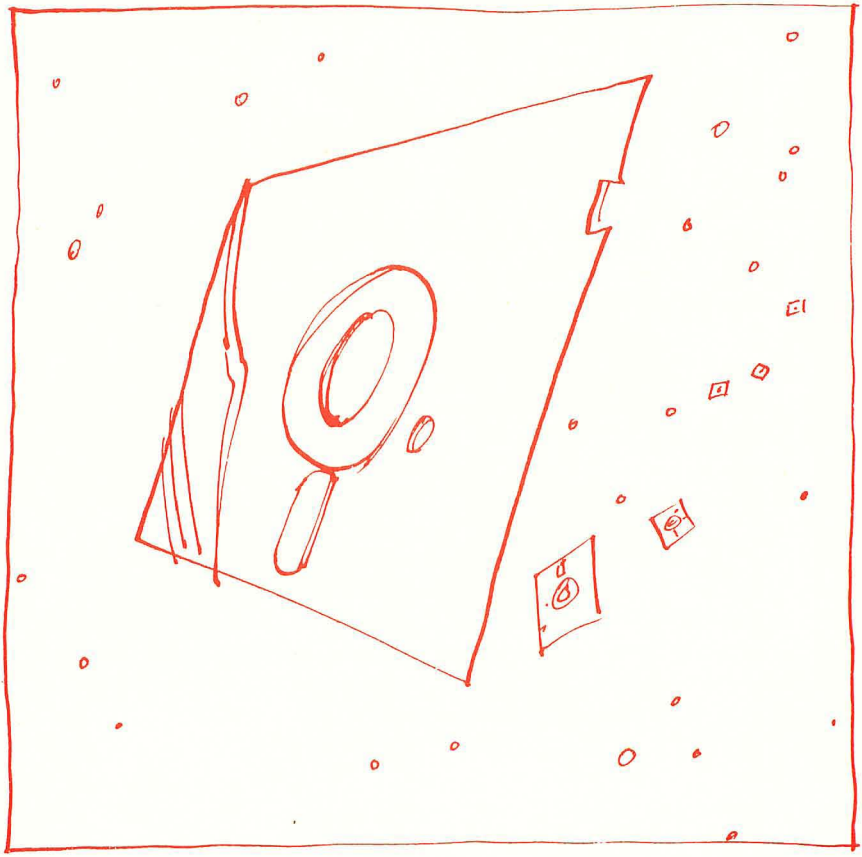
```
10 REM DEMONSTRATION OF ATARI TIMING
20 REM FOR TUTORIALS USING VOICE AND
30 REM TIMING LOOPS
40 REM PROGRAM DESIGN, INC.
50 CLOCK=5000:QUESTIONASK=4000:QUESTIONA
NSWER=3000:REM SUBROUTINE LABELS AND LOC
ATIONS
60 DIM ANSWER$(10),RESPONSE$(10),LINE$(4
0)
100 GRAPHICS 2+16:POSITION 0,2:PRINT #6;
"   BASIC TUTORIAL":PRINT #6;"   DEMONST
RATION":PRINT #6
105 PRINT #6;"    with voice"
110 TIME=300:GOSUB CLOCK
200 GRAPHICS 0:PRINT :PRINT
205 PRINT "This is a demonstration of th
e ATARI":PRINT "computer's ability to ut
ilize a"
206 PRINT "sound-voice track.  I will as
k four":PRINT "sample questions about AT
ARI BASIC.":PRINT
207 PRINT "Place audio cassette in playe
r and":PRINT "rewind to beginning.":PRIN
T
210 PRINT "Before starting this demonstr
ation,"
215 PRINT "make sure that the PLAY botto
n is"
220 PRINT "pressed down on your cassette
 player."
230 PRINT :PRINT :PRINT "PRESS RETURN TO
 START.":INPUT RESPONSE$
250 POKE 54018,52:REM STARTS TAPE
300 GOSUB QUESTIONASK:TIME=1274:GOSUB CL
OCK:GOSUB QUESTIONANSWER
310 GOSUB QUESTIONASK:TIME=681:GOSUB CLO
```

```
CK:GOSUB QUESTIONANSWER
320 GOSUB QUESTIONASK:TIME=683:GOSUB CLO
CK:GOSUB QUESTIONANSWER
330 GOSUB QUESTIONASK:TIME=800:GOSUB CLO
CK:GOSUB QUESTIONANSWER
340 GOSUB QUESTIONASK:TIME=653:GOSUB CLO
CK:GOSUB QUESTIONANSWER
400 GRAPHICS 1:SETCOLOR 2,0,14:SETCOLOR
4,0,14:POSITION 0,8:PRINT #6;"    END OF
 DEMO ":TIME=392:GOSUB CLOCK
410 POKE 54018,60:REM SHUT OFF CASSETTE
500 GRAPHICS 0:END

2999 REM ANSWERING SUBROUTINE
3000 POKE 54018,60:REM SHUTS OFF CASSETT
E
3010 PRINT :PRINT "YOUR ANSWER IS ";:INP
UT RESPONSE$
3020 IF RESPONSE$=ANSWER$ THEN PRINT CHR
$(253):PRINT :PRINT "CORRECT!":GOTO 3100

3040 PRINT :PRINT "NO, THE ANSWER IS ";A
NSWER$
3100 PRINT :PRINT "PRESS RETURN TO CONTI
NUE...":INPUT RESPONSE$
3110 POKE 54018,52:REM TURN CASSETTE BAC
K ON
3120 RETURN
3999 REM QUESTION SUBROUTINE
4000 GRAPHICS 0:READ NUMBER,COLOR,LINES,
ANSWER$
4010 SETCOLOR 2,COLOR,4:PRINT :PRINT :PR
INT "QUESTION #";NUMBER:PRINT :PRINT
4020 FOR COUNT=1 TO LINES:READ LINE$:PRI
NT LINE$:NEXT COUNT
4030 RETURN
4999 REM TIMING LOOP
5000 POKE 19,0:POKE 20,0:REM SETS CLOCK
TO 0
5010 IF PEEK(19)*256+PEEK(20)<TIME THEN
5010
```

```
5020 RETURN
6000 DATA 1,5,3,CLOAD,What is the usual
BASIC command to,tell the computer to lo
ad a program,from cassette tape?
6010 DATA 2,10,2,LIST,What command will
show you the pro-,gram stored in the com
puter memory?
6020 DATA 3,1,2,RUN,What command execute
s a program in,the computer's memory?
6030 DATA 4,3,3,CSAVE,What is the most c
ommonly-used ATARI,BASIC command used to
 record programs,to cassette tape?
6040 DATA 5,14,2,NEW,What command wipes
out the program,in memory?
```



The Atari Disk Operating System

Roger Beseke

A thorough examination of the disk operating modes.

Now that you have your ATARI 810 disk system up and running and have undoubtedly saved and entered numerous programs and data, you are probably wondering what else this machine can do. Well, to date Atari has not released their DOS system manual, but there is a preliminary manual which is available and contains a wealth of information. The purpose of this article is to bring into the light some of the features hidden away in the preliminary manual.

As we all know, after having the disk up and running, there is a disk system menu which is displayed upon entry of the command "DOS, RETURN". Some of these commands are straightforward and require little or no explanation, but we are going to take a look at all of them.

There are two neat characters we must discuss before we go into the DOS menu of commands. They are the asterisk (*) and the question mark (?). When these characters are used in a DOS command, they are referred to as wild carding. They allow excellent flexibility which can be used to great benefit or dismay depending what the operator is using the wild card character for. It probably goes without saying that these characters should not be used in a file label.

In the ATARI DOS, the (*) is used to free form a file name for most of the commands. The asterisk can follow a portion of a label in either the main file label or the extension. Note: The file name does not have to be eight characters to use the extension. The asterisk can be used in numerous ways to provide as many results. I will cover a few here and leave the rest to your imagination. By the way, all the commands in this article are in quotes. If the command requires quotes, there will be double quotation marks. Also when return is spelled out in caps, it means the "RETURN" key is to be pressed.

A command of the form "*.*" will display all files on the screen if used with the disk directory command (A). A command of

the form 'PROG*.*" would list all programs that met the first four character format. Similarly the command "*.U*" would only list files that had an extension in either the main or extension field, all characters following it are ignored.

The (?) in the ATARI DOS is used to set a character to a do not care condition when wild carding is used. The following example, "WORD?S.*", shows that all files having the form "WORD" and any other character in the don't care character field will be operated on. These wild card characters can be used anyplace in a legal label field.

Disk directory (A): The disk directory takes care of finding and listing the files of a diskette. The files may be listed on the screen or on your ATARI 820 printer. It is common knowlege that, to get a display of the files on a particular disk, you must issue the command "A RETURN RETURN" and they are displayed on the monitor. This is fine if you do not have many files or if you want to see all the files there are on the disk. If you do not want to see them all, there are commands that can be sent to select a certain group of files. They also can be printed on the printer. To get hard copy, issue a command of ",P:" before the second "RETURN". A command of the form "RA*.B?,P:RETURN" will list all files with the first two characters "RA" in the main field and characters in the extension which begin with a "B" followed by one character.

Run cartridge (B). This command exits the DOS and executes in the left cartridge if one is inserted. It will not exit the DOS if a cartridge is not inserted in the left slot.

Copy (C). The copy command enables the operator to copy a file from one device to the disk or copy a file from one disk to another file on another disk. For instance, a command of the form "D1:FILE,D2:PROG" will copy a file named "FILE" from disk one to a file named "PROG" on disk two. You can write a file from the screen editor to a disk file by a command similar to one of the form "E:,NAME". This command must be terminated with a "CTRL 3" key entry.

Delete (D). The delete command does allow wild card commands and can take the form of any of the previous examples. The DOS displays a cue to the operator to delete the file shown. The operator makes the appropriate entry and the DOS brings up another file if there are wild cards used and files that meet the wild card form. A typical deletion of all files with an extension of B1 thru B34 could be deleted one at a time with the command "*.B??RETURN". If "/N" is appended to the command, it will

delete the appropriate files without a cue, so be careful. It must be remembered that locked files cannot be deleted.

Rename (E). The rename command allows you to change the name of a file to another, and wild cards are allowed. A typical command would be "FILE,KEEPFILE RETURN". This command will change the name of the present file "FILE" to "KEEPFILE." It must be noted that extreme care is recommended with this command when using wild cards because you can end up with a group of files with the same name.

Lock (F). The lock command as mentioned previously keeps you from inadvertently writing to or deleting those files. A locked file can be recognized readily in the disk directory mode because of the asterisk ahead of the file name. Wild cards are also allowed in this command. A typical command to lock all files would be "*.*RETURN".

Unlock (G). The unlock command is the reverse of the previous lock command and the same protocol is allowed. But again, a word of caution using wild cards: you may be unlocking something you do not want to.

Write "DOS" File (H). This command writes the DOS on a formatted disk so that it can be booted into the computer at turn on. This command allows you to make all your disks boot-loadable and gives you a backup for the DOS.

Format Disk (I). This command is required for all new disks before they can be written on. The DOS cues the operator as to which disk to format. Again a double check is made to make sure that is the disk the operator wants formatted because, if the wrong one is formatted, all files are lost on that disk.

Duplicate Disk (J). The duplicate disk command allows you to make a copy of your present disk on another even if you do not have two drives to copy with. A typical entry might be "1,2RETURN" where 1 is the source disk and 2 is the destination disk. If you do not have two drives, the DOS will issue commands on which disk to insert for writing or reading. Programs in memory are destroyed when using this command and the DOS reminds you of that fact when this command is entered.

Binary Save (K). Binary save is the command that one can use to save all those machine code programs you generate if you have an assembler. The binary save, unlike most of the other commands utilized by the ATARI, uses hex numbers as opposed to decimal. I suppose if you want to save those machine code programs, you can count to sixteen using letters anyway. A typical

save binary program appears like "D2:MACHINE.CDE,4FEO, 6BAC RETURN". This would write a file called "MACHINE.CDE" on disk 2. The data would be saved from addresses 4FEO to 6BAC inclusive. This command also allows the append syntax by placing it immediately following the file name. An example is as follows: "D:OPCODE/A,54E2,2BC3".

Now I am going to give you a clue as to how to automatically execute your program from a binary load command. Before you become too elated, there are some pains with all neat things, even in the world of ATARI. You have to poke addresses 736 and 737 with the starting address of your binary program. Address 736 is the low order byte of the starting address and 737 is the high order byte. For you machine code users, the addresses are 02E0 and 02E1. Now just append this to your program and away you go.

Binary Load (L). This is the command you use to load the previously saved binary program. There really is not too much to say about it, especially if you append the starting address of your program. You just enter the file name and let the system do the rest.

Run at Address (M). Run at address is for those of us who did not have the book of how to do it. The DOS asks you run from what address and you enter the address in hex, of course. After all, we are binary programmers, are we not?

Define Device (N). The preliminary manual does not recommend using this command as it is not perfected. Rumor has it that there will be a revision out soon to fix it, however. To me, that is a challenge to find out what about it works and if it is useful. The intent was to essentially change the name of a device and create pseudo files and names. One example is "P:FILE" where, whenever "P" is referenced, it will write to a file "FILE" of whatever target you directed it.

Duplicate File (O). Duplicate file is like the J command of duplicating the disk except you do not duplicate as much: to be exact, a file at a time. Again, if you only have one drive like some of us, you can do it the same way as the duplicate disk command.

This has been a very brief description of what you can do with the DOS and how it can work for you. I am sure that when the DOS operator's manual comes out, it will explain everything much better, but until then, maybe this will keep some of you file manipulators out there happy.

Review of the Atari 810 Disk System

Ron Jeffries and Glenn Fisher

The Atari 810 disk system is very easy to install: unpack it, read a couple of pages of the Operator's Manual, plug in two cords, turn it on, insert a diskette, and you are up and running. (If, and only if, you have 16K or more memory. Otherwise, the screen does strange things, including producing some fascinating patterns.)

The Disk Drive Operator's Manual shipped with the early units is actually only an 11 page looseleaf booklet. The information in the booklet is clear, with an excellent diagram that should make it possible for almost anyone to set up the disk system correctly. Maybe that seems minor, but things haven't always been this way, folks. On the other hand, 11 pages is not enough to say all the things that need to be said to a person that just bought their first disk. We didn't have any real problems, but then again the Atari isn't the first disk we've used. As of late January, the Disk Operating System (DOS) Reference Manual isn't yet available. Atari has done a great job getting a "total system" out, including disk and printer. But documentation seems to be much harder to get out the door than either hardware or software.

The disk drives are nicely packaged in injection-molded plastic cases. You can stack two disk drives, and even put the 820 printer on top and still have a stable arrangement that takes only a 10 inch by 14 inch area. There are small indentations on the top of each disk cabinet that provide a solid platform for the one stacked on top of it. Everyone who has seen our unit has commented on how attractive the packaging is, and how it looks like a consumer product. One fact of life with the Atari is that there are lots of cords to connect everything together, as well as to supply power. Since Atari uses separate UL-approved power adaptors for everything except the cassette recorder and the 820 printer, you soon find that there are a lot of power adaptors to put somewhere. On the other hand, having the transformers separate from the disks and the computer probably contributes to their compact look.

To load the DOS, the 810 disk is turned on and the Master Diskette (containing the DOS) is inserted. The Atari computer itself is then turned on, which automatically drags the DOS into

memory. After about ten seconds, the message "READY" appears on the screen. Now, when you type the command "DOS", a menu will appear:

```
DISK OPERATING SYSTEM 9/24/79
COPYRIGHT 1979 ATARI

A. DISK DIRECTORY
B. RUN CARTRIDGE
C. COPY FILE
D. DELETE FILE
E. RENAME FILE
F. LOCK FILE
G. UNLOCK FILE
H. WRITE DOS FILE
I. FORMAT DISK
J. DUPLICATE DISK
K. BINARY SAVE
L. BINARY LOAD
M. RUN AT ADDRESS
N. DEFINE DEVICE
O. DUPLICATE FILE

SELECT ITEM
```

"Run Cartridge" means "leave DOS." At least for now, the DOS can't be used unless you are using the BASIC cartridge. Later on there may be other languages. One that we hope to see soon is an assembler and editor for working with 6502 machine language.

A good feature of the Atari DOS is the ability to "lock" a file, so that it can't be deleted, renamed, or written into. This can be very handy if you have an important file that you want to protect. (As an aside, we've heard that the same people that wrote the Apple DOS worked on the Atari version. Guess what? Apple is the only other micro system we know of that has a "lock" capability.)

"Write DOS" is how you make new copies of the DOS. Unlike some systems, the Atari DOS is a normal file, instead of being hidden away in some secret location on the disk. Each diskette can hold 709 sectors of 128 bytes each. The DOS takes 64 of these sectors, leaving 645 sectors, or about 86K bytes, for your files.

Alas, all is not sweetness and light.

First, the DOS uses about 9K of your memory. So, on a 16K Atari, when you first turn on the system you'll have about 4300 bytes left of the 16K. (Here is the math: a "pristine" 16K Atari has 13326 bytes of memory available for your program. The rest is used by BASIC, the operating system, and as screen memory. The Atari DOS comes configured for four drives, and when it is loaded into

the computer you have 4328 bytes left. If you change a couple of parameters to tell the system you only have one drive you can free enough memory to have a total of 4622 bytes available.)

There is a short BASIC program that you can run which throws away most of the DOS, leaving only the ability to Load from and Save to the disk, as well as access the disk from BASIC programs. Howver, when you do this, you can't even look at the directory ofthe disk without running a special program, nor is it possible to save this small DOS so that you can "boot" from it, since the ability to write a DOS file went away when you threw out the menu. So, if you want to use "Tiny DOS," each time you boot the system you'll have to run the BASIC program. In this "stripped down" mode you have about 9.4K available.

What can we say? Well, although the menu seemed friendly and handy at first, when you consider what it costs in memory, it may not be worth it. A more important issue is which DOS functions are crucial, and which can be shunted off into a separate "disk utility." Given the tight memory situation, we'd vote for the following as essential DOS functions, with everything else exiled to Siberia: directory, delete file, and, of course, load and save files. These important DOS functions would ideally be direct commands, such as "DIR" or "CATALOG" for the directory.

Atari file names can only be UPPER CASE letters and digits. Why they chose such a restricted set is a mystery, since only comma, period, colon, asterisk and the question mark have special meaning to the DOS. File names consist of eight characters followed by a three-character "extension." Eight-character names are too short to be really meaningful. (Just because CP/M and DEC made that mistake doesn't mean it should be repeated. Commodore allows 16 character names, and they can contain almost any characters you like.) Speaking of UPPER CASE, the 800 itself has a "feature" we find frustrating: it doesn't understand lower-case BASIC keywords!

To summarize, we find many things about the system that we like, as well as some things that aren't what they could have been with a little better planning and design. Atari has put together a good system, one that we think will sell like gangbusters. It's available now, at obscure places like Sears and J.C. Penneys and the like, as well as your friendly local computer store.

An Atari Tutorial: Atari Disk Menu

Len Lindsay

This program permits greater efficiency when using disks.

Anyone with an ATARI disk will really appreciate this program. You will probably put a copy of MENU on each of your diskettes.

MENU will display the programs on the diskette along with an ID number (1-44). It then asks you which program you wish to RUN. If you wish to RUN program number 8, you simply answer 8. It then LOADS and RUNs that program. No more hassles trying to remember exactly what name you used for the program, or typing the name exactly. MENU does it all for you.

Since MENU uses some special techniques, I will explain how it works. You should be able to apply many of these concepts to your own programs.

LINE 10-11 — Dimension the STRINGs. ARRAY$ will hold all the names of the programs on the disk (12 characters per name). FILE$ and NAME$ are used for the program names. DISK$ is used to hold the drive number prefix.

LINE 15 — Set the margins to default, in case the previous program used differed ones.

LINE 20 — Use GRAPHICS 0 full screen text mode. It also clears the screen for you.

LINE 30 — Turn the cursor off — it looks nicer while writing the program names on the screen.

LINE 40 — Set the color registers to the preferred colors. A light orange background with warm brown letters is the easiest on your eyes.

LINE 50 — Set DISK$ to the disk drive to be used. See modification notes to make this more flexible.

LINE 60-70 — "D1:*.*" will refer to the disk directory. It is a two step process to add the DISK$ with "*.*" and call it NAME$.

LINE 100 — Open the disk directory for a READ. This line should be useful for other applications.

LINE 110 — Initialize the counter which counts each program as it

is read from the directory. This also acts as the program ID number.

LINE 120 — READ one file from the directory. A program entry is 17 characters long. It is two spaces, 8 characters for name, 3 characters for extension, one space, 3 characters for sectors used. After all the programs, there is a separate record of the number of free sectors left on the diskette.

LINE 130 — Check if this is the short record of tracks left on diskette. If it is, then we are done and should go on to the next part starting at line 500.

LINE 140 — Since we read in another program name, add one to the counter.

LINE 150 — If this is the 23rd program, we must switch to the right half of the screen (prevent scrolling and fit more on the screen this way). To do this we set the margin to 20 and position the cursor at the top line, 20th spot.

LINE 160 — Check if the screen is completely filled with program names (44 is the maximum display allowed on one screen). If it is full, ignore all the rest, adjust the counter accordingly. See modification notes for other ideas.

LINE 200 — Initialize the name field. To manipulate the string by character position, the positions all must exist. Initializing to " " (null) will not work. Note the extention dot is in position 9.

LINE 210 — If there is no extension, get rid of the dot in position 9.

LINE 220 — Assign the program name from FILE$ which we just READ from the diskette. This is only the first 8 characters of the name, not including the extension.

LINE 230 — Assign the extension of the program name. If there is no extension, we still can assign it since the dot has already been removed.

LINE 300 — To keep a justified column of ID numbers, we must allow for *one* digit numbers. So if the number is less than 10, print an extra space.

LINE 310 — Print the ID number followed by) and a space.

LINE 320 — Print the program name.

LINE 400 — Add the name onto the ARRAY$ we are building. It can now be referenced by number times 12 (since every name is exactly 12 characters long).

LINE 410 — Processing complete for the program just read. Go and do the next one.

LINE 500 — Set the trap to come back and redo the input if an error occurs.

LINE 505 — CLOSE the file used to input the programs from the directory.

LINE 510 — Turn the cursor back on for the INPUT request.

LINE 520 — Position the cursor on the message line (line 22). First print a line-delete to erase the previous message. Then print the current message. End the message with a BEEP (control 2).

LINE 525 — Set the left margin back to default so the next program will not be affected.

LINE 530 — INPUT the ID number of the program to be RUN.

LINE 540 — Get rid of any fractions.

LINE 550 — If the choice was not in the range available, go and ask again.

LINE 600 — Start FILE$ with the disk number. The rest of the name is assigned in line 630.

LINE 610 — Assign the name of the program chosen to NAME$ (taken from the ARRAY$ we just put together).

LINE 620 — Start a loop to go through the whole 12 character program name and remove all spaces (spaces cannot be imbedded within a program name when you ask for a LOAD or RUN).

LINE 630 — Add the characters in the program name one at a time to FILE$. Ignore spaces.

LINE 640 — Do the next character.

LINE 700 — Set the trap to line 900 to print a can't load message if there is a disk error.

LINE 720 — Position the cursor to the message line. First do a line delete to erase the previous message. Then print the message LOADING with the file name. Then print a BEEP (control 2).

LINE 730 — RUN the program and spring the trap.

LINE 900 — Print message the program can't be run (maybe diskette was switched or removed since the directory was read).

LINE 910 — Pause to allow message to be read.

LINE 920 — Go and ask for program to RUN again.

Possible Modifications

MENU is set up to work with disk drive 1. It is easy to have it work with both drive 1 and drive 2, and even alternate between them for

a wider MENU choice. Line 50 sets the disk drive number prefix to be used by the MENU program. Some possible modifications follow; the first asks you which drive to use for the MENU, while the second can flip back and forth from drive to drive. I have implemented the second set of modifications and find it works quite well. Either way, it seems that it doesn't like trying to give you a MENU for an empty drive.

Modification Set 1 — Ask Which Drive

```
50 PRINT "[CLEAR] WHAT DISK DRIVE TO USE)";
51 OPEN #1, 4, 0, "K:" :REM OPEN KEYBOARD FOR
GET
52 TRAP 52: GET #1, DRIVE : REM GET KEY PRESSED ATASCII VALUE
53 NAME$ = CHR$(DRIVE) : REM CONVERT TO STRING — USE
NAME$ SINCE IT IS DIMed
54 IF NAME$<"1" OR NAME$> "4" THEN 52 : REM TRY AGAIN
55 PRINT NAME$ : REM PRINT THE REPLY
56 CLOSE #1 : REM CLOSE THE FILE
57 DISKS$ = "D1:" : REM INITIALIZE STRING
58 DISK$(2,2) = NAME$ : REM INSERT DRIVE NUMBER
```

Modifications For Alternating Drives

```
17 DRIVE = 2 : REM INITIALIZE FOR A TWO DRIVE SYSTEM —
DRIVE 1 WILL BE FIRST
18 DISK$ = "D1:" : REM INITIALIZE
50 DRIVE = 3-DRIVE : REM SWITCH DRIVES, WILL DO DRIVE 1 FIRST
55 DISK$ (2,2) = STR$ (DRIVE) : REM PUT CORRECT DRIVE NUMBER
INTO DISK$
59 TRAP 50 : REM TRAP DISK ERROR
105 ARRAY$ = "  " : REM INITIALIZE
115 PRINT "[CLEAR]"; : REM CLEAR SCREEN
520 POKE 82, 2 : REM LEFT MARGIN TO DEFAULT
525 POSITION 2,22 : PRINT "[DELETE LINE]0=NEXT DRIVE WHICH
TO RUN[CONTROL 2]";
535 IF CHOICE = 0 THEN 50 : REM SWITCH DRIVES ON CHOICE OF 0
```

Another modification you may wish to make has to do with the ability to jump into DOS immediately directly from MENU. If you try to RUN it as your MENU choice, it will say "can't run dos." Thus, if you think you may need to jump directly to DOS add this line:

```
615 IF NAME$ = "DOS .SYS" THEN DOS
```

Modifications To Overcome The 44 Program Limit

The MENU can only display 44 program choices at one time, thus line 160 checks if the screen is full(44). If it is, it skips all the rest of the programs. In practice this should not be a problem since most diskettes will be filled before they reach the 45th program unless the programs are all short.

Modifications might be made so that after 44 programs, they no longer are printed on the screen, but still are added to ARRAY$ with FILECOUNT continuing its count. The DIM in line 10 for ARRAY$ should be increased accordingly. The message line (520-525) should also be appropriately changed. Perhaps a choice of 99 would mean "display second screen of menu." A subroutine could calculate what program number to start with (filecount minus 43) and another subroutine could print the menu from ARRAY$ as appropriate.

```
0 REM *** MENU (44 PROGRAM MAX)
1 REM *** (C) 1980 LEN LINDSAY
2 REM *** LAST REVISION 11-15-80
3 REM
10 DIM ARRAY$(528),FILE$(20),NAME$(20)
11 DIM DISK$(3)
15 POKE 82,2:POKE 83,39:REM DEFAULT MARG
INS
20 GRAPHICS 0:REM CLEAR SCREEN AND GO IN
TO TEXT MODE 0
30 POKE 752,1:REM CURSOR OFF
40 SETCOLOR 2,2,6:SETCOLOR 4,2,6:SETCOLO
R 1,2,2
50 DISK$="D1:":REM THE DISK TO BE USED F
OR A MENU
60 NAME$=DISK$:REM THE NAME MUST START W
ITH THE DISK DRIVE NUMBER
70 NAME$(LEN(NAME$)+1)="*.*":REM LOADING
 D1:*.* GIVES THE DISKS DIRECTORY
100 OPEN #1,6,0,NAME$:REM OPEN THE DISK
DIRECTORY FOR A READ
110 FILECOUNT=0:REM INITIALIZE COUNT
120 INPUT #1,FILE$:REM READ NEXT PROGRAM
 NAME
130 IF LEN(FILE$)<5 THEN 500:REM NOT A P
ROGRAM - THIS IS THE SECTORS LEFT COUNT
140 FILECOUNT=FILECOUNT+1:REM ADD ONE TO
 COUNT
150 IF FILECOUNT=23 THEN POKE 82,20:POSI
TION 20,0:REM SWITCH TO RIGHT HALF OF SC
REEN (CHANGE LEFT MARGIN TOO)
160 IF FILECOUNT>44 THEN FILECOUNT=44:GO
```

```
TO 120:REM TOO MANY PROGRAMS - JUST KEEP
 READING
200 NAME$="              ":REM INITIALIZE
THE NAME FIELD TO ALL BLANKS EXCEPT THE
DOT BEFORE THE EXTENSION
210 IF FILE$(11,13)="   " THEN NAME$(9,9
)=" ":REM THERE IS NO EXTENSION SO GET R
ID OF THE DOT
220 NAME$(1,8)=FILE$(3,10):REM ASSIGN TH
E FIRST 8 CHARACTERS OF THE PROGRAM NAME

230 NAME$(10,12)=FILE$(11,13):REM ASSIGN
 THE EXTENSION OF THE PROGRAM NAME
300 IF FILECOUNT<10 THEN PRINT " ";:REM
ADD AN EXTRA SPACE BEFORE THE SINGLE DIG
IT NUMBERS TO ALIGN WITH 2 DIGITS
310 PRINT FILECOUNT;") ";:REM PRINT FILE
 NUMBER
320 PRINT NAME$:REM PRINT THE PROGRAM NA
ME
400 ARRAY$(LEN(ARRAY$)+1)=NAME$:REM ADD
ON THE LATEST NAME TO END OF STRING OF N
AMES THUS FAR
410 GOTO 120:REM GO READ NEXT FILE NAME
500 TRAP 500:REM SET TRAP FOR BAD INPUT
505 CLOSE #1:REM CLOSE THE FILE USED TO
INPUT DISK DIRECTORY
510 POKE 752,0:REM TURN CURSOR BACK ON
520 POSITION 2,22:PRINT " RUN NUMBER? ";
REM PRINT MESSAGE ON MESSAGE LINE
525 POKE 82,2:REM SET LEFT MARGIN TO DEF
AULT
530 INPUT CHOICE:REM GET THE NUMBER OF T
HE PROGRAM TO RUN
540 CHOICE=INT(CHOICE):REM GET RID OF FR
ACTIONS
550 IF CHOICE<1 OR CHOICE>FILECOUNT THEN
 500:REM OUT OF RANGE FOR THIS MENU
600 FILE$=DISK$:REM THE NAME TO USE WITH
 A RUN STATEMENT MUST BEGIN WITH THE DIS
K DRIVE NUMBER
610 NAME$=ARRAY$(CHOICE*12-11,CHOICE*12)
```

```
:REM THE NAME OF THE PROGRAM INCLUDING E
XTRA SPACES
620 FOR LOOP=1 TO 12
630 IF NAME$(LOOP,LOOP)<>" " THEN FILE$(
LEN(FILE$)+1)=NAME$(LOOP,LOOP)
640 NEXT LOOP
700 TRAP 900:REM SET TRAP FOR DISK ERROR

720 POSITION 10,22:PRINT " LOADING )";NA
ME$:REM PRINT MESSAGE ON MESSAGE LINE
730 RUN FILE$:TRAP 34567:REM RUN THE PRO
GRAM AND TURN OFF TRAP
900 POSITION 10,22:PRINT " CAN'T RUN )";
NAME$:REM PRINT MESSAGE ON MESSAGE LINE
910 FOR PAUSE=1 TO 999:NEXT PAUSE:REM DE
LAY TO ALLOW TIME TO READ MESSAGE
920 GOTO 500:REM GO AND TRY AGAIN
```

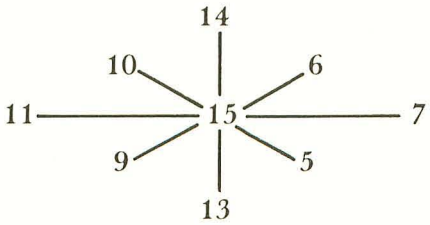
What To Do If You Don't Have Joysticks

Steven Schulman

Use of joysticks with the ATARI computer can add excitement to your programs. But what do you do if you don't have joysticks yet and aren't ready to buy them? Are you out of luck? Do you have to type in numbers to select from a menu of answers? Does it mean you can't use games like *IRIDIS' ZAP* or the latest from your computing magazines? No! There's another way.

We can look in amongst the bits and bytes that make up the memory of your ATARI. Any time you press a key on your keyboard, the value of the 764th word changes. By taking a PEEK at what number is there you can find out which key it was. Listing I shows you how to find out what the value will be when any key is pressed. Try running it and pressing different keys, shifted and unshifted, reverse video, etc. When you finish, use the BREAK key to stop the program.

"How does this help solve my problem of not having joysticks?" you may ask. To see this you have to know what happens when you use the joysticks. If your program has a line I = STICK(1), the value of I will be one of 9 possible values depending on the position of the joystick when that line is reached. The values will be



where the value of I = 15 means that the joystick is in the upright position. In addition, J = STRIG(1) will have a value J = 0 if the fire button is pressed and a value of J = 1 if the fire button is not pressed.

Returning to what we know about the value of the last key

pressed, we found that the values for the arrows were:

[up-arrow] = 14
[down-arrow] = 15
[right-arrow] = 7
[left-arrow] = 6

and the values for the shifted arrows were

Shift = 78 Shift = 71
Shift = 79 Shift = 70

Finally, the value for the space bar is 33.

We can therefore have the same results as we would get from using a joystick by using the arrows, shift arrows and space bar. The space bar will be our firing button, the arrows will be the obvious up, down, left and right, and the shift up will be to the upper left, the shift down will be to the upper right, the shift left will be to the lower left, and the shift right will be to the lower right. Any other key, or no key at all, being pressed is equal to the joysticks being in an upright position.

The routine in listing II will play the part of a joystick. After calling the subroutine the value of I will be the same as would have been returned by I = STICK(1) and the value of J will be the same as what would have been returned by J = STRIG(1). When you do buy your joysticks, simply replace the subroutine call and remove the subroutine from your program. Happy computing!

Listing I

```
100 I=PEEK(764)
110 ? "I=";I:REM PRINT THE VALUE OF THE
KEY PRESSED
120 POKE 764,255:REM TELL THE COMPUTER T
HAT NO KEY WAS PRESSED
130 FOR PAUSE=1 TO 500:NEXT PAUSE:REM SL
OW DOWN THE MACHINE SO YOU CAN READ THE
RESULTS
140 GOTO 100
```

Listing II

```
100 JOYSTICK=1000:REM LOCATION OF SUBROU
TINE
110 GOSUB JOYSTICK:REM CHECK THE 'JOYSTI
CK'
```

```
120 ? "THE 'JOYSTICK' HAS VALUE=";I
130 ? "THE 'FIRE BUTTON' HAS VALUE=";J
140 FOR PAUSE=1 TO 500:NEXT PAUSE
150 GOTO 110
1000 REM JOYSTICK SUBROUTINE
1010 I=PEEK(764)
1020 J=1
1030 POKE 764,255
1040 IF I=14 THEN I=14:RETURN
1050 IF I=79 THEN I=6:RETURN
1060 IF I=7 THEN I=7:RETURN
1070 IF I=71 THEN I=5:RETURN
1080 IF I=15 THEN I=13:RETURN
1090 IF I=70 THEN I=9:RETURN
1100 IF I=6 THEN I=11:RETURN
1110 IF I=78 THEN I=10:RETURN
1120 IF I=33 THEN I=15:J=0:RETURN :REM F
IRE BUTTON
1130 I=15:RETURN
```

Using the Atari Console Switches

James L. Brunn

If only one key is pressed, you can use these values: (PEEK (53279)):7 = no key, 6 = START, 5 = SELECT, and 3 = OPTION.

The colored console switches to the right of the typewriter keyboard are just the ticket for programs with special features. The names seem to indicate just the kind of things one might wish to do in a program. OPTION — What better key to step through a choice of options. SELECT — After stepping through the options, this key could be used to select the current option. START — This key might be used to transfer control back to the beginning of a sequence or to start the program over again.

The problem is, how does one read these keys? Well, read on: here is a method that works well for me. First, we note the memory location 53279 is used to indicate the condition of all three switches. It's done like this. If we just PEEK (53279) with no switches pressed, we find a seven. Holding down one or more of the keys while doing our PEEK returns a different number. The table below summarizes the values returned when a console key is pressed. X means that the key or keys are pressed.

Table 1

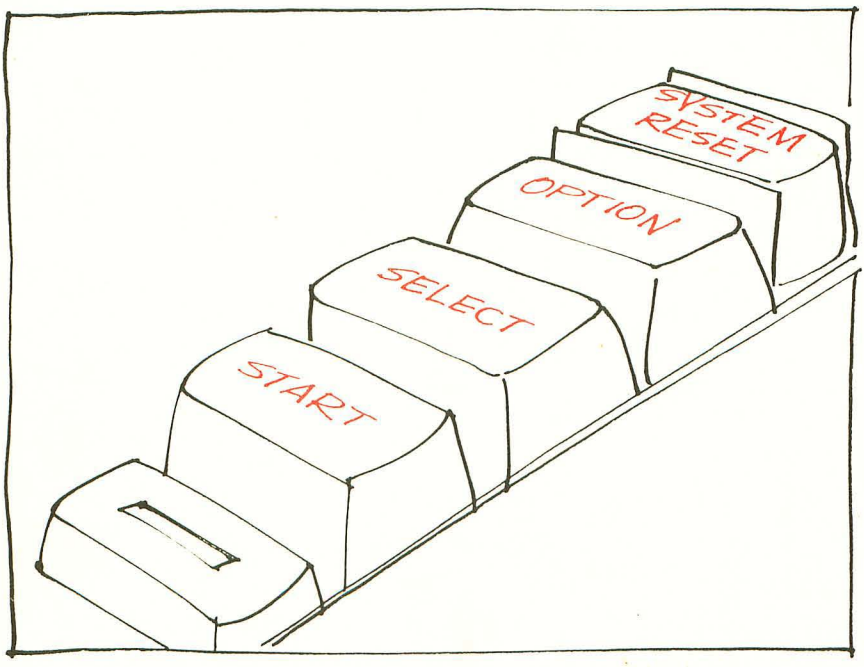
KEY VALUE	0	1	2	3	4	5	6	7
OPTION	X	X	X	X				
SELECT	X	X			X	X		
START	X		X		X		X	

Now let's use this knowledge in a program.

```
10 DIM DISPLAY$(23)
20 PRINT "{CLEAR}":POKE 752,1
30 POSITION 5,5
40 KEYS=PEEK(53279)
50 ON KEYS+1 GOSUB 100,110,120,130,140,1
50,160,170
60 PRINT DISPLAY$
70 GOTO 30
```

```
100 DISPLAY$="OPTION + SELECT + START ":
RETURN
110 DISPLAY$="OPTION + SELECT         ":
RETURN
120 DISPLAY$="OPTION + START          ":
RETURN
130 DISPLAY$="OPTION                  ":
RETURN
140 DISPLAY$="SELECT + START          ":
RETURN
150 DISPLAY$="SELECT                  ":
RETURN
160 DISPLAY$="START                   ":
RETURN
170 DISPLAY$="NO KEYS ARE PRESSED     ":
RETURN
```

Of course the subroutines here are very simple, but this method can easily be expanded to fit your needs.



Atari Meets The Real World

Richard Kushner

You've had your Atari computer for a while, reached the level of Commander in Star Raiders, killed 754 aliens in Space Invaders, learned the difference between PUT and GET and written some programs to amuse and astound your friends and family. Now you're looking for new worlds to conquer. Lurking out there past the peripheral plug on your Atari, and just beyond the end of your telephone line, is the real world. To get there from here means connecting your Atari to something that speaks the language of the outside world. As often as not, that means using an RS-232 compatible device. All this really means is that many devices that can hook onto your computer require a connection with voltage and signal specifications given by the RS-232 technical standard. Printers use it, modems use it, and a wide range of other peripherals are most comfortable communicating across it.

So what do you do? You can build an RS-232 interface, but if you're like me, your interest lies more at the programming end and you'd prefer something that comes ready to go. I'm pleased to report that such a device exists and works very well, thank you. It is the Atari 850 Interface Module and it does a lot of things to make the interfacing easy and understandable while, at the same time, providing versatility and supporting future expansion. This article will give a rundown of many of its useful features.

The Model 850 plugs into the Atari peripheral port and provides connections to daisy chain other devices (like the tape cassette) that do not require the interface. It has its own power supply (identical to the power supply for the Atari computer) and supports four RS-232 serial ports and one parallel port. The parallel port is intended for the Atari 825 Printer (a slightly disguised Centronics 737 Printer) and the manual describes the leads to all the pins in case your parallel device is not Centronics compatible. The four serial ports have different levels of support. Port #1 is intended for modems, Ports #2 and #3 are intended for serial printers and other generally receive-only devices, and Port #4 supports a 20ma current loop for teletype interfacing. The key word here is RS-232 "compatible." The connections are nine pin as

opposed to the 25 pin on standard RS-232 connectors and therefore cannot support all the possible RS-232 interconnections. Table 1 shows the connections that are available on the four ports. They should be adequate for most personal computer hook-ups. You'll have to make a connector to bridge the gap between the Model 850 and your RS-232 device. Carefully note that the pin designations are relative to each device, i.e., "receive" on the Model 850 goes to "send" on the peripheral and vice versa. Understanding that fact makes the interconnection reasonable and straightforward.

So far we've just scratched the surface. Inside the Model 850 resides its very own microprocessor. When the system is powered up, the Model 850 passes a handler routine and serial port information up to the computer and then waits for instructions. To transmit or receive over any port you must configure the port (or accept the default configuration). You can specify baud rate (45.5 to 9600, concurrent (two way) or block (output only) communication, port number, translation (how to send Atari ASCII so that your ASCII only device won't hiccup), bits per word, parity, and whether or not to monitor signals from the device at the other end. The length of the list and the variety within each item should give an indication of the versatility of this device. The cost to you (besides the purchase price, of course) is the 1,762 bytes of memory used when the interface loads its handler and tables into the computer memory. The following is a brief description of some of the features:

1. Baud Rate: virtually all common baud rates from 45.5 (60 words per minute for Baudout teletypes) to 9600 baud are supported and software-selectable. 300 Baud is the default value, making it immediately compatible with modems. I currently have a 1200 Baud serial printer running.
2. Translation Modes: three modes can be used —
   a.) no translation — just like it says, no changes are made on sending or receiving characters. This is only useful if you have some way of processing the characters or if your peripheral device understands the Atari version of ASCII, as, for example, if you are talking to another Atari computer.
   b.) light translation — on output End Of Line (EOL) is translated into Carriage Return (CR) and vice versa on input. Also the high bit is set to zero. This is the default mode.
   c.) heavy translation — this mode does what light translation does plus, on input, it looks for

correspondence between Atari ASCII and regular ASCII. If there is correspondence then it passes the character on and, if not, it translates the character into whatever you have specified as the "won't translate" character. On output, however, untranslatable characters are not sent at all.

4. I/O Modes: the interface handles either concurrent or block output. Input must always be done in the concurrent mode. In the block output mode, data is sent to the interface module in 32 character blocks, then the computer waits for the block to be transmitted before sending the next block. It is possible to force the computer to send a block of less than 32 characters so that data need not be lost. Concurrent mode output sends characters to a 32 character buffer which continuously empties out the other end. Programs are not held up in this operation unless the buffer fills up, in which case the computer must wait until space becomes available.

There are other capabilities built into the interface, but it is my intention here to give you a feel for the power of this device rather than to give a recitation of the technical manual. Don't be afraid to try to interface non-Atari RS-232 devices to your Atari. This interface module should be able to support whatever you have, although you'll probably have to experiment with the library of commands to get communication to take place. For example, the Atari directly supports the LPRINT command which outputs data to a parallel printer on the appropriate connection of the interface module. For my serial printer, it was necessary to 1) establish a port for the device (calling OPENing a channel; 2) configure the port to output data at the 1200 Baud rate that the printer required; and 3) look into whether or not I needed a line feed after a carriage

TABLE I

PORT -1		PORT -2	PORT -3	PORT -4
XMT	Transmit	XMT	XMT	XMT
RCV	Receive	RCV	RCV	RCV
DTR	Data Terminal Ready	DTR	DTR	DTR
DSR	Data Set Ready	DSR	DSR	
RTS	Request to Send			RTS
CRX	Carrier Detect			
CTS	Clear to Send			

return. All of this really required only two statements near the beginning of my program and the use of PRINT #4 (where 4 is the device number I had set up) rather than LPRINT, when I wished to

send data to the printer.

The Model 850 Interface Module has been carefully thought out to provide a great deal of versatility to the user. Several months of experience with this device has convinced me that it is a good investment in future expansion of my Atari computer system and a worthwhile item to have now.

Appendix A

Atari Memory Locations

Ronald Marcuse

DEC ADDR	HEX ADR	LABEL	DESCRIP
00014	000D	APPMHI	BASIC HIGHEST LOC- LSB
00015	000E	APPMHI	BASIC HIGHEST LOC- MSB
00016	0010	POKMSK	OS INTERUPT REQ ENABLE
00018	0012	RTCLOK	TV FRAME CNTR- LSB
00019	0013	RTCLOK	TV FRAME CNTR- NSB
00020	0014	RTCLOK	TV FRAME CNTR- MSB
00065	0041	SOUNDR	NOISY I/O FLAG (0=QUIET)
00077	004D	ATTRMOD	ATTRACT MODE FLAG,128=YES
00082	0052	LMARGIN	LEFT SCREEN MARGIN
00083	0053	RMARGIN	RIGHT SCREEN MARGIN
00084	0054	ROWCRS	CUR CURSOR ROW,GR WINDOW
00085	0055	COLCRS	CUR CURSOR COL, GR LSB
00086	0056	COLCRS	CUR CURSOR COL, GR MSB
00090	005A	OLDROW	PREV CURSOR ROW, GR WIND
00091	005B	OLDCOL	PREV CURSOR COL, GR LSB
00092	005C	OLDCOL	PREV CURSOR COL,GR MSB
00093	005D	DATCURS	DATA UNDER CURS, GR/MD 0
00096	0060	NEWROW	CURSOR ROW FOR DRAWTO
00097	0061	NEWCOL	CURSOR COL FOR DRAWTO LSB
00098	0062	NEWCOL	CURSOR COL FOR DRAWTO MSB
00106	006A	RAMTOP	TOP OF MEMORY,# OF PAGES
00128	0080	LOMEM	BASIC LOW MEMORY PNTR LSB
00129	0081	LOMEM	BASIC LOW MEMORY PNTR MSB
00144	0090	MEMTOP	BASIC MEMORY TOP PNTR LSB
00145	0091	MEMTOP	BASIC MEMORY TOP PNTR MSB
00186	00BA	STOPLN	1st HALF STOP/TRAP LINE #
00187	00BB	STOPLN	2nd HALF STOP/TRAP LINE #
00195	00C3	ERRSAV	ERROR NUMBER
00201	00C9	PTABW	PRINT TAB WIDTH (DEF 10)
00212	00D4	FRO	LOW BYTE VAL,USR FUNC
00213	00D5	FRO	HIGH BYTE VAL,USR FUNC
00251	00FB	RADFLG	RAD/DEG FLAG 0-RAD,6-DEG
00559	022F	SDMCTL	OS DIRECT MEM ACCESS CON
00560	0230	SDLSTL	OS DISPLAY LIST PNTR LSB
00561	0231	SDLSTH	OS DISPLAY LIST PNTR MSB
00562	0232	SSKCTL	OS SERIAL PORT CONTROL
00564	0234	LPENH	LIGHT PEN HORIZ VAL
00565	0235	LPENV	LIGHT PEN VERTICAL VAL
00580	0244	SYSRES	SYS RESET,COLD START >0
00623	026F	GPRIOR	PRIORITY SELECT (OS)
00624	0270	PADDL0	PADDLE 0
00625	0271	PADDL1	PADDLE 1
00626	0272	PADDL2	PADDLE 2
00627	0273	PADDL3	PADDLE 3
00628	0274	PADDL4	PADDLE 4
00629	0275	PADDL5	PADDLE 5
00630	0276	PADDL6	PADDLE 6
00631	0277	PADDL7	PADDLE 7
00632	0278	STICK0	JOYSTICK 0
00633	0279	STICK1	JOYSTICK 1
00634	027A	STICK2	JOYSTICK 2
00635	027B	STICK3	JOYSTICK 3
00644	0284	STRIG0	JOYSTICK TRIG 0

DEC ADDR	HEX ADR	LABEL	DESCRIP
00645	0285	STRIG1	JOYSTICK TRIG 1
00646	0286	STRIG2	JOYSTICK TRIG 2
00647	0287	STRIG3	JOYSTICK TRIG 3
00656	0290	TXTROW	CURSOR ROW, TEXT WINDOW
00657	0291	TXTCOL	CURSOR COL, TEXT 1st HALF
00658	0292	TXTCOL	CURSOR COL, TEXT 2nd HALF
00704	02C0	PCOLR0	OS PLAYER-MISSILE 0 COLOR
00705	02C1	PCOLR1	OS PLAYER-MISSILE 1 COLOR
00706	02C2	PCOLR2	OS PLAYER-MISSILE 2 COLOR
00707	02C3	PCOLR3	OS PLAYER-MISSILE 3 COLOR
00708	02C4	COLOR0	COLOR REGISTER 0
00709	02C5	COLOR1	COLOR REGISTER 1
00710	02C6	COLOR2	COLOR REGISTER 2
00711	02C7	COLOR3	COLOR REGISTER 3
00712	02C8	COLOR4	COLOR REGISTER 4
00741	02E5	MEMTOP	OS MEMORY TOP POINTER LSB
00742	02E6	MEMTOP	OS MEMORY TOP POINTER MSB
00743	02E7	MEMLO	OS LOW MEMORY POINTER LSB
00744	02E8	MEMLO	OS LOW MEMORY POINTER MSB
00752	02F0	CRSINH	CURSOR INHIBIT 0-ON,1-OFF
00755	02F3	CHACT	CHAR REG 1-BL,2-NOR,4-^
00756	02F4	CHBAS	CHAR BASE 224-UP,226-low
00763	02FB	ATACHR	LAST ATASCII CHAR
00764	02FC	CH	LAST KEY HIT, 255 CLEARS
00765	02FD	FILDAT	GR. FILL DATA (XIO)
00766	02FE	DSPFLG	DISPLAY FLAG 1=DIS CON CH
00767	02FF	SSFLAG	START/STOP PAGING (CON/1)
00794	031A	HATABS	HANDLER ADDR TBL,3 BY/HND
00832	0340	IOCB	IO CON BLOCKS,16 BYT/IOCB
53248	D000	HPOSP0	HORIZ POS, PLAYER 0
53248	D000	M0PF	MIS 0 - PLAYFIELD COLLIS
53249	D001	HPOSP1	HORIZ POS, PLAYER 1
53249	D001	M1PF	MIS 1 - PLAYFIELD COLLIS
53250	D002	HPOSP2	HORIZ POS, PLAYER 2
53250	D002	M2PF	MIS 2 - PLAYFIELD COLLIS
53251	D003	HPOSP3	HORIZ POS, PLAYER 3
53251	D003	M3PF	MIS 3 - PLAYFIELD COLLIS
53252	D004	HPOSM0	HORIZ POS, MISSILE 0
53252	D004	P0PF	PLAY 0 - PLAYFIELD COLLIS
53253	D005	HPOSM1	HORIZ POS, MISSILE 1
53253	D005	P1PF	PLAY 1 - PLAYFIELD COLLIS
53254	D006	HPOSM2	HORIZ POS, MISSILE 2
53254	D006	P2PF	PLAY 2 - PLAYFIELD COLLIS
53255	D007	HPOSM3	HORIZ POS, MISSILE 3
53255	D007	P3PF	PLAY 3 - PLAYFIELD COLLIS
53256	D008	M0PL	MIS 0 - PLAYER COLLISION
53256	D008	SIZEP0	SIZE- PLAYER 0
53257	D009	M1PL	MIS 1 - PLAYER COLLISION
53257	D009	SIZEP1	SIZE- PLAYER 1
53258	D00A	M2PL	MIS 2 - PLAYER COLLISION
53258	D00A	SIZEP2	SIZE- PLAYER 2
53259	D00B	M3PL	MIS 3 - PLAYER COLLISION
53259	D00B	SIZEP3	SIZE- PLAYER 3
53260	D00C	P0PL	PLAY 0 - PLAYER COLLISION
53260	D00C	SIZEM	SIZES FOR ALL MISSILES
53261	D00D	P1PL	PLAY 1 - PLAYER COLLISION
53261	D00D	GRAFP0	GRAPHICS, PLAYER 0
53262	D00E	GRAFP1	GRAPHICS, PLAYER 1
53262	D00E	P2PL	PLAY 2 - PLAYER COLLISION
53263	D00F	GRAFP2	GRAPHICS, PLAYER 2
53263	D00F	P3PL	PLAY 3 - PLAYER COLLISION
53264	D010	GRAFP3	GRAPHICS, PLAYER 3
53264	D010	TRIG0	JOYSTICK TRIGGER 0
53265	D011	GRAFM	GRAPHICS, ALL MISSILES

DEC ADDR	HEX ADR	LABEL	DESCRIP
53265	D011	TRIG1	JOYSTICK TRIG 1
53266	D012	COLPM0	PLAYER-MISSILE 0 COLOR
53266	D012	TRIG2	JOYSTICK TRIG 2
53267	D013	COLPM1	PLAYER-MISSILE 1 COLOR
53267	D013	TRIG3	JOYSTICK TRIG 3
53268	D014	PAL	PAL/NTSC INDICATOR
53268	D014	COLPM2	PLAYER-MISSILE 2 COLOR
53269	D015	COLPM3	PLAYER-MISSILE 3 COLOR
53270	D016	COLPF0	PLAYFIELD 0 COLOR
53271	D017	COLPF1	PLAYFIELD 1 COLOR
53272	D018	COLPF2	PLAYFIELD 2 COLOR
53273	D019	COLPF3	PLAYFIELD 3 COLOR
53274	D01A	COLBK	BACKGRND COLOR
53275	D01B	PRIOR	PRIORITY SELECT
53277	D01D	GRACTL	GRAPHIC CONTROL
53278	D01E	HITCLR	COLLISION CLEAR
53279	D01F	CONSOL	CONSOLE SWITCHES
53760	D200	POT0	POT 0
53760	D200	AUDF1	AUDIO CHANNEL 1 FREQ
53761	D201	AUDC1	AUDIO CHANNEL 1 CONTROL
53761	D201	POT1	POT 1
53762	D202	AUDF2	AUDIO CHANNEL 2 FREQ
53762	D202	POT2	POT 2
53763	D203	POT3	POT 3
53763	D203	AUDC2	AUDIO CHANNEL 2 CONTROL
53764	D204	AUDF3	AUDIO CHANNEL 3 FREQ
53764	D204	POT4	POT 4
53765	D205	POT5	POT 5
53765	D205	AUDC3	AUDIO CHANNEL 3 CONTROL
53766	D206	POT6	POT 6
53766	D206	AUDF4	AUDIO CHANNEL 4 FREQ
53767	D207	POT7	POT 7
53767	D207	AUDC4	AUDIO CHANNEL 4 CONTROL
53768	D208	AUDCTL	AUDIO CONTROL
53768	D208	ALLPOT	LINE POT PORT ST,READ B
53769	D209	KBCODE	LAST KEY (INTERNAL CODE)
53769	D209	STIMER	START TIMER
53770	D20A	RANDOM	RANDOM NUMBER GENERATOR
53770	D20A	SKREST	SERIAL PORT STATUS RESET
53771	D20B	POTGO	START POT SCAN SEQUENCE
53773	D20D	SEROUT	SERAIL PORT OUTPUT
53774	D20E	SERIN	SERIAL PORT INPUT
53774	D20E	IRQST	INTERUPT REQUEST STATUS
53774	D20E	IRQEN	INTERRUPT REQUEST ENABLE
53775	D20F	SKSTAT	SERIAL PORT STATUS
53775	D20F	SKCTL	SERIAL PORT CONTROL REG
54016	D300	PORTA	PIA CON JACK I/O (A) $3C
54017	D301	PORTB	PIA CON JACK I/O (A) $3C
54018	D302	PACTL	PORT A CONTROL REG
54019	D303	PBCTL	PORT B CONTROL REGISTER
54272	D400	DMACTL	DIRECT MEM ACCESS CON
54273	D401	CHACTL	CHARACTER CONTROL
54274	D402	DLISTL	DISPLAY LIST POINTER LSB
54275	D403	DLISTH	DISPLAY LIST POINTER MSB
54276	D404	HSCROL	HORIZONTAL SCROLL
54276	D01C	VDELAY	VERTICAL DELAY
54277	D405	VSCROL	VERTICAL SCROLL
54279	D407	PMBASE	PLAYER MISSILE BASE ADR
54281	D409	CHBASE	CHARACTER BASE ADR
54282	D40A	WSYNC	WAIT FOR HORIZ SYNC
54283	D40B	VCOUNT	VERTICAL LINE CNTR
54284	D40C	PENH	LIGHT PEN HORIZ VAL
54285	D40D	PENV	LIGHT PEN VERTICAL POS
54286	D40E	NMIEN	NON-MASK INTERUPT ENABLE
54287	D40F	NMIRES	NON-MASK INTERUPT RESET

Index

Notes

Notes