ATARI INFORMATION SERIES - VOLUME I

Carl M. Evans

ATARI BASIC

FASTER AND BETTER™

Carl M. Evans

ATARI BASIC Faster and Better

Editor — Charles Trapp – *Gunslinger*
Production — Cindy Hall – *Bouncer* . . .
Production Assistant — Debbie Cooke – *Runner* . . .
Everybody's Assistant — David Moore – *Magic*
Cover — D. J. Smith – *Bass Guitar*

This book is dedicated to Lewis Rosenfelder . . . *in grateful acknowledgment of his being that special kind of pioneer, who, being the first to go, carried the lantern high.*

ISBN 0 936200 29 4

10 9 8 7 6 5 4 3 2 1

 Printed in the United States of America

Final Thoughts

About the Author

Carl studied electronic engineering at the Georgia Institute of Technology, specializing in electro-optical communications. He first became involved with computers in 1971. Fortunately, for those of us wishing to know the secrets of "Atari magic," Carl's involvement appears deep and lasting. A champion has appeared, carrying a book of spells and incantations . . . *Atari BASIC Faster and Better.*

Carl is currently the manager of IJG's publications department. IJG publishes technical books – in non-technical language – about home computers.

He also runs *VERVAN Software*; a software and documentation consulting firm that has developed an extensive series of machine-language utilities for the Atari computer.

Carl has been writing on a professional basis since 1978, and is widely published in various technical and home computer magazines. He's been writing a regular tutorial column; *Tape Topics*, and a technical help column; *Tangle Angles*, for *ANTIC* magazine since 1982.

David E. Moore — *Wizard's Assistant*

From the Author

As I write my final thoughts about this book I am in a strange frame of mind. I started writing this book like I would have started any other project. I scoped the task and laid out a Gantt chart for it. Now, this book has changed my life. I was a successful project engineer for an aerospace company and had a good shot at climbing the corporate ladder. Now, I am the publications manager for a book publisher – namely IJG. I owe the change in my career to this book, and Harv Pennington. I have always loved to write, but I never thought I could make a career of it. Harv showed me that I could. As publications manager for IJG, I can continue my writing, and help other authors bring their hopes to fruition. I couldn't be happier. *Thanks, Harv.*

Carl M. Evans

Carl M. Evans

August 1983

Contents

Preface

The Atari 800 (and the 400) is a powerful computer... I've had my 800 since September of 1981, and each day I become ever more convinced of this.

You might think that the inherent limitations of a low-cost, mass-produced, eight bit computer would be frustrating. I've found quite the opposite to be true. The primary frustration I have with my 800 is that it is so complex that I can never seem to learn "all there is to know" about any one aspect of the thing. Every time I think I have it all down pat, I see a new program that does something I didn't even know could be done. Each day, I become more and more impressed with its capabilities.

Learning to program the 800 is like learning to play the piano. It's easy to play simple tunes (and you can really play tunes on the 800!) from the very first day, but you can spend a lifetime improving your technique and expanding your repertoire.

I started programming back in 1971, in college. I started out on a Burroughs 5500 and rapidly got involved with several other large computers (commonly referred to as mainframes) such as PDP/11, CDC 6400 and UNIVAC 1108. The very first programming I did was called "BATCH" programming. That means that all computer inputs are made using punched cards. After discovering the wonders of interactive programming on a CRT (video screen), I was of the opinion that batch programming was a diabolical device created to prevent people from learning how to program computers. I still have not changed that opinion.

Once I got out of college, I went to work for an aerospace company as an electro-optical engineer and spent most of my first two years writing special analytical programs for electro-optical guidance systems. I went along in this manner until June, 1979, when I bought myself a Radio Shack TRS-80 Model I with 16K of RAM. I soon found that working with 16K was analagous to memorizing only the left side of an equation, so I almost immediately upgraded to 48K. I felt much better, but kept seeing all those really eye-catching arcade games on the Atari computers, and I finally trashed my TRS-80 for a game machine — the Atari 800. I had learned from my previous experience, so I bought it with 48K of RAM and a disk drive. I have been delirious ever since.

The first problem I ran into was — yep, you guessed it — not enough memory for what I wanted to do. I never have been a fan of machine language, but I learned it to enhance my BASIC programs. Not being a masochist, I decided that I didn't want to have to rewrite a machine language subroutine every time that I needed one, so I started stuffing them into

BASIC subroutines that I could save as an ever expanding-library of "cook book" add ons to any other program I might want to write. This book is a spin off of those efforts.

This book is the result of the efforts I've made to make my BASIC programs run better and faster. Every time I'd have to stop to figure out a routine or technique, I'd put it in my programming notebook. Many times, I've had to throw out a routine and come up with an improvement, because the real test was whether or not it would work successfully on a day-to-day basis.

You won't find any trivia here. Each routine and technique solves one or more specific problems that you are likely to encounter when programming the Atari computer. Everything we'll discuss is pragmatic, with the goal of making the computer do what you want it to do, with the least programming effort.

The subroutines and techniques in this book don't attempt to be "all things to all people." I suppose it would be possible to write a sorting subroutine or a disk file-handling subroutine that could handle every possible operation you might want to perform. But why sacrifice execution speed? Why waste the memory? Instead, this book gives you relatively flexible routines with the documentation that allows you to modify them as your application requires.

I hope you'll find this book as valuable to you as it is to me. I use it daily as a reference in my programming work. Though some of the information can be found elsewhere, this book gives you a handy "one-source" reference, and now that these routines and techniques are explained in book format, documentation efforts for any program I write are greatly simplified. I can now refer anyone who reads one of my program listings back to this book, instead of filling up the program with memory-wasting remarks. If you adopt the same techniques and standards, you too can save a lot of time on documentation. You will be free to concentrate on the logic of the application, rather than the specific techniques required to make the computer perform better and faster!

Carl M. Evans

October, 1982

Introduction

What Is Faster and Better?

If we could define "faster" and "better" in a way that would apply to all programming problems, it would be a much simpler matter to design programs. Programming would become less an art and more of a science. It would be a simple matter of starting at point "A" and working to point "B."

A large part of our programming problem is deciding exactly what point "B" is. In programming and system design, we are working in a world of trade-offs. To make a system better in one way, we often have to make it not quite as good in another way. We must balance our limited resources to arrive at the best overall solution.

Let's talk about some of the trade-offs we must work with. Each can be maximized only at the expense of one or more other considerations. Every programming technique in your bag of tricks has its own advantages and disadvantages. If you can decide on the "mix" that is best for your application, you've cleared away one of the main roadblocks to developing your system.

Efficiency

How economically does the program use limited disk and memory space? We can save disk space through data compression at the expense of memory space, execution time and compatibility. We can conserve memory space at the expense of execution speed.

Execution Speed

How fast is it overall? How fast is it in those operations that are most critical? How fast and responsive is it for operator-paced operations? We can often make one operation faster by making another operation slower. We can often make a system faster at the expense of reliability or portability.

Programming Time

How long will it take to develop? Can deadlines be met? Given enough time, we can improve on many aspects of performance, but nearly every other performance consideration is achieved at the expense of programming time.

Function

Does it do the job intended? By limiting the project to only certain parts of the overall problem, we can save on programming time. By doing some things manually, we can improve on computer execution speed.

Workability

Does it do the job in a way that is practical and worthwhile to the user? We can maximize the functions performed by the computer, but by doing so, we often sacrifice workability.

Reliability

Is it vulnerable to operator errors or equipment malfunctions? Is it *crash-proof*? Is it bug free? We can improve on reliability at the expense of programming time, execution speed and efficiency.

Recoverability

How easily can the results of operator errors or equipment malfunctions be overcome? We can improve on recoverability at the expense of function, workability, design and programming time. We can improve on recoverability with special utility programs that reconstruct data that has been lost. We can live more dangerously in terms of reliability if the system is easily recoverable.

Ease of Operation

Is it operator-oriented? Are keystrokes minimized? Are operator entries consistent so that it can be run instinctively? We can usually make a system easy to operate at the expense of programming and design time, and memory efficiency.

Capacity

How much data can it handle? Programming a system to handle a small amount of data in memory can be a simple matter. For larger amounts of data, we get into the complexities of disk storage. To allow for capacity beyond that of a single disk adds even more complexity.

Portability

How easily can it be transferred for use on a different computer system? We can maximize portability at the expense of efficiency and execution speed. We can make a system easier to transfer by ignoring many of the capabilities and advantages that are unique to the system we are using.

Compatibility

How well does it tie-in with other systems the user might have? We can make the system perform more functions and work faster if we don't have to allow for compatibility with other systems.

Maintainability

If something goes wrong, how easy will it be to find the problem and correct it? We can improve on maintainability at the expense of function and efficiency. By conforming to programming standards we make the system more maintainable, but we sometimes sacrifice the ability to use procedures that are best suited to the application.

Ease of Modification

How easy will it be to modify the system to perform other functions that were not originally considered in the design? We can usually make it easier to modify with more programming and design time.

Understandability

How easily can a programmer other than the one who wrote the program understand the system? We can improve on understandability with extra programming and design time. By sacrificing some techniques that make the system more efficient or faster, we can make it more understandable to others.

Documentation

How well are the operating procedures, capabilities and limitations of the system explained? We can always improve on documentation by spending more time. Internal documentation, by inserting remarks in the body of the program text, can be achieved at the expense of execution speed and memory efficiency.

Attractiveness

How well designed are the video displays and printouts? Does it "sell" itself to those who must use it? We can make a program look good with more programming time and slower execution speed.

With the "tools" presented in this book, you can maximize the performance of your system according to the goals you have defined for the project at hand. Every function and program has been carefully designed to achieve one or more specific purposes. Most of the routines provide exceptional speed. Others operate slower than alternative techniques, but can provide a great savings in programming time. It is up to you to select your programming tools wisely and to test them for your specific application.

How To Use This Book

This book can be valuable to you whether you're a beginner, with only a few weeks' experience, or an expert programmer with many years of experience.

If you are new to programming, or the Atari 400/800 is new to you, you'll need first to get familiar with the capabilities and peculiarities of the Atari and the BASIC programming language. The best way is to work through the examples shown in your operating manuals, and to modify and experiment with them. Then you can give yourself simple programming challenges, and expand and modify your programs. There is no better teacher for programming than your own computer! It'll tell you when you've made an error, and you can try again and again. When you start looking at the examples in this book, you'll get ideas on how to do things differently (and, hopefully, better).

If you are new to assembly language programming, or if you have not been exposed to it at all, don't let the assembler listings in this book scare you off! Just gloss over them. You don't need to know 6502 assembly language, and you don't need to own an assembler/editor to use any of the routines in this book. If you want to learn assembly language for the Atari, I recommend *The Atari Assembler* by Don and Kurt Inman as a good introductory book. *6502 Assembly Language Programming* by Lance A. Leventhal and *Programming the 6502* by Rodney Zaks are excellent all around references. You can pick them up at most good computer stores. Then, after you get a feel for assembly language, you can start studying and modifying the assembly language subroutines shown here.

I've made no attempt in this book to duplicate anything that can be found in your instruction manuals, except where some amplification, clarification or summarization for your convenience is required.

The first four chapters of this book cover programming techniques that are important to the implementation of the routines found in the remainder of the book. They discuss subroutines, USR routines and techniques for managing the memory of your computer. Again, even if you are an experienced programmer, be sure to go through these chapters first. I guarantee you'll find new ideas and techniques that you've never seen published anywhere else!

Chapters 5 through 15 contain hundreds of ideas, tricks, subroutines and USR routines that can be implemented in your programs. It's unavoidable that when you use them, you will need to skip around, because video routines sometimes interact with disk routines, printer routines with disk routines, and so forth. So, before you begin using any of them, be sure to at least "skim" through the whole book so you'll know what's included.

To get the maximum usefulness from this book, you'll want to create a disk library of the subroutines, functions, test programs and utilities. That way you can merge what you need into any program that you might be writing.

Subroutines, Handlers and Shell Programs

The BASIC language, as you'll find it in the Atari computer, has around 82 commands and built-in functions. Have you ever considered which commnds and capabilities are the most important to you? My answer to this might suprise you, but to me, LIST and ENTER are without a doubt the most powerful and important commands!

I wouldn't have said that a year ago, but now that I've built up a library of programs, subroutines and functions, I almost never start a program from scratch. You could take away the NEW command (which clears out memory so you can begin writing a new program), and I wouldn't miss it.

A few years back I was in a computer store having a discussion with a salesman. He thought it was foolish to be in the programing business because "in a couple of years, every program will have been written!" Of course, that statement has turned out to be quite false, but from a programming productivity standpoint, we who program computers would do well to take the attitude that everything has already been written. Our job is to rearrange, modify, combine, insert and delete so as to come up with programs that can perform any one of an endless range of useful applications.

Subroutines

It doesn't take long to realize that the subroutine capability of BASIC can save you countless hours of work. The GOSUB command lets your program branch to another line, execute some logic, and then RETURN to resume execution with the next command following the GOSUB. Let's consider the advantages of a liberal use of subroutines:

- Subroutines save memory. Any significant operation that has to be performed more than once in your program only needs to appear once as a subroutine.

- Subroutines save programming time. With subroutines, you are not continually retyping the same logic over and over again.

- Subroutines provide flexibility. Simple modifications to a program having a liberal use of subroutines can make it perform new functions that were never considered when the program was originally written.

- Subroutines simplify testing and debugging. They let you break your program down into logical modules. Once you've completely tested a subroutine, you can forget about it.

- Subroutines free you. They allow you to concentrate on the overall logic and design of the application. You can forget about the details and complexities of those operations you perform again and again.

- Subroutines increase understanding. They make programs more readable and understandable. The details and complexities of common operations don't interrupt the "train-of-thought" in your main program. Even if a routine is used only once in a program, the benefits of readability can sometimes make it worthwhile to design that routine as a subroutine.

- Subroutines ease conversions. They can make your program more easily convertable to other computers and operating systems. For example, if a new computer system differs only in its disk handling instructions, you simply modify your disk handling subroutines. The rest of your program can remain unchanged.

- Subroutines can be libraries. You can create a library of subroutines on disk, and as you need them, merge them into the program you are writing.

This book gives you an extensive library of subroutines that can be used as you need them. Nearly all of them are shown with specific line numbers ranging from 19000 to 32000. You'll find no overlapping of subroutine line numbers shown in this book, except in a few cases where two subroutines perform the same function in a different way, and there would be no reason to have them both in the same program.

If you wish, you can change the line numbers and variables used by any of the standard subroutines in this book. But be aware that by doing so, you'll be missing out on one of the main benefits that this book provides — the pre-written documentation and detailed explanations. The line numbers and variables shown are arbitrary, but I've found that they work well for me. I trust that you'll find similar success with them.

Handlers

A "handler" is a group of subroutines and procedures that work together to perform a major function within a program.

In this book, for example, we'll be introducing a video display handler for the simplified programming of data entry and video display inquiries.

Handlers provide all the benefits of subroutines, but they go a level above and beyond single subroutines to provide system-wide standards for program organization, disk file organization and standardized operator-computer dialogues.

A handler gives you specific procedures for using a set of subroutines. To set up a handler within a program, you simply merge the subroutines required and modify, insert, or delete specific lines according to the instructions provided. A handler provides a starting point for you to begin the modifications required for any particular application. No attempt is made to make any one handler do everything for every possible application. Handlers are designed so that they can be modified for maximum efficiency in a particular application.

You'll find that the time-saving and standardization benefits of handlers are enormous. Once you adopt standard handlers into your programs, you'll wonder how you ever got along without them!

Shell Programs

A "shell program" can be any program that you've designed to be easily modified to perform entirely different applications.

For example, I have used a sophisticated shell program for nearly three years to develop hundreds of different applications. My accounts receivable system has all the handlers for menu selection, video display additions, changes, inquiries, transaction entry, report printing and disk file handling. By deleting certain routines, I've got a mailing list system. Other changes have made it into a general ledger system, an inventory control system, an accounts payable system and many other specialized applications.

When considering a new application, your first question should be, "What other applications that are already written have the same general structure?" When you think about it, just a few well-designed shell programs can be modified to perform almost any application, with up to a 90 percent savings in programming time!

Programming Conventions Used In This Book

Every serious programmer I have ever talked to has a special system for naming variables and organizing the code (program statements). I also have adopted a system or set of conventions that I use whenever I am developing a program. Every programmer's system is unique, but they all have certain common characteristics. The conventions outlined in the following two charts are those that I have been using with Atari BASIC. You may have already created your own system, but I suggest that you familiarize yourself with this system, since it is used extensively throughout this book. I invite you to adopt these conventions and to modify them or add to them as your needs dictate.

Figure 1.1 — *Variable Naming Conventions*

```
WORKING VARIABLES:

Temporary storage (very brief time)      - X, Y, Z, X$, Y$, Z$
Temporary storage (not so brief time)    - X1, X2, X3
                                           X1$, X2$, X3$
Flags (used to control branching)        - FLAG1, FLAG2, FLAG3, etc.
                                           or a key word (e.g., I
                                           might use a flag called
                                           "DEAD" and set it to 1
                                           when a monster is killed).

COUNTERS:

FOR-NEXT loops                           - LOOP1, LOOP2, LOOP3, etc.
                                           Replace "LOOP" with I, J,
                                           or K to save memory
Accumulators (long term counters)        - I usually use the name of
                                           the thing I am counting
                                           (e.g., I might use "SCORE"
                                           to keep a running total of
                                           how many points I have
                                           made during a combat).
```

CONSTANTS:	
Line numbers (used for indirect GOTO's and GOSUB's)	- Here, again, I usually use a descriptive name that tells me where the program is going to (e.g., I might use "DELAY" to identify a time delay routine).
Never changed numbers	- The number preceded by by a "Z" (e.g., Z1Ø=1Ø).
Seldom changed numbers	- Here, once again, I use a descriptive name (e.g., DAY, MONTH, YEAR).
String constants	- Descriptive names are also used here (e.g., NAMES$="Johnny").

Figure 1.2 — *Line Numbering Conventions*

1ØØ - 199	- Program name, copyright, author, version number (i.e., the title page).
2ØØ - 299	- Program initialization (e.g., DIMensioning, setting constants and variables).
3ØØ - 999	- Set USR variables and GOTO main program
1ØØØ - 9999	- ALL frequently used subroutines and loops. Put the most often called ones first.
1ØØØØ - 18999	- Main program
19ØØØ - 32ØØØ	- Seldom used subroutines and program closeout

I use these conventions extensively during the development of a program. If the program is likely to be used over and over with variations in the specific subroutines, then I leave it with these line numbers.

On the other hand, if I have written a program that is dedicated to a single narrow function and is therefore unlikely to need changing, I will renumber the program with a starting line number of 100 and step by 1, or a starting line number of 1000 and step by 10. I recommend against arbitrarily renumbering a program that is in development or one that you do not thoroughly understand.

How to Program Efficiently in BASIC

I remember the very first college course I took on computer programming. The professor devoted the majority of the class time to something called "flow charts." We were taught that organization was the real key to good programming. The professor was right. Most of you simply sit down at your computer and start entering code when you are trying to write a new program. If you are writing a relatively short program, you may get away without any planning. However, if the program is of some complexity, you will rapidly become lost in an ever deepening morass of confusion. The first lesson of efficient programming in BASIC or any other language is: **Plan the Program.** I will explain this concept in more detail in the following sections and then show you some general methods for minimizing the size and maximizing the speed of your programs.

Fundamental Concepts

A program can be written in many ways, but I usually go through the following steps:

1. I come up with an idea for a program.

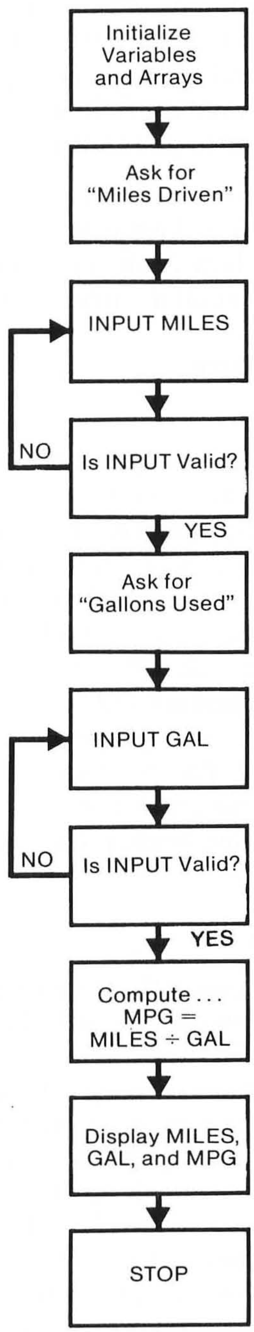
2. I write down everything that comes to mind about the idea. This is where I really define what I want the program to do. This step also serves to get my ideas down on paper so they won't be forgotten.

3. Now I categorize the notes I took in step two and assign labels that will relate to routines in my program. Any ideas that I get later can then be added to the proper catagory. If the program is going to be a very large one, I may even put my notes on 3x5 cards for easy indexing.

4. At this point I start a flow chart of the program. A flow chart is simply a block diagram of the program. Figure 2.1 illustrates a simple flow chart for a program that computes gas milage given the amount of gas used and the number of miles driven.

A flow chart is a road map of your program. Think of the lines as being roads and the boxes as towns and interchanges. Professional programmers make very fancy road maps that may go on for dozens of pages and use a special set of pictorial symbols to represent different kinds of operations. For example, a diamond tells you that the operation at that point in the program is a "branch on decision." A rectangle at that point would mean something completely different. However, this chapter is not meant to be a tutorial on flow charting. The point I am trying to make is that you should walk through the logic of your program and make sure that all possible results of an action are covered. Have you ever hit the wrong key in the middle of a game and had the program crash on you? The reason the program crashed is that the programmer was sloppy and wrote the program without a complete road map. The result was a one-way dead-end road that should not have been there. I suggest that you always flow chart any program that cannot be listed on a single piece of paper.

Figure 2.1 — *A Simple Flow Chart*



5. Once I have a flow chart, I can actually begin work on the program code. I tend to write my programs in modules that perform some particular function in the program. For example, an INPUT module, a SCREEN DISPLAY module, and so on. Each of these modules is written as a stand alone routine and is saved on my development disk with a descriptive file name, such as SETUP or DISPLAY1. Note the "1" on the second file name. I use a number like that to distinguish between different routines with similar functions. If I later revise a module, I use the extender to indicate the revision number. I *never* delete an old version until I am certain that the new version is working. I'll talk about backups a little later.

6. As I get my modules completed, I start to combine them into a program. This is particularly easy using the LIST and ENTER commands. I LIST each module to disk when I originally write it. Since the ENTER command does not erase memory like the LOAD command, I can concatenate the various modules by simply ENTERing each one from disk. You have to be careful to make sure that each module uses a different block of line numbers. The ENTER command will replace any existing lines with the newly entered lines if they have the same line number. While we have to be wary of this restriction at this time, I will show you how to use it to your advantage later in this book.

7. This is perhaps the most tedious and difficult step of all; I debug my program. Sure, I debugged each module as I wrote it, but the interaction of the modules is sometimes hard to predict. I don't know how I can express the importance of this step. Let's just say, "It only takes one bad bug to spoil a barrel . . . er, program."

Good Habits to Form

Making Backup Copies

You now know the general approach to use when developing a program, but there are some other things that you may need more guidance on. For example, I am sure that you have heard the term "backup" before. This term refers to more than just making an extra copy of a completed program. It is an essential tool that every serious programmer should use in the process of developing a program. I have no sympathy for the programmer who cannot meet a deadline because "the only copy of my source was stolen!" or lost, or whatever. Any programmer worth his salt will have at least one backup of anything he is working on. I am particularly paranoid and not only have several backup copies, but I also make it a point to backup anything I am working on every hour. That way I will never lose more than an hour's worth of programming. You don't have to go quite that far, but I do strongly suggest that you never let yourself be caught in the position of losing your "only copy" of a source. In other words, keep at least one backup copy of your work on a separate disk that is stored in a separate location.

Planning Video Layouts

When running a program, the primary interface between the program and you is the video screen. You should plan the video displays in your program with extreme care. The displays should give you the needed information easily and without straining your eyes or your mind. I have seen some programs that try to do everything with a single video display, when two or more displays would have been much better. There are no hard and fast rules for how much information should be on a given video display, but you should keep the display as simple as possible. Arcade games are an obvious exception to this rule of thumb. When designing a program, I generally use a video layout planning sheet similar to the ones shown in Figures 2.2 through 2.6. All that you need to have your own video layout planning forms is some rectangular graph paper, or you can simply photo-copy the ones in this book. Try this technique the next time you are designing a program, and I'm sure that you will not only save an appreciable amount of time, but your video displays will look much more professional.

Figure 2.2 — *GRAPHICS 0 Video Display Planning Sheet*

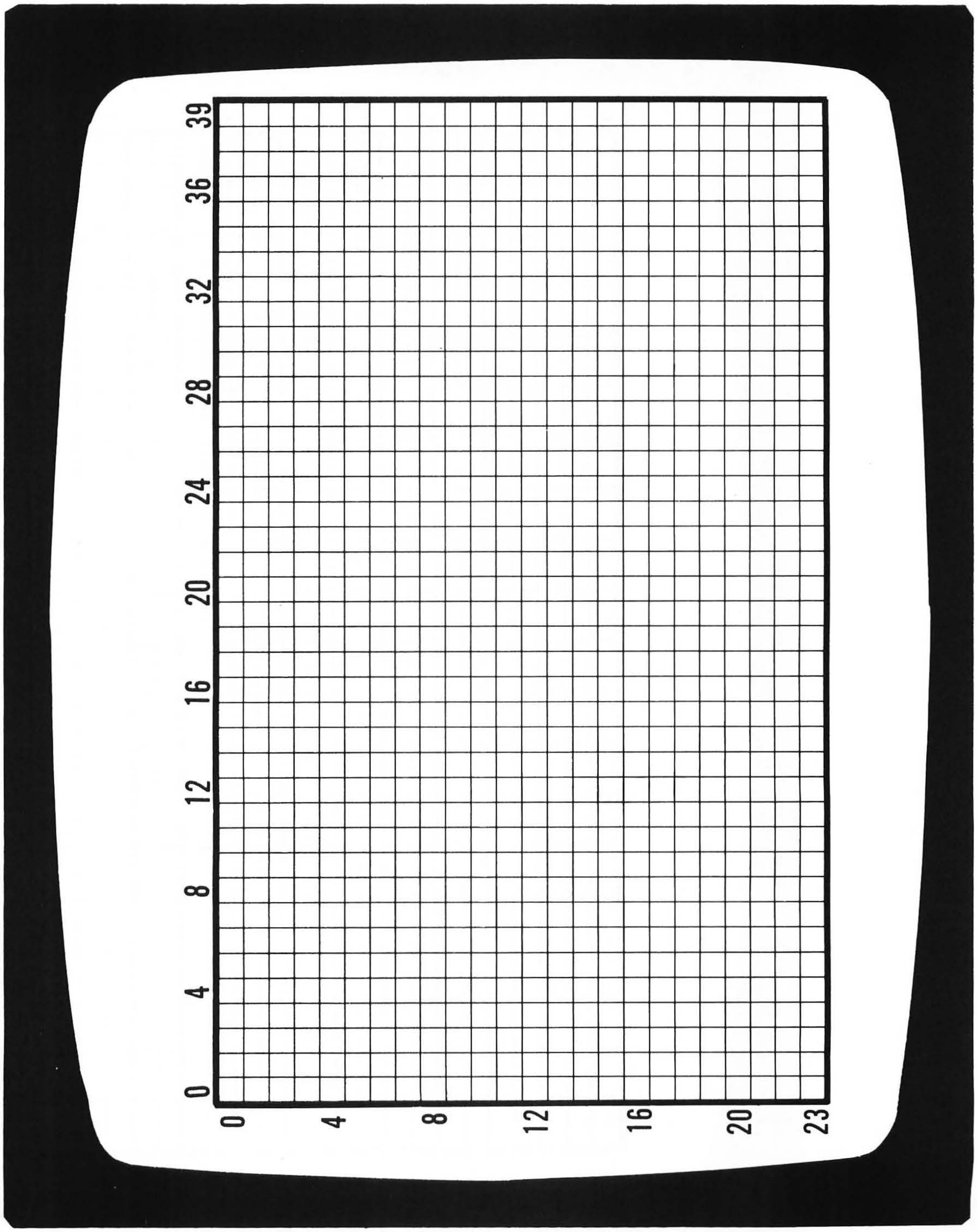


Figure 2.3 — *GRAPHICS 1 Video Display Planning Sheet*

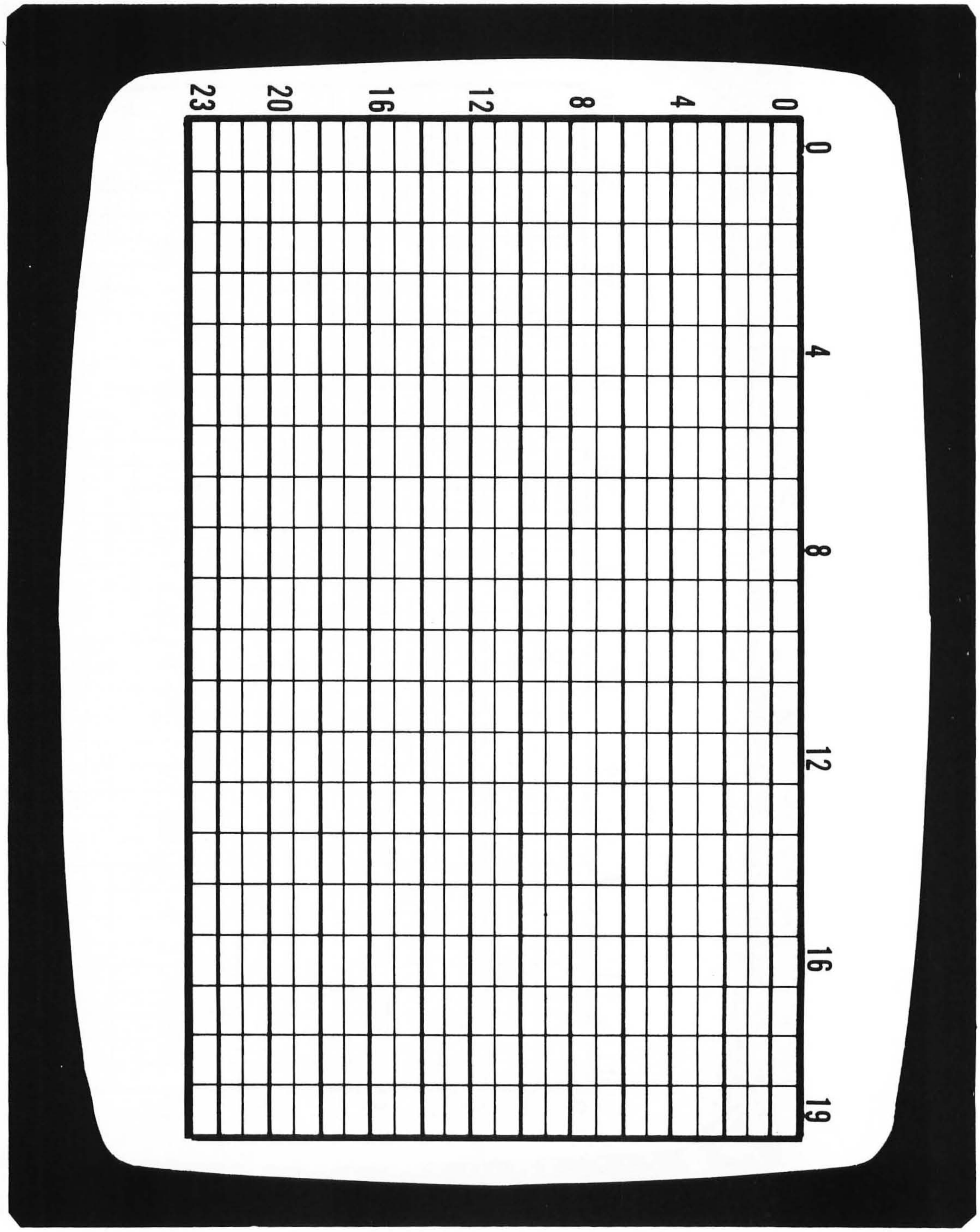


Figure 2.4 — *GRAPHICS 2 Video Display Planning Sheet*

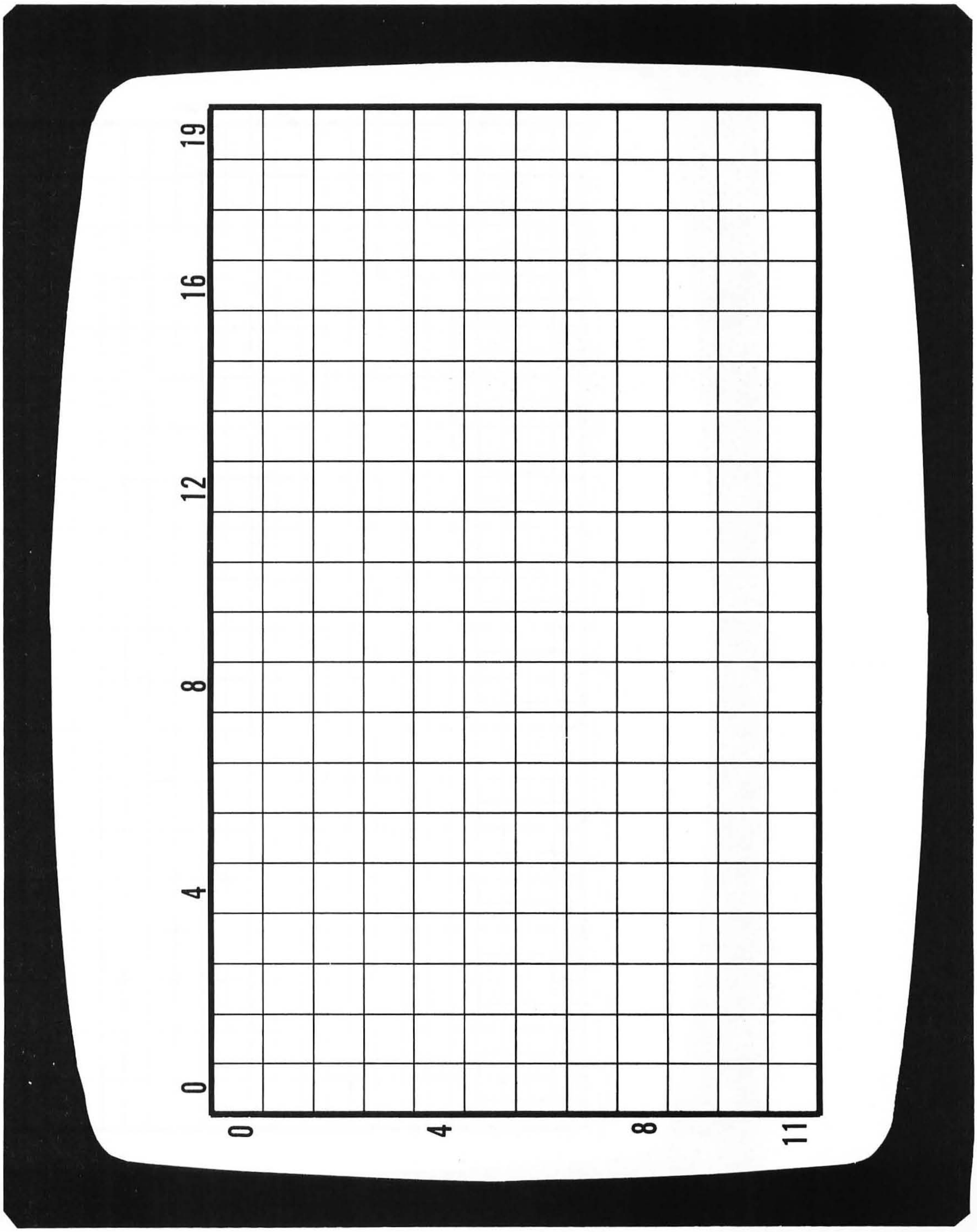


Figure 2.5 — *GRAPHICS 1 with Text Window Video Display Planning Sheet*

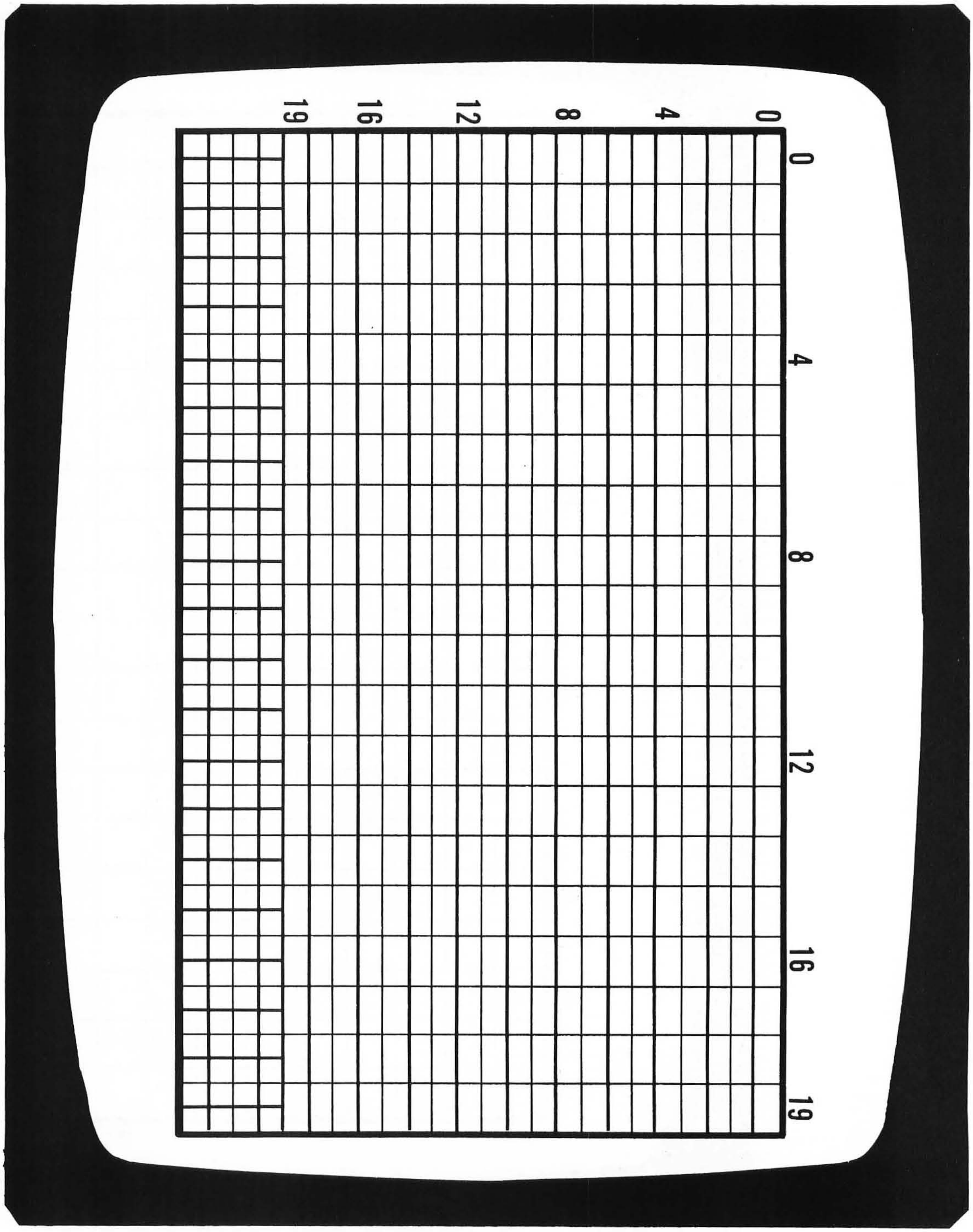
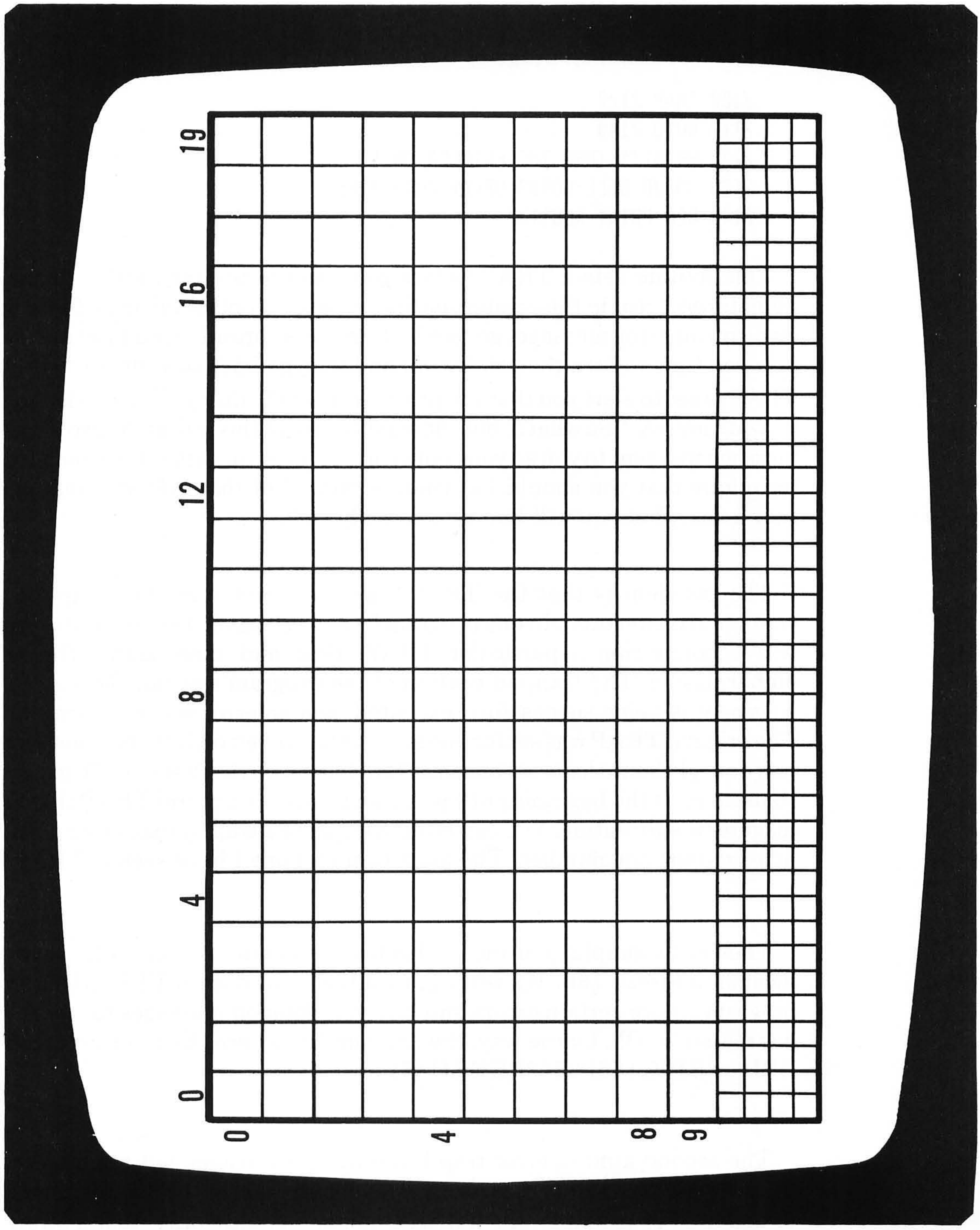


Figure 2.6 — *GRAPHICS 2 with Text Window Video Display Planning Sheet*



Setting Up Error Traps

An error trap is something used to prevent your program from crashing when a mistake is made while the program is running. Error traps take two general forms. The first form is the one that you are probably the most familiar with. The general form, as given in your BASIC handbook, is

```
TRAP aexp
```

The most common usage of the TRAP command looks something like this

```
2100 TRAP 2120
2110 GOTO 2140
2120 PRINT"* TURN THE PRINTER ON !"
2130 GOSUB BELL:GOSUB DELAY:GOTO 2100
2140 REM PRINT ROUTINE
```

This routine sets a TRAP before going to the printer routine. If the operating system detects *any* error in the printer routine, program control will transfer to line 2120 which will give you an error message, go to a bell ringing subroutine, go to a time delay subroutine (to give you time to turn the printer on) and then go back to your printer routine. This kind of TRAP is set to alert you that the printer is not ON after you have tried to send something to it. You may say, "So what?" but the significance of this is that the error was intercepted and a message was sent to you so you could correct the error *without crashing the program*! This is a technique that you should incorporate into all of the programs that you write. This may sound easy, but you will find that it really isn't.

The problem is that the TRAP stays set until something trips it or another TRAP command is executed. In one program I was working on, I spent hours trying to find out why I was encountering a particular TRAP, time and time again. The answer was almost embarrassing. The trapped portion of the program was not the source of the error. That routine was being successfully executed, and an error was occurring later in the program. The original TRAP was set for one specific kind of error, but once that section of the program was done, I should have set another TRAP to close the first one. To get around this problem, sometimes at the beginning of a program I will set a general TRAP that branches to an error diagnostic subroutine. You can effectively close a trap by executing a TRAP that points to a non-existent line number. The most common one I have seen is TRAP 40000.

The key to complex routines is that the error code you normally see displayed is stored in decimal address 195. By writing a routine that does a PEEK(195), you can have your program ignore certain errors and give you detailed messages for those errors that you are interested in. Oh, by the way, the line number where the error occurred is easily found by ERL=PEEK(186)+256*PEEK(187).

The second kind of error trap is a little less obvious, but nonetheless important. *Error trapping code* is a safety measure to prevent errors that the TRAP command is not designed to handle. For example, the TRAP command could detect that a string had been INPUT when a number had been asked for and either ignore the bad input, or tell you it was bad input and then ask for the input again. The TRAP command could not, however, tell whether or not a numerical input was a valid number for that routine. This kind of error trapping is usually handled by error trapping code. Let's say that a routine is asking for you to input a day of the month. The error trapping code would need to make sure that no number less than 1 or greater than 31 was input. The code might look something like this

```
100 PRINT"ENTER THE DAY OF THE MONTH ";
110 TRAP 100
120 INPUT DAY
130 IF DAY<1 OR DAY>31 THEN 120
```

This is a very simple example and more sophisticated input routines will be discussed later in this book, but the principles should be made clear here. The person using your program should not be allowed to make any input to the program that will cause the program to crash. In the trade, this is commonly referred to as "idiot proofing" your program. In Chapter 11 I'll show you how you can even prevent the BREAK key from stopping your program.

Minimizing Program Execution Time

The speed of a BASIC program is affected by many factors. The position of the code, the form of the code and the logic of the code all have some impact on program speed. There are a number of simple guidelines for maximizing the speed of a program.

The following list can be useful in helping you to speed your programs up. The methods are listed roughly in the order of most effective to least effective. The methods at the top of the list will typically be more effective than the methods at the bottom of the list.

1. Use machine language subroutines — tremendous time savings can be made by packing a loop in machine language and calling the loop by using the USR function.

2. Recode — there is generally more than one way to write a given routine. Restructuring the logic of a routine can sometimes yield great time savings.

3. Put frequently called subroutines and loops at the start of the program — since BASIC starts at the first line number in its search for a particular line number regardless of the position of the call, you can chop a good bit of time off your program's execution time by placing all frequently called subroutines and loops at the top of your program.

4. In loops, replace GOSUBs with in-line code — the additional time savings here is due to the fact that BASIC has to add and remove entries from the run time stack each time it encounters a GOSUB. If you eliminate the GOSUB, you also delete the time BASIC would use to keep track of the subroutine.

5. Replace "*" and "/" operators with equivalent "+" and "-" operators — the multiply and divide routines in Atari BASIC are very slow compared to the addition and subtraction routines.

6. Put multiple statements on a single program line — this is especially effective with loops since BASIC won't have to fetch the next line to continue the loop. It also serves the purpose of reducing the number of lines that BASIC will have to search through each time a search is needed.

7. Disable the screen display — the video screen display is maintained using a process called Direct Memory Access (DMA) that steals time from your computer when it isn't looking. This theft amounts to about 30 percent of your computer's time. You can regain this lost time by disabling the DMA process. To do this, you must POKE 559,0. This will speed up your program by 30 percent, but your video display will black out. To restore the DMA, simply POKE 559,34.

8. Use a lower graphics mode — using high resolution graphics will make your entire program run slower. You can save as much as 25 percent of the run time by using a lower graphics mode.

9. Replace seldom called subroutines with in-line code — BASIC spends a lot of time searching for line numbers and adding and subtracting subroutine pointers from the run time stack. If a routine is only used every once in a while, you can save some time by replacing the subroutine with in-line code.

10. When using nested loops, put the higher frequency loops inside — this gets back to the run time stack again. By putting a 100 cycle loop inside a 10 cycle loop, the number of times that BASIC has to update the run time stack is minimized, and your program runs faster.

11. Replace constants with variables — every time that BASIC encounters a constant, it must "interpret" it as a new number and convert it to BCD format. This takes up valuable time, and as you will see in the next section, it also eats up precious memory. So, you should replace any constants used more than three times with variables. For example, Z100=100:Z0=0:Z1=1 and so forth. This is especially important for constants inside large loops.

12. Reference variable names early in your program — now that you have replaced your constants with variables, you should be told the price you paid. The more variables that you use in a program, the longer it takes BASIC to find variables that were first referenced late in the program. If you set aside a special statement to intialize your most frequently used variables early in your program, you will speed up your program.

13. Delete all unnecessary spaces and remarks — the larger the program, the slower it is. Also, every REM statement is just one more line number for BASIC to sort through.

14. Use indirect addressing for GOTOs and GOSUBs — the reasons for this are similar to those given in (11) above.

15. Pack IF-THEN logic statements — this one is a little less obvious, and the rewards will vary, but you can replace an IF-THEN statement sequence such as

```
100 IF X<101 THEN Y = 0
110 IF X>100 AND X<301 THEN Y = 1
120 IF X>300 AND X<801 THEN Y = 2
130 IF X>800 THEN Y = 3
```

with a logical statement like

```
Y = (X>100)*((X>100)+(X>300)+(X>800))
```

Before you try anything like this, I suggest that you go back and re-read the section on logical operators in your BASIC manual. You will find that with a little study you can replace whole blocks of IF-THEN statements with a few logical expressions.

16. Use X*RND(0) rather than RND(0)*X — I'm still not sure why this one works, but experience has shown that it does save time. The same holds true for X*COS(Y) and the rest of the special functions.

Minimizing the Size of a Program

You will often find that you are hard pressed between two desires that usually conflict with one another. You would like to have your program run as fast as possible and yet use up as little memory as possible. There are a number of tricks you can use to accomplish one desire or the other. The previous section addressed the various ways you can speed up your program. This section will show you some other tricks you can use to reduce the amount of memory that your program will require. Those techniques that are listed in both sections deserve your close study.

1. Use machine language subroutines — a BASIC routine can take more than six times as much memory as a machine language subroutine that performs the same function.

2. Recode — inefficient code can easily take five times the memory as tight (efficient) code.

3. Remove all remark statements and unnecessary spaces — these things are not needed to run the program, and each one takes up valuable bytes of memory.

4. Replace constants with variables — this is especially good if the constant is referenced more than three times. BASIC stores each variable *once* as a six byte BCD number. Each reference to that variable uses only one byte. A constant, on the other hand, uses seven bytes each and every time it is used in your program. The savings is obvious.

5. Initialize numeric variables with a READ statement — this one is not an obvious technique. The trick is that DATA statements are stored in ATASCII code with each character using one byte. The normal assignment statement (e.g., Z100=100) uses seven bytes for each constant. This trick is most useful when you have a large number of constants and variables.

6. Use indirect addressing in GOTOs and GOSUBs — Atari BASIC allows you to use a variable instead of a line number in a GOTO or a GOSUB statement. Using this technique saves you roughly six bytes each time you use it. A side benefit is that using descriptive names for routines within a program makes it easier to follow the program's logic when you are analyzing it.

7. Get the garbage out of the variable name table — this applies primarily to the actual writing of the program. Every time you use a variable, it is added to the variable name table. This entry in the table stays there even though you may delete all uses of that variable in your program. You can get this garbage out of the variable name table by LISTing the program to cassette or disk, typing NEW, and then ENTERing the program back into memory. You can now SAVE the program with the cleaned up table. The savings here will range anywhere from a dozen bytes to truly large numbers. One program I was working on had gone through extensive changes, and I saved 500 bytes by getting rid of the garbage in the name table.

8. Minimize the number of variables in your program — each new variable requires an additional 8 bytes plus the bytes for its name.

9. Keep the variable names as short as possible — each character in the variable name uses up one more byte in the variable name table. It is tempting to use long descriptive names, but I believe that you will find that a short name can be just as good.

10. Put multiple statements on a single program line — you save three bytes each time you eliminate a program line by putting a statement on a line with another statement.

11. If a subroutine is only used once, replace it with in-line code — each unnecessary GOSUB and RETURN wastes bytes, and even more bytes are wasted if they are on separate lines.

12. String pack numerical arrays if the numbers are integers between zero and 255 — this allows you to store a six byte number in a single byte. I used this technique in a program where I had a numerical array with almost 3000 elements. I saved over 15K bytes using this technique.

13. Use self-deleting code — I'll show you later in this book how you can write a program in BASIC that can rewrite itself as it is running!

14. Replace the SETCOLOR statement with the proper POKE commands — this will save you 8 bytes each time you use it.

15. Initialize string variables with assignment statements — unlike technique (5), it takes less space for a string assignment statement than it does for the equivalent READ and CHR$ statements.

16. Chain your programs — using this technique allows you to run programs that would otherwise be too large for your computer. I'll show you some detailed examples in Chapter 5.

Using Machine Language in BASIC

Nothing beats the BASIC language for a quick and simple way to program your computer applications. BASIC lets us talk to the computer with commands and mathematical formulas that are quite consistent with the way we think and communicate. However, when super-fast execution speed and truly economical memory usage is required, we must speak to the computer in its native tongue, 6502 machine language. Once we have relieved the computer of the burden of translating from BASIC to 6502 commands, its true speed and power can take over.

It is usually not necessary to write a complete program in 6502 machine language. This is fortunate since writing machine language is a tedious, time consuming job. There are applications (such as arcade games) where the entire program has to be written in machine language, but for most home applications the memory savings is not needed, and the enhanced speed can be achieved by using machine language subroutines. I have found that the most useful approach is to set up a library of short machine language subroutines that you can call from BASIC when and where you need them. The USR command is a BASIC command that calls a machine language subroutine from a BASIC program. By making proper use of this technique, you can have the speed of 6502 machine language at your fingertips and still write your programs in BASIC.

In this book, we will discuss many special purpose machine language subroutines and illustrate how you can use them in your programs without ever having to learn 6502 machine language. Each subroutine will be listed in assembly code as well as the USR format. When you are ready to take the plunge into programming your own 6502 routines (if you haven't already), the listings will provide you with a good starting place. You can use an assembler/editor program to combine or modify any of the routines in this book.

Most of the USR routines in this book have one very important characteristic — they are relocatable, so you can load and execute them at any location in available RAM. In fact, in some cases, we will be using a technique where a USR routine might be relocated several times during the execution of the BASIC program.

You may have seen, or purchased, some of the excellent machine language programs for high speed sorting or other purposes, that are available for the ATARI. Although most of them perform their functions very well, there are four fundamental problems with many of these products:

1. They are designed to load at a specific location in memory. If you have a printer driver or some other USR routine that also must load at the same address, you are out of luck. My biggest gripe is with those programs that overwrite the disk operating system.

2. The programs are usually "protected" so you can't examine them, and the source listings are not provided with them. Without this kind of information you cannot see how they work, so it is very difficult to learn from them or modify them.

3. The programs often contain many routines in a single load package. You must load all of the routines you don't need in order to get the one that you need. This wastes valuable memory space.

4. If you write a program that uses a routine from one of these commercial packages, you will have to pay royalties if you decide to sell your program.

The USR routines in this book avoid those four problems. This way you get the maximum in flexibility and performance. You also don't need to worry about paying royalties as long as you don't resell these routines as a library or copy the pages out of this book to serve as your documentation.

Writing USR Routines with an Assembler/Editor

Let's look at how you would go about creating a 6502 machine language subroutine. I won't be too specific because your assembler/editor manual will give you detailed instructions, and the exact commands will depend upon the particular one that you are using. All examples in this book will be shown in the Atari Assembler/Editor Cartridge format. If you don't have an assembler/editor program, then just follow along — you don't need one to use the routines in this book!

For a sample program, we will write a short subroutine that will "instantly" fill the entire video screen with any character that you specify.

With an assembler/editor we can type in the following:

Figure 3.1 — *Screen Fill Assembly Listing*

```
1000 ;SFILL - USR ROUTINE TO FILL VIDEO SCREEN WITH ANY CHARACTER
1010 ;
1020         *=     $600          ;SET ORIGIN TO PAGE SIX
1030 ;
1040 POINT   =      $CC           ;POINTER LOCATION ON PAGE ZERO
1050 SCREEN  =      $58           ;HOLDS ADDRESS OF SCREEN MEMORY
1060 ;
1070         PLA                  ;GRAB NUMBER OF ARGUMENTS
1080         CMP    #$1           ;IS THERE ONLY ONE ARGUMENT?
1090 DEAD    BNE    DEAD          ;NO? THEN KILL THE COMPUTER
1100         PLA                  ;GRAB MSB OF ARGUMENT
1110         PLA                  ;GRAB LSB OF ARGUMENT
1120         TAX                  ;STORE LSB IN X REGISTER
1130 ;
1140         LDA    SCREEN        ;SETUP PAGE ZERO POINTER
1150         STA    POINT
```

```
1160          LDA    SCREEN+1
1170          STA    POINT+1
1180
1190          TXA                    ;RETRIEVE THE ARGUMENT
1200          LDY    #0              ;SET OFFSET TO ZERO
1210 LOOP     STA    (POINT),Y       ;WRITE CHARACTER TO SCREEN
1220          INC    POINT           ;POINT TO NEXT SPOT ON SCREEN
1230          BNE    LOOP            ;IF POINT<=FF THEN GO BACK
1240          INC    POINT+1         ;INC MSB
1250          LDX    POINT+1         ;ARE WE FINISHED?
1260          CPX    #$A0
1270          BNE    LOOP            ;NO? THEN GO BACK
1280          RTS                    ;RETURN TO BASIC
1290          .END
```

1. Line 1020 specifies an origin for the USR routine. We have selected **$600**, which is on *page six*. There are 256 bytes starting at **$600** that are almost always available for USR routines since BASIC normally does not use that area of memory. This location is great for machine language subroutines. You might think that 256 bytes is small, but a 256 byte machine language subroutine is really a very large subroutine! As long as you design the routine to be relocatable (i.e., no JSR's or JMP's within the routine), then the origin you select need not be the address you'll be using when you execute the routine. So if page six gets crowded, you can always move the subroutines.

For assembly and test purposes, I usually use page six. The tough decision as to exactly where I want all of my USR routines to reside I can leave to a later date. No matter where I put such a routine, I generally find out later that I'll need to move it again, so I end up string packing most of my routines.

Most assembler listings in this book will show an origin command specifying **$600** as the starting point of the program or routine. You can assemble them to any other origin that is compatible with your needs.

2.Lines 1040 through 1280 provide the actual program logic for the routine. One of the peculiar things about 6502 machine language is that certain commands are only possible using *page zero* memory locations. Line 1040 sets up a pointer on page zero, and line 1050 identifies a particular address on page zero that holds the starting address of the screen memory. When a USR command is used, the number of arguments being passed to the machine language routine is given by the first number on the *stack*. Line 1070 pulls this number off of the stack, and the next two lines make sure that only one argument is in the USR call. If you have done everything correctly so far, the next two bytes on the stack should be the MSB and LSB of the character you want printed on the screen. Since all ATASCII characters are (by definition) only one byte long, the MSB can simply be discarded.

The LSB is temporarily stored in the X register so we can recall it later. One side note at this point is that these numbers must be pulled from the stack to bring the BASIC return address to the top of the stack. The next few lines set up a counter on page zero with the initial value of the counter being the address of the start of screen memory. We then retrieve the character value to write to the screen and go through a little loop that writes this character to each location on the screen. The RTS command tells the computer to return control to the BASIC program at the address on the top of the stack.

3. Line 1290 satisfies the assembler requirement that there be an END statement.

Now that we have typed the routine in, we can assemble it to disk or tape as a machine language *object* file.

We can also save the *source code* that we just typed in to another file on disk or cassette. I always save my source code in case I want to modify the routine later. That way I won't have to type all of the code in again. Here is what the assembled listing of the screen fill USR routine will look like if you dump it to a printer:

Figure 3.2 — *Listing of Assembled Screen Fill Routine*

```
          1000 ;SFILL - USR ROUTINE TO FILL VIDEO SCREEN WITH ANY CHARACTER
          1010 ;
0000      1020         *=     $600        ;SET ORIGIN TO PAGE SIX
          1030 ;
00CC      1040 POINT   =      $CC         ;POINTER LOCATION ON PAGE ZERO
0058      1050 SCREEN  =      $58         ;HOLDS ADDRESS OF SCREEN MEMORY
          1060 ;
0600 68   1070         PLA                ;GRAB NUMBER OF ARGUMENTS
0601 C901 1080         CMP    #$1         ;IS THERE ONLY ONE ARGUMENT?
0603 D0FE 1090 DEAD    BNE    DEAD        ;NO? THEN KILL THE COMPUTER
0605 68   1100         PLA                ;GRAB MSB OF ARGUMENT
0606 68   1110         PLA                ;GRAB LSB OF ARGUMENT
0607 AA   1120         TAX                ;STORE LSB IN X REGISTER
          1130 ;
0608 A558 1140         LDA    SCREEN      ;SETUP PAGE ZERO POINTER
060A 85CC 1150         STA    POINT
060C A559 1160         LDA    SCREEN+1
060E 85CD 1170         STA    POINT+1
          1180 ;
0610 8A   1190         TXA                ;RETRIEVE THE ARGUMENT
0611 A000 1200         LDY    #0          ;SET OFFSET TO ZERO
0613 91CC 1210 LOOP    STA    (POINT),Y   ;WRITE CHARACTER TO SCREEN
0615 E6CC 1220         INC    POINT       ;POINT TO NEXT SPOT ON SCREEN
0617 D0FA 1230         BNE    LOOP        ;IF POINT<=FF THEN GO BACK
0619 E6CD 1240         INC    POINT+1     ;INC MSB
061B A6CD 1250         LDX    POINT+1     ;ARE WE FINISHED?
061D E0A0 1260         CPX    #$A0
061F D0F2 1270         BNE    LOOP        ;NO? THEN GO BACK
0621 60   1280         RTS                ;RETURN TO BASIC
0622      1290         .END
```

As a matter of comparison, try the following BASIC routine that does the same thing as the USR routine:

```
100 FOR X=40000 TO 40959:POKE X,10:NEXT X
110 GOTO 110
```

This BASIC routine takes almost seven seconds to fill the screen with a character as compared to the almost instantaneous action of the machine language subroutine! In addition, the BASIC routine uses 61 bytes of memory as compared to the 34 bytes used by the machine language routine.

How to Load and Execute USR Routines from Disk

Let's suppose that we have assembled the screen fill routine to a disk file named "SFILL.OBJ". We could also assemble the routine to memory and execute it from the assembler/editor's DEBUG facility, but the true test is whether or not you can load the routine while BASIC is in the computer. If you boot your computer with BASIC and ATARI DOS II, the computer will respond with READY. Type 'DOS' and hit the RETURN. This will put you in DOS, and the DOS menu will be displayed on the screen. Type "L" followed by a RETURN, and DOS will ask "LOAD FROM WHAT FILE?" You should type in "SFILL.OBJ". DOS will access the disk and load the object file for you. Since no run address was specified, DOS will simply redisplay the menu when the file is loaded. Now we need to get back to BASIC. From the DOS menu type a "B" followed by a RETURN, and you will be returned to BASIC.

Once you have gone back to BASIC, you can LOAD a BASIC program or write one to call up the machine language subroutine. The following short BASIC program is all you need to try out this USR routine:

```
200 PRINT CHR$(125)
210 PRINT "ENTER CHARACTER ";
220 OPEN #2,4,0,"K:":
    TRAP 220
230 GET #2,KEY
240 CLOSE #2
250 X=USR(1536,KEY-32)
260 GOTO 260
```

To execute the screen fill routine, enter this program into memory and RUN the program. The screen will "instantaneously" fill up with whatever character you specify.

The general form of the USR function, as given in your user's manual is:

```
X=USR(ADDRESS,aexp1,aexp2,aexp3,aexp4)
```

You can pass up to 126 *arguments* to the machine language routine by simply adding more *arithmetic expressions* to the USR call. The ADDRESS, which technically is also an arithmetic expression, is the memory location of the machine language subroutine. The address and the arguments are normal base ten (i.e., decimal) numbers. The arguments are always passed to the machine language routine as two byte numbers and are stored on the stack with the MSB on top of the LSB. A program can have any number of USR calls in it and you won't get into trouble as long as the proper routine is stored at the ADDRESS used in each call. I can't really teach you all about USR's since that is beyond the scope of this book. If you are really interested, I suggest that you pick up one of the books I mentioned back in the introduction to this book.

POKEing USR Routines into Memory

Each USR routine in this book is shown in POKE format. In other words, you will be given a list of the numbers that you will need if you want to POKE the routine into memory. This way you don't need an assembler/editor program, and you don't need to understand 6502 machine language. The screen fill USR routine we have been discussing can be loaded by POKEing the following 34 numbers into any 34 contiguous bytes of RAM:

```
104 201   1 208 254 104 104 170
165  88 133 204 165  89 133 205
138 160   0 145 204 230 204 208
242  96
```

Try these steps to see how it works:

1. Boot your computer with BASIC.
2. Type in the following program:

```
100 REM SFILL.DEM-SCREEN FILL FROM BASIC
110 DATA 104,201,1,208,254,104,104,170
120 DATA 165,88,133,204,165,89,133,205
130 DATA 138,160,0,145,204,230,204,208
140 DATA 250,230,205,166,205,224,160,208
150 DATA 242,96
160 MLSTART=1536:MLEND=1569
170 FOR X=MLSTART TO MLEND
180 READ Y:POKE X,Y:NEXT X
200 PRINT CHR$(125):PRINT
210 PRINT"SFILL.DEM - SCREEN FILL FROM BASIC"
220 PRINT:PRINT:PRINT"ENTER CHARACTER: ";
230 OPEN #2,4,0,"K:":TRAP 230
240 GET #2,KEY:CLOSE #2
250 X=USR(1536,KEY-32)
260 GOTO 260
```

3. RUN it. The program will "load" the machine language subroutine and ask you to enter a character. When you enter the character, the screen will "instantly" fill with that character.

The DATA statements in lines 120 through 160 specify a list of numbers which correspond to the 34 bytes in the USR routine. Lines 170 and 180 put the values into the first 34 bytes of page six memory, starting at **$600** (1536 decimal).

Since the screen fill routine is relocatable, you can replace the addresses in line 110 with another set of addresses, and it will run the same. You might try using another location. Just be sure that the location is safe and that the value of MLEND is 33 more than MLSTART.

Are you wondering where I got the numbers to be POKEd? The assembly listing gave us the *hexadecimal* codes for the USR routine. The command **STA POINT** in line 1150 generated the machine language instruction **85CC**. Converting this to decimal:

```
85 is 133 decimal
CC is 204 decimal.
```

The rest of the program was translated in a similar fashion. We could have also gotten the decimal numbers by PEEKing the appropriate memory locations after loading the object program from disk or cassette. Then from BASIC we could have printed the PEEK values from the first byte to the last byte of the routine by issuing the command:

```
FOR X=1536 TO 1569: PRINT PEEK(X),: NEXT X
```

I find even this a pain so I wrote a program that will read an object file from a disk and create a BASIC subroutine that I can later add to any other BASIC program. The program listed below is that program. I call it CONVERT:

Figure 3.3 — *CONVERT Program*

```
100 REM CONVERT 1.1- A PROGRAM THAT
        CONVERTS A ML OBJ FILE INTO
        BASIC DATA STATEMENTS
110 DIM FILE$(16),RESPONSE$(16):
    FIRST=30000
120 FILE$="D1:":
    PRINT CHR$(125):
    PRINT
130 PRINT "CONVERT.BAS ":
    PRINT :PRINT :
    PRINT "CAUTION! USE ONLY OBJECT FILES":
    PRINT :PRINT
140 PRINT "ENTER NAME OF FILE ";
150 FILE$(4,14)="           ":
    TRAP 140:
    CLOSE #1
160 INPUT RESPONSE$
170 FILE$(4,14)=RESPONSE$
180 OPEN #1,4,0,FILE$
190 TRAP 380
200 GET #1,X:
    GET #1,X:
    GET #1,X:
    GET #1,Y
210 MLSTART=X+256*Y
220 GET #1,X:
    GET #1,Y
230 MLEND=X+256*Y:
    SIZE=INT((MLEND-MLSTART)/8+1):
    LAST=FIRST+2*SIZE
240 PRINT CHR$(125):
    POSITION 2,4
250 PRINT LAST+10;" MLSTART = ";MLSTART
260 PRINT LAST+20;" MLEND   = ";MLEND
270 PRINT :
    PRINT "CONT":
    POSITION 2,0
280 POKE 842,13:
    STOP
290 POKE 842,12
300 FOR LINE=FIRST TO LAST STEP 2
310 PRINT CHR$(125):
    POSITION 2,4
320 PRINT LINE;" DATA ";
330 FOR I=1 TO 7:
```

```
    GET #1,X:
    PRINT X;",";:
    NEXT I
340 GET #1,X:
    PRINT X:
    PRINT :
    PRINT "CONT":
    POSITION 2,0
350 POKE 842,13:
    STOP
360 POKE 842,12
370 NEXT LINE
380 PRINT :
    PRINT "CONT":
    POSITION 2,0:
    POKE 842,13:
    STOP
390 POKE 842,12
400 CLOSE #1:
    PRINT CHR$(125):
    POSITION 2,4
410 PRINT LAST+30;" FOR X=MLSTART TO MLEND"
420 PRINT LAST+40;" READ Y:
    POKE X,Y:
    NEXT X":
    PRINT :
    PRINT "CONT":
    POSITION 2,0
430 POKE 842,13:
    STOP
440 POKE 842,12
450 PRINT CHR$(125):
    PRINT "PRESS SELECT TO LIST TO CASSETTE"
460 PRINT "PRESS START TO LIST TO DISK":
    PRINT "(FILENAME IS ML.BAS) ";
470 IF PEEK(53279)=6 THEN 500
480 IF PEEK(53279)=5 THEN 520
490 GOTO 470
500 LIST "D:
    ML.BAS",FIRST,FIRST+40+2*SIZE
510 GOTO 530
520 LIST "C:",FIRST,FIRST+40+2*SIZE
530 END
```

Object File into BASIC Data Statements

CONVERT will read any DOS compatible binary load (i.e., object) file from a disk, create a BASIC subroutine starting at line 30000 and then save the routine to either cassette or disk for later recall. The resulting subroutine is in the LIST format, so it can be added to the end of any of your other BASIC programs by typing ENTER"ML.BAS". You may then SAVE your BASIC program with the built-in machine language subroutine. The full power of this merging capability will be discussed in more detail in Chapter Five. We will also discuss some other interesting techniques for embedding machine language routines later in this chapter.

Saving USR Routines To Disk

Each machine language routine in this book is shown in POKE format. That is, you will be given a list of numbers that you can POKE, starting at any safe address in memory. Once you have POKEd the numbers indicated for the USR routine, you can record that routine to disk, using any valid disk file name. Suppose that you want to save the screen fill USR routine that we have been using for our example:

1. First boot your computer up with BASIC and an ATARI DOS II disk.

2. Write or load a program that will POKE the required values at the proper addresses in memory. Here is a program that does the job for the SFILL routine:

```
19000 REM SFILL.LST - SCREEN FILL
19001 DATA 104,201,1,208,254,104,104,170
19002 DATA 165,88,133,204,165,89,133,205
19003 DATA 138,160,0,145,204,230,204,208
19004 DATA 250,230,205,166,205,224,160,208
19005 DATA 242,96
19006 MLSTART=1536:
      MLEND=1569
19007 FOR X=MLSTART TO MLEND
19008 READ Y:
      POKE X,Y:
      NEXT X
19009 END
```

3. Run the program. This reads the data statements and POKEs the numbers into memory.

4. Now go to DOS. To do so, type 'DOS' followed by a RETURN.

5. When in DOS you can use the binary save command to save the machine language subroutine to disk. To do this, enter a K in response to the DOS menu. DOS will respond with SAVE-GIVE FILE, START, END, INIT, RUN. You can then save the screen fill routine by entering the following command:

```
SFILL.OBJ,600,622
```

Remember to use hexadecimal addresses with this command. Ignore the INIT and RUN parameters at this time.

6. From now on, whenever you know that you will be calling the SFILL routine in a BASIC program, you can either perform a binary load (the DOS L command) as described before, or you can use CONVERT to create a BASIC subroutine. In the next section, I'll show you an even better technique — *string packing.*

If you wish, you can rename SFILL.OBJ to any other valid file name. To do this, you will use the rename command on the DOS menu (the E command). If you do rename it, for example to FILLSCRN, and it no longer has the OBJ extension, your command to load it from the DOS menu will be L followed by entering FILLSCRN.

If you are using one of the other disk operating systems that is available for the ATARI, you will have to refer to your DOS manual to translate what we have just discussed. Although I am aware that a number of such programs are on the market, I have found that Atari's DOS II meets my needs. I bought two of these other DOS's but I seldom find a need to use them.

Loading USR Routines Into Strings

We can load *any* relocatable USR routine into a string! There are some tremendous advantages to this technique. First, if the code is relocatable, we no longer have to worry about where the routine is stored. The starting address of the routine can easily be found by using the ADR command in BASIC. Second, we can store the *string packed* routine in an ordinary BASIC disk file, which may contain a whole library of routines, for faster and more convenient loading from BASIC.

The screen fill routine can be loaded into the string SFILL$ with the following program commands:

```
100 DIM SFILL$(34)
110 SFILL$(1)=CHR$(104):SFILL$(2)=CHR$(201)
120 SFILL$(3)=CHR$(1):SFILL$(4)=CHR$(208) .... etc.
```

Note that ATARI BASIC does *not* support a command like

```
SFILL$ = CHR$(104)+CHR$(201)+CHR$(1)+CHR$(208) .... etc.
```

This is a tedious and time consuming method, but it does work. I am essentially lazy so I modified the program CONVERT to do all of this work for me. I call the new program DATAPAK.

Figure 3.4 — *DATAPAK – A Program to Pack Machine Code into a String Array*

```
100 REM DATAPAK.BAS - A STRING PACKER
110 GOTO 260
120 REM ** KEYBOARD ENTRY ROUTINE **
130 OPEN #3,4,0,"K:"
140 GET #3,RES:
    IF RES<68 OR RES>155 THEN 140
150 CLOSE #3:
    RETURN
160 REM ** TIME DELAYS **
170 FOR Z=1 TO 500:
    NEXT Z:
    Z=0:
    RETURN
180 FOR Z=1 TO 25:
    NEXT Z:
    Z=0:
    RETURN
190 REM ** INITIALIZE DATA ARRAY **
200 PAC$(1)="":
    PAC$(NP)="":
    PAC$(2)=PAC$:
    RETURN
```

```
21Ø REM ** AUTO RETURN ROUTINE **
22Ø POSITION 2,Ø
23Ø POKE 842,13:
    STOP
24Ø POKE 842,12:
    RETURN
25Ø REM ** MAIN PROGRAM **
26Ø DELAY2Ø=18Ø:
    DELAY=17Ø:
    KEY=13Ø:
    ARM=2Ø2:
    TITLE=21Ø:
    Z=Ø:ZZ=Ø
27Ø PRINT CHR$(125):
    GRAPHICS 2+16:
    SETCOLOR 4,8,Ø
28Ø POSITION 16,2:
    PRINT #6,"datapak"
29Ø POSITION 14,8:
    PRINT #6," (C) 1982"
3ØØ POSITION 12,1Ø:
    PRINT #6,"vervan software":
    GOSUB DELAY:
    GOSUB DELAY
31Ø GRAPHICS Ø:
    POKE 752,1:
    GOTO 1Ø3Ø
32Ø TRAP 32Ø:
    GOSUB 142Ø:
    PRINT CHR$(253):
    POSITION 2,12:
    PRINT "ENTER NUMBER OF DATA ELEMENTS ";
33Ø INPUT N:
    IF N<1 THEN 32Ø
34Ø IF FRE(X)>2.5*N THEN 36Ø
35Ø GOSUB 142Ø:
    POSITION 2,12:
    PRINT "INSUFFICIENT MEMORY FOR THAT NUMBER":
    GOSUB DELAY:
    GOTO 32Ø
36Ø GOSUB 142Ø:
    PRINT CHR$(253):
    POSITION 2,12:
    PRINT "IS ";N;" CORRECT ?";:
    GOSUB KEY
37Ø IF RES=121 OR RES=89 OR RES=155 THEN 39Ø
38Ø N=Ø:
    RES=Ø:
    GOTO 32Ø
39Ø LL=8Ø:
    X=1:
    IF N/LL=INT(N/LL) THEN X=Ø
```

```
400 LNUM=INT(N/LL)+X:
    NP=LL*LNUM:
    NL=N
410 DIM PAC$(NP),FIX34(20),FIX9B(20):
    FIX34(1)=0:
    FIX9B(1)=0
420 GOSUB 1420:
    POSITION 2,12:
    PRINT "RAW DATA MUST BE INTEGER (0-255)"
430 PRINT :
    PRINT "DATA MUST BEGIN AT LINE # >2000":
    PRINT
440 PRINT "FOR SPEEDIER PROCESSING":
    PRINT "THE SCREEN WILL BLACKOUT DURING RUN."
450 TRAP 450:
    GOSUB DELAY:
    GOSUB DELAY:
    GOSUB 1420:
    POSITION 2,12:
    PRINT "ENTER FIRST LINE # FOR PACKED DATA"
460 POSITION 2,14:
    PRINT "MUST BE =>2000 AND <=32000  ";
470 INPUT FIRST:
    IF FIRST<2000 OR FIRST>32000 THEN 450
480 POSITION 2,16:
    PRINT "ENTER LINE # INCREMENT =>10 ";
490 INPUT DELTA:
    IF DELTA<10 OR DELTA>1000 THEN 480
500 IF DELTA*LNUM+1+FIRST>32500
    THEN PRINT "TRY A SMALLER INCREMENT OR FIRST#":
    GOSUB DELAY:
    GOTO 450
510 REM ** TURN OFF VIDEO DMA **
520 POKE 559,0:
    GOSUB 200:
    TRAP 530:
    GOTO 570
530 POKE 559,34:
    PRINT CHR$(125):
    POSITION 2,12
540 IF PEEK(195)=6 THEN PRINT "*** OUT OF DATA ERROR ***":
    PRINT "THERE ARE ONLY ";X-1;" DATA."
550 IF PEEK(195)=8 THEN PRINT "*** BAD DATA AT ";X;" ***"
560 GOTO 1010
570 IF MERGE THEN FOR X=1 TO N:
    READ DAT
580 IF  NOT MERGE THEN FOR X=1 TO N:
    GET #2,DAT
590 IF DAT=INT(DAT) THEN 610
```

```
600 POKE 559,34:
    PRINT CHR$(125):
    POSITION 2,12:
    PRINT "*** NON-INTEGER DATA AT ";X;" ***":
    GO TO 1010
610 IF DAT>=0 AND DAT<=255 THEN 640
620 PRINT CHR$(125):
    POSITION 2,12:
    PRINT "*** DATA OUT OF RANGE AT ";X;" ***"
630 POKE 559,34:
    GO TO 1010
640 IF DAT=34 THEN DAT=32:
    FIX34(Z)=X:
    Z=Z+1
650 IF DAT=155 THEN DAT=32:
    FIX9B(ZZ)=X:
    ZZ=ZZ+1
660 PAC$(X,X)=CHR$(DAT):
    NEXT X:
    PRINT CHR$(125):
    POSITION 2,12:
    PRINT "RAW DATA LOADED..."
670 REM ** TURN ON VIDEO DMA **
680 POKE 559,34:
    PRINT CHR$(253):
    GOSUB DELAY
690 REM ** WRITE NEW LINES OF CODE **
700 FOR LOOP=1 TO LNUM:
    X=(LOOP-1)*LL+1:
    Y=LOOP*LL:
    IF NL<LL AND NL>0 THEN Y=X-1+NL
710 NL=NL-LL:
    PRINT CHR$(125):
    POSITION 2,12:
    PRINT FIRST+DELTA*(LOOP-1);" DAT$(LEN(DAT$)+1)=";CHR$(34);
720 FOR Z=X TO Y:
    PRINT CHR$(27);PAC$(Z,Z);:
    NEXT Z:
    PRINT CHR$(34)
730 PRINT :
    PRINT "CONT":
    GOSUB ARM:
    NEXT LOOP
740 REM INSTALL FIX FOR 34 AND 155
750 IF FIX34(1)=0 THEN 800
760 FOR N=1 TO 25:
    PRINT CHR$(125)
770 IF FIX34(N)=0 THEN POP :
    GOTO 800
780 LNUM=LNUM+1:
    PRINT FIRST+DELTA*LNUM;"DAT$(";FIX34(N);",";FIX34(N);")=CHR$(34)"
```

```
790 PRINT :
    PRINT "CONT":
    GOSUB ARM:
    NEXT N
800 IF FIX9B(1)=0 THEN 850
810 FOR N=1 TO 25:
    PRINT CHR$(125)
820 IF FIX9B(N)=0 THEN POP :
    GOTO 850
830 LNUM=LNUM+1:
    PRINT FIRST+DELTA*LNUM;"DAT$(";FIX9B(N);",";FIX9B(N);")=CHR$(155)"
840 PRINT :
    PRINT "CONT":
    GOSUB ARM:
    NEXT N
850 FINAL=FIRST+DELTA*LNUM
860 PRINT CHR$(125):
    POSITION 2,12:
    PRINT "PACKING COMPLETE.":
    PRINT CHR$(253):
    GOSUB DELAY
870 REM ** OUTPUT ROUTINE **
880 GOSUB 1420:
    POSITION 2,12:
    PRINT "PRESS SELECT TO WRITE STRING TO TAPE"
890 PRINT "PRESS  START TO WRITE STRING TO DISK":
    PRINT " (THE DISK FILE WILL BE PACKED.DAT)"
900 IF PEEK(53279)=5 THEN 990
910 IF PEEK(53279)=6 THEN 930
920 GOTO 900
930 PRINT :
    PRINT :
    PRINT "PUT DISK INTO DRIVE#1 AND PRESS RETURN":
    GOSUB KEY
940 TRAP 970
950 LIST "D:
    PACKED.DAT",FIRST,FINAL
960 GOTO 1010
970 IF PEEK(195)=139 THEN PRINT CHR$(125):
    PRINT CHR$(253):
    POSITION 2,12:
    PRINT "TURN ON YOUR DISK DRIVE"
980 GOSUB DELAY:
    GOTO 930
990 PRINT :
    PRINT :
    PRINT "PREPARE BLANK TAPE AND PRESS RETURN"
1000 LIST "C:
     ",FIRST,FINAL
1010 POKE 752,0:
     END
1020 REM ** DATA INPUT ROUTINE **
```

```
1030 GOSUB 1420:
     POSITION 2,12:
     POKE 752,1:
     PRINT "PRESS OPTION FOR DISK/TAPE INPUT":
     TAPE=0
1040 PRINT "PRESS SELECT FOR OBJECT/MERGE FILE":
     MERGE=0:
     PRINT "PRESS  START TO  CONTINUE"
1050 IF PEEK(53279)>5 THEN 1120
1060 IF PEEK(53279)=3 THEN TAPE= NOT TAPE:
     GOSUB DELAY20
1070 IF PEEK(53279)=5 THEN MERGE= NOT MERGE:
     GOSUB DELAY20
1080 IF TAPE THEN POSITION 19,12:
     PRINT "DISK/TAPE"
1090 IF  NOT TAPE THEN POSITION 19,12:
     PRINT "DISK/TAPE"
1100 IF MERGE THEN POSITION 19,13:
     PRINT "OBJECT/MERGE"
1110 IF  NOT MERGE THEN POSITION 19,13:
     PRINT "OBJECT/MERGE"
1120 IF PEEK(53279)=6 THEN 1140
1130 GO TO 1050
1140 IF  NOT MERGE THEN 1260
1150 IF  NOT TAPE THEN 1190
1160 PRINT :
     PRINT :
     PRINT "PUT THE TAPE IN THE RECORDER":
     PRINT "PRESS START TO BEGIN INPUT":
     GOSUB DELAY20
1170 IF PEEK(53279)<>6 THEN 1170
1180 POKE 764,12:
     ENTER "C:":
     POKE 752,0:
     GOTO 320
1190 PRINT :
     PRINT :
     PRINT " PUT THE DISK IN DRIVE #1"
1200 PRINT "THE FILE MAY HAVE ANY LEGAL NAME,BUT":
     PRINT "THE EXTENDER MUST BE .LST"
1210 PRINT "ONLY THE FIRST *.LST FILE ON":
     PRINT "THE DISK WILL BE LOADED"
1220 PRINT :
     PRINT "PRESS START TO BEGIN INPUT":
     GOSUB DELAY20
1230 IF PEEK(53279)<>6 THEN 1220
1240 POKE 764,12:
     ENTER "D:*.LST"
1250 GOTO 320
1260 IF TAPE THEN 1400
1270 DIM FILE$(16),RESPONSE$(16):
     FIRST=30000
```

```
1280 FILE$="D1:":
     GOSUB 1420:
     PRINT
1290 PRINT :
     PRINT :
     PRINT "CAUTION! USE ONLY OBJECT FILES":
     PRINT :
     PRINT
1300 PRINT "ENTER NAME OF FILE ";
1310 FILE$(4,14)="           ":
     TRAP 1300:
     CLOSE #2
1320 INPUT RESPONSE$
1330 FILE$(4,14)=RESPONSE$
1340 OPEN #2,4,0,FILE$
1350 GET #2,X:
     GET #2,X:
     GET #2,X:
     GET #2,Y
1360 MLSTART=X+256*Y
1370 GET #2,X:
     GET #2,Y
1380 MLEND=X+256*Y:
     N=MLEND-MLSTART+1
1390 GOTO 390
1400 OPEN #2,4,0,"C:"
1410 GOTO 1350
1420 PRINT CHR$(125):
     POSITION 14,2:
     PRINT "DATAPAK":
     PRINT :
     PRINT "STRING PACK MACHINE LANGUAGE PROGRAMS"
1430 RETURN
1440 REM ** START DATA AFTER THIS  **
```

DATAPAK, like CONVERT, will read a machine language object file on disk and create a BASIC subroutine (in this case a simple string assignment) and store it on disk or cassette. In addition, you may load the data by using a LISTed file that can be merged to the end of DATAPAK. This way it is possible to use the DATA statements created by CONVERT as inputs to DATAPAK.

You can follow the technique outlined earlier to get the POKE codes onto disk as an object file and then use DATAPAK on the file to generate the packed strings.

Using a string packed machine language routine is really quite easy. All you need to do is modify the USR call to reference the starting address of the string. For example, the USR call to use SFILL was

```
X=USR(1536,KEY-32)
```

You can change this call to refer to a string, SFILL$, that contains the machine code as follows

```
X=USR(ADR(SFILL$),KEY-32)
```

It is possible to use strings for routines that are longer than 255 bytes by concatenating smaller strings into one large string. Be sure to DIMension the large string to the extended size that you want to use. This is really an advanced technique that is going beyond the scope of the current discussion, so I will leave exploring that topic up to you. I will give you a hint on where to start looking — DATAPAK uses this technique.

You will find that the string packing techniques we have discussed in this section provide one of the fastest, most flexible, and most memory efficient methods for handling USR routines. It really would not be useful to show you every routine in this book in the string packed format, so I won't. However, I do recommend that you take the POKE values, load them into memory, save them to disk as an object file, and then use DATAPAK on the file to create nice, neat, string packed arrays.

Magic Memory Techniques

General Methods

"Any given program will expand to fill all available memory."

If you have been programming the ATARI home computer for any length of time, you will be able to attest to the truth of that statement. It always seems that, no matter how much memory or disk space your computer has, it is never enough. This chapter will give you the techniques that you will need to make the most of the memory space you have.

We have all seen entertainers who dazzle their audiences by feats of "memory," such as memorizing everyone's name or the contents of each page in a magazine. These "super" memory powers are really based upon simple techniques that anyone can learn. This section will give you some simple techniques that can, likewise, give your computer some amazing memory powers. You will find that when you know how to control your computer's memory and move data quickly, your programs can reach a new generation of performance!

How Much Memory Do You Really Have?

The 6502 microprocessor is an "8-bit" device that uses two 8-bit bytes to define an address. Since each bit of each byte can be either a zero or a one (i.e., two states), the 6502 computer chip can uniquely identify (address) 2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2 16-bit addresses. That list of two's is equal to 65536. This number is usually referred to as 64K. Your computer is therefore a 64K computer regardless of how much "memory" you have. Your typical "48K" computer really is a 64K computer that has 48K of *available* memory and 16K of dedicated memory. The 48K Atari actually has only 37K of memory for BASIC programmers.

Dedicated memory usually consists of a ROM based (i.e., you can read it but not write to it) operating system. A 32K computer is the same 64K machine with only 32K (32768 bytes) of *available* memory. The missing 16K can still be "addressed," but since there isn't any RAM or ROM at any of those addresses, you can't use this addressing capability for anything. For example, you could write a BASIC program that resided in 32K, but PEEKed and POKEd data in the missing 16K. This program could be written on a 32K computer, but obviously you would have to RUN it on a 48K computer. I am not recommending that you try this, since debugging would be almost impossible. The point is that you should try to think of memory size as being under your control.

Figure 4.1 — *Table of Available Memory Limits without DOS*

Atari Catalog	Top Byte Hexadecimal	Top Byte Decimal	Bottom Byte Hexdecimal	Bottom Byte Decimal
16K	3FFF	16383	7ØØ	1792
32K	7FFF	32767	7ØØ	1792
48K	BFFF	49151	7ØØ	1792

Note: Cartridge A uses AØØØ-BFFF (4Ø96Ø-49151)
Cartridge B uses 8ØØØ-9FFF (32768-4Ø959)

Figure 4.1 gives you a table of available memory addresses for different *size* Atari 800 computers. The cartridges will use the addresses shown regardless of how much "memory" you have.

When you put a cartridge in your computer, the computer switches the connection of those addresses to point to the cartridge. If you normally have RAM at those addresses, then the RAM is disabled. The bottom 1535 bytes of your computer's RAM is reserved and used by the operating system and/or BASIC.

There are some small areas of memory below 1792 that you can use. You have already heard me refer to *page six* in the last chapter. A "page" is defined to be 256 bytes. The pages in your computer's memory are numbered by using the first part of the lower address boundary. Thus page zero starts at hexadecimal (hex) address **0000** and page six starts at hex address **0600**. Available memory starts at page seven. Page six is reserved for your private use. The operating system, BASIC, DOS and the assembler/editor will normally not use this page of memory. This is why I usually locate my machine language subroutines on page six. There are only a handful of addresses available on page zero for you to use. Later in this chapter I will demonstrate a way to use the page zero addresses from BASIC. The addresses you can use are shown in Figure 4.2.

Figure 4.2 — *Available Memory on Page ZERO*

Hex Address	Decimal Address	Restrictions
BO...CA	176-2Ø2	Not from BASIC
CB...D1	2Ø3-2Ø9	None
D4...D5	212-213	Used to return an argument to BASIC from a USR call

PEEKing a Two Byte Address

As you know, when you PEEK any location in memory, the result is a number from 0 to 255. Likewise, the second argument of a POKE command must be a number from 0 to 255.

Often, it is necessary to work with a decimal address that is located somewhere in memory. A memory address can be defined by two hex numbers **MM** and **LL**. Two bytes are needed because the largest possible eight bit binary number is 11111111, which is **FF** in hex and 255 in decimal. Any number larger than 255 is defined as **MMLL** and is stored in two consecutive bytes of memory as **LLMM**. **LL** is commonly called the least significant byte (or LSB). **MM** is called the most significant byte (or MSB). A decimal address can be calculated from **LL** and **MM** by the following equation:

```
ADDRESS = 256*PEEK(address of MM) + PEEK(address of LL)
```

An example for an address stored on page zero might be

```
ADDRESS = 256*PEEK(2Ø4) + PEEK(2Ø3)
```

POKEing A Two Byte Address Into Memory

From time to time, you may want to change an address that is stored in memory at a known location. To POKE a two byte address into any two contiguous memory locations, your command is

```
POKE LOCATION+1,INT(ADDRESS/256):
POKE LOCATION,(ADDRESS-256*INT(ADDRESS/256))
```

An example that stores the number 40000 in memory locations 203 and 204 on page zero is

```
POKE 2Ø4,INT(4ØØØØ/256):
POKE 2Ø3,(4ØØØØ-256*INT(4ØØØØ/256))
```

Be very careful when you are POKEing things into memory. If you POKE the wrong number into the wrong location, you can cause your computer to behave *very* strangely.

How to Reserve a Block of Memory for Private Use

Sometimes you would like to reserve a block of memory that is safe from the depredations of BASIC and the operating system. You might use this reserved area to store data or machine language subroutines. When you power your computer up with BASIC, the bottom of memory available for a BASIC program is established by the operating system as a number that we will call LOMEM. LOMEM is stored in addresses 743 and 744. BASIC, also, keeps its own pointer at the bottom of memory in addresses 128 and 129. We will call this number MEMLO. Likewise, the top of available memory is stored somewhere.

I have seen many different methods used to reserve a block of memory so that it is safe from being modified by BASIC or the operating system. Many of these involve moving the apparent top of memory. I recommend against using any of those techniques. RAMTOP changes are only safe *after* the program has executed a GRAPHICS command for the highest graphics mode that is used anywhere in the program, and even then the reserved memory isn't truly safe.

Some people use a special machine language subroutine to alter the LOMEM and MEMLO pointers. The problem with this technique is that the special machine language subroutine also has to be stored somewhere. A much safer method is to first load a small BASIC program that changes the pointers to the bottom of memory and RUN it just before running your main program. The program listed below will move the bottom of BASIC memory pointers:

Figure 4.3 — *RESERVE.LST – Protects a Block of Memory*

```
19930 REM RESERVE.LST - PROTECTS A BLOCK OF MEMORY
19931 REM SIZE = NUMBER OF BYTES TO RESERVE
19932 ADDRESS=256*PEEK(744)+PEEK(743)+SIZE
19933 MM=INT(ADDRESS/256):
      LL=ADDRESS-256*MM
19934 POKE 743,LL:
      POKE 744,MM
      REM MOVE MEMLO UP
19935 POKE 128,LL:
      POKE 129,MM
      REM MOVE LOMEM UP
19936 POKE 8.0:
      REM RESET WARM START FLAG
19937 X=USR(40960)
      RESTART BASIC
```

This routine examines the LOMEM and MEMLO pointers, and changes them to a new value. Any program that is LOADed or RUN after it, will ignore the reserved block of addresses. The "trick" to this routine is that BASIC only loads a new MEMLO value from the operating system's LOMEM pointer when a NEW command is used. MEMLO is not updated for a LOAD or a RUN. A curious tidbit is that even SYSTEM RESET does not trigger an update of MEMLO under normal conditions. RESERVE.BAS fixes it so NEW, RUN, LOAD and SYSTEM RESET will reset the pointers to the place *you* specify.

BASIC Variable Lister

Many times I have been writing a program and mis-typed a variable name. I have found it useful to have a way to quickly and easily obtain a complete list of all of the variable names in a program. There are several programs on the market such as VERVAN'S FULMAP that will generate a complete list of variable names along with the lines that they are used in. I use FULMAP(a machine language program) when I am documenting a new program for my files or when I am trying to analyze someone else's BASIC program. However, when I am simply interested in how many variables I have used and what they are, I use the little program listed below:

Figure 4.4 — *VLIST.LST – A Routine to List the Variables in a BASIC Program*

```
19940 REM VLIST.- VARIABLE ANALYZER
19941 PRINT CHR$(125):
      PRINT :
      PRINT "VLIST.LST - A BASIC VARIABLE ANALYZER"
```

```
19942 PRINT :
      PRINT :
      PRINT "PRESS START FOR OUTPUT TO THE SCREEN":
      PRINT "PRESS SELECT FOR OUTPUT TO A PRINTER"
19943 TRAP 19961:
      IF PEEK(53279)=6 THEN OPEN #3,8,Ø,"S:":
      GOTO 19946
19944 IF PEEK(53279)=5 THEN OPEN #3,8,Ø,"P:":
      GOTO 19946
19945 GOTO 19943
19946 POKE 2Ø3,Ø:
      POKE 2Ø4,PEEK(13Ø):
      POKE 2Ø5,PEEK(131):
      POKE 2Ø6,PEEK(134):
      POKE 2Ø7,PEEK(135):
      POKE 2Ø8,1
19947 POKE 752,1:
      PRINT #3:
      PRINT #3:
      PRINT #3;"VLIST - BASIC VARIABLE ANALYSIS":
      PRINT #3:
      PRINT #3:
      PRINT #3
19948 PRINT #3;"VARIABLE NUMBER = ";PEEK(2Ø3);"  ":
      PRINT #3;"VARIABLE NAME   = ";
19949 IF PEEK(PEEK(2Ø4)+256*PEEK(2Ø5))<128 THEN
      PRINT #3;CHR$(PEEK(PEEK(2Ø4)+256*PEEK(2Ø5)));
1995Ø IF PEEK(PEEK(2Ø4)+256*PEEK(2Ø5))>127 THEN
      PRINT #3;CHR$(PEEK(PEEK(2Ø4)+256*PEEK(2Ø5))-128);
19951 IF PEEK(PEEK(2Ø4)+256*PEEK(2Ø5))<128 THEN
      GOSUB 19984:
      GOTO 19949
19952 IF  NOT (PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)))
      THEN POKE 2Ø8,Ø:
      GOSUB 19962
19953 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=64 OR PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=65
      THEN POKE 2Ø8,Ø:
      GOSUB 1ØØ32
19954 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=128 OR
      PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=129 THEN POKE 2Ø8,Ø:
      GOSUB 19978
19955 IF PEEK(2Ø8) THEN GOTO 19961
19956 POKE 2Ø9,Ø:
      GOSUB 19986:
      GOSUB 19984:
      POKE 2Ø3,PEEK(2Ø3)+1
19957 IF (PEEK(2Ø4)+256*PEEK(2Ø5))<(PEEK(132)+256*PEEK(133))
      THEN 19948
19958 PRINT #3;"END OF VARIABLE NAME AND VALUE TABLES.":
      PRINT #3;"NUMBER OF VARIABLES FOUND=";PEEK(2Ø3)
19959 PRINT #3;"STRING/ARRAY TABLE LENGTH= ";
      ((PEEK(142)+256*PEEK(143))-(PEEK(14Ø)+256*PEEK(141)));
      " BYTES"
```

```
1996Ø POKE 752,Ø:
      CLOSE #3:
      END
19961 POKE 752,Ø:
      PRINT #3:
      PRINT #3;"ERROR! ":
      END
19962 PRINT #3
19963 PRINT #3;"SCALAR VARIABLE":
      PRINT #3;"CURRENT VALUE   = ";
19964 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+2)=Ø THEN PRINT #3;"ZERO":
      PRINT #3:
      RETURN
19965 PRINT #3;INT(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+3)/16);
19966 PRINT #3;(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+3)-
      (INT(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+3 )/16))*16);".";
19967 POKE 2Ø9,4
19968 PRINT #3;INT(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+PEEK(2Ø9))/16);
19969 PRINT #3;(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+PEEK(2Ø9))-
      (INT(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+PEEK(2Ø9)/16))*16);
1997Ø IF PEEK(2Ø9)<7 THEN POKE 2Ø9,PEEK(2Ø9)+1:
      GOTO 19966
19971 PRINT #3;"*";:
      PRINT #3;((PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+2)-64)*1ØØ):
      PRINT #3:
      RETURN
1ØØ72 PRINT #3:
      PRINT #3;"ARRAY  ";
19973 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=64 THEN PRINT #3;"NOT DIMENSIONED ";:
      PRINT #3
19974 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=65 THEN PRINT #3;"DIMENSIONED";:
      PRINT #3
19975 PRINT #3;"FIRST  DIMENSION=";((PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+4)+
      256*PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+5))-1)
19976 PRINT #3;"SECOND DIMENSION= ";
      ((PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+6)+
      256*PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+7))-1)
19977 PRINT #3:
      RETURN
19978 PRINT #3:
      PRINT #3;"STRING  ";
19979 IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=128 THEN
      PRINT #3;"NOT DIMENSIONED";:
      PRINT #3
1998Ø IF PEEK(PEEK(2Ø6)+256*PEEK(2Ø7))=129 THEN
      PRINT #3;"DIMENSIONED";:
      PRINT #3
19981 PRINT #3;"MAXIMUM LENGTH  = ";(PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+6)+
      256*PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+7))
19982 PRINT #3;"CURRENT LENGTH  = ";
      (PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+4)+
      256*PEEK(PEEK(2Ø6)+256*PEEK(2Ø7)+5))
```

```
19983 PRINT #3:
      RETURN
19984 IF PEEK(2Ø4)=255 THEN POKE 2Ø4,Ø:
      POKE 2Ø5,PEEK(2Ø5)+1:
      RETURN
19985 POKE 2Ø4,PEEK(2Ø4)+1:
      RETURN
19986 IF PEEK(2Ø6)=255 THEN POKE 2Ø6,Ø:
      POKE 2Ø7,PEEK(2Ø7)+1:
      GOTO 1ØØ48
19987 POKE 2Ø6,PEEK(2Ø6)+1
19988 IF PEEK(2Ø9)=7 THEN POKE 2Ø9,Ø:
      RETURN
19989 POKE 2Ø9,PEEK(2Ø9)+1:
      GOTO 1ØØ46
```

The neat thing about this routine is that it contains no variable names to clutter the list of variable names in your program. The following are examples of the outputs you will get from VLIST.LST for the three types of variables:

```
VARIABLE NUMBER = 5
VARIABLE NAME   = DAYS
SCALAR VARIABLE
CURRENT VALUE   = 27.ØØØØØØØØ*Ø

VARIABLE NUMBER = 6
VARIABLE NAME   = MONTHS(
ARRAY  DIMENSIONED
FIRST  DIMENSION= 12
SECOND DIMENSION= Ø

VARIABLE NUMBER = 7
VARIABLE NAME   = NAME$
STRING DIMENSIONED
MAXIMUM LENGTH  = 19
CURRENT LENGTH  = 11

END OF VARIABLE NAME AND VALUE TABLES.
NUMBER OF VARIABLES FOUND= 7
STRING/ARRAY TABLE LENGTH= 1729 BYTES
```

Here is how you can use VLIST.LST:

1. LOAD the program that you want to analyze.

2. ENTER the list routine by using the command ENTER"D:VLIST.LST". Note that the routine must have previously been LISTed to disk with the file name VLIST.LST.

3. RUN your program to initialize the variables, then simply execute a GOSUB 19940.

4. A short menu will come up asking you to press the START botton for output to the screen or SELECT for output to a printer. If you select the printer, be sure to turn your printer ON before pressing SELECT.

5. If you modify your program, make sure that you delete the lister routine when you LIST the new version to disk.

If you don't have enough available memory to use VLIST.LST, you can use the following abbreviated version of it. VSHORT.LST does not have all the features of VLIST.LST, but it only takes up about 343 bytes. It will give you a list of all of your variables, thus telling you how many variables you have in your program. It does not perform the complete analysis that VLIST.LST does, but VSHORT.LST still can come in handy.

Here is a simple program that initializes some variables so we can see how VSHORT.LST works:

```
100 DIM MONTH$(15),COSTS(10):
    UNITS=100:
    BREAKAGE=0.1
```

Now if we merge VSHORT.LST and type RUN, here's what we get:

```
MONTHS$
COSTS(
UNITS
BREAKAGE
```

Notice that each name ends with an inverse video character.

Figure 4.5 — *VSHORT.LST – A 343 Byte Version of VLIST.LST*

```
19990 REM VSHORT.LST - A SHORT VLIST
19991 POKE 204,PEEK(130):
      POKE 205,PEEK(131)
19992 IF PEEK(204)=PEEK(132) AND PEEK(205)=PEEK(133)
      THEN STOP
19993 PRINT CHR$(PEEK(PEEK(204)+256*PEEK(205)));
19994 IF PEEK(PEEK(204)+256*PEEK(205))>127 THEN PRINT
19995 IF PEEK(204)=255 THEN POKE 204,0:
      POKE 205,PEEK(205)+1:
      GOTO 19992
19996 POKE 204,PEEK(204)+1:
      GOTO 19992
```

If you want to make your program unlistable, you can achieve that dubious goal by using the following routine:

```
20000 REM SCRAMBLE.LST
20001 FOR VTABLE=PEEK(130)+256*(131) TO PEEK(132)+256*PEEK(133)-1
20002 POKE VTABLE,99
20003 NEXT VTABLE
20004 POKE PEEK(138)+;256*PEEK(139)+2,0
20005 SAVE "D:FILENAME"
20006 NEW
```

This routine replaces all of the variable names in the table with garbage, and the POKE in line 20004 makes it so your program can only be RUN. It will not be listable at all and a LOAD command won't work correctly. I really do not recommend that you ever do this to one of your programs. The reason that I showed you this trick is so you will better understand what the problem is when some nefarious programmer gives you a program that has a garbled variable name table. If you run across such a program, you can use VERVAN'S FULMAP to make it listable again.

The Two Bit Shuffle, or Moving Data in Memory

Many special effects and high speed techniques involve nothing more than moving (or copying) a block of data from one location in memory to another. *Copying* a block of data means that the contents of certain memory locations are nondestructively duplicated in another location. *Moving* a block of data means that the original memory locations no longer contain the data. Think of this in terms of a photo-copy process. When you photo-copy a magazine article, the copy is made without destroying the original article (that is, if you don't use the photo-copy machine I have in my office). On the other hand, moving a block of data is analogous to moving a pile of leaves from the front of your house to the back of your house. When your mother (or wife) looks at the front yard, she congratulates you on doing such a fine job. However, you know that you did not really destroy the leaves. You simply *moved* them to a different storage location.

You can use special machine language subroutines to rapidly move or copy blocks of data from one location to another. In general, I will use the word "move" for both types of movement.

I have a little machine language routine that will do all of this by a simple call from BASIC. I call the routine MOVER. Figure 4.6 gives the assembly listing of MOVER:

Figure 4.6 — *Assembly Language Listing of MOVER – A Block Move (or Copy) Routine*

```
                ;MOVER - A BLOCK MEMORY MOVER
                ;
                ;CALLED FROM BASIC USING:
                ;X=USR(ADDR,START,END,NEWSTART,OPTION)
                ;
                ;CAUTION! - USE OPTION=Ø CAREFULLY
                ;
ØØØØ                    *=      $6ØØ        ;COMPLETELY RELOCATABLE
                ;
                ;SET UP PAGE ZERO POINTERS
                ;
ØØCB            FROMT   =       $CB         ;START ADDRESS OF OLD BLOCK
ØØCD            FROMB   =       $CD         ;END ADDRESS OF OLD BLOCK
ØØCF            TO      =       $CF         ;START ADDRESS OF NEW BLOCK
ØØD1            OPTION  =       $D1         ;Ø=MOVE  <>Ø=COPY
                ;
                ;INTIALIZE POINTERS
                ;
Ø6ØØ 68                 PLA                 ;GRAB NUMBER OF ARGUMENTS
Ø6Ø1 C9Ø4               CMP     #4
Ø6Ø3 FØØ7               BEQ     GOOD        ;IF ONLY 4 THEN CONTINUE
```

```
0605 AA     1210         TAX                     ;WRONG NUMBER OF ARGUMENTS
0606 68     1220 KILL    PLA                     ;RETRIEVE PROPER RTS ADDRESS
0607 68     1230         PLA
0608 CA     1240         DEX
0609 D0FB   1250         BNE    KILL
060B 60     1260 EXIT    RTS                     ;GO BACK TO BASIC
060C 68     1270 GOOD    PLA
060D 85CC   1280         STA    FROMT+1
060F 68     1290         PLA
0610 85CB   1300         STA    FROMT
0612 68     1310         PLA
0613 85CE   1320         STA    FROMB+1
0615 68     1330         PLA
0616 85CD   1340         STA    FROMB
0618 68     1350         PLA
0619 85D0   1360         STA    TO+1
061B 68     1370         PLA
061C 85CF   1380         STA    TO
061E 68     1390         PLA
061F 68     1400         PLA
0620 85D1   1410         STA    OPTION
            1420 ;
            1430 ;IS THIS A MOVE TO THE LEFT OR THE RIGHT?
            1440 ;
0622 A000   1450         LDY    #0               ;SET INDEX TO ZERO FOR LATER
0624 A5CC   1460         LDA    FROMT+1
0626 C5D0   1470         CMP    TO+1
0628 3030   1480         BMI    RIGHT
062A 1006   1490         BPL    LEFT
062C A5CB   1500         LDA    FROMT
062E C5CF   1510         CMP    TO
0630 3028   1520         BMI    RIGHT
            1530 ;
            1540 ;MOVE A BLOCK TO THE LEFT
            1550 ;
0632 B1CB   1560 LEFT    LDA    (FROMT),Y        ;GRAB A BYTE
0634 91CF   1570         STA    (TO),Y           ;MOVE IT LEFT
0636 A5D1   1580         LDA    OPTION           ;DO WE ERASE OLD LOCATION?
0638 C900   1590         CMP    #0
063A D003   1600         BNE    CHECK1           ;NO? THEN CONTINUE
063C 98     1610         TYA                     ;YES, ERASE OLD LOCATION
063D 91CB   1620         STA    (FROMT),Y
063F A5CC   1630 CHECK1  LDA    FROMT+1          ;ARE WE FINISHED?
0641 C5CE   1640         CMP    FROMB+1
0643 D006   1650         BNE    CHECK2
0645 A5CB   1660         LDA    FROMT
0647 C5CD   1670         CMP    FROMB
0649 F0C0   1680         BEQ    EXIT             ;YES? THEN RETURN TO BASIC
064B E6CB   1690 CHECK2  INC    FROMT            ;UPDATE READ/WRITE POINTERS
064D D002   1700         BNE    CHECK3
064F E6CC   1710         INC    FROMT+1
0651 E6CF   1720 CHECK3  INC    TO
```

```
Ø653 DØDD          BNE    LEFT
Ø655 E6DØ          INC    TO+1
Ø657 18            CLC
Ø658 9ØD8          BCC    LEFT
              ;
              ;MOVE A BLOCK TO THE RIGHT
              ;
Ø65A A5CD     RIGHT LDA   FROMB           ;COMPUTE BLOCK LENGTH
Ø65C 38            SEC
Ø65D E5CB          SBC    FROMT
Ø65F 48            PHA
Ø66Ø A5CE          LDA    FROMB+1
Ø662 E5CC          SBC    FROMT+1
Ø664 AA            TAX
Ø665 68            PLA
Ø666 18            CLC                    ;ADD IT TO NEW BLOCK START
Ø667 65CF          ADC    TO
Ø669 85CF          STA    TO
Ø66B 8A            TXA
Ø66C 65DØ          ADC    TO+1
Ø66E 85DØ          STA    TO+1
Ø67Ø B1CD     MOVE  LDA   (FROMB),Y       ;GRAB A BYTE
Ø672 91CF          STA    (TO),Y          ;MOVE IT RIGHT
Ø674 A5D1          LDA    OPTION          ;DO WE ERASE OLD LOCATION?
Ø676 C9ØØ          CMP    #Ø
Ø678 DØØ3          BNE    CHECK4          ;NO? THEN CONTINUE
Ø67A 98            TYA                    ;YES, ERASE OLD LOCATION
Ø67B 91CD          STA    (FROMB),Y
Ø67D A5CC     CHECK4 LDA  FROMT+1         ;ARE WE FINISHED?
Ø67F C5CE          CMP    FROMB+1
Ø681 DØØ6          BNE    CHECK5
Ø683 A5CB          LDA    FROMT
Ø685 C5CD          CMP    FROMB
Ø687 FØ82          BEQ    EXIT            ;YES? THEN RETURN TO BASIC
Ø689 C6CD     CHECK5 DEC  FROMB           ;UPDATE READ/WRITE POINTERS
Ø68B DØØ2          BNE    CHECK6
Ø68D C6CE          DEC    FROMB+1
Ø68F C6CF     CHECK6 DEC  TO
Ø691 DØDD          BNE    MOVE
Ø693 C6DØ          DEC    TO+1
Ø695 18            CLC
Ø696 9ØD8          BCC    MOVE
Ø698               .END
```

Figure 4.7 — *POKE Value Table for MOVER*

```
199ØØ REM MOVER.LST
199Ø1 DATA 1Ø4,2Ø1,4,24Ø,7,17Ø,1Ø4,1Ø4
199Ø2 DATA 2Ø2,2Ø8,251,96,1Ø4,133,2Ø4,1Ø4
```

```
19903 DATA 133,203,104,133,206,104,133,205
19904 DATA 104,133,208,104,133,207,104,104
19905 DATA 133,209,160,0,165,204,197,208
19906 DATA 48,48,16,6,165,203,197,207,
19907 DATA 48,40,177,203,145,207,165,209
19908 DATA 201,0,208,3,152,145,203,165
19909 DATA 204,197,206,208,6,165,203,197
19910 DATA 205,240,192,230,203,208,2,230
19911 DATA 204,230,207,208,221,230,208,24
19912 DATA 144,216,165,205,56,229,203,72
19913 DATA 165,206,229,204,170,104,24,101
19914 DATA 207,133,207,138,101,208,133,208
19915 DATA 177,205,145,207,165,209,201,0
19916 DATA 208,3,152,145,205,165,204,197
19917 DATA 206,208,6,165,203,197,205,240
19918 DATA 130,198,205,208,2,198,206,198
19919 DATA 207,208,221,198,208,24,144,216
19920 MLSTART=1536
19921 MLEND=1687
19922 FOR X=MLSTART TO MLEND
19923 READ Y:
      POKE X,Y:
      NEXT X
19924 RETURN
```

When you are shuffling blocks of numbers around in memory, you will have to be very careful not to crash your computer. Always save your program to disk and remove the disk from your disk drive before trying any new block move that you have not successfully done before. It is possible not only to crash the computer, but you could also very easily cause the operating system to destroy the files on your disk!

MOVER is called from BASIC by a USR call in the following format:

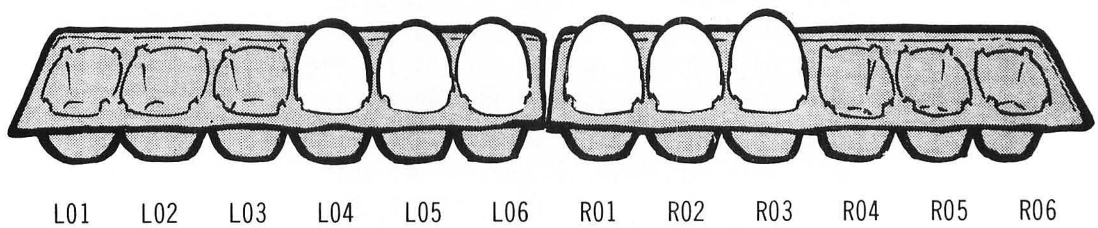
```
X = USR(1536,START,END,NEWSTART,OPTION)
```

START and END define the first and last memory addresses of the block that you want to move. NEWSTART is the address where MOVER is to start loading the data. The data will end up in the same order that it was in the original block. OPTION tells the routine whether you want to move or copy the block. A value of zero tells MOVER to move the block, thus deleting the block from its previous location. Any non-zero value in this argument will cause MOVER to copy the designated block of data. This means that you will then have two copies of the same block of data in two separate locations in memory.

There are two general kinds of block movements. If START and NEWSTART are separated by at least END-START+1 bytes, then this is called a *non-overlapping* movement. Think of it as laying two sheets of paper side-by-side in such a way that the two pieces of paper do not touch each other. It does not matter whether one piece of paper is to the right or the left of the other one. This type of movement is okay most of the time, but you will also find need for another kind of movement that allows the pieces of paper to overlap. The second kind of movement involves block locations that overlap one another. This kind of movement is further differentiated by the direction of the movement. A good example is to take an egg carton, cut it down the middle length-wise, and position the pieces end-to-end.

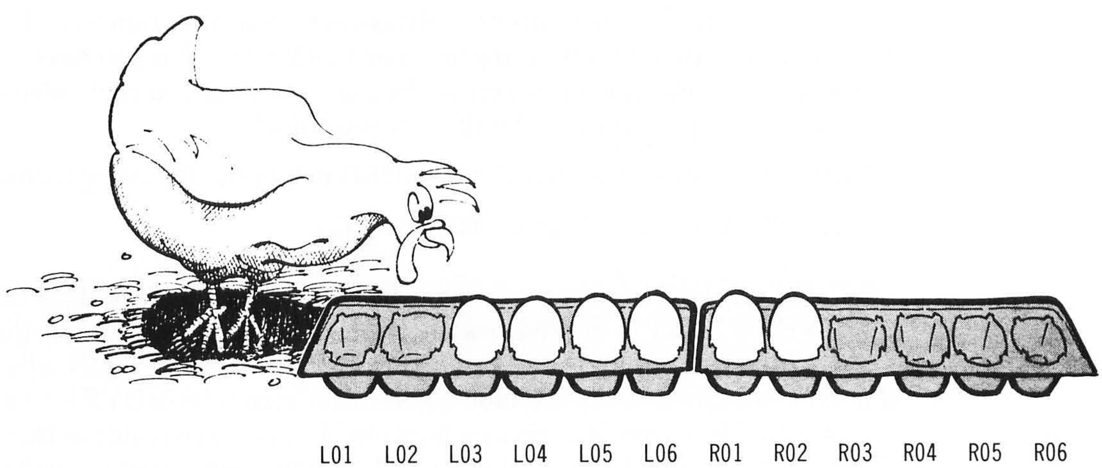
Put three eggs in the right side of the lefthand "half carton," and put three more eggs in the left side of the carton on the right. When you get it set up, it should look something like this:

Figure 4.8 — *Eggs in Carton*



Now, move all the eggs to the left one position so they look like this:

Figure 4.9 — *Eggs Moved to Left*



How did you move the eggs? I would wager that you did it the same way most other people would do it. You started by moving the egg in position L04 to position L03. Next you moved the egg in position L05 to position L04. You then continued in this fashion until all of the eggs were moved. This is exactly the same technique that a computer uses when moving a block of data in an overlapping move to the left (down, for you purists).

Now, reverse the experiment and shift the "block" of eggs right one position to restore the original configuration. What did you do differently? That's correct, you started the movement with the egg in position R02. You made the change in method almost instinctively. When a machine language subroutine moves a block of data in memory, it must use the technique that is correct for the kind of move it is doing.

The possible applications you might have for a routine such as MOVER would include:

1. Moving relocatable USR routines from one address in memory to another.
2. Instant duplication of array elements.
3. Clearing a section of memory.
4. Inserting and deleting array elements.
5. Moving data to protected memory so it can be passed to another program.
6. Insert and delete operations on the video display.
7. Saving the video display in protected memory for later recall. (There is also a technique called "page flipping" that we will discuss later in this book.)
8. Downloading the Atari character set.

The following program will demonstrate some of the uses of MOVER:

Figure 4.10 — *MOVER.DEM – Demonstration Program for MOVER*

```
100 REM MOVER.DEM
110 PRINT CHR$(125):
    POKE 752,1
120 POSITION 2,2:
    PRINT "**********"
130 X=USR(1536,40082,40091,40722,0)
140 FOR I=1 TO 100:
    NEXT I
150 X=USR(1536,40722,40731,40082,0)
160 FOR I=1 TO 100:
    NEXT I
170 GOTO 130
```

This program uses MOVER to simply move a string display from the top of the screen to the bottom. Try changing the fourth argument in both USR calls from zero to one. Note that the screen now appears to have two permanent copies of the string.

To move the original string to the right six places, use the following arguments in MOVER.DEM:

```
X = USR(1536,40082,40091,40088,0)
```

The routine WINDOW.DEM listed below, uses MOVER to show the contents of any page in your computer, one page at a time.

Figure 4.11 — *WINDOW.DEM – A Window Into Your Computer's Memory*

```
100 REM WINDOW.DEM - A WINDOW INTO
110 REM YOUR COMPUTER'S MEMORY
120 PRINT CHR$(125):
    PRINT "WINDOW.DEM - ":
    PRINT "A WINDOW INTO YOUR COMPUTER"
130 PRINT :
    PRINT "WHAT PAGE NUMBER (0 TO 255):";
140 TRAP 140:
    POSITION 30,4:
    PRINT CHR$(253);CHR$(254);CHR$(254);CHR$(254);:
    INPUT PAGE
150 IF PAGE<0 OR PAGE>255 THEN 140
160 X=USR(1536,256*PAGE,256*PAGE+255,40240,1)
170 GOTO 140
```

This routine shows you any page of memory, one at a time.

BASIC Overlays

Passing Variables Between Programs

Any time you use a RUN or LOAD command, all variables that were already active in your previous program are cleared to allow the newly LOADed program to start with a fresh slate. This is not always the result you would like to achieve. There are many applications where you will not want the variables cleared as you go from one program to another, or RUN a program again.

If you could pass variables between programs, you could divide a large applications program into several smaller *run modules*. By using smaller "programs," you will have more memory available for data storage. One program, for example, might load data from the keyboard or from a disk file. The next program might process the data, and a third program might take care of dumping the results to a printer or disk.

Before you can effectively use variable-passing subroutines, you need to have some understanding of how BASIC stores variables. Three areas are set aside in memory to store information about the variables in your program. We talked about the first in the last chapter. It is called the Variable Name Table. Everytime a new variable is used in a program *or in direct mode*, the name of that variable is added to the end of the Variable Name Table. The table will not allow you to have more than 128 variables. If you exceed this limit, you will get an ERROR 4, and your program will crash. The real utility of VLIST.BAS is that it gives you a count of how many variables are in your program. The length of this table depends upon the number of variables in your program. The second area that BASIC reserves for variable housekeeping is called the Variable Value Table. This is where BASIC stores the BCD value of each numeric variable in your program. I won't go into the specifics of exactly how the numbers are stored since that is well documented in De Re Atari, a book published by Atari. The piece of information that you need here is that this table is stored on top of the Variable Name Table below your program. The starting address of this table is stored on page zero at 134 and 135, and the length of this table depends upon the number of variables in your program.

The third reserved area is the one we need to really watch out for whenever we are doing simple overlays in BASIC. This is the String/Array Table. This table contains all of the string variables in your program, as well as the BCD values of all of your dimensioned numeric arrays. The key to understanding whether or not a variable is contained in this table is the DIMENSION statement. If the variable is dimensioned, it is in this table. BASIC requires all string variables to be dimensioned, so they are always included in this table. This table is of particular concern from the overlay point of view because it is enlarged during program execution. Everytime a new DIMENSION statement is encountered the table is expanded to make room for the new data. The amount of additional space that is eaten up is equal to the size of a string dimension, or six times the size of a numeric variable dimension. This table is located immediately below your BASIC program. You can find the start address of this table by looking at page zero, locations 140 and 141.

The Ultimate Memory Saver

Large computers use sophisticated techniques that automatically load small blocks of program logic from disk as they are needed. This makes it possible to execute programs that are, in effect, larger than the available memory. With the techniques I will describe here, you can do the same thing with your Atari 800! I am sure you will find, as I did, that when you implement these techniques, your programs will enter a whole new generation of performance capabilities.

We will call each group of BASIC program lines loaded with these techniques an overlay, and refer to the lines that remain in memory as our main (or master) program. Overlays can be loaded for limited operations that would normally be done by subroutines or for more global operations where the overlays are main programs in their own right. They can also be major blocks of program logic which act as sub-programs. Here are some of the advantages of using BASIC overlays:

1. You can, in effect, go from one program to another, retaining all variables that are in use. You can also leave your disk files open as you roll in overlays.

2. Common routines and subroutines can remain in memory as you go from one overlay to another. Because of this, you don't have to repeat your housekeeping logic in each program, and you don't need to repeat those subroutines that are standard to the overall application in each program. Because you can look at every application as a group of run modules, with little or no logic repeated, you save disk space. Since you load only what you need, when you need it, your effective load time may be faster.

3. Because your overlays share the same standard run modules and housekeeping logic, you save time when you need to make modifications. Let's say, for example, you want to change a disk file layout. Instead of changing it in several different programs, you only need to change it once if you have your disk handling subroutine in a run module.

4. Program execution speeds can improve because you have less text in memory at any one time. BASIC doesn't have to search as far when it receives a GOTO or GOSUB command.

5. An overlay program can GOTO or GOSUB to any line in the main program. The main program can execute GOTOs or GOSUBs to any line in the overlay program. One overlay program can even load another one.

6. You can make almost any large application run in as little as 8K of memory! Of course, you would not want to run that "tight" since performance would be seriously degraded by the continual loading of overlays from disk. In practice, however, the ability to reduce the memory space required for program text lets you have more space for string and variable storage and (if you need it) more space for protected memory at the bottom of memory.

We will be discussing two general methods of loading overlays. A *merged* overlay overwrites a section of code via an ENTER command while assuming that BASIC is holding the values of all of the variables that you want to pass to the new routine. A *protected memory* overlay utilizes a section of protected memory to make sure that the loading or running of a new routine or program won't damage the resident data.

Overlay Techniques In BASIC

Using the ENTER Command

There are two general methods for doing overlays in Atari BASIC. The first way is probably the easiest, but is also the least safe. You can merge two programs by using the ENTER command. You can do this either from the direct mode or during the execution of a program. When a program is stored on disk with the ENTER command, the resulting file is what is called an ASCII file (I guess ATASCII is what the form should be called on the Atari 800). When you ENTER this file from disk, it is treated like keyboard input! This holds true even if you are loading it during the execution of another program.

The trick to getting the new program to overlay part of your first program is to make sure that the line numbers of the overlay are *exactly the same* as the line numbers of the code to be replaced. For example, write the following three routines and LIST them to disk as three different files:

```
100 PRINT"MY PROGRAM #1"
110 PRINT
120 X=10:
    Y=25
130 Z=X*X+Y/X
140 PRINT X,Y,Z
```

```
110 PRINT"WITH A DIRECT OVERLAY"
130 Z=X*Y-X/Y
```

```
110 PRINT"WITH AN INTERLEAVED OVERLAY"
115 W=12
130 Z=X+Y+W
```

Now, type NEW and ENTER the first routine. When you RUN the first routine, the words "MY PROGRAM #1" will be printed on the screen along with the values of X, Y, and Z (10, 25, 102 5). When you use the ENTER command to load the second routine, the screen output of the next RUN will be:

```
MY PROGRAM #1
WITH A DIRECT OVERLAY
10      25      249.75
```

Why is this? Well, when you entered the second routine, lines 110 and 130 were replaced by the new lines in your overlay. If you did a LIST at this point, you would get

```
100 PRINT"MY PROGRAM #1"
110 PRINT"WITH A DIRECT OVERLAY"
120 120 X=10:
    Y=25
130 Z=X*Y-X/Y
140 PRINT X,Y,Z
```

Loading the overlay with the ENTER command had exactly the same effect as if you had typed the new lines in through the keyboard. Now ENTER the third routine from disk and RUN it. You got a different output because of the new program lines. Now LIST the program. You should pay special attention to the presence of line 115. This line did not exist in either the original routine or the first overlay. If you later perform another overlay, you will have to make doubly sure that the next overlay won't be messed up by line 115. This is why I emphasized the words *exactly the same* in the earlier discussion.

The merge technique of doing overlays may therefore be broken down into *direct* and *interleaved* methods. A direct overlay always replaces lines of code in the host program. It never introduces any new line numbers. This is what I would consider to be the safer of these two methods. Using interleaved overlays can lead to unforeseen trouble if more than one overlay is going to be used. You may accidentally put lines of code in the wrong place and wreak havoc with your data.

The same technique can be used to perform an overlay where you want to preserve the existing variable values for use by the overlayed run module. All that is required is one small change: you have to eliminate the need to use the RUN command after the new module is ENTERed. Unfortunately, the only way to do this is for you to know where you want the program to resume execution and for you to type the appropriate GOSUB command from direct mode. Needless to say, this is less than ideal, so I will show you a better way in a moment. So far I haven't told you why we need to worry about the String/Array Table. Remember that this table is expanded everytime a new array or string variable is dimensioned. As long as you are doing partial overlays, you probably won't run into a problem with the fact that this table is dynamically updated. However, I once had an application where I used one program to set up a massive amount of look-up tables that were to be used by a completely separate program which was to be loaded via the ENTER command. It bombed! No matter what I tried, I could not get it to work. I finally gave up and called Atari's question department to find out what I was doing wrong. They couldn't help me. They said that the ENTER technique was originally intended for *small* overlays of less than 8K or so. They had never tried to do an overlay of the size I was trying, and I was probably getting messed up by the way the String/Variable Table updates itself. Oh well. That told me that I probably should find another way to pack my 100K program into my little 48K computer. I found it. I could store my data in reserved memory by using POKEs, and get it out again by using PEEKs.

Using Protected Memory Overlays

The second of the general overlay techniques is more complex to set up, but it is much safer and can also be used in conjunction with the ENTER technique. This technique requires that you use some scheme like RESERVE.LST to protect a section of memory. You can then store your data in the protected area using POKEs. This way you can ENTER, LOAD a new program or even RUN"D:FILENAME", and the data will remain safe until you need it. I've seen this technique used in several graphics adventure games, such as Temple of Apshai, that are remarkably fast considering that they are written in BASIC. This technique has the added benefit of reducing the number of variables, which leaves more room for your program.

For example, type in the following routine and save it to disk:

```
100 PROLAY.DEM
110 SIZE=25
120 ADDRESS=256*PEEK(744)+PEEK(743)+SIZE
```

```
130 MM=INT(ADDRESS/256):
    LL=ADDRESS-256*MM
140 POKE 743,LL:
    POKE 744,MM
150 POKE 128,LL:
    POKE 129,MM
160 FOR X=1 TO 24:
    READ Y:
    POKE ADDRESS-X,Y:
    NEXT X
170 REM IF Y>255 OR Y<>INT(Y) THEN YOU WILL NEED
180 REM TO STORE IT IN TWO BYTES WHICH MEANS YOU
190 REM WOULD HAVE TO MODIFY THIS ROUTINE.
200 DATA 10,20,30,40,50,60,70,80
210 DATA 90,100,110,120,130,140,150,160
220 DATA 170,180,190,200,210,220,230,240
230 END
```

PROLAY.DEM first reserves 25 bytes at the bottom of memory. Then it reads a bunch of data that could just as easily be from a numeric array or several variables. Each byte of data is then POKEd into the reserved area of memory for later use by another program. The data is now safe from being destroyed by a RUN or LOAD command. You can now LOAD and RUN a new overlay module that will be able to use the data stored by the first routine. Typically, how this is done is to make the last line of the first routine automatically load the overlay with a RUN"D:filespec.extension" command. The overlay can process the data and end its function with a RUN"D:MAIN" command.

This technique can be used almost "as is" for data that consists of integers. If your data also includes non-integer values, you can still use this technique if you first multiply the data by 10 or 100 or 1000, that is, whatever power of ten that will get rid of the decimal fraction. If you have to do this, make sure that the overlay decodes the number properly. For example, I have a set of routines that use a 3000 element look-up table that contains numbers between zero and 35.5. All of the numbers have either a zero or a five to the right of the decimal place. I used the protected memory overlay technique by simply multiplying each number by two (I could have used 10, but decided against it to keep all of my data values below 255, which avoids two byte numbers), then I POKEd the "new" table into protected memory for use by the overlay routines. Of course, I had to make sure that the overlay divided the peeked value by two before using it.

If you want to store string data in this manner, you will have to write a routine that looks at each character in the string and computes the ATASCII value of the character. The resulting number can then be stored in your protected area. The follow on program will have to retrieve the numbers in the proper order, do a CHR$ operation, and concatenate them into the new string variable.

Protected memory overlays is a general technique that will handle large amounts of data. If you have less than 256 bytes of data to store, you could always store them on page six in memory.

Another application of this technique is to load special machine language subroutines during the execution of a main program. I haven't used the technique in this way very much, since I usually string pack my machine language subroutines and incorporate them directly into my main program. This only works, however, if the routine is relocatable. If the machine language routine that you need in your program is not relocatable, you can reserve a block of memory for it at the bottom of memory. This works only if that is where it is nonrelocatable to.

There are several different ways to load such a routine without halting program execution like the ENTER command does. If you analyze either CONVERT.BAS or DATAPAK.BAS, you will see where I am actually "loading" a machine language routine from disk during the middle of program execution. The only difference is that those two routines process the machine language code into some other format instead of simply POKEing the machine language program into the proper memory locations.

The simplest way to perform a "load" during program execution is to GOSUB to a subroutine that will open the disk file and loop through a series of GET statements which will grab the machine language routine from the disk file. You follow this with a short routine that takes the byte grabbed from the disk and POKEs it into the designated location in memory (usually page six). There are more sophisticated methods, but they are typically a variation on this one. For example, De Re Atari contains a program that performs the same kind of GET routine by POKEing certain parameters into the operating system and persuades the OS to load a machine language program from disk. HEADER.BAS in Chapter Four makes use of this technique.

Number Crunchers and Munchers

Regardless of the application, almost every program involves some addition, subtraction, multiplication or division. Whether you are computing a scientific formula, an accounting balance, or the number of points accumulated by each player in a game, you soon become accustomed to talking to your computer with numbers and equations. However, the problem presented by the application is only the beginning. Simple housekeeping chores, such as formatting the screen output or retrieving the desired information from an array or from a disk file, may often involve many numbers and equations.

This chapter provides many tricks and subroutines that can save you hours of programming time. We'll be looking at some mathematical techniques that are often required for everyday programs. In addition, we'll discuss ways to compress numerical data for more efficient disk and memory storage. You will also see some quick routines that will allow you to format numerical data. Finally, have you ever seen a computer book that didn't cover the subject of hexadecimal number conversions? We'll be discussing some efficient subroutines that can put this subject to bed, once and for all!

Finding Remainders

You will find that the remainder obtained when you divide one number by another has many applications in programming. A memory location, for example, can be broken down into byte sized pieces by dividing the decimal value of the memory location by 256. The integer of the result is the MSB and the remainder is the LSB. The specific equations are:

```
MSB = INT(ADDRESS/256)
LSB = ADDRESS-256*MSB
```

In disk applications, when we divide the sector number by the number of tracks on a disk, the integer of the result tells us which track the desired record is on, and the remainder tells us which sector within that track is the right one. In doing base conversions, we typically divide the original number by the new base repeatedly to obtain the remainder. BASIC provides no built-in command or function that will allow you to automatically fetch the remainder of a divide operation. You've got to use a simple formula (equation). The following subroutine, REMAIN.LST, computes the remainder, LEFTOVERS, of the first argument, NUMBER, divided by the second argument, DIVISOR:

```
20010 REM REMAIN.LST
20011 LEFTOVERS=NUMBER-DIVISOR*INT(NUMBER/DIVISOR)
20012 RETURN
```

Compare this equation to the way we computed MSB and LSB, and you will note that we used the same mathematical technique. When using this technique for general purpose applications, be sure that your program will not allow DIVISOR to ever be equal to zero. If that should ever occur, your program will be interrupted by an ERROR 11 (attempt to divide by zero error). This routine can be LISTed to a library disk and appended to other programs by using the ENTER command.

Rounding Numbers

Rounding a number is a mathematical technique that limits the number of digits in a number while trying to minimize the amount of error in the *rounded* number. You use this technique when you go to a store and compare the "per ounce" cost of two products. For example, if you are looking at two products that cost 87 cents for 9 ounces and 77 cents for 8 ounces, respectively, you may compute their relative per unit cost as 9.66666667 and 9.625 cents per ounce, respectively. However, this is not exactly what you really do. Typically, you will say that the first product costs *about* 9.66 cents per ounce and the second product costs *about* 9.63 cents per ounce. You conclude that the second product is a better bargain. What you have just done, almost automatically, is to *round* the awkward numbers to a format that is more managable. Of course, if you are like me, you had to use a pocket calculator in the process.

You will often find that you need to round numbers in application programming. We will discuss two rounding methods that are useful in various circumstances. The first of these is ROUNDINT.LST, which rounds a number to the nearest whole integer. If the decimal portion of the number is greater than or equal to 0.5, the number will be rounded UP to the next whole number for positive numbers, and DOWN to the next whole number if the number is negative. If the decimal portion is less than 0.5, then the decimal fraction will be truncated. This subroutine works with both positive and negative numbers.

```
20020 REM ROUNDINT.LST
20021 ROUNDINT = SGN(NUMBER)*ABS((INT(NUMBER)+INT(NUMBER-INT(NUMBER+0.5)))
20022 RETURN
```

The second technique, ROUNDDEC.LST, rounds a NUMBER to two decimal places for the proper handling of dollars and cents. The result will be the nearest cent, taking into account positive and negative numbers.

```
20030 REM ROUNDDEC.LST
20031 NUMBER = 100*NUMBER
20032 GOSUB  20020
20033 ROUNDDEC = ROUNDINT/100
20034 RETURN
```

In programming rounding functions, the challenge is to properly handle positive and negative numbers. You will be able to handle such problems with relative ease after you have experimented with ROUNDINT.LST and ROUNDDEC.LST.

Rounding Down

This subroutine, ROUNDDWN.LST, requires two arguments. It finds the first multiple of the first argument, LIMIT, that is less than or equal to the other argument, NUMBER. Let's say, for example, that we need to round a number down to the nearest 100. Calling ROUNDDWN.LST with NUMBER = 392 and LIMIT = 100 will return ROUNDDWN = 300. Setting NUMBER = 3100 and LIMIT = 100 will return ROUNDDWN = 3100.

```
20040 REM ROUNDDOWN.LST
20041 ROUNDDWN = LIMIT*INT(NUMBER/LIMIT)
20042 RETURN
```

If you want to find the corresponding left position on the video screen for a POSITION statement, you can use this routine. For example:

```
20050 REM ROW.LST
20051 ADDRESS = 40082:
      REM 40082 IS FOR DEMO ONLY
20052 NUMBER = ADDRESS-40000:
      LIMIT = 40:
      GOSUB (20040)
20054 ROW=LIMIT*INT(NUMBER/LIMIT)
20055 RETURN
```

will return a value of ROUNDDWN=2, thus telling you that the row number is two. Remember that the ATARI 400/800 normally indents the left side of the screen by two columns, so 40082 will actually point to the first PRINT position instead of the third. By the way, you can find the column for your POSITION by using the following routine:

```
20060 REM COLUMN.LST
20061 ADDRESS=40082:
      REM 40082 IS FOR DEMO ONLY
20062 NUMBER=ADDRESS-40000
20063 DIVISOR=40
20064 COLUMN=NUMBER-DIVISOR*INIT(NUMBER/DIVISOR)
20065 RETURN
```

Rounding Up

The ROUNDUP.LST suroutine is similar to the ROUNDDWN.LST routine, except that it finds the first multiple of LIMIT that is greater than NUMBER. For example, ROUNDUP.LST will return ROUNDUP=3100 for NUMBER=3022 and LIMIT=100. Changing NUMBER to 3100 will yield ROUNDUP=3200.

```
20070 REM ROUNDUP.LST
20071 ROUNDUP = LIMIT*INT(NUMBER/LIMIT)+LIMIT
20072 RETURN
```

Saving Space with One-Byte Numbers

If you know that a numeric field to be stored on disk (or in a program) will always contain an integer in the range 0 to 255, you can use the CHR$ and ASC functions to store and retrieve the data. The advantage is that your data will only be using one byte to store each number instead of six!

If you want to store an array which contains integers in the range 0 to 255, you can store each number in a string by converting each number to an equivalent ATASCII character. Call the number to be so stored, VALUE; then you can convert it into an ATASCII character by using PACK$(X,X) = CHR$(VALUE), where X is the position in the string, PACK$, that the character is to be stored in. To recall the number, simply use VALUE = ASC(PACK$(X,X)). I used this in one particular program that had a 3000 element array that was loaded via DATA statements. Between the array and the DATA statements, over 36K bytes would have been required. By packing the data into a string dimensioned to a length of 3000, I was able to reduce the 36K data base to about 3K! An unanticipated side effect was that my program ran faster because the size of the program had been reduced by almost 30K, and I no longer had to use overlays.

Saving Space with Two-Byte Numbers

Since the ATARI does not support true integers, all numeric values are stored as six-byte BCD. This can be a real pain if you have an application like the one mentioned in the previous paragraph. I have shown you a way to handle a special case by packing the six-byte numbers into one-byte strings. That technique isn't much help, however, if your data won't fit into the 0 to 255 range of integers. This next technique goes one step further. It assumes that you still have integers, but they can take on any positive value from 0 to 65535. All you have to do is combine several of the techniques that we have already discussed. Namely, you separate the number into an LSB and an MSB and pack these numbers into a string. You will have to be more careful in recovering the data to make sure you don't call an LSB an MSB and vice versa. Note that you could store prices in this way by multiplying the price by 100.

Print Without USING

Many BASICs have a built-in formatted print capability that is called PRINT USING. Unfortunately ATARI BASIC does not support this command. Although it is possible to write a machine language program to add such a command to BASIC, it probably isn't worth the time it would take, since you still would have to write the format statement that the PRINT USING command would use. I have found it simpler to write a set of special format subroutines that give me comparable capability in special cases.

Formatted Money Values

The MONEY.LST routine will take a number and force it into a dollar and cents format. The largest whole dollar must be less than $10 million. Fractional cents are rounded to the nearest penny. The dollar sign, "$", is placed at the immediate left of the number. The print field, however, does not vary. It is always 15 spaces wide. If you are formatting a column of prices and want the dollar sign to be printed in a certain column, then change the value of MONEY$ appropriately. DIGITS should be set for the largest dollar figure.

Figure 6.1 — *FORMATTED MONEY Subroutine*

```
20080 REM MONEY.LST
20081 TRAP 20083
20082 COLUMNS=DIGITS+3:
      DIM MONEY$(COLUMNS)
20083 IF LEN(STR$(ABS(INT(VALUE))))>=COLUMNS-5
      THEN PRINT "INTEGER VALUE OF NUMBER IS TOO LARGE":
      GOTO 20088
20084 TRAP 20088:
      MONEY$=" $            ":
      IF VALUE<0 THEN VALUE=-VALUE:
      MONEY$(1,1)="-"
```

```
20085 MONEY$(COLUMNS-2-LEN(STR$(INT(VALUE))),COLUMNS-3)=
      STR$(INT(VALUE))
20086 REM ROUND TO NEAREST PENNY
20087 MONEY$(COLUMNS-1,COLUMNS)=
      STR$(100+INT((VALUE-INT(VALUE))*100+0.5)):
      MONEY$(COLUMNS-2,COLUMNS-2)="."
20088 RETURN
```

Formatted Telephone Numbers

Another very useful format routine is PHONE.LST. The following routine requires the area code to be stored in AREA, and a telephone number which has been separated and stored in PREFIX and NUMBER. The routine will return a string, PHONE$, that contains the telephone number in the format (XXX) XXX-XXXX. This makes the number much easier to understand and use in printed listings.

Figure 6.2 — *FORMATTED TELEPHONE NUMBERS Subroutine*

```
20090 REM FORMATTED TELEPHONE NUMBERS
20091 REM USE LINE 20092 ONLY ONCE
20092 DIM PHONE$(14)
20093 PHONE$="(XXX) XXX-XXXX"
20094 PHONE$(2,4)=STR$(AREA):
      PHONE$(7,9)=STR$(PREFIX):
      PHONE$(11,14)=STR$(NUMBER):
      RETURN
20095 REM LINE 20096 IS FOR DEMO ONLY
20096 AREA=714:
      PREFIX=555:
      NUMBER=2121:
      PRINT AREA,PREFIX,NUMBER:
      GOSUB 20092:
      PRINT PHONE$:
      STOP
```

Base Conversions

Hexadecimal-to-Decimal Conversions

In many cases it is much more efficient to work with hexadecimal (hex) notation than with decimal. In fact, it is almost mandatory if you expect to do much machine language programming. To convert from hex to decimal is easy. You can use the following routine to convert any two or four place (in other words, one or two byte) hex number into a decimal number by storing your hex number in HEXNUMBER$ and using a GOSUB to HEXDEC.LST. The routine uses a common mathematical trick that will return the proper decimal number in DECNUMBER.

```
20100 REM HEXDEC.LST - CONVERT HEX NUMBERS TO DECIMAL
20101 DIM HEXDEC$(23),HEXNUMBER$(4)
20102 HEXDEC$ = "ABCDEFGHIJ*******KLMNOP":
20103 REM THIS IS THE MAIN ENTRY POINT
20104 DECNUMBER=0:HEX=LEN(HEXNUMBER$)
      FOR X = 1 TO HEX:
      DECNUMBER = 16*DECNUMBER +
      ASC(HEXDEC$(ASC(HEXNUMBER$(X)-47))-65:
      NEXT X:
      RETURN
```

This routine is particularly useful if you are writing a BASIC program that requires you to INPUT hex numbers. All you have to do is make the input variable a string and store the input in HEXNUMBER$ before calling the HEXDEC routine.

Decimal-to-Hexadecimal conversions

DECHEX.LST is very similar mathematically to HEXDEC.LST. The primary difference is the direction of the conversion. This routine will take a decimal number stored in the variable DECNUMBER, and convert it to a hex number, stored in HEXNUMBER$. The variable, BYTES, specifies the size of the hex number as either one or two bytes.

```
20110 REM DECHEX.LST - CONVERT DECIMAL NUMBERS TO HEX
20111 DIM DECHEX$(16)
20112 DECHEX$="0123456789ABCDEF":
20013 KHEX=4096: PRINT "$":IF BYTES=-1 THEN KHEX=16:2Z=2
20014 FOR I=1 TO Z4:
      J=INT(DECNUMBER/KHEX):PRINT DECHEX$(J+1,J+D);
20015 DECNUMBER=DECNUMBER-KHEX*J:
      KHEX=KHEX/16:
      NEXT I: PRINT:IF BYTES=1 THEN BYTES=2:Z4=4
20116 RETURN
```

The following program, HEADER.BAS, is a practical application program that uses the DECHEX.LST routine, albeit in a slightly modified form that reduces the repetition of the DIM and initial assignment statement. HEADER.BAS reads the file header on a disk file and tells you whether the file is for a BASIC program or a binary load machine language program. If the file is a binary load file, then HEADER.BAS will tell you certain important parameters for the file. Namely, you will be told the START and END addresses of where the file is loaded into memory. The length of the file will be displayed and you will be given the option of tracing the RUN and INIT addresses. If you chose to find out the RUN and INIT addresses, then HEADER.BAS will search the file for these parameters and display them for you. If the file turns out to be what is called a compound load file, you will be notified of this fact and given the option of continuing the trace operation. All addresses and lengths are printed in both decimal and hex format. If you press CTRL-R, HEADER will go into an auto-scan mode looking for an INIT or RUN address. The auto scan feature stops when one of these is found, the end of file is reached, or you press the space bar.

Figure 6.3 — *HEADER.BAS – Disk File Analyzer*

```
1000 REM HEADER.BAS-DOS FILE ANALYZER
1010 Z0=0:
     Z1=1:
     Z2=2:
     Z4=4:
     Z8=8:
     Z16=16
1020 IOCB=3:
     POKE 752,1
1030 DIM FILE$(Z16),
     RESPONSE$(Z16),
     DECHEX$(116),
     BLANK$(32),
     CIO$(31)
1040 FILE$="D1:":
     SEGMENT=Z0:
     DECNUMBER=Z0:
     FLAG1=Z0:
     FLAG2=Z0
1050 DECHEX$="0123456789ABCDEF"
1060 BLANK$(1)=" ":
     BLANK$(32)=" ":
     BLANK$(2)=BLANK$
1070 GOSUB 2560
1080 GOTO 1400
1090 REM COMMAND ROUTINE
1100 TRAP 1100
1110 POSITION Z2,18:
     PRINT "PRESS OPTION TO QUIT"
1120 PRINT "PRESS SELECT TO LOOK FOR RUN/INIT"
1130 PRINT "PRESS START  TO LOAD NEW FILE"
1140 IF PEEK(53279)=6
     THEN POKE 764,255:
     CLOSE #IOCB:
     RUN
1150 IF PEEK(53279)=5
     THEN 2070
1160 IF PEEK(53279)=3
     THEN 1210
1170 REM CTRL-R AUTO SCANS FOR INIT
1180 IF PEEK(764)=168
     THEN 2070
1190 POKE 77,0
1200 GOTO 1140
1210 POKE 752,Z0:
     POKE 764,255:
     CLOSE #IOCB
1220 END
```

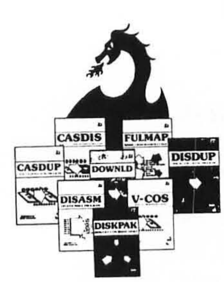
```
1230 REM DECIMAL-TO-HEX CONVERTER
1240 KHEX=4096:
     PRINT "$";
1250 IF BYTES=Z1
     THEN KHEX=Z16:
     Z4=Z2
1260 FOR I=Z1 TO Z4
1270 J=INT(DECNUMBER/KHEX)
1280 PRINT DECHEX$(J+Z1,J+Z1);
1290 DECNUMBER=DECNUMBER-KHEX*J
1300 KHEX=KHEX/Z16
1310 NEXT I:
     PRINT
1320 IF BYTES=Z1
     THEN BYTES=Z2:
     Z4=4
1330 RETURN
1340 IF SEGMENT
     THEN PRINT "SEGMENT":
     GOTO 1100
1350 PRINT "FILE":
     GOTO 1100
1360 REM CLEAR THE MESSAGE BOARD
1370 POSITION Z2,Z8
1380 PRINT BLANK$
1390 RETURN
1400 PRINT CHR$(125):
     POSITION Z2,Z2:
     PRINT "HEADER.BAS - DOS 2.0 FILE ANALYZER"
1410 POSITION Z2,6:
     PRINT "ENTER NAME OF FILE":
     GOTO 1460
1420 POSITION Z2,Z8
1430 PRINT "INPUT ERROR - TRY AGAIN";CHR$(253)
1440 FOR I=Z1 TO 200:
     NEXT I
1450 GOSUB 1370
1460 POSITION 21,6
1470 PRINT BLANK$(1,13)
1480 POSITION 21,6
1490 FILE$(Z4,Z16)=BLANK$(1,12)
1500 TRAP 1420
1510 CLOSE #IOCB
1520 INPUT RESPONSE$
1530 IF RESPONSE$<>"DOS.SYS"
     AND RESPONSE$<>"DUP.SYS"
     AND RESPONSE$<>"MEM.SAV"
     THEN 1600
1540 GOSUB 1370
1550 POSITION Z2,Z8
1560 PRINT "DO NOT USE DOS FILES";CHR$(253)
```

```
1570 FOR I=Z1 TO 200:
     NEXT I
1580 GOTO 1420
1590 REM FETCH FIRST TWO HEADER BYTES
1600 IF FLAG1=1
     THEN 1100
1610 FILE$(Z4,Z16)=RESPONSE$
1620 OPEN #IOCB,Z4,Z0,FILE$
1630 GET #IOCB,T:
     GET #IOCB,U
1640 IF T OR U
     THEN 1770
1650 GOSUB 1370
1660 POSITION Z2,Z8
1670 PRINT "THAT IS A BASIC FILE";CHR$(253)
1680 GOTO 1100
1690 POSITION Z2,10
1700 PRINT "FIRST  BYTE = ";
1710 DECNUMBER=T:
     BYTES=Z1
1720 GOSUB 1240
1730 PRINT "SECOND BYTE = ";
1740 DECNUMBER=U:
     BYTES=Z1
1750 GOSUB 1240
1760 RETURN
1770 GOSUB 1690
1780 REM FETCH NEXT FOUR HEADER BYTES
1790 GET #IOCB,V:
     GET #IOCB,W
1800 GET #IOCB,X:
     GET #IOCB,Y
1810 FLAG1=1
1820 REM COMPUTE START AND END
1830 MLSTART=V+256*W
1840 MLEND=X+256*Y
1850 SIZE=INT(MLEND-MLSTART)+Z1:
     IF SIZE>30000
     OR SIZE<0
     THEN 2660
1860 POSITION 21,12
1870 PRINT BLANK$(1,5)
1880 POSITION 21,13
1890 PRINT BLANK$(1,5)
1900 POSITION 21,14
1910 PRINT BLANK$(1,5)
1920 POSITION Z2,12
1930 PRINT "STARTING ADDRESS = ";MLSTART
1940 DECNUMBER=MLSTART
1950 POSITION 28,12
1960 GOSUB 1240
1970 PRINT "ENDING ADDRESS   = ";MLEND
```

```
1980 DECNUMBER=MLEND
1990 POSITION 28,13
2000 GOSUB 1240
2010 PRINT "LENGTH OF FILE   = ";SIZE
2020 DECNUMBER=SIZE
2030 POSITION 28,14
2040 GOSUB 1240
2050 GOTO 1100
2060 REM MOVE POINTER TO SEGMENT END
2070 BLOCK=SIZE:
     SUM=Z0:
     IF SIZE>30000
     OR SIZE<0
     THEN 2660
2080 IF SIZE<=Z0
     THEN 2140
2090 IF SIZE<=2
     AND (T=224 OR T=226)
     THEN SIZE=0:
     GOTO 2150
2100 IF BLOCK>125
     THEN BLOCK=BLOCK-125:
     GOTO 2100
2110 X=USR(ADR(CIO$),BLOCK)
2120 SUM=SUM+BLOCK:
     BLOCK=SIZE-SUM:
     IF BLOCK
     THEN 2100
2130 GOTO 2150
2140 GOSUB 1370:
     POSITION Z2,Z8:
     PRINT "END OF FILE REACHED":
     GOTO 1100
2150 TRAP 2140:
     GET #IOCB,T:
     GET #IOCB,U
2160 POSITION Z2,10:
     PRINT BLANK$
2170 POSITION Z2,11:
     PRINT BLANK$
2180 IF T=255
     AND U=255
     THEN GOSUB 1690:
     GET #IOCB,T:
     GET #IOCB,U
2190 IF T=224
     AND U=2
     THEN 2410
2200 IF T=226
     AND U=2
     THEN 2290
2210 REM COMPOUND LOAD FILE
```

```
2220 V=T:
     W=U
2230 GOSUB 1370:
     POSITION Z2,Z8:
     PRINT "THIS IS A COMPOUND LOAD FILE":
     SEGMENT=SEGMENT+Z1
2240 POSITION Z2,9:
     PRINT "THESE PARAMETERS ARE FOR SEGMENT #";SEGMENT+Z1
2250 POSITION Z2,15:
     PRINT BLANK$
2260 POSITION Z2,16:
     PRINT BLANK$
2270 GET #IOCB,X:
     GET #IOCB,Y:
     GOTO 1830
2280 REM SIMPLE LOAD FILE
2290 GET #IOCB,V:
     GET #IOCB,W
2300 IF V<>227
     OR W<>2
     THEN 1100
2310 GET #IOCB,INITL:
     GET #IOCB,INITH
2320 INIT=INITL+256*INITH
2330 POSITION Z2,15
2340 IF FLAG2
     THEN POSITION Z2,16
2350 PRINT "INIT   ADDRESS   = ";INIT
2360 DECNUMBER=INIT
2370 POSITION 28,15
2380 IF FLAG2
     THEN POSITION 28,16
2390 GOSUB 1240:
     POKE 764,255:
     GOSUB 2710
2400 SIZE=1:
     GOTO 1100
2410 GET #IOCB,V:
     GET #IOCB,W
2420 IF V=225
     AND W=2
     THEN FLAG2=0:
     GOTO 2450
2430 IF V=227
     AND W=2
     THEN FLAG2=1:
     GOTO 2450
2440 GOTO 1100
2450 GET #IOCB,LRUN:
     GET #IOCB,HRUN
2460 GOADDR=LRUN+256*HRUN
2470 POSITION Z2,15
```

```
2480 PRINT "RUN    ADDRESS   = ";GOADDR:
     DECNUMBER=GOADDR
2490 POSITION 28,15
2500 GOSUB 1240:
     POKE 764,255:
     GOSUB 2710
2510 IF FLAG2
     THEN 2310
2520 FLAG2=0:
     SIZE=1:
     GOTO 1100
2530 REM MACHINE LANGUAGE BYTE READER
2540 REM DATA FROM DISK IS NOT SAVED.
2550 REM EXECUTED ON FIRST RUN ONLY
2560 FOR X=Z1 TO 30
2570 READ Y
2580 CIO$(X,X)=CHR$(Y)
2590 NEXT X
2600 RETURN
2610 DATA 104,162,48,169,7,157,66,3
2620 DATA 169,0,157,68,3,169,224,157
2630 DATA 69,3,104,157,73,3,104,157
2640 DATA 72,3,32,86,228,96
2650 REM CHECK FOR END OF FILE
2660 TRAP 2680:
     GET #IOCB,ERROR
2670 PRINT CHR$(125):
     POSITION 2,10:
     PRINT "FATAL ERROR":
     GOTO 1210
2680 IF PEEK(195)=136
     THEN 2140
2690 GOTO 2670
2700 REM TONE
2710 SOUND 0,50,10,4
2720 FOR X=1 TO 50:
     NEXT X
2730 SOUND 0,0,0,0:
     RETURN
```



Using Strings

The string handling capabilities of BASIC provide countless opportunities to design powerful program utilities. This chapter will give you some ideas and some standard subroutines that will multiply the power of your programs.

PEEKs, POKEs, and Strings

There are three special string commands that are very useful. ADR(STRING$) will return the value of START. Thus you can look at and modify the contents of a string by PEEKing or POKEing directly at the memory locations that hold the contents of your string. Try the following example:

```
100 DIM DUMMY$(10)
110 DUMMY$="1234567890":
    PRINT DUMMY$
120 FOR X=0 TO 9 :
    POKE (ADR(DUMMY$)+X),65+X:
    NEXT X:
    PRINT DUMMY$
```

This example replaces the numbers in DUMMY$ with the alphabet letters A through J. You could use this technique to POKE a machine language subroutine into a string. However, the ADR command is used primarily to set the jump address of a USR function. For example, X=USR(ADR(STRING$)).

The ADR command has another interesting property. If you say X=USR(ADR"XYZ"), the value returned by ADR is the location in your program of "XYZ". More specifically, it is the address of that particular reference to "XYZ"! Generally, in this usage, the actual XYZ will be a small machine language routine.

The second string command of special interest is LEN(STRING$). This command returns the current length (LENGTH in Figure 7.1) of the string variable. This value is dynamically updated everytime you modify the string. The following program statements will show you what I mean:

```
100 DIM DUMMY$(10)
110 DUMMY$="ABCDEF":
    PRINT LEN(DUMMY$)
120 DUMMY$="ABC":
    PRINT LEN(DUMMY$)
```

The third special string command is CLR. This command UN-dimensions all of your string variables and makes the computer "forget" that you ever used them! In essence, CLR zeros out the Variable Value Table and sets the string array space to zero length. CLR has the same effect on any dimensioned variable or string. Scalar variables are all set to zero, but their names are left in the VNT; and although the names of your dimensioned variables and strings are left in the Variable Name Table, they must be dimensioned all over again if they are needed after a CLR.

Before we start manipulating strings, it is useful to know how BASIC stores and handles them. For each string variable in a program, BASIC maintains an eight-byte pointer in the Variable Value Table (VVT). The first byte will always be equal to 128 or 129 for a string variable since this is how BASIC distinguishes a string variable from some other variable type. If the string variable has not yet been dimensioned, the first byte will be set to 128. A value of 129 indicates that the string variable has been dimensioned. The second byte is the variable number (equal to its relative position in the VNT) which will range from 0 to 127. The third and fourth bytes contain the LSB and MSB of the *offset* of that particular string. The offset is the number of bytes from the beginning of the String Array Table Pointer (STARP) to the actual storage location of the contents of that string variable. Bytes five and six are the LSB and MSB of the *dimensioned* length of the string. Bytes seven and eight contain the LSB and MSB of the last location in the string that has information written in it. With these definitions, the following table is useful:

Figure 7.1 — *String Storage Pointers*

NAME	HOW TO FIND IT	WHAT IT MEANS
VNUMBER	Use VLIST.LST	Variable's ID number
VVTP	PEEK(134)+256*PEEK(135)	Start of VVT
STARP	PEEK(140)+256*PEEK(141)	Start of string storage area
REF	VVTP+(VNUMBER-1)*8	Reference to your string
TYPE	PEEK(REF+1)	128=not DIM; 129=DIMensioned
VNUM	PEEK(REF+2)	Same as VNUMBER
OFFSET	PEEK(REF+3)+256*PEEK(REF+4)	Value to add to STARP
MAXSIZE	PEEK(REF+5)+256*PEEK(REF+6)	DIMensioned length of string
LAST	PEEK(REF+7)+256*PEEK(REF+8)	Last used element of string
START	STARP+OFFSET	Where string contents start (same as result of ADR)
LENGTH	LAST-START	Actual length of your string (same as result of LEN)

The equations in this table were used in VLIST.LST to analyze string variables. The only easy way to obtain the value of VNUMBER is to use VLIST.LST. Why Atari chose to repeat this value in the VVT is a mystery to all of us.

It is important to note that BASIC does not move a string to the string array table unless it is used as a variable. For example, if line 100 of your program says:

```
100 DIM A$(10):A$="CAT":PRINT A$;" KILLS DOG"
```

... the string A$ is stored as discussed above. The string "KILLS DOG" is a literal string that is not stored in the string table. In fact, it is only "stored" in the position in memory where it occurs in line 100. So, though two strings were defined in line 100, only one of them was stored in the string storage area. Keeping this in mind, you can judge the ramifications of various methods of programming your application. Note that A$ uses a fixed amount of memory for overhead and a small amount for each reference to the string. The literal string, on the other hand, will use the same amount of memory everytime you use the string.

If we use a command that "lengthens" A$ during a BASIC program, the contents of the string array table are dynamically updated. The most obvious mistake made in these cases is to try to set A$ equal to something that is longer than the maximum dimensioned length of A$. The computer will barf if this happens and halt your program with an ERROR 5.

Blanking a String

If you need to pre-set a string to all blanks or some other character, you can use the following trick:

```
STRING$(1)=" ":STRING$(MAX)=" ":STRING$(2)=STRING$
```

Note that the "blank" between the quotes can be replaced by any other valid character. This trick works due to how BASIC performs a string equate. It literally does a sequential byte-by-byte transfer. Try the following experiment:

```
100 DIM STRING$(15):STRING$="123456789      "
110 STRING$(7,15)=STRING$(1,9):PRINT STRING$
```

The result that is printed isn't what you think it will be. . . .

Now that you have a better understanding of how BASIC handles and stores strings, we can discuss some special purpose subroutines for string handling. Each of these routines has been of use to me in one or more application programs, and I am sure that you will also find them to be indispensible time savers.

Stripping Trailing Blanks from a String

Here's a subroutine that you can use when you want to insure that there are no trailing blanks on a string. STRIPPER.LST returns the contents of WORD$ with any trailing blanks removed.

Figure 7.2 — *STRIPPER.LST*

```
20120 STRIPPER.LST
20121 REM WORD$ MUST BE PRESET
20122 REM BY THE CALLING PROGRAM
20123 FOR X=LEN(WORD$) TO 2 STEP -1
20124 IF WORD$(X,X)<>" " THEN POP :
      GOTO 20126
20125 NEXT X:RETURN
20126 WORD$=WORD$(1,X):RETURN
```

The only restrictions are that the calling program must have previously dimensioned WORD$ and that the string that you want stripped must be stored in WORD$. Also, you should be careful to make sure that the only thing stored in WORD$ is your string. This is easily done by presetting WORD$ to all blanks using the method we just discussed:

```
WORD$(1)=" ":WORD$(MAX)=" ":WORD$(2)=WORD$
```

In this case MAX is the dimensioned length of WORD$.

Justifying and Centering Strings

The RIGHT.LST, LEFT.LST and CENTER.LST subroutines are very useful when you are working with variable length strings and you want to print them in special formats on the video display or line printer.

Right Justifying a String

RIGHT.LST pads enough spaces to the left of a string, WORD$, so that its current length will be COLUMNS and forces the original contents of WORD$ to be right justified. Any trailing blanks are automatically stripped before the contents are right justified. The primary restrictions are that WORD$ and a temporary string called TEMP$ must be dimensioned to the same length before the routine is called. Additionally, the length of the the final string must be preset in COLUMNS. X is a temporary variable only. This subroutine is handy when you want to print variable length strings in nice, neat columns on a line printer. RIGHT.LST makes all of the right hand edges line up.

Figure 7.3 — *RIGHT.LST*

```
20130 REM RIGHT.LST
20131 REM COLUMNS,WORD$,AND TEMP$ MUST
20132 REM BE PRESET BY CALLING PROGRAM
20133 TEMP$(1)=" ":
      TEMP$(COLUMNS)=" ":
      TEMP$(2)=TEMP$
20134 FOR X=LEN(WORD$) TO 2 STEP -1:
      IF WORD$(X,X)<>" " THEN POP :
      GOTO 20136
20135 NEXT X:
      RETURN
20136 WORD$=WORD$(1,X)
20137 TEMP$(COLUMNS+1-LEN(WORD$),COLUMNS)=WORD$:
      WORD$=TEMP$:
      RETURN
```

Here is an example of RIGHT.LST:

```
WORD$="CALIFORNIA     ":COLUMNS=15: GOSUB 20130
```

returns WORD$="   CALIFORNIA".

Left Justifying A String

LEFT.LST pads enough blanks to the right of a string to left justify it. The routine works very much like RIGHT.LST and the same restrictions apply to both routines.

Figure 7.4 — *LEFT.LST*

```
20140 REM LEFT.LST
20141 REM FIELDS AND WORD$ MUST BE
20142 REM PRESET BY THE CALLING PROGRAM
20143 FOR X=1 TO LEN(WORD$):
      IF WORD$(X,X)<>" " THEN POP :
      GOTO 20145
20144 NEXT X
20145 WORD$=WORD$(X,LEN(WORD$):
      RETURN
```

Here is an example of LEFT.LST:

```
WORD$="     CALIFORNIA":COLUMNS=15:GOSUB 20140
```
returns WORD$="CALIFORNIA   ".

If you don't want the trailing blanks left on the string, do another call to STRIPPER.LST.

Centering a String

CENTER.LST pads enough blanks before a string and after it to center the string. The same restrictions that applied to the previous two routines also apply to this routine.

Figure 7.5 — *CENTER.LST*

```
20150 REM CENTER.LST
20151 REM COLUMNS, WORD$, AND TEMP$ MUST
20152 REM BE PRESET BY THE CALLING PROGRAM
20153 TEMP$(1)=" ":
      TEMP$(COLUMNS)=" ":
      TEMP$(2)=TEMP$
20154 FOR X=LEN(WORD$) TO 2 STEP -1:
      IF WORD$(X,X)<>" " THEN POP :
      GOTO 20156
20155 NEXT X
20156 WORD$=WORD$(1,X):
      FOR X=1 TO LEN(WORD$):
      IF WORD$(X,X)<>" " THEN POP :
      GOTO 20158
20157 NEXT X
20158 WORD$=WORD$(X,LEN(WORD$)):
      X=INT((COLUMNS-LEN(WORD$))/2)+1
```

```
20159 TEMP$(X,LEN(WORD$)+X)=WORD$:
      WORD$=TEMP$:
      RETURN
```

Here are a couple of examples using CENTER.LST:

```
WORD$=" CALIFORNIA       ":COLUMNS=16:GOSUB 20150
```

returns WORD$="   CALIFORNIA   ".

```
WORD$="CALIFORNIA":COLUMNS=20:GOSUB 20150
```

returns WORD$="   CALIFORNIA   "

The Last Shall Be First and the First Shall Be Last

In mailing lists, payroll and many other applications, it is useful to store names on disk with the last name of a person preceding his first name. This storage method makes it easier to sort the name file and put it in alphabetical order by the last name of each person. The REVERSE.LST routine converts a string stored in "last,first" format to a string in "first last" format. The routine looks for a comma in a string and swaps the data on the left side of the comma with the data on the right of the comma. If a comma is not found in the string, the string is not modified.

Here are some examples:

```
WORD$="JONES,SALLY":GOSUB 20160
```

returns WORD$="SALLY JONES".

```
WORD$="JOHNSON,MR. & MRS. BILL":GOSUB 20160
```

returns WORD$="MR. & MRS. BILL JOHNSON".

```
WORD$="ABC SUPPLY COMPANY":GOSUB 20160
```

returns WORD$="ABC SUPPLY COMPANY".

```
WORD$="Strings,How To Sort":GOSUB 20160
```

returns WORD$="How To Sort Strings".

The only restriction with REVERSE.LST is that the strings WORD$, TEMP$ and TEMP1$ must be dimensioned in your main program before calling the subroutine. I usually dimension all three strings to a length of 40. This shouldn't be a problem since the routine automatically strips any trailing blanks before it reverses the string.

Figure 7.6 — *REVERSE.LST*

```
20160 REM REVERSE.LST
20161 REM WORD$, TEMP$, AND TEMP1$
20162 REM MUST BE PRESET BY CALLING PROGRAM
20163 TEMP$=" ":
      TEMP1$=" "
```

```
20164 FOR X=LEN(WORD$) TO 2 STEP -1:
      IF WORD$(X,X)<>" " THEN POP :
      GOTO 20166
20165 NEXT X
20166 WORD$=WORD$(1,X):
      FOR X=LEN(WORD$) TO 2 STEP -1:
      IF WORD$(X,X)="," THEN POP :
      GOTO 20168
20167 NEXT X
20168 TEMP1$=WORD$(1,X-1):
      TEMP$=WORD$(X+1,LEN(WORD$)):
      TEMP$(LEN(TEMP$)+1)=" "
20169 WORD$=TEMP$:
      WORD$(LEN(WORD$)+1)=TEMP1$:
      RETURN
```

If you want to modify REVERSE.LST so that it will use a delimiter other than a comma to separate the two substrings, then replace the quoted comma in line 20166 with the character that you want to use.

Peeling Words Off of a String

Here's a subroutine that you can use to process a list of words entered by the operator. The PEELOFF.LST subroutine gets, one by one, each word in a string of words separated by commas. Upon each call to this subroutine, WORD$ contains a list of words. Upon return, ORDER$ contains the next word. When all words have been accessed, a value of -1 will be returned in the variable X. For all other calls, this variable will contain the length of the word that is returned in ORDER$.

Figure 7.7 — *PEELOFF.LST*

```
20180 REM PEELOFF.LST
20181 REM ORDER$ AND WORD$ MUST BE
20182 REM PRESET BY CALLING PROGRAM
20183 IF X<0 THEN WORD$="":
      X=0:
      RETURN
20184 ORDER$=WORD$:
      FOR X=1 TO LEN(ORDER$)
20185 IF ORDER$(X,X)="," THEN POP :
      GOTO 20187
20186 NEXT X:
      X=-1:
      RETURN
20187 ORDER$=WORD$(1,X-1):
      WORD$=WORD$(X+1,LEN(WORD$)):
      X=LEN(ORDER$):
      RETURN
```

Here is an example of PEELOFF.LST:

Make three calls to PEELOFF.LST.
Start with WORD$="JOHNSON,PAT,ERIC".

The first GOSUB 20180 will return
ORDER$="JOHNSON" and WORD$="PAT,ERIC" and X=7.

The second GOSUB 20180 will return
ORDER$="PAT" and WORD$="ERIC" and X=3.

The third GOSUB 20180 will return
ORDER$="ERIC" and WORD$="ERIC" and X=-1.

Massaging an Unruly String

Some processes require a string to be in a special form. The two that I have encountered most often are "upper vs. lower case" and "positive vs. inverted characters." LOWTOCAP.LST takes care of the first case, and INVERT.LST handles the second.

Converting a Lower Case String to Upper Case

The subroutine LOWTOCAP.LST searches a string for lower case characters and converts them to upper case characters. The string to be scanned must be stored in WORD$ before your program calls this subroutine.

Figure 7.8 — *LOWTOCAP.LST*

```
20190 REM LOWTOCAP.LST
20191 REM WORD$ MUST BE PRESET
20192 REM BY THE CALLING PROGRAM
20193 FOR X=1 TO LEN(WORD$):
      IF ASC(WORD$(X,X))>96
      AND ASC(WORD$(X,X))<123
      THEN GOSUB 20195
20194 NEXT X:
      RETURN
20195 WORD$(X,X)=CHR$(ASC(WORD$(X,X))-32):
      RETURN
```

Here is an example of LOWTOCAP.LST:

Set WORD$="John Paul Jones".
GOSUB 20190 returns WORD$="JOHN PAUL JONES".

Inverting the Characters in a String

The second special case is converting all the inverted characters in a string to non-inverted characters. INVERT.LST is a subroutine that will convert all normal characters into inverted ones or vice versa. The string to be inverted must be stored in WORD$, and the flag variable INVERT must be set to 0, 1 or -1. If INVERT=0, then only inverse characters will be flipped. If INVERT=1, only normal characters will be flipped. If INVERT=-1, then all normal characters will become inverted, and all inverted characters will become normal.

Figure 7.9 — *INVERT.LST*

```
20200 REM INVERT.LST
20201 REM WORD$ AND INVERT MUST
20202 BE PRESET BY CALLING PROGRAM
20203 FOR X=1 TO LEN(WORD$):
      Y=ASC(WORD$(X,X)
20204 IF Y<32 OR Y>250 OR (Y>122 AND Y<160)
      THEN GOTO 20207
20205 IF (Y>31 AND Y<123) AND(INVERT=1 OR INVERT=-1)
       THEN WORD$(X,X)=CHR$(ASC(WORD$(X,X)+128):
      GOTO 20207
20206 IF (Y>159 AND Y<251) AND (INVERT=0 OR INVERT=-1)
      THEN WORD$(X,X)=CHR$(ASC(WORD$(X,X)-128)
20207 NEXT X
20208 RETURN
```

Here is a brief summary of the possible options:

INVERT=	FUNCTION PERFORMED
-1	All alphanumeric characters are flipped
0	Only inverse alphanumeric characters are flipped
1	Only normal alphanumeric characters are flipped

If you want to modify INVERT.LST to work on keyboard graphic characters as well, delete line 20204 and change the limits of the IF statement in line 20205 to (Y>0 AND Y<129) and line 20206 to (Y>128 AND Y<256).

Messing Around Inside a String

The second biggest deficiency of ATARI BASIC is the lack of true string arrays in the same sense that we can have numerical arrays. (The biggest deficiency is the lack of true integers.) The resulting problems are fortunately not insurmountable. The routines in this section will show you how to verify that a substring is in a string and also several ways to simulate real string arrays. The PEELOFF.LST routine was a first step in this direction.

Verifying That a Substring is Really There

VERIFY.LST is a subroutine that searches a string for the presence of a specific substring. The string to be searched must be stored in WORD$, and the substring you are searching for must be stored in CODE$. The variable X will return the location of the first character in the substring. If the substring is not found in the target string, then X will be set to -1.

Figure 7.10 — *VERIFY.LST*

```
20170 REM VERIFY.LST
20171 REM CODE$ AND WORD$ MUST BE
20172 REM PRESET BY THE CALLING PROGRAM
20173 FOR X=LEN(WORD$) TO 1 STEP -1
20174 IF WORD$(X+1-LEN(CODE$),X)=CODE$ THEN POP :
      X=X+1-LEN(CODE$):
      GOTO 20176
20175 NEXT X:
      X=-1
20176 CODE$=" ":
      RETURN
```

Performing a VERIFY in Machine Language

BASIC is OK for verifying short strings, but a long string can take many seconds to search if you are using BASIC. When you have a long string, I recommend that you use SEEKER, which is a machine language subroutine that will search WORD$, element-by-element, for the target string, CODE$. If CODE$ is found in WORD$, then the variable SEARCH will contain the element number in WORD$ where CODE$ occurs. If CODE$ is not found, then SEARCH will be set to zero. If you made a mistake in the USR call to SEEKER, a value of 40000 will be stored in SEARCH to let you know that an error was found. Figure 7.11 is an assembly listing of SEEKER. The POKE values are given in Figure 7.12. The assembly listing tells you how to call SEEKER from BASIC, so I won't repeat all of that information here.

Figure 7.11 — *Assembled Listing of SEEKER*

```
1000 ;SEEKER - STRING SEARCH SUBROUTINE
1010 ;
1020 ;CALLED FROM BASIC USING
1030 ;SEARCH=USR(AEXP0,AEXP1,AEXP2,AEXP3,AEXP4)
1040 ;WHERE  AEXP0 = ADR(SEEKER$)
1050 ;       AEXP1 = ADR(WORD$)
1060 ;       AEXP2 = INT(LEN(WORD$)/LEN(CODE$))
1070 ;       AEXP3 = ADR(CODE$)
1080 ;       AEXP4 = LEN(CODE$)
1090 ;
1100 ;UPON RETURN TO BASIC, THE VARIABLE 'SEARCH' WILL BE
1110 ;           0 = CODE$ NOT FOUND
1120 ;           X = ELEMENT NUMBER WHERE CODE$ WAS FOUND
1130 ;       40000 = ERROR DURING INPUT
1140 ;
1150            *=      $600            ;COMPLETELY RELOCATABLE
1160 ;
1170 ;SET UP ZERO PAGE POINTERS
1180 ;
```

```
ØØCB        119Ø AWORDL  =      $CB           ;START ADDRESS OF STRING
ØØCC        12ØØ AWORDH  =      $CC           ;ARRAY TO BE SEARCHED
ØØCD        121Ø TOTALL  =      $CD           ;NUMBER OF ELEMENTS IN
ØØCE        122Ø TOTALH  =      $CE           ;THE STRING ARRAY
ØØCF        123Ø ACODEL  =      $CF           ;START ADDRESS OF CODE$
ØØDØ        124Ø ACODEH  =      $DØ
ØØD1        125Ø LCODE   =      $D1           ;LENGTH OF CODE$
ØØD4        126Ø COUNTL  =      $D4           ;LOCATION OF CODE$ INSIDE
ØØD5        127Ø COUNTH  =      $D5           ;THE STRING ARRAY
            128Ø ;
            129Ø ;INPUT ERROR TRAP
            13ØØ ;
Ø6ØØ 68     131Ø         PLA                  ;GRAB NUMBER OF ARGUMENTS
Ø6Ø1 C9Ø4   132Ø         CMP    #4
Ø6Ø3 FØØ9   133Ø         BEQ    GOOD          ;IF ONLY 4, THEN CONTINUE
Ø6Ø5 AA     134Ø         TAX                  ;WRONG NUMBER? THEN
Ø6Ø6 68     135Ø KILL    PLA                  ;RETRIEVE PROPER RTS ADDRESS
Ø6Ø7 68     136Ø         PLA
Ø6Ø8 CA     137Ø         DEX
Ø6Ø9 DØFB   138Ø         BNE    KILL
Ø6ØB 18     139Ø         CLC
Ø6ØC 9Ø66   14ØØ         BCC    ERROR
            141Ø ;
            142Ø ;INITIALIZE POINTERS
            143Ø ;
Ø6ØE 68     144Ø GOOD    PLA
Ø6ØF 85CC   145Ø         STA    AWORDH
Ø611 68     146Ø         PLA
Ø612 85CB   147Ø         STA    AWORDL
Ø614 68     148Ø         PLA
Ø615 85CE   149Ø         STA    TOTALH
Ø617 68     15ØØ         PLA
Ø618 85CD   151Ø         STA    TOTALL
Ø61A 68     152Ø         PLA
Ø61B 85DØ   153Ø         STA    ACODEH
Ø61D 68     154Ø         PLA
Ø61E 85CF   155Ø         STA    ACODEL
            156Ø ;
            157Ø ;MAKE SURE  Ø < LEN(CODE$) < 256
            158Ø ;
Ø62Ø 68     159Ø         PLA
Ø621 AA     16ØØ         TAX
Ø622 68     161Ø         PLA
Ø623 C9ØØ   162Ø         CMP    #Ø
Ø625 FØ4D   163Ø         BEQ    ERROR
Ø627 85D1   164Ø         STA    LCODE
Ø629 8A     165Ø         TXA
Ø62A C9ØØ   166Ø         CMP    #Ø
Ø62C DØ46   167Ø         BNE    ERROR
            168Ø ;
            169Ø ;SEARCH LOOP
            17ØØ ;
```

```
062E A5CD   1710         LDA   TOTALL
0630 85D4   1720         STA   COUNTL
0632 A5CE   1730         LDA   TOTALH
0634 85D5   1740         STA   COUNTH
0636 A000   1750 MAIN    LDY   #0            ;SET INDEX TO FIRST OF ELEMENT
0638 B1CF   1760 LOOP1   LDA   (ACODEL),Y    ;COMPARE BYTE OF CODE TO
063A D1CB   1770         CMP   (AWORDL),Y    ;A BYTE OF THE ELEMENT
063C D01C   1780         BNE   LOOP2         ;NO MATCH? THEN NEXT ELEMENT
063E C8     1790         INY                 ;MATCH? THEN DO BYTE-BY-BYTE
063F C5D1   1800         CMP   LCODE         ;COMPARE TO REST OF ELEMENT
0641 D0F5   1810         BNE   LOOP1
0643 A5CD   1820         LDA   TOTALL        ;WE FOUND IT!!
0645 38     1830         SEC                 ;STORE ELEMENT NUMBER OF
0646 E5D4   1840         SBC   COUNTL        ;CODE$ IN VARIABLE 'SEARCH'
0648 85D4   1850         STA   COUNTL
064A A5CE   1860         LDA   TOTALH
064C E5D5   1870         SBC   COUNTH
064E 85D5   1880         STA   COUNTH
0650 18     1890         CLC
0651 E6D4   1900         INC   COUNTL
0653 D028   1910         BNE   EXIT
0655 E6D5   1920         INC   COUNTH
0657 18     1930         CLC
0658 9023   1940         BCC   EXIT
065A A5CB   1950 LOOP2   LDA   AWORDL        ;MOVE POINTER TO NEXT ELEMENT
065C 18     1960         CLC
065D 65D1   1970         ADC   LCODE
065F 85CB   1980         STA   AWORDL
0661 9002   1990         BCC   LOOP3
0663 E6CC   2000         INC   AWORDH
0665 A5D4   2010 LOOP3   LDA   COUNTL        ;HAVE WE REACHED THE END
0667 D006   2020         BNE   LOOP4         ;OF THE STRING ARRAY?
0669 A5D5   2030         LDA   COUNTH
066B F010   2040         BEQ   EXIT          ;YES? THEN CODE$ IS NOT HERE
066D C6D5   2050         DEC   COUNTH        ;NO? THEN CONTINUE
066F C6D4   2060 LOOP4   DEC   COUNTL
0671 18     2070         CLC
0672 90C2   2080         BCC   MAIN
0674 A900   2090 ERROR   LDA   #0            ;STORE ERROR CODE 40000
0676 85D4   2100         STA   COUNTL        ;IN THE VARIABLE 'SEARCH'
0678 A9A0   2110         LDA   #$A0
067A 85D5   2120         STA   COUNTH
067C EA     2130         NOP                 ;NEEDED FOR DATAPAK.BAS
067D 60     2140 EXIT    RTS                 ;RETURN TO BASIC
067E        2150         .END
```

Figure 7.12 — *BASIC POKE Version of SEEKER*

```
20240 REM SEEKER.LST
20241 DATA 104,201,4,240,9,170,104,104
20242 DATA 202,208,251,24,144,102,104,133
20243 DATA 204,104,133,203,104,133,206,104
20244 DATA 133,205,104,133,208,104,133,207
20245 DATA 104,170,104,201,0,240,77,133
20246 DATA 209,138,201,0,208,70,165,205
20247 DATA 133,212,165,206,133,213,160,0
20248 DATA 177,207,209,203,208,28,200,197
20249 DATA 209,208,245,165,205,56,229,212
20250 DATA 133,212,165,206,229,213,133,213
20251 DATA 24,230,212,208,40,230,213,24
20252 DATA 144,35,165,203,24,101,209,133
20253 DATA 203,144,2,230,204,165,212,208
20254 DATA 6,165,213,240,16,198,213,198
20255 DATA 212,24,144,194,169,0,133,212
20256 DATA 169,160,133,213,234,96
20257 MLSTART=1536
20258 MLEND=1661
20259 FOR X=MLSTART TO MLEND
20260 READ Y:POKE X,Y:NEXT X
20261 RETURN
```

Simulating Real String Arrays

The entity that you are used to calling a string array really isn't a real array. It is simply a string that has to be dimensioned. An *array* is a means of referring to a set of such strings. Typically, each element of such an array is of a uniform length to simplify retrieving the particular element that you need. You can have real arrays like those only indirectly. Your program must do all of the bookkeeping that is done automatically on most other computer systems or by Atari Microsoft BASIC. The following three subroutines will do most of that kind of work for you. LOOKUP1D.LST is a subroutine that will fetch a particular ELEMENT of a one dimensional string array where each element is of length SIZE. The element you are looking for will be returned in the string TEMP$.

Figure 7.13 — *LOOKUP1D.LST*

```
20210 REM LOOKUP1D.LST
20211 REM SIZE, ELEMENT, TEMP$, AND WORD$
20212 REM MUST BE PRESET BY THE CALLING PROGRAM
20213 START=SIZE*(ELEMENT-1)+1
20214 TEMP$=WORD$(START,SIZE*ELEMENT):
      RETURN
```

Here is an example of LOOKUP1D.LST:

Set SIZE=4, ELEMENT=3, and WORD$="GREGPAULERICCARL".
GOSUB 20210 returns TEMP$="ERIC".

LOOKUP2D.LST is similar to LOOKUP1D.LST except that it retrieves an element from a two dimensional string array. In this routine we have replaced the one dimensional ELEMENT with X and Y, which are the *coordinates* of the desired element in the two dimensional array. In addition to SIZE, WORD$ and TEMP$, this routine also needs to know the maximum value of X. We will call this variable XMAX.

Figure 7.14 — *LOOKUP2D.LST*

```
20220 REM LOOKUP2D.LST
20221 REM SIZE, XMAX, X, Y, TEMP$, AND WORD$
20222 REM MUST BE PRESET BY THE CALLING PROGRAM
20223 START=(X-1)*SIZE+1+(Y-1)*SIZE*XMAX:
      LAST=START+SIZE-1:
      TEMP$=WORD$(START,LAST):
      RETURN
```

Here is a graphic representation of a two dimensional string array that we will use LOOKUP2D.LST on:

Figure 7.15 *Graphic Representation of a 2-D String Array*

```
                  <------ COLUMNS (X's) ------>

                    1      2      3      4      5      6      7

             +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
             +       +       +       +       +       +       +       +
  ^     1    + JOHN1 + PAUL1 + ERIC1 + CARL1 + GREG1 + MARK1 + MIKE1 +
  |          +       +       +       +       +       +       +       +
  |          +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ROWS         +       +       +       +       +       +       +       +
(Y's)   2    + JOHN2 + PAUL2 + ERIC2 + CARL2 + GREG2 + MARK2 + MIKE2 +
  |          +       +       +       +       +       +       +       +
  |          +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  v          +       +       +       +       +       +       +       +
        3    + JOHN3 + PAUL3 + ERIC3 + CARL3 + GREG3 + MARK3 + MIKE3 +
             +       +       +       +       +       +       +       +
             +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

We will use this 7-by-3, two dimensional array in a couple of examples to illustrate LOOKUP2D.LST. In these examples, SIZE=5 and XMAX=7:

EXAMPLE 1 — Find the element (4,2).

```
X=4:Y=2:GOSUB 20220
```

returns TEMP$="CARL2".

EXAMPLE 2 — Find the element (7,3).

```
X=7:Y=3:GOSUB 20220
```

returns TEMP$="MIKE3".

Another situation occurs every now and then in which you know where the element is (or have found it by searching the string), and you need to translate this number into the appropriate X and Y coordinates. LOOKUPXY.LST performs that function. This routine requires you to supply SIZE, XMAX and START (the location of the first byte of the target element). You might use LOOKUP1D.LST or VERIFY.LST to find the proper value of START.

Figure 7.16 — *LOOKUPXY.LST*

```
20230 REM LOOKUPXY.LST
20231 REM SIZE, XMAX, AND START MUST BE
20232 REM PRESET BY THE CALLING PROGRAM
20233 Y=INT(START/(SIZE*XMAX))+1:
      X=INT((START-(Y-1)*XMAX*SIZE)/SIZE)+1:
      RETURN
```

Using the same 2-D array we just used, we can set START=51 and GOSUB 20230. The resulting X and Y are X=4 and Y=2.

Date and Time Manipulation

Sooner or later in your programming efforts, you are likely to work with date or time computations. Why be the millionth programmer to spend hours and hours re-inventing the old wheel? Here are some plug-in subroutines that can save you programming time while conserving valuable computer memory and disk space.

The Eight Byte Date

The "eight byte date" is simply a string that expresses the month, day and year in the format, **MM/DD/YY**, where:

```
MM is a two digit month number ranging from Ø1 to 12,
DD is a two digit day number ranging from Ø1 to 31, and
YY is a two digit year number ranging from ØØ to 99.
```

The string, "02/16/83" is an example of an eight byte date that stands for "February 16, 1983".

A Simple Date Validity Check

VALIDATE.LST is a subroutine that checks the validity of a date entered by the operator. VALIDATE.LST verifies for the date string, DATE$:

```
The month (in positions 1 and 2) is between Ø1 and 12.
The day (in positions 4 and 5) is between Ø1 and 31.
The year (in positions 7 and 8) is greater than or equal to QUERY$.
The string, DATE$, is eight characters long.
```

To use the VALIDATE.LST subroutine, you must first merge it with your program:

```
2Ø25Ø REM VALIDATE.LST
2Ø251 REM DATE$ MUST BE PRESET
2Ø252 MONTH=VAL(DATE$(1,2)):
      DAY=VAL(DATE$(4,5)):
      YEAR=VAL(DATE$(7,8))
2Ø253 VDATE=MONTH>Ø AND MONTH<13 AND
      DAY>Ø AND DAY<32 AND YEAR>=QUERY
```

```
20254 VDATE=(VDATE AND LEN(DATE$)=8)
      OR DATE$="00/00/00":
      RETURN
```

Here is an example of how you might use VALIDATE.LST in one of your own programs:

```
130 PRINT"ENTER DATE (MM/DD/YY)":
    INPUT DATE$
140 REM CHECK IF DATE IS VALID AND
    THE YEAR IS 1983 OR GREATER
150 GOSUB 20250
    IF NOT VDATE THEN PRINT"INVALID":
    GOTO 130
160 REM PROGRAM FALLS THROUGH TO HERE
    IF THE DATE IS VALID
```

A big advantage of the validate routine is that you can handle the validity test in one line of program logic. The subroutine returns a VDATE=1 for a valid date and a value of zero if the date is invalid. If you don't want to check on a minimum year, you can simply use zero as the value of QUERY.

Note that we are accepting 00/00/00 as a valid date. If you don't want to accept a zero date, then modify the subroutine by deleting the 'OR DATE$="00/00/00"' from line 20254.

The Three Byte Date

For disk and in-memory array storage, it is quite convenient to store dates in a three byte format. If the eight byte date is stored in DATE8$, a GOSUB to IIXTOII.LST will return the equivalent three byte date in DATE3$. We use a month-day-year so the three byte date can be sorted, and we can use "greater-than" and "less than" tests if necessary.

You will find the three byte approach is much more convenient than storing a date as three BCD scalar variables or as an eight byte string. Besides the advantage of using only three bytes instead of eight or more, the execution speed for conversions will normally be much faster.

Here are two subroutines that you can use when working with three byte dates. IIXTOIII.LST converts an eight byte date string, DATE8$, to a three byte data string, DATE3$. IIITOIIX.LST uncompresses a three byte string back to an eight byte string:

```
20260 REM IIXTOIII.LST
20261 REM DATE8$,DATE3$ MUST BE PRESET
20262 DATE3$(1,1)=CHR$(VAL(DATE8$(1,2))):
      DATE3$(2,2)=CHR$(VAL(DATE8$(4,5)))
20263 DATE3$(3,3)=CHR$(VAL(DATE8$(7,8))):
      RETURN
```

```
20270 REM IIITOIIX.LST
20271 REM DATE3$,DATE8$ MUST BE PRESET
20272 DATE8$(1,2)=STR$(ASC(DATE3$(1,1)):
      DATE8$(3,3)="/"
20273 DATE8$(4,5)=STR$(ASC(DATE3$(2,2)):
      DATE8$(6,6)="/"
20274 DATE8$(7,8)=STR$(ASC(DATE3$(3,3)):
      RETURN
```

Find a Day of the Year

Here is a subroutine that computes the day within any year from 1901 to 2099. You simply provide the four digit year, the month and the day of the month. FINDAY.LST takes into account whether or not a year is a leap year.

```
20280 REM FINDAY.LST
20281 REM MONTH,DAY,YEAR,& STRING$
20282 REM MUST BE PRESET
20283 STRING$="000303060811131619212426"
20284 NUMBER=28*(MONTH-1)+
      VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-1)+2))+DAY
20285 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN NUMBER=NUMBER+1
20286 RETURN
```

If you look carefully at this subroutine, you will see that the day number is computed first by figuring the number of preceding months multiplied by 28 days. Next a table is accessed based upon the number of days beyond 28 for all of the preceding months. Then, if the year is evenly divisible by four (leap year), and the month is greater than two, one day is added to account for 29 days in February. Finally, the day within the month is added.

After adding this subroutine to a program, we could, for instance, issue the following command:

```
MONTH=5:DAY=14:YEAR=1983:GOSUB 20280:PRINT NUMBER
```

. . . to find that MAY 14,1983 is the 134th day of the year.

Simplified Date Computing

To find the number of days between dates, the day of the week or the date that will be any number of days in the future, I have found that the best way is to convert each date to a number. Then, for example, the number of days between dates is a simple subtraction.

COMPDAY.LST is a subroutine that returns a single number which I call a "computational date." The computional day number, as provided by COMPDAY.LST, is useful for any date between the years 1901 and 2099. If you are curious about the reasons for limiting the valid range from 1901 to 2099 you can consult any good almanac. In brief, however, even numbered centuries, unless divisible by 400, are exceptions to the rule that leap years are divisible by four. Thus, 2000 is a leap year, while 1900 and 2100 are not.

Note that the computational dates we are discussing here are only useful for certain date computations. Because of changes in the calandar in past centuries, and leap year variations every century, they do not represent a number that is useful for any other purpose, such as astronomical calculations.

Here is the computational date subroutine. The inputs are the four digit year, a one or two digit month, and a one or two digit day of the month:

```
20290 REM COMPDAY.LST
20291 REM MONTH,DAY,YEAR,& STRING$
20292 REM MUST BE PRESET
20293 STRING$="000303060811131619212426"
20294 NUMBER=365*YEAR+INT((YEAR-1)/4)+28*(MONTH-1)+
      VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-1)+2))+DAY
20295 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN NUMBER=NUMBER+1
20296 RETURN
```

Days Between Dates

To find the number of days between two dates, merge the computational date subroutine, shown above, into your program. Then subtract the computational day number of the first date from the computational day number of the second date. For example, the number of days between November 8, 1982 and February 16, 1983 is 100. I'll show you a program a little later that will perform many such functions for you.

Day of the Week

This subroutine returns a nine byte string, DAY$, that contains the day of the week for any date between 1901 and 2099. WEEKDAY.LST assumes that the computational day has already been calculated.

```
20300 REM WEEKDAY.LST
20301 REM ASSUMES GOSUB TO 20290 (COMPDAY.LST) FIRST
20303 WEEK$="FRIDAY^^^SATURDAY^SUNDAY^^^MONDAY^^^
      TUESDAY^^WEDNESDAYTHURSDAY^"
20304 TEMP=9*(NUMBER-7*INT(NUMBER/7))+1:
      DAY$=WEEK$(TEMP,TEMP+8):
      RETURN
```

To find the day of the week for February 16, 1983, you can use the following commands:

```
MONTH=2:DAY=16:YEAR=1983:GOSUB 20290:
GOSUB 20300:PRINT DAY$
```

Back to Eight-Byte Dates

The computations to convert from a computational day number to an eight byte date are rather complex, but you will need them if you want to find out something like, what will the date be 200 days from now. To do it, we will use four new subroutines.

YEARCOM.LST recalls the year from a computational date. DAYCOM1.LST recalls the day number within the year for any computational date. MONTHCOM.LST recalls the month based on the day number within the year, NUMBER, and the year, YEAR. DAYCOM2.LST recalls the day of the month based on the YEAR, the MONTH and NUMBER.

To find the date 200 days into the future, we can use the following commands:

```
MONTH=1:DAY=15:YEAR=1983:GOSUB 20290:GOSUB 20310:
GOSUB 20320:GOSUB 20330:GOSUB 20340:
PRINT MONTH;"/";DAY;"/";YEAR
```

```
20310 REM YEARCOM.LST
20311 REM ASSUMES GOSUB TO 20290 (COMPDAY.LST) FIRST
20313 YEAR=INT((NUMBER-NUMBER-1461)/365):
      RETURN
```

```
20320 REM DAYCOM1.LST
20331 REM ASSUMES GOSUB TO 20310 (YEARCOM.LST) FIRST
20323 DAY=NUMBER-(365*YEAR+INT((YEAR-1)/4)):
      RETURN
```

```
20330 REM MONTHCOM.LST
20331 REM ASSUMES GOSUB TO 20320 (DAYCOM1.LST) FIRST
20333 X=0:
      IF YEAR/4-INT(YEAR/4) THEN X=1
20334 MONTH=1+(DAY>31)+(DAY>(59+X))+
      (DAY>(90+X))+(DAY>(120+X))+
      (DAY>(151+X))+(DAY>(181+X))+(DAY>212+X))
20335 MONTH=MONTH+(DAY>(243+X))+(DAY>(273+X))+
      (DAY>(304+X))+(DAY>(334+X)):
      RETURN
```

```
20340 REM DAYCOM2.LST
20341 REM ASSUMES GOSUB TO 20310 (MONTHCOM.LST) FIRST
20343 DAY=DAY-28*(MONTH-1)-VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-1)+2))
20344 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN DAY=DAY-1
20345 RETURN
```

Going Fiscal

It is necessary in some application programs to provide for a fiscal month and year that differs from the calendar month and year. The following subroutine computes the two digit fiscal year, FYEAR, the fiscal month, FMONTH, based on the calendar year, YEAR, and the calendar month, MONTH. The variable, FYSTART, specifies the first calendar month of the fiscal year.

Suppose that the fiscal year begins in October. The current calendar month is 12, and the current calendar year is 1982. You would load FYSTART with 10, MONTH with 12, and YEAR with 82. A GOSUB 20350 would yield FYSTART=83 and FMONTH=3.

```
20350 REM FISCAL.LST
20351 REM FYSTART,MONTH,& YEAR MUST BE PRESET
```

```
2Ø353 X=Ø:
      IF MONTH<1 OR MONTH>12 OR
      YEAR<Ø OR YEAR>99 THEN PRINT"INPUT ERROR":
      X=-1:
      RETURN
2Ø354 IF FYSTART=1 THEN FMONTH=MONTH:
      FYEAR=YEAR:
      RETURN
2Ø355 IF FYSTART<1 OR FYSTART>12 THEN PRINT"BAD START":
      X=-1:
      RETURN
2Ø356 IF MONTH>=FYSTART THEN FMONTH=MONTH+1-FYSTART:
      FYEAR=YEAR+1:
      RETURN
2Ø357 FMONTH=MONTH+13-FYSTART:
      FYEAR=YEAR:
      RETURN
```

1901 — 2099 Perpetual Calendar

The program, DATECOMP.BAS, will let you test the subroutines we have discussed. In addition, it will come in handy whenever you need to perform a date computation. To use the program, type it in as shown and then RUN it. DATECOMP.BAS will compute days between dates, a day of the week, a day within year, or the date X days hence. Note that the subroutines were slightly modified to minimize the size of the program.

Figure 8.1 — *DATECOMP.BAS*

```
1ØØØ REM DATECOMP.BAS
1ØØ1 GRAPHICS Ø:
     POKE 752,1:
     PRINT CHR$(125)
1ØØ2 POSITION 14,2:
     PRINT "DATE COMPUTER"
1ØØ3 POSITION 1Ø,4:
     PRINT "1 = DAYS BETWEEN DATES"
1ØØ4 POSITION 1Ø,5:
     PRINT "2 = DAY OF THE WEEK"
1ØØ5 POSITION 1Ø,6:
     PRINT "3 = DAY WITHIN THE YEAR"
1ØØ6 POSITION 1Ø,7:
     PRINT "4 = DATE, X DAYS HENCE"
1ØØ7 POSITION 1Ø,2Ø:
     PRINT "SELECT AN OPTION...."
1ØØ8 CLR :
     DIM STRING$(24),DATE$(8),DAY$(9),WEEK$(63),BLANK$(27)
1ØØ9 STRING$="ØØØ3Ø3Ø6Ø811131619212426"
1Ø1Ø WEEK$="FRIDAY^^^SATURDAY^SUNDAY^^^MONDAY^^^
     TUESDAY^^WEDNESDAYTHURSDAY^"
```

```
1011 DATE$(1)="":
     DATE$(8)="":
     DATE$(2)=DATE$:
     DAY$(1)="":
     DAY$(9)="":
     DAY$(2)=DAY$
1012 BLANK$(1)=" ":
     BLANK$(27)=" ":
     BLANK$(2)=BLANK$
1013 GOSUB 1040:
     IF X<49 OR X>52 THEN 1013
1014 GOSUB 1039:
     ON X-48 GOTO 1025,1028,1030,1032
1015 POSITION 10,11:
     PRINT "ENTER MONTH NUMBER : ";
1016 TRAP 1016:
     POSITION 31,11:
     INPUT MONTH
1017 IF MONTH<1 OR MONTH>12 THEN 1016
1018 POSITION 10,12:
     PRINT "ENTER DAY OF MONTH : ";
1019 TRAP 1019:
     POSITION 31,12:
     INPUT DAY
1020 IF DAY<1 OR DAY>31 THEN 1019
1021 POSITION 10,13:
     PRINT "ENTER 4-DIGIT YEAR : ";
1022 TRAP 1022:
     POSITION 31,13:
     INPUT YEAR
1023 IF YEAR<1901 OR YEAR>2099 THEN 1022
1024 RETURN
1025 POSITION 10,10:
     PRINT "ENTER FIRST DATE   : ":
     GOSUB 1015:
     GOSUB 1046:
     GOSUB 1037:
     X=DAY
1026 POSITION 10,10:
     PRINT "ENTER SECOND DATE  : ":
     GOSUB 1015
1027 GOSUB 1046:
     GOSUB 1037:
     DAY=ABS(X-DAY):
     POSITION 10,15:
     PRINT "DAYS BETWEEN DATES = ";DAY:
     GOSUB 1040:
     RUN
1028 POSITION 10,10:
     PRINT "ENTER DATE         : ":
     GOSUB 1015:
     GOSUB 1046
```

```
1Ø29 POSITION 1Ø,15:
     PRINT "DAY OF THE WEEK    : ";:
     GOSUB 1Ø49:
     PRINT DAY$:
     GOSUB 1Ø4Ø:
     RUN
1Ø3Ø POSITION 1Ø,1Ø:
     PRINT "ENTER DATE         : ":
     GOSUB 1Ø15
1Ø31 POSITION 1Ø,15:
     PRINT "DAY WITHIN THE YEAR: ";:
     GOSUB 1Ø43:
     PRINT DAY:
     GOSUB 1Ø4Ø:
     RUN
1Ø32 GOSUB 1Ø15:
     POSITION 1Ø,14:
     PRINT "ENTER DAYS HENCE   : "
1Ø33 TRAP 1Ø33:
     POSITION 31,14:
     INPUT Y
1Ø34 IF Y<Ø THEN 1Ø33
1Ø35 GOSUB 1Ø46:
     DAY=Y+DAY:
     GOSUB 1Ø5Ø:
     GOSUB 1Ø51:
     GOSUB 1Ø52:
     GOSUB 1Ø55
1Ø36 POSITION 1Ø,15:
     PRINT "DATE = ";MONTH;"/";DAY;"/";YEAR:
     GOSUB 1Ø4Ø:
     RUN
1Ø37 POSITION 1Ø,1Ø:
     PRINT BLANK$:
     POSITION 1Ø,11:
     PRINT BLANK$:
     POSITION 1Ø,12:
     PRINT BLANK$
1Ø38 POSITION 1Ø,13:
     PRINT BLANK$:
     POSITION 1Ø,14:
     PRINT BLANK$:
     POSITION 15,12:
     PRINT BLANK$
1Ø39 POSITION 1Ø,2Ø:
     PRINT BLANK$:
     RETURN
1Ø4Ø OPEN #3,4,Ø,"K:"
1Ø41 GET #3,X
1Ø42 CLOSE #3:
     RETURN
```

```
1043 DAY=28*(MONTH-1)+
     VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-N)+2))+DAY
1044 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN DAY=DAY+1
1045 RETURN
1046 DAY=365*YEAR+INT((YEAR-1)/4)+28*(MONTH-1)+
     VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-1)+2))+DAY
1047 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN DAY=DAY+1
1048 RETURN
1049 TEMP=9*(DAY-7*INT(DAY/7))+1:
     DAY$=WEEK$(TEMP,TEMP+8):
     RETURN
1050 YEAR=INT((DAY-DAY/1461)/365):
     RETURN
1051 DAY=DAY-(365*YEAR+INT((YEAR-1)/4)):
     RETURN
1052 X=0:
     IF YEAR/4-INT(YEAR/4) THEN X=1
1053 MONTH=1+(DAY>31)+(DAY>(59+X))+(DAY>(90+X))+
     (DAY>(120+X))+(DAY>(151+X))+(DAY>(181+X))+(DAY>(212+X))
1054 MONTH=MONTH+(DAY>(243+X))+(DAY>(273+X))+
     (DAY>(304+X))+(DAY>(334+X)):
     RETURN
1055 DAY=DAY-28*(MONTH-1)-
     VAL(STRING$(2*(MONTH-1)+1,2*(MONTH-1)+2))
1056 IF YEAR/4=INT(YEAR/4) AND MONTH>2 THEN DAY=DAY-1
1057 RETURN
```

Timing Benchmark Tests

A "benchmark" is simply a timed test of a program or routine. You can use the real time clock program, CLOCK, to compare the speed of alternative programming methods. You will have to use the BASIC clock loader program, CLOCK.BAS to set the clock up and initialize the time. Once this is done, the time will be displayed in the upper right hand corner of the screen regardless of what you are doing in BASIC. This clock is put on page six and protects itself despite the actions of DOS or even the SYSTEM RESET button. The clock has a "switch" that starts out ON, but can be turned OFF by POKEing any non-zero value into $600, such as POKE 1536,1. This is a must if you are reading the time from BASIC (since BASIC is so slow) for timing a benchmark. The time will be temporarily "frozen" and may be read by PEEKing 1537 for the hours, 1538 for the minutes, and 1539 for the seconds.

Figure 8.2 — *CLOCK Listing*

```
1000 ;CLOCK - A REAL TIME CLOCK
1010 ;
1020 ;
1030 ;THE INIT ROUTINE AT $400 IS EXECUTED ONLY ONCE.
1040 ;THE MAIN ROUTINE IS STORED ON PAGE SIX.
1050 ;
```

```
0000        1060        *=      $400         ;THIS IS LATER OVER-WRITTEN
            1070 ;
            1080 ;
            1090 ;SET UP OS POINTERS
            1100 ;
0002        1110 CASINI  =      $2           ;CASSETTE INIT VECTOR
0009        1120 BOOTF   =      $9           ;BOOT MODE FLAG
0042        1130 CRITIC  =      $42          ;CRITICAL OPERATION FLAG
0222        1140 VBLANK  =      $222         ;IMMEDIATE VBLANK VECTOR
0230        1150 SCREEN  =      $230         ;CONTAINS LOCATION OF SCREEN
159D        1160 DUP     =      $159D        ;OS FLAG TO DETECT DUP.SYS
E45C        1170 SETVBI  =      $E45C        ;SET-VBI VECTOR ENTRY
E45F        1180 SYSVBI  =      $E45F        ;OS VBLANK SERVICE ROUTINE
            1190 ;
            1200 ;SET UP PAGE ZERO POINTER
            1210 ;
00CC        1220 VIDEO   =      $CC          ;USED AND THEN RESTORED
            1230 ;
            1240 ;
            1250 ;SET UP PRIVATE INTERRUPT
            1260 ;
0400 A509   1270        LDA     BOOTF        ;IF A CASSETTE HAS BOOTED
0402 2902   1280        AND     #2           ;THEN SAVE CASINI FOR LATER
0404 F00A   1290        BEQ     INIT
0406 A602   1300        LDX     CASINI       ;RE-VECTOR CASSETTE INIT
0408 A403   1310        LDY     CASINI+1     ;TO INCLUDE OUR CLOCK
040A 8E0D06 1320        STX     DETOUR+1
040D 8C0E06 1330        STY     DETOUR+2
0410 A907   1340 INIT   LDA     #RESET&$FF
0412 8502   1350        STA     CASINI
0414 A906   1360        LDA     #RESET/256
0416 8503   1370        STA     CASINI+1
0418 AD2202 1380        LDA     VBLANK       ;DETOUR NORMAL HOUSEKEEPING
041B 8D1F06 1390        STA     EXIT+1
041E AD2302 1400        LDA     VBLANK+1
0421 8D2006 1410        STA     EXIT+2
0424 A509   1420        LDA     BOOTF
0426 0902   1430        ORA     #2
0428 8509   1440        STA     BOOTF
042A A206   1450        LDX     #MAIN/256    ;POINT VBLANK TO OUR CLOCK
042C A021   1460        LDY     #MAIN&$FF
042E A906   1470        LDA     #6
0430 205CE4 1480        JSR     SETVBI
0433 68     1490        PLA                  ;RETURN TO BASIC
0434 60     1500        RTS
            1510 ;
            1520 ;THIS IS THE PART WE WANT TO PRESERVE
            1530 ;
0435        1540        *=      $600         ;PROGRAM IS NOT RELOCATABLE
            1550 ;
            1560 ;
            1570 ;SAVE SPACE FOR ON/OFF SWITCH
            1580 ;
```

```
0600 00      1590 SWITCH  .BYTE0                    ;0=ON,ANY OTHER VALUE=OFF
             1600 ;
             1610 ;
             1620 ;SAVE SPACE FOR CLOCK TIME
             1630 ;
0601 00      1640 HOURS   .BYTE0
0602 00      1650 MIN     .BYTE0
0603 00      1660 SEC     .BYTE0
             1670 ;
             1680 ;
             1690 ;SAVE SPACE FOR COUNTERS
             1700 ;
0604 3C      1710 TICKS   .BYTE60                   ;1 SECOND=60 TICKS
0605 00      1720 CSEC    .BYTE0                    ;SECONDS COUNTER
0606 0D      1730 FUDGE   .BYTE13                   ;CORRECTION TO CSEC
             1740 ;
             1750 ;
             1760 ;KEEP THE WOLVES AT BAY
             1770 ;
0607 A9A9    1780 RESET   LDA     #$A9              ;SYSTEM RESET COMES HERE
0609 8D1306  1790         STA     PATCH
060C 201806  1800 DETOUR  JSR     NULL
060F A206    1810         LDX     #MAIN/256         ;TELL VBI WHERE CLOCK IS
0611 A021    1820         LDY     #MAIN&$FF
0613 A906    1830 PATCH   LDA     #6
0615 205CE4  1840         JSR     SETVBI            ;SET THE GEARS IN MOTION
0618 60      1850 NULL    RTS
0619 68      1860 THAW    PLA                       ;RESTORE COMPUTER REGISTERS
061A A8      1870         TAY
061B 68      1880         PLA
061C AA      1890         TAX
061D 68      1900         PLA
061E 4C1806  1910 EXIT    JMP     NULL              ;CHANGED DURING SETUP
0621 48      1920 MAIN    PHA                       ;SAVE CURRENT REGISTERS
0622 8A      1930         TXA
0623 48      1940         PHA
0624 98      1950         TYA
0625 48      1960         PHA
0626 AD9D15  1970         LDA     DUP               ;IF DUP.SYS IS IN COMPUTER
0629 C900    1980         CMP     #0                ;THEN PATCH IT SO IT
062B F00F    1990         BEQ     CLOCK             ;WILL NOT KILL CLOCK
062D A94C    2000         LDA     #$4C
062F 8D2A27  2010         STA     $272A
0632 A912    2020         LDA     #$12
0634 8D2B27  2030         STA     $272B
0637 A919    2040         LDA     #$19
0639 8D2C27  2050         STA     $272C
             2060 ;
             2070 ;CHRONOMETER ROUTINE
             2080 ;
063C CE0406  2090 CLOCK   DEC     TICKS
063F D0D8    2100         BNE     THAW
```

```
0641 EE0506 2110        INC   CSEC
0644 A23C   2120        LDX   #60
0646 8E0406 2130        STX   TICKS
0649 A542   2140        LDA   CRITIC       ;NEED THIS TO AVOID BAD I/O
064B 0D0006 2150        ORA   SWITCH       ;IF SWITCH IS OFF THEN
064E D0C9   2160        BNE   THAW         ;BYPASS CLOCK
0650 CE0606 2170 LOOP   DEC   FUDGE        ;KEEP CLOCK CALIBRATED
0653 D008   2180        BNE   DING
0655 A90D   2190        LDA   #13
0657 8D0606 2200        STA   FUDGE
065A CE0406 2210        DEC   TICKS
065D EE0306 2220 DING   INC   SEC          ;KEEP TRACK OF TIME
0660 EC0306 2230        CPX   SEC
0663 D01D   2240        BNE   DONG
0665 A000   2250        LDY   #0
0667 8C0306 2260        STY   SEC
066A EE0206 2270        INC   MIN
066D EC0206 2280        CPX   MIN
0670 D010   2290        BNE   DONG
0672 8C0206 2300        STY   MIN
0675 EE0106 2310        INC   HOURS
0678 A918   2320        LDA   #24
067A CD0106 2330        CMP   HOURS
067D D003   2340        BNE   DONG
067F 8C0106 2350        STY   HOURS
0682 CE0506 2360 DONG   DEC   CSEC
0685 D0C9   2370        BNE   LOOP
            2380 ;
            2390 ;VIDEO DISPLAY ROUTINE
            2400 ;
0687 A5CC   2410        LDA   VIDEO        ;SAVE CURRENT PAGE ZERO
0689 48     2420        PHA
068A A5CD   2430        LDA   VIDEO+1
068C 48     2440        PHA
068D 18     2450        CLC
068E AD3002 2460        LDA   SCREEN       ;FIND THE SCREEN DISPLAY AND
0691 6940   2470        ADC   #$40         ;POINT TO WHERE WE WANT
0693 85CC   2480        STA   VIDEO        ;TO WRITE THE TIME
0695 AD3102 2490        LDA   SCREEN+1
0698 6900   2500        ADC   #$0
069A 85CD   2510        STA   VIDEO+1
069C A000   2520        LDY   #0           ;WRITE THE TIME ON THE SCREEN
069E AD0106 2530        LDA   HOURS        ;HOURS ONE DIGIT AT A TIME
06A1 20D106 2540        JSR   DIVIDE
06A4 91CC   2550        STA   (VIDEO),Y
06A6 C8     2560        INY
06A7 8A     2570        TXA
06A8 91CC   2580        STA   (VIDEO),Y
06AA C8     2590        INY
06AB B1CC   2600        LDA   (VIDEO),Y    ;BLINK COLON ON AND OFF
06AD C99A   2610        CMP   #$9A
06AF F005   2620        BEQ   BLANK
```

```
06B1 A99A   263 OFF
06AD C99A   2610        CMP     #$9A
06AF F005   2620        BEQ     BLANK
06B1 A99A   26380
06B8 91CC   2660 COLON  STA     (VIDEO),Y
06BA C8     2670        INY
06BB AD0206 2680        LDA     MIN             ;MINUTES ONE DIGIT AT A TIME
06BE 20D106 2690        JSR     DIVIDE
06C1 91CC   2700        STA     (VIDEO),Y
06C3 C8     2710        INY
06C4 8A     2720        TXA
06C5 91CC   2730        STA     (VIDEO),Y
06C7 C8     2740        INY
06C8 68     2750        PLA                     ;RESTORE PAGE ZERO
06C9 85CD   2760        STA     VIDEO+1
06CB 68     2770        PLA
06CC 85CC   2780        STA     VIDEO
06CE 4C1906 2790        JMP     THAW
06D1 A200   2800 DIVIDE LDX     #0              ;SEPARATE 1'S FROM 10'S
06D3 38     2810        SEC                     ;AND CONVERT TO PROPER
06D4 E8     2820 LOOP2  INX                     ;DISPLAY CODE
06D5 E90A   2830        SBC     #$A
06D7 B0FB   2840        BCS     LOOP2
06D9 699A   2850        ADC     #$9A
06DB 8DE506 2860        STA     TEMP
06DE 8A     2870        TXA                     ;RETURNS  1'S DIGIT IN A
06DF 698E   2880        ADC     #$8E
06E1 AEE506 2890        LDX     TEMP            ;RETURNS 10'S DIGIT IN X
06E4 60     2900        RTS
06E5 00     2910 TEMP   .BYTE0
06E6        2920        .END
```

Figure 8.3 — *CLOCK.BAS Listing*

```
100 REM CLOCK.BAS - REAL TIME CLOCK
110 DIM A$(3):
    PRINT CHR$(125):
    POSITION 5,5:
    PRINT "CLOCK.BAS - A REAL TIME CLOCK"
120 TRAP 120:
    POSITION 2,10:
    PRINT "ENTER THE CORRECT TIME (HHMM)";:
    INPUT TIME
130 HOUR=INT(TIME/100):
    MINUTE=INT(TIME-HOUR*100):
    IF HOUR=0 AND MINUTE>=0 AND MINUTE<60 THEN 200
140 IF MINUTE<0 OR MINUTE>59 OR HOUR<1 OR HOUR>23 THEN 120
150 IF HOUR<>12 THEN 180
```

```
160 POSITION 2,12:
    PRINT "IS THIS NOON ";:
    INPUT A$:
    IF A$(1,1)<>"Y" THEN HOUR=0
170 GOTO 200
180 IF HOUR>12 THEN 200
190 TRAP 120:
    POSITION 2,12:
    PRINT "IS THIS AM OR PM";:
    INPUT A$:
    IF A$(1,1)="P" THEN HOUR=HOUR+12
200 GOSUB 290:
    POKE 1537,HOUR:
    POKE 1538,MINUTE:
    CLOCK=USR(1024)
210 END
220 DATA 165,9,41,2,240,10,166,2
230 DATA 164,3,142,13,6,140,14,6
240 DATA 169,7,133,2,169,6,133,3
250 DATA 173,34,2,141,31,6,173,35
260 DATA 2,141,32,6,165,9,9,2
270 DATA 133,9,162,6,160,33,169,6
280 DATA 32,92,228,104,96
290 MLSTART=1024:
    MLEND=1076
300 FOR X=MLSTART TO MLEND:
    READ Y:
    POKE X,Y:
    NEXT X
310 DATA 0,0,0,0,60,0,13,169
320 DATA 169,141,19,6,32,24,6,162
330 DATA 6,160,33,169,6,32,92,228
340 DATA 96,104,168,104,170,104,76,24
350 DATA 6,72,138,72,152,72,173,157
360 DATA 21,201,0,240,15,169,76,141
370 DATA 42,39,169,18,141,43,39,169
380 DATA 25,141,44,39,206,4,6,208
390 DATA 216,238,5,6,162,60,142,4
400 DATA 6,165,66,13,0,6,208,201
410 DATA 206,6,6,208,8,169,13,141
420 DATA 6,6,206,4,6,238,3,6
430 DATA 236,3,6,208,29,160,0,140
440 DATA 3,6,238,2,6,236,2,6
450 DATA 208,16,140,2,6,238,1,6
460 DATA 169,24,205,1,6,208,3,140
470 DATA 1,6,206,5,6,208,201,165
480 DATA 204,72,165,205,72,24,173,48
490 DATA 2,105,64,133,204,173,49,2
500 DATA 105,0,133,205,160,0,173,1
510 DATA 6,32,209,6,145,204,200,138
520 DATA 145,204,200,177,204,201,154,240
530 DATA 5,169,154,76,184,6,169,128
```

```
540 DATA 145,204,200,173,2,6,32,209
550 DATA 6,145,204,200,138,145,204,200
560 DATA 104,133,205,104,133,204,76,25
570 DATA 6,162,0,56,232,233,10,176
580 DATA 251,105,154,141,229,6,138,105
590 DATA 142,174,229,6,96,0
600 MLSTART=1536:
    MLEND=1765
610 FOR X=MLSTART TO MLEND:
    READ Y:
    POKE X,Y:
    NEXT X:
    RETURN
```

The Eight Byte Time

The eight byte time is simply a string that expresses the time in the format **HH:MM:SS**, where:

```
HH is a two digit hour number ranging from 01 to 12
(note that the range is 00 to 23 for military time),
MM is a two digit minute number ranging from 00 to 59, and
SS is a two digit second number ranging from 00 to 59.
```

The string "10:15:35" is an example of an eight byte time that stands for 15 minutes and 35 seconds after the hour of 10. The CLOCK machine language program uses a 24 hour military format. The clock starts at 00 (midnight), and 12 is added to the hours number for any time after noon. Thus 4 o'clock in the afternoon is shown as hour 16. The machine language program is divided into two parts. The first part is stored in the cassette bufer at $400. This part of the program needs to be executed only once, so it is stored in a region of memory that will be wiped clean the next time you do any cassette or disk I/O. All this routine does is to tell the SYSTEM RESET to not kill the other routine, which is our clock. In addition, it sets up what is called a special vertical blank interrupt so our clock will be updated every 1/60th of a second. See the *Atari Technical User Notes* for a detailed description of vertical blank interrupts.

The second part of the machine language program is the real meat of the clock. This part is stored on page six, so loading a BASIC program won't smash the clock. A better solution is to re-assemble the CLOCK to another block of memory that is protected via RESERVE.LST. Another necessary countermeasure is a harmless patch to DUP.SYS whenever that program is loaded. Without the patch, the clock will be killed when you go from the DOS menu back to BASIC. The patched DOS will work exactly like the original.

The clock "stops" for critical I/O operations and when the "switch" is turned OFF. Although the display is not updated during that freeze, the clock keeps track of how long it is OFF and corrects the display as soon as possible. Don't stop the clock for more than about nine minutes, or it may lose track of how long it has been OFF.

To time a benchmark test, design your test program so it stops the clock for an initial reading which you store. Then start the clock and your benchmark routine. When the routine finishes its task, stop the clock and calculate the difference between the first reading and this one. Typically a benchmark program will consist of a routine inside a FOR-NEXT that executes the routine for 1000 times. If you are using this technique, don't forget to have the resulting time divided by 1000 to get the benchmark time of your test routine.

Here are two subroutines that you will find useful when working with time quantities. HMSTOSEC.LST converts a time in the HH:MM:SS format to an equivalent number of seconds. The time needs to be stored in the string, HMS$, in an eight-byte format as indicated above. The resulting number of seconds is returned in the scalar variable, SECONDS. SECTOHMS.LST performs the inverse of this transform. It converts a number of seconds stored in SECONDS back to the eight byte HH:MM:SS format.

Once you have converted "hours, minutes, and seconds" to seconds, you can compute elapsed time by simply subtracting the two "seconds" quantities. If you wish to express the elapsed time in hours, minutes, and seconds, you can use SECTOHMS.LST to convert them back.

```
20360 REM HMSTOSEC.LST
20361 REM HMS$ MUST BE PRESET
20362 SECONDS=3600*VAL(HMS$(1,2)+
      60*VAL(HMS$(4,5)+VAL(HMS$(7,8):
      RETURN
```

```
20370 REM SECTOHMS.LST
20371 REM HMS$ MUST BE PRESET
20372 HMS$="00/00/00":
      X=INT(SECONDS/3600):
      HMS$(1,1)=STR$(INT(X/10)):
      HMS$(2,2)=STR$(X-10*INT(X/10))
20373 SECONDS=SECONDS-3600*INT(SECONDS/3600):
      X=INT(SECONDS/60):
      HMS$(4,4)=STR$(INT(X/10))
20374 HMS$(5,5)=STR$(X-10*INT(X/10)):
      SECONDS=SECONDS-60*INT(SECONDS/60):
      X=INT(SECONDS)
20375 HMS$(7,7)=STR$(INT(X/10)):
      HMS$(8,8)=STR$(X-10*INT(X/10)):RETURN
```

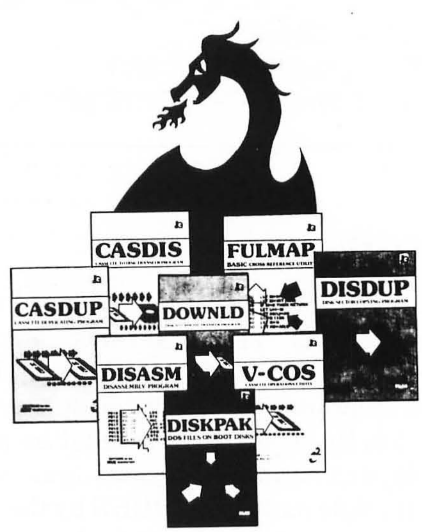
Time Clock Math

You will want to use this subroutine if you ever have to compute the elapsed time in hours and 10ths of an hour. The most obvious application is to compute the "amount of time worked" given the time you punched in and the time you punched out. CLOCKMATH.LST will do these computations when you supply a start time in CLOCKIN$ and a stop time in CLOCKOUT$. Both string times must be in HH:MM:SS format. CLOKMATH.LST will return the elapsed hours to the next lowest tenth of an hour in the scalar variable HOURS. You can then easily multiply HOURS by the appropriate pay rate to compute wages. Please note that the two times cannot differ by more than twelve hours, or the answer will not be correct.

```
20380 REM CLOKMATH.LST
20381 REM CLOCKIN$,& CLOCKOUT$
20382 REM MUST BE PRESET
20383 X=VAL(CLOCKIN$):
      Y=VAL(CLOCKOUT$):
      IF X<1 OR X>12 OR Y<1 OR Y>12 THEN X = -1:
      RETURN
20384 IF X>Y THEN Y=Y+12
20385 HOURS=Y-X:
      X=VAL(CLOCKIN$(4,5)):
      Y=VAL(CLOCKOUT$(4,5)):
      IF X<0 OR X>59 OR Y<0 OR Y>59 THEN X = -1:
      RETURN
20386 IF X>Y THEN Y=Y+60:
      HOURS=HOURS-1
20387 HOURS=HOURS+INT((Y-X)/6)/10:
      RETURN
```

Here is an example of how you might use CLOCKMATH.LST:

```
110 PRINT CHR$(125):
    PRINT:
    PRINT"TIME CLOCK SUBTRACTION TEST PROGRAM":
    PRINT
120 PRINT "ENTER START TIME";:
    INPUT CLOCKIN$
130 PRINT "ENTER STOP TIME";:
    INPUT CLOCKOUT$
140 GOSUB 20380:
    PRINT "ELAPSED TIME = ";HOURS;" HOURS"
```



Bits, Bytes, and Boole

A Bucket of Bits

Each byte of memory in your ATARI computer contains eight bits, giving a total of 393,216 bits in the memory of a 48K ATARI 800. Additionally, the 707 sectors on a diskette formatted by an Atari 810 disk drive give you another 707,000 usable bits on every diskette! Are you getting your money's worth?

In this chapter, we will look at ways to access and make use of the eight bits in a byte. We will discuss two machine language subroutines that will open up whole new avenues of possibilities for you to use in your programs.

Binary Numbers — Fundamental Building Blocks

The byte is the most common unit of measure in modern computer applications. A byte is usually described as one character of information, such as a letter ("A," "B," "C"), a single digit ("1," "2," "3") or a special character ("$," "?," "%"). In reality, a byte is any of 256 possible codes interpreted from the "ON/OFF" status of the eight bits in the byte.

A bit is the smallest unit of information storage in a computer. In fact, you could say that a bit is the only real unit of storage in your computer. The 6502 microprocessor does not have the capability of recognizing bytes any more than it can inherently handle disk I/O. The fundamental unit of information is the bit, which is either a one or a zero indicating the ON or OFF status of a specific electronic or magnetic location in memory or on a diskette.

In an eight-bit byte we can store any whole number from 0 to 255, or we can store the "YES/NO" status of eight different conditions. These "YES/NO" flags are sometimes referred to as *binary* numbers. This is just a special label that tells you that each number of that type (in other words, binary) can never have any value other than one or zero.

Working With Binary Numbers in BASIC

BASIC lets us create one-byte strings with the CHR$ function. CHR$(1), for example, generates a byte with the zero bit set and all other bits "cleared." CHR$(2) generates a byte in which bit one is set. CHR$(3) generates a byte in which bits zero and one are set. CHR$(65) generates a byte, which by ATASCII standards, represents the letter "A". For the letter "A", bit zero and six are set. "Set" means that the bit is equal to one, while "Clear" means that the bit is equal to zero.

One peculiarity of computer jargon that you will sooner or later have to get comfortable with is the screwy way counting is done. You always start counting with the number "zero" rather than "one." For example, the eight bits in a byte are numbered *from right-to-left* starting with zero:

Figure 9.1 — *Bits Within a Byte*

```
$ØF = 15 = Ø Ø Ø Ø 1 1 1 1
           ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕
BIT #    = 7 6 5 4 3 2 1 Ø
```

To convert the bits in a byte to a number, we look at each bit as a "power" of two and add them. For example, to represent the number three, bits zero and one are set. The three was obtained by adding two to the zeroth power, which is one, and two to the first power, which is two. The 65 was obtained by adding two to the zeroth power, which is one and two to the sixth power, which is 64. You will find it very useful to know the powers of two. They are:

Figure 9.2 — *The Powers of Two*

Power	Decimal	Hexadecimal	(MSB) Binary (LSB)
2^0	ØØØØ1	ØØØ1	ØØØØ/ØØØØ ØØØØ/ØØØ1
2^1	ØØØØ2	ØØØ2	ØØØØ/ØØØØ ØØØØ/ØØ1Ø
2^2	ØØØØ4	ØØØ4	ØØØØ/ØØØØ ØØØØ/Ø1ØØ
2^3	ØØØØ8	ØØØ8	ØØØØ/ØØØØ ØØØØ/1ØØØ
2^4	ØØØ16	ØØ1Ø	ØØØØ/ØØØØ ØØØ1/ØØØØ
2^5	ØØØ32	ØØ2Ø	ØØØØ/ØØØØ ØØ1Ø/ØØØØ
2^6	ØØØ64	ØØ4Ø	ØØØØ/ØØØØ Ø1ØØ/ØØØØ
2^7	ØØ128	ØØ8Ø	ØØØØ/ØØØØ 1ØØØ/ØØØØ
2^8	ØØ256	Ø1ØØ	ØØØØ/ØØØ1 ØØØØ/ØØØØ
2^9	ØØ512	Ø2ØØ	ØØØØ/ØØ1Ø ØØØØ/ØØØØ
2^{10}	Ø1Ø24	Ø4ØØ	ØØØØ/Ø1ØØ ØØØØ/ØØØØ
2^{11}	Ø2Ø48	Ø8ØØ	ØØØØ/1ØØØ ØØØØ/ØØØØ
2^{12}	Ø4Ø96	1ØØØ	ØØØ1/ØØØØ ØØØØ/ØØØØ
2^{13}	Ø8192	2ØØØ	ØØ1Ø/ØØØØ ØØØØ/ØØØØ
2^{14}	16384	4ØØØ	Ø1ØØ/ØØØØ ØØØØ/ØØØØ
2^{15}	32768	8ØØØ	1ØØØ/ØØØØ ØØØØ/ØØØØ

I have one bone I would like to pick with Atari BASIC. Unlike Microsoft BASIC, Atari BASIC does not allow us to directly access the bits in a number. The "logical" operators compare quantities only on the byte level, so we have no easy way to SET, CLEAR or TEST an individual bit in a byte.

Mapping Bits in Machine Language

The machine language subroutine, BITMAP, alleviates this deficiency by enabling us to SET, CLEAR or TEST any bit in a byte. BITMAP is called by a USR command of the general form:

```
RESULT = USR(ADDR,BYTE,BIT,OPTION)

 where: ADDR   = address of this machine language subroutine
        BYTE   = a number between 0 and 255
        BIT    = the target bit number inside BYTE
        OPTION = 0 means we wish to CLEAR that bit
               = 1 means the bit is to be SET
               = 2 will test the current value of the bit
```

The variable "RESULT" will contain the appropriate answer upon the return to BASIC.

Figure 9.3 — *BITMAP – Assembly Language Listing*

```
                1000 ;BITMAP - A BIT MANIPULATION ROUTINE
                1010 ;
                1020 ;
                1030 ;CALLED FROM BASIC USING:
                1040 ;X=USR(ADDR,BYTE,BIT,OPTION)
                1050 ;WHERE
                1060 ;        ADDR   = ADDRESS OF THIS ROUTINE
                1070 ;        BYTE   = ARGUMENT #1
                1080 ;        BIT    = ARGUMENT #2
                1090 ;        OPTION = 0 MEANS 'CLEAR THE BIT'
                1100 ;        OPTION = 1 MEANS 'SET THE BIT'
                1110 ;        OPTION = 2 MEANS 'TEST THE BIT'
                1120 ;
                1130 ;
0000            1140        *=     $600          ;COMPLETELY RELOCATABLE
                1150 ;
                1160 ;
                1170 ;SET UP PAGE ZERO POINTERS
                1180 ;
00CC            1190 BYTE   =      $CC
00CE            1200 BIT    =      $CE
00D0            1210 MASK   =      $D0
00D4            1220 RESULT =      $D4
                1230 ;
                1240 ;
                1250 ;INITIALIZE POINTERS
                1260 ;
0600 A900       1270        LDA    #0            ;SET RESULT TO ZERO
0602 85D4       1280        STA    RESULT
0604 85D5       1290        STA    RESULT+1
```

```
Ø6Ø6 68      13ØØ          PLA                  ;MAKE SURE THERE ARE NO
Ø6Ø7 C9Ø3    131Ø          CMP    #3            ;MORE THAN THREE ARGUMENTS
Ø6Ø9 FØØ7    132Ø          BEQ    GOOD
Ø6ØB AA      133Ø          TAX
Ø6ØC 68      134Ø KILL     PLA
Ø6ØD 68      135Ø          PLA
Ø6ØE CA      136Ø          DEX
Ø6ØF DØFB    137Ø          BNE    KILL
Ø611 6Ø      138Ø EXIT     RTS                  ;GO BACK TO BASIC
Ø612 68      139Ø GOOD     PLA
Ø613 68      14ØØ          PLA                  ;GET LSB OF BYTE
Ø614 85CC    141Ø          STA    BYTE          ;AND IGNORE THE MSB
Ø616 68      142Ø          PLA
Ø617 68      143Ø          PLA                  ;GET LSB OF BIT
Ø618 85CE    144Ø          STA    BIT           ;AND IGNORE THE MSB
             145Ø ;
             146Ø ;SET UP BIT MASK
             147Ø ;
Ø61A AA      148Ø          TAX                  ;SET SPECIFIED BIT IN MASK
Ø61B A9Ø1    149Ø          LDA    #1            ;ALL OTHER BITS ARE ZERO
Ø61D EØØØ    15ØØ          CPX    #Ø
Ø61F FØØ4    151Ø          BEQ    SETMASK
Ø621 ØA      152Ø IDMASK   ASL    A
Ø622 CA      153Ø          DEX
Ø623 DØFC    154Ø          BNE    IDMASK
Ø625 85DØ    155Ø SETMASK  STA    MASK
             156Ø ;
             157Ø ;SELECT WHICH OPTION
             158Ø ;
Ø627 68      159Ø          PLA
Ø628 68      16ØØ          PLA
Ø629 AA      161Ø          TAX
Ø62A EØØØ    162Ø          CPX    #Ø
Ø62C FØØ8    163Ø          BEQ    CLEAR         ;OPTION = Ø
Ø62E CA      164Ø          DEX
Ø62F FØ14    165Ø          BEQ    SET           ;OPTION = 1
Ø631 CA      166Ø          DEX
Ø632 FØ1A    167Ø          BEQ    TEST          ;OPTION = 2
             168Ø ;
Ø634 DØDB    169Ø          BNE    EXIT          ;OPTION = SOMETHING ELSE
             17ØØ ;
             171Ø ;BIT MAP ROUTINES
             172Ø ;
Ø636 A9ØF    173Ø CLEAR    LDA    #$ØF          ;CLEAR BIT IN A
Ø638 45DØ    174Ø          EOR    MASK
Ø63A 85DØ    175Ø          STA    MASK
Ø63C A5CC    176Ø          LDA    BYTE
Ø63E 25DØ    177Ø          AND    MASK
Ø64Ø 85D4    178Ø          STA    RESULT
Ø642 18      179Ø          CLC
Ø643 9ØCC    18ØØ          BCC    EXIT
```

```
0645 A5CC   1810 SET    LDA    BYTE          ;SET BIT IN A
0647 05D0   1820        ORA    MASK
0649 85D4   1830        STA    RESULT
064B 18     1840        CLC
064C 90C3   1850        BCC    EXIT
064E A5CC   1860 TEST   LDA    BYTE          ;TEST BIT IN A
0650 25D0   1870        AND    MASK
0652 F007   1880        BEQ    NEXT
0654 A901   1890        LDA    #1
0656 85D4   1900        STA    RESULT
0658 18     1910        CLC
0659 90B6   1920        BCC    EXIT
065B A900   1930 NEXT   LDA    #0
065D 85D4   1940        STA    RESULT
065F 18     1950        CLC
0660 90AF   1960        BCC    EXIT
0662        1970        .END
```

Figure 9.4 — *BITMAP.LST*

```
20390 REM BITMAP.LST
20391 DATA 169,0,133,212,133,213,104,201
20392 DATA 3,240,7,170,104,104,202,208
20393 DATA 251,96,104,104,133,204,104,104
20394 DATA 133,206,170,169,1,224,0,240
20395 DATA 4,10,202,208,252,133,208,104
20306 DATA 104,170,224,0,240,8,202,240
20397 DATA 20,202,240,26,208,219,169,15
20398 DATA 69,208,133,208,165,204,37,208
20399 DATA 133,212,24,144,204,165,204,5
20400 DATA 208,133,212,24,144,195,165,204
20401 DATA 37,208,240,7,169,1,133,212
20402 DATA 24,144,182,169,0,133,212,24
20403 DATA 144,175
20404 MLSTART=1536
20405 MLEND=1633
20406 FOR X=MLSTART TO MLEND
20407 READ Y:POKE X,Y:NEXT X
```

I recommend that you use DATAPAK.BAS to string pack BITMAP for your everyday use. That way the value of ADDR in the USR call will be ADDR=ADR(BITMAP$). All of the examples in this chapter will assume that you have done this first. If you decide to use the DATA statement version, then ADDR will be equal to the address where you store the first byte of data.

Clearing a Bit in a Byte

To CLEAR any bit in a byte, you start by selecting a particular byte that needs such a service. BITMAP will treat any number from 0 to 255 as a single byte. If you are trying to operate on a single byte of a string, you will have to define BYTE=ASC(STRING$(X,X)) before calling BITMAP. BIT is any number between zero and seven. If you try to use a BIT outside this range, BITMAP will not CLEAR any bits in BYTE. OPTION should be set to zero. The USR call to CLEAR the third bit in BYTE=7 is:

```
RESULT = USR(ADR(BITMAP$),7,2,Ø)
```

Note that the third bit is bit number two.

The result of this operation is stored in the variable RESULT. If you were to use the command PRINT RESULT after returning from BITMAP, you would get "3", which is what you should get when 00000111 is changed to 00000011.

Setting a Bit in a Byte

Setting any bit in a byte is done in a manner very similar to clearing a bit. In the previous example, keep everything the same except change OPTION=1 and BIT=3. The USR call to SET the fourth bit of BYTE=7 is:

```
RESULT = USR(ADR(BITMAP$),7,3,1)
```

This time 00000111 was changed to 00001111, so the number stored in RESULT should be (and is) RESULT=15.

Testing a Bit in a Byte

When we "test" a bit, we are checking to see whether it has been SET or not. We can test any bit in a byte by using OPTION=2 and calling BITMAP. A "true" test, meaning that the bit is SET, will return a value of RESULT=1. A "false" test, meaning that the bit is not set, will return a value of RESULT=0. We can use the answer stored in RESULT to perform an IF/THEN operation in our main program.

Let's test the third bit in BYTE=5. BIT will be equal to two and OPTION=2. The USR call is:

```
RESULT = USR(ADR(BITMAP$),5,2,2)
```

If we now PRINT RESULT, we will get a "1" indicating that the third bit is set. The binary representation of "5" is 00000101, so the answer is correct.

Don't worry about the previous contents of RESULT messing up the answer. BITMAP sets RESULT=0 before starting any CLEAR, SET or TEST operation.

A Practical Example of Bit Mapping

The ability to CLEAR, SET and TEST any bit in a byte lets us store eight "YES/NO" status indicators (or "flags") in a single byte. By setting up a string of such bytes, we can use this capability to obtain a marked savings in memory and disk storage requirements. For example, in a mailing list program we want to store as many names and addresses as possible. Additionally, we may need to store several other pieces of information about each of the people or companies that are in our list. By assigning only one more byte to each client's record, we can store eight additional information codes for each name, where each code is a YES/NO indicator. Say, for example, in our mailing list we want to keep track of which customer had been sent a copy of our latest catalog, and what other actions were taken in regards to that customer. The program was designed so that BIT=0 indicated that a new catalog was sent to that customer. BIT=1 was used to indicate whether or not the customer was a retail or wholesale customer. BIT=2 was used to indicate whether the customer's account was in good standing, and so forth. There were also bytes set aside to indicate which products the customer had or had not bought. This information was used for market analysis and to allow us to send special promotional literature to a customer on products that he had not yet bought.

Another application of bit mapping is an invoicing program I wrote where a byte associated with each of the products would indicate various stocking and inventory codes. If a bit is set, the condition applies to that particular product. For example:

Figure 9.5 — *An Example of Bit Mapping*

```
BIT     MEANING

Ø       This product is sold on disk
1       This product is sold on cassette
2       This product is for the ATARI 4ØØ/8ØØ/12ØØ
3       This product is for the TRS-8Ø Model I or III
4       This product is for the IBM PC
5       This product is for the APPLE II
6       This product is for the COMMODORE VIC-2Ø
7       This product is for the COMMODORE 64
```

Here is another idea I have used. When I am performing a series of operations on a set of records, I reserve one or two bytes of each record as an "update status" flag. Certain bits within each status flag are set after particular kinds of record updates and are not cleared until a special "clear status" routine is run. This has been especially useful since I seldom have the time to sit down and update every record in one sitting. Usually I get interrupted with one crisis or another that keeps me away for several hours or even several days. When I can finally get back to my chores, I can resume right where I left off by running a routine that tests the appropriate status flag of each record until it finds the first one that is CLEAR. This is really simple to do if you concatenate all status flags of a given type into one long string. I am sure that you will find many other ways to take advantage of bit mapping.

Boolean Operators — Logical Building Blocks

Boolean operators, which take their name from the famous mathematician George Boole, are a set of mathematical relationships that are used to perform logic operations. Such "logical" (or Boolean) operators enable a programmer to easily program logic functions into a program.

A Brief Tutorial on Boolean Logic

There are four fundamental Boolean operators: **OR**, **AND**, **NOT**, and the often confusing "exclusive OR", **XOR**. The exclusive OR is sometimes abreviated **EOR** instead of **XOR**. We will discuss each of these operators in more detail and show you some new ways to use them.

The Boolean OR Operator

The **OR** operator looks at the TRUE/FALSE condition of two arguments, BYTE1 and BYTE2. Each argument is either TRUE or FALSE. The **OR** operator returns a combined condition of TRUE anytime BYTE1 or BYTE2 is TRUE. The result is always FALSE if both BYTE1 and BYTE2 are FALSE.

Visualize this example: You and a friend are trying to decide which arcade game to play. If both of you do not like the game, then the two of you will look for another game to play. On the other hand, you are willing to play a game you do not like if your friend likes it and vice versa. Let's define all of the possible outcomes of your decision as a "truth table." It would look something like this:

Figure 9.6 — *Truth Table for a Boolean OR*

```
Your Opinion      Friend's Opinion     Result

Hate The Game     Hate The Game        Go Play Something Else
Hate The Game     Like The Game        Play This Game
Like The Game     Hate The Game        Play This Game
Like The Game     Like The Game        Play This Game
-------------------------------------------------------------
FALSE (Ø)         FALSE (Ø)            FALSE (Ø)
FALSE (Ø)         TRUE  (1)            TRUE  (1)
TRUE  (1)         FALSE (Ø)            TRUE  (1)
TRUE  (1)         TRUE  (1)            TRUE  (1)
-------------------------------------------------------------
```

The Boolean AND Operator

The **AND** operator looks at the TRUE/FALSE condition of two arguments and returns a combined condition of TRUE only if *both* of the arguments are TRUE. In our previous example, this would be like saying that you and your friend will play the arcade game only if both of you like the game. Otherwise, the two of you will look for another game to play. The truth table for the possible outcomes looks like this:

Figure 9.7 — *Truth Table for a Boolean AND*

```
You           Friend        Result

FALSE (Ø)     FALSE (Ø)     FALSE (Ø)
FALSE (Ø)     TRUE  (1)     FALSE (Ø)
TRUE  (1)     FALSE (Ø)     FALSE (Ø)
TRUE  (1)     TRUE  (1)     TRUE  (1)
```

The Boolean NOT Operator

The Boolean **NOT** operator is different from the first two we have talked about in that it operates on only one argument. It serves the function of changing a TRUE to a FALSE or a FALSE to a TRUE. The truth table for the **NOT** operator is:

Figure 9.8 — *Truth Table for a Boolean NOT*

You	Result
FALSE (Ø)	TRUE (1)
TRUE (1)	FALSE (Ø)

The Boolean XOR Operator

This poor operator is one of the most misunderstood mathematical operators that was ever invented. This operator looks at the TRUE/FALSE condition of two arguments and returns a TRUE only if the two arguments have *different* values. If both arguments are TRUE or both arguments are FALSE, the result of the **XOR** operation is FALSE. The truth table for this operator looks like this:

Figure 9.9 — *Truth Table for a Boolean XOR*

You	Friend	Result
FALSE (Ø)	FALSE (Ø)	FALSE (Ø)
FALSE (Ø)	TRUE (1)	TRUE (1)
TRUE (1)	FALSE (Ø)	TRUE (1)
TRUE (1)	TRUE (1)	FALSE (Ø)

Combining Boolean Operators

When the Boolean operators are combined with each other in the same logical expression, you have what is called "Boolean algebra." The expressions "A OR B" and "C AND D" are called "Boolean expressions." The expression "(A OR B) AND (C AND D)" is also a Boolean expression. There are many rules governing the relationships in Boolean algebra. I can't discuss all of them here in detail since that would take another couple of hundred pages. All I will do here is summarize the more frequently used rules and refer you to your local library or book store for more detailed information.

First, Boolean operators, like the arithmetic operators (+,-,*,/) obey the laws of precedence. With arithmetic operators, addition is done before subtraction, which is done before multiplication, which is done before division. The order of precedence for Boolean operators is:

Figure 9.10 — *Order of Precedence for Boolean Operators*

OPERATOR	PRECEDENCE	MEANING
( )	FIRST	Group contents of ( )
NOT	SECOND	Logical complement
AND	THIRD	Logical AND
OR	FOURTH	Logical OR
XOR	LAST	Logical exclusive OR

You will note that, like arithmetic expressions, the Boolean expressions inside parentheses take precedence.

When you studied math in school, you were probably taught that "1+2=2+1" and "3*(2+4)=(3*2)+(3*4)." Similar properties apply to Boolean expressions. Briefly stated, they are:

Figure 9.11 — *Axioms for Boolean Expressions*

```
(1)   A OR B         = B OR A
(2)   A AND B        = B AND A
(3)   A AND (B OR C) = (A AND B) OR (A AND C)
(4)   A OR (B AND C) = (A OR B) AND (A OR C)
(5)   A OR Ø         = A
(6)   A AND 1        = A
(7)   A OR NOT A     = 1
(8)   A AND NOT A    = Ø
(9)   0<>1
```

Where: A, B and C are arithmetic (or Boolean) expressions that evaluate to a value of one or zero. That is as far as I will take you in the study of the theory of Boolean logic. These types of expressions are used frequently in the coding of computer programs. The primary reason for going into this much detail is so you will recognize a "Bool" when you see one. The rest of this discussion will explain to you how Atari BASIC treats Boolean expressions and, also, explore some new ways to use Boolean expressions via machine language.

How Atari BASIC Treats Boolean Expressions

Atari BASIC only supports three of the standard Boolean operators: **OR**, **AND**, and **NOT**. Unfortunately, Atari BASIC treats Boolean operations only at the byte level, and not at the bit level like most other BASICs. Hence, "A = 1 OR 2" will result in A=0 instead of A=3. One special note of interest is that the statement

```
1ØØ IF NOT A THEN B=1Ø
```

This statement is a test for A=0. If A is equal to *any* other value, then the argument of the IF/THEN is automatically assumed to be "1" for logical purposes.

Boolean Logic in Machine Language

Sometimes it is really much more efficient to use logical operators on the bit level. The machine language routine BOOLEAN enables you to do **AND**, **OR** and exclusive OR **XOR** operations at the bit level. You can use BOOLEAN for **OR**, **AND**, **XOR** operations. The **NOT** operator is unary and can be easily performed by first testing the appropriate bit with BITMAP and then using BITMAP to CLEAR or SET the bit as needed.

Figure 9.12 — *BOOLEAN – Assembly Language Listing*

```
                1000 ;BOOLEAN - BIT-BY-BIT LOGICAL OPERATORS
                1010 ;
                1020 ;
                1030 ;CALLED FROM BASIC USING:
                1040 ;X=USR(ADDR,BYTE1,BYTE2,OPTION)
                1050 ;WHERE
                1060 ;       ADDR   = ADDRESS OF THIS ROUTINE
                1070 ;       BYTE1  = ARGUMENT #1
                1080 ;       BYTE2  = ARGUMENT #2
                1090 ;       OPTION = 0 MEANS 'OR'
                1100 ;                1 MEANS 'AND'
                1110 ;                2 MEANS 'XOR'
                1120 ;
                1130 ;
0000            1140        *=      $600            ;COMPLETELY RELOCATABLE
                1150 ;
                1160 ;
                1170 ;SET UP PAGE ZERO POINTERS
                1180 ;
00CC            1190 BYTE1  =       $CC
00CD            1200 BYTE2  =       $CD
00D4            1210 RESULT =       $D4
                1220 ;
                1230 ;
                1240 ;INITIALIZE POINTERS
                1250 ;
0600 A900       1260        LDA     #0              ;SET RESULT TO ZERO
0602 85D4       1270        STA     RESULT
0604 85D5       1280        STA     RESULT+1
0606 68         1290        PLA                     ;MAKE SURE THERE ARE NO
0607 C903       1300        CMP     #3              ;MORE THAN THREE ARGUMENTS
0609 F007       1310        BEQ     GOOD
060B AA         1320        TAX
060C 68         1330 KILL   PLA
060D 68         1340        PLA
060E CA         1350        DEX
060F D0FB       1360        BNE     KILL
0611 60         1370 EXIT   RTS                     ;GO BACK TO BASIC
```

```
0612 68     1380 GOOD   PLA                 ;GET BYTE1 AND BYTE2 LSB'S
0613 68     1390        PLA                 ;AND IGNORE MSB'S
0614 85CC   1400        STA     BYTE1
0616 68     1410        PLA
0617 68     1420        PLA
0618 85CD   1430        STA     BYTE2
            1440 ;
            1450 ;SELECT WHICH OPTION
            1460 ;
061A 68     1470        PLA
061B 68     1480        PLA
061C AA     1490        TAX
061D E000   1500        CPX     #0
061F F008   1510        BEQ     OR          ;OPTION = 0
0621 CA     1520        DEX
0622 F00C   1530        BEQ     AND         ;OPTION = 1
0624 CA     1540        DEX
0625 F010   1550        BEQ     XOR         ;OPTION = 2
            1560 ;
0627 D0E8   1570        BNE     EXIT        ;OPTION = SOMETHING ELSE
            1580 ;
            1590 ;BOOLEAN LOGIC ROUTINES
            1600 ;
0629 A5CC   1610 OR     LDA     BYTE1       ;BYTE1 OR BYTE2
062B 05CD   1620        ORA     BYTE2
062D 18     1630        CLC
062E 900B   1640        BCC     OUTPUT
0630 A5CC   1650 AND    LDA     BYTE1       ;BYTE1 AND BYTE2
0632 25CD   1660        AND     BYTE2
0634 18     1670        CLC
0635 9004   1680        BCC     OUTPUT
0637 A5CC   1690 XOR    LDA     BYTE1       ;BYTE1 XOR BYTE2
0639 45CD   1700        EOR     BYTE2
            1710 ;
            1720 ;STORE RESULT IN THE CALLING VARIABLE
            1730 ;
063B 85D4   1740 OUTPUT STA     RESULT
063D 18     1750        CLC
063E 90D1   1760        BCC     EXIT
0640        1770        .END
```

Figure 9.13 — *BOOLEAN.LST*

```
20410 REM BOOLEAN.LST
20411 DATA 169,0,133,212,133,213,104,201
20412 DATA 3,240,7,170,104,104,202,208
20413 DATA 251,96,104,104,133,204,104,104
20414 DATA 133,205,104,104,170,224,0,240
```

```
20415 DATA 8,202,240,12,202,240,16,208
20416 DATA 232,165,204,5,205,24,144,11
20417 DATA 165,204,37,205,24,144,4,165
20418 DATA 204,69,205,133,212,24,144,209
20419 DATA
20420 MLSTART=1536
20421 MLEND=1599
20422 FOR X=MLSTART TO MLEND
20423 READ Y:POKE X,Y:NEXT X
20424 RETURN
```

You can call BOOLEAN from BASIC with a USR command of the form:

```
RESULT = USR(ADDR,BYTE1,BYTE2,OPTION)

where: ADDR   = address of this machine language subroutine
       BYTE1  = a number between 0 and 255
       BYTE2  = another number between 0 and 255
       OPTION = 0 means a Boolean OR
              = 1 means a boolean AND
              = 2 means a boolean XOR
```

The variable RESULT will contain the appropriate answer upon the return to BASIC. As I recommended for BITMAP, you should use DATAPAK to string pack BOOLEAN for use in your programs.

Machine Language Boolean OR

To **OR** any two bytes, you must first make sure that they are in numerical form. Whole numbers between 0 and 255 are treated as a single byte by BOOLEAN. The USR call to **OR** BYTE1=7 and BYTE2=9 is:

```
RESULT = USR(ADR(BOOLEAN$),7,9,0)
```

If you did a PRINT RESULT, then the number 15 would be printed. This is the result of ORing 00000111 and 00001001, which is the correct answer.

Machine Language Boolean AND

Take the previous example and leave everything the same, except change OPTION=0 to OPTION=1. The result should be the result of ANDing 00000111 and 00001001, which is 00000001, or RESULT=1.

```
RESULT = USR(ADR(BOOLEAN$),7,9,1)
```

Machine Language Boolean XOR

Once again, keep everything the same and change OPTION to a value of one. This time, the result is 14. Look at the bits in the two arguments. They are the same except in the second, third and fourth bits. The exclusive OR of those two numbers is therefore 00001110, which is binary for 14.

```
RESULT = USR(ADR(BOOLEAN$),7,9,2)
```

An Un-Real World Example of Bit Level Logic

Once upon a time, I wrote a BASIC adventure game for my computer. This game had many different levels and many different rooms on each level (8*8*8 if memory serves). I wanted a way to code each of the rooms so the person playing the game could not cheat, and so I could pack a lot of room data in a minimum amount of space. The technique I ended up using involved the bit level encoding of certain key information. The chief problem was how to recognize what were valid exits from a room. This is what I did:

Figure 9.14 — *Exits from a Room*

Direction	Bit number
NORTH	0
SOUTH	1
EAST	2
WEST	3
UP	4
DOWN	5
TELEPORT	6

This way, if a room had three exits, I would SET the bit assigned to each valid direction. The resulting binary number, when converted to a decimal number, ROOMID, gave me a code that I could assign to that room which would identify every exit from the room. When the player tried to go in a particular direction from a room, the program would set DIRECTION equal to the direction number and call BITMAP with the following:

```
GATE = USR(ADR(BITMAP$),ROOMID,DIRECTION,2)
```

If the chosen direction was a valid exit, then the GATE would open for GATE=1, else it stayed closed. The TELEPORT direction would work only in certain special rooms when a special magic word was said.

You can use BITMAP to create the codes for the room IDs by using the following sequence of commands:

Figure 9.15 — *Setting Up Room Codes with Bit Level Logic*

```
100 ROOMID = 0
110 DIRECTION = 3 :REM IF WEST IS A VALID EXIT
120 GATE = USR(ADR(BITMAP$),ROOMID,DIRECTION,1)
130 ROOMID = GATE
140 DIRECTION = 4 :REM IF UP IS A VALID EXIT
150 GATE = USR(ADR(BITMAP$),ROOMID,DIRECTION,1)
```

and so forth. . .

Being essentially lazy, I wrote a short program that would ask for a room's matrix code in the dungeon, the name of that room and the legal exits from that room. The program would then generate a special code for that room and create a BASIC data statement in the same way that CONVERT.BAS does.

This is only a small sample of what you can do with Boolean expressions and bit mapping. There are many more ways to use them just waiting out there for you to figure out. Have fun.

Sorting Things Out

When programming your Atari computer, you will often find a need to work with lists of data. When you think about it, a major percentage of computer programming involves the storage and retrieval of information in one way or another.

In this chapter, we will reveal some techniques that can give you dramatic increases in data handling capability and some fantastic improvements in program execution speed. We will be dealing with the general topic of sorting data. We will discuss the most popular techniques and explain why they are so popular. These sorting techniques will be demonstrated in pure BASIC, and in one case I will show you how to include a fast machine language sort routine in your BASIC programs.

All Sorts of Sorts

Every time you separate the dimes from the nickels in your pocket change, you are using one or more forms of a technique called "sorting." According to my dictionary, the verb "sort" means:

```
1. To arrange according to class, kind, or size; classify
```

Computer programmers use this word in a much narrower sense. We think of "sorting" as arranging things in ascending or descending order. A more appropriate name would be "ordering." We will, with all due apologies to Mr. Webster, use the word "sorting" to mean "arranging things into an orderly sequence."

Sorting has been a common activity since before recorded history, but its most prevalent modern application is arranging data in numerical or alphabetical order for purposes of segregating groups of a similar type. Suppose that your record album collection has grown to more than 5,000 albums. It would be much easier to find your Janis Joplin albums (that dates me, doesn't it?) if the records are in some sort of order. The way you arrange your albums is one form of "sorting." The same principle holds true for the millions and millions of pieces of information that today's highly technological society must keep track of. Have you ever looked at a bank statement for a checking account? Typically my statements contain a list of my checks in numerical order. This was accomplished by "sorting."

Sorting, in the software sense, can be classified into two major categories. "In-memory" sorting means that the data to be sorted is contained in the available RAM of your computer. This method tends to be very fast since you can take advantage of the extremely fast access time of a computer's memory. The other major catagory is called "external" sorting. This method utilizes a large data file on disk in those cases where there is too much data to fit in the limited memory of your computer. There are special techniques for both types of sorting that take advantage of various computer configurations. The nature of the data to be sorted also plays a role in the choice of a particular sorting algorithm. We will concentrate our discussion in this chapter on two of the more popular in-memory sorting routines. The methods I will show you can readily be adapted for extremely large data files, but I suspect that most of your needs can be met by these relatively simple sort routines.

Let's take a moment to define some terminology that is commonly used when talking about sorts. First, a collection of data is typically called a "file." This file will, in turn, usually consist of one or more "records." For example, the list of the names of the students in a class might be your "file." Each name in the list would be a "record." Each record might also include a list of that student's test scores and the average score to date. When a list (file) is to be sorted, you many times will want to sort on a particular "field" or "key." In the previous example, you might want to sort the students in the class from "highest average score" to "lowest average score" or vice versa. The average score of each student is one "key" in the "record" for each student. The phrase "key" defines the part of a record that you will sort on. Each value for a key is called an "element."

When you are sorting a file, you typically want the records sorted in ascending or descending order, based on one or more keys. In the process of performing the sort, you must examine every record in the file at least once. This is what is called a "pass" on the file. Usually a sort operation will require many such passes before the sort is completed.

Bubble, Bubble, Toil and Trouble

By far the most popular sorting technique in use today is a method called a "bubble" sort. The popularity of this particular technique is due to its simplicity. A bubble sort is performed by simply comparing every two adjacent elements and swapping them if the first one is "greater than" the second. A flag is set every time two records are swapped. The sort is complete when a pass is made where no records are swapped, hence the flag is not set at the end of a pass. As this process moves its way through a list of data, the smaller elements appear to "bubble" their way to the top of the list.

This technique, which is also known as a simple interchange sort, is more than adequate for small data files that are not *too* much out of order. If either of these two conditions is not met, the time needed to sort a file can rapidly become very inconvenient. For example, The BASIC bubble sort shown in Figure. 10.1 takes about 43 seconds to sort 50 records, and takes almost three minutes to sort 100 records. The time required for this BASIC program to sort 1000 records is in the neighborhood of several hours. The dramatic increase in sorting time is due to the fact that the sort time goes up by roughly a factor of four everytime you double the number of records.

There are a variety of ways to modify the bubble sort to improve its speed. The most obvious is to use a bi-directional bubble sort, where we first make a pass from top-to-bottom looking for RECORD1>RECORD2 and then reverse the direction of the sort to go from bottom-to-top looking for RECORD2>RECORD1. No matter what minor modifications you make, the bubble sort is not very fast. Even if you write the bubble sort in machine language, it still will take an appreciable amount of time to sort a large file. There must be a better way to sort stuff that is faster without being overly complex. There is — the Shell sort.

Figure 10.1 — *BUBBLE.DEM – A BASIC Bubble Sort*

```
100 REM BUBBLE.DEM - BENCHMARK TEST FOR BASIC BUBBLE SORT
110 REM
120 REM SET UP TEST ROUTINE
130 REM
140 NUM=50
150 DIM TEST$(NUM),TEMP$(5):N=NUM:Z1=1
160 FOR X=1 TO NUM
170 Y=INT(200*RND(0)):TEST$(X,X)=CHR$(Y)
180 IF  NOT ((Y>47 AND Y<58) OR (Y>64 AND Y<91) OR (Y>96 AND Y<123)) THEN 170
190 NEXT X
200 REM
210 REM RESET CLOCK
220 REM
230 POKE 1536,1:POKE 1537,0:POKE 1538,0:POKE 1539,0:POKE 1536,0
240 REM
250 REM ROUTINE TO BE TESTED
260 REM
270 REM BUBBLE SORT ROUTINE
280 FLAG=0
290 FOR COUNT=Z1 TO N-Z1:
    IF ASC(TEST$(COUNT,COUNT))<=ASC(TEST$(COUNT+Z1,COUNT+Z1)) THEN 330
300 TEMP$=TEST$(COUNT,COUNT)
310 TEST$(COUNT,COUNT)=TEST$(COUNT+Z1,COUNT+Z1)
320 TEST$(COUNT+Z1,COUNT+Z1)=TEMP$:FLAG=Z1
330 NEXT COUNT
340 IF FLAG=Z1 THEN N=N-Z1:GOTO 280
350 REM
360 REM FIND ELAPSED TIME
370 REM
380 PRINT CHR$(125):PRINT CHR$(253):POSITION 2,5:PRINT "BENCHMARK TEST COMPLETED"
390 POSITION 2,9:POKE 1536,1:
    PRINT "ELAPSED TIME = ";PEEK(1537);:POSITION 20,9:PRINT " HOURS"
400 POSITION 17,10:PRINT ;PEEK(1538);:POSITION 20,10:PRINT " MINUTES"
410 POSITION 17,11:PRINT ;PEEK(1539);:POSITION 20,11:PRINT " SECONDS"
420 POKE 1536,0:END
```

Note: BUBBLE.DEM requires you to run CLOCK.BAS first.

The SHELL Game

Almost as simple in concept and far faster in execution, the Shell sort is a popular sort routine with many programmers. For 50 records, the shell sort is about three times faster than a bubble sort, and the differences in execution time for larger files is even more notable. For example, a BASIC shell sort will sort 100 records in about 37 seconds, as compared to the three minutes required for a bubble sort. The primary reason for the dramatic increase in speed is due to the fact that a shell sort requires fewer comparisons to sort the same data file.

The theory of a shell sort is relatively straightforward. In a list of N items, a boundary, FLAG, is computed such that 2EXP(FLAG)< N < 2EXP(FLAG+1). FLAG is then set equal to 2EXP(FLAG-1). A loop counts from 1 to N-FLAG checking for RECORD(COUNT) <= RECORD(COUNT+FLAG). If this condition is met, the counter is incremented and the next pair of records is compared. If the test fails, the two records are swapped before the counter is updated. When the counter reaches N, FLAG is divided by two, the counter is reset and a new loop (pass) is started. The sort is complete when FLAG reaches zero.

The name given to this sorting technique is *not* due to any similarity to the infamous "shell" con game. The origin of the name is a little more prosaic. The shell sort technique was originally proposed and published by Donald L. Shell back in 1959.

SHELL.DEM is a BASIC program that demonstrates the shell sort in such a way that the algorithm is easier to follow. The program assumes that you have previously run the real time clock we discussed earlier. The sort will still run properly if you haven't done this, but the actual sort time will not be computed for you. SHELL.DEM creates a data file, TEST$, which holds a collection of numbers and alpha characters. This file is then sorted using the shell sort technique, and the elapsed time is printed out for you. The program uses a general benchmark routine that I have used for a number of different tests. It can easily be modified to perform a benchmark test on just about any other routine that you might need to test.

Figure 10.2 — *SHELL.DEM – A BASIC SHELL SORT*

```
100 REM SHELL.DEM - BENCHMARK TEST FOR BASIC SHELL SORT
110 REM
120 REM SET UP TEST ROUTINE
130 REM
140 NUM=50
150 DIM TEST$(NUM),TEMP$(5):
    N=NUM:
    Z1=1
160 FOR X=1 TO NUM
170 Y=INT(200*RND(0)):
    TEST$(X,X)=CHR$(Y)
180 IF NOT ((Y>47 AND Y<58) OR
    (Y>64 AND Y<91) OR
    (Y>96 AND Y<123)) THEN 170
190 NEXT X
200 REM
210 REM RESET CLOCK
220 REM
230 POKE 1536,1:
    POKE 1537,0:
    POKE 1538,0:
    POKE 1539,0:
    POKE 1536,0
240 REM
250 REM ROUTINE TO BE TESTED
260 REM
270 REM SHELL SORT ROUTINE
280 FLAG=Z1
```

```
290 FLAG=2*FLAG:
    IF FLAG<=NUM THEN 290
300 FLAG=INT(FLAG/2):
    IF FLAG=0 THEN 390
310 FOR COUNT=Z1 TO N-FLAG:
    FLIP=COUNT
320 OTHER=FLIP+FLAG:
    X=ASC(TEST$(FLIP,FLIP)):
    Y=ASC(TEST$(OTHER,OTHER)):
    IF X<=Y THEN 350
330 TEMP$=CHR$(X):
    TEST$(FLIP,FLIP)=TEST$(OTHER,OTHER):
    TEST$(OTHER,OTHER)=TEMP$:
    FLIP=FLIP-FLAG
340 IF FLIP>0 THEN 320
350 NEXT COUNT:
    GOTO 300
360 REM
370 REM FIND ELAPSED TIME
380 REM
390 PRINT CHR$(125):
    PRINT CHR$(253):
    POSITION 2,5:
    PRINT "BENCHMARK TEST COMPLETED"
400 POSITION 2,9:
    POKE 1536,1:
    PRINT "ELAPSED TIME = ";PEEK(1537);:
    POSITION 20,9:
    PRINT " HOURS"
410 POSITION 17,10:
    PRINT ;PEEK(1538);:
    POSITION 20,10:
    PRINT " MINUTES"
420 POSITION 17,11:
    PRINT ;PEEK(1539);:
    POSITION 20,11:
    PRINT " SECONDS"
430 POKE 1536,0:
    END
```

The Shell Game Speeds Up

Even though the BASIC Shell sort is much faster than a BASIC bubble sort, sorting large files will still take longer than is convenient in a data processing environment. We have already demonstrated the superior speed of machine language programs. Let's see what kind of performance we can get out of a machine language shell sort.

The listing shown in Figure 10.3 is an assembled listing of a machine language shell sort routine. I tried to show in the comments column the parallel operations in the BASIC version of the routine. The machine language version takes up only 486 bytes, and works fast enough for most applications. This routine will sort 100 items in about four seconds. It is possible to create sort routines that will sort records much faster than this, but they are not only considerably more complex, they almost always require more memory. Those kinds of routines are more suitable for external sorting applications. Even those routines, however, can still take a relatively long time to sort very large files. You just have to face the reality that sorting is an inherently time consuming task.

If you are interested in studying other kinds of sorts, I recommend Volume 3 of the "programmers bible," *Sorting and Searching* by Donald E. Knuth. The three volume Knuth series is rather expensive and goes very heavily into the mathematics of programming theory. If complex math is not your strong point, you will have difficulty understanding the material. A book that might be easier for you to understand and make use of is *Sorting and Sort Systems* by H. Lorin.

The listing in Figure 10.4 is the BASIC POKE version, SORT.LST, of the machine language shell sort routine. I have it set it up like any other BASIC subroutine so that you can add it to your own BASIC programs. The machine code is completely relocatable, so I recommend that you use DATAPAK to string pack the routine for your every day use.

Figure 10.3 — *SHELL – Assembled Source Listing*

```
                1000 ;SHELL - A SHELL SORT ROUTINE
                1010 ;
                1020 ;
0000            1030        .OPT NOEJECT
                1040 ;
                1050 ;
                1060 ;CALLED FROM BASIC USING:
                1070 ;X=USR(ADDR,FILE,RECSIZE,NUMBER,
                1080 ;        KEYPOS,KEYLEN,DIRECT)
                1090 ;WHERE
                1100 ;        ADDR    = ADDRESS OF THIS ROUTINE
                1110 ;        FILE    = ADDRESS OF STRING HOLDING FILE
                1120 ;        RECSIZE = LENGTH OF EACH RECORD (<=256 BYTES)
                1130 ;        NUMBER  = NUMBER OF RECORDS TO SORT
                1140 ;        KEYPOS  = POSITION OF KEY IN RECORD
                1150 ;        KEYLEN  = LENGTH OF KEY
                1160 ;        DIRECT  = 0 MEANS ASCENDING SORT
                1170 ;                = 1 MEANS DESCENDING SORT
                1180 ;
                1190 ;
0000            1200        *=      $2B00          ;COMPLETELY RELOCATABLE
                1210 ;
                1220 ;
                1230 ;SET UP POINTERS
                1240 ;
00B4            1250 TEMP    =      $B4
00B6            1260 FILE    =      $B6
00B8            1270 RECSIZE =      $B8
00B9            1280 NUMBER  =      $B9
00BB            1290 KEYPOS  =      $BB
00BC            1300 KEYLEN  =      $BC
```

```
00BD          1310 DIRECT  =       $BD
00BE          1320 FLAG    =       $BE
00C0          1330 COUNT   =       $C0
00C2          1340 FLIP    =       $C2
00C4          1350 OTHER   =       $C4
00C6          1360 RECORD1 =       $C6
00C8          1370 RECORD2 =       $C8
00CA          1380 LIMIT   =       $CA
00CC          1390 KEY1    =       $CC
00CE          1400 KEY2    =       $CE
00D4          1410 ERROR   =       $D4
03E0          1420 VAULT   =       $3E0
              1430 ;
              1440 ;SAVE PAGE ZERO ON PAGE THREE
              1450 ;
2B00 A21C     1460         LDX     #$1C
2B02 B5B3     1470 SAVE    LDA     $B3,X
2B04 9DE003   1480         STA     VAULT,X
2B07 CA       1490         DEX
2B08 D0F8     1500         BNE     SAVE
              1510 ;
              1520 ;INPUT ERROR CHECK #1
              1530 ;
2B0A A900     1540         LDA     #0              ;SET ERROR FLAG TO ZERO
2B0C 85D4     1550         STA     ERROR
2B0E 85D5     1560         STA     ERROR+1
2B10 68       1570         PLA                     ;MAKE SURE THERE ARE
2B11 C906     1580         CMP     #6              ;EXACTLY SIX ARGUMENTS
2B13 F00D     1590         BEQ     GOOD
2B15 AA       1600         TAX
2B16 68       1610 KILL    PLA                     ;IF NOT, THEN ERROR=1 AND
2B17 68       1620         PLA                     ;GO TO EXIT
2B18 CA       1630         DEX
2B19 D0FB     1640         BNE     KILL
2B1B A201     1650         LDX     #1
2B1D 86D4     1660 BAD     STX     ERROR           ;STORE ERROR CODE
2B1F 18       1670         CLC
2B20 9025     1680         BCC     EXIT
              1690 ;
              1700 ;INITIALIZE POINTERS
              1710 ;
2B22 68       1720 GOOD    PLA
2B23 85B7     1730         STA     FILE+1
2B25 68       1740         PLA
2B26 85B6     1750         STA     FILE
2B28 68       1760         PLA
2B29 68       1770         PLA
2B2A 85B8     1780         STA     RECSIZE
2B2C 68       1790         PLA
2B2D 85BA     1800         STA     NUMBER+1
2B2F 68       1810         PLA
```

```
2B30 85B9    1820          STA     NUMBER
2B32 68      1830          PLA
2B33 68      1840          PLA
2B34 85BB    1850          STA     KEYPOS
2B36 C6BB    1860          DEC     KEYPOS          ;CONVERT TO BASE ZERO
2B38 85B4    1870          STA     TEMP            ;SAVE FOR ERROR CHECK
2B3A 68      1880          PLA
2B3B 68      1890          PLA
2B3C 85BC    1900          STA     KEYLEN
2B3E C6BC    1910          DEC     KEYLEN          ;CONVERT TO BASE ZERO
2B40 68      1920          PLA
2B41 68      1930          PLA
2B42 85BD    1940          STA     DIRECT
2B44 18      1950          CLC
2B45 900B    1960          BCC     CHECK2
             1970 ;
             1980 ;RESTORE PAGE ZERO VALUES
             1990 ;
2B47 A21C    2000 EXIT     LDX     #$1C
2B49 BDE003  2010 RESTORE  LDA     VAULT,X
2B4C 95B3    2020          STA     $B3,X
2B4E CA      2030          DEX
2B4F D0F8    2040          BNE     RESTORE
2B51 60      2050          RTS                     ;RETURN TO BASIC
             2060 ;
             2070 ;INPUT ERROR CHECK #2
             2080 ;
2B52 A202    2090 CHECK2   LDX     #2              ;IF FILE<2 THEN ERROR=2
2B54 A5B6    2100          LDA     FILE            ;AND GO TO BAD EXIT
2B56 C900    2110          CMP     #0
2B58 D004    2120          BNE     ERR3
2B5A A5B7    2130          LDA     FILE+1
2B5C F0BF    2140          BEQ     BAD
2B5E E8      2150 ERR3     INX                     ;IF RECSIZE<1 THEN ERROR=3
2B5F A5B8    2160          LDA     RECSIZE         ;AND GO TO BAD EXIT
2B61 D004    2170          BNE     ERR4
2B63 A5B9    2180          LDA     RECSIZE+1
2B65 F0B6    2190          BEQ     BAD
2B67 E8      2200 ERR4     INX                     ;IF NUMBER<2 THEN ERROR=4
2B68 A5B9    2210          LDA     NUMBER          ;AND GO TO BAD EXIT
2B6A D008    2220          BNE     ERR5
2B6C C901    2230          CMP     #1
2B6E D004    2240          BNE     ERR5
2B70 A5BA    2250          LDA     NUMBER+1
2B72 F0A9    2260          BEQ     BAD
2B74 E8      2270 ERR5     INX                     ;IF KEYPOS<0 THEN ERROR=5
2B75 24BB    2280          BIT     KEYPOS          ;AND GO TO BAD EXIT
2B77 1003    2290          BPL     ERR6
2B79 18      2300          CLC
2B7A 90A1    2310          BCC     BAD
2B7C E8      2320 ERR6     INX                     ;IF KEYLEN<0 OR
2B7D 24BC    2330          BIT     KEYLEN          ;IF KEYLEN>(RECSIZE-KEYPOS+1)
```

```
2B7F 1003   2340        BPL    ERR6A        ;THEN ERROR=6 AND
2B81 18     2350        CLC                 ;GO TO BAD EXIT
2B82 9099   2360        BCC    BAD
2B84 A5B8   2370 ERR6A  LDA    RECSIZE
2B86 C6B4   2380        DEC    TEMP
2B88 38     2390        SEC
2B89 E5B4   2400        SBC    TEMP
2B8B C5BC   2410        CMP    KEYLEN
2B8D 308E   2420        BMI    BAD
            2430 ;
            2440 ;SET UP SORT VARIABLES
            2450 ;
2B8F A900   2460        LDA    #0
2B91 85BF   2470        STA    FLAG+1       ;FLAG = 1
2B93 A901   2480        LDA    #1
2B95 85BE   2490        STA    FLAG
2B97 06BE   2500 LOOP1  ASL    FLAG         ;FLAG = 2*FLAG
2B99 26BF   2510        ROL    FLAG+1
2B9B A5B9   2520        LDA    NUMBER       ;IF FLAG<=NUMBER THEN LOOP1
2B9D C5BE   2530        CMP    FLAG
2B9F A5BA   2540        LDA    NUMBER+1
2BA1 E5BF   2550        SBC    FLAG+1
2BA3 B0F2   2560        BCS    LOOP1
2BA5 46BF   2570 LOOP3  LSR    FLAG+1       ;FLAG = FLAG/2
2BA7 66BE   2580        ROR    FLAG
2BA9 A5BE   2590        LDA    FLAG         ;IF FLAG = 0 THEN
2BAB 05BF   2600        ORA    FLAG+1       ;SORT IS COMPLETE AND
2BAD F098   2610        BEQ    EXIT         ;GO TO NORMAL EXIT
2BAF A901   2620 MAIN   LDA    #1           ;FOR COUNT=1 TO (NUMBER-FLAG)
2BB1 85C0   2630        STA    COUNT
2BB3 A900   2640        LDA    #0
2BB5 85C1   2650        STA    COUNT+1
2BB7 38     2660        SEC
2BB8 A5B9   2670        LDA    NUMBER
2BBA E5BE   2680        SBC    FLAG
2BBC 85CA   2690        STA    LIMIT
2BBE A5BA   2700        LDA    NUMBER+1
2BC0 E5BF   2710        SBC    FLAG+1
2BC2 85CB   2720        STA    LIMIT+1
2BC4 A5C0   2730 LOOP2  LDA    COUNT        ;FLIP = COUNT
2BC6 85C2   2740        STA    FLIP
2BC8 A5C1   2750        LDA    COUNT+1
2BCA 85C3   2760        STA    FLIP+1
2BCC 18     2770        CLC
2BCD 9006   2780        BCC    AGAIN
2BCF 90D4   2790 DUM3   BCC    LOOP3        ;LILLY PADS
2BD1 B0F1   2800 DUM2   BCS    LOOP2
2BD3 D000   2810 DUM1   BNE    AGAIN
2BD5 18     2820 AGAIN  CLC                 ;OTHER = FLIP+FLAG
2BD6 A5C2   2830        LDA    FLIP
2BD8 65BE   2840        ADC    FLAG
2BDA 85C4   2850        STA    OTHER
```

```
2BDC A5C3   2860         LDA    FLIP+1
2BDE 65BF   2870         ADC    FLAG+1
2BE0 85C5   2880         STA    OTHER+1
            2890 ;
            2900 ;POINT TO TEST RECORDS
            2910 ;
2BE2 A5B6   2920         LDA    FILE
2BE4 85C6   2930         STA    RECORD1
2BE6 85C8   2940         STA    RECORD2
2BE8 A5B7   2950         LDA    FILE+1
2BEA 85C7   2960         STA    RECORD1+1
2BEC 85C9   2970         STA    RECORD2+1
2BEE A901   2980         LDA    #1              ;POINT TO FIRST RECORD
2BF0 85B4   2990         STA    TEMP
2BF2 C5C2   3000         CMP    FLIP
2BF4 D006   3010         BNE    SET1A
2BF6 A900   3020         LDA    #0
2BF8 C5C3   3030         CMP    FLIP+1
2BFA F02E   3040         BEQ    SETKEY1
2BFC A900   3050 SET1A   LDA    #0
2BFE 85B5   3060         STA    TEMP+1
2C00 18     3070 SET1B   CLC
2C01 A5C6   3080         LDA    RECORD1
2C03 65B8   3090         ADC    RECSIZE
2C05 85C6   3100         STA    RECORD1
2C07 A5C7   3110         LDA    RECORD1+1
2C09 6900   3120         ADC    #0
2C0B 85C7   3130         STA    RECORD1+1
2C0D E6B4   3140         INC    TEMP
2C0F D002   3150         BNE    SET1C
2C11 E6B5   3160         INC    TEMP+1
2C13 A5B4   3170 SET1C   LDA    TEMP
2C15 C5C2   3180         CMP    FLIP
2C17 D0E7   3190         BNE    SET1B
2C19 A5B5   3200         LDA    TEMP+1
2C1B C5C3   3210         CMP    FLIP+1
2C1D D0E1   3220         BNE    SET1B
2C1F 18     3230         CLC
2C20 9008   3240         BCC    SETKEY1
2C22 D0AF   3250 DUM1B   BNE    DUM1            ;MORE LILLY PADS
2C24 B0AB   3260 DUM2B   BCS    DUM2
2C26 90A7   3270 DUM3B   BCC    DUM3
2C28 D0D6   3280         BNE    SET1B
2C2A 18     3290 SETKEY1 CLC                    ;POINT TO KEY IN RECORD1
2C2B A5C6   3300         LDA    RECORD1
2C2D 65BB   3310         ADC    KEYPOS
2C2F 85CC   3320         STA    KEY1
2C31 A5C7   3330         LDA    RECORD1+1
2C33 6900   3340         ADC    #0
2C35 85CD   3350         STA    KEY1+1
2C37 A901   3360 SET2A   LDA    #1              ;POINT TO SECOND RECORD
2C39 85B4   3370         STA    TEMP
```

```
2C3B C5C4    3380         CMP    OTHER
2C3D D006    3390         BNE    SET2B
2C3F A900    3400         LDA    #0
2C41 C5C5    3410         CMP    OTHER+1
2C43 F023    3420         BEQ    SETKEY2
2C45 A900    3430 SET2B   LDA    #0
2C47 85B5    3440         STA    TEMP+1
2C49 18      3450 SET2C   CLC
2C4A A5C8    3460         LDA    RECORD2
2C4C 65B8    3470         ADC    RECSIZE
2C4E 85C8    3480         STA    RECORD2
2C50 A5C9    3490         LDA    RECORD2+1
2C52 6900    3500         ADC    #0
2C54 85C9    3510         STA    RECORD2+1
2C56 E6B4    3520         INC    TEMP
2C58 D002    3530         BNE    SET2D
2C5A E6B5    3540         INC    TEMP+1
2C5C A5B4    3550 SET2D   LDA    TEMP
2C5E C5C4    3560         CMP    OTHER
2C60 D0E7    3570         BNE    SET2C
2C62 A5B5    3580         LDA    TEMP+1
2C64 C5C5    3590         CMP    OTHER+1
2C66 D0E1    3600         BNE    SET2C
2C68 18      3610 SETKEY2 CLC                    ;POINT TO KEY IN RECORD2
2C69 A5C8    3620         LDA    RECORD2
2C6B 65BB    3630         ADC    KEYPOS
2C6D 85CE    3640         STA    KEY2
2C6F A5C9    3650         LDA    RECORD2+1
2C71 6900    3660         ADC    #0
2C73 85CF    3670         STA    KEY2+1
             3680 ;
             3690 ;MAIN SORT ROUTINE
             3700 ;
2C75 A000    3710 SORT    LDY    #0             ;PICK A SORT DIRECTION
2C77 A5BD    3720         LDA    DIRECT
2C79 D013    3730         BNE    DOWN
2C7B B1CC    3740 UP      LDA    (KEY1),Y       ;SORT IN ASCENDING ORDER
2C7D D1CE    3750         CMP    (KEY2),Y
2C7F 3054    3760         BMI    BMPREC
2C81 C8      3770         INY
2C82 C4BC    3780         CPY    KEYLEN
2C84 90F5    3790         BCC    UP
2C86 B011    3800         BCS    SWAP
2C88 D098    3810 DUM1A   BNE    DUM1B          ;EVEN MORE LILLY PADS
2C8A B098    3820 DUM2A   BCS    DUM2B
2C8C 9098    3830 DUM3A   BCC    DUM3B
2C8E B1CE    3840 DOWN    LDA    (KEY2),Y       ;SORT IN DESCENDING ORDER
2C90 D1CC    3850         CMP    (KEY1),Y
2C92 3041    3860         BMI    BMPREC
2C94 C8      3870         INY
2C95 C4BC    3880         CPY    KEYLEN
```

```
2C97 90F5   3890          BCC     DOWN
            3900 ;
            3910 ;SWAP RECORD1 WITH RECORD2
            3920 ;
2C99 A000   3930 SWAP     LDY     #0               ;POINT BUFFER TO PAGE FOUR
2C9B A904   3940          LDA     #4
2C9D 85B5   3950          STA     TEMP+1
2C9F A900   3960          LDA     #0
2CA1 85B4   3970          STA     TEMP
2CA3 B1C6   3980 SWAPA    LDA     (RECORD1),Y      ;TEMP$ = TEST$(REC1)
2CA5 91B4   3990          STA     (TEMP),Y
2CA7 C8     4000          INY
2CA8 C4B8   4010          CPY     RECSIZE
2CAA D0F7   4020          BNE     SWAPA
2CAC A000   4030          LDY     #0               ;TEST$(REC1) = TEST$(REC2)
2CAE B1C8   4040 SWAPB    LDA     (RECORD2),Y
2CB0 91C6   4050          STA     (RECORD1),Y
2CB2 C8     4060          INY
2CB3 C4B8   4070          CPY     RECSIZE
2CB5 D0F7   4080          BNE     SWAPB
2CB7 A000   4090          LDY     #0               ;TEST$(REC2) = TEMP$
2CB9 B1B4   4100 SWAPC    LDA     (TEMP),Y
2CBB 91C8   4110          STA     (RECORD2),Y
2CBD C8     4120          INY
2CBE C4B8   4130          CPY     RECSIZE
2CC0 D0F7   4140          BNE     SWAPC
            4150 ;
            4160 ;UPDATE COUNTERS
            4170 ;
2CC2 A5C2   4180          LDA     FLIP             ;FLIP = FLIP-FLAG
2CC4 38     4190          SEC
2CC5 E5BE   4200          SBC     FLAG
2CC7 85C2   4210          STA     FLIP
2CC9 A5C3   4220          LDA     FLIP+1
2CCB E5BF   4230          SBC     FLAG+1
2CCD 85C3   4240          STA     FLIP+1
2CCF 9004   4250          BCC     BMPREC
2CD1 05C2   4260          ORA     FLIP
2CD3 D0B3   4270 FROG1    BNE     DUM1A
2CD5 E6C0   4280 BMPREC   INC     COUNT            ;NEXT COUNT
2CD7 D002   4290          BNE     NEXT
2CD9 E6C1   4300          INC     COUNT+1
2CDB A5CA   4310 NEXT     LDA     LIMIT            ;IS THIS PASS COMPLETED?
2CDD C5C0   4320          CMP     COUNT
2CDF A5CB   4330          LDA     LIMIT+1
2CE1 E5C1   4340          SBC     COUNT+1
2CE3 B0A5   4350 FROG2    BCS     DUM2A            ;NO? THEN GO BACK TO LOOP2
2CE5 90A5   4360 FROG3    BCC     DUM3A            ;START A NEW PASS (LOOP3)
2CE7        4370          .END
```

Figure 10.4 — *SORT.LST - BASIC POKE Version of SHELL*

```
20430 REM SORT.LST
20431 DATA 162,28,181,179,157,224,3,202
20432 DATA 208,248,169,0,133,212,133,213
20433 DATA 104,201,6,240,13,170,104,104
20434 DATA 202,208,251,162,1,134,212,24
20435 DATA 144,37,104,133,183,104,133,182
20436 DATA 104,104,133,184,104,133,186,104
20437 DATA 133,185,104,104,133,187,198,187
20438 DATA 133,180,104,104,133,188,198,188
20439 DATA 104,104,133,189,24,144,11,162
20440 DATA 28,189,224,3,149,179,202,208
20441 DATA 248,96,162,2,165,182,201,0
20442 DATA 208,4,165,183,240,191,232,165
20443 DATA 184,208,4,165,185,240,182,232
20444 DATA 165,185,208,8,201,1,208,4
20445 DATA 165,186,240,169,232,36,187,16
20446 DATA 3,24,144,161,232,36,188,16
20447 DATA 3,24,144,153,165,184,198,180
20448 DATA 56,229,180,197,188,48,142,169
20449 DATA 0,133,191,169,1,133,190,6
20450 DATA 190,38,191,165,185,197,190,165
20451 DATA 186,229,191,176,242,70,191,102
20452 DATA 190,165,190,5,191,240,152,169
20453 DATA 1,133,192,169,0,133,193,56
20454 DATA 165,185,229,190,133,202,165,186
20455 DATA 229,191,133,203,165,192,133,194
20456 DATA 165,193,133,195,24,144,6,144
20457 DATA 212,176,241,208,0,24,165,194
20458 DATA 101,190,133,196,165,195,101,191
20459 DATA 133,197,165,182,133,198,133,200
20460 DATA 165,183,133,199,133,201,169,1
20461 DATA 133,180,197,194,208,6,169,0
20462 DATA 197,195,240,46,169,0,133,181
20463 DATA 24,165,198,101,184,133,198,165
20464 DATA 199,105,0,133,199,230,180,208
20465 DATA 2,230,181,165,180,197,194,208
20466 DATA 231,165,181,197,195,208,225,24
20467 DATA 144,8,208,175,176,171,144,167
20468 DATA 208,214,24,165,198,101,187,133
20469 DATA 204,165,199,105,0,133,205,169
20470 DATA 1,133,180,197,196,208,6,169
20471 DATA 0,197,197,240,35,169,0,133
20472 DATA 181,24,165,200,101,184,133,200
20473 DATA 165,201,105,0,133,201,230,180
20474 DATA 208,2,230,181,165,180,197,196
20475 DATA 208,231,165,181,197,197,208,225
20476 DATA 24,165,200,101,187,133,206,165
20477 DATA 201,105,0,133,207,160,0,165
```

```
20478 DATA 189,208,19,177,204,209,206,48
20479 DATA 84,200,196,188,144,245,176,17
20480 DATA 208,152,176,152,144,152,177,206
20481 DATA 209,204,48,65,200,196,188,144
20482 DATA 245,160,0,169,4,133,181,169
20483 DATA 0,133,180,177,198,145,180,200
20484 DATA 196,184,208,247,160,0,177,200
20485 DATA 145,198,200,196,184,208,247,160
20486 DATA 0,177,180,145,200,200,196,184
20487 DATA 208,247,165,194,56,229,190,133
20488 DATA 194,165,195,229,191,133,195,144
20489 DATA 4,5,194,208,179,230,192,208
20490 DATA 2,230,193,165,202,197,192,165
20491 DATA 203,229,193,176,165,144,165
20492 MLSTART=11008
20493 MLEND=11494
20494 FOR X=MLSTART TO MLEND
20495 READ Y:
      POKE X,Y:
      NEXT X
20496 RETURN
```

Using SORT.LST is easy. After you have loaded, POKEd, or packed the routine into memory, you access it with a USR call of the form:

```
ERROR=USR(ADDR,FILE,RECSIZE,NUMBER,KEYPOS,KEYLEN,DIRECT)

where:
       ERROR   = Error code returned to BASIC
       ADDR    = Address of the sort routine
       FILE    = Address of the file to be sorted
       RECSIZE = The number of bytes in each record
       NUMBER  = The number of records in the file
       KEYPOS  = The position of the sort key in each record
       KEYLEN  = The number of bytes in the sort key
       DIRECT  = The direction of the sort (ascending or descending)
```

ADDR is computed the same way we have done this in previous cases. FILE is equal to ADR(FILE$), where FILE$ is the string array in which you have stored your data. You can also set FILE equal to a particular memory address if you have POKEd your data into memory. I will show you an example of both methods a little later. RECSIZE must be set equal to the number of bytes you have allocated for each record. NUMBER is simply the number of such records you want the routine to sort on. DIRECT specifies the direction of the sort. A value of zero means the data will be sorted from the smallest to the largest (ascending). Any non-zero value will cause the sort to be in descending order, from largest to smallest.

Note the two "key" related arguments. This sort routine is a little more sophisticated than its BASIC counterpart. By specifying a key position greater than one, you can have the routine sort on any key anywhere in a record. Of course, all of the records must have the same key in the same location inside each record. The key can have a maximum length of 128 bytes. This length is specified by the argument KEYLEN.

The variable, ERROR, is used to call the routine. Under most circumstances, this variable will always be be equal to zero. The machine language routine does some input error checking to make sure you have used the right number of arguments and that each argument is assigned a legal value. The possible error codes are:

```
Error Codes For Machine Language Shell Sort
ERROR CODE   MEANING
Ø            No errors
1            Number of arguments is not equal to 6
2            File address is less than Ø
3            RECSIZE is less than 1
4            Number of records is less than 2
5            Key position is less than 1 (or>128)
6            Key length is less than 1   (or>128)
```

You will note that there is no upper limit specified for the number of records. This is because the number will depend upon how much memory your system has. One advantage of this sort routine is that you can sort any number of records, as long as you can fit them all into your computer.

Making Numeric Data Sortable

The need to sort numbers is a special problem. Since numbers are stored in the six-byte BCD format and you don't have a numeric equivalent of the "ADR(STRING$)" command, numbers not only are hard to find; they are difficult to sort once you do find them. The solution to this problem is simple, fortunately. All you have to do is convert all of the numeric data to strings using the STR$(X) command. Yes, I realize that this could be a time consuming process, but the alternative is to do without the capabilty of sorting numeric tables.

The next case to consider is sorting a table of positive and negative numbers. This breaks down into two problem areas. The first is that the ATASCII value of a "+" is less than the ATASCII value of a "-". When sorting negative and positive numbers, you should not have a "+" sign in front of your positive numbers. That solves this particular problem since the ATASCII value of the "-" sign is less than the ATASCII value of any number and therefore will be sorted as "less than" any positive number. Be careful not to have any leading spaces on your positive numbers either, or you will run into the same problem that you had with the "+" sign.

The second problem with sorting such numbers is leading zeroes (not spaces). If you are comparing the number "9" to the number "27", the result will tell you that 27 is less than 9 because the comparison is made a digit at a time. The solution to this problem is to add leading zeroes to the smaller numbers so all of the numbers will , in effect, have the same number of digits in front of the decimal (including the "-" sign). This is easily done when you are converting the numeric data to strings. You avoid any problem with the "-" sign because zero has a higher ATASCII value than it. Now when you sort the "numbers," the number "09" will correctly be sorted as being less than "27."

Unfortunately, there is still another problem you have to handle before you start sorting numbers. If any of the numbers in the sort file have decimals (as in dollar and cents), these numbers will not be sorted properly because a "." has a higher ATASCII value than a space and a lower value than a number. The solution to this problem is to either eliminate the decimal point (by multiplying the numbers by a large enough factor of ten), or by adding enough trailing zeroes, to make sure that all of the decimal points line up. For monetary numbers, you can easily do this using the MONEY.LST routine that we discussed earlier in this book. In any case, the sorted numbers can be recovered as numbers by using a "VAL(STRING$)" command on them.

Sorting With Assorted Keys

Let's suppose that you have data for several retail stores. Working at each store you have several salesmen, and your computer program has accumulated total sales for each salesman. You have stored the data in a file with 10 bytes allocated to each key, giving a total record length of 30 bytes:

```
Store Sales Data File
STORE     SALESMAN  SALES

Ø123456789Ø123456789Ø123456789
CHINO     JONES     532.4Ø
AZUSA     DIETL     221.28
UPLAND    MARRACK   223.32
UPLAND    JOHNSON   332.22
ONTARIO   SAMMS     Ø52.48
ONTARIO   BURKE     299.ØØ
```

To sort the data in alphabetical order by store and within each store in alphabetical order by salesman, you simply set RECSIZE=30, NUMBER=6, KEYPOS=1, KEYLEN=20, DIRECT=0 and call up the sort routine. Change KEYLEN to 10 if you only want to sort by store location, and change KEYPOS to 11 if you only want to sort by salesman.

```
Ø123456789Ø123456789Ø123456789
CHINO     JONES     532.4Ø
AZUSA     DIETL     221.28
UPLAND    MARRACK   223.32
UPLAND    JOHNSON   332.22
ONTARIO   SAMMS     Ø52.48
ONTARIO   BURKE     299.ØØ
```

After the data is sorted in ascending sequence, you can split out the keys and store them back in their separate arrays for further processing. Here's what you get:

```
Ø123456789Ø123456789Ø123456789
AZUSA     DIETL     221.28
CHINO     JONES     532.4Ø
ONTARIO   BURKE     299.ØØ
ONTARIO   SAMMS     Ø52.48
UPLAND    JOHNSON   332.22
UPLAND    MARRACK   223.32
```

Now suppose you want to sort so that the salesman with the lowest sales total is shown at the top of the list and if more than one salesman has the same sales figure, they are to be listed alphabetically. To do this, set KEYPOS=11 and KEYLEN=10. Run the sort routine to get the following:

```
Ø123456789Ø123456789Ø123456789
ONTARIO   SAMMS     Ø52.48
AZUSA     DIETL     221.28
UPLAND    MARRACK   223.32
ONTARIO   BURKE     299.ØØ
UPLAND    JOHNSON   332.22
CHINO     JONES     532.ØØ
```

Since there are no duplicated sales figures we are done; if there had been duplicated sales figures, we could handle this situation in one of two ways. The first is to create a small sort file that consists of the records for a particular duplicated sales figure and sort that small file on the name key by setting KEYPOS=11 and KEYLEN=10. This is how I do it most of the time since I usually have only a couple of duplicated figures. If you have a large number of duplicated figures, you probably should create a brand new sort file that looks like this:

```
Ø123456789Ø123456789Ø123456789
CHINO     532.4Ø    JONES
AZUSA     221.28    DIETL
UPLAND    223.32    MARRACK
UPLAND    332.22    JOHNSON
ONTARIO   Ø52.48    SAMMS
ONTARIO   299.ØØ    BURKE
```

This new file could now be sorted in the desired way.

Now let's suppose you want the salesman with the highest sales total to be shown at the top of the list. In other words, you want the list sorted in descending sequence by sales total, ascending sequence by salesman, and ascending sequence by store location. One method you can use is to sort in descending sequence similar to what we did above (but with DIRECT=1).

The only problem with this technique is that the names of the salesmen will not be in the right order, and the store locations will also be in in the wrong order if two sales totals are equal.

A better solution is to use INVERT.LST to complement the keys that we want to be sorted in the opposite order of the primary key. The complement of "AAA" is greater than the complement of "BBB".

In our example, we would want to complement the sales amount key before we do the sort. Be sure to use a dummy file! After the sort, we use INVERT.LST to complement the sales amount keys again to restore them to their original values.

Sorting Demonstration Programs

The programs listed in Figures 10.5 and 10.6 are demonstration programs that illustrate ways to use SORT.LST. The first program SHELL2.DEM is a simple benchmark program similar to SHELL.DEM. You should RUN the real time clock program before running this demo. The sort will still be performed if you don't, but the elapsed time won't be correct. The demo is set up to sort 50 random alpha-numeric characters. If you would like to sort a larger number, change the value of the variable NUM in line 160 to the desired number. If you want the sort to be in descending order instead of ascending order, change DIRECT to one. The sorted file is stored in the string TEST$. To have a look at the sorted data, just PRINT TEST$.

The second demo program is a lot more interesting. SHELL3.DEM is a visual sorting routine. The data file is POKEd to the video screen, and the sort routine is told to use the screen display as the source file to be sorted. The results are very interesting! You can actually watch the sort take place. This should enable you to better understand exactly how the shell sort really works. The demo is set up to display 320 lower case alpha characters on the bottom half of the screen. I used a file this size so you would have time to see the sort routine working. I think you will like the results. Have fun!

Figure 10.5 — *SHELL2.DEM – A Shell Sort Benchmark Test*

```
100 REM SHELL2.DEM -
110 REM BENCHMARK TEST FOR
120 REM MACHINE LANGUAGE SHELL SORT
130 REM
140 REM SET UP TEST ROUTINE
150 REM
160 NUM=50:
    DIM SORT$(487),TEST$(NUM):
    N=NUM:
    Z1=1:
    Z200=200:
    Z47=47:
    Z58=58:
    Z64=64:
    Z91=91:
    Z96=96:
    Z123=123
170 FOR X=Z1 TO NUM
180 Y=INT(Z200*RND(0)):
    TEST$(X,X)=CHR$(Y)
190 IF NOT ((Y>Z47 AND Y<Z58) OR
    (Y>Z64 AND Y<Z91) OR
    (Y>Z96 AND Y<Z123)) THEN 180
200 NEXT X
210 FILE=ADR(TEST$):
    RECSIZE=1:
    NUMBER=NUM:
    KEYPOS=1:
    KEYLEN=1:
    DIRECT =0
220 FOR X=ADR(SORT$) TO ADR(SORT$)+486:
    READ Y:
    POKE X,Y:
    NEXT X
```

```
230 REM
240 REM RESET CLOCK
250 REM
260 POKE 1536,1:
    POKE 1537,0:
    POKE 1538,0:
    POKE 1539,0:
    POKE 1536,0
270 REM
280 REM ROUTINE TO BE TESTED
290 REM
300 REM SHELL SORT ROUTINE
310 ERROR=USR(ADR(SORT$),FILE,RECSIZE,NUMBER,KEYPOS,KEYLEN,DIRECT)
320 REM
330 REM FIND ELAPSED TIME
340 REM
350 PRINT CHR$(125):
    PRINT CHR$(253):
    POSITION 2,5:
    PRINT "BENCHMARK TEST COMPLETED"
360 POSITION 2,9:
    POKE 1536,1:
    PRINT "ELAPSED TIME = ";PEEK(1537);:
    POSITION 20,9:
    PRINT " HOURS"
370 POSITION 17,10:
    PRINT ;PEEK(1538);:
    POSITION 20,10:
    PRINT " MINUTES"
380 POSITION 17,11:
    PRINT ;PEEK(1539);:
    POSITION 20,11:
    PRINT " SECONDS"
390 POKE 1536,0:
    END
400 REM
410 REM MERGE POKE DATA FOR
420 REM ML SORT ROUTINE HERE
430 REM
440 DATA 162,28,181,179,157,224,3,202
450 DATA 208,248,169,0,133,212,133,213
460 DATA 104,201,6,240,13,170,104,104
470 DATA 202,208,251,162,1,134,212,24
480 DATA 144,37,104,133,183,104,133,182
490 DATA 104,104,133,184,104,133,186,104
500 DATA 133,185,104,104,133,187,198,187
510 DATA 133,180,104,104,133,188,198,188
520 DATA 104,104,133,189,24,144,11,162
530 DATA 28,189,224,3,149,179,202,208
540 DATA 248,96,162,2,165,182,201,0
550 DATA 208,4,165,183,240,191,232,165
560 DATA 184,208,4,165,185,240,182,232
```

```
570 DATA 165,185,208,8,201,1,208,4
580 DATA 165,186,240,169,232,36,187,16
590 DATA 3,24,144,161,232,36,188,16
600 DATA 3,24,144,153,165,184,198,180
610 DATA 56,229,180,197,188,48,142,169
620 DATA 0,133,191,169,1,133,190,6
630 DATA 190,38,191,165,185,197,190,165
640 DATA 186,229,191,176,242,70,191,102
650 DATA 190,165,190,5,191,240,152,169
660 DATA 1,133,192,169,0,133,193,56
670 DATA 165,185,229,190,133,202,165,186
680 DATA 229,191,133,203,165,192,133,194
690 DATA 165,193,133,195,24,144,6,144
700 DATA 212,176,241,208,0,24,165,194
710 DATA 101,190,133,196,165,195,101,191
720 DATA 133,197,165,182,133,198,133,200
730 DATA 165,183,133,199,133,201,169,1
740 DATA 133,180,197,194,208,6,169,0
750 DATA 197,195,240,46,169,0,133,181
760 DATA 24,165,198,101,184,133,198,165
770 DATA 199,105,0,133,199,230,180,208
780 DATA 2,230,181,165,180,197,194,208
790 DATA 231,165,181,197,195,208,225,24
800 DATA 144,8,208,175,176,171,144,167
810 DATA 208,214,24,165,198,101,187,133
820 DATA 204,165,199,105,0,133,205,169
830 DATA 1,133,180,197,196,208,6,169
840 DATA 0,197,197,240,35,169,0,133
850 DATA 181,24,165,200,101,184,133,200
860 DATA 165,201,105,0,133,201,230,180
870 DATA 208,2,230,181,165,180,197,196
880 DATA 208,231,165,181,197,197,208,225
890 DATA 24,165,200,101,187,133,206,165
900 DATA 201,105,0,133,207,160,0,165
910 DATA 189,208,19,177,204,209,206,48
920 DATA 84,200,196,188,144,245,176,17
930 DATA 208,152,176,152,144,152,177,206
940 DATA 209,204,48,65,200,196,188,144
950 DATA 245,160,0,169,4,133,181,169
960 DATA 0,133,180,177,198,145,180,200
970 DATA 196,184,208,247,160,0,177,200
980 DATA 145,198,200,196,184,208,247,160
990 DATA 0,177,180,145,200,200,196,184
1000 DATA 208,247,165,194,56,229,190,133
1010 DATA 194,165,195,229,191,133,195,144
1020 DATA 4,5,194,208,179,230,192,208
1030 DATA 2,230,193,165,202,197,192,165
1040 DATA 203,229,193,176,165,144,165
```

Figure 10.6 — *SHELL3.DEM – A Visual Sort of Experience*

```
100 REM SHELL3.DEM -
110 REM A VISUAL SORT OF EXPERIENCE
120 REM
130 REM SET UP DATA TO BE SORTED
140 REM
150 NUM=320:
    DIM SORT$(487):
    Z1=1:
    Z25=25:
    Z97=97.5:
    POKE 752,1
160 PRINT CHR$(125):
    PRINT :PRINT :PRINT :
    PRINT "     SHELL3.DEM - A VISUAL SORT":
    FOR X=Z1 TO NUM
170 Y=INT(Z25*RND(0)+Z97)
180 POKE 40399+X,Y:
    NEXT X
190 FILE=40400:
    RECSIZE=1:
    NUMBER=NUM:
    KEYPOS=1:
    KEYLEN=1:
    DIRECT=0
200 FOR X=ADR(SORT$) TO ADR(SORT$)+486:
    READ Y:
    POKE X,Y:
    NEXT X
210 REM
220 REM SHELL SORT ROUTINE
230 REM
240 ERROR=USR(ADR(SORT$),FILE,RECSIZE,NUMBER,KEYPOS,KEYLEN,DIRECT )
250 END
260 REM
270 REM MERGE POKE DATA FOR
280 REM ML SORT ROUTINE HERE
290 REM
300 DATA 162,28,181,179,157,224,3,202
310 DATA 208,248,169,0,133,212,133,213
320 DATA 104,201,6,240,13,170,104,104
330 DATA 202,208,251,162,1,134,212,24
340 DATA 144,37,104,133,183,104,133,182
350 DATA 104,104,133,184,104,133,186,104
360 DATA 133,185,104,104,133,187,198,187
370 DATA 133,180,104,104,133,188,198,188
380 DATA 104,104,133,189,24,144,11,162
390 DATA 28,189,224,3,149,179,202,208
400 DATA 248,96,162,2,165,182,201,0
```

```
410 DATA 208,4,165,183,240,191,232,165
420 DATA 184,208,4,165,185,240,182,232
430 DATA 165,185,208,8,201,1,208,4
440 DATA 165,186,240,169,232,36,187,16
450 DATA 3,24,144,161,232,36,188,16
460 DATA 3,24,144,153,165,184,198,180
470 DATA 56,229,180,197,188,48,142,169
480 DATA 0,133,191,169,1,133,190,6
490 DATA 190,38,191,165,185,197,190,165
500 DATA 186,229,191,176,242,70,191,102
510 DATA 190,165,190,5,191,240,152,169
520 DATA 1,133,192,169,0,133,193,56
530 DATA 165,185,229,190,133,202,165,186
540 DATA 229,191,133,203,165,192,133,194
550 DATA 165,193,133,195,24,144,6,144
560 DATA 212,176,241,208,0,24,165,194
570 DATA 101,190,133,196,165,195,101,191
580 DATA 133,197,165,182,133,198,133,200
590 DATA 165,183,133,199,133,201,169,1
600 DATA 133,180,197,194,208,6,169,0
610 DATA 197,195,240,46,169,0,133,181
620 DATA 24,165,198,101,184,133,198,165
630 DATA 199,105,0,133,199,230,180,208
640 DATA 2,230,181,165,180,197,194,208
650 DATA 231,165,181,197,195,208,225,24
660 DATA 144,8,208,175,176,171,144,167
670 DATA 208,214,24,165,198,101,187,133
680 DATA 204,165,199,105,0,133,205,169
690 DATA 1,133,180,197,196,208,6,169
700 DATA 0,197,197,240,35,169,0,133
710 DATA 181,24,165,200,101,184,133,200
720 DATA 165,201,105,0,133,201,230,180
730 DATA 208,2,230,181,165,180,197,196
740 DATA 208,231,165,181,197,197,208,225
750 DATA 24,165,200,101,187,133,206,165
760 DATA 201,105,0,133,207,160,0,165
770 DATA 189,208,19,177,204,209,206,48
780 DATA 84,200,196,188,144,245,176,17
790 DATA 208,152,176,152,144,152,177,206
800 DATA 209,204,48,65,200,196,188,144
810 DATA 245,160,0,169,4,133,181,169
820 DATA 0,133,180,177,198,145,180,200
830 DATA 196,184,208,247,160,0,177,200
840 DATA 145,198,200,196,184,208,247,160
850 DATA 0,177,180,145,200,200,196,184
860 DATA 208,247,165,194,56,229,190,133
870 DATA 194,165,195,229,191,133,195,144
880 DATA 4,5,194,208,179,230,192,208
890 DATA 2,230,193,165,202,197,192,165
900 DATA 203,229,193,176,165,144,165
```

Keyboard Trickery

On the Atari, like most other home computers, the primary interface between you and the computer is the keyboard. There are many tricks to using the keyboard interface more efficiently. In this chapter we will discuss many tricks of the trade and show you how to make your programs easier to use and more professional in their operation.

Avoiding Operator Crashes

The most annoying thing I have noticed in some commercial programs is that they have this nasty tendency to crash if you hit the wrong key. For example, the expected input is a number between one and nine, and you accidentally hit a "Q" or some other alpha character. In a well written program this mistake causes no harm, but in many so-called "professional" programs the result is a system, or at least, a program crash. All user inputs should be anticipated, and the program should not have a nervous breakdown just because you hit the wrong key. Normally, when most professional programmers write a program, they fully buffer all user inputs. This means that if the program asks for a specific input, such as a number from one to nine, the only inputs the program will act on are those numbers, and the rest of the keyboard is locked out. I strongly suggest that you adopt this policy in all of your programs, too. The routines in this chapter will be of help in doing this.

The Single Key Input Routine

I use this neat little routine in just about every BASIC program I write. You will find that it provides quite a programming convenience when you want to use a single key to answer a prompt or a question displayed on the screen. Subroutine KEY.LST simply tells the computer to wait for the operator to press a key on the keyboard. Upon return from the subroutine, you will have the ATASCII value of the character, corresponding to the key that was pressed, stored in KEY. Here's the subroutine:

Figure 11.1 — *KEY.LST*

```
20440 REM KEY.LST
20441 OPEN #6,4,0,"K:"
20442 GET #6,KEY
20443 REM PUT SPECIAL EXIT #1 HERE
20444 REM PUT SPECIAL EXIT #2 HERE
20445 CLOSE #6:RETURN
```

Essentially, this routine opens the keyboard as a "device", sort of like you might open a printer or disk drive for special I/O. In this particular routine the device number is "6". If the program you want to put this routine into is already using this device number for something else, you can change it to "5", "4" or some other legal number. You should avoid using device zero or seven since the operating system uses them, and the results could become unpredictable. The screen editor uses device zero. Device seven is used by LIST, LOAD, PRINT and RUN. CAUTION: always CLOSE a device when you are through with it.

When this routine is called, the ATASCII code for the key you hit is stored in the variable KEY. Any special exit conditions must test KEY against the proper ATASCII codes. I will give you a few examples a little later to help make this part clearer.

When you want the operator to press a single key, just GOSUB 20440. I use this routine in:

1. Menu programs, where I want the operator to select a program or subroutine by pressing a number or letter key without having to press the "RETURN" key.

2. Applications where a message or data is displayed on the screen and I want the operator to press "RETURN" to continue.

3. Applications where I want the operator to give a simple one-key response.

The advantages of the single-key input routine are:

1. You don't have to clutter your program logic with a number of two-or-more line routines to accept a single key entry. This saves you memory.

2. Your video display is not disturbed (as it could be with INPUT). Nothing is printed on the screen until your program logic has had a chance to analyze the contents of KEY. Inadvertent use of the control keys can't destroy your screen display.

3. You provide more convenience and fewer key strokes for the person using your program.

The menu routine shown in the next section is an example of one way that you can use the single-key subroutine.

Quick and Easy Menu Routines

A menu routine is a video display module that gives you a list of alternative functions to perform and the ability to select one of those functions (usually by entering a letter or a number). I've included a couple of sample menu routines to illustrate a few of the tricks in program design. Here is the menu to be displayed:

Figure 11.2 — *Sample Menu*

```
    SELECT A CHANGE OPTION
[1] CHANGE A PLAYER'S NAME
[2] DELETE A PLAYER
[3] CHANGE THE BONUS FACTOR
[4] SUBTRACT BONUS FROM A PLAYER
[5] RESTORE PREVIOUS DESTINATION
[6] START A NEW GAME
[7] RESURRECT A PLAYER
```

The selection of an item from a menu can be done by keyboard, paddle or joystick inputs, depending upon your application. The following sections will illustrate two of these input options. Keyboard input is demonstrated in MENU1.LST, and paddle input is demonstrated in MENU2.LST. I won't show an example for joystick inputs since they are very similar to paddle inputs in concept.

Keyboard Menus

In MENU1.LST, the PRINT CHR$(125) simply clears the screen so we won't have a messed up display. We then display the various options and their associated code numbers to prompt the user to select one of them.

Figure 11.3 — *MENU1.LST – Sample Menu Subroutine*

```
20451 PRINT CHR$(125):POKE 752,1
20452 PRINT:PRINT"       SELECT A CHANGE OPTION"
20453 POSITION 4,5:PRINT"[1]    CHANGE A PLAYER'S NAME"
20454 POSITION 4,7:PRINT"[2]    DELETE A PLAYER"
20455 POSITION 4,9:PRINT"[3]    CHANGE THE BONUS FACTOR"
20456 POSITION 4,11:PRINT"[4]    SUBTRACT BONUS FROM A PLAYER"
20457 POSITION 4,13:PRINT"[5]    RESTORE PREVIOUS DESTINATION"
20458 POSITION 4,15:PRINT"[6]    START A NEW GAME"
20459 POSITION 4,17:PRINT"[7]    RESURRECT A PLAYER"
20461 GOSUB 20440:IF KEY=155 THEN 20464
20462 IF KEY<49 OR KEY>55 THEN 20461
20463 ON KEY-48 GOSUB 1000,2000,3000,4000,5000,6000,7000
20464 RETURN
```

In this particular case, we are saying that the specialized subroutines referred to in the menu are located at line numbers 1000, 2000, 3000, 4000, 5000, 6000 and 7000. You could put them somewhere else if you so desired. If you want to use GOTO instead of GOSUB in line 20463, be sure to execute a POP command before leaving the subroutine. Everytime you go into a subroutine, the computer saves the RETURN address on the STACK. If you use an exit from the subroutine other than a RETURN, that address is left on the STACK. If you do this very often, the STACK can become full of useless addresses, and your program can crash with an ERROR 10. Executing a POP command before leaving the subroutine removes the unwanted return address from the stack.

The parameters used in this menu routine can easily be changed to work in whatever program you are writing. You can use the numbers I used, or you can select a different set of input codes by referring to the ATASCII keycode chart in the back of this book and changing the IF-THEN statement in line 20462. The line numbers referred to in line 20463 would then become the line numbers of your special routines.

I recommend that you always leave the user an easy exit from the subroutine that performs none of the functions, just in case he got into the menu by mistake. In this menu routine, that easy exit is hitting the "RETURN" key. If the key value returned by the routine is equal to a "RETURN" (155), we assume that the user wanted none of the options, and we return to the main calling routine without performing any of the possible options. The keycodes for the numbers 1–7 are 49–55 (see the ATASCII keycode list). If the keycode is outside of this range, we assume that an incorrect key was pressed by mistake and ignore it. Once we have gotten a valid input from the user, we execute a GOSUB to the selected routine before returning to the main program. To change the key to some other one, use the appropriate code from the keycode table.

You can also dress up the menu by using colors, reverse video or special bars and lines, but that is an embellishment I will leave up to you for now. The routines in the next chapter should be of some help to you in this area.

Note that each of the options is enclosed by brackets. A consistent use of brackets in this way will make things easier for the user of your program since he will tend to automatically assume that he must input something anytime he sees the brackets. If you use this technique, the brackets should also appear in any documentation that you write for the program.

Menus are much easier to understand if they have a distinctive title. In this particular case, I combined the title with the prompt for the user to enter a number. If you want to have a title that is separate from the prompt, you can simply insert a new line between 20459 and 20461 that asks for a user input. Line 20460 was deliberately left out so that you could do this without having to renumber the subroutine.

One thing that I also do in many of my programs is to set a TRAP that will go to the main program's control menu in case some unforeseen error does come up. For example, I might have forgotten to account for the printer being OFF. Normally this could cause a fatal execution error. Most of the time I would have a special error check in the print routine, but my failsafe TRAP would catch it if for some reason I forgot to put one there.

Paddle Driven Menus

The general menu philosophy we discussed in the last section really applies to all types of menus, so I will only describe the differences between keyboard and paddle menu control in this section. Figure 11.4 Shows you an example of a typical paddle driven menu.

Figure 11.4 — *MENU2.LST – A Paddle Driven Menu*

```
20470 REM MENU2.LST - A PADDLE DRIVEN MENU
20471 PRINT CHR$(125):POKE 752,1:PRINT:
      PRINT"       SELECT A CHANGE OPTION"
20472 POSITION 4,5:PRINT "[ ]    CHANGE A PLAYER'S NAME"
20473 POSITION 4,7:PRINT "[ ]    DELETE A PLAYER"
20474 POSITION 4,9:PRINT "[ ]    CHANGE THE BONUS FACTOR"
20475 POSITION 4,11:PRINT "[ ]    SUBTRACT BONUS FROM A PLAYER"
20476 POSITION 4,13:PRINT "[ ]    RESTORE PREVIOUS DESTINATION"
20477 POSITION 4,15:PRINT "[ ]    START A NEW GAME"
20478 POSITION 4,17:PRINT "[ ]    RESURRECT A PLAYER"
20479 POSITION 4,22:PRINT "[ ]    ESCAPE FROM THIS ROUTINE"
20480 FOR N=1 TO 7:POSITION 5,2*N+3:PRINT " ":
      NEXT N:POSITION 5,22:PRINT " "
20481 IF PADDLE(0)<26 THEN POSITION 5,5:OPTION=1:GOTO 20489
20482 IF PADDLE(0)<51 THEN POSITION 5,7:OPTION=2:GOTO 20489
20483 IF PADDLE(0)<76 THEN POSITION 5,9:OPTION=3:GOTO 20489
20484 IF PADDLE(0)<101 THEN POSITION 5,11:OPTION=4:GOTO 20489
20485 IF PADDLE(0)<126 THEN POSITION 5,13:OPTION=5:GOTO 20489
20486 IF PADDLE(0)<151 THEN POSITION 5,15:OPTION=6:GOTO 20489
20487 IF PADDLE(0)<210 THEN POSITION 5,17:OPTION=7:GOTO 20489
20488 POSITION 5,22:OPTION=155
```

```
20489 PRINT "*":IF PTRIG(0) THEN 20480
20490 IF OPTION=155 THEN 20492
20491 ON OPTION GOSUB 1000,2000,3000,4000,5000,6000,7000
20492 RETURN
```

This routine displays the menu options and puts a flashing asterisk in the "box" to the left of an option. The position of this "cursor" will move from box to box as you turn the control of your paddle. If you press the trigger, the asterisk will stop flashing, and the program will go to the selected routine. Note the new line in the menu. We now have to call out the escape option explicitly. The keyboard menu assumed that the user knew (from the instruction manual) the escape command.

Personally, I do not like to use this type of menu routine because it is actually slower than a keyboard menu if there are more than three or four options. One reason for the slow down is the accuracy with which you can set a paddle control. If there are only a few options, the range of 228 can be divided into large easily controlled blocks for the IF-THEN statements. When you have a lot of options in the menu, these blocks must be much smaller and are therfore much more difficult to select by turning the paddle. If you use either paddle or joystick menus, I strongly recommend that you limit any single menu to no more than four options.

Using the Function Keys to Better Advantage

The Atari home computers have three built-in function keys labeled "OPTION", "SELECT" and "START". You will probably have already noticed that I like to use these keys a lot when soliciting a response from the operator. This section will discuss these keys in more detail and show you how to use them in your own programs for simple inputs or even a special variation of the keyboard menu.

The key to using these function keys is a single, special memory location. The only way to tell if one of these keys has been pressed is to test on 53279. This opens some interesting possibilities. Normally, you test 764 to see if a particular key has been pressed, but 764 is not affected by pressing one of the function keys. This means that you can have a program that uses the main keyboard for its normal processing and maintains an interrupt system based upon whether or not a function key has been pressed. The Atari Word Processor is an excellent example of such a program. I have also noticed that many games use the function keys in this way.

The FUNKEY.LST routine is a simple way to check the state of the function keys. In some ways they are like a simple application of BITMAP.LST. If none of the function keys are pressed, the value stored in 53279 is seven. Each of the function keys clear a particular bit in the number stored at 53279 when they are pressed. The START key clears bit zero; SELECT clears bit one, and OPTION clears bit two. Yes, this means that you can also test for combinations of the function keys. If all three keys are pressed, the value stored in 53279 would be 00000000 or a decimal value of zero. The following chart shows you all of the possible key combinations and their effect on 53279.

Figure 11.5 — *FUNKEY.LST – Function Key Test Routine*

```
20500 REM FUNKEY.LST
20501 IF PEEK(53279)=6 THEN 1000:REM START
```

```
20502 IF PEEK(53279)=5 THEN 2000:REM SELECT
20503 IF PEEK(53279)=3 THEN 3000:REM OPTION
20504 GO TO 20501
```

The starting line numbers of the routine that the key is to invoke are 1000, 2000 and 3000.

Figure 11.6 — *Function Key Value Chart*

KEYS PRESSED	PEEK(53279)	BINARY CODE
NONE	7	00000111
START	6	00000110
SELECT	5	00000101
START & SELECT	4	00000100
OPTION	3	00000011
START & OPTION	2	00000010
SELECT & OPTION	1	00000001
ALL THREE	0	00000000

Figure 11.7 — *MENU3.LST – A Function Key Menu*

```
20510 REM MENU3.LST
20511 PRINT CHR$(125):POKE 752,1:POSITION 2,12:
      PRINT "PRESS OPTION FOR DISK/TAPE INPUT":TAPE=0
20512 PRINT "PRESS SELECT FOR OBJECT/MERGE FILE":
      MERGE=0:PRINT "PRESS  START TO  CONTINUE"
20513 IF PEEK(53279)>5 THEN 20520
20514 IF PEEK(53279)=3 THEN TAPE= NOT TAPE:
      FOR X=1 TO 20:
      NEXT X
20515 IF PEEK(53279)=5 THEN MERGE= NOT MERGE:
      FOR X=1 TO 20:NEXT X
20516 IF TAPE THEN POSITION 19,12:PRINT "DISK/TAPE"
20517 IF NOT TAPE THEN POSITION 19,12:PRINT "DISK/TAPE"
20518 IF MERGE THEN POSITION 19,13:PRINT "OBJECT/MERGE"
20519 IF NOT MERGE THEN POSITION 19,13:PRINT "OBJECT/MERGE"
20520 IF PEEK(53279)=6 THEN 20522
20521 GO TO 20513
20522 IF TAPE AND MERGE THEN 1000
20523 IF TAPE AND NOT MERGE THEN 2000
20524 IF MERGE THEN 3000
20525 GOTO 4000
```

```
Where the following routines are:
LINE 1ØØØ = ENTER a BASIC file from cassette
LINE 2ØØØ = GET an object file from cassette
LINE 3ØØØ = ENTER a BASIC file from disk
LINE 4ØØØ = GET an object file from disk
```

NOTE: See the listing of DATAPAK.BAS for more info.

There are many possible ways to use the function keys to better advantage. The examples in this section are only a starting point. With the information we have discussed here, you should be able to design your own uses for the Atari function keys.

Disabling the BREAK Key

Some operations, such as disk I/O, can run into catastrophic failures if the BREAK key is pressed at the wrong time. This is also true for many of those programs that play around with the Display List. One solution to this problem is to disable the BREAK key so the user can not accidentally, or deliberately, press it at the wrong time. The routine to disable the BREAK key is:

Figure 11.8 — *BREAKLOK.LST – Lock Out the BREAK Key*

```
2Ø53Ø BREAKLOK.LST - DISABLE THE BREAK KEY
2Ø531 CODE=PEEK(16)
2Ø532 IF CODE>127 THEN CODE=CODE-128
2Ø533 POKE 16,CODE
2Ø534 POKE 53774,CODE
2Ø535 RETURN
```

Be sure that you don't use this routine until you have saved your program if you are in the midst of debugging it. Once you have your program debugged, then you can safely put this routine at the top of your program where it will be executed as soon as the program is RUN. To unlock the BREAK key again, use POKE 16,192 and POKE 53774,247.

Repeating Keys and Combinations

Did you ever want to repeat a function as long as you were holding a key down? Here's a subroutine that will help you:

Figure 11.9 — *REPEAT.LST – Infinitely Repeat a Function*

```
2Ø54Ø REM REPEAT.LST
2Ø541 IF PEEK(764)=TEST THEN
      GOSUB FUNCTION:GO TO 2Ø541
2Ø542 POKE 764,255:RETURN
```

In this subroutine, the variable TEST should be previously set to the *keyboard* (not ATASCII) code of the key you wish to test for. The value of the variable FUNCTION is assumed to have been previously set to the line number of the function that you want to have repeated while the test key is depressed. The keyboard codes are listed in the back of this book, but if you want to have them displayed on the screen, then type in this short routine and run it to display the keyboard keycodes:

```
100 IF PEEK (764)=255 THEN 100
110 PRINT PEEK (764)
120 POKE 764,255: GO TO 100
```

Now press any key or key combination and notice the number that is displayed. This value will correspond to one of the keycodes given in Appendix B. To set up repeat keys in your programs, simply test on PEEK (764) for the proper keycode and direct the program to the desired subroutine!

Special Keys And Their Codes

Here is a list of the more important special keys found on the keyboard and the effect you will get by printing the CHR$ function for the ATASCII code for the key:

Figure 11.10 — *Special Keys and Their Character Codes*

KEY	CHR$ CODE	EFFECT GENERATED
SHIFT-CLEAR	125	Clear the screen
SHIFT-INSERT	157	Insert a line
SHIFT-DELETE	156	Delete a line
RETURN	155	End-of-line
BACKSPACE	126	Delete character to left
CONTROL-UP ARROW	28	Move cursor up one line
CONTROL-DOWN ARROW	29	Move cursor down one line
CONTROL-LEFT ARROW	30	Move cursor left one spot
CONTROL-RIGHT ARROW	31	Move cursor right one spot
CONTROL-CLEAR	125	Clear the screen
CONTROL-2	253	Activate keyboard buzzer
CONTROL-INSERT	255	Insert one character here
CONTROL-DELETE	254	Delete current character

I only listed those codes that I have found useful for creating special effects while a program is running. For example, the "move cursor" codes can be combined with the "delete character" code to eliminate a faulty input from a formatted input. Formatted input routines will be covered in the next chapter. The best way to learn how to use these special codes effectively is to try them out. One odd thing is that there is apparently no difference between a CONTROL-CLEAR and a SHIFT-CLEAR.

Controlled Keyboard Input Routines

The routines in this section will work very well with the formatted input routines in the next chapter. In this section, we will concentrate on how to get multi-key inputs from the keyboard without using the INPUT command.

Controlled String Input

Many applications require the user to input a string of characters, such as a person's name, in response to a prompt. The routine in Figure 11.9 illustrates a simple, but effective technique for this purpose:

Figure 11.11 — *INKEY1.LST – Controlled String Input*

```
20550 REM INKEY1.LST - CONTROLLED STRING INPUT
20551 OPEN #6,4,0,"K:":SIZE=9:FOR X=1 TO SIZE
20552 GET #6,KEY:IF KEY=155 THEN POP :GOTO 20559
20553 IF KEY<48 OR KEY>122 THEN 20552
.
. any other conditions would go here
.
20558 PRINT CHR$(KEY);:RESPONSE$(X,X)=CHR$(KEY):NEXT X
20559 CLOSE #6:RETURN
```

The allowed length of the input string is set by the variable SIZE. If you want to limit the legal characters to some other set, you will need to change the values in line 20553. Note that the string may be shorter than SIZE. The input sequence is terminated by either reaching the maximum string length or by pressing a RETURN.

Controlled Numeric Input

The single-key input routine we discussed at the beginning of this chapter is fine where a single key input is sufficient, but many applications require two or more keys in response. For example, a program might need a date or dollar amount entered. The single-key input routine is not suitable for such cases without some modification. The routine listed below, INKEY2.LST, is one solution to this problem.

Figure 11.12 — *INKEY2.LST – Controlled Numeric Input*

```
20560 REM INKEY2.LST - CONTROLLED NUMERIC INPUT
20561 SIGN=1:NUMBER=0:SIZE=3:OPEN #6,4,0,"K:":
      FOR X=1 TO SIZE
20562 GET #6,KEY:IF KEY=155 THEN POP :GOTO 20569
20563 IF KEY=45 AND SIGN=1 THEN SIGN=-1:
      PRINT"-";:GOTO 20562
20564 IF KEY<48 OR KEY>57 THEN 20562
.
. additional conditions would go here
.
20568 PRINT CHR$(KEY);:
      NUMBER=10*NUMBER+VAL(CHR$(KEY)):
      NEXT X
20569 NUMBER=SIGN*NUMBER:CLOSE #6:RETURN
```

This routine is very similar to INKEY1.LST in the way the characters are grabbed one at a time. As before, the length of the input field is set by the variable SIZE. In this particular case, SIZE is set to three. This count does not include the space used by the minus sign. So, any positive or negative three digit number could be entered by this routine. We will use this routine in the next chapter along with special video prompts to achieve what I have been calling "controlled input".

Controlled Data Entry

You could easily spend 75 per cent or more of your programming time trying to develop an attractive, easy-to-use and water-tight data entry system. Once you have gotten good clean information in the computer, processing the information and printing it out is comparatively easy.

A good menu or other data entry routine should always provide prompts that make it clear what kind of input is required. The trade-off in using prompts is to supply enough prompts for the new user of your program while at the same time limiting the prompts so they will not slow down the experienced user.

You also need input validation that will ignore bad inputs instead of crashing the system or halting program execution. If the inputs are processed properly by the input routine, your job of processing the information becomes much simpler. In a really good (i.e., professional) program, each input is controlled so that only those keys which are considered valid will have any effect at all. In situations like this, you must avoid the screen destroying effects of the CLEAR key and the BREAK key.

Finally, you need to provide simple and consistent ways for the operator to correct entry errors. The operator should always be allowed to back up and correct the previous entry. This is sometimes difficult to achieve, but if you ignore this requirement, you are programming automatic operator frustration into your programs!

This chapter will take many of the techniques that we have discussed in previous chapters and show you how to combine them with a few new video techniques to create good, user friendly menus and other video displays. The demonstration program at the end of this chapter should be especially useful to you.

Video Formatting

Video formatting, in the sense we will use it here, refers to those techniques that you might use to set up special data entry fields. I think of such routines as being in three major categories. The first category is "positional input fields." The second one, "special input fields," is a category by itself, but can be used quite effectively with the first category. The third category, "scrolled inputs," is a powerful technique that you will find useful in many different applications.

Positional Input Fields

Most of the time, when setting up a program, there will arise a need for asking the user to enter numbers or alpha characters of limited length. We discussed in the last chapter how you can create a routine to actually get those inputs, but so far we have not really discussed the impact of those inputs on the video display. The easiest, and most useless solution is to get the inputs without echoing them on the screen. This leads to instant confusion. A better solution is to not only control the nature of the allowed inputs, but to also control the possible results on the video display.

The following routines are examples of two ways you can control the video display during an input routine. FIELDB.LST sets up a blank field of length SIZE at a particular location on the screen. When you use this routine in conjunction with the inkey routines in the last chapter, you can not only restrict the number of characters the user can input, but you can also make sure that any characters the user may enter will be printed only where you want them to be. In this routine, the position on the screen is specified by the two variables X and Y. We will give you a working example later in this chapter.

Figure 12.1 — *FIELDB.LST – Set Up a Blank Field*

```
2Ø58Ø REM FIELDI.LST - INVERSE INPUT FIELD
2Ø581 REM DIM INVERSE$(4Ø) ELSEWHERE
2Ø582 INVERSE$(1)=" ■ ":INVERSE$(4Ø)=" ■ ": INVERSE$(2)=INVERSE$
2Ø583 X=2:Y=12:SIZE=9 POKE 752,1
2Ø584 POSITION X,Y
2Ø585 PRINT INVERSE$(1,SIZE)
2Ø586 POSITION X,Y
2Ø587 REM GOSUB INPUT ROUTINE
2Ø588 REM GOSUB ERROR CHECK ROUTINE
2Ø589 RETURN
```

The FIELDI.LST routine is very similar to the other routine. The primary difference is that the input field is highlighted in inverse video. The variables and operation of the routine are the same. Be sure to replace any field locations with inverse blanks during error corrections and to eliminate any unused inverse field locations from the field when input is finished.

Figure 12.2 — *FIELDI.LST – Set Up an Inverse Field*

```
2Ø58Ø REM FIELDI.LST - SET UP INVERSE INPUT FIELDS
2Ø581 REM POSITION OF FIELD IS SET BY X&Y
2Ø582 X=2:Y=12:SIZE=9:POKE 752,1
2Ø583 POSITION X,Y:FOR Z=1 TO SIZE:PRINT "■";:NEXT Z
2Ø584 POSITION X,Y
2Ø585 REM GOSUB INPUT ROUTINE
2Ø856 REM GOSUB ERROR CHECK ROUTINE
2Ø857 RETURN
```

Special Input Fields

The techniques described in the previous section can be tailored for special input requirements. The more common special input fields are for money, dates, and time values. The routines in this section, while not covering everthing, will show you how to handle these particular special input fields.

The routine FDOLLARS.LST sets up a limited video field for the general input of dollar figures. It is intended to be used with the INKEY routines we discussed in the last chapter. The routine first prints a "$" sign and a decimal point in the screen position specified by the variables X and Y. The number of digits to the left of the decimal place is controlled by the variable SIZE. Once the format has been printed on the screen, you are expected to use the proper calls to the INKEY routines. Any error correction routine should take the field sizes into account.

Figure 12.3 — *FDOLLARS.LST – Special Fields Dollars & Cents*

```
20590 REM FDOLLARS.LST - SPECIAL FIELDS DOLLARS & CENTS
20591 X=2:Y=12:SIZE=4:POKE 752,1
20592 POSITION X,Y:PRINT "$":POSITION X+SIZE+1,Y:PRINT "."
20593 POSITION X+1,Y:FOR Z=1 TO SIZE:PRINT " ";:
      NEXT Z:POSITION X+SIZE+2,Y:PRINT "  "
20594 POSITION X+1,Y
20595 REM GOSUB DOLLAR INPUT ROUTINE
20596 POSITION X+SIZE+2,Y:SIZE=2
20597 REM GOSUB CENTS INPUT ROUTINE
20598 REM GOSUB ERROR CHECK ROUTINE
20599 RETURN
```

The next two subroutines are very similar to each other in operation. FDATES.LST sets up a display field in the standard MM/DD/YY format. You then use the INKEY2.LST routine to grab three two-digit numbers. FTIMES.LST sets up an HH:MM time display format; then all you need to do is to grab two two-digit numbers.

Figure 12.4 — *FDATES.LST – Special Fields Dates*

```
20600 REM FDATES.LST - SPECIAL FIELDS DATES
20601 X=2:Y=20:SIZE=2:POKE 752,1
20602 POSITION X,Y:PRINT "  /  /  "
20603 POSITION X,Y
20604 REM GOSUB TWO DIGIT INPUT
20605 POSITION X+3,Y
20606 REM GOSUB TWO DIGIT INPUT
20607 POSITION X+6,Y
20608 REM GOSUB TWO DIGIT INPUT
20609 REM GOSUB ERROR CHECK ROUTINE
20610 RETURN
```

Figure 12.5 — *FTIMES.LST – Special Fields Clock Time*

```
20620 FTIMES.LST - SPECIAL FIELDS CLOCK TIME
20621 X=2:Y=20:SIZE=2:POKE 752,1
20622 POSITION X,Y:PRINT "  :  "
20623 POSITION X,Y
20624 REM GOSUB TWO DIGIT INPUT
20625 POSITION X+3,Y
20626 REM GOSUB TWO DIGIT INPUT
20627 REM GOSUB ERROR CHECK ROUTINE
20628 RETURN
```

The actual values of X and Y should be set before you go into FDATES.LST or FTIMES.LST. You should probably delete those variables from the actual subroutines before using them. Do not, however, alter the value of the variable SIZE within the routines, or you will mess up the format operation.

Scrolling Window Inputs

Most of you have seen those programs that use a high-res graphics display in the top 20 lines of the display and use the bottom four display lines to give you prompts and solicit responses from you. Did you realize that you can have the same kind of independent scrolling window in GRAPHICS 0? It is as easy as POKEing a number. The following routine, FSCROLL.LST, contains the codes to set up or remove such a window in GRAPHICS mode 0:

Figure 12.6 — *FSCROLL.LST – Special Fields Scrolling Window*

```
20630 REM FSCROLL.LST - SPECIAL FIELDS SCROLLING WINDOW
20631 REM RESTRICT INPUTS TO LAST 4 LINES OF SCREEN
20632 POKE 703,4
20634 RETURN
20635 REM RESTORE NORMAL DISPLAY
20636 POKE 703,24
20637 RETURN
```

The window thus set up may be used exactly like a normal full screen without affecting the top 20 display lines. All of your normal screen controls will affect only the window. If you were to LIST a program, it would be listed only in the last four lines of the screen and would scroll off the top of your four-line display like any other normal display would. I will show you a working example of this technique at the end of this chapter.

The only tricky part of using the scrolling window is how to make changes in the top part of the screen while you are in the window mode. There are two general solutions. The first is to simply POKE any changes into the proper part of the screen buffer. Thus a POKE 40520,104 will cause a lower case "h" to appear on the screen regardless of whether you are in the window mode or not. This method is sometimes awkward, however, so Atari built in a much simpler solution.

If you do not use a GRAPHICS 0 command anywhere in the program, then the above condition will prevail when you go into window mode. All screen controls will affect only the window area of the display. However, if you take care to use a GRAPHICS 0 command before going into the window mode, things change somewhat. First, and most noticable, is that your screen controls once again affect the whole 24 line screen.

The second effect is a little more subtle. When a GRAPHICS command is executed, the operating system automatically OPENs device number 6 for output and defines the device to be the video display. (A full screen graphics mode without a text window will have the entire screen opened for output.) The net result of going into window mode at this stage is that you can use a PRINT statement to print information in the window area of a mode zero screen and still use a PRINT #6 statement to print directly to the top portion of the screen!

I have used this technique to display a customized menu in the top 20 lines of the screen while using the window as a work area for displaying prompts and soliciting user inputs that might do such things as save data to disk. You can also use this technique to get new data to display in the static area. For example, you display a menu, go into window mode, and ask the user to select a program option. This option, in turn, might clear the top area and display a new sub-menu for the execution of the selected option. The Atari Word Processor uses this kind of nested menu technique, although it puts the window (non-scrolling) at the top four lines instead of the bottom four lines of the video display. There are many possible variations you could use in your next program. The window mode can be set up by using a GOSUB 20632. When you are finished with it and want to restore the screen to normal, you can do this by a GOSUB 20635.

Error Handling

Error handling, in the end, is probably the single most important feature in distinguishing between an amateur program and a truly professional one. The professional programmer takes all possible operations into account and buffers the program so a bad user input or some other mistake will not cause the program or the system to crash. This is sometimes easier said than done, since it is difficult to make sure that every possible error has been anticipated.

Errors which occur during program execution are usually controlled by an "error detection" routine coupled with an "error correction" routine. Error detection routines are used to intercept and/or prevent a run-time error. If an error is found, then an error correction routine allows the mistake to be corrected.

Error Detection Techniques

Error detection routines generally consist of TRAPs or filters or a combination of the two. Both of these error detectors have their strengths and weaknesses. TRAPs are very well suited for intercepting general I/O errors and as a general catch-all error trap. Filters, which usually consist of one or more IF-THEN statements, are more suitable for screening user inputs and preventing other types of errors.

TRAP commands, as we discussed previously, are most frquently used to intercept errors after they have occurred and redirect the results of the error to an error correction routine. For example, you might set a TRAP that will be tripped if an error occurs during disk I/O.

This TRAP could detect an attempt to write to a disk that was not even turned ON and would redirect program control to an error correction routine that tells you to turn your disk drive ON. A really common error is trying to dump something to a printer without first turning the printer ON. By using TRAPs, you could prevent these kinds of mistakes from crashing your program.

Error Correction Techniques

There are at least as many error correction techniques as there are errors, if not more. An error correction routine can range from something as simple as a single IF-THEN statement to a whole program which is dedicated to correcting a data base. We will leave the latter to the more energetic of you. The discussion here will limit itself to those simple correction techniques that you will find useful in your every day programming tasks.

The simplest of error correction techniques is also an error detection routine. It consists of one or more IF-THEN statements that examine a user input or some other variable and compare it to a specific value or a range of values. If the variable does not pass this test, control is tranferred back to the input routine or to some other routine that tells the user that a mistake has occurred. Usually the second method will also tell the user what the error was and ask for a corrective action.

You can also use a routine that takes its own corrective action. For example, you are using a menu routine that has just asked for you to input your name. As you are entering the fourth letter, you find that you have entered "Kohn" instead of "John". Assuming that your correct name is Johnny, some programs might prevent you from correcting the mistake and force you to keep right on going. On the other hand, many programs will let you alter the input by pressing the BACK SPACE key to the error and then re-entering the name. There is nothing wrong with either method as long as you have some means of correcting the error.

An in-line error correction routine checks each entry and acts on the single key by accepting it as new valid input, ignoring the input as invalid, or recognizing the input as an instruction to back up to a previous input. The simplest way to do this using the KEY routine is to compare the value of RESPONSE to 126 (BACK SPACE) along with the other normal IF-THEN comparisons. If a BACK SPACE is detected, then the program moves the cursor back one position in the input field and erases that input. If you look at the example program at the end of this chapter, you will see this technique illustrated. Note also that the BACK SPACE is not allowed to move the cursor back any further than the beginning of the input field. This technique is probably the one most commonly used.

The second most popular error correction technique is to have a separate routine in the program that allows the user to correct any of a number of possible errors. Look again at the menu routines in the last chapter. These are examples of a menu for just such an error correction routine. For example, the program might ask you for a whole list of various inputs one right after the other. In this particular case, the BACK SPACE routine was also used, but it is of little help after you have entered the last character in an input field. The main program menu listed "CORRECT ERRORS" as one option. If you select this option from the main menu, the detailed error correction menu is displayed. In the menus from the last chapter (as I actually used them), if you selected "CHANGE THE NAME OF A PLAYER", a little routine would be called up which would allow you to select the name you wanted to change and then ask you for the correct name. The new name could then be entered and would be stored in all of the arrays as appropriate.

Detailed error correction routines of this nature can be very simple, like the one I used, or they can be so large and complex that they end up being programs in their own right. The right size and complexity for your program is sometimes a tough decision. All I can suggest is that you sit back and ask yourself what you would want in a similar program if you were buying it. You might be surprised at the answer.

We will close this brief discussion of error correction techniques with this little sermon: NEVER write a program you intend to sell without making sure that all user inputs are *fully* buffered. There is nothing more aggravating to me than spending my hard earned cash on a new program that crashes the first time I try to use it. I even bought a program one time that not only failed to load properly, but it had some kind of protection scheme that caused it to promptly erase itself from the disk just because my drive speed was faulty! Moral of this story: always keep your customer in mind.

Attracting and Distracting the Operator

A program should be easy to use, have adequate error checking and error correction options, and perform the desired functions smoothly and swiftly. We have already discussed many of these attributes of a good program. The most subtle and difficult to master is the art of user prompting. A good user prompt should be clear enough for the novice user of your program without being so slow that it impedes the experienced user. One easy way to accomplish this is to have a special option which will eliminate long prompts for those people who don't want or need them. A perfect example of this technique is the ZORK adventure game series which has three levels of prompts that are under user control.

A good prompt consists of more than a lone "?" sitting on the screen. Time after time I buy programs that use this simple prompt. When it pops up I have no idea whether I am supposed to input a number or a letter. In most cases, I guess wrong and the program crashes. When you want the user to input a number, the program should plainly and clearly ask for a numeric input. In most cases, it is also nice to give the range of valid numbers. Go back to Chapter Three and look at the prompts I used in the program CONVERT.BAS. You will quickly see how awkward the program would be to use if the prompt were nothing more than a "?".

Another kind of prompt that is popular is a buzzer or bell to get the operator's attention. Once again, you have to be careful to avoid over using such a thing. Have you ever played a war game called EASTERN FRONT? It is a classic example of the graphics capability of the Atari computer, but it is a program with one very irritating feature. If you make an incorrect input, the program blatts this loud raspberry sound at you. This noise got on my nerves to the point where I simply put the program disk back in its box and left it there. That was over a year ago.

This brings me to my next point about program design. Prompts and audio cues should not be designed so they distract the user from the main purpose of the program. This principle applies to video prompts as well. Use a simple menu when you can and break the screen up into "information" and "input" sections. The window technique we just discussed is one approach. In the next chapter I will show you how to get four colors in a GRAPHICS 2 text display at the same time. That capability will allow you to design user friendly screen displays that will also dress up your programs.

One technique for attracting the eye of the user is to make use of flashing cursors or other flashing prompts. This can be illustrated by the following routine:

```
100 POKE 755,0
110 FOR X=1 TO 50:NEXT X
120 POKE 755,2
130 FOR X=1 TO 50:NEXT X
140 GOTO 100
```

This little routine POKEs the cursor control flag (755) with zero to turn the cursor OFF, waits a little while, POKEs the cursor control flag with two to turn the cursor back ON, waits a little while, and then repeats the whole process. While this is cute, it is not very practical since BASIC can't be doing anything else while it is flashing our little cursor. The solution is to employ a small machine language routine.

BLINK is a machine language program that sets up a vertical blank interrupt routine to turn the cursor ON and OFF while BASIC goes on about your business.

The time delay between changes is set at what I consider a comfortable pace. If you want to change the rate, stop the routine by POKEing a zero into 359 with POKE 359,0. Now that you have the routine stopped, POKE address 377 with a new number. The default value is eight. A larger number will slow the rate down, and a smaller one will speed it up. Once you have installed your new rate number, press SYSTEM RESET to activate the routine again.

There is an interesting side effect of this routine. More than just the cursor will be flashing. Any character that is in inverse video will also be flashing! I use this technique combined with the FIELDI.LST routine to cause the next user input field to flash. This can be a very catchy video technique, but don't get carried away with it to the point that you give the user eye strain.

Those of you out there who know a little something about the Atari are probably wondering why I didn't use the cursor inhibit flag (752) to flash the cursor without causing the inverse video to flash also. The answer is simple. A POKE to address 752 affects the cursor only if the cursor is moved after the POKE. I tried to use that address for my flashing cursor and ended up with a flashing cursor that jumped. I found this jittering to be very irritating after a while and ended up writing the routine you see in this book. If one of you readers come up with another solution to this problem, please write to me in care of this publisher and show me how you did it. Anyway, I have found that I seldom have any difficulty with the routine the way it is right now.

Figure 12.7 — *BLINK Assembled Source Listing*

```
            1ØØØ ;BLINK - CREATES BLINKING CURSOR
            1Ø1Ø ;
            1Ø2Ø ;
            1Ø3Ø ;
            1Ø4Ø ;THIS ROUTINE CREATES A BLINKING CURSOR
            1Ø5Ø ;
            1Ø6Ø ;SET $167(359 decimal)=Ø TO STOP FLASHING
            1Ø7Ø ;
            1Ø8Ø ;
            1Ø9Ø ;**********************************************************
            11ØØ ;
            111Ø ;SET UP OS POINTERS
            112Ø ;
ØØØ2        113Ø CASINI  =     $2         ;CASSETTE INIT VECTOR
ØØØ9        114Ø BOOTF   =     $9         ;BOOT MODE FLAG
ØØ42        115Ø CRITIC  =     $42        ;CRITICAL I/O FLAG
Ø222        116Ø VBLANK  =     $222       ;IMMEDIATE VBLANK VECTOR
Ø2F3        117Ø FLASHER =     $2F3       ;CURSOR CONTROL FLAG
159D        118Ø DUP     =     $159D      ;OS FLAG TO DETECT DUP.SYS
E45C        119Ø SETVBI  =     $E45C      ;SET-VBI VECTOR ENTRY
```

```
E45F          SYSVBI  =       $E45F           ;OS VBLANK SERVICE ROUTINE
              ;
              ;*******************************************************
              ;
              ;THE INIT ROUTINE AT $400 IS EXECUTED ONLY ONCE.
              ;THE MAIN ROUTINE IS STORED ON PAGE ONE.
              ;
0000                  *=      $400            ;THIS IS LATER OVER-WRITTEN
              ;
              ;SET UP PRIVATE INTERRUPT
              ;
0400 A509             LDA     BOOTF           ;IF A CASSETTE HAS BOOTED
0402 2902             AND     #2              ;THEN SAVE CASINI FOR LATER
0404 F00A             BEQ     INIT
              ;
0406 A602             LDX     CASINI          ;RE-VECTOR CASSETTE INIT
0408 A403             LDY     CASINI+1        ;TO INCLUDE OUR ROUTINE
040A 8E6E01           STX     DETOUR+1
040D 8C6F01           STY     DETOUR+2
0410 A968     INIT    LDA     #RESET&$FF
0412 8502             STA     CASINI
0414 A901             LDA     #RESET/256
0416 8503             STA     CASINI+1
0418 AD2202           LDA     VBLANK          ;DETOUR NORMAL HOUSEKEEPING
041B 8D8901           STA     EXIT+1
041E AD2302           LDA     VBLANK+1
0421 8D8A01           STA     EXIT+2
0424 A509             LDA     BOOTF
0426 0902             ORA     #2
0428 8509             STA     BOOTF
042A A201             LDX     #MAIN/256       ;POINT VBLANK TO OUR ROUTINE
042C A08B             LDY     #MAIN&$FF
042E A906             LDA     #6              ;USE IMMEDIATE VBI
0430 205CE4           JSR     SETVBI
0433 60               RTS
              ;
              ;*******************************************************
              ;
              ;THIS IS THE PART WE WANT TO PRESERVE
              ;
0434                  *=      $165            ;PROGRAM IS NOT RELOCATABLE
              ;
              ;SAVE SPACE FOR COUNTERS
              ;
0165 00       FLAG    .BYTE0                  ;0=TURN CURSOR OFF
0166 00       COUNTER .BYTE0                  ;HOW MANY DELAYS SO FAR
0167 08       LIMIT   .BYTE8                  ;THIS CONTROLS TIME DELAY
              ;
              ;
              ;THIS RESTORES OUR ROUTINE WHEN SYSTEM RESET IS PRESSED
              ;
```

```
0168 A9A9   1710 RESET   LDA    #$A9          ;SYSTEM RESET COMES HERE
016A 8D8101 1720         STA    PATCH
016D 208601 1730 DETOUR  JSR    NULL
0170 A900   1740         LDA    #0            ;RESTORE DEFAULT VALUES
0172 8D6601 1750         STA    COUNTER       ;TO TIME DELAY & COUNTER
0175 8D6501 1760         STA    FLAG
0178 A908   1770         LDA    #8
017A 8D6701 1780         STA    LIMIT
017D A201   1790         LDX    #MAIN/256     ;TELL VBI WHERE OUR ROUTINE IS
017F A08B   1800         LDY    #MAIN&$FF
0181 A906   1810 PATCH   LDA    #6            ;IMMEDIATE VBI
0183 205CE4 1820         JSR    SETVBI        ;SET THE GEARS IN MOTION
0186 60     1830 NULL    RTS
            1840 ;
            1850 ;THIS IS THE MAIN ROUTINE
            1860 ;
0187 68     1870 THAW    PLA                  ;RESTORE ACCUMULATOR
0188 4C8601 1880 EXIT    JMP    NULL          ;(NULL CHANGED DURING SETUP)
            1890 ;
            1900 ;VBLANK INTERRUPT COMES HERE
            1910 ;
018B 48     1920 MAIN    PHA                  ;SAVE ACCUMULATOR
            1930 ;
            1940 ;CHECK FOR CRITICAL I/O FLAG
            1950 ;
018C A542   1960         LDA    CRITIC        ;IF CRITIC IS SET,
018E D0F7   1970         BNE    THAW          ;ALL DONE, LET'S GO HOME
            1980 ;
0190 AD6701 1990         LDA    LIMIT         ;IF LIMIT=0 THEN EXIT
0193 F01E   2000         BEQ    ON
            2010 ;
            2020 ;TIMING LOOP
            2030 ;
0195 EE6601 2040         INC    COUNTER
0198 AD6601 2050         LDA    COUNTER
019B CD6701 2060         CMP    LIMIT         ;IF TIME<>LIMIT THEN EXIT
019E D0E7   2070         BNE    THAW
            2080 ;
01A0 A900   2090         LDA    #0            ;WHEN TIME LIMIT REACHED
01A2 8D6601 2100         STA    COUNTER       ;RESET COUNTER AND
01A5 AD6501 2110         LDA    FLAG          ;TOGGLE CURSOR FLAG
01A8 4901   2120         EOR    #1            ;FLAG=NOT FLAG
01AA 8D6501 2130         STA    FLAG
01AD D004   2140         BNE    ON
            2150 ;
01AF A900   2160 OFF     LDA    #0            ;COMMAND CURSOR OFF
01B1 F002   2170         BEQ    FLASH
            2180 ;
01B3 A902   2190 ON      LDA    #2            ;COMMAND CURSOR ON
01B5 8DF302 2200 FLASH   STA    FLASHER       ;EXECUTE COMMAND
01B8 18     2210         CLC
```

```
Ø1B9 9ØCC   222Ø        BCC     THAW
            223Ø ;
Ø1BB        224Ø        .END
```

Figure 12.8 — *BLINK – Blinking Cursor In BASIC*

```
1ØØ REM BLINK.BAS - FLASHING CURSOR
11Ø REM
12Ø REM THIS IS THE VBI SETUP ROUTINE
13Ø REM THAT IS TEMPORARILY STORED ON
14Ø REM PAGE FOUR.       $4ØØ (1Ø24)
15Ø DATA 165,9,41,2,24Ø,1Ø,166,2
16Ø DATA 164,3,142,11Ø,1,14Ø,111,1
17Ø DATA 169,1Ø4,133,2,169,1,133,3
18Ø DATA 173,34,2,141,137,1,173,35
19Ø DATA 2,141,138,1,165,9,9,2
2ØØ DATA 133,9,162,1,16Ø,139,169,6
21Ø DATA 32,92,228,1Ø4,96
22Ø REM NOTE THAT THE NUMBER '1Ø4'
23Ø REM IN LINE 21Ø IS ONLY IN THIS
24Ø REM BASIC VERSION. IT IS NOT IN
25Ø REM THE BINARY LOAD VERSION.
26Ø MLSTART=1Ø24
27Ø MLEND=1Ø76
28Ø FOR X=MLSTART TO MLEND
29Ø READ Y:
    POKE X,Y:
    NEXT X
3ØØ REM THIS IS THE MAIN ROUTINE.
31Ø REM IT IS STORED ON PAGE ONE.
32Ø DATA Ø,Ø,8,169,169,141,129,1
33Ø DATA 32,134,1,169,Ø,141,1Ø2,1
34Ø DATA 141,1Ø1,1,169,8,141,1Ø3,1
35Ø DATA 162,1,16Ø,139,169,6,32,92
36Ø DATA 228,96,1Ø4,76,134,1,72,165
37Ø DATA 66,2Ø8,247,173,1Ø3,1,24Ø,3Ø
38Ø DATA 238,1Ø2,1,173,1Ø2,1,2Ø5,1Ø3
39Ø DATA 1,2Ø8,231,169,Ø,141,1Ø2,1
4ØØ DATA 173,1Ø1,1,73,1,141,1Ø1,1
41Ø DATA 2Ø8,4,169,Ø,24Ø,2,169,2
42Ø DATA 141,243,2,24,144,2Ø4,
43Ø MLSTART=357
44Ø MLEND=442
45Ø FOR X=MLSTART TO MLEND
46Ø READ Y:
    POKE X,Y:
    NEXT X
47Ø REM NOW THAT WE HAVE THE ROUTINE
```

```
480 REM IN MEMORY, WE TURN IT ON BY
490 X=USR(1024)
500 END
```

Putting It All Together

O.K., up to this point we have discussed a number of professional program design techniques and laid a couple of philosophical sermons on you. Now let's put what we have discussed into practice. The following program, CONTROL.DEM, is a demonstration program that shows you how to tie all of these techniques together into a functioning program. CONTROL.DEM displays a sample "fill-in-the-blank" menu which asks you for your name and birthday and then demonstrates the scrolling window technique. As you try the program out, deliberately make a mistake while entering your name or birth date and see how the program allows you to easily correct the mistake. Note how the inverse input fields not only back up to the point where you want to re-enter the data, but the unused portions of the name field disappear as soon as you hit the RETURN or complete the maximum input length.

The scrolling window display does not try to do anything fancy. Refer to the write-up on that technique for other possible uses of the scrolling window.

Figure 12.9 — *CONTROL.DEM – A Menu Using Controlled Input*

```
100 REM CONTROL.DEM - MENU USING CONTROLLED INPUT
110 GRAPHICS 0:
    POKE 752,1
120 INKEY1=390:
    INKEY2=450:
    CORRECT=540:
    DELAY=580
130 DIM NAME$(10)
140 NAME$(1)=" ":NAME$(10)=" ":NAME$(2)=NAME$
150 PRINT CHR$(125):
    POSITION 10,3:
    PRINT "CONTROLLED MENU"
160 POSITION 2,7:
    PRINT "ENTER YOUR NAME ";:
    POSITION 23,7:
    PRINT "■■■■■■■■■■";
170 POSITION 23,7:
    SIZE=10:
    GOSUB INKEY1:
    POSITION 23,7:
    PRINT NAME$
180 POSITION 2,9:
    PRINT "ENTER YOUR BIRTHDAY";:
    POSITION 23,9:
    PRINT "■■/■■/■■";
```

```
190 POSITION 23,9:
    SIZE=2:
    GOSUB INKEY2:
    POSITION 26,9:
    GOSUB INKEY2:
    POSITION 29,9:
    GOSUB INKEY2
200 POSITION 2,14:
    PRINT "CONTROLLED MENU IS AN EXAMPLE OF A"
210 POSITION 2,15:
    PRINT "METHOD FOR GETTING SPECIAL INPUTS"
220 POSITION 2,17:
    PRINT "NOW WE WILL TRY THE SCROLLING WINDOW"
230 POSITION 2,19:
    PRINT "*************************************"
240 POKE 703,4:
    PRINT CHR$(125)
250 PRINT "YOU ARE NOW IN SCROLLING WINDOW MODE."
260 GOSUB DELAY:
    GOSUB DELAY:
    GOSUB DELAY:
    PRINT :
    PRINT :
    PRINT :
    GOSUB DELAY
270 PRINT "USING THIS WINDOW, YOU CAN ASK THE":
    GOSUB DELAY
280 PRINT "USER FOR ADDITIONAL INPUTS.":
    GOSUB DELAY:
    PRINT :
    PRINT :
    PRINT :
    GOSUB DELAY
290 PRINT "FOR EXAMPLE, YOU COULD ASK FOR THE":
    GOSUB DELAY
300 PRINT "USER TO PRESS ONE OF THE FUNCTION":
    GOSUB DELAY
310 PRINT "KEYS TO INITIATE LOADING A DISK FILE.":
    GOSUB DELAY:
    PRINT :
    PRINT :
    PRINT :
    GOSUB DELAY
320 PRINT "ONE THING YOU MAY HAVE NOTICED BY NOW":
    GOSUB DELAY
330 PRINT "IS THAT THIS GARBABE IS BEING":
    GOSUB DELAY
340 PRINT "DISPLAYED WITHOUT INTERFERING WITH":
    GOSUB DELAY
```

```
35Ø PRINT "THE DISPLAY ON THE MAIN SCREEN.":
    GOSUB DELAY:
    PRINT :
    PRINT :
    PRINT :
    GOSUB DELAY
36Ø PRINT "ENTER A GRAPHICS Ø COMMAND"
37Ø PRINT "TO RESTORE THE NORMAL SCREEN";:
    END
38Ø REM INKEY1.LST - CONTROLLED STRING INPUT
39Ø OPEN #5,4,Ø,"K:":
    FOR X=1 TO SIZE
4ØØ GET #5,KEY:
    IF KEY=155 THEN POP :
    GOTO 44Ø
41Ø IF KEY=126 THEN GOSUB CORRECT:
    GOTO 4ØØ
42Ø IF (KEY<48 OR KEY>122) AND KEY<>32 THEN 4ØØ
43Ø PRINT CHR$(KEY);:
    NAME$(X,X)=CHR$(KEY):
    NEXT X
44Ø CLOSE #5:
    RETURN
45Ø REM INKEY2.LST - CONTROLLED NUMERIC INPUT
46Ø SIGN=1:
    NUMBER=Ø:
    OPEN #5,4,Ø,"K:":
    FOR X=1 TO SIZE
47Ø GET #5,KEY:
    IF KEY=155 THEN POP :
    GOTO 52Ø
48Ø IF KEY=126 THEN GOSUB CORRECT:
    GOTO 47Ø
49Ø IF KEY=45 AND SIGN=1 THEN SIGN=-1:
    PRINT "-";:
    GOTO 47Ø
5ØØ IF KEY<48 OR KEY>57 THEN 47Ø
51Ø PRINT CHR$(KEY);:
    NUMBER=1Ø*NUMBER+VAL(CHR$(KEY)):
    NEXT
52Ø NUMBER=SIGN*NUMBER:
    CLOSE #5:
    RETURN
53Ø REM ERROR CORRECTION ROUTINE
54Ø X=X-1:
    IF X<1 THEN X=1:
    RETURN
55Ø PRINT CHR$(3Ø);" ";CHR$(3Ø);
56Ø RETURN
57Ø REM TIME DELAY ROUTINE
```

```
580 FOR Y=1 TO 500:
    NEXT Y:
    RETURN
```

If you want to see the effects of blinking fields, then install the BLINK routine before loading this one.

Video Antics

There are so many features available in the Atari video display graphics system that it would take a good sized book to do proper justice to even a fraction of them. Hence, we will be able to cover only a small sample in the space of a single chapter. We will talk about the ever popular marquee (video banner) style programs, how to use colors for a more dramatic effect in GRAPHICS 2, and how to use "page flipping" to create your own video slide show. We will show you a way to slow down those fast BASIC video listings to a pace that is easier to read. And last, but not least, we will show you some screen dump and retrieval routines.

Le Marquee D'Atari

A marquee program displays a specified message on the video screen and "scrolls" it from right-to-left across the screen until the end of the message is reached, at which point the message is repeated. There are three basic ways to solve this programming problem. The most elegant way is to redefine your display list using the methods outlined in *De Re Atari* to achieve smooth horizontal scrolling (a combination of coarse and fine scrolling). While this method is the right approach for assembly programmers, it is by no means a trivial task. We will look at two of the methods here. They may not be as elegant, but they make up for it by being much easier to understand and implement. Even here we must make trade offs.

The first technique involves altering, rather than replacing the display list. While this is easier than replacing the entire display list, this method still is not exactly what you would call easy. The routine shown in Figure 13.1 is a demonstration program that illustrates how you can create a simple scrolling banner with a few alterations to the display list. SCROLL.DEM is faster than using a pure BASIC scroll, so a time delay was inserted into the routine at LINE 260.

This demo program is short and simple, but not very flexible. The message must be stored on an even "page" boundary to satisfy ANTIC. In this particular example, I put the message on page six.

Figure 13.1 — *SCROLL.DEM – A Coarse Scrolling Demonstration*

```
100 REM SCROLL.DEM
110 GRAPHICS 2+16:
    DLIST=PEEK(560)+256*PEEK(561):
    L=0:
    H=6:
    NUM=119:
    GO=9:
    DIM STRING$(200)
120 REM SET UP MESSAGE STRING
130 STRING$="                    "
140 STRING$(LEN(STRING$)+1)="4()3 )3 ! -%33!'%"
150 REM STORE MESSAGE ON PAGE SIX
160 FOR X=1 TO LEN(STRING$):
    POKE 1535+X,ASC(STRING$(X,X))
170 IF ASC(STRING$(X,X))=32 THEN POKE 1535+X,0
180 NEXT X
190 REM FOR X=1 TO 254:
    POKE 1535+X,0:
    NEXT X
200 REM POINT DISPLAY TO MESSAGE
210 POKE DLIST+GO,NUM:
    POKE DLIST+GO+1,L:
    POKE DLIST+GO+2,H
220 REM SCROLL OUR MESSAGE
230 FOR X=0 TO 30240 POKE DLIST+GO,NUM:
    POKE DLIST+GO+1,L+X:
    POKE DLIST+GO+2,H
250 REM POKE 54276,15-X
260 FOR Y=1 TO 65:
    NEXT Y
270 REM RANDOMLY CHANGE COLORS
280 POKE 708,16*INT(16*RND(0))+7+INT(5*RND(0))
290 POKE 709,16*INT(16*RND(0))+7+INT(5*RND(0))
300 POKE 710,16*INT(16*RND(0))+7+INT(5*RND(0))
310 POKE 711,16*INT(16*RND(0))+7+INT(5*RND(0))
320 NEXT X
330 REM REPEAT MESSAGE ALL DAY
340 GOTO 230
```

Let's take a look at each line in the program and point out things of interest. In LINE 110, we set the graphics mode to full screen "two". The variable DLIST is then set equal to the address of the display list. "L" and "H" are the low and high bytes that define where the message will be located. "L" should always start out with a value of zero to make sure that you are on an even page boundary. "H" (in this case) points to page six. You might try experimenting with other values of "H". The effects can be pretty strange.

The variable NUM is the command code that we will later POKE into the display list. This variable tells the computer to perform a particular display function. I chose 119 to cause a scrolling row to be placed in the middle of the video display. I suggest that you try other values and experiment to see what they do. You can find more detailed technical information on display lists in *De Re Atari* as well as the *Technical User Notes* from Atari.

The next line of interest is 140. This is where we specify what our message will be. Looks like a bunch of garbage, doesn't it? This is due to the fact that we are playing directly with the display list, and it uses "display codes" rather than the normal ATASCII codes. If you take the ASC(X) of each element of the message string and add 32 to it, you will see the real message we used. I don't like this awkward translation process, but it has to be done. Of course, we could write a little routine to translate the various ATASCII codes to their proper display codes, but there is a simpler solution. We will talk more about that shortly.

Lines 160 and 170 POKE our message into page six. Note that the code we have to use to get a "SPACE" is zero. This is done by LINE 170. If you remove the "REM" at the beginning of LINE 190, the program will clear all of page six. Be sure to put the REM back when you are through, or you will never be able to see your message. This comes in handy when you are trying out different codes to compose a message.

LINE 210 initializes the modified display list, and the loop that starts in LINE shifts the display data to the left, resulting in a reasonably good scrolling effect. If you want the message to scroll to the right, change the loop limits to something like "FOR X=30 TO 0 STEP -1". You can also use the POKE statement in LINE 250 to do your scrolling, but the screen flickers annoyingly. If you were using a vertical blank interrupt machine language routine, you could get rid of the flicker. The address 54276 is the HORIZONTAL SCROLL register.

As you can see, this is not a clean straight topic to discuss. Let's move on to line 280 through 310. These lines change the OS color registers. By changing them inside a loop like this, it is possible to change the colors of upper/lower case and normal/inverse video characters on the fly, so to speak. I will cover this topic in more detail later in this chapter. Now let's look at another scrolling banner program.

The program in Figure 13.2 is a more sophisticated solution to the horizontal scrolling problem. This program is far more flexible in that you can enter your message in plain English. Your message can be up to 200 characters. The marquee is set up for a GRAPHICS 2 display. If you want to add additional color to your message, try using lower-case and inverse video.

When you run MARQUEE.BAS, the first prompt will ask you to define the length of your message. A message is defined in terms of 40 character lines. Since the longest message allowed in the program is 200 characters, you can have a message as short as one line or as long as five lines. Any unused spaces in a message line will be displayed as blanks in the marquee.

The screen will alter dramatically as soon as you enter the message length. Simply type in your message in the special input field. You may use any standard characters, including lower-case and inverse video. *Do not* press the <RETURN> key! If you chose a length greater than one, the cursor will automatically wrap around to the next input line. When you are finished entering/editting your message, use <CTRL>-<Down Arrow> to move the cursor down to the line where CONT is displayed. Once you have the cursor on that line, press the <RETURN>. The screen will go blank for a moment, and then your message will begin scrolling across the screen.

Figure 13.2 — *MARQUEE.BAS – A Banner Program*

```
100 REM Le Marquee D'Atari
110 REM With many thanks to John Weber
120 REM DEFINE BEG OF SCREEN ADDR
130 SAVMSC=PEEK(88)+256*PEEK(89)
140 DLIST=PEEK(560)+256*PEEK(561)
150 REM DEFINE MACH LANG LOCATIONS
160 HSON=203:LMS=205:HORZ=204:LIMIT=207:COUNT=1791:POKE 82,2
170 REM LOAD VERT BLANK INTERUPT ROUTINE
180 FOR N=1536 TO 1536+93:READ A:POKE N,A:NEXT N
190 REM ROUTINE TO GET MESSAGE
200 GRAPHICS 0:POSITION 1,3:PRINT "LE MARQUEE D'ATARI"
210 POSITION 2,6:PRINT "This program will generate a":
    PRINT "scrolling message 1 to 5 lines long."
220 PRINT "Enter number of lines (1 - 5)":PRINT
230 TRAP 230:OPEN #3,4,0,"K:"
240 GET #3,KEY:IF KEY<49 OR KEY>53 THEN 240
250 CLOSE #3:A=KEY-48
260 REM POKE IN BLANK LINES
270 POKE DLIST+17,112:POKE DLIST+18+A,112
280 POSITION 2,6:PRINT "MOVE THE CURSOR TO THE AREA BETWEEN"
290 PRINT "THE LINES AND ENTER YOUR MESSAGE.   "
300 PRINT "WHEN DONE, POSITION THE CURSOR AFTER  'CONT' AND PRESS 'RETURN'."
310 POSITION 2,13+A:PRINT "CONT":POSITION 2,10:POKE 82,0:STOP
320 REM TURN OFF ANTIC
330 B=PEEK(559):POKE 559,0:POKE LIMIT,40+A*40
340 POKE COUNT,0:POKE HSON,1:POKE HORZ,0
350 REM PUT DISPLAY LIST AT SDLSTL
360 FOR N=DLIST TO DLIST+23:READ A:POKE N,A:NEXT N
370 REM SET LMS OF SCROLLING LINE INTO LIST AND AT PAGE ZERO ADDR
380 C=INT((DLIST+11)/256):POKE LMS+1,C:POKE LMS,(DLIST+11)-C*256
390 C=INT((SAVMSC+460)/256):POKE DLIST+12,C:POKE DLIST+11,(SAVMSC+460)-C*256
400 REM PUT IN LMS OF TOP OF GR. DATA
410 C=INT(SAVMSC/256):POKE DLIST+5,C:POKE DLIST+4,SAVMSC-C*256
420 REM PUT IN LMS OF BOTTOM OF GR. DATA
430 C=INT((SAVMSC+100)/256):POKE DLIST+15,C:POKE DLIST+14,(SAVMSC+100)-C*256
440 REM SET BEGINNING ADDR OF LIST AT BOTTOM OF LIST
450 POKE DLIST+22,PEEK(560):POKE DLIST+23,PEEK(561)
460 POKE 548,0:POKE 549,6:REM ENABLE INTERRUPT
470 POKE 559,B:REM TURN ANTIC BACK ON
480 REM READ FUNCTION KEYS
490 C=6:GOTO 500
500 IF C=6 THEN POKE HSON,0:POKE 53279,7:GOTO 530
510 IF C=5 THEN POKE HSON,1:POKE 53279,7:GOTO 530
520 IF C=3 THEN POKE HSON,1:POKE 53279,7:GRAPHICS 0:RUN
530 C=PEEK(53279):POKE 77,0:GOTO 500
540 REM DATA STATEMENTS FOR SCROLLING ROUTINE
550 DATA 216,165,203,208,86,166,204,202,224,255,144,74
560 DATA 24,173,255,6,105,1,141,255,6,197,207,144
```

```
570 DATA 33,240,31,56,160,7,132,204,140,4,212,160
580 DATA 0,140,255,6,177,205,229,207,145,205,176,43
590 DATA 200,177,205,233,0,145,205,24,144,33,169,7
600 DATA 133,204,141,4,212,169,1,160,0,24,113,205
610 DATA 145,205,144,7,200,177,205,105,0,145,205,76
620 DATA 98,228,142,4,212,134,204,76,98,228
630 REM GR 2 DISPLAY LIST
640 DATA 112,112,112,71,16,159,7,7,7,7,87,116
650 DATA 159,71,136,159,7,7,7,7,7,65,0,6
```

The program is heavily commented so you can more easily see what is being done by each routine. I would like to thank my friend, John Weber, for his invaluable help with this program.

You can halt the scrolling by pressing the <SELECT> key. Once you have stopped the scroll, you can restart it by pressing the <START> key. If you get tired of the message and want to enter a new one, press the <OPTION> key.

Four Color Text In GRAPHICS 2

The Atari computer is a truly amazing color machine. There are dozens of colors in varying degrees of resolution from the coarse graphics of mode one to the ultra fine graphics of mode eight (there are three more modes if you have the GTIA chip or the model 1200 computer). It is even possible to get four different colors at a time on the screen in GRAPHICS mode zero by altering the Display List or by the careful use of a technique called *artifacting* with redefined characters. The easiest color shifts are accomplished by simply altering the color registers like we did in the two previous programs. This latter topic is what we will discuss here.

Memory locations 704 to 712 are the color registers for players, missiles and playfields. We are only going to concern ourselves with four of these registers: 708, 709, 710 and 711. Each of these registers corresponds to one of the COLOR commands.

Figure 13.3 — *COLOR Commands vs. Color Registers*

COLOR COMMAND	MEMORY REGISTER	DEFAULT VALUE	OPERATIVE GRAPHICS MODE	WHAT IT CONTROLS THE COLOR OF
COLOR 0	708	40	1 and 2	Normal upper case
COLOR 1	709	212	1 and 2	Normal lower case
COLOR 2	710	148	1 and 2	Inverse upper case
COLOR 3	711	70	1 and 2	Inverse lower case

NOTE: Normally "lower case" can refer only to alphabet characters.

The COLOR 4 command can be simulated by POKEing 712, but that is a topic for another discussion.

When you are in GRAPHICS 1 or 2, you can have multi-colored letters on the screen by carefully making some of the letters normal upper case, some of them inverse upper case, some of them lower case, and some of them inverse lower case. Then, by POKEing new values into the color registers, you can make each type of character a different color. This is particularly good for the title page of your program.

Here is an example of such a title page:

Figure 13.4 — *GRAPHICS 2 Sample Title Page*

```
2Ø64Ø REM TITLE.LST
2Ø641 PRINT CHR$(125):SETCOLOR 2,Ø,Ø:GRAPHICS 18
2Ø643 POSITION 16,1:PRINT #6,"my game"
2Ø645 POSITION 18,4:PRINT #6,"BY"
2Ø646 POSITION 14,5:PRINT #6,"YOURS TRULY"
2Ø647 POSITION 5,1Ø:PRINT #6;"(C) 1983"
2Ø648 POSITION 2,11:PRINT #6;"YOURS TRULY,INC"
```

Running this title, as is, will give you a pretty, multi-colored title page. If you don't like the default colors we used here, add some POKEs in LINE 20642 to change the color registers. If you are wondering how you can select certain colors, just hold on. We are coming to that shortly.

First, however, try this little routine:

Figure 13.5 — *A "GLOWING" Message Routine*

```
1ØØ REM GLOW1.DEM
11Ø GRAPHICS 2+16:
    SETCOLOR 2,Ø,Ø:
    POSITION 16,5:
    PRINT #6,"GLOWING"
12Ø FOR X=1 TO 2ØØ:
    POKE 7Ø8,16*INT(16*RND(Ø))+7+INT(5*RND(Ø)):
    NEXT X
13Ø GOTO 12Ø
```

When you RUN this routine, you will see the message changing colors so fast that it almost seems to glow. Try the same routine with the addition of lower case and inverse video characters in the message. Note that only the normal upper case characters glow this time. You will have to add three more lines to the routine before it will make your new message glow.

This routine will cause each of the letters in the message to glow slightly different from one another. I used Z-code variables to speed the loop up a little. You can now take the loop out of the glow routine and add it to the title page to add one more special effect. I suggest that you use the glow routine only on your program title, and simply set the other parts of the title page to a non-glowing color.

Change the glow routine to look like this:

Figure 13.6 — *A Better "GLOW" Routine*

```
100 REM GLOW2.DEM
110 GRAPHICS 2+16:
    SETCOLOR 2,0,0:
    POSITION 16,5:
    PRINT #6,"GlOwInG":
    Z0=0:
    Z5=5:
    Z7=7:
    Z16=16
120 FOR X=1 TO 200:
    FOR Y=708 TO 711
130 POKE Y,Z16*INT(Z16*RND(Z0))+Z7+INT(Z5*RND(Z0))
140 NEXT Y:
    NEXT X:
    GOTO 120
```

The equations used in the color changes in both glow routines were carefully chosen after a lot of experimentation. If you POKE completely random values from zero to 255 into the color registers, you will end up with a rather dull display. The following table will help to show you why this is true.

Figure 13.7 — *Atari Color Value Table*

COLOR NUMBER	BASE COLOR	POKE VALUE	POKE RANGE (LOW-HIGH)
0	BLACK	0	0 - 14
1	RUST	16	16 - 30
2	RED-ORANGE	32	32 - 46
3	DARK ORANGE	48	48 - 62
4	RED	64	64 - 78
5	DARK LAVENDER	80	80 - 94
6	COBALT BLUE	96	96 - 110
7	ULTRAMARINE	112	112 - 126
8	MEDIUM BLUE	128	128 - 142
9	DARK BLUE	144	144 - 158
10	BLUE-GREY	160	160 - 174
11	OLIVE GREEN	176	176 - 190
12	MEDIUM GREEN	192	192 - 206
13	DARK GREEN	208	208 - 222
14	ORANGE-GREEN	224	224 - 238
15	ORANGE	240	240 - 254

This table identifies the base colors produced by the Atari computer. Other "colors" are achieved by using a number in the *POKE range* of the color. These other colors are actually made by adding a *luminance* value of 0-15 to the base POKE value. The higher the luminance, the brighter the color. Note that each range ends in an even number. The color registers ignore the zeroth bit, so there are never really any odd numbers in one of the registers even if you POKE an odd number into one of them.

If you POKE a dark color value and use a low luminance value, the resulting characters will be very hard to see. On the other hand, if you pick a light color and give it a real high luminance, the characters will be so fuzzy that they will be hard to read. You can also get apparent color changes. For example, a color of 1(RUST) with a luminance of 14 (POKE value=30) is almost yellow. The obvious solution is to choose only medium values for luminance. Now let's go back and look at the glow routines again.

The glow routines use the following statement:

```
POKE Y,16*INT(16*RND(Ø))+7+INT(5*RND(Ø))
```

Let's examine this statement a little closer. The first part is 16*INT(16*RND(0)). This selects one of the 16 basic color POKE values. We can live with dark colors if they are bright enough. That is where the second part of the statement comes in. "7+INT(5*RND(0))" will pick a luminance value somewhere between seven and twelve. These are medium high luminance values (we do want the colors to be clearly visible) that are not so dim that the dark colors are invisible and not so bright that the light colors get blurry. Try your own experiments. Change the color select part to pick a limited range of colors, or change the "7" or "5" to higher or lower values.

Using Page Flipping for a "SLYDESHO"

The Atari home computer allows us to access, modify and store the Display Data and the Display List in any free area of memory. We have already played with this concept a little in our scrolling program. This flexibility in storage allows us to define and store the information for many different screens (pages) of text or graphics. This is a very powerful capability if we now couple it with a simple set of POKEs that "flip" the start of the actual video display from one of our stored screens to another. We can achieve some very interesting effects by creating and storing a number of screens ahead of time and then recalling them as needed.

There are only a few technical details that you will need to understand before you can start flipping pages. First is, "how to flip a page." The second is, "why is that page folded in the middle?"

Two memory locations control the apparent location of the video memory, and therefore also control what the computer will display on the screen. First, the fourth and fifth bytes in the display list point to the display data, which is what is printed on the screen. We can find this address by using the following lines of code:

```
1ØØ DLIST=PEEK(56Ø)+256*PEEK(561)
11Ø SCRDAT=PEEK(DLIST+4)+256*PEEK(DLIST+5)
```

Two other memory locations are important in this application. The lowest address of the screen display is stored in decimal addresses 88 and 89. Try the following lines of code:

```
12Ø SCREEN=PEEK(88)+256*PEEK(89)
13Ø FOR X=1Ø TO 9ØØ STEP 1Ø:POKE X,1Ø4:NEXT X
```

You will see lower case h's appear in different spots on the video display. When flipping pages, it is best to change both sets of addresses to point to your new page. You can run into strange problems if you are not careful when using these addresses.

Normally, when you use a GRAPHICS command, the computer sets up a display list and a display data area just below the top of free memory. By changing the values of SCRDAT and SCREEN, we can cause the computer to look someplace else for the display information. Let's look at a practical example of using this technique. The program in Figure 13.8 is what I call a "SLYDESHO" projector.

Figure 13.8 — *SLYDESHO.DEM – A Page Flipping Demonstration*

```
100 REM SLYDESHO.DEM -
110 REM A SLIDE SHOW VIEWER
120 REM DEMONSTRATION PROGRAM
130 PRINT CHR$(125):
    GRAPHICS 2+16:
    SETCOLOR 4,8,0
140 POSITION 16,3:
    PRINT #6,"slydesho":
    POSITION 14,8:
    PRINT #6," (C) 1983"
150 POSITION 12,10:
    PRINT #6," carl m evans":
    FOR X=1 TO 1300:
    NEXT X:
    GRAPHICS 0:
    POKE 752,1:
    POKE 82,0 160
    POSITION 8,12:
    PRINT "INITIALIZING...."
170 Z0=0:
    Z1=1:
    Z2=2:
    Z3=3:
    Z4=4:
    Z5=5:
    Z6=6:
    Z7=7:
    Z8=8:
    Z9=9:
    Z10=10:
    Z11=11:
    Z12=12:
    Z13=13:
    Z14=14:
    Z15=15:
    Z16=16
```

```
180 Z17=17:
    Z18=18:
    Z19=19:
    Z20=20:
    ZT=53279:
    ZR=4096:
    ZP=960:
    ZQ=959:
    Z256=256
190 NUMBER=7:
    REM SET PAGE LIMIT
200 DIM A1$(ZP),A2$(ZP),A3$(ZP),A4$(ZP),A5$(ZP),A6$(ZP),
    A7$(ZP),A8$(ZP),A9$(ZP),Q10$(ZP)
210 DIM A11$(ZP),A12$(ZP),A13$(ZP),A14$(ZP),A15$(ZP),
    A16$(ZP),A17$(ZP),A18$(ZP),A19$(ZP),A20$(ZP)
220 DIM PAGE(20),ADRLO(20),ADRHI(20),FLAG(20)
230 GOSUB 730:
    GOSUB 1080
240 GOSUB 590:
    GOSUB 640:
    ON RESULT GOTO 430,320,250,550
250 PRINT CHR$(125):
    POSITION Z5,Z12:
    PRINT "***  UNDER CONSTRUCTION  ***":
    FOR X=1 TO 500:
    NEXT X
260 GOTO 240
270 REM SAVE A PAGE
280 FOR Z=Z0 TO Z1:
    POKE (ADDRESS+Z),PEEK(SCR+Z):
    NEXT Z
290 FOR Z=Z2 TO ZQ:
    POKE (ADDRESS+Z),PEEK(SCR+Z):
    POKE (SCR+Z-Z1),30
300 POKE (SCR+Z-Z2),0:
    NEXT Z:
    RETURN
310 REM SLYDE EDITOR
320 Y=Z0:
    FOR X=Z1 TO NUMBER
330 IF FLAG(X)=Z1 THEN X=X+Z1:
    GOTO 330
340 IF X>NUMBER THEN X=NUMBER
350 IF FLAG(X)=Z1 THEN 240
360 PRINT CHR$(125):
    Y=Y+Z1
370 REM GET PAGE TO SAVE
380 POSITION Z2,Z2*(Z1+Y):
    PRINT "THIS IS TEST SCREEN #";Y
390 POSITION Z2,23:
    PRINT "THE END OF TEST SCREEN #";Y;
```

```
400 I=X:
    FLAG(I)=Z0:
    ADDRESS=PAGE(I):
    GOSUB 280:
    NEXT X
410 GOTO 240:
    REM DUMMY UNTIL LATER
420 REM SLYDE VIEWER
430 I=Z0:
    GOSUB 1030
440 GOSUB 640:
    ON RESULT GOTO 470,560,550,550
450 GOTO 440
460 FOR X=1 TO 500:
    NEXT X:
    GOTO 540
470 IF I>=NUMBER THEN 540
480 I=I+Z1:
    IF I>20 OR I>NUMBER THEN 540
490 IF I<Z1 THEN I=Z1
500 IF FLAG(I)=Z1 THEN 480
510 IF FLAG(I)=-Z1 THEN PRINT CHR$(253):
    PRINT CHR$(125):
    POSITION Z13,Z12:
    PRINT "OUT OF SLYDES":
    GOTO 460
520 IF FLAG(I)=-Z1 OR (FLAG(I)=Z1 AND I=NUMBER) THEN 540
530 GOSUB 1010:
    IF I<=NUMBER THEN 440
540 GOSUB 990:
    GOTO 430
550 GOSUB 990:
    GOTO 240
560 IF I<Z1 THEN I=Z1
570 I=I-Z2-FLAG(I-Z1):
    GOTO 480
580 REM MAIN MENU PAGE
590 PRINT CHR$(125):
    POSITION Z2,Z5:
    PRINT "SLIDE SHOW EDITOR/VIEWER MAIN MENU"
600 POSITION Z2,Z12:
    PRYNT "PRESS OPTION TO LOAD PAGES FROM DISK"
610 POSITION Z2,Z13:
    PRINT "PRESS SELECT TO ENTER EDIT MODE"
620 POSITION Z2,Z14:
    PRINT "PRESS START  TO START SLYDE SHOW";:
    RETURN
630 REM FUNCTION KEY MONITOR
640 RESULT=Z0:
    IF PEEK(764)=28 THEN RESULT=Z4:
    POKE 764,255:
    GOTO 690
```

```
650 IF PEEK(ZT)=Z6 THEN RESULT=Z1:
    GOTO 690
660 IF PEEK(ZT)=Z5 THEN RESULT=Z2:
    GOTO 690
670 IF PEEK(ZT)=Z3 THEN RESULT=Z3:
    GOTO 690
680 GOTO 640
690 FOR X=1 TO 50:
    NEXT X:
    RETURN
700 REM TWO BYTE ADDRESS SPLIT
710 ADRHI(I)=INT(ADDRESS/Z256):
    ADRLO(I)=ADDRESS-Z256*ADRHI(I):
    RETURN
720 REM INITIALIZE PAGE STORAGE
730 I=Z0:
    GOSUB 1310:
    ADDRESS=ADR(A1$):
    GOSUB 970
740 GOSUB 1320:
    ADDRESS=ADR(A2$):
    GOSUB 970
750 GOSUR 1330:
    ADDRESS=ADR(A3$):
    GOSUB 970
760 GOSUB 1340:
    ADDRESS=ADR(A4$):
    GOSUB 970
770 GOSUB 1350:
    ADDRESS=ADR(A5$):
    GOSUB 970
780 GOSUB 1360:
    ADDRESS=ADR(A6$):
    GOSUB 970
790 GOSUB 1370:
    ADDRESS=ADR(A7$):
    GOSUB 970
800 GOSUB 1380:
    ADDRESS=ADR(A8$):
    GOSUB 970
810 GOSUB 1390:
    ADDRESS=ADR(A9$):
    GOSUB 970
820 GOSUB 1400:
    ADDRUSS=ADR(A10$):
    GOSUB 970
830 GOSUB 1410:
    ADDRESS=ADR(A11$):
    GOSUB 970
840 GOSUB 1420:
    ADDRESS=ADR(A12$):
    GOSUB0970
```

```
85Ø GOSUB 143Ø:
    ADDRESS=ADR(A13$):
    GOSUB 97Ø
86Ø GOSUB 144Ø:
    ADDRESS=ADR(A14$):
    GOSUB 97Ø
87Ø GOSUB 145Ø:
    ADDRESS=ADR(A15$):
    GOSUB 97Ø
88Ø GOSUB 146Ø:
    ADDRESS=ADR(A16$):
    GOSUB 97Ø
89Ø GOSUB 147Ø:
    ADDRESS=ADR(A17$):
    GOSUB 97Ø
9ØØ GOSUB 148Ø:
    ADDRESS=ADR(A18$):
    GOSUB 97Ø
91Ø GOSUB 149Ø:
    ADDRESS=ADR(A19$):
    GOSUB 97Ø
92Ø GOSUB 15ØØ:
    ADDRESS=ADR(A2Ø$):
    GOSUB 97Ø
93Ø SCRLO=PEEK(88):
    SCRHI=PEEK(89):
    SCR=SCRLO+Z256*SCRHI
94Ø DLISTO=PEEK(56Ø)
    DLISTHI=PEEK(561):
    DLIST=DLISTLO+Z256*DLISTHI:
    LO=DLIST+Z4:
    HI=DLIST+Z5
95Ø SAVL=PEEK(LO):
    SAVH=PEEK(HI)
96Ø FOR X=ZØ TO Z2Ø:
    FLAG(X)=-Z1:
    NEXT X:
    RETURN
97Ø I=I+Z1:
    GOSUB 71Ø:
    PAGE(I)=ADDRESS:
    RETURN
98Ø REM RESTORE ORIGINAL SCREEN
99Ø POKE LO,SAVL:
    POKE HI,SAVH:
    POKE 88,SCRLO:
    POKE 89,SCRHI:
    RETURN
1ØØØ REM FLIP TO A NEW PAGE
```

```
1010 POKE LO,ADRLO(I):
     POKE HI,ADRHI(I):
     POKE 88,ADRLO(I):
     POKE 89,ADRHI(I):
     RETURN
1020 REM VIEWER MENU PAGE
1030 PRINT CHR$(125):
     POSITION 10,Z5:
     PRINT "SLYDESHO VIEWER MENU"
1040 POSITION Z2,Z12:
     PRINT "PRESS OPTION TO EXIT TO MAIN MENU"
1050 POSITION Z2,Z13:
     PRINT "PRESS SELECT TO BACKUP TO LAST PAGE"
1060 POSITION Z2,Z14:
     PRINT "PRESS START  TO ADVANCE TO NEXT PAGE";:
     RETURN
1070 REM 4K BOUNDARY CHECK
1080 FOR X=Z1 TO Z8
1090 IF ADR(A1$)/ZR<=X AND (ADR(A1$)+ZQ)/ZR>=X THEN
     FLAG(Z1)=Z1
1100 IF ADR(A2$)/ZR<=X AND (ADR(A2$)+ZQ)/ZR>=X THEN
     FLAG(Z2)=Z1
1110 IF ADR(A3$)/ZR<=X AND (ADR(A3$)+ZQ)/ZR>=X THEN
     FLAG(Z3)=Z1
1120 IF ADR(A4$)/ZR<=X AND (ADR(A4$)+ZQ)/ZR>=X THEN
     FLAG(Z4)=Z1
1130 IF ADR(A5$)/ZR<=X AND (ADR(A5$)+ZQ)/ZR>=X THEN
     FLAG(Z5)=Z1
1140 IF ADR(A6$)/ZR<=X AND (ADR(A6$)+ZQ)/ZR>=X THEN
     FLAG(Z6)=Z1
1150 IF ADR(A7$)/ZR<=X AND (ADR(A7$)+ZQ)/ZR>=X THEN
     FLAG(Z7)=Z1
1160 IF ADR(A8$)/ZR<=X AND (ADR(A8$)+ZQ)/ZR>=X THEN
     FLAG(Z8)=Z1
1170 IF ADR(A9$)/ZR<=X AND (ADR(A9$)+ZQ)/ZR>=X THEN
     FLAG(Z9)=Z1
1180 IF ADR(A10$)/ZR<=X AND (ADR(A10$)+ZQ)/ZR>=X THEN
     FLAG(Z10)=Z1
1190 IF ADR(A11$)/ZR<=X AND (ADR(A11$)+ZQ)/ZR>=X THEN
     FLAG(Z11)=Z1
1200 IF ADR(A12$)/ZR<=X AND (ADR(A12$)+ZQ)/ZR>=X THEN
     FLAG(Z12)=Z1
1210 IF ADR(A13$)/ZR<=X AND (ADR(A13$)+ZQ)/ZR>=X THEN
     FLAG(Z13)=Z1
1220 IF ADR(A14$)/ZR<=X AND (ADR(A14$)+ZQ)/ZR>=X THEN
     FLAG(Z14)=Z1
1230 IF ADR(A15$)/ZR<=X AND (ADR(A15$)+ZQ)/ZR>=X THEN
     FLAG(Z15)=Z1
1240 IF ADR(A16$)/ZR<=X AND (ADR(A16$)+ZQ)/ZR>=X THEN
     FLAG(Z16)=Z1
1250 IF ADR(A17$)/ZR<=X AND (ADR8A17$)+ZQ)/ZR>=X THEN
     FLAG(Z17)=Z1
```

```
1260 IF ADR(A18$)/ZR<=X AND (ADR(A18$)+ZQ)/ZR>=X THEN
     FLAG(Z18)=Z1
1270 IF ADR(Q19$)/ZR<=X AND (ADR(A19$)+ZQ)/ZR>=X THEN
     FLAG(Z19)=Z1
1280 IF ADR(A20$)/ZR<=X AND (ADR(A20$)+ZQ)/ZR>=X THEN
     FLAG(Z20)=Z1
1290 NEXT X:
     RETURN
1300 REM INITIALIZE ARRAYS
1310 A1$(Z1)=" ":
     A1$(ZP)=" ":
     A1$(Z2)=A1$:
     RETURN
1320 A2$(Z1)=" ":
     A2$(ZP)=" ":
     A2$(Z2)=A2$:
     RETURN
1330 A3$(Z1)=" ":
     A3$(ZP)=" ":
     A3$(Z2)=A3$:
     RUTURN
1340 A4$(Z1)=" ":
     A4$(ZP)=" ":
     A4$(Z2)=A4$:
     RETURN
1350 A5$(Z1)=" ":
     A5$(ZP)=" ":
     A5$(Z2)=A5$:
     RETURN
1360 A6$(Z1)=" ":
     A6$(ZP)=" ":
     A6$(Z2)=A6$:
     RETURN
1370 A7$(Z1)=" ":
     A7$(ZP)=" ":
     A7$(Z2)=A7$:
     RETURN
1380 A8$(Z1)=" ":
     A8$(ZP)=" ":
     A8$(Z2)=A8$:
     RETURN
1390 A9$(Z1)=" ":
     A9$(ZP)=" ":
     A9$(Z2)=A9$:
     RETURN
1400 A10$(Z1)=" ":
     A10$(ZP)=" ":
     A10$(Z2)=A10$:
     RETURN
```

```
1410 A11$(Z1)=" ":
     A11$(ZP)=" ":
     A11$(Z2)=A11$:
     RETURN
1420 A12$(Z1)=" ":
     A12$(ZP)=" ":
     A12$(Z2)=A12$:
     RETURN
1430 A13$(Z1)=" ":
     A134(ZP)=" ":
     A13$(Z2)=A13$:
     RETURN
1440 A14$(Z1)=" ":
     A14$(ZP)=" ":
     A14$(Z2)=A14$:
     RETURN
1450 A15$(Z1)=" ":
     A15$(ZP)=" ":
     A15$(Z2)=A15$:
     RETURN
1460 A16$(Z1)=" ":
     A16$(ZP)=" ":
     A16$(Z2)=A16$:
     RETURN
1470 A17$(Z1)=" ":
     A17$(ZP)=" ":
     A17$(Z2)=A17$:
     RETURN
1480 A18$(Z1)=" ":
     A18$(ZP)=" ":
     A18$(Z2)=A18$:
     RETURN
1490 A19$(Z1)=" ":
     A19$(ZP)=" ":
     A19$(Z2)=A19$:
     RETURN
1500 A20$(Z1)=" ":
     A20$(ZP)=" ":
     A20$(Z2)=A20$:
     RETURN
```

SLYDESHO is only set up to save a few limited display pages to memory and to allow you to recall them one after the other in quick succession, hence the name of the program. I built in the command framework for you to add in a full screen editor and disk load/save routines.

The program is set up to handle about 16 different GRAPHICS 0 screens. You will note, however, that it looks like it is set up to handle twenty pages. This leads us to the second technical detail that you must understand before you can use page flipping in your own programs.

A normal video display cannot cross a 4K memory boundary without some kind of modification to the Display List. In this particular case, we did not want to have to do this, so we designed our BASIC program to scan the addresses of the proposed pages to determine if any of them crossed such a boundary. When this occurs a flag is set to prevent that page from being used. The flag is named FLAG, and its values and meanings are shown in Figure 13.9. If you want to see some weird text displays, add a loop in at LINE 495 that sets all of the flags back to zero.

Figure 13.9 — *SLYDESHO Page Flag Table*

FLAG VALUE	WHAT IT MEANS
-1	This page is not in use
Ø	This page is in use
1	This page cannot be used

The technique used here can be adapted for use with graphic modes other than mode zero, but be careful to dimension the string variables A1$-A20$ to the proper size. Since they will usually have to be larger than 960 bytes, you will have to settle for a smaller number of pages in memory at one time.

Slower BASIC Listings

When you type in the LIST command in BASIC or while using the assembler/editor cartridge, the lines whir by you and off the top of the screen so fast that they are almost impossible to read. Atari built-in an interrupt (CNTRL-1) that will stop the video display. Pressing CNTRL-1 again restarts the listing. This seems awfully awkward. Wouldn't it be nice if we could slow down or speed up the display at will and stop or start the listing with a single key? The machine language program in Figure 13.10 gives us this capability. The BASIC POKE version of the program is given in Figure 13.11.

Figure 13.10 — *SLOWLIST – A Machine Language Slow Lister*

```
            1ØØØ ;SLOWLIST - SLOWS VIDEO LISTINGS
            1Ø1Ø ;
            1Ø2Ø ;
ØØØØ        1Ø3Ø        .OPT NOEJECT
            1Ø4Ø ;
            1Ø5Ø ;
            1Ø6Ø ;THIS ROUTINE ALLOWS YOU TO CHANGE THE SPEED OF
            1Ø7Ø ;THE VIDEO DISPLAY WHILE A PROGRAM IS LISTING.
            1Ø8Ø ;
            1Ø9Ø ;
            11ØØ ;ONCE THIS ROUTINE HAS BEEN LOADED, THE NEW CONTROLS ARE:
            111Ø ;
            112Ø ;OPTION & START    : ON/OFF SWITCH FOR THIS ROUTINE
            113Ø ;OPTION           : SPEEDS LISTING UP
```

```
                ;SELECT            : SLOWS LISTING DOWN
                ;START             : START/STOP SWITCH FOR LISTING
                ;
                ;************************************************************
                ;
                ;SET UP OS POINTERS
                ;
0002            CASINI  =      $2              ;CASSETTE INIT VECTOR
0009            BOOTF   =      $9              ;BOOT MODE FLAG
0042            CRITIC  =      $42             ;CRITICAL I/O FLAG
0222            VBLANK  =      $222            ;IMMEDIATE VBLANK VECTOR
02FF            SSFLAG  =      $2FF            ;SCREEN START/STOP FLAG
159D            DUP     =      $159D           ;OS FLAG TO DETECT DUP.SYS
D01F            CONSOLE =      $D01F           ;CONSOLE FUNCTION KEYS
E45C            SETVBI  =      $E45C           ;SET-VBI VECTOR ENTRY
E45F            SYSVBI  =      $E45F           ;OS VBLANK SERVICE ROUTINE
                ;
                ;************************************************************
                ;
                ;THE INIT ROUTINE AT $400 IS EXECUTED ONLY ONCE.
                ;THE MAIN ROUTINE IS STORED ON PAGE SIX.
                ;
0000                    *=     $400            ;THIS IS LATER OVER-WRITTEN
                ;
                ;
                ;SET UP PRIVATE INTERRUPT
                ;
0400 A509               LDA    BOOTF           ;IF A CASSETTE HAS BOOTED
0402 2902               AND    #2              ;THEN SAVE CASINI FOR LATER
0404 F00A               BEQ    INIT
0406 A602               LDX    CASINI          ;RE-VECTOR CASSETTE INIT
0408 A403               LDY    CASINI+1        ;TO INCLUDE OUR ROUTINE
040A 8E0B06             STX    DETOUR+1
040D 8C0C06             STY    DETOUR+2
0410 A905       INIT    LDA    #RESET&$FF
0412 8502               STA    CASINI
0414 A906               LDA    #RESET/256
0416 8503               STA    CASINI+1
0418 AD2202             LDA    VBLANK          ;DETOUR NORMAL HOUSEKEEPING
041B 8D3306             STA    EXIT+1
041E AD2302             LDA    VBLANK+1
0421 8D3406             STA    EXIT+2
0424 A509               LDA    BOOTF
0426 0902               ORA    #2
0428 8509               STA    BOOTF
042A A206               LDX    #MAIN/256       ;POINT VBLANK TO OUR ROUTINE
042C A035               LDY    #MAIN&$FF
042E A906               LDA    #6              ;USE IMMEDIATE VBI
0430 205CE4             JSR    SETVBI
0433 60                 RTS
                ;
```

```
            165Ø ;**********************************************************
            166Ø ;
            167Ø ;THIS IS THE PART WE WANT TO PRESERVE
            168Ø ;
Ø434        169Ø        *=     $6ØØ           ;PROGRAM IS NOT RELOCATABLE
            17ØØ ;
            171Ø ;SAVE SPACE FOR BYPASS SWITCH
            172Ø ;
Ø6ØØ Ø1     173Ø SWITCH  .BYTE1               ;Ø=BYPASS, 1=EXECUTE
            174Ø ;
            175Ø ;SAVE SPACE FOR COUNTERS
            176Ø ;
Ø6Ø1 ØØ     177Ø COUNTER .BYTEØ               ;HOW MANY DELAYS SO FAR
Ø6Ø2 ØØ     178Ø LIMIT   .BYTEØ               ;THIS CONTROLS TIME DELAY
            179Ø ;
            18ØØ ;SAVE SPACE FOR START/STOP SWITCH
            181Ø ;
Ø6Ø3 ØØ     182Ø START   .BYTEØ               ;0=GO,1=STOP LISTING
            183Ø ;
            184Ø ;SAVE SPACE FOR CONSOLE FLAG
            185Ø ;
Ø6Ø4 ØØ     186Ø CONFLAG .BYTEØ               ;USED TO PREVENT KEYBOUNCE
            187Ø ;
            188Ø ;THIS RESTORES OUR ROUTINE WHEN SYSTEM RESET IS PRESSED
            189Ø ;
Ø6Ø5 A9A9   19ØØ RESET   LDA   #$A9           ;SYSTEM RESET COMES HERE
Ø6Ø7 8D27Ø6 191Ø         STA   PATCH
Ø6ØA 2Ø2CØ6 192Ø DETOUR  JSR   NULL
Ø6ØD A9ØØ   193Ø         LDA   #Ø             ;RESTORE DEFAULT VALUES
Ø6ØF 8DFFØ2 194Ø         STA   SSFLAG         ;TO OUR FLAGS & COUNTERS
Ø612 8DØ2Ø6 195Ø         STA   LIMIT
Ø615 8DØ1Ø6 196Ø         STA   COUNTER
Ø618 8DØ3Ø6 197Ø         STA   START
Ø61B 8DØ4Ø6 198Ø         STA   CONFLAG
Ø61E A9Ø1   199Ø         LDA   #1
Ø62Ø 8DØØØ6 2ØØØ         STA   SWITCH
Ø623 A2Ø6   2Ø1Ø         LDX   #MAIN/256      ;TELL VBI WHERE OUR ROUTINE IS
Ø625 AØ35   2Ø2Ø         LDY   #MAIN&$FF
Ø627 A9Ø6   2Ø3Ø PATCH   LDA   #6             ;IMMEDIATE VBI
Ø629 2Ø5CE4 2Ø4Ø         JSR   SETVBI         ;SET THE GEARS IN MOTION
Ø62C 6Ø     2Ø5Ø NULL    RTS
            2Ø6Ø ;
            2Ø7Ø ;THIS IS THE MAIN ROUTINE
            2Ø8Ø ;
Ø62D 68     2Ø9Ø THAW    PLA                  ;RESTORE COMPUTER REGISTERS
Ø62E A8     21ØØ         TAY
Ø62F 68     211Ø         PLA
Ø63Ø AA     212Ø         TAX
Ø631 68     213Ø         PLA                  ;CONTINUE DEFAULT VBI
Ø632 4C2CØ6 214Ø EXIT    JMP   NULL           ;(NULL CHANGED DURING SETUP)
            215Ø ;
```

```
                2160 ;VBLANK INTERRUPT COMES HERE
                2170 ;
Ø635 48         2180 MAIN    PHA                    ;SAVE CURRENT REGISTERS
Ø636 8A         2190         TXA
Ø637 48         22ØØ         PHA
Ø638 98         221Ø         TYA
Ø639 48         222Ø         PHA
Ø63A AD9D15     223Ø         LDA     DUP            ;IF DUP.SYS IS IN COMPUTER
Ø63D C9ØØ       224Ø         CMP     #Ø             ;THEN PATCH IT SO IT
Ø63F FØØF       225Ø         BEQ     MONITOR        ;WILL NOT KILL OUR ROUTINE
Ø641 A94C       226Ø         LDA     #$4C           ;UPON EXIT TO CARTRIDGE
Ø643 8D2A27     227Ø         STA     $272A
Ø646 A912       228Ø         LDA     #$12
Ø648 8D2B27     229Ø         STA     $272B
Ø64B A919       23ØØ         LDA     #$19
Ø64D 8D2C27     231Ø         STA     $272C
                232Ø ;
                233Ø ;CHECK FOR CRITICAL I/O FLAG
                234Ø ;
Ø65Ø A542       235Ø MONITOR LDA     CRITIC         ;IF CRITIC IS SET,
Ø652 DØD9       236Ø         BNE     THAW           ;THEN EXIT NOW
                237Ø ;
                238Ø ;CHECK FOR MANUAL OVER RIDE
                239Ø ;
Ø654 A9Ø7       24ØØ         LDA     #7             ;ARE ANY CONSOLE KEYS PRESSED?
Ø656 CD1FDØ     241Ø         CMP     CONSOLE
Ø659 FØ52       242Ø         BEQ     TEST           ;IF NO, THEN EXIT
Ø65B AD1FDØ     243Ø         LDA     CONSOLE        ;COMPARE CURRENT CONSOLE KEY(S)
Ø65E CDØ4Ø6     244Ø         CMP     CONFLAG        ;TO VALUE DURING LAST VBI
Ø661 FØ4A       245Ø         BEQ     TEST           ;IF NO CHANGE, THEN EXIT
                246Ø ;
Ø663 A9Ø2       247Ø         LDA     #2             ;LOOK FOR BOTH OPTION AND START
Ø665 CD1FDØ     248Ø         CMP     CONSOLE
Ø668 DØØ8       249Ø         BNE     BYPASS         ;IF NO, THEN EXIT
Ø66A ADØØØ6     25ØØ         LDA     SWITCH         ;SWITCH = NOT SWITCH
Ø66D 49Ø1       251Ø         EOR     #1
Ø66F 8DØØØ6     252Ø         STA     SWITCH
                253Ø ;
Ø672 ADØØØ6     254Ø BYPASS  LDA     SWITCH         ;IS BYPASS ACTIVATED?
Ø675 FØ55       255Ø         BEQ     FINIS          ;IF YES, THEN EXIT
Ø677 DØØØ       256Ø         BNE     GAS            ;IF NO, THEN SCAN KEYS
                257Ø ;
                258Ø ;FUNCTION KEY SWEEP SCAN
                259Ø ;
Ø679 A9Ø3       26ØØ GAS     LDA     #3             ;CHECK FOR OPTION KEY
Ø67B CD1FDØ     261Ø         CMP     CONSOLE
Ø67E DØØD       262Ø         BNE     BRAKE          ;IF OPTION IS PRESSED,
Ø68Ø CEØ2Ø6     263Ø         DEC     LIMIT          ;DECREASE MAXIMUM TIME DELAY
Ø683 1Ø19       264Ø         BPL     FLIP           ;MAKE SURE MAX DELAY=>Ø
Ø685 A9ØØ       265Ø         LDA     #Ø
Ø687 8DØ2Ø6     266Ø         STA     LIMIT
Ø68A 4C9EØ6     267Ø         JMP     FLIP
```

```
               2680 ;
068D A905      2690 BRAKE  LDA    #5          ;CHECK FOR SELECT KEY
068F CD1FD0    2700        CMP    CONSOLE
0692 D00A      2710        BNE    FLIP        ;IF SELECT IS PRESSED,
0694 EE0206    2720        INC    LIMIT       ;INCREASE MAXIMUM TIME DELAY
0697 D005      2730        BNE    FLIP        ;MAKE SURE MAX DELAY<=$FF
0699 A9FF      2740        LDA    #$FF
069B 8D0206    2750        STA    LIMIT
               2760 ;
069E A906      2770 FLIP   LDA    #6          ;CHECK FOR START KEY
06A0 CD1FD0    2780        CMP    CONSOLE
06A3 D008      2790        BNE    TEST        ;IF START IS PRESSED,
06A5 A901      2800        LDA    #1
06A7 4D0306    2810        EOR    START       ;TOGGLE START/STOP SWITCH
06AA 8D0306    2820        STA    START
               2830 ;
               2840 ;START/STOP VIDEO LISTING
               2850 ;
06AD AD0306    2860 TEST   LDA    START       ;TEST START/STOP SWITCH
06B0 D008      2870        BNE    STOP        ;STOP LISTING IF SWITCH IS OFF
               2880 ;
06B2 AD0106    2890 SLOW   LDA    COUNTER     ;COUNT DOWN DELAY COUNTER
06B5 F00A      2900        BEQ    ZERO
06B7 CE0106    2910        DEC    COUNTER
               2920 ;
06BA A901      2930 STOP   LDA    #1          ;STOP LISTING
06BC 8DFF02    2940        STA    SSFLAG
06BF D00B      2950        BNE    FINIS       ;EXIT ROUTINE
               2960 ;
06C1 AD0206    2970 ZERO   LDA    LIMIT       ;WHEN COUNTER=0, RESET IT
06C4 8D0106    2980        STA    COUNTER     ;TO THE CHOSEN TIME DELAY
06C7 A900      2990        LDA    #0          ;ENABLE NORMAL LISTING
06C9 8DFF02    3000        STA    SSFLAG
               3010 ;
06CC AD1FD0    3020 FINIS  LDA    CONSOLE     ;SAVE CURRENT CONSOLE KEY(S)
06CF 8D0406    3030        STA    CONFLAG
06D2 4C2D06    3040        JMP    THAW        ;ALL DONE, LET'S GO HOME
06D5           3050        .END
```

Figure 13.11 — *SLOWLIST.BAS – A BASIC POKE Version of SLOWLIST*

```
100 REM SLOWLIST.BAS
110 REM
120 REM THIS IS THE VBI SETUP ROUTINE
130 REM THAT IS TEMPORARILY STORED ON
140 REM PAGE FOUR.      $0400 (1024)
150 DATA 165,9,41,2,240,10,166,2
```

```
160 DATA 164,3,142,11,6,140,12,6
170 DATA 169,5,133,2,169,6,133,3
180 DATA 173,34,2,141,51,6,173,35
190 DATA 2,141,52,6,165,9,9,2
200 DATA 133,9,162,6,160,53,169,6
210 DATA 32,92,228,104,96
220 REM NOTE THE NUMBER '104' IN LINE
230 REM 210 IS ONLY IN THIS BASIC
240 REM VERSION. IT IS NOT IN THE
250 REM BINARY LOAD FILE VERSION
260 MLSTART=1024
270 MLEND=1076
280 FOR X=MLSTART TO MLEND
290 READ Y:POKE X,Y:
    NEXT X
300 REM THIS IS THE MAIN ROUTINE
310 REM IT IS STORED ON PAGE SIX
320 DATA 1,0,0,0,0,169,169,141
330 DATA 39,6,32,44,6,169,0,141
340 DATA 255,2,141,2,6,141,1,6
350 DATA 141,3,6,141,4,6,169,1
360 DATA 141,0,6,162,6,160,53,169
370 DATA 6,32,92,228,96,104,168,104
380 DATA 170,104,76,44,6,72,138,72
390 DATQ 152,72,173,157,21,201,0,240
400 DATA 15,169,76,141,42,39,169,18
410 DATA 141,43,39,169,25,141,44,39
420 DATA 165,66,208,217,169,7,205,31
430 DATA 208,240,82,173,31,208,205,4
440 DATA 6,240,74,169,2,205,31,208
450 DATA 208,8,173,0,6,73,1,141
460 DATA 0,6,173,0,6240,85,208
470 DATA 0,169,3,205,31,208,208,13
480 DATA 206,2,6,16,25,169,0,141
490 DATA 2,6,76,158,6,169,5,205
500 DATA 31,208,208,10,238,2,6,208
510 DATA 5,169,255,141,2,6,169,6
520 DATA 205,31,208,208,8,169,1,77
530 DATA 3,6,141,3,6,173,3,6
540 DATA 208,8,173,1,6,240,10,206
550 DATA 1,6,169,1,141,255,2,208
560 DATA 11,173,2,6,141,1,6,169
570 DATA 0,141,255,2,173,31,208,141
580 DATA 4,6,76,45,6
590 MLSTART=1536
600 MLEND=1748
610 FOR X=MLSTART TO MLEND
620 READ Y:
    POKE X,Y:
    NEXT X
630 REM NOW THAT WE HAVE THE ROUTINE
```

```
640 REM LOADED INTO MEMORY, WE TURN
650 REM IT ON BY THIS LINE
660 X=USR(1024)
670 END
```

The program works by taking control of the same memory location that the CNTRL-1 command uses. This address is 767 decimal and is used by the operating system to start or stop the scrolling of the screen. A value of zero in this location enables the normal listing functions. A non-zero value stops the scrolling function. When you press CNTRL-1, this address is toggled between zero and 255.

SLOWLIST takes over the task of monitoring this address and assigns certain new powers to the console function keys. These new controls are outlined in Figure 13.12.

Figure 13.12 — *SLOWLIST Commands*

FUNCTION KEY(S)	PURPOSE
OPTION & START	ON/OFF switch for this routine
OPTION	Speeds up a slowed listing
SELECT	Slows down the listing
START	Start/stop the listing

SLOWLIST works through a vertical blank interrupt routine stored on page six. Every 1/60th of a second, this routine scans the function keys and changes the listing flag accordingly. The routine automatically shuts itself off whenever any critical I/O, such as writing to a disk, is being done.

Using the routine is very simple. LOAD the program and, if you are using the BASIC POKE version, then just RUN it. The program will take over from there.

Saving and Retrieving Screen Data

There are many occasions when you will need to save the contents of a video display for later recall. If you don't need to save the data on disk, you can use the page flipping technique we discussed earlier. However, what do you do if you just spent three hours plotting a beautiful GRAPHICS 8 masterpiece? Naturally you would like to save the picture to disk for recall at some future time.

There are a number of special graphics utilities, such as Micro Painter, that make it "easy" for you to create artistic masterpieces and save them to disk. Programs like that have two major drawbacks. First, they are "drawing" programs. In my case, all I wanted to do was plot 3-D mathematical functions, so I wrote a little routine that saves a GRAPHICS 8 screen to disk. Of course, once I had the picture on disk, I needed another routine to retrieve it from disk and yet another routine to dump my pretty plots to my printer in graphics mode.

Don't get me wrong. I think that special drawing programs are great, if you have the artistic talent to make proper use of one, but even the most sophisticated computer graphics program can't turn you into an artist. I bought the Micro Painter program, but all I was able to do was transfer my crude crayon drawings from a piece of paper to the video screen. I know the program is good, because I have seen what a professional artist friend of mine has been able to do with it. Look at some of the printer plots in this chapter to get a rough idea of what you can achieve if you have artistic talent.

Another problem with commercial graphics programs is that they usually don't tell you how to use the resulting pictures with your own programs. The routines in this chapter will let you save any GRAPHICS 8 picture, and (with minor modifications) they will also save screen displays in other graphics modes. The operation of each routine is explained in detail so you will understand how to make these modifications. I will also show you how to use these routines to put Micro Painter pictures with your own programs.

The first routine, GR8PUT.DSK, will put a GRAPHICS 8 screen on disk. Once you have a picture on disk, you can use GR8GET.DSK to load the picture back into memory and CITOH.GR8 to plot it on a C-ITOH 8510 (Prowriter) printer. Once you know how the screen data is stored in memory and on your disk file, you can modify the routines to handle a number of other situations.

The data (picture) you see on the video screen is stored in normal memory much like a program would be. The starting location of the "screen memory" will vary (as will the number of bytes used in the screen memory) depending upon a number of factors. The amount of available RAM determines the default location of screen memory. You can also change the apparent location of screen memory by using the page flipping techniques we discussed earlier. In any case, the actual start of the screen memory that you will see on the video display can be found by PEEKing decimal addresses 88 and 89. We will use this as the starting address for a data buffer in the first two routines in this section. The equation for finding the address of the screen memory is:

```
SCREEN=PEEK(88)+256*PEEK(89)
```

Figure 13.13 — *Summary of Screen Memory Sizes*

GRAPHICS Mode	Horizontal Elements	Vertical Elements	Screen Data Size (bytes)	Actual Memory Usage (bytes)
0	40	24	960	992
1	20	24	480	672
2	20	12	240	420
3	40	24	240	432
4	80	48	480	696
5	80	48	960	1176
6	160	96	1920	2184
7	160	96	3840	4200
8	320	192	7680	8138
9	80	192	7680	8138
10	80	192	7680	8138
11	80	192	7680	8138

Note: Modes 9-11 are not in the old CTIA graphics chip.

Now that we know where the screen data is, we have to determine how many bytes are in the screen memory so our save routine will know how many bytes to save on the disk. The normal GRAPHICS 0 screen has 40*24 or 960 bytes in it. If you have a 48K computer, the screen memory will default to address 40000. When you go into GRAPHICS 8, the screen memory eats up 7680 bytes per screen. See Figure 13.13 for a brief summary of the screen memory required for each of the BASIC graphics modes. The "Screen Data Size" is the actual number of bytes you have to save to store the picture on disk. The "Actual Memory Usage" will be slightly higher due to system overhead for the different modes. See *De Re Atari* for more detail.

Okay, we have located the block of memory that contains the screen display, and we wrote a little routine to save and retrieve the screen on disk. But what is this? The colors are all messed up. We neglected to save the screen colors! Lets backtrack to the discussion we had about color registers. You will recall that we made changes in the screen display by POKEing certain numbers in memory locations 708-712. Maybe we can save the screen colors by saving the contents of these registers along with our screen data? Lets try it. Hey! It works! We can now save a GRAPHICS 8 picture on disk and recall it later with no loss of detail. The routines we end up with are shown below.

Figure 13.14 — *GR8PUT.DSK – A Screen Save Utility*

```
20960 REM GR8PUT.DSK
20961 REM PUT A GRAPHICS 8 SCREEN
20962 REM IN A DISK FILE.
20963 REM SAVE COLORS UP FRONT
20964 RESTORE 20977
20965 DIM N$(13),NAME$(16)
20966 NAME$="":
      NAME$(1,3)="D1:":
      N$="":
      READ N$:
      NAME$(4,LEN(N$)+3)=N$
20967 SCREEN=PEEK(88)+256*PEEK(89)
20968 NUM=7680:
      OPEN #1,8,0,NAME$
20969 REM SAVE GRAPHICS MODE
20970 MODE=PEEK(87):
      PUT #1,MODE
20971 REM SAVE COLORS
20972 FOR X=0 TO 4:
      COL=PEEK(708+X):
      PUT #1,COL:
      NEXT X
20973 REM SAVE SCREEN DATA
20974 BEGIN=SCREEN:
      FINIS=SCREEN+NUM-1
20975 FOR X=BEGIN TO FINIS:
      BYTE=PEEK(X):
      PUT #1,BYTE:
      NEXT X
```

```
20976 REM DESTINATION FILE NAME
20977 DATA SHIP2
20978 CLOSE #1
20979 END
```

GR8PUT.DSK is a simple BASIC program that transfers the contents of the screen memory byte-by-byte to a disk file. The screen mode is PEEKed out of address 87 and sent as the first byte of the disk file. Then each of the color registers are sent to the disk followed by the actual screen memory data. The file name used by the program is specified by the contents of the DATA statement in LINE 20977. Note the variable NUM in LINE 20968. This variable is set to the number of bytes in the screen memory (Screen Data Size as defined by Figure 13.13). If you want to save a screen from a different graphics mode, then change this variable to the appropriate value from that column. Of course, you will have to append this routine to the end of whatever drawing routine you are using before you can save a screen.

The second routine, GR8GET.DSK, is a tad more complex. I couldn't see taking the time neccessary to write a machine language routine for saving my screen displays since the few minutes used to save a routine were negligable compared to the time I had to spend creating a new screen. However, retrieving the pictures from a disk was another matter. If I were using a special plot or picture in a program, I wanted it to load rapidly so the "flow" of the main program would not be adversly affected. So I wrote a small machine language subroutine to use in GR8GET.DSK. The little machine routine in GR8GET.DSK is extremely simple. All it does is stuff the address of the screen memory (a source buffer) into the proper IOCB control registers, tell the computer how many bytes are in the buffer, and do a JSR (Jump SubRoutine) to the resident CIO handler in the operating system. If all of this is Greek to you, don't worry. You don't have to understand all of it to use it.

For those of you who want the actual source code, I will jot down a brief synopsis here. The routine is so simple that I assembled it by hand. The ASSEMBLER EDITOR would have been overkill.

Figure 13.15 — *Set Up IOCB With Machine Language*

```
PLA          ;DROP NUMBER OF ARGUMENTS
PLA          ;DROP MSB OF IOCB NUMBER
PLA          ;GET LSB OF IOCB NUMBER
TAX          ;PUT IT IN THE X REGISTER (X=16*DEVICE NUMBER)
PLA          ;GET MSB OF BUFFER ADDRESS
STA 837,X    ;STUFF IT IN ICBAH FOR THIS IOCB
PLA          ;GET LSB OF BUFFER ADDRESS
STA 836,X    ;STUFF IT IN ICBAL FOR THIS IOCB
PLA          ;GET MSB OF BUFFER LENGTH
STA 841,X    ;STUFF IT IN ICBLH FOR THIS IOCB
PLA          ;GET LSB OF BUFFER LENGTH
STA 840,X    ;STUFF IT IN ICBLL FOR THIS IOCB
JSR $E456    ;GET CIO TO LOAD THE SCREEN DATA FROM DISK
RTS          ;RETURN TO BASIC
```

With the exception of the machine language subroutine, GR8GET.DSK is a mirror operation of GR8PUT.DSK. First, the screen mode is retrieved and POKEd into SCREEN to set up the proper graphics mode before the picture is loaded. Then the colors are fetched from the file and stored in the appropriate color register. Finally, the machine language routine is called to load the actual screen data. The GOTO in LINE 20953 is there to keep the picture on the video display. Pressing BREAK will return you to normal GRAPHICS 0 BASIC.

Figure 13.16 — *GR8GET.DSK – A Screen Load Utility*

```
2Ø93Ø REM GR8GET.DSK
2Ø931 REM GET A GRAPHICS 8 PICTURE
2Ø932 REM FROM DISK AND DISPLAY IT.
2Ø933 DATA 1Ø4,1Ø4,1Ø4,17Ø,1Ø4,157,69,3
2Ø934 DATA 1Ø4,157,68,3,1Ø4,157,73,3
2Ø935 DATA 1Ø4,157,72,3,32,86,228,96
2Ø937 GRAPHICS 8+16
2Ø938 DIM F$(24),N$(13),NAME$(16)
2Ø939 FOR X=1 TO 24:
      READ Y
2Ø94Ø F$(X,X)=CHR$(Y):
      NEXT X
2Ø941 RESTORE 2Ø952
2Ø942 NAME$="":
      NAME$(1,3)="D1:":
      N$=""
2Ø943 READ N$:
      NAME$(4,LEN(N$)+3)=N$
2Ø944 IF N$="END" THEN 2Ø953
2Ø945 NUM=768Ø:
      OPEN #1,4,Ø,NAME$
2Ø946 GET #1,MODE
2Ø947 SCREEN=PEEK(88)+256*PEEK(89)
2Ø948 POKE SCREEN,MODE
2Ø949 FOR X=Ø TO 4:
      GET #1,COL:
      POKE 7Ø8+X,COL:
      NEXT X
2Ø95Ø X=USR(ADR(F$),16,SCREEN+1,NUM-1)
2Ø951 CLOSE #1:
      GOTO 2Ø942
2Ø952 DATA VSOFT,END
2Ø953 GOTO 2Ø953
```

You should also note that the word "END" now appears in the file name DATA statement. GR8GET.DSK is designed to get multiple pictures from a disk. All you have to do to load the three pictures — VSOFT, SHIP and HISEAS — is to put all three file names in the DATA statement (LINE 20952). If you want to keep one of the pictures on the screen longer than the others, put its name in the DATA statement more than once. Alternatively, you could put a small FOR/NEXT loop somewhere in the program. The loading of pictures is terminated when the routine encounters the file name "END" in the DATA statement.

Those of you who don't have a disk drive yet, don't despair. You can use these routines almost as is by simply changing the OPEN statements so they open the cassette recorder instead of the disk drive. Specifically, change LINE 20968 to:

```
20968 NUM=7680:OPEN #1,8,0,"C:"
```

and change LINE 20945 to:

```
20945 NUM=7680:OPEN #1,4,0,"C:"
```

Of course, you will have to expect the saving and loading process to be a lot slower.

We now have a routine for saving a picture and another routine for getting the picture back again. These routines are what you will want to use most of the time, but what do you do when you want to show your friends at work (or school) an example of what you have been doing? The answer is to write one more small routine that will dump your pictures out to your printer. The printer I currently have is a C-ITOH 8510 dot matrix printer, so the screen dump routine I wrote is customized for this particular printer. If you have a different kind of printer, you will have to translate my printer control codes to those used by your printer. It is also possible that your printer doesn't even have graphics capability. If that is the case, then I guess all you can do now is drool.

Figures 13.18 through 13.21 are some examples of the results I have been getting with the CITOH.GR8 screen dump routine. The pictures are shown in the actual size they came out on the printer. If you are planning a picture, particularly for printer output, you might want to reverse your light and dark colors on the video display. I plotted two pictures, one of a ship at night and the other of a ship during the day. The daytime scene looked like it was nighttime and the nighttime scene looked like it was daytime. The reason is that I set the screen background to BLACK before drawing the pictures. So the night scene had very little data in the night sky areas, while the daytime scene had a lot of data to achieve the blue sky. Try experimenting. It is a lot of fun as well as being educational.

Figure 13.17 — *CITOH.GR8 – Dump a Screen to a Printer*

```
21000 REM CITOH.GR8
21001 REM DUMP GRAPHICS 8 SCREEN
21002 REM TO 8510 C-ITOH PRINTER
21003 OPEN #1,8,0,"P:"
21004 PRINT #1,CHR$(27);"T02"
21005 FOR Y=1 TO 191
21006 PRINT #1;CHR$(27);"S0320";
21007 FOR X=0 TO 319
21008 LOCATE X,Y,A
21009 PUT #1,A
21010 NEXT X
21011 PRINT #1
21012 NEXT Y
21013 CLOSE #1
21014 END
```

Figure 13.18 *Sample of a GRAPHICS 8 Screen Dump*

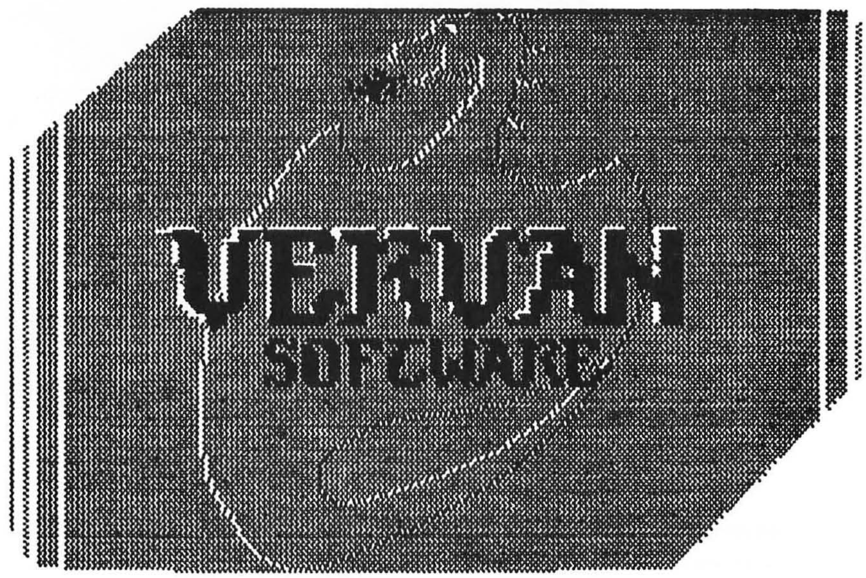


Figure 13.19 *Screen Dump of a Micro Painter Picture*

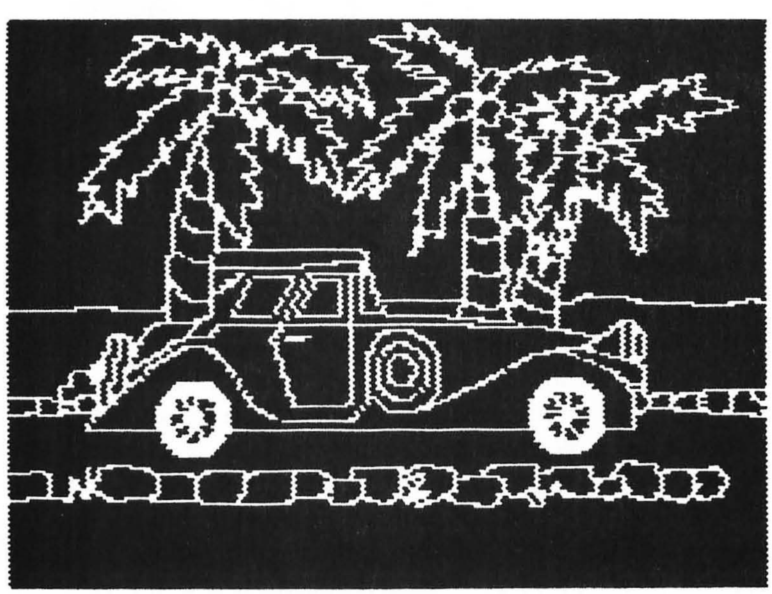


Figure 13.20 *Screen Dump of a World Map*

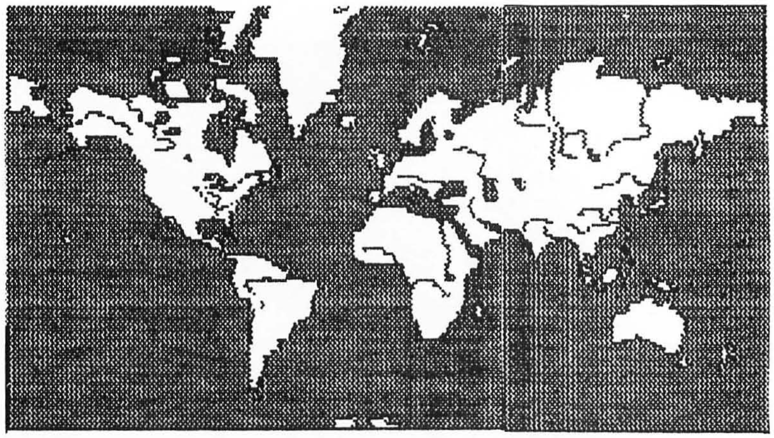
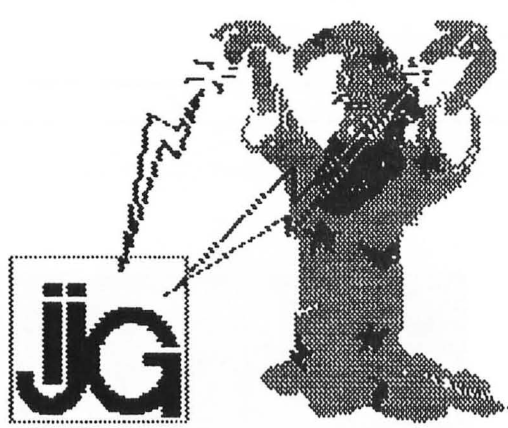


Figure 13.21 *Screen Dump of the IJG Wizard*



Now let's go full circle and discuss something I mentioned at the beginning of this section: MICRO-PAINTER (MP) picture files. The picture files for that particular graphics package are stored on disk in what I consider to be a slightly peculiar format. The first byte of an MP picture file is the mode value like we used before, but there are no color values in the front of the file. The next data you come to is screen data. If you delete LINE 20949 in GR8GET.DSK, you could use it to fetch the picture, but the colors would be all wrong. A close examination of an MP picture file reveals that the color data is stored at the end of the screen data. This is odd enough, but the color values are not in sequential order. Once you figure out what values do what to your picture, you finally can decode their sequence. The last four bytes of data are in the following order: COLOR4, COLOR0, COLOR1 and COLOR2. By POKEing the values in 712, 708, 709 and 710 respectively, you end up with a picture that has the proper colors.

The routine shown below, PAINTGET.DSK, is a short subroutine you can add to one of your programs to get an MP picture from disk. I find it awkward to have my pictures in several different formats, so I usually combine these routines to read an MP file from disk and then save it back out to disk in the GR8GET.DSK format. There are similar technical problems with pictures created by other graphics programs, so don't get the impression that I am picking on MICRO-PAINTER. I like the program.

Figure 13.22 — *PAINTGET.GET - Load a MICRO-PAINTER Picture*

```
20900 REM PAINTGET.DSK
20901 REM GET MICRO-PAINTER PICTURE
20902 REM FROM DISK AND DISPLAY IT.
20903 DATA 104,104,104,170,104,157,69,3
20904 DATA 104,157,68,3,104,157,73,3
20905 DATA 104,157,72,3,32,86,228,96
20907 GRAPHICS 8+16
20908 DIM F$(24),N$(13),NAME$(16)
20909 FOR X=1 TO 24:
      READ Y
20910 F$(X,X)=CHR$(Y):
      NEXT X
20911 RESTORE 20927
20912 NAME$="":
      NAME$(1,3)="D1:":
      N$=""
20913 READ N$:i
      NAME$(4,LEN(N$)+3)=N$
20914 IF N$="END" THEN 20928
20915 NUM=7680:
      OPEN #1,4,0,NAME$
20916 GET #1,MODE
20917 SCREEN=PEEK(88)+256*PEEK(89)
20918 POKE SCREEN,MODE
20919 X=USR(ADR(F$),16,SCREEN+1,NUM-1)
20920 REM FOR MICROPAINTER FILES
20921 TRAP 20926
20922 GET #1,COL4:
      POKE 712,COL4
20923 GET #1,COL0:
      POKE 708,COL0
20924 GET #1,COL1:
      POKE 709,COL1
20925 GET #1,COL2:
      POKE 710,COL2
20926 CLOSE #1:
      GOTO 20912
20927 DATA VSOFT,END
20928 GOTO 20928
```

Sound Advice

The Atari home computer is first, and foremost, a graphics machine, but what makes it a real competitor with the arcades is its built-in sound capability. The Atari computer has four independent sound channels that each cover three-and-a-half octaves. What this means, in non-technical terms, is that your Atari computer can produce great sound effects and even music!

This chapter will concern itself primarily with the creation of sound effects. When it comes to music, I enjoy listening to it, but I have never had the urge to learn how to create music. One of these days, a real music buff will write a book about synthesizing music with an Atari computer, but I am not that person.

The movie industry was quick to realize that sound was an important ingredient to an effective presentation. Background music was used early on as a mood setter. Later, when it became technically possible to synchronize the sound with the movie, music and, later, sound effects were used to add impact to the action on the screen. The results can be astounding. In fact, people have become so used to this form of entertainment that the only silent film anyone will pay to see is one of the old classics.

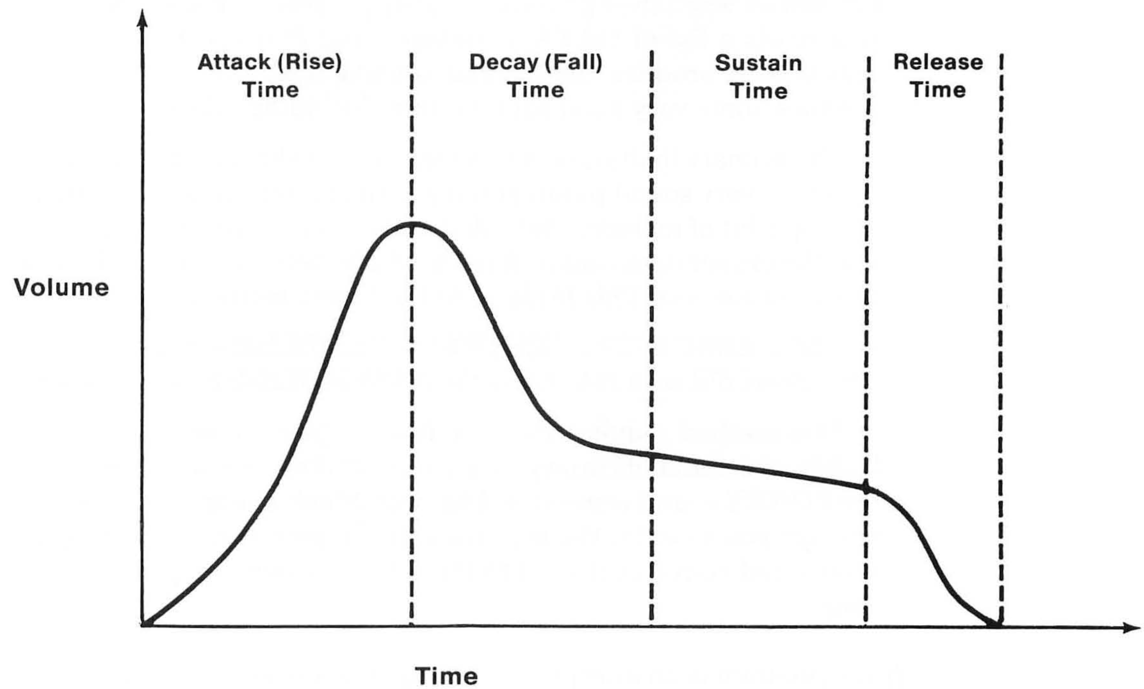
We can see the same kind of development in the home computer field. In the old days of home computers, all we could do was sit and stare intently at the silent video screen. Then we got a real thrill! The computer could beep at us during a game. Now, the computers make a wide variety of sounds. Games today make explosion sounds, let you hear the spaceships zoom by, and even play mood setting background music. A few games even talk to you!

Sounds like that are an integral part of a good game. The aliens in Space Invaders are merely lights moving on the screen until you add sound effects. The rhythmic thrumming sound they make starts out slow and comes faster and faster as the aliens descend on your bases. The sounds reach out and grab your emotions, getting you more and more deeply involved in the action on the screen. This emotional effect is probably one of the reasons why arcade games are so addicting.

What Is A Sound?

Rather than trying to describe sound in terms that a physicist would understand, let's look at sound as a simple curve on a piece of paper. The curve in Figure 14.1 shows what is called the *envelope* of a sound. We will discuss each of the four parts of that envelope briefly and show you how they relate to the sound registers in your computer.

Figure 14.1 — *The Envelope of a Sound*



The envelope of a sound is composed of four basic parts. The standard engineering terms are: "attack," "decay," "sustain" and "release." The "attack" time or, as it is also sometimes called, the "rise" time refers to the amount of time needed for the sound to reach its maximum volume (amplitude). The "decay" or "fall" time refers to how long it takes the volume to go back down. In the case of a pure sine wave, these two values are used to define the *frequency* of the sound. If this were all there was to sound, it would be easy to synthesize them. Life would sound kind of strange, however. Most real life sounds also have a period in their cycle where the sound hasn't quite died out. This is called the *sustain* time. The point at which this sustain level ends and the last of the sound goes away is called the *release* time.

Regardless of the computer used, there are four fundamental methods that are used to program these parameters into a synthesizer:

1. STATIC — this is where a sound is turned on and left alone while you go off and do something else. Obviously, this method doesn't allow you to have sophisticated sounds like music or even most of the simpler sound effects but there are a few interesting effects that we will illustrate for you in the demo portion of this chapter.

2. DYNAMIC ALGORITHM — this method involves using an equation to calculate the various sound parameters to use. This method uses very little memory for even complex sound effects. This is typically the method we use from BASIC. The "equation" in this case ends up being a short series of FOR-NEXT loops

containing PEEKs, POKEs and IF-THEN statements. This method becomes progressively more difficult with the complexity of the sound effect. Music, for example, is much more easily created using one of the following methods.

3. DYNAMIC TABLE LOOKUP — this is the technique that you see in the magazines when they give you a music program. This technique uses a short loop that reads a list of DATA statements and POKEs those data into the POKEY registers to produce the desired sounds. This method is very flexible and can produce some very good sound effects including music.

The primary limitation of this technique is the fact that you must have a piece of data for every sound parameter for each of many, many sounds. This not only can take up a lot of memory, but, what is even worse, you must "assemble" the sounds into the proper data values. What would be better is a method that would do a lot of this work for you. This leads us to the fourth method.

4. DYNAMIC INTERPRETER/ASSEMBLER — this technique is analagous to what Atari did with BASIC or the ASSEMBLER/EDITOR Cartridge.

This method requires that you have a special program that will accept certain high level commands from you and convert these commands into the proper data for the POKEY sound generator. The Atari Music Composer is one such program that enables you to enter the musical notes in your song. The program analyzes your inputs and converts them into the numbers your computer needs to produce the song.

If the program is an interpreter, your inputs are stored in essentially the same form that you entered them. When you go to play the song, the program reads the stored commands and *interprets* them just before it sends the next note's data to the computer.

A music assembler operates on your commands in a slightly different manner. This kind of program converts your musical notes into a data file that contains the data needed by the POKEY sound generator. This is just an easier way to achieve the same end result we got in method number three.

I used music as the primary example in the above discussion since most of the synthesizer programs on the market are especially written for music. There is only one true sound effects program on the market that I know of at this time. That program is INSOMNIA, which is available from the Atari Program Exchange.

In each case, sound effects programs act to shape the envelope of the desired sound (I bet you thought we were through with envelopes). Here is where the sticky part comes in. In many sound effects, several different sounds must be produced in rapid succession. For some applications, BASIC is simply not fast enough. If you want to do really fancy sounds, such as a fast paced violin concerto, you will be forced to do it in machine language. Another benefit of using machine language is that you can use vertical blank interrupts to have your music continue to play while the main program is doing something else.

If I were going to use a routine like that, I would use the VBI routine that we used earlier in this book and have the routine pick up the data for the notes from page six or a string array. Fortunately, however, most of the kinds of sound effects I use in my programs can usually be achieved in BASIC using method number two or method three. Before we show you some of these sound effects routines, let's examine the way that the Atari computer generates sounds.

A Sound POKE Gets You in the POKEY

Whenever you use a SOUND statement, you are really changing the value stored in one or more memory locations. Let's look at those locations to see exactly what happens when a SOUND command is invoked.

Tone Control

The memory locations in the Atari computer that are used in sound production are listed in Figure 14.2. These registers can be divided into three types or functions. The first are the tone controls AUDF1-AUDF4. These four registers control the frequency or tone of the sound produced by each of the four "voices" in the Atari. The allowable range of POKE values for these registers is the familiar 0-255. The lower the number, the higher pitched the tone. It's also possible to alter the tone by POKEing AUDCTL, but we will cover that a little later.

The locations 53760, 53762, 53764 and 53766 use the number stored in them to determine the tone by the following equation:

```
TONE = CLOCK/NUMBER
where: NUMBER = Value stored in AUDCF1-4
CLOCK  = Current clock rate
```

The clock is set at one of three frequencies. You can use the main system clock which runs at 1.79 MHz, or you can use either of two other clocks that run at 64 KHz and 15 KHz respectively. We will talk more about this in a few minutes.

Figure 14.2 — *The Sound Control Registers*

MEMORY LOCATION	NAME	FUNCTION
53760	AUDF1	Tone of Voice 1 (SOUND 0)
53761	AUDC1	Distortion and Volume of Voice 1
53762	AUDF2	Tone of Voice 2 (SOUND 1)
53763	AUDC2	Distortion and Volume of Voice 2
53764	AUDF3	Tone of Voice 3 (SOUND 2)
53765	AUDC3	Distortion and Volume of Voice 3
53766	AUDF4	Tone of Voice 4 (SOUND 3)
53767	AUDC4	Distortion and Volume of Voice 4
53768	AUDCTL	Special Control Register

Controlling Volume and Distortion

Each of the four voice channels has a dedicated control register. As you can see from Figure 14.2, these registers are memory locations 53761, 53763, 53765 and 53767. Usually, POKE values are in some specific range, such as 0-15 or 0-255. The POKE range for these registers is a little weird. To explain that assessment, let's have a look at the bit assignments for those registers as shown in Figure 14.3.

Figure 14.3 — *AUDC1-4 Bit Assignments*

BIT NUMBER	FUNCTION
Ø	Volume Bit Ø (LSB)
1	Volume Bit 1
2	Volume Bit 2
3	Volume Bit 3 (MSB)
4	Special Control Bit
5	Distortion Bit Ø (LSB)
6	Distortion Bit 1
7	Distortion Bit 2 (MSB)

The volume bits allow us to use any volume from zero to fifteen, which is the highest number you can have with only four bits. The use of these bits is simple. A volume value of zero results in silence, while a volume value of fifteen results in the loudest noise that channel can produce.

Let's skip over the special control bit for a moment and get the distortion bits put away first. The first thing to note is that those three bits are treated as a 3-bit number even though they occur in the high order bit positions of this byte. If all we wanted to do was set each of those three bits, we would have to use some multiple of 32. For example, 1*32 sets the first distortion bit, and 2*32 sets the second distortion bit. This means that there are really only 2*2*2, or eight possible distortion values.

This all seems simple and straightforward enough. There must be a catch! There is. The "distortion" value used in the SOUND command operates on all four of the high bits in AUDC1-4. The extra bit is taken care of by giving you a number that is twice the real distortion value. For example, if you use the common "pure tone" distortion value of "10", you are really using a distortion of "5". The POKE value commonly called out in the magazines for this is 160, where they define this value as 16*DISTORTION and limit the distortion values to even numbers. There is nothing wrong with this definition. It works quite well as long as you obey the rule of using only even distortion values, but this kind of rule does not explain the "why" of it all.

I, too, suggest that you use the rule described in the magazines, but I think you should understand why you are limited to even distortion values. The key to this understanding lies in that mysterious special control bit that we skipped over earlier. If you use an odd multiple of 16 in your POKE to one of the audio control registers, you set that control bit. When this bit is set (i.e., it equals one), you have put that voice channel in what Atari calls the "VOLUME ONLY" mode. In simple terms, this means that the distortion value in the three high bits is *totally ignored*!

What the computer does in this case is immediately send the current VOLUME value to the speaker. You say, "So what?" The answer is not nice. Normally when you send a sound to the speaker, the speaker whooshes in and out, like a good speaker should, to create the particular sound you sent to it. This is not the case in the VOLUME ONLY mode. When you send a sound in this mode, the speaker moves to the position that corresponds to the volume value you just sent to it, but the speaker *does not relax to normal*! The speaker stays in that particular position until you send it another move command. This allows you to closely control the exact movement of the speaker if you use the proper sequence of movement commands. Note that I don't call such a command a SOUND command. The problem that you will run into is that normal sound effects don't use the right kinds of values for proper movement control. Thus, when you use an odd POKE value in the distortion equation, you usually get a short POP and then silence.

Figure 14.4 — *VOLUME ONLY POKE Values for AUDC1-4*

Avoid these values when poking directly into the audio control registers when doing music or sound effects:

```
16-31,48-63,8Ø-95,112-127,144-159,176-191,2Ø8-223,24Ø-255
```

Note that this does *not* apply to the tone registers.

The ability to alter the distortion enables us to create a wide range of sound effects. It is possible, in machine language, to simulate almost any sound envelope that you will ever need. Knowing exactly what envelope you need is the real hard part. Before showing you how to put all of this information to use, let's look at the last sound register, 53768.

Special Sound Control Register — AUDCTL

In Figure 14.2 we identified memory location 53768 as a special control register. This register, AUDCTL, is able to affect the operation of any or all of the sound channels. Figure 14.5 Shows the bit assignments for AUDCTL and the effect they have on the sound channels.

Figure 14.5 — *AUDCTL Bit Assignments*

BIT NUMBER	FUNCTION WHEN SET
Ø	Change clock base from 64 KHz to 15 KHz
1	Put hi-pass filter in channel 2, clocked by channel 4
2	Put hi-pass filter in channel 1, clocked by channel 3
3	Merge outputs of channels 3 and 4 (16 bit resolution)
4	Merge outputs of channels 1 and 2 (16 bit resolution)
5	Change channel 3 clock to 1.79 MHz
6	Change channel 1 clock to 1.79 MHz
7	Change distortion base from 17 counts to 9 counts

I won't try to cover all of the technical details of these bits here. We will discuss these bits briefly and try to give you some idea as to their detailed effects on the sound registers. If you want to get into the technical aspects in more depth, I suggest that you study chapter seven in *DE RE ATARI*.

Bits zero, five and six are used to directly change the clock rate used by one or more of the sound registers. This is the clock rate we referred to a few pages back. The normal default clock is the 64 KHz one. If you decrease the clock rate, you will be able to get higher pitched sounds at the cost of giving up some of the lower pitched tones. You can also get *much* lower pitched tones by increasing the clock to 1.79 MHz. Of course, you lose most of your high pitched tones, but sometimes that is ok.

Bits one and two are probably the hardest to understand and the most difficult to use properly. These two bits enable you to limit the tones in the filtered channel. The limit is set to whatever the current tone is in the "clocking" channel. In particular, the filter cuts out any tone that is lower than the "clocking" tone and passes only those which are equal to or higher than the clocking tone. Thus the name "HIGH-PASS" filter. You will have to experiment with this one a lot before you find the right effect. Try the demo program in Figure 14.6 with the paddle values set to 254 and 127 to see one of the effects you can get. This demo program should help you find a number of static sound effects.

Figure 14.6 — *SOUND1.DEM - Sound Effects Demo Number One*

```
100 REM SOUND1.DEM
110 POKE 752,1:
    PRINT CHR$(125):
    SOUND 0,0,0,0:
    VOLUME1=8:
    VOLUME3=8
120 POSITION 8,2:
    PRINT "SOUND EFFECTS DEMO #1"
130 POSITION 11,9:
    PRINT "PADDLE (0) = "
140 POSITION 11,11:
    PRINT "PADDLE (1) = "
150 POKE 53768,4:
    REM AUDCTL
160 POKE 53761,160+VOLUME1:
    REM AUDC1
170 POKE 53765,160+VOLUME3:
    REM AUDC3
180 POKE 53760,PADDLE(0):
    REM AUDF1
190 POKE 53764,PADDLE(1):
    REM AUDF3
200 POSITION 23,9:
    PRINT "    ";CHR$(30);
    CHR$(30);CHR$(30);PADDLE(0)
210 POSITION 23,11:
    PRINT "    ";CHR$(30);
    CHR$(30);CHR$(30);PADDLE(1)
220 GOTO 180
```

Bit numbers three and four give us the ability to perform sounds using 16 bits of resolution. Usually eight-bit resolution is adequate, but there are some cases where eight bit resolution leaves us in the lurch. For example, try the following:

```
FOR PITCH=255 TO Ø STEP -1:SOUND 1,PITCH,1Ø,8:NEXT PITCH
```

When you execute this statement, the tone starts out low and relatively smoothly goes up, but as the loop gets near the end, the tone changes in a jerky manner. This is because the default clock is 64 KHz. Dividing the clock by 255 is almost the same as dividing it by 254, but as the loop gets closer to zero, the apparent change in frequency becomes much larger. For example, 64000/40=1600 and 64000/20=3600, which is a delta of 100%! Using 16-bit sound lets us count down by smaller increments. Basically this method uses the even numbered channel as the MSB and the odd numbered channel as the LSB of the 16 bit counter. For example, putting a "1" in the odd channel causes a divide by one and putting a "1" in the even channel causes a divide by 256. Putting a "1" in both channels causes a divide by 257. See *DE RE ATARI* for more information.

Bit number seven of AUDCTL changes the way the computer acts on a distortion value. It is beyond the scope of this book to discuss this parameter in any real detail. If you want something to occupy your mind for a couple of weeks, read the write up in *DE RE ATARI* and learn all about poly counters. Suffice it to say here that the effects of this control register bit can be quite dramatic.

Using What We Have Learned

The first thing to do before applying what we have learned so far is to study one more short topic. What is the difference between POKEing the sound registers and using a SOUND statement? The answer is easy. A POKE command changes only the memory location you POKE to. As long as you understand the consequences of the particular number you used in the POKE, you will be ok. This is why we went into the detail we did in the previous sections of this chapter.

The difference between POKEs and SOUND lies in the way a SOUND statement works. Each time you use a SOUND statement, the computer initializes the POKEY chip for sound output and clears the special control register, AUDCTL. Obviously if you want to use the special features of AUDCTL, you will have to use POKEs instead of SOUND statements. If you don't want to use any SOUND statements, then be sure to POKE 53768,0 and POKE 53775,3 before POKEing the sound registers to activate POKEY. A SOUND 0,0,0,0 statement (or any other SOUND statement) is usually used prior to POKEing the sound registers. Failure to initialize POKEY will be silence. So, the proper method for using the POKE techniques for sound generation is to first issue a SOUND 0,0,0,0 statement (or POKE 53768 and 53775) followed by any AUDCTL commands you wish to use and ending up with the actual POKEs to the sound registers.

The SOUND Statement

When you invoke a SOUND statement of the form SOUND S,T,U,V, the decimal values you use for S, T, U and V are analyzed by the operating system and used to determine the proper values to store in the sound registers. First, as we just mentioned, 53768 is set to zero regardless of the values used. Second, the value of "S" is used to select the proper AUDF and AUDC registers. The legal values of "S" are the integers 0, 1, 2 and 3. See Figure 14.2 to find out what registers are selected in each case.

The second parameter "T" is stored in the selected AUDF register. This determines the tone of the sound. The legal range of "T" is 0-255. Any fractions are rounded to the nearest integer. A T=10.5 has the same effect as T=11, while T=10.495 acts like T=10.

The third parameter "U" is the distortion value. When used in the SOUND format the zeroth bit is *not* ignored! The remaining number is stored in the high three bits of the selected AUDC register. This is the parameter that you have to watch carefully when you use POKEs instead of the SOUND statement. The legal range of "U" is the even numbers from 0-14, with zero being an even number. Do *not* use odd numbers! Also, like "T", fractions are rounded to the nearest integer.

The fourth, and last, parameter is the volume you desire for the selected channel. The computer will store this value in the first four bits of the proper AUDC register. The legal range of values for volume is 0-15, with zero meaning silence and 15 meaning maximum volume. Fractions are rounded here, too.

The volume of each channel is additive, so if you are using two channels, each with a volume of 15, the actual POKEY volume is the sum of the two, or 30. Try not to ever use a combined volume that is greater than 32. This will cause the POKEY to overload and might damage it.

Now let's look at a short comparison between a SOUND statement and the equivalent POKEs. We will assume that a SOUND 0,0,0,0 has already been used to activate the POKEY.

```
SOUND Ø,128,1Ø,8 is the same as
POKE 5376Ø,128:POKE 53761,32*INT(102)+8
```

I used 32*INT(1Ø2) to prevent the POKE from activating the VOLUME ONLY control bit in the off chance that the distortion value had been odd instead of even.

I think we are now ready to try our hand at a few special sound effects. This next section should prove interesting.

Special Effects Routines

Here is one last demo program for you to try before we get into the sound effects routines. This demo shows you some of the strange effects you can get with static sound by making use of waveform interference effects. I won't try to explain them here. Try them and see what you think.

Figure 14.7 — *Sound Effects Demo Number Two*

```
1ØØ REM SOUND2.DEM
11Ø POKE 752,1:
    PRINT CHR$(125):
    SOUND Ø,Ø,Ø,Ø:
    DELTA=1
12Ø POSITION 8,2:
    PRINT "SOUND EFFECTS DEMO #2"
13Ø POSITION 11,11:
    PRINT "PADDLE(Ø) = ":
    REM POKE 53768,4
14Ø SOUND 1,PADDLE(Ø),1Ø,14
15Ø SOUND 2,ABS(PADDLE(Ø)-DELTA),1Ø,7
```

```
160 POSITION 23,11:
    PRINT "     ";CHR$(30);
    CHR$(30);CHR$(30);PADDLE(0)
170 FOR DELAY=0 TO 1:
    NEXT DELAY:GOTO 140
```

Here is a steam train complete with steam whistle. I like this one.

```
20650 REM TRAIN.LST
20651 REPEAT=0:
      DELTA=10:
      FOR TIME=1 TO 90:
      GOTO 20653
20652 DELTA=75:
      FOR TIME=1 TO 50
20653 FOR VOLUME=15 TO 4 STEP -DELTA/100:
      SOUND 0,15,0,VOLUME:
      NEXT VOLUME
20654 DELTA=DELTA+1:
      IF DELTA>75 THEN DELTA=75
20655 NEXT TIME:
      SOUND 0,0,0,0:
      IF REPEAT>2 THEN 20652
20656 REPEAT=REPEAT+1:
      FOR WHISTLE=1 TO 2
20657 FOR VOLUME=2 TO 10 STEP 0.5
20658 SOUND 1,50,10,VOLUME:
      SOUND 2,70,10,VOLUME:
      SOUND 3,90,10,VOLUME:
      NEXT VOLUME
20659 FOR DELAY=1 TO 400:
      NEXT DELAY:
      SOUND 0,0,0,0
20660 FOR VOLUME=10 TO 1 STEP -2
20661 SOUND 1,50,11,VOLUME:
      SOUND 2,70,11,VOLUME:
      SOUND 3,90,11,VOLUME:
      NEXT VOLUME
20662 FOR DELAY=1 TO 50:
      NEXT DELAY
20663 SOUND 1,0,0,0:
      SOUND 2,0,0,0:
      SOUND 3,0,0,0:
      NEXT WHISTLE
20664 GOTO 20652
```

Here is an American police car for your next crime adventure.

```
20670 REM POLICAR.LST
20671 X=50:
      Y=35:
      STEPP=-1
20672 FOR TIME=1 TO 10:
      FOR PITCH=X TO Y STEP STEPP
20673 SOUND 1,PITCH,10,15
20674 FOR DELAY=1 TO 15:
      NEXT DELAY:
      NEXT PITCH
20675 TEMP=X:
      X=Y:
      Y=TEMP:
      STEPP=-STEPP:
      NEXT TIME
20676 GOTO 20671
```

If you ever need a tank, here is a good one.

```
20680 REM TANK.LST
20681 FOR VOICE=0 TO 3:
      SOUND VOICE,255,2,4:
      NEXT VOICE:
      GOTO 20681
20682 REM USE THIS TO STOP TANK MOTORS
20683 FOR VOICE=0 TO 3:
      SOUND VOICE,0,0,0:
      NEXT VOICE
20684 REM PUT REST OF YOUR PROGRAM HERE
```

I'll show you a storm in a few minutes. In the mean time, here are a few peals of thunder to get you in the mood.

```
20690 REM THUNDER.LST
20691 FOR PITCH=5 TO 100 STEP RND(0)+0.2
20692 SOUND 0,PITCH,8,(100*RND(0)+50)/PITCH
20693 SOUND 1,PITCH+20,8,(100*RND(0)+50)/PITCH
20694 NEXT PITCH:SOUND 0,0,0,0:
      SOUND 1,0,0,0
```

I can't think of too many applications that would need a swarm of house flies, but here is one anyway.

```
20700 REM FLIES.LST
20701 SOUND 0,0,0,0
20702 POKE 53760,INT(6*RND(0))+249
20703 POKE 53761,INT(4*RND(0))+167
20704 GOTO 20702
```

Here is a good motor boat sound for your next "Attack of the Swamp Monster" game.

```
20710 REM MOTRBOAT.LST
20711 SOUND 0,255,11,10
20712 FOR COUNT=1 TO 6:
      SOUND 0,0,0,0:
      NEXT COUNT
20713 GOTO 20711
```

Have you ever heard the sound of a manhole cover slowly settling down on a sidewalk? No? This routine gives you an idea of what it would sound like.

```
20720 REM MANHOLE.LST
20721 FOR COUNT=10 TO 0 STEP -0.15
20722 FOR VOLUME=1 TO COUNT:
      SOUND 0,255,8,VOLUME:
      NEXT VOLUME
20723 FOR VOLUME=2*COUNT TO 1 STEP -1:
      SOUND 0,255,8,VOLUME:
      NEXT VOLUME
20724 NEXT COUNT
```

Ahh... There is nothing like being down at the beach, except maybe listening to this sound of the surf.

```
20730 REM SURF.LST
20731 FOR PITCH=0 TO 10:
      SOUND 2,PITCH,8,4
20732 FOR DELAY=1 TO 30:
      NEXT DELAY:
      NEXT PITCH
20733 FOR PITCH=10 TO 0 STEP -1:
      SOUND 2,PITCH,8,4
20734 FOR DELAY=1 TO 300:
      NEXT DELAY:
      NEXT PITCH
20735 GOTO 20731
```

In case your next adventure takes you to the old country, here is an example of a typical European police siren.

```
20740 REM EUROCOP.LST
20741 X=57:
      Y=45:
      TEMP=45
20742 FOR TIME=0 TO 10:
      SOUND 1,TEMP,10,15
20743 FOR DELAY=1 TO 180:
      NEXT DELAY
20744 TEMP=X:
      X=Y:
      Y=TEMP:
      NEXT TIME
20745 GOTO 20742
```

Here is a nice creepy thunder storm for your next "Death in the Crypts" adventure program.

```
20750 REM STORM.LST
20751 FOR COUNT=1 TO 2:
      MAX=INT(256*RND(0))+50:
      WAIT=200*RND(0)
20752 FOR PITCH=1 TO MAX:
      SOUND 0,PITCH,8,15:
      NEXT PITCH
20753 FOR DELAY=1 TO WAIT:
      NEXT DELAY:
      NEXT COUNT:
      SOUND 0,0,0,0
20754 SOUND 1,0,0,15
20755 FOR DELAY=1 TO INT(3000*RND(0)):
      NEXT DELAY
20756 GOTO 20751
```

Shades of Edgar Allen Poe! Is that a tell tale heart I hear?

```
20760 REM HEART.LST
20761 FOR COUNT=1 TO 40:
      SOUND 0,12,3,15:
      NEXT COUNT
20762 FOR COUNT=1 TO 150:
      SOUND 0,0,0,0:
      NEXT COUNT
20763 GOTO 20761
```

Warp three, Mr. Golu. Alpha Centari, here we come!

```
20770 REM TAKEOFF.LST
20771 FOR PITCH=255 TO 1 STEP -1:
      SOUND 0,PITCH,8,8
20772 FOR DELAY=1 TO 5:
      NEXT DELAY:
      NEXT PITCH
```

I tossed an egg into the air. Where it lands I do not care.

```
20780 REM SPLAT.LST
20781 FOR PITCH=30 TO 125 STEP 3
20782 SOUND 1,PITCH,10,INT(PITCH/10)
20783 FOR DELAY=1 TO INT(PITCH/10):
      NEXT DELAY:
      NEXT PITCH
20784 SOUND 1,20,0,14:
      SOUND 2,255,10,15
20785 FOR DELAY=1 TO 100:
      NEXT DELAY
```

Captain, there's a Romulan off the port bow.

```
20790 REM SAUCER1.LST
20791 FOR PITCH=255 TO 195 STEP -1
20792 SOUND 1,PITCH,10,10:
      SOUND 2,PITCH/2,10,15
20793 FOR DELAY=1 TO 10:
      NEXT DELAY
20794 SOUND 1,PITCH+5,0,5:
      SOUND 2,PITCH/2,0,10
20795 FOR DELAY=1 TO 5:
      NEXT DELAY:
      NEXT PITCH
```

The aliens are coming! The aliens are coming!

```
20800 REM SAUCER2.LST
20801 SOUND 0,0,0,0:
      REM INIT POKEY
20802 VOLUME=8:
      PITCH=100
20803 POKE 53768,4:
      REM AUDCTL
20804 POKE 53761,160+VOLUME:
      REM AUDC1
20805 POKE 53765,160+VOLUME+4:
      REM AUDC3
20806 POKE 53760,PITCH:
      REM AUDF1
20807 POKE 53764,PITCH/2:
      REM AUDF3
20808 GOTO 20802
```

This is your typical klaxon siren.

```
20810 REM KLAXON.LST
20811 FOR COUNT=1 TO 10:
      FOR PITCH=1 TO 10
20812 SOUND 0,100-PITCH,10,10:
      NEXT PITCH
20813 SOUND 0,90,10,14:
      SOUND 1,95,10,14:
      SOUND 2,20,2,4
20814 FOR DELAY=1 TO 200:
      NEXT DELAY
20815 SOUND 1,0,0,0:
      SOUND 2,0,0,0
20816 FOR PITCH=1 TO 5:
      SOUND 0,90+PITCH,10,8:
      NEXT PITCH
20817 SOUND 0,0,0,0:
      FOR DELAY=1 TO 100:
      NEXT DELAY:
      NEXT COUNT
```

The Nazi dive bombers are on us, sir.

```
20820 REM BOMB.LST
20821 DURATION=10:
      VOLUME1=4
20822 FOR PITCH=30 TO 75:
      SOUND 0,PITCH,10,VOLUME1:
      SOUND 1,PITCH+3,10,0.7*VOLUME1
20823 FOR DELAY=1 TO 3*DURATION:
      NEXT DELAY
20824 VOLUME1=1.03*VOLUME1:
      NEXT PITCH
20825 SOUND 2,35,8,12:
      VOLUME1=15:
      VOLUME2=15:
      VOLUME3=15:
      PITCH=DURATION+5:
      DELTA=0.79+DURATION/100
20826 SOUND 0,PITCH,8,VOLUME1:
      SOUND 1,PITCH+20,8,VOLUME2:
      SOUND 2,PITCH+50,8,VOLUME3
20827 VOLUME1=DELTA*VOLUME1:
      VOLUME2=(DELTA+0.05)*VOLUME2:
      VOLUME3=(DELTA+0.08)*VOLUME3
20828 IF VOLUME3>1 THEN 20826
20829 SOUND 0,0,0,0:
      SOUND 1,0,0,0:
      SOUND 2,0,0,0:
      GOTO 20821
```

The bombs are exploding all around us.

```
20830 REM EXPLODE.LST
20831 DURATION=8:
      VOLUME1=5
20832 SOUND 2,35,8,12:
      VOLUME1=15:
      VOLUME2=15:
      VOLUME3=15:
      PITCH=DURATION+5:
      DELTA=0.79+DURATION/100
20833 SOUND 0,PITCH,8,VOLUME1:
      SOUND 1,PITCH+20,8,VOLUME2:
      SOUND 2,PITCH+50,8,VOLUME3
20834 VOLUME1=DELTA*VOLUME1:
      VOLUME2=(DELTA+0.05)*VOLUME2:
      VOLUME3=(DELTA+0.08)*VOLUME3
20835 IF VOLUME3>1 THEN 20833
20836 SOUND 0,0,0,0:
      SOUND 1,0,0,0:
      SOUND 2,0,0,0
```

Useful Utilities

The Atari home computer is a very powerful tool, if you have the correct software for it. So far in this book we have talked about many specific applications, but it is impossible to cover everything in a single volume. Our chapters have been thematic in that respect even though we have covered a lot of topics in a few hundred pages. This chapter is my catch-all for useful programs that I couldn't work into the rest of the book. You will hopefully find the special routines in this chapter of as much use as I have.

There are basically four routines in this chapter. The first three are things I have found to be absolute necessities when using disk drives. The first program is a special utility that creates AUTORUN.SYS files for you. The second program is a handy disk catalog routine, and the third program is a helpful diagnostic tool for your disk drives. All three of these programs are either BASIC or a hybrid BASIC containing a machine language subroutine. The fourth program is a complete miniature DOS that is callable from either BASIC or the ASSEMBLER/EDITOR cartridge.

AUTOGO — Creates AUTORUN.SYS Files

This is one of the first routines I wrote. AUTOGO is a program that will help you to create your own AUTORUN.SYS files. Before we go into how to use AUTOGO, let's briefly touch on exactly what an AUTORUN.SYS file is.

When you boot up your computer (turn it on) with a DOS compatible disk in drive 1, the computer first looks for DOS.SYS. If DOS.SYS is found on the disk, the computer loads DOS.SYS and starts running it. One of the first things DOS.SYS does is to go back to the disk and look for a file named AUTORUN.SYS. If DOS finds such a file, the file is automatically loaded in and started running.

If you had a binary load file (machine language) program that you wanted to be started everytime you booted a particular disk, you could make this happen by renaming that file to AUTORUN.SYS. After that, whenever you booted that disk, your program would run automatically. Things are a little bit different if your program is in BASIC. You will need a different kind of AUTORUN.SYS file.

AUTOGO creates a special AUTORUN.SYS file that will be booted in as we just described, but rather than being a game, this AUTORUN.SYS file tells the computer to RUN"D:FILENAME. The assumptions are: (1) that the BASIC cartridge is in the computer, and (2) that the file name referred to is actually on that disk. This is an extremely powerful tool for BASIC programmers.

AUTOGO is extremely easy to use. First, load AUTOGO and run it. The screen will go into a colorful GRAPHICS 2 display that shows you the title and purpose of the routine. You are then prompted to enter the name of a file. The filename follows the usual conventions. The primary name of the file must be no longer than eight characters and must begin with a capital letter from A-Z. The rest of the name must contain only capital letters and/or numbers. The file extender can be any combination of capital letters and numbers, but it may not be any longer than three characters. You don't have to specify a drive number. The drive is always assumed to be "D1:". AUTOGO will stuff this file name into a customized AUTORUN.SYS file on drive #1.

Let's work a quick example. Suppose that you have a BASIC program which is stored on a disk under the file name "HOTSTUFF.V01", and you would like to have this program automatically load and run each time you booted that disk. First, make sure that a copy of DOS.SYS is also on the disk. Then, load and run AUTOGO. When you are prompted to enter a file name, type in HOTSTUFF.V01, and a special AUTORUN.SYS file will be written on the disk. Once this is done and AUTOGO has put the computer back in GRAPHICS 0, turn the computer OFF and then back ON again to cause a boot. If all goes well, your program "HOTSTUFF.V01" should automatically be loaded and run as if you had typed in "RUN"D:HOTSTUFF.V01".

AUTOGO is written to prevent accidental damage to your BASIC program. All inputs and outputs are carefully checked before execution, and an appropriate ERROR message is printed out whenever you make a mistake. Of course, this won't prevent you from entering the wrong file name, but that error is hardly fatal.

AUTOGO will not accept anything except a capital letter for the first character of the file name. It also rejects any illegal input for any other character in the file name. When the file name you are entering reaches a length of eight or when you press the "." key, a period, ".", is inserted at the end of the file name. If you make a mistake before you reach the end of the file name, you can correct the error by pressing the BACK SPACE key. This will erase the last character you entered. You can then re-enter the proper character.

That's all there is to it! When AUTOGO is finished, it stops running and puts the computer back into GRAPHICS 0. At this point you can test the results by turning the computer OFF and then back ON again. Your BASIC program should boot up and begin running all by itself.

Figure 15.1 — *AUTOGO – Creates AUTORUN.SYS Files*

```
100 REM AUTOGO - CREATE AUTORUN.SYS
110 GRAPHICS 18:
    POKE 752,1
120 PRINT #6;"       autogo"
130 PRINT #6;"███████████████████"
140 PRINT #6;" CREATES DOS 2.0S"
150 PRINT #6;" AUTORUN.SYS FILE"
160 DIM A$(128),FILE$(12)
170 PRINT #6:
    PRINT #6:
    PRINT #6;"  enter filename "
180 TRAP 650:
    OPEN #1,4,0,"K:":
    FIRST=1
```

NOTE: █ = Inverse Shift Dash!

```
190 FOR X=1 TO 8
200 GET #1,KEY
210 IF FIRST AND (KEY<65 OR KEY>90) THEN FIRST=1:
    GOTO 200
220 IF KEY=155 THEN POP :
    GOTO 330
230 FIRST=0:
    IF KEY=46 THEN POP :
    GOTO 270
240 IF KEY=126 AND X>1 THEN X=X-1:
    POSITION X+3,8:
    PRINT #6;" ";:
    GOTO 200
250 IF  NOT ((KEY>47 AND KEY<58) OR (KEY>64 AND KEY<91))
    THEN 200
260 POSITION X+3,8:
    PRINT #6;CHR$(KEY);:
    FILE$(X,X)=CHR$(KEY):
    NEXT X
270 PRINT #6;".";:
    FILE$(LEN(FILE$)+1)=".":
    SIZE=LEN(FILE$)+1
280 FOR X=SIZE TO SIZE+3
290 GET #1,KEY:
    IF KEY=155 THEN POP :
    GOTO 330
300 IF KEY=126 AND X>12 THEN X=X-1:
    POSITION X+3,8:
    PRINT #6;" ";:
    GOTO 290
310 IF  NOT ((KEY>47 AND KEY<58) OR (KEY>64 AND KEY<91))
    THEN 290
320 POSITION X+3,8:
    PRINT #6;CHR$(KEY);:
    FILE$(LEN(FILE$)+1)=CHR$(KEY):
    NEXT X
330 CLOSE #1
340 TRAP 630:
    OPEN #1,8,0,"D:AUTORUN.SYS"
350 TRAP 640:
    FOR COUNT=1 TO 4:
    READ BYTE:
    PUT #1,BYTE:
    NEXT COUNT
360 A$(1,3)="RUN":
    A$(4,4)=CHR$(34):
    A$(5,6)="D:"
370 A$(LEN(A$)+1)=FILE$:
    A$(LEN(A$)+1)=CHR$(34)
```

```
380 L=123+LEN(A$)-1:
    PUT #1,L:
    PUT #1,6:
    FOR COUNT=1 TO 123:
    READ BYTE
390 IF COUNT=64 THEN PUT #1,LEN(A$)-1:
    GOTO 410
400 PUT #1,BYTE
410 NEXT COUNT
420 FOR COUNT=LEN(A$) TO 1 STEP -1:
    PUT #1,ASC(A$(COUNT,COUNT)):
    NEXT COUNT
430 FOR COUNT=1 TO 8:
    READ BYTE:
    PUT #1,BYTE:
    NEXT COUNT:
    CLOSE #1:END
440 DATA 255,255,0,6
450 DATA 162,0,189,26,3,201,69,240
460 DATA 5,232,232,232,208,244,232,142
470 DATA 105,6,189,26,3,133,205,169
480 DATA 107,157,26,3,232,189,26,3
490 DATA 133,206,169,6,157,26,3,160
500 DATA 0,162,16,177,205,153,107,6
510 DATA 200,202,208,247,169,67,141,111
520 DATA 6,169,6,141,112,6,169,10
530 DATA 141,106,6,96,172,106,6,240
540 DATA 9,185,123,6,206,106,6,160
550 DATA 1,96,138,72,174,105,6,165
560 DATA 205,157,26,3,232,165,206,157
570 DATA 26,3,104,170,169,155,160,1
580 DATA 96,0,0,0,0,0,0,0
590 DATA 0,0,0,0,0,0,0,76
600 DATA 0,0,0
610 DATA 255,255,226,2,227,2,0,6
620 REM ERROR HANDLERS
630 GOSUB 660:
    PRINT "ERROR WHILE OPENING DISK FILE":
    GOSUB 680:RUN
640 GOSUB 660:
    PRINT "ERROR WHILE WRITING TO DISK":
    GOSUB 680:RUN
650 GOSUB 660:
    PRINT "ERROR DURING KEYBOARD INPUT":
    GOSUB 680:
    RUN
660 CLOSE #1:
    GRAPHICS 0:
    POSITION 2,10:
    POKE 752,1:
    RETURN
```

```
670 REM TIME DELAY
680 FOR DELAY=1 TO 500:
    NEXT DELAY:
    RETURN
```

CATALOG – Disk Catalog Program

If you have reached the point where you have at least ten disks of software, this program will be of use to you. I use CATALOG in two ways. First, I use it to make a handy label for the jacket of each disk, showing what is on the disk inside. Second, I use it to make a "catalog," or complete listing of all of the programs I have in my library.

When you RUN CATALOG it comes up with my favorite GRAPHICS 2 display showing the name of the program. In this particular program the purpose is implicit in the title so the routine jumps right in and displays three lines of 17 dashes. These lines are the limits of the title you can give to a particular disk. You can get a printout of a directory from DOS by using the "A" option and answering it with "D1:,P:", but this doesn't let you assign titles to the disks. This catalog program will let you assign detailed titles to each of your disks.

As usual, the program is fully buffered against input or output errors. When you are finished with one of the 17 character lines, the program automatically moves down to the next line until all three lines are filled. If you wish to end any line with less than 17 characters, all you have to do is press the RETURN key. When the third line is completed, CATALOG will LPRINT a neat title and disk directory and display it on the screen at the same time. Once the label has been printed, the program re-initializes itself and waits for you to catalog another disk. You can stop the program at any time by pressing the BREAK key. You can back up to correct a mistake by pressing the BACK SPACE key.

Figure 15.2 — *CATALOG – Disk Catalog Program*

```
100 REM CATALOG - DISK CATALOGER
110 DIM FIRST$(17),SECOND$(17),THIRD$(17),NAME$(17):
    POKE 752,0:
    POKE 82,2:
    POKE 83,39
120 REM PUT FORM ON THE SCREEN
130 GRAPHICS 18:
    POKE 708,40:
    POKE 709,216:
    POKE 710,150:
    POKE 711,254
140 PRINT #6;"   disk catalog":
    PRINT #6;"■■■■■■■■■■■■■■■■■■■■":
    PRINT #6;" enter disk title"
150 OPEN #1,4,0,"K:"
160 FOR LINE=1 TO 3:
    POSITION 1,4+2*LINE:PRINT #6;"■■■■■■■■■■■■■■■■■":
    NEXT LINE
170 REM GET USER INPUTS
```

Note: ■ = *Inverse Shift Dash*

```
180 FOR LINE=1 TO 3:
    FOR CHAR=1 TO 17
190 POSITION CHAR,4+2*LINE:
    PRINT #6;"█";:
    POSITION CHAR,4+2*LINE:
    PRINT #6;"-";:
    IF PEEK(753)=0 THEN 190
200 POKE 753,0:
    POSITION CHAR,4+2*LINE:
    GET #1,KEY
210 REM RETURN ENDS USER INPUT
220 IF KEY=155 THEN POSITION CHAR,4+2*LINE:
    PRINT #6;"█":
    GOTO 280
230 REM CHECK FOR DELETE CHARACTER
240 IF CHAR>1 AND KEY=126 THEN CHAR=CHAR-1:
    PRINT #6;"█";:
    GOTO 190
250 IF KEY<32 OR KEY>122 THEN 190
260 POSITION CHAR,4+2*LINE:
    PUT #6,KEY
270 NEXT CHAR
280 NEXT LINE
290 REM PREPARE DATA FOR PRINTING
300 FOR CHAR=1 TO 17:
    POSITION CHAR,6:
    GET #6,KEY:
    IF CHR$(KEY)="█" THEN KEY=32
310 FIRST$(CHAR,CHAR)=CHR$(KEY):
    NEXT CHAR
320 FOR CHAR=1 TO 17:
    POSITION CHAR,8:
    GET #6,KEY:
    IF CHR$(KEY)="█" THEN KEY=32
330 SECOND$(CHAR,CHAR)=CHR$(KEY):
    NEXT CHAR
340 FOR CHAR=1 TO 17:
    POSITION CHAR,10:
    GET #6,KEY:
    IF CHR$(KEY)="█" THEN KEY=32
350 THIRD$(CHAR,CHAR)=CHR$(KEY):
    NEXT CHAR:
    GRAPHICS 0:
    POKE 82,10:
    PRINT :
    CLOSE #1
360 TRAP 470:
    IF FIRST$<>"                 " THEN LPRINT FIRST$:      Note: 17 spaces
    PRINT FIRST$
370 IF SECOND$<>"                 " THEN LPRINT SECOND$:    Note: 17 spaces
    PRINT SECOND$
```

```
380 IF THIRD$<>"                 " THEN LPRINT THIRD$:
    PRINT THIRD$:
    LPRINT :
    PRINT
390 LPRINT :
    PRINT
400 REM OPEN DISK DIRECTORY
410 TRAP 450:
    OPEN #1,6,0,"D:*.*":
    TRAP 460
420 REM READ DISK DIRECTORY
430 INPUT #1,NAME$:
    LPRINT NAME$:
    PRINT NAME$:
    TRAP 480:
    GOTO 430
440 REM ERROR HANDLERS
450 GOSUB 490:
    PRINT "ERROR WHILE OPENING DISK DIRECTORY":
    GOSUB 510:
    RUN
460 GOSUB 490:
    PRINT "ERROR WHILE READING DISK DIRECTORY":
    GOSUB 510:
    RUN
470 GOSUB 490:
    PRINT "ERROR WHILE TRYING TO ACCESS PRINTER":
    GOSUB 510:
    RUN
480 LPRINT :
    LPRINT :
    LPRINT :
    GOSUB 510:
    RUN
490 CLOSE #1:
    GRAPHICS 0:
    POKE 82,2:
    POKE 752,1:
    PRINT CHR$(125);CHR$(253):
    POSITION 2,10:
    RETURN
500 REM TIME DELAY
510 FOR DELAY=1 TO 500:
    NEXT DELAY:
    RETURN
```

Note: 17 Spaces

RPMTEST - Disk RPM Tester

Most of the time I don't have any trouble with my disk drives. However, every disk drive I have owned, and those that belong to friends of mine, all have one peculiar problem. The disk speed (RPM's) tends to drift as time goes by. This RPM drift was a major problem with some of the early Atari 810 drives, but the severity of the problem has been reduced tremendously with the newer models. One difficulty is that Atari doesn't bother to change the part number when they change the design of the part. Even a letter change would help. Even if we knew whether we had one of the newer, better designs or one of the older ones, all of us should periodically monitor the actual speed (RPM) of our disk drives. RPMTEST is a tool for doing this task.

When you run RPMTEST, it will come up with my favorite GRAPHICS 2 display showing the name of the program. You will be prompted to enter a drive number between one and four. Your response will be displayed on the screen and you will hear your drive come ON. Be sure a formatted disk is in the designated drive before you do this. You can use any old disk since the only I/O operation used is a READ, thus insuring the safety of whatever you have on the disk.

RPMTEST will then take a sample of 100 disk reads and compute the speed of the specified drive. The proper range for this reading is 288 ±4 RPM's. A reading of 288 is flagged as "PERFECT" in a congenial blue. Any reading between 284 and 292 other than 288 is shown as "O.K.", also in blue. Any other reading means that something is not normal, and the results will be printed in red to alert you. Any reading that comes up red means that you should either adjust the speed back into the proper range or take your drive to a service center.

There is one special exception to the "repair on red" rule. This program was written specifically for the Atari 810 disk drive. It is possible that other brands will operate at some other RPM. Most non-Atari drives will also operate only at the normal 288 RPM to be compatable with those copy-protected programs that will only load at 288 ±4 RPM's (programs using something called "duplicate sector IDs" and "sector skewing" fall into this category). However, I have modified 810 disk drive that runs at an apparent 740 RPM, so I Know that there may be occasions where you might run this program on a non-standard disk drive. If the RPM reading is 500 or greater, RPMTEST will assume that you have one of those special drives and will tell you so.

Figure 15.3 — *RPMTEST - Disk RPM Tester*

```
100 REM RPMTEST - DISK RPM TESTER
110 GRAPHICS 18:
    POKE 752,1:
    GOSUB 390
120 PRINT #6;"  DISK RPM TESTER "
130 PRINT #6;"■■■■■■■■■■■■■■■■■■"
140 PRINT #6;" push system reset "
150 PRINT #6;" to terminate read ":
    PRINT #6
160 PRINT #6;"DRIVE NUMBER? ";
170 OPEN #1,4,0,"K:":
    GET #1,KEY:
    CLOSE #1
```

Note: ■ = *Inverse Shift Dash*

```
180 DRIVE=KEY-48:
    IF DRIVE<1 OR DRIVE>4 THEN 170
190 PRINT #6;DRIVE:
    PRINT #6;"RPM READING:    ":
    POKE 1610,DRIVE
200 DUMMY=USR(1536):
    LSB=PEEK(1611):
    MSB=PEEK(1612):
    TIME=(LSB+256*MSB)/3600:
    RPM=INT(100TIME+0.5)
210 SOUND 1,50,10,15:
    FOR FLASH=1 TO 100:
    POSITION 14,6:
    PRINT #6;"   ":
    POSITION 14,6:
    PRINT #6;RPM
220 NEXT FLASH:
    SOUND 1,0,0,0
230 IF RPM>500 THEN PRINT #6:
    PRINT #6;" nonstandard drive":
    PRINT #6;" reading not valid":
    GOTO 200
240 PRINT #6:
    PRINT #6;"SPEED IS          "
250 POSITION 10,8
260 IF RPM<284 THEN PRINT #6;"too slow":
    GOTO 200
270 IF RPM>292 THEN PRINT #6;"too fast":
    GOTO 200
280 IF RPM=288 THEN PRINT #6;"PERFECT":
    GOTO 200
290 PRINT #6;"O.K.":
    GOTO 200
300 DATA 104,169,1,141,10,3,169,0
310 DATA 141,11,3,141,4,3,169,5
320 DATA 141,5,3,173,74,6,141,1
330 DATA 3,169,82,141,2,3,169,5
340 DATA 141,73,6,32,83,228,206,73
350 DATA 6,208,248,169,100,141,73,6
360 DATA 169,0,133,19,133,20,32,83
370 DATA 228,206,73,6,208,248,165,20
380 DATA 164,19,141,75,6,140,76,6,96
390 MLSTART=1536:
    MLEND=1608
400 FOR X=MLSTART TO MLEND:
    READ Y:
    POKE X,Y:
    NEXT X:
    RETURN
```

MINIDOS — DOS Functions From BASIC

This program, MINIDOS, is probably the most sophisticated routine in this book. In simple terms, it is a miniature disk operating system (DOS) that enables you to perform many disk functions without ever leaving BASIC or the ASSEMBLER/EDITOR. I have included the source listing, for those of you who would like to analyze or play with the program. Also included is a BASIC program that will create a binary load file version of MINIDOS for you. Both versions are fully remarked to help you see what does what.

MINIDOS supports the following functions:

Figure 15.4 — *MINIDOS Functions*

COMMAND	FUNCTION
A	Go to Atari DOS
B	RUN BASIC (or ASM/ED Cartridge)
D	Display a disk directory
F	Format a disk
K	Kill (delete) a file
L	Lock a file
R	Rename a file
U	Unlock a file

Anytime you type in "DOS", you will see the MINIDOS menu instead of going to normal DOS. The MINIDOS menu looks like this:

Figure 15.5 — *MINIDOS Menu*

```
 MINIDOS
ADOS
BASIC
DIRECTORY
FORMAT
KILL
LOCK
RENAME
UNLOCK
```

MINIDOS sits down on page six, so you will have to be careful not to overwrite it. Essentially MINIDOS does some of the functions that DOS.SYS does. Both routines use the CIO routines in the operating system to do what they do. MINIDOS simply resides in a different memory block and contains fewer commands than a normal DOS. I went to extreme lengths to minimize the size of MINIDOS and I was still forced to use the stack area to keep the program to a size that would fit on a single page.

The most important advantage of a program like MINIDOS is that it enables you to quickly execute many of the more common DOS functions without ever actually leaving the cartridge. When you are using MINIDOS, all of your existing program in memory is left undisturbed. When you exit MINIDOS, you are returned to the cartridge program with all of your program still there. You can achieve something like this by using the MEM.SAV option in normal Atari DOS, but that method is *very* slow. On top of that, MEM.SAV requires that the disk you are using have a file named MEM.SAV on it. My problem was that I kept forgetting to create the MEM.SAV file and had to do needed file manipulations in an even more time consuming roundabout way. I suggest that you store MINIDOS on your program development disks with the file name AUTORUN.SYS so it will always be in the computer when you need it. Using MINIDOS should save you many hours of needless hassle.

MINIDOS Command Descriptions

ADOS — Once you have loaded MINIDOS, the MINIDOS menu will be displayed anytime you enter the command DOS from BASIC or the ASM/ED cartridge. If you really need to get back to the full Atari DOS program to perform file or disk copying, then you can get to Atari DOS by using this command, ADOS. Doing this will not damage MINIDOS. It will still be there when you go back to the cartridge. Of course, your program won't be there unless you had a MEM.SAV file on the disk.

BASIC — Using this command will cause you to exit MINIDOS and return to BASIC or the ASM/ED cartridge via a warm start. If you are using some other kind of BASIC or ASM/ED, I can't guarantee the results. The exit from MINIDOS jumps (transfers program control) to a particular location (**$A04D**) in the BASIC and ASM/ED cartridges. Another cartridge program or a disk based program probably won't use these particular addresses in the same way. The only thing you can do is try it and see what happens. When you return to BASIC, you will get the normal READY on the screen. When you return to the Assembler/Editor cartridge you are dumped into DEBUG. All you have to do is enter an "X" followed by a RETURN to get back to the EDIT mode.

DIRECTORY — This command operates similar to the "A" command in the Atari DOS menu. When you call this command, it will ask, "D#?". You should respond with a number from 1 to 4. MINIDOS will then search that disk's directory and display a directory listing for you. The only display option is "to the screen," and no search parameters are allowed. This shouldn't be a problem since those DOS options are not frequently used anyway. Do not enter a "D" before the number.

FORMAT — This is a standard disk format option. When you call up this routine, it will display "FD#?". When you respond with "D" and a number from 1 to 4, the routine will call up the format routine in the operating system. A disk in the specified drive will be formatted. Be careful with this routine. To keep the code to a minimum, it was necessary to leave out any failsafe options. If the first character in your response in not "D" the command will be aborted.

KILL — This is just your familiar DELETE FILE command under a new name. I like the term "KILL" used to denote this operation. There are several reasons. First, I have already used "D" for my directory command. Second, I initially learned about computers by using an old mainframe dinosaur that deleted files with, yes — you guessed it, a "KILL" command. Also, "delete" doesn't sound too serious, but "KILLing" something sounds serious. "Sorry, Honey, I can't come to supper right now. I have to *KILL* a file!" See what I mean?

LOCK — This command will ask you for a "FN?". You must specify the entire file spec. For example, "D1: TESTFILE.005". If the first two characters of your answer are not a "D" followed by a number from 1 to 4, then the command is aborted. Personally, I almost never use a LOCK or UNLOCK command since the only thing that pays any attention to them is a normal Atari DOS. A true sector copier or a FORMAT command will wipe out a file whether it is "LOCKED" or not. All that using LOCKed files can do is give you a false sense of security. I stuck LOCK and UNLOCK in MINIDOS at the request of a friend who does like to use them. Oh well, each to his own.

RENAME — This command is what led to the original creation of MINIDOS. Many times I would be working on a program and want to save it under a special file name. Previous versions of the program would be stored under the same name with a different extender. This command allowed me to change the last "new" version to some other name. I could then save the real new version under the name I wanted. This command asks you for "FN?". You should respond with the complete file spec for the original file followed by a comma and the new file name. Do not use "D#" after the comma! This command assumes that the renamed file is in the same drive as it started. After all, you are renaming a file, not moving it. For example, suppose you wanted to change the name of FILETEST.005 on a disk in drive #1 to FILETEST.006. You would do this by first going into MINIDOS with the DOS command from BASIC. Then you would press the "R" key after the MINIDOS menu is displayed. When the "FN?" prompt appears, you would answer — D1:TESTFILE.005,TESTFILE.006 followed by a RETURN. The file name will then be changed. There is no provision in MINIDOS for error messages since they take up valuable memory. Therefore, if you give a wrong parameter, the command is simply aborted.

UNLOCK — This command is the complement of the LOCK command. It removes the "LOCK" from a locked file. I have already told you my opinion of LOCK and UNLOCK, so let's leave it at that.

There you have it — a miniature user's manual for a miniature disk operating system. MINIDOS is, without a doubt, the most useful routine in this book. After you have used it a few times, I am sure you will agree with me.

Figure 15.6 — *MINIDOS.BAS - DOS Functions From BASIC*

```
100 REM MINIDOS.BAS - DOS FROM BASIC
110 REM
120 REM CREATE MINIDOS DISK FILE
130 OPEN #2,8,0,"D:MINIDOS.OBJ"
140 FOR X=1 TO 342:
    READ Y:
    PUT #2,Y:
    NEXT X
150 CLOSE #2:
    END
160 REM SET UP DISK FILE HEADER
170 DATA 255,255,0,6,253,6
180 REM MINIDOS PROGRAM FOR PAGE SIX
190 DATA 160,37,162,50,169,11,32,209
200 DATA 6,32,19,6,32,221,6,16
210 DATA 239,48,246,160,245,162,2,32
220 DATA 84,6,32,90,6,160,6,140
```

```
230 DATA 90,3,136,240,63,185,234,6
240 DATA 205,8,1,208,245,185,239,6
250 DATA 72,201,254,208,19,160,250,162
260 DATA 3,32,84,6,32,90,6,173
270 DATA 8,1,201,68,208,30,240,10
280 DATA 160,247,162,3,32,84,6,32
290 DATA 90,6,104,96,169,11,32,186
300 DATA 6,96,160,8,162,40,169,5
310 DATA 32,209,6,96,173,8,1,201
320 DATA 77,208,62,160,251,162,3,32
330 DATA 84,6,160,1,162,1,169,5
340 DATA 32,209,6,160,58,140,2,1
350 DATA 160,0,169,3,32,223,6,169
360 DATA 19,141,88,3,169,5,32,221
370 DATA 6,48,11,160,7,162,20,169
380 DATA 9,32,209,6,16,233,169,12
390 DATA 32,221,6,32,90,6,76,0
400 DATA 6,201,65,208,10,169,68,205
410 DATA 9,1,208,3,76,159,23,76
420 DATA 77,160,72,169,6,141,69,3
430 DATA 104,140,68,3,142,72,3,162
440 DATA 0,142,73,3,142,89,3,240
450 DATA 19,72,169,1,141,69,3,141
460 DATA 85,3,104,208,228,160,8,140
470 DATA 84,3,162,16,157,66,3,32
480 DATA 86,228,96,76,85,68,82,70
490 DATA 35,36,33,32,254,29,62,70
500 DATA 78,63,198,68,35,63
510 REM CHANGE DOSVEC TO $600
520 REM $A=00:$B=06
530 DATA 10,0,11,0,0,6
540 REM CHANGE DOS COMMAND TO $600
550 REM $1546=00
560 DATA 70,21,70,21,0
570 REM $154A=06
580 DATA 74,21,74,21,6
590 REM PUT D*.* AT $100
600 DATA 0,1,7,1,68,49,58,42
610 DATA 46,42,155,127
620 REM PUT MINIDOS COMMANDS AT $125
630 DATA 37,1,86,1,125
640 REM UNLOCK
650 DATA 204,79,67,75,155
660 REM LOCK
670 DATA 213,78,76,79,67,75,155
680 REM DELETE
690 DATA 196,69,76,69,84,69,155
700 REM RENAME
710 DATA 210,69,78,65,77,69,155
720 REM FORMAT
730 DATA 198,79,82,77,65,84,155
740 REM MENU - DISK DIRECTORY
```

```
750 DATA 205,69,78,85,155
760 REM ADOS - ATARI DOS 2.0S
770 DATA 193,196,79,83,155
780 REM BASIC - GOTO CARTRIDGE
790 DATA 194,65,83,73,67,155
```

Figure 15.7 — *MINIDOS - Machine Language Source Listing*

```
               1000 ;MINIDOS - DOS FUNCTIONS FROM BASIC
               1010 ;
               1020 ;ALLOWS ACCESS TO SOME DOS FUNCTIONS
               1030 ;WHILE USING BASIC OR THE ASM/ED CARTRIDGE.
               1040 ;
               1050 ;MAIN ROUTINE RESIDES ON PAGE SIX.
               1060 ;"DOS" COMMAND IS CHANGED TO POINT TO THIS ROUTINE.
               1070 ;MINIDOS MENU IS STORED AT BOTTOM OF PAGE ONE.
               1080 ;
               1090 ;PLEASE OBSERVE USUAL PAGE SIX CAVEATS
               1100 ;
               1110 ;ORIGIN ON PAGE SIX
               1120 ;
0000           1130        *=     $0600         ;NOT RELOCATABLE
               1140 ;
               1150 ;SET UP POINTERS
               1160 ;
0102           1170 FILE   =      $0102
0108           1180 ANSWER =      $0108
0342           1190 IOCB   =      $0342
0340           1200 IOCB0  =      $0340
0350           1210 IOCB1  =      $0350
0600           1220 ORG    =      $0600
179F           1230 ADOS   =      $179F
A04D           1240 CART   =      $A04D
E456           1250 CIOV   =      $E456
               1260 ;
0600 A025      1270        LDY    #$25          ;DISPLAY MINIDOS MENU
0602 A23F      1280        LDX    #$3F
0604 A90B      1290        LDA    #$0B
0606 20BA06    1300        JSR    MENU
0609 201306    1310 AGAIN  JSR    OPTION        ;GET MENU OPTION
060C 20C606    1320        JSR    EXIT
060F 10EF      1330        BPL    ORG           ;START MINIDOS AGAIN
0611 30F6      1340        BMI    AGAIN         ;GO GET ANOTHER INPUT
0613 A0F0      1350 OPTION LDY    #$F0          ;MOVE DOWN ONE LINE AND
0615 A202      1360        LDX    #$02          ;PRINT A ">"
0617 20D406    1370        JSR    FETCH
061A 20DA06    1380        JSR    FUNC
061D A006      1390        LDY    #$06          ;CHECK ANSWER VS DICTIONARY
```

```
061F 8C5A03 1400        STY     IOCB1+10
0622 88     1410 PARSE  DEY
0623 F02F   1420        BEQ     WHEN
0625 B9E406 1430        LDA     INPUTS,Y
0628 CD0801 1440        CMP     ANSWER          ;COMPARE INPUT WITH ANSWER
062B D0F5   1450        BNE     PARSE           ;NO MATCH? THEN CHECK AGAIN
062D B9EA06 1460        LDA     TOOLS,Y         ;GET SPECIAL CODE
0630 48     1470        PHA
0631 C9FE   1480        CMP     #$FE            ;OK TO FORMAT A DISK?
0633 D013   1490        BNE     HERE            ;NO? MUST BE SOMETHING ELSE
0635 A0F5   1500        LDY     #$F5            ;PRINT "FD#?"
0637 A204   1510        LDX     #$04
0639 20D406 1520        JSR     FETCH
063C 20DA06 1530        JSR     FUNC            ;ASK FOR DISK TO FORMAT
063F AD0801 1540        LDA     ANSWER
0642 C944   1550        CMP     #$44            ;IS FIRST CHAR A "D"?
0644 D00E   1560        BNE     WHEN            ;NO? THEN ABORT COMMAND
0646 F00A   1570        BEQ     THERE           ;YES? THEN GET DRIVE NUMBER
0648 A0F2   1580 HERE   LDY     #$F2            ;PRINT "FN?"
064A  203   1590        LDX     #$03
064C 20D406 1600        JSR     FETCH
064F 20DA06 1610        JSR     FUNC            ;DO OTHER DOS FUNCTION
0652 68     1620 THERE  PLA
0653 60     1630        RTS
0654 AD0801 1640 WHEN   LDA     ANSWER
0657 C944   1650        CMP     #$44            ;IS INPUT A "D"?
0659 D03E   1660        BNE     DOSCHEK
065B A0F6   1670        LDY     #$F6            ;PRINT "D#?"
065D  203   1680        LDX     #$03
065F 20D406 1690        JSR     FETCH
0662 A001   1700        LDY     #$01            ;GET DRIVE NUMBER
0664 A201   1710        LDX     #$01
0666 A905   1720        LDA     #$05
0668 20BA06 1730        JSR     MENU
066B A03A   1740        LDY     #$3A
066D 8C0201 1750        STY     FILE
0670 A000   1760        LDY     #$00
0672 A903   1770        LDA     #$03
0674 20C806 1780        JSR     DITT
0677 A913   1790 DUTT   LDA     #$13            ;SET BUFFER FOR FILE NAME
0679 8D5803 1800        STA     IOCB1+8
067C A905   1810        LDA     #$05
067E 20C606 1820        JSR     EXIT
0681 300B   1830        BMI     GREBO
0683 A007   1840        LDY     #$07            ;OPEN DISK DIRECTORY
0685 A214   1850        LDX     #$14
0687 A909   1860        LDA     #$09
0689 20BA06 1870        JSR     MENU
068C 10E9   1880        BPL     DUTT
068E A90C   1890 GREBO  LDA     #$0C            ;CLOSE DISK DIRECTORY
0690 20C606 1900        JSR     EXIT
0693 20DA06 1910        JSR     FUNC
```

```
0696 4C0006 1920          JMP     ORG             ;RESTART MINIDOS
0699 C941   1930 DOSCHEK  CMP     #$41            ;IS INPUT AN "A"?
069B D003   1940          BNE     BASIC
069D 4C9F17 1950          JMP     ADOS            ;GO TO ATARI DOS MENU
06A0 4C4DA0 1960 BASIC    JMP     $A04D           ;GO TO CARTRIDGE
06A3 48     1970 TOOLKIT  PHA
06A4 A906   1980          LDA     #$06            ;POINT BUFFER TO $600
06A6 8D4503 1990          STA     IOCB0+5
06A9 68     2000          PLA
06AA 8C4403 2010 ONE      STY     IOCB0+4         ;POINT BUFFER INSIDE PAGE
06AD 8E4803 2020          STX     IOCB0+8         ;SET BUFFER LENGTH
06B0 A200   2030          LDX     #$00            ;MAKE SURE BUFFER LENGTH <255
06B2 8E4903 2040          STX     IOCB0+9
06B5 8E5903 2050          STX     IOCB1+9
06B8 F013   2060          BEQ     DOIT
06BA 48     2070 MENU     PHA
06BB A901   2080          LDA     #$01            ;POINT BUFFER TO $100
06BD 8D4503 2090          STA     IOCB0+5
06C0 8D5503 2100          STA     IOCB1+5
06C3 68     2110          PLA
06C4 D0E4   2120          BNE     ONE
06C6 A008   2130 EXIT     LDY     #$08
06C8 8C5403 2140 DITT     STY     IOCB1+4         ;SET BUFFER LENGTH TO 8
06CB A210   2150          LDX     #$10            ;THIS ACTIVATES IOCB1
06CD 9D4203 2160 DOIT     STA     IOCB,X          ;OTHERWISE USE IOCB0
06D0 2056E4 2170          JSR     CIOV            ;GENERAL CIO VECTOR
06D3 60     2180          RTS
06D4 A90B   2190 FETCH    LDA     #$0B            ;GET AN ANSWER
06D6 20A306 2200          JSR     TOOLKIT
06D9 60     2210          RTS
06DA A008   2220 FUNC     LDY     #$08            ;EXECUTE A FUNCTION
06DC A228   2230          LDX     #$28
06DE A905   2240          LDA     #$05
06E0 20BA06 2250          JSR     MENU
06E3 60     2260          RTS
            2270 ;
            2280 ;
            2290 ;DICTIONARY
            2300 ;
06E4 20     2310 INPUTS   .BYTE " LUKRF"
06E5 4C
06E6 55
06E7 4B
06E8 52
06E9 46
06EA 20     2320 TOOLS    .BYTE " #$! ",254
06EB 23
06EC 24
06ED 21
06EE 20
06EF FE
```

```
06F0 9B     2330         .BYTE 155,">"
06F1 3E
06F2 46     2340         .BYTE "FN?"
06F3 4E
06F4 3F
06F5 C6     2350         .BYTE 198
06F6 44     2360         .BYTE "D#?"
06F7 23
06F8 3F
            2370 ;
            2380 ;CHANGE DOSVEC TO $600
            2390 ;
06F9        2400         *=      $A
            2410 ;
000A 00     2420         .BYTE 0,6
000B 06
            2430 ;
            2440 ;CHANGE DOS COMMAND TO $600
            2450 ;
000C        2460         *=      $1546
            2470 ;
1546 00     2480         .BYTE 0
            2490 ;
1547        2500         *=      $154A
            2510 ;
154A 06     2520         .BYTE 6
            2530 ;
            2540 ;PUT D1*.* AT $100
            2550 ;
154B        2560         *=      $100
            2570 ;
0100 44     2580         .BYTE "D1:*.*",155,127
0101 31
0102 3A
0103 2A
0104 2E
0105 2A
0106 9B
0107 7F
            2590 ;
            2600 ;PUT MINIDOS COMMANDS AT $125
            2610 ;
0108        2620         *=      $0125
            2630 ;
0125 7D     2640         .BYTE 125
0126 9B     2650         .BYTE 155,"MINIDOS",155,155
0127 4D
0128 49
0129 4E
012A 49
012B 44
```

```
012C 4F
012D 53
012E 9B
012F 9B
0130 C1      2660        .BYTE "ADOS",155
0131 44
0132 4F
0133 53
0134 9B
0135 C2      2670        .BYTE "BASIC",155
0136 41
0137 53
0138 49
0139 43
013A 9B
013B C4      2680        .BYTE "DIRECTORY",155
013C 49
013D 52
013E 45
013F 43
0140 54
0141 4F
0142 52
0143 59
0144 9B
0145 C6      2690        .BYTE "FORMAT",155
0146 4F
0147 52
0148 4D
0149 41
014A 54
014B 9B
014C CB      2700        .BYTE "KILL",155
014D 49
014E 4C
014F 4C
0150 9B
0151 CC      2710        .BYTE "LOCK",155
0152 4F
0153 43
0154 4B
0155 9B
0156 D2      2720        .BYTE "RENAME",155
0157 45
0158 4E
0159 41
015A 4D
015B 45
015C 9B
015D D5      2730        .BYTE "UNLOCK",155
015E 4E
```

```
Ø15F 4C
Ø16Ø 4F
Ø161 43
Ø162 4B
Ø563 9B
Ø164        274Ø        .END
```

The Faster and Better Disks

The *Atari BASIC Faster & Better* program disks contain the major BASIC subroutines, machine language subroutines (USR routines), assembly language source code, demonstration programs and application (utility) programs presented in this book. In addition to saving you hours of work, typing and correcting the programs, these disks give you a convenient program library that you can call on whenever you want.

The three library packages are supplied on standard Atari 810 DOS 2.0 compatible diskettes. The three packages are:

ABFABLIB — (2 diskettes) which contains 65 of the major subroutines in this book.

ABFABASM — (1 diskette) which contains the source code and binary load files for the 10 machine language routines in this book.

ABFABDEM — (1 diskette) which contains the 12 application programs and the 14 demonstration programs in this book.

In general, each file name has a descriptive extention that will tell you what disk it would be on. The following table defines all of the extenders.

FIGURE 16.1 — *File Naming Conventions*

EXTENDER	TYPE OF FILE	DISK NAME
LST	BASIC subroutine	ABFABLIB
ASM	65Ø2 source code	ABFABASM
OBJ	Binary load file	ABFABASM
BAS	BASIC utility	ABFABDEM
DEM	Demonstration	ABFABDEM

The LST extender is used on most of the routines in the BASIC library. In a few cases I used another special extender. In any case, these routines are stored on the disks in LISTed format (ATASCII) and can be merged with any of your own application programs. These subroutines all have non-overlapping line numbers, so you could conceivably ENTER all of them into memory at the same time if RAM size permitted.

The ASM files contain the Assembler/editor source code for all of the 6502 routines in this book. Each of these files is in the format used by Atari's Assembler/editor cartridge. You can modify these source listings to perform in any manner you need. In addition, these listings will serve as excellent learning tools for the programmer who is new to 6502 assembly language.

The OBJ extender is used for the object code files that result from assembling the ASM files. In general, these OBJ files will be in DOS 2.0 binary load format. These files can be loaded with the <L> option from the DOS menu. NOTE: There are no DUP.SYS files on any of these disks.

The BAS extender is used on all of the application programs in this book. All of these programs are executable from BASIC even though some of them contain machine language subroutines. Each of these programs is a stand-alone utility. All of these programs are stored on the disk in tokenized format and can be called up with either the LOAD or RUN commands.

The DEM extender is used on those programs whose primary purpose is to illustrate some principle discussed in the text. In some cases, these programs can be modified to serve a number of other special purpose functions. All of the demonstration programs are executable from BASIC. Once again, some of the programs will contain machine language subroutines. All of these programs are stored on the disk in tokenized format also.

The Subroutine Library Disks (ABFABLIB)

The two disks in this library package contain all of the subroutines discussed in this book. Gathered together like this, the subroutines create a massive library of useful routines that can be almost instantly called up for use in your programs.

DISK #1 The First Half

FIGURE 16.2 shows you a file listing from the first subroutine library disk. The following paragraphs give you a brief synopsis of what each of the subroutines could be used for.

Figure 16.2 — *Subroutines From ABFABLIB Disk #1*

```
PHONE   LST
SFILL   LST
VLIST   LST
VSHORT  LST
RESERVE LST
MOVER   LST
SCRAMBLELST
REMAIN  LST
ROUNDDECLST
ROUNDDWNLST
ROW     LST
COLUMN  LST
ROUNDUP LST
MONEY   LST
```

```
DECHEX  LST
HEXDEC  LST
ROUNDINTLST
STRIPPERLST
RIGHT   LST
LEFT    LST
CENTER  LST
REVERSE LST
VERIFY  LST
PEELOFF LST
LOWTOCAPLST
INVERT  LST
LOOKUP1DLST
LOOKUP2DLST
LOOKUPXYLST
SEEKER  LST
VALIDATELST
IIXTOIIILST
IIITOIIXLST
FINDAY  LST
COMPDAY LST
WEEKDAY LST
YEARCOM LST
MONTHCOMLST
DAYCOM1 LST
DAYCOM2 LST
FISCAL  LST
HMSTOSECLST
SECTOHMSLST
CLOKMATHLST
BITMAP  LST
BOOLEAN LST
SORT    LST
```

PHONE.LST — This special formatting routine will take an area code, prefix, and phone number and force them into a (XXX) XXX-XXXX format for use in a computerized address book.

- For more details see page 75.

SFILL.LST — You can use this handy little subroutine to instantly fill the entire video screen with the character of your choice.

- For more details see page 41.

VLIST.LST — This UTILITY analyzes all of the variables in your programs. To use it, you will have to load your program and then ENTER this one to add it to the end of yours. When you are debugging a program and want to display all of the variable names you have used along with what kind of variable each one is, you can temporarily merge VLIST.LST to the end of your program. VLIST.LST will also tell you the current value of all scalar variables as well as the DIMensioned and current lengths of all of your strings. You can call VLIST.LST at any time by pressing <BREAK> and then "GOSUB 19940".

- For more details see page 53.

VSHORT.LST — This subroutine is a condensed version of VLIST.LST for those occasions where you are working on a very large program that doesn't leave enough room for VLIST.LST. VSHORT.LST doesn't have all of the frills its big brother has. All it gives you is a list of your variables and what type of variable they are.

- For more details see page 57.

RESERVE.LST — You can use this UTILITY the next time you want to reserve a block of memory to hold a machine language subroutine or a custom character set. This routine will move LOMEM up and set aside a chunk of memory that is protected from everything except POKEs or power failures. You can choose how much memory to reserve.

- For more details see page 53.

MOVER.LST — This UTILITY uses a fast machine language subroutine to move a block of memory to a new location. You can use this routine for moving screen data or filling the section of memory you protected with RESERVE.LST.

- For more details see page 58.

SCRAMBLE.LST — I don't really approve of what this routine does, but the next time you want to make one of your BASIC programs unlistable, you can do it with this scramble routine.

- For more details see page 57.

REMAIN.LST — Routine for finding the remainder of a divide operation.

- For more details see page 71.

ROUNDDEC.LST — Routine for rounding a number to the nearest chosen decimal.

- For more details see page 72

ROUNDDWN.LST — Routine for rounding a number to the next smaller integer.

- For more details see page 73.

ROW.LST — Routine for finding a PLOT row on the screen.

- For more details see page 73.

COLUMN.LST — Routine for finding a PLOT column on the screen.

- For more details see page 73.

ROUNDUP.LST — Routine for rounding a number to the next larger integer.

- For more details see page 73.

MONEY.LST — Special formatting routine for dollars and cents.

- For more details see page 74.

DECHEX.LST — I have never heard of a faster UTILITY in BASIC for converting decimal numbers into hexadecimal. The only way to do these conversions faster would be to use machine language.

- For more details see page 76.

HEXDEC.LST — This UTILITY is the mirror of the previous routine and comes just as highly recommended. If you find a faster or better BASIC routine for converting hexadecimal numbers to decimal numbers, please let me know.

- For more details see page 76.

ROUNDINT.LST — Routine for rounding positive & negative numbers.

- For more details see page 72.

STRIPPER.LST — This routine and the next three routines are excellent tools for formatting strings for output to the screen or a printer. This routine strips all of the "trailing" blanks from the end of a string.

- For more details see page 85.

RIGHT.LST — This routine right-justifies a string inside a field defined by you. This is an easy way to get columns on a printer to line up on the right hand side.

- For more details see page 86.

LEFT.LST — This routine left-justifies a string inside a field defined by you. It is also quite useful for formatted printer outputs.

- For more details see page 87.

CENTER.LST — Sometimes you need to have a string centered in a special field. This routine will do the trick for you.

- For more details see page 87.

REVERSE.LST — This routine takes a "last-name,first-name" string and converts it to a "first-name,last-name" string.

- For more details see page 88.

VERIFY.LST — This is a BASIC subroutine for finding a substring inside a long string.

- For more details see page 92.

PEELOFF.LST — Routine for "peeling" a command from a list of commands. You have to read the write-up on this one to fully appreciate it.

- For more details see page 89.

LOWTOCAP.LST — This subroutine changes lower case letters in a string to upper case ones.

- For more details see page 90.

INVERT.LST — This routine comes in especially handy when working in GRAPHICS 1 or 2. INVERT.LST will selectively toggle normal characters to inverse ones. It will also flip them the other way. You can set it to work on just one type of character or all characters.

- For more details see page 91.

LOOKUP1D.LST — This BASIC subroutine will search a string for a particular substring and tell you the index of the first character of the substring. See SEEKER.LST for a machine language variation of this routine.

- For more details see page 95.

LOOKUP2D.LST — Although Atari BASIC doesn't support true string arrays, you can use this subroutine to simulate a two-dimensional string array.

- For more details see page 96.

LOOKUPXY.LST — You can use this subroutine to find the two-dimensional (X,Y) address of a substring in a simulated string array.

- For more details see page 97.

SEEKER.LST — This BASIC routine uses a machine language subroutine to rapidly search a one dimensional string for a particular substring. This routine returns the index of the first character in your substring.

- For more details see page 95.

VALIDATE.LST — This is the first of a number of calendar oriented utilities. This routine checks a given date to see if it is a valid date.

- For more details see page 98.

IIXTOIII.LST — Eight byte dates are nice for printed displays, but they take up a lot more memory than three byte dates. This routine compresses an 8-byte date down to a 3-byte date.

- For more details see page 99.

IIITOIIX.LST — You can use this routine to "un-compress" an 3-byte date and get back the longer 8-byte version.

- For more details see page 100.

FINDAY.LST — This routine is useful for computing the day of the year.

- For more details see page 100.

COMPDAY.LST — This routine calculates a special "compday" value for any day of the year. To really use it to best effect you will need the following routines.

- For more details see page 101.

WEEKDAY.LST — Use this routine if you would like to find the day of the week for a given date.

- For more details see page 101.

YEARCOM.LST — Routine for finding a year given the compday.

- For more details see page 102.

MONTHCOM.LST — Routine for finding a month given the compday.

- For more details see page 102.

DAYCOM1.LST — Routine for finding a day of the year given the compday.

- For more details see page 102.

DAYCOM2.LST — Routine for finding a day of the month given the compday.

- For more details see page 102.

FISCAL.LST — Routine to convert calendar dates to fiscal dates.

- For more details see page 102.

HMSTOSEC.LST — This routine and the next two are useful for doing many clock computations. This routine takes a clock time in HH:MM:SS format and calculates the number of equivalent seconds.

- For more details see page 113.

SECTOHMS.LST — This routine is the reverse of the last one. It takes a number of seconds and computes the equivalent number of hours, minutes, and seconds.

- For more details see page 113.

CLOKMATH.LST — This routine is great for computing elapsed time.

- For more details see page 114.

BITMAP.LST — Atari BASIC does not support logical operations at the bit level. This routine will selectively SET, CLEAR, or TEST any bit within a byte.

- For more details see page 119.

BOOLEAN.LST — This UTILITY goes hand-in-hand with the last routine. This routine will perform a variety of logical operations on the bit level from BASIC.

- For more details see page 126.

SORT.LST — This UTILITY uses a fast machine language subroutine to perform an in-memory Shell sort from BASIC.

- For more details see page 142.

DISK #2 The Other Half

FIGURE 16.3 shows you a list of the subroutines on the second subroutine library disk. The following paragraphs summarize what each of the routines do.

Figure 16.3 — *Subroutines From ABFABLIB Disk ±2*

```
KEY     LST
MENU1   LST
MENU2   LST
FUNKEY  LST
MENU3   LST
BREAKLOKLST
REPEAT  LST
INKEY1  LST
INKEY2  LST
FIELDB  LST
FIELDI  LST
FDOLLARSLST
FDATES  LST
FTIMES  LST
FSCROLL LST
TITLE   LST
POLICAR LST
TRAIN   LST
TANK    LST
THUNDER LST
FLIES   LST
MOTRBOATLST
MANHOLE LST
SURF    LST
EUROCOP LST
STORM   LST
HEART   LST
TAKEOFF LST
SPLAT   LST
SAUCER1 LST
SAUCER2 LST
KLAXON  LST
BOMB    LST
EXPLODE LST
PAINTGETDSK
CITOH   GR8
GR8PUT  DSK
GR8GET  DSK
```

KEY.LST — This is a single key input routine that comes in useful for many applications. KEY.LST will get a single key input from the keyboard without requiring you to press the <RETURN> key.

- For more details see page 152.

MENU1.LST — This is a kernel for a keyboard driven menu routine. It is a ready-made menu that you can use in your own programs by making a few minor changes to specify the menu options you want.

- For more details see page 154.

MENU2.LST — This is another kernel menu that has been set up to accept inputs from a paddle controller. A few minor changes will customize it for your programs. Some other changes will convert the input device to a joystick.

- For more details see page 155.

FUNKEY.LST — This is a handy routine for testing to see if one of the keyboard function keys is pressed.

- For more details see page 156.

MENU3.LST — This is another kernal menu that you can easily customize for your own applications. This menu uses the FUNKEY.LST subroutine and is ideal for small limited option menus.

- For more details see page 157.

BREAKLOK.LST — Pressing the <BREAK> key during certain I/O operations can cause the computer to lose valuable data. This routine can be used to disable the <BREAK> key so you can avoid that kind of accident.

- For more details see page 158.

REPEAT.LST — This routine repeats a function as long as you press a key.

- For more details see page 158.

INKEY1.LST — This routine uses KEY.LST to give you controlled string input for strings of whatever size you specify.

- For more details see page 160.

INKEY2.LST — This routine controls multi-key numeric inputs in much the same way INKEY1.LST controls string inputs.

- For more details see page 160.

FIELDB.LST — This routine is useful for creating "blank" input fields on the screen. These come in handy for "fill in the blank" menus and forms.

- For more details see page 163.

FIELDI.LST — This routine creates inverse video input fields on the screen.

- For more details see page 163.

FDOLLARS.LST — Routine for controlled input of dollars and cents.

- For more details see page 164.

FDATES.LST — Routine for controlled input of caledar dates.

- For more details see page 164.

FTIMES.LST — Routine for controlled input of the time of the day.

- For more details see page 165.

FSCROLL.LST — This routine uses the text window at the bottom of the video screen in GRAPHICS 0 as a scrolling window for user inputs.

- For more details see page 165.

TITLE.LST — This UTILITY routine creates colorful title pages in GRAPHICS 2 for your programs.

- For more details see page 182.

The following routines are a collection of sound effects for your programs. Since there are so many of them, I will only give you a short note on what each one does.

- For more details see Chapter 14.

```
POLICAR.LST  - SOUND EFFECT of American police car siren
TRAIN.LST    - SOUND EFFECT of old timey steam train with a whistle
TANK.LST     - SOUND EFFECT of an army tank's engine
THUNDER.LST  - SOUND EFFECT of rolling peals of thunder
FLIES.LST    - SOUND EFFECT of a swarm of house flies
MOTRBOAT.LST - SOUND EFFECT of an outboard motor for a small boat
MANHOLE.LST  - SOUND EFFECT of a manhole cover slowly settling on a sidewalk
SURF.LST     - SOUND EFFECT of the gentle wash of ocean waves on a beach
EUROCOP.LST  - SOUND EFFECT of a European police car siren
STORM.LST    - SOUND EFFECT of a raging thunder storm
HEART.LST    - SOUND EFFECT of the slow beating of a human heart
TAKEOFF.LST  - SOUND EFFECT of a starship taking off
SPLAT.LST    - SOUND EFFECT of a long fall with a sudden stop
SAUCER1.LST  - SOUND EFFECT of a fying saucer hovering overhead
SAUCER2.LST  - SOUND EFFECT of a starship hovering overhead
KLAXON.LST   - SOUND EFFECT of an old timey klaxon siren
BOMB.LST     - SOUND EFFECT of a bomb dropping and exploding
EXPLODE.LST  - SOUND EFFECT of a sharp explosion
```

The following four routines are a couple of last minute additions to chapter 13. I think you will find them useful.

PAINTGET.DSK — UTILITY to load a Micro-Painter picture into your program.
- For more details see page 206.

CITOH.GR8 — UTILITY to dump a GRAPHICS 8 screen to a C-ITOH 8510 printer.
- For more details see page 204.

GR8PUT.DSK — UTILITY to save a GRAPHICS 8 screen to a disk file.
- For more details see page 201.

GR8GET.DSK — UTILITY to load a GRAPHICS 8 screen from a disk file
- For more details see page 203.

The Assembly Library Disk (ABFABASM)

This disk contains the 6502 source code and binary object code for the various machine language routines in this book. Most of them are designed to operate independent of any particular BASIC program, but a few of them are specially designed to be called from BASIC. Figure 16.4 shows you a directory listing from this disk.

FIGURE 16.4 — *Directory Listing From ABFABASM*

```
DOS     SYS Ø39
----------- ØØØ
Source Code ØØØ
----------- ØØØ
SFILL   ASM Ø12
MOVER   ASM Ø3Ø
SEEKER  ASM Ø32
CLOCK   ASM Ø5Ø
BITMAP  ASM Ø22
BOOLEAN ASM Ø17
SHELL   ASM Ø81
SLOWLISTASM Ø6Ø
MINIDOS ASM Ø45
BLINK   ASM Ø34
            ØØØ
----------- ØØØ
Object Code ØØØ
----------- ØØØ
SFILL   OBJ ØØ1
MOVER   OBJ ØØ2
SEEKER  OBJ ØØ3
CLOCK   OBJ ØØ3
BITMAP  OBJ ØØ1
BOOLEAN OBJ ØØ1
SHELL   OBJ ØØ4
SLOWLISTOBJ ØØ3
MINIDOS OBJ ØØ3
BLINK   OBJ ØØ2
----------- ØØØ
            ØØØ
*********** ØØØ
            ØØØ
Atari BFAB  ØØØ
Assembly    ØØØ
Language    ØØØ
Programs    ØØØ
            ØØØ
Copyright   ØØØ
1983 by IJG ØØØ
            ØØØ
*********** ØØØ
            ØØØ
263 FREE SECTORS
```

Here is a description of these programs:

SFILL.ASM — This is a machine language subroutine designed to be called from BASIC. This routine will fill the video screen with the character of your choice.

- For more details see page 34.

MOVER.ASM — This is another machine language subroutine that is designed to be called from BASIC. This routine will move a block of data from one place in memory to another. The routine handles both overlapping and non-overlapping moves either up or down in memory.

- For more details see page 58.

SEEKER.ASM — This is a BASIC callable machine language routine for searching a long string for the presence of a smaller string (substring).

- For more details see page 92.

CLOCK.ASM — This is a stand alone real time clock routine that can be accessed from BASIC. This clock is self calibrating and doesn't lose time during disk I/O.

- For more details see page 106.

BITMAP.ASM — This is a BASIC callable UTILITY that will SET, CLEAR or TEST a single bit in a byte.

- For more details see page 117.

BOOLEAN.ASM — This is a BASIC callable UTILITY that gives you bit level Boolean logic from BASIC.

- For more details see page 125.

SHELL.ASM — This is a BASIC callable UTILITY for doing a Shell sort in memory.

- For more details see page 135.

SLOWLIST.LST — This is a stand alone UTILITY for controlling the speed of a video listing. It works with either BASIC or the ASM/EDITOR cartridge.

- For more details see page 193.

MINIDOS.ASM — This is an excellent stand alone UTILITY that gives you simple DOS functions in BASIC or with the ASM/EDITOR cartridge.

- For more details see page 235.

BLINK.ASM — This is a stand alone UTILITY that gives you a blinking cursor in BASIC or ASM/EDITOR mode. You can control the blink rate. This routine also causes all inverse video characters to blink.

- For more details see page 169.

Each of these source files has a matching binary load file on this disk.

The Demonstration/Applications Library Disk (ABFABDEM)

The programs on this disk are all stand alone BASIC programs that either demonstrate some principle discussed in the text or serve some useful UTILITY function. Figure 16.5 shows you a directory listing of this disk.

Figure 16.5 — *Directory Listing From ABFABDEM*

```
DOS      SYS ØØ39
------------ ØØØ
Utilities    ØØØ
------------ ØØØ
CONVERT BAS Ø14
DATAPAK BAS Ø49
HEADER  BAS Ø38
CLOCK   BAS Ø16
DATECOMPBAS Ø26
SLOWLISTBAS Ø13
MARQUEE BAS Ø13
AUTOGO  BAS Ø21
CATALOG BAS Ø18
RPMTEST BAS Ø13
BLINK   BAS Ø1Ø
MINIDOS BAS Ø16
            ØØØ
----------- ØØØ
Demo Progs  ØØØ
----------- ØØØ
SFILL   DEM ØØ5
WINDOW  DEM ØØ4
MOVER   DEM ØØ6
SHELL   DEM Ø11
BUBBLE  DEM Ø1Ø
SHELL3  DEM Ø24
SHELL2  DEM Ø29
CONTROL DEM Ø19
SLYDESHODEM Ø49
SCROLL  DEM ØØ9
GLOW1   DEM ØØ3
GLOW2   DEM ØØ3
SOUND1  DEM ØØ5
SOUND2  DEM ØØ4
----------- ØØØ
            ØØØ
*********** ØØØ
            ØØØ
Atari BFAB  ØØØ
Application ØØØ
and Demo    ØØØ
Programs    ØØØ
            ØØØ
Copyright   ØØØ
1983 by IJG ØØØ
            ØØØ
*********** ØØØ
            ØØØ
 24Ø FREE SECTORS
```

The following paragraphs give you a brief description of each program.

Application Programs

CONVERT.BAS — This is a UTILITY that converts a machine language binary load file into BASIC DATA statements. The resulting DATA statements are automatically LISTed to either cassette or disk. This program actually creates an entire subroutine for POKEing a machine language routine into memory.

- For more details see page 39.

DATAPAK.BAS — This UTILITY is similar to CONVERT, but it is for the more advanced user who wants to string pack his machine language subroutines. This program string packs a machine language binary load file or the equivalent BASIC DATA statements (outputs from CONVERT for example) and LISTs the resulting string to cassette or disk.

- For more details see page 42.

HEADER.BAS — This UTILITY decodes the header on a binary load file and tells you how long the file is, where it normally loads into memory, and what the INIT/RUN addresses are.

- For more details see page 77.

CLOCK.BAS — This program is simply a service routine for setting up the real time clock.

- For more details see page 110.

DATECOMP.BAS — This program is a handy UTILITY that computes various calendar functions. Aside from being a perpetual calendar, it will also calculate "days between dates", "day of the week", "day within the year", and the date X days hence. In other words, this program integrates all of the major calendar routines into a single utility.

- For more details see page 103.

SLOWLIST.BAS — This program is primarily just a service UTILITY that loads the SLOWLIST machine routine into memory from BASIC.

- For more details see page 197.

MARQUEE.BAS — This program is a UTILITY that creates a custom scrolling banner display. It will prompt you to enter a message which will then be displayed in a scrolling fashion on the video screen.

- For more details see page 179.

AUTOGO.BAS — This program is a useful UTILITY that creates an AUTORUN.SYS file for automatically running BASIC programs. All you need to supply is a file name.

- For more details see page 225.

CATALOG.BAS — This program is a UTILITY that creates a custom listing of the files on all of your DOS files on disk. This is a handy routine for making a real printed catalog of all of your software.

- For more details see page 228.

RPMTEST.BAS — This UTILITY program helps you to keep your disk drives in top running condition. With this program you can easily monitor the speed of your disk drives. This RPM test program even handles non-standard drives like those with the HAPPY modification.

- For more details see page 231.

BLINK.BAS — This program is just a service utility for loading the blinking cursor machine language routine into memory from BASIC.

- For more details see page 172.

Demonstration Programs

These demonstration programs were included primarily to illustrate one or more of the things we discussed in the book, but these programs, in many cases, can be adapted for your custom application with only a moderate amount of work.

SFILL.DEM — This program shows you how to use the screen fill machine language subroutine. It prompts you for a character input and instantly fills the screen with that character.

- For more details see page 38.

WINDOW.DEM — This program demonstrates the MOVER block move machine language subroutine by using it to display any page of memory on the video screen.

- For more details see page 64.

MOVER.DEM — This is a much simpler demo of the MOVER subroutine. This program displays a string of characters at the top of the video screen and rapidly moves the string to the bottom of the display and back to the top of the screen. In fact, this routine is so fast that a special time delay had to be put in the program so you could see the movement.

- For more details see page 63.

SHELL.DEM — This program is a benchmark timing demo of a BASIC Shell sort. It assumes that the real time clock program has been loaded.

- For more details see page 133.

BUBBLE.DEM — This program is a benchmark timing demo that demonstrates just how slow a BASIC bubble sort really is. This program also makes use of the real time clock routine.

- For more details see page 132.

SHELL3.DEM — This program is a highly visual demo that sorts a file on the screen where you can watch the sorting process. By watching the patterns used by the Shell sort, you can get a better feel for exactly how it works. On top of that, the effect is hypnotizing.

- For more details see page 150.

SHELL2.DEM — This demo program is almost an application program. It will take a block of data and sort it with a fast machine language subroutine. When it is finished it will display the elapsed time for the sort as computed from the real time clock. To use this program for your own application, simply replace the dummy data it uses for the demo with your own data.

- For more details see page 147.

CONTROL.DEM — This is actually a menu from an applications program I wrote a while back. It demonstrates many of the concepts for menus, error trapping, and controlled inputs.

- For more details see page 173.

SLYDESHO.DEM — One of these days I will go back to this routine and add in all of the frills it is set up to handle. In the meantime, this program functions as a clear example of what you can achieve with the powerful technique called "page flipping". By setting this kind of routine up for GRAPHICS 8, you could do page flip animation. In this particular demo, I used normal GRAPHICS 0 and flipped from one page of text to another.

- For more details see page 185.

SCROLL.DEM — A demo of coarse scrolling in BASIC

- For more details see page 178.

GLOW1.DEM — This is a useful little routine that gives your GRAPHICS 1 or 2 messages a lovely glowing effect.

- For more details see page 182.

GLOW2.DEM — This glow routine creates a colorful display by randomly shifting the colors of every character on a GRAPHICS 1 or 2 display.

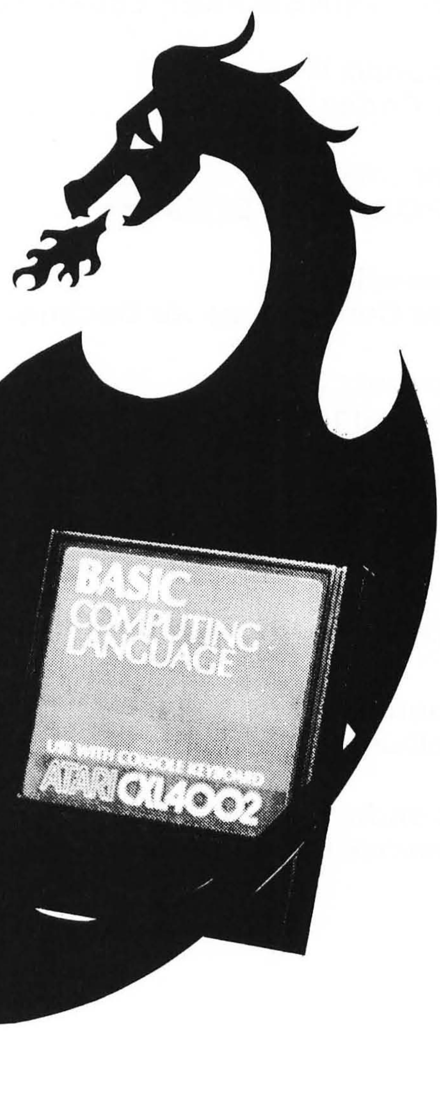
- For more details see page 182.

SOUND1.DEM — Demo of the effects of using 16 bit sound.

- For more details see page 214.

SOUND2.DEM — Demo of the effects of using interference effects in sound waves.

- For more details see page 216.



Appendix Table of Contents

Useful POKE & PEEK Locations

Address	Description
16	POKE 64 here (and at 53774) to disable the BREAK key. (192 to restore it)
18,19,20	Clock. Address 20 increases increments 60 times per second
65	POKE a zero here to stop normal program loading sounds.
77	POKE a zero here to turn off the screen "attract" mode.
82	Screen left margin (default=2)
83	Screen right margin (default=39)
84	Current cursor row (GRAPHICS 0)
85,86	Current cursor column for all modes (ranges from 0 to 319)
87	Graphics mode number for screen output
88,89	Upper left hand screen corner address
93	Code for the character that is under the cursor
106	Size of available memory in 256 byte pages
128,129	BASIC's LOMEM pointer
130,131	Contains location of the Variable Name Table
132,133	Points to the end of the Variable Name Table plus one byte
134,135	Contains location of the Variable Value Table
136,137	Points to the beginning of a BASIC program
138,139	BASIC's current statement pointer
140,141	Contains location of the String and Array Table, also the end of a BASIC program.

144,145	BASIC's top of memory pointer
186,187	The line number where a BASIC program stopped due to ERROR, TRAP, STOP or BREAK.
195	The OS code for an error during execution
212,213	Used to return a value from a USR call
559	Direct Memory Access (DMA) control: POKE zero here to turn the video display OFF; 34 restores the screen.
56Ø,561	Contains the location of the display list
58Ø	POKE 1 here to cause a reboot when SYSTEM RESET is pressed.
624	Contains current value of PADDLEØ (Ø-228)
625	PADDLE1
626	PADDLE2
627	PADDLE3
628	PADDLE4
629	PADDLE5
63Ø	PADDLE6
631	PADDLE7
632	Contains current value of STICKØ (15,7,6,14,1Ø,11,9,13,5)
633	STICK1
634	STICK2
635	STICK3
636	PTRIGØ: contains Ø if PADDLEØ trigger is pressed; otherwise contains 1.
637	PTRIG1
638	PTRIG2
639	PTRIG3
64Ø	PTRIG4
641	PTRIG5
642	PTRIG6
643	PTRIG7
644	STRIGØ: contains Ø if joy STICKØ trigger is pressed; otherwise contains 1.
645	STRIG1
646	STRIG2
647	STRIG3
66Ø,661	Contains location of upper left corner of text window

694	Inverse video flag: 0=normal, 128=inverse
702	Caps-lock flag: 0=lowercase, 64=uppercase 128=control characters
703	POKE 4 here to create a text window in GRAPHICS 0. (default is 24).
708	COLOR0: used for upper case in GRAPHICS 1 and 2 (default is 40)
709	COLOR1: used for lower case in GRAPHICS 1 and 2 (default is 202)
710	COLOR2: used for inverse upper case in GRAPHICS 1 and 2; used for background in GRAPHICS 0 (default is 148)
711	COLOR3: used for inverse lower case in GRAPHICS 1 and 2 (default is 70)
712	COLOR4: used for the background (border) in GRAPHICS 0 (default is 0)
736,737	Used by DOS to hold the RUN address of a binary load file
738,739	Immediate execution address used by DOS to hold the INIT address of a binary load file.
740	RAMSIZ: same as location 106
741,742	MEMTOP for BASIC and the OS (minus 1 to get highest free memory.)
743,744	MEMLO points to the bottom of user memory for BASIC programs
752	Cursor inhibit: 0=visible, 1=invisible
755	Character mode: 1=Blank, 2=Normal, 3=Inverse
756	Character base register: 226=lowercase 224=uppercase
764	Contains value of last key pressed (internal code)
767	Scroll start/stop flag: Toggled by pressing <CNTL>-1; 0=Scroll enabled, otherwise disabled
832-847	IOCB0: default device for the screen editor * POKE 838,166 & POKE 839,238 to send all screen outputs to the printer. POKE 838,163 & POKE 839,246 to return to normal. * POKE 842,13 to go into auto input mode. POKE 842,12 to return to normal.
848-863	IOCB1
864-879	IOCB2
880-895	IOCB3
896-911	IOCB4

912-927	IOCB5
928-943	IOCB6: used by GRAPHICS for screen channel
944-959	IOCB7: used by LPRINT, LOAD, SAVE and LIST
2147,2148	One of two locations used by DOS to store LOMEM
2152,2153	The other DOS pointer to LOMEM
5533	Used by DOS to check for presence of DUP.SYS: Zero means DUP.SYS is not there.
4Ø96Ø	USR here to COLD START the BASIC cartridge
41Ø37	USR here to WARM START the BASIC cartridge
53277	POKE a 4 here to put paddle and joystick triggers in latch mode. In latch mode, once a trigger is pressed it stays "pressed" until this location is POKEd with zero.
53279	The keyboard function keys alter this register when they are pressed.
5376Ø	AUDF1: controls the frequency of audio channel one
53761	AUDC1: controls volume and distortion of audio channel one
53762	AUDF2: channel two frequency control
53763	AUDC2: channel two volume and distortion control
53764	AUDF3: channel three frequency control
53765	AUDC3: channel three volume and distortion control
53766	AUDF4: channel four frequency control
53767	AUDC4: channel four volume and distortion control
53768	AUDCTL: master audio channel control byte
53774	IRQEN: interrupt control register. POKE 64 here to disable the BREAK key. (247 to restore it)
54Ø18	PACTL: POKE 52 here to turn the cassette motor ON. POKE 6Ø here to turn the motor back OFF.
58454	CIOV: (more commonly known as E456) is the entry vector to the central I/O utility in the OS.
5846Ø	SETVBV: vertical blank interrupt setup vector
58484	WARM START: Do a USR to here to cause the computer to WARM START
58487	COLD START: Do a USR to here to cause the entire system to reboot.

Key Codes

Table B-1 --- Standard Upper Case Keycodes

CHARACTER	ATASCII CODE	KEYBOARD CODE
A	65	63
B	66	21
C	67	18
D	68	58
E	69	42
F	7Ø	56
G	71	61
H	72	57
I	73	13
J	74	1
K	75	5
L	76	Ø
M	77	37
N	78	35
O	79	8
P	8Ø	1Ø
Q	81	47
R	82	4Ø
S	83	62
T	84	45
U	85	11
V	86	16
W	87	46
X	88	22
Y	89	43
Z	9Ø	23
1	49	31
2	5Ø	3Ø
3	51	26
4	52	24
5	53	29
6	54	27
7	55	51
8	56	53
9	57	48
Ø	48	5Ø
<	6Ø	54
>	62	55
-	45	14
=	61	15
+	43	6
*	42	7
;	59	2
,	44	32
.	46	34
/	47	38

Table B-2 --- Inverse Upper Case Keycodes

CHARACTER	ATASCII CODE	KEYBOARD CODE
A	193	63
B	194	21
C	195	18
D	196	58
E	197	42
F	198	58
G	199	61
H	200	57
I	201	13
J	202	1
K	203	5
L	204	0
M	205	37
N	206	35
O	207	8
P	208	10
Q	209	47
R	210	40
S	211	62
T	212	45
U	213	11
V	214	16
W	215	46
X	216	22
Y	217	43
Z	218	23
1	177	31
2	178	30
3	179	26
4	180	24
5	181	29
6	182	27
7	183	51
8	184	53
9	185	48
0	176	50
<	188	54
>	190	55
-	173	14
=	189	15
+	171	6
*	170	7
;	187	2
,	172	32
.	174	34
/	175	38

Table B-3 --- Standard Lower Case Keycodes

CHARACTER	ATASCII CODE	KEYBOARD CODE
a	97	63
b	98	21
c	99	18
d	100	58
e	101	42
f	102	56
g	103	61
h	104	57
i	105	13
j	106	1
k	107	5
l	108	0
m	109	37
n	110	35
o	111	8
p	112	10
q	113	47
r	114	40
s	115	62
t	116	45
u	117	11
v	118	16
w	119	46
x	120	22
y	121	43
z	122	23

Table B-4 --- Inverse Lower Case Keycodes

CHARACTER	ATASCII CODE	KEYBOARD CODE
a	225	63
b	226	21
c	227	18
d	228	58
e	229	42
f	230	56
g	231	61
h	232	57
i	233	13
j	234	1
k	235	5
l	236	0
m	237	37
n	238	35
o	239	8
p	240	10
q	241	47
r	242	40
s	243	62
t	244	45
u	245	11
v	246	16
w	247	46
x	248	22
y	249	43
z	250	23

Table B-5 --- Keycodes With Shift Key Pressed

CHARACTER	ATASCII CODE	KEYBOARD CODE
A	65	127
B	66	85
C	67	82
D	68	122
E	69	106
F	70	120
G	71	125
H	72	121
I	73	77
J	74	65
K	75	69
L	76	64
M	77	101
N	78	99
O	79	72
P	80	74
Q	81	111
R	82	104
S	83	126
T	83	109
U	85	75
V	86	80
W	87	110
X	88	86
Y	89	107
Z	90	87
!	33	95
"	34	94
#	35	90
$	36	88
%	37	93
&	38	91
'	39	115
@	64	117
(	40	112
)	41	114
[	91	96
]	93	98
?	63	102
:	58	66
\	92	70
^	94	71
|	124	79
_	95	78

Table B-6 --- Keycodes With Inverse Shift Key Pressed

CHARACTER	ATASCII CODE	KEYBOARD CODE
A	193	127
B	194	85
C	195	82
D	196	122
E	197	106
F	198	120
G	199	125
H	200	121
I	201	77
J	202	65
K	203	69
L	204	64
M	205	101
N	206	99
O	207	72
P	208	74
Q	209	111
R	210	104
S	211	126
T	212	109
U	213	75
V	214	80
W	215	110
X	216	86
Y	217	107
Z	218	87
!	161	95
"	162	94
#	163	90
$	164	88
%	165	93
&	166	91
'	167	115
@	192	117
(	168	112
)	169	114
[	219	96
]	221	98
?	191	102
:	186	66
\	220	70
^	222	71
|	252	79
-	223	78

Table B-7 --- Standard Upper Case Keycodes With <CTRL> Key Pressed

CHARACTER	ATASCII CODE	KEYBOARD CODE
<CTRL> - A	1	191
<CTRL> - B	2	149
<CTRL> - C	3	146
<CTRL> - D	4	186
<CTRL> - E	5	170
<CTRL> - F	6	184
<CTRL> - G	7	189
<CTRL> - H	8	185
<CTRL> - I	9	141
<CTRL> - J	10	129
<CTRL> - K	11	133
<CTRL> - L	12	128
<CTRL> - M	13	165
<CTRL> - N	14	163
<CTRL> - Ø	15	136
<CTRL> - P	16	138
<CTRL> - Q	17	175
<CTRL> - R	18	168
<CTRL> - S	19	190
<CTRL> - T	20	173
<CTRL> - U	21	139
<CTRL> - V	22	144
<CTRL> - W	23	174
<CTRL> - X	24	150
<CTRL> - Y	25	171
<CTRL> - Z	26	151

Table B-8 --- Keycodes for Miscellaneous Special Keys

CHARACTER	ATASCII CODE	KEYBOARD CODE
<CTRL> - CLEAR	125	182
<CTRL> - INSERT	225	183
<CTRL> - BACK S	254	180
<CTRL> - UP ARROW	28	142
<CTRL> - DOWN ARROW	29	143
<CTRL> - LEFT ARROW	30	134
<CTRL> - RIGHT ARROW	31	135
<SHFT> - CLEAR	125	118
<SHFT> - INSERT	157	119
<SHFT> - DELETE	156	116
SPACE BAR	32	33
ESCAPE KEY	27	28
RETURN KEY	155	12

ERROR Codes Explained

This is a complete list of the error codes you could encounter while you are running a BASIC program or performing operations in direct mode. Many of these error codes will be familiar to you. Others may not be quite as well known.

1 No error. This is the code that you get when whatever you were trying do was executed successfully without any error detectable by the operating system. This code is not normally displayed and is usually of interest only to machine language programmers.

2 Memory insufficient. This code means that your program is trying to use more memory than is available for it to use. This code usually pops up when you are dimensioning an array. You can also get this error when you are entering a new line of code for a BASIC program which causes the program to exceed the maximum size allowed.

3 Value error. This code means that a number used as a subscript is outside of the legal range for that particular function.

```
100 POKE -37,10            ——— wrong
110 PRINT PEEK(99999)

100 POKE 752,1             ——— correct
110 PRINT PEEK(764)
```

4 Too many variables. This one is simple. Atari BASIC allows a program to have a maximum of 128 variables referenced. If you are lucky, you can solve this problem by getting the garbage out of the variable name table. If you have already cleaned up the VNT, you will have to eliminate some variables from the actual program.

5 String length error. This is probably one of the most common errors you will run into. This code means that you have either used 0 as the index for a string or you have tried to store data in an index location that is larger than the dimensioned length of the string.

```
100 DIM TEMP$(20)                    ——— wrong
110 FOR X=0 TO 30:TEMP$(X)=X+99
120 NEXT X

100 DIM TEMP$(20)                    ——— correct
110 FOR X=1 TO 20:TEMP$(X)=X+99
120 NEXT X
```

6 Out of data error. This code means that your program tried to read data that wasn't there.

```
100 FOR X=1 TO 8:READ NUM:NEXT X     ——— wrong
110 DATA 1,2,3,4,5,6,7

100 FOR X=1 TO 8:READ NUM:NEXT X     ——— wrong
110 DATA 1,2,3,4,
120 DATA 5,6,7,8

100 FOR X=1 TO 8:READ NUM:NEXT X     ——— correct
110 DATA 1,2,3,4,5,6,7,8

100 FOR X=1 TO 8:READ NUM:NEXT X     ——— correct
110 DATA 1,2,3,4
120 DATA 5,6,7,8
```

7 Line number greater than 32767. If you get this error code, your program is trying to transfer control to a line number larger than the maximum allowable number.

```
100 GOTO 38000                ——— wrong
110 GOSUB 99999
120 RESTORE 767879
130 GOTO 40000

100 GOTO 3800                 ——— correct
110 GOSUB 9999
120 RESTORE 7678
130 GOTO 4000
```

8 Input error statement. This code means that you tried to enter a string value for an INPUT statement that was asking for a numeric input.

```
100 PRINT "ENTER THE NUMBER "
110 INPUT NUMBER

RESPONSE= SEVEN                 ——— wrong

RESPONSE= 7                     ——— correct
```

9 Array or string DIM error. There are several causes for this error code. First, you might have tried to DIM an array that had already been dimensioned. Second, you might have tried to dimension an array beyond the maximum allowable size. Numeric arrays cannot exceed 5460 and string arrays cannot exceed 32767 in length. Third, your program may be trying to use an undimensioned numeric array or string.

```
100 DIM TEMP$(37),TEMP$(45)              ——— wrong

100 DIM TEMP$(59634),PHONES(6000)        ——— wrong

100 DIM TEMP$(40)                        ——— wrong
110 TEMP4(35)=65

100 DIM TEMP$(37),TEMP$2(45)             ——— correct

100 DIM TEMP$(5963),PHONES(600)          ——— correct

100 DIM TEMP$(40)                        ——— correct
110 TEMP$(35)=CHR$(65)
```

10 Expression too complex. This is a rare error that is caused by an expression that overflows the argument stack. According to the technical manuals, you can get this error by using an equation that has too many levels of parentheses, but I have tried to create such a statement without any success. Therefore, I can't show you an example of how it might work.

11 Numeric overflow. This error code usually means that you tried to divide something by zero. This error could also happen if the result of an arithmatic operation gives a result that is greater than 9.99999999*10EXP(97).

```
100 X=37:Y=0                  ——— wrong
110 Z=X/Y

100 X=37:IF Y=0 THEN Y=1      ——— correct
110 Z=X/Y
```

12 Line not found. You will get this error code anytime your program tries to transfer control to a non-existent line number with a GOTO, GOSUB, IF/THEN, ON/GOSUB or ON/GOTO. Note that an out of range line number will give an error code 7. The TRAP statement simply clears all traps when you give it a non-existent line number.

```
100 GOTO 1200                 ——— wrong
110 X=Y
120 GOSUB 1400
130 Z=X+Y
140 PRINT Z

100 GOTO 120                  ——— correct
110 X=Y
120 GOSUB 140
130 Z=X+Y
140 PRINT Z
```

13 No matching FOR statement. This error code means that a NEXT statement was encountered that did not have a matching FOR statement. Note that a POP command essentially erases the last existing FOR statement from the stack.

```
100 FOR X=1 TO 10:Z=Z+1          ——— wrong
110 NEXT Z

100 FOR X=1 TO 10:POP            ——— wrong
110 NEXT X

100 FOR X=1 TO 10:Z=Z+1          ——— correct
110 NEXT X

100 FOR X=1 TO 10                ——— correct
110 IF Z(X)=1 THEN POP :GOTO130
120 NEXT X
130 END
```

14 Line too long. The only way I have ever managed to get this error message is to try to ENTER a tokenized file from disk instead of LOADing it.

15 GOSUB or FOR line deleted. You will really have to go out of your way to get this error code. This code means that you halted a program while it was executing a loop or a subroutine, and then you CONTinued the program after deleting the FOR or GOSUB statement that had started that loop or subroutine. That seems like a lot of work. Here is an example of how you can create this error code for demonstration purposes.

Type in the following program and run it —

```
100 FOR X=1 TO 20000
110 Z=COS(0.1)
120 NEXT X
```

Now that you have it running, press the <BREAK> key and type in the line number 100 followed by a <RETURN>. This effectively erases line number 100 from the program. Now type in the command CONT to resume running the program. You will almost immediately get an error 15 message.

16 RETURN error. You will get this error code whenever you try to execute a RETURN statement that doesn't have a matching GOSUB.

```
100 RETURN              ——— wrong

100 GOSUB 130           ——— correct
110 Z=Y
120 END
130 Z=37:RETURN
```

17 Syntax error. I can understand how this error could be created, but you would have to go out of your way to get it since BASIC performs an automatic syntax check everytime you enter a new line of code. However, this error could be created if a line of BASIC code is garbled by a machine language subroutine or a POKE to the BASIC program area in memory. The most unlikely cause of this error would be a faulty RAM cell.

18 VAL function error. You will get this error code if you try to use the VAL function on a string whose first character is not a number.

```
100 PRINT VAL("XYZ")          ——— wrong

100 PRINT VAL("7YZ")          ——— correct
```

19 LOAD program too long. This is an uncommon error, but it does crop up every once in a while. Essentially, this error code means that you tried to LOAD a program that exceeds the amount of memory you have. I have only encountered this error once. I once tried to LOAD a BASIC program from cassette while I had DOS in the computer so I could write the file out to disk. It turned out that the program would not load because it was so long that it needed the little bit of memory used by DOS.

20 Device number error. This is a simple error. If you get this error code, then you tried to use a device number that was less than 1 or greater than 7.

```
100 OPEN #9,4,0,"C:"          ——— wrong

100 OPEN #3,4,0,"C:"          ——— correct
```

21 LOAD file error. You will get this error code if you try to LOAD a file from disk (or cassette) that is not a normal tokenized file. "Bad" files like that are usually stored in binary load format or have been LISTed instead of SAVEd to disk.

22-127 These error codes have not been assigned to anything. If you get one of these, you are probably in deep yogurt.

128 BREAK abort. This "error" code is generated anytime you interrupt an I/O operation by pressing the <BREAK> key. Although this code is called an error code, it is technically a status code. You can eliminate this error in your programs by locking out the <BREAK> key.

129 IOCB already open. You will get this code when you try to OPEN a device that is already open. The solution to this error is to either CLOSE the needed device number before trying to OPEN it or to use a different device number.

```
100 OPEN #3,4,0,"D:PROG.ASM"          ——— wrong
110 OPEN #3,4,0,"D:PROG.OBJ"

100 OPEN #3,4,0,"D:PROG.ASM"          ——— correct
110 OPEN #4,4,0,"D:PROG.OBJ"
```

130 Nonexistent device. This code means that you tried to access a device that the operating system doesn't recognize. The most common cause of this error is trying to access a disk file without specifying the device.

```
100 RUN"PROG.OBJ"              ——— wrong

100 RUN"D:PROG.OBJ"            ——— correct
```

Another common cause is trying to access the RS232 ports without loading the RS232 handler first. The solution to that problem is to load the AUTORUN.SYS file that is on your DOS 2.0 master disk. That file is Atari's RS232 handler.

A less likely cause is that you tried to access a custom device that hasn't been defined in the OS handler table yet. Only machine language programmers are likely to run into this kind of problem, but you can get this error from BASIC by specifying an illegal device in an I/O request.

```
100 OPEN #3,4,0,"Q:"           ——— wrong

100 OPEN #3,4,0,"C:"           ——— correct
```

131 IOCB write only. This error occurs when you try to read from a device that was opened in "write only" mode. If you need to read from such a device, and input from that device is legal, then CLOSE the device and OPEN it either for "read only" or for "update" (read and write enabled).

```
100 OPEN #3,8,0,"D:PROG.DAT"     ——— wrong
110 GET #3,NUM

100 OPEN #3,4,0,"D:PROG.DAT"     ——— correct (read only)
110 GET #3,NUM

100 OPEN #3,12,0,"D:PROG.DAT"    ——— correct (update mode)
110 GET #3,NUM
```

132 Illegal handler command. The only time you should see this error code in BASIC is when you have given an XIO command a command number (cmdno) less than three. Go back to your BASIC manual and check your syntax. Machine language programmers will run across this code when they pass the wrong command code to the device handler.

```
100 XIO 1,#3,0,0,"C:"          ——— wrong

100 XIO 3,#3,0,0,"C:"          ——— correct
```

133 Device/file not open. You will get this error code if you try to access a device or a disk file that has not been opened.

```
100 GET #3,NUM                   ——— wrong

100 OPEN #3,4,0,"D:PROG.DAT"     ——— correct
110 GET #3,NUM
```

134 Bad IOCB number. The operating system only supports IOCB numbers between 0 and 7. Any attempt to access a device with a number outside this range will result in this error code. Note that IOCB 0 is not directly accessible from BASIC. I have never gotten this error code from BASIC. If you try to access an illegal device number in BASIC, you usually get an error code 20. I have managed to get the 134 error code while using the assembler/editor, but that is not the topic of this book.

135 IOCB read only. The causes of this error code are similar to those for code 131, but in this case you have attempted to write something to a device that was opened in the "read only" mode. The weird thing is I can't get this error to occur.

136 End-of-file. The operating system has encountered an end-of-file (EOF) marker in whatever file you are working with and consequently will not believe you when you tell it to get more data from that file. About all you can do if you get this error is to re-examine your control program and make sure that it is trying to get the correct number of records or bytes.

137 Truncated record. This error code is a nasty one. It usualy means that the disk file you are trying to access is mortally wounded. If you are lucky, you may recover the data by tuning the speed of your disk drive so it is running within the normal speed range. It is also possible that you are trying to use record-oriented input commands on a file that was created by a byte-oriented PUT command.

138 Device timeout. This is one of the most common error codes that you will see. Essentially, what it means is that the operating system went looking for a device you asked for and couldn't find it. For example, try doing an LPRINT with your printer turned OFF. There are many possible causes of this error code and there is a solution for each one. In general, however, you should check all of your cables and make sure that they are properly connected. You should also make sure that the requested device is turned ON.

139 Device NAK. The causes of this error code will depend upon what device was being accessed when the error occurred. Basically, this error code means that the computer did not receive the proper response from the given device when an I/O command was executed. Specifically, you might have asked the disk drive to read an illegal sector number. Another example is that you might be trying to send data through a serial channel at a faster rate than it can handle. I suggest that you simply check all of your cable connections and try the command over again.

140 Serial frame error. You will get this error code when communications between the computer and a device get garbled. This is a very rare error, and it is always fatal. Go back to the beginning and start over again with whatever you were trying to do. If the error persists, you may have a hardware problem.

141 Cursor out of range. The cursor position you requested would place the cursor somewhere off of the video display. The actual X,Y position limits are defined by the graphics mode you are in at the time. Check the logic of your cursor movement code and install limits to keep the cursor on the screen.

```
100 GRAPHICS 0:POSITION 10,75          ——— wrong

100 GRAPHICS 0:POSITION 10,10          ——— correct
```

142 Serial bus overrun. This is another rare error code. It usually means that POKEY (the computer) received a second eight-bit word over the serial data bus before the computer could finish processing the previous word. This error is also fatal. All you can do is try the operation again. If the problem persists, you should get your computer serviced.

143 Checksum error. This is a common error code for cassette users. It means that the information coming into the computer is garbage. This could be due to a bad recording on a tape or simply improper cueing of the tape before you tried to load it. Try the loading process again. The problem is probably more serious if you got this error code during disk I/O. You probably have a bad disk or disk file. Try to load the file again, and if the error persists, resort to the standard data recovery techniques.

144 Device done error. The device is unable to execute the command you gave it. This usually means that you tried to save something on a write-protected disk. If this is the case, remove the write-protect tab and try the operation again. The other major cause of this error is trying to write to a damaged sector on a disk.

145 Illegal screen mode. You will get this error code if you try to go into a non-existent graphics mode. Check your GRAPHICS command or your IOCB parameters. You can also get this error message if you are doing a write with verify, and the verify detects an error. This means that the computer wrote one thing to the disk and read something else back when it tried to verify the write. This could be caused by a faulty drive or by a bad disk. Hope it is the disk.

```
100 GRAPHICS 37          ——— wrong

100 GRAPHICS 7           ——— correct
```

146 Illegal function. This error code means that you tried to execute an I/O operation that is illegal for the specified device. For example, you might have tried to write to the keyboard or to read from a printer. Check your I/O command for the correct device and command.

```
100 OPEN #3,4,0,"K:"     ——— wrong
110 PUT #3,NUM

100 OPEN #3,4,0,"K:"     ——— correct
110 GET #3,NUM
```

147 Insufficient RAM. The only time you should ever get this error code is when you are trying to execute a GRAPHICS command which asks for more memory than you have available in your computer. The most common situation would be where you have a large BASIC program that leaves less than 8138 bytes available and you try to execute a GRAPHICS 8 command.

148-159 These error codes are not currently assigned to anything. You should never see any of them.

160 Drive number error. DOS is preset to support only two disk drives. If you add a third drive without altering the DOS, you will get this error code anytime you try to acces the third drive. The solution to this is easy. In BASIC you can POKE 1802,3 for three drives or POKE 1802,4 for four drives. Once you have altered the DOS, have DOS write itself out to disk using the <H> command from the DOS menu.

161 Too many files. Normally, a maximum of three disk files can be open at the same time. If you exceed that number, you will get this error code. If you really need to have more than three files open at the same time, POKE 1801,NUM, where NUM is the number of files you want to have open at the same time. If you will be needing this capability often, use the DOS <H> option to save the modified DOS to disk.

162 Disk full. You only have 707 sectors on a DOS formatted disk. You will get this error code if you reach this limit in the middle of saving something to disk. You should try another disk or delete some files to make room on the disk.

163 Unknown fatal error. This is a catch-all error code that pops up if DOS runs across an error that it can't identify. Try using a different DOS disk.

164 Bad disk file. This error comes up every now and then if you try transfering files from one DOS to another. It means that the DOS you are using can't follow the sector links contained in the file. Go back to the original DOS. If you still can't load the file, it is damaged, and you either to have to go to a backup copy or try the standard file recovery techniques. File recovery is not a task to be attempted by any but advanced level programmers. You can also get this error code if you use a POINT command that moves the file pointer outside of the specified disk file. When you do this, the computer thinks you have a bad file.

165 File name error. The syntax of a legal file name is rather strict. If you use an illegal character or use a chracter improperly in a file name, you will get this error code. Usually, this means that you used a file name that started with a lower case letter, contained an illegal character, or used one of the wild card values improperly.

```
100 OPEN #3,4,0,"D:cat,dat"        ——— wrong

100 OPEN #3,4,0,"D:CAT.DAT"        ——— correct
```

166 POINT data length error. This error code means that you used a POINT command improperly. The highest byte number in a normal 810 disk file is 125. If you try to point to a number larger than this, you will get an error 166.

```
1ØØ POINT #3,525,188          ——— wrong

1ØØ POINT #3,525,18           ——— correct
```

167 File locked. You will get this error code if you try to alter a disk file that has been "locked." Specifically, you can not write to, erase, or save on top of a locked file. If you need to alter a locked file, use the DOS <G> function to to unlock the file first. Note that you can still format a disk that contains locked files.

168 Unknown I/O command. This a catch-all error code for illegal I/O commands.

```
1ØØ OPEN #3,4,Ø,"D:CAT.DAT"  ——— wrong
11Ø PUT #3,NUM

1ØØ OPEN #3,8,Ø,"Ø:CAT.DAT   ——— correct
11Ø PUT #3,NUM
```

169 Directory full. DOS 2.0 is set up to allow a maximum of 64 file names in the disk directory. If you try to save a file after that point is reached, you will get this error message. It is possible to alter the DOS to allow more files to be on a disk, but you pay a penalty. The file names have to be shorter, and your modified DOS will not be compatible with the unmodified DOS. I suggest that you either delete a file to make room for your new file or use a new disk.

170 File not found. This error code means that you tried to access a file by a name that is not in the directory. There are several possible causes of this error. First, you may have misspelled the file name. Second, you may be searching the wrong disk. Third, you may have specified the wrong drive number. Fourth, the file may have been deleted and no longer exists. For the first three, simply correct the problem and the error should go away. The last one means you probably will have to recreate the file.

171 POINT invalid. This error code goes hand-in-hand with code 166. When you get a 166, you have tried to use an invalid byte number in a sector. Error code 171 means that you have tried to POINT to a sector that is not in a proper file. A proper file is one that has been opened for update (I/O code 12).

172 Illegal append. This is a special error code that you probably will never see. You will only get this code when you try to use DOS 2.0 to OPEN a DOS 1.0 file for append. If you get this error, simply copy the DOS 1.0 file over to a DOS 2.0 disk and repeat the OPEN command.

173 Format error. This error code means that the drive could not format the disk you are currently trying to format. There are two possible causes of this error. First, the disk may have a bad sector. Second, there may be a fault in the disk drive hardware. The usual hardware fault, in these cases, is out-of-tolerance drive speed. Check the speed of the drive and adjust it back into the allowable range. If the problem persists, take your drive to a repair center.

Base Conversions for Decimal, Binary and Hexadecimal Numbers

Decimal	Binary	Hex	Decimal	Binary	Hex	Decimal	Binary	Hex
0	00000000	0000	25	00011001	0019	50	00110010	0032
1	00000001	0001	26	00011010	001A	51	00110011	0033
2	00000010	0002	27	00011011	001B	52	00110100	0034
3	00000011	0003	28	00011100	001C	53	00110101	0035
4	00000100	0004	29	00011101	001D	54	00110110	0036
5	00000101	0005	30	00011110	001E	55	00110111	0037
6	00000110	0006	31	00011111	001F	56	00111000	0038
7	00000111	0007	32	00100000	0020	57	00111001	0039
8	00001000	0008	33	00100001	0021	58	00111010	003A
9	00001001	0009	34	00100010	0022	59	00111011	003B
10	00001010	000A	35	00100011	0023	60	00111100	003C
11	00001011	000B	36	00100100	0024	61	00111101	003D
12	00001100	000C	37	00100101	0025	62	00111110	003E
13	00001101	000D	38	00100110	0026	63	00111111	003F
14	00001110	000E	39	00100111	0027	64	01000000	0040
15	00001111	000F	40	00101000	0028	65	01000001	0041
16	00010000	0010	41	00101001	0029	66	01000010	0042
17	00010001	0011	42	00101010	002A	67	01000011	0043
18	00010010	0012	43	00101011	002B	68	01000100	0044
19	00010011	0013	44	00101100	002C	69	01000101	0045
20	00010100	0014	45	00101101	002D	70	01000110	0046
21	00010101	0015	46	00101110	002E	71	01000111	0047
22	00010110	0016	47	00101111	002F	72	01001000	0048
23	00010111	0017	48	00110000	0030	73	01001001	0049
24	00011000	0018	49	00110001	0031	74	01001010	004A

Decimal	Binary	Hex	Decimal	Binary	Hex	Decimal	Binary	Hex
75	01001011	004B	110	01101110	006E	145	10010001	0091
76	01001100	004C	111	01101111	006F	146	10010010	0092
77	01001101	004D	112	01110000	0070	147	10010011	0093
78	01001110	004E	113	01110001	0071	148	10010100	0094
79	01001111	004F	114	01110010	0072	149	10010101	0095
80	01010000	0050	115	01110011	0073	150	10010110	0096
81	01010001	0051	116	01110100	0074	151	10010111	0097
82	01010010	0052	117	01110101	0075	152	10011000	0098
83	01010011	0053	118	01110110	0076	153	10011001	0099
84	01010100	0054	119	01110111	0077	154	10011010	009A
85	01010101	0055	120	01111000	0078	155	10011011	009B
86	01010110	0056	121	01111001	0079	156	10011100	009C
87	01010111	0057	122	01111010	007A	157	10011101	009D
88	01011000	0058	123	01111011	007B	158	10011110	009E
89	01011001	0059	124	01111100	007C	159	10011111	009F
90	01011010	005A	125	01111101	007D	160	10100000	00A0
91	01011011	005B	126	01111110	007E	161	10100001	00A1
92	01011100	005C	127	01111111	007F	162	10100010	00A2
93	01011101	005D	128	10000000	0080	163	10100011	00A3
94	01011110	005E	129	10000001	0081	164	10100100	00A4
95	01011111	005F	130	10000010	0082	165	10100101	00A5
96	01100000	0060	131	10000011	0083	166	10100110	00A6
97	01100001	0061	132	10000100	0084	167	10100111	00A7
98	01100010	0062	133	10000101	0085	168	10101000	00A8
99	01100011	0063	134	10000110	0086	169	10101001	00A9
100	01100100	0064	135	10000111	0087	170	10101010	00AA
101	01100101	0065	136	10001000	0088	171	10101011	00AB
102	01100110	0066	137	10001001	0089	172	10101100	00AC
103	01100111	0067	138	10001010	008A	173	10101101	00AD
104	01101000	0068	139	10001011	008B	174	10101110	00AE
105	01101001	0069	140	10001100	008C	175	10101111	00AF
106	01101010	006A	141	10001101	008D	176	10110000	00B0
107	01101011	006B	142	10001110	008E	177	10110001	00B1
108	01101100	006C	143	10001111	008F	178	10110010	00B2
109	01101101	006D	144	10010000	0090	179	10110011	00B3

Decimal	Binary	Hex
180	10110100	00B4
181	10110101	00B5
182	10110110	00B6
183	10110111	00B7
184	10111000	00B8
185	10111001	00B9
186	10111010	00BA
187	10111011	00BB
188	10111100	00BC
189	10111101	00BD
190	10111110	00BE
191	10111111	00BF
192	11000000	00C0
193	11000001	00C1
194	11000010	00C2
195	11000011	00C3
196	11000100	00C4
197	11000101	00C5
198	11000110	00C6
199	11000111	00C7
200	11001000	00C8
201	11001001	00C9
202	11001010	00CA
203	11001011	00CB
204	11001100	00CC
205	11001101	00CD
206	11001110	00CE
207	11001111	00CF
208	11010000	00D0
209	11010001	00D1
210	11010010	00D2
211	11010011	00D3
212	11010100	00D4
213	11010101	00D5
214	11010110	00D6
215	11010111	00D7
216	11011000	00D8
217	11011001	00D9
218	11011010	00DA
219	11011011	00DB
220	11011100	00DC
221	11011101	00DD
222	11011110	00DE
223	11011111	00DF
224	11100000	00E0
225	11100001	00E1
226	11100010	00E2
227	11100011	00E3
228	11100100	00E4
229	11100101	00E5
230	11100110	00E6
231	11100111	00E7
232	11101000	00E8
233	11101001	00E9
234	11101010	00EA
235	11101011	00EB
236	11101100	00EC
237	11101101	00ED
238	11101110	00EE
239	11101111	00EF
240	11110000	00F0
241	11110001	00F1
242	11110010	00F2
243	11110011	00F3
244	11110100	00F4
245	11110101	00F5
246	11110110	00F6
247	11110111	00F7
248	11111000	00F8
249	11111001	00F9
250	11111010	00FA
251	11111011	00FB
252	11111100	00FC
253	11111101	00FD
254	11111110	00FE
255	11111111	00FF

Subroutines — by Line Number

*Note: * Programs contain machine language routines.*

CHAPTER	LINE #	FILE NAME	PURPOSE
* 3	19000	SFILL.LST	Screen fill using machine language
* 4	19900	MOVER.LST	Move a block of memory
4	19930	RESERVE.LST	Protect a section of memory
4	19940	VLIST.LST	BASIC variable analyzer
4	19990	VSHORT.LST	A short version of VLIST.LST
4	20000	SCRAMBLE.LST	Make your program unlistable
6	20010	REMAIN.LST	Find the remainder of a divide
6	20020	ROUNDINT.LST	Round to nearest integer
6	20030	ROUNDDEC.LST	Round to nearest given decimal
6	20040	ROUNDDWN.LST	Round down routine
6	20050	ROW.LST	Find row on screen
6	20060	COLUMN.LST	Find column on screen
6	20070	ROUNDUP.LST	Round up routine
6	20080	MONEY.LST	Formatted dollars and cents

CHAPTER	LINE #	FILE NAME	PURPOSE
6	20090	PHONE.LST	Formatted telephone numbers
6	20100	HEXDEC.LST	Hex-to-decimal converter
6	20110	DECHEX.LST	Decimal-to-hex converter
7	20120	STRIPPER.LST	Remove trailing blanks from string
7	20130	RIGHT.LST	Right justify a string
7	20140	LEFT.LST	Left justify a string
7	20150	CENTER.LST	Center a string
7	20160	REVERSE.LST	Last-name-first to last-name-last
7	20170	VERIFY.LST	Verify substring is in string
7	20180	PEELOFF.LST	Deconcatenate command string
7	20190	LOWTOCAP.LST	Convert lower case to upper case
7	20200	INVERT.LST	Convert normal to inverse
7	20210	LOOKUP1D.LST	Find substring in BASIC
7	20220	LOOKUP2D.LST	Two dimensional string search
7	20230	LOOKUPXY.LST	Find X and Y given element number
* 7	20240	SEEKER.LST	Find substring in machine language
8	20250	VALIDATE.LST	Is a date valid?
8	20260	IIXTOIII.LST	Convert 8-byte date to 3-byte date
8	20270	IIITOIIX.LST	Convert 3-byte date to 8-byte date
8	20280	FINDAY.LST	Find a day of the year
8	20290	COMPDAY.LST	Calculate computational date number
8	20300	WEEKDAY.LST	Find a day of the week
8	20310	YEARCOM.LST	Find year from COMPDAY
8	20320	MONTHCOM.LST	Find month from COMPDAY

CHAPTER	LINE #	FILE NAME	PURPOSE
8	2Ø33Ø	DAYCOM1.LST	Find day of year from COMPDAY
8	2Ø34Ø	DAYCOM2.LST	Find day of month from COMPDAY
8	2Ø35Ø	FISCAL.LST	Convert calendar to fiscal date
8	2Ø36Ø	HMSTOSEC.LST	Convert HH:MM:SS to seconds
8	2Ø37Ø	SECTOHMS.LST	Convert seconds to HH:MM:SS
8	2Ø38Ø	CLOKMATH.LST	Time clock subtraction
* 9	2Ø39Ø	BITMAP.LST	Set, clear, or test bits in a byte
* 9	2Ø41Ø	BOOLEAN.LST	Bit-level boolean operators
* 1Ø	2Ø43Ø	SORT.LST	Fast machine language shell sort
11	2Ø44Ø	KEY.LST	Single key input
11	2Ø45Ø	MENU1.LST	Keyboard menu
11	2Ø47Ø	MENU2.LST	Paddle driven menu
11	2Ø5ØØ	FUNKEY.LST	Function key tester
11	2Ø51Ø	MENU3.LST	Function key menu
11	2Ø53Ø	BREAKLOK.LST	Disable the BREAK key
11	2Ø54Ø	REPEAT.LST	Repeat as long as key is pressed
11	2Ø55Ø	INKEY1.LST	Controlled string input
11	2Ø56Ø	INKEY2.LST	Controlled numeric input
12	2Ø57Ø	FIELDB.LST	Create blank input field
12	2Ø58Ø	FIELDI.LST	Create inverse video input field
12	2Ø59Ø	FDOLLARS.LST	Create dollar input field
12	2Ø6ØØ	FDATES.LST	Create date input field
12	2Ø62Ø	FTIMES.LST	Create time input field
12	2Ø63Ø	FSCROLL.LST	Create a scrolling window display

CHAPTER	LINE #	FILE NAME	PURPOSE
=======	======	=========	=====================
13	2Ø64Ø	TITLE.LST	GRAPHICS 2 title page
14	2Ø65Ø	TRAIN.LST	Sound effect
14	2Ø67Ø	POLICAR.LST	Sound effect
14	2Ø68Ø	TANK.LST	Sound effect
14	2Ø69Ø	THUNDER.LST	Sound effect
14	2Ø7ØØ	FLIES.LST	Sound effect
14	2Ø71Ø	MOTRBOAT.LST	Sound effect
14	2Ø72Ø	MANHOLE.LST	Sound effect
14	2Ø73Ø	SURF.LST	Sound effect
14	2Ø74Ø	EUROCOP.LST	Sound effect
14	2Ø75Ø	STORM.LST	Sound effect
14	2Ø76Ø	HEART.LST	Sound effect
14	2Ø77Ø	TAKEOFF.LST	Sound effect
14	2Ø78Ø	SPLAT.LST	Sound effect
14	2Ø79Ø	SAUCER1.LST	Sound effect
14	2Ø8ØØ	SAUCER2.LST	Sound effect
14	2Ø81Ø	KLAXON.LST	Sound effect
14	2Ø82Ø	BOMB.LST	Sound effect
14	2Ø83Ø	EXPLODE.LST	Sound effect
13	2Ø9ØØ	PAINTGET.DSK	Get a Micro-Painter picture from disk
13	2Ø93Ø	GR8GET.DSK	Get a GRAPHICS 8 picture from disk
13	2Ø96Ø	GR8PUT.DSK	Put a GRAPHICS 8 picture on disk
13	21ØØØ	CITOH.GR8	Dump a GRAPHICS 8 picture to a printer

Subroutines — Alphabetically

*Note: * Programs contain machine language routines.*

LINE #	CHAPTER	FILE NAME	PURPOSE
======	=======	=========	=======
2Ø39Ø	* 9	BITMAP.LST	Set, clear, or test bits in a byte
2Ø82Ø	14	BOMB.LST	Sound effect
2Ø414	* 9	BOOLEAN.LST	Bit-level boolean operators
2Ø53Ø	11	BREAKLOK.LST	Disable the BREAK key
2Ø15Ø	7	CENTER.LST	Center a string
21ØØØ	13	CITOH.GR8	Dump a GRAPHICS 8 picture to a printer
2Ø38Ø	8	CLOKMATH.LST	Time clock subtraction
2ØØ6Ø	6	COLUMN.LST	Find column on screen
2Ø29Ø	8	COMPDAY.LST	Calculate computational date number
2Ø33Ø	8	DAYCOM1.LST	Find day of year from COMPDAY
2Ø34Ø	8	DAYCOM2.LST	Find day of month from COMPDAY
2Ø11Ø	6	DECHEX.LST	Decimal-to-hex converter
2Ø74Ø	14	EUROCOP.LST	Sound effect
2Ø83Ø	14	EXPLODE.LST	Sound effect

LINE #	CHAPTER	FILE NAME	PURPOSE
20600	12	FDATES.LST	Create date input field
20590	12	FDOLLARS.LST	Create dollar input field
20570	12	FIELDB.LST	Create blank input field
20580	12	FIELDI.LST	Create inverse video input field
20280	8	FINDAY.LST	Find a day of the year
20350	8	FISCAL.LST	Convert calenday to fiscal date
20700	14	FILES.LST	Sound effect
20630	12	FSCROLL.LST	Create a scrolling window display
20620	12	FTIMES.LST	Create time input field
20500	11	FUNKEY.LST	Function key tester
20930	13	GR8GET.DSK	Get a GRAPHICS 8 picture from disk
20960	13	GR8PUT.DSK	Put a GRAPHICS 8 picture on disk
20760	14	HEART.LST	Sound effect
20100	6	HEXDEC.LST	Hex-to-decimal converter
20360	8	HMSTOSEC.LST	Convert HH:MM:SS to seconds
20260	8	IIXTOIII.LST	Convert 8-byte date to 3-byte date
20270	8	IIITOIIX.LST	Convert 3-byte date to 8-byte date
20550	11	INKEY1.LST	Controlled string input
20560	8	INKEY2.LST	Controlled numeric input
20200	7	INVERT.LST	Convert normal to inverse
20440	11	KEY.LST	Single key input
20810	14	KLAXON.LST	Sound effect
20140	7	LEFT.LST	Left justify a string
20210	7	LOOKUP1D.LST	Find substring in BASIC

LINE #	CHAPTER	FILE NAME	PURPOSE
20220	7	LOOKUP2D.LST	Two dimensional string search
20230	7	LOOKUPXY.LST	Find X and Y given element number
20190	7	LOWTOCAP.LST	Convert lower case to upper case
20720	14	MANHOLE.LST	Sound effect
20450	11	MENU1.LST	Keyboard menu
20470	11	MENU2.LST	Paddle driven menu
20510	11	MENU3.LST	Function key menu
20080	6	MONEY.LST	Formatted dollars and cents
20320	8	MONTHCOM.LST	Find month from COMPDAY
20710	14	MOTRBOAT.LST	Sound effect
19900	* 4	MOVER.LST	Move a block of memory
20900	13	PAINTGET.DSK	Get a Micro-Painter picture from disk
20180	7	PEELOFF.LST	Deconcatenate command string
20090	6	PHONE.LST	Formatted telephone numbers
20670	14	POLICAR.LST	Sound effect
20010	6	REMAIN.LST	Find the remainder of a divide
20540	11	REPEAT.LST	Repeat as long as key is pressed
19930	4	RESERVE.LST	Protect a section of memory
20160	7	RESERVE.LST	Last-name-first to last-name-last
20130	7	RIGHT.LST	Right justify a string
20030	6	ROUNDDEC.LST	Round to nearest given decimal
20040	6	ROUNDDWN.LST	Round down routine
20020	6	ROUNDINT.LST	Round to nearest integer
20070	6	ROUNDUP.LST	Round up routine

LINE #	CHAPTER	FILE NAME	PURPOSE
20050	6	ROW.LST	Find row on screen
20790	14	SAUCER1.LST	Sound effect
20800	14	SAUCER2.LST	Sound effect
20000	4	SCRAMBLE.LST	Make your program unlistable
20370	8	SECTOHMS.LST	Convert seconds to HH:MM:SS
20240	* 7	SEEKER.LST	Find substring in machine language
19000	* 3	SFILL.LST	Screen fill using machine language
20430	* 10	SORT.LST	Fast machine language shell sort
20780	14	SPLAT.LST	Sound effect
20750	14	STORM.LST	Sound effect
20120	7	STRIPPER.LST	Remove trailing blanks from string
20730	14	SURF.LST	Sound effect
20770	14	TAKEOFF.LST	Sound effect
20680	14	TANK.LST	Sound effect
20690	14	THUNDER.LST	Sound effect
20640	13	TITLE.LST	GRAPHICS 2 title page
20650	14	TRAIN.LST	Sound effect
20250	8	VALIDATE.LST	Is a date valid?
20170	7	VERIFY.LST	Verify substring is in string
19940	4	VLIST.LST	BASIC variable analyzer
19990	4	VSHORT.LST	A short version of VLIST.LST
20300	8	WEEKDAY.LST	Find a day of the week
20310	8	YEARCOM.LST	Find year from COMPDAY

Assembly Language Routines — by Chapter

CHAPTER	FILE NAME	PURPOSE
3	SFILL	Fill the screen with a character
4	MOVER	Block memory move
7	SEEKER	Substring search
8	CLOCK	Real time clock
9	BITMAP	Test and toggle bits in a byte
9	BOOLEAN	Bit level Boolean operators
1Ø	SHELL	Shell sort
12	BLINK	Blinking cursor
13	SLOWLIST	Control LISTing speed
15	MINIDOS	DOS functions from BASIC

Application Programs — by Chapter

CHAPTER	FILE NAME	PURPOSE
3	CONVERT.BAS	Convert DOS binary load file into BASIC DATA statements
3	DATAPAK.BAS	Convert DOS binary load file or BASIC DATA statements into a string packed array
6	HEADER.BAS	Analyze header information of DOS binary load files
8	DATECOM.BAS	Perpetual calendar 1901-2099
8	CLOCK.BAS	Real time clock
12	BLINK.BAS	Blinking cursor
13	MARQUEE.BAS	A banner program
13	SLOWLIST.BAS	Control LISTing speed from BASIC
15	AUTOGO	Create AUTORUN.SYS files
15	CATALOG	Disk catalog program
15	RPMTEST	Disk RPM tester
15	MINIDOS.BAS	DOS functions from BASIC

Demonstration Programs — by Chapter

CHAPTER	FILE NAME	PURPOSE
3	SFILL.DEM	Fill screen with an ATASCII character
4	MOVER.DEM	Move string from top to bottom of video screen and then back again
4	WINDOW.DEM	Take a visual trip through memory
5	PROLAY.DEM	BASIC Overlay
10	BUBBLE.DEM	A BASIC bubble sort benchmark
10	SHELL.DEM	A BASIC shell sort benchmark
10	SHELL2.DEM	A machine language sort benchmark
10	SHELL3.DEM	A visual sort of experience
12	CONTROL.DEM	A menu using controlled inputs
13	SCROLL.DEM	Coarse scrolling demo
13	GLOW1.DEM	Make screen messages glow
13	GLOW2.DEM	A better glow routine
13	SLYDESHO.DEM	Page flipping demo
14	SOUND1.DEM	Channel Mixing (16 bit) effects
14	SOUND2.DEM	Interference effects

Index

*** E ***

*** F ***

*** G ***

*** H ***

*** I ***

*** K ***

*** L ***

*** M ***

*** N ***

*** O ***

*** P ***

*** V ***

*** W ***

*** X ***

COMPUTER BOOKS FROM IJG

APPLE

THE CUSTOM APPLE & OTHER MYSTERIES by Winfried Hofacker & Ekkehard Floegel. The complete guide to customizing the APPLE II. 190 pages, ISBN 0-936200-05-7 $24.95 retail.

ATARI 400/800/1200

ATARI BASIC, Learning By Using by Thomas E. Rowley. Learn ATARI BASIC easily through the many short programs provided. 73 pages, ISBN 3-921682-86-X $5.95 retail.

FORTH ON THE ATARI - Learning By Using by Ekkehard Floegel. Forth application examples for the novice and expert programmer. 118 pages, ISBN 3-88963-170-3 $7.95 retail.

GAMES FOR THE ATARI by Sam D. Roberts. Provides ideas and examples of computer games that can be written in BASIC. 115 pages, ISBN 3-911682-84-3 $7.95 retail.

HOW TO PROGRAM YOUR ATARI IN 6502 MACHINE LANGUAGE by Sam D. Roberts. Teaches machine language, the use of an assembler and how to call subroutines from the BASIC interpreter. 106 pages, ISBN 3-921682-97-5 $9.95 retail.

COMMODORE VIC-20

TRICKS FOR VICs by Sam D. Roberts. Introduction to BASIC and machine language programming. Also includes instructions for hardware modifications. 114 pages, ISBN 3-88963-176-2 $9.95 retail.

IBM PC

ELECTRIC PENCIL OPERATORS MANUAL by Progressive Software Design. (Available Soon) ISBN 0-936200-12-X

The IBM ELECTRIC PENCIL PROGRAMMING GUIDE by Progressive Software Design. A guide for programmers and hobbyists who wish to add their own programs to the IBM PC Electric Pencil Word Processing System (Release date to be announced).

SHARP 1500

GETTING STARTED ON THE SHARP 1500 & RADIO SHACK PC-2 by H.C. Pennington, G. Camp & R. Burris. Master BASIC programming fundamentals. Step by step instructions. For those with no previous programming experience. 280 pages, ISBN 0-936200-11-1 $16.95

TIMEX/SINCLAIR

ZX-81/TIMEX™ - Programming in BASIC and Machine Language by Ekkehard Floegel. For the Sinclair ZX-80(81) or Timex 1000. 139 pages, ISBN 3-921682-98-3 $9.95 retail.

If It's From IJG, IT'S JUST GREAT!

IJG, Inc. 1953 W. 11th Street
Upland, California 91786
Phone: 714/946-5805

ATARI BASIC Faster and Better
by Carl M. Evans

Unlock the full power of BASIC in your ATARI computer! In this book you'll find new BASIC programming ideas and techniques that have never been seen in print before. With the concepts, tricks and routines in this book, you will tap unimagined powers of the BASIC language to make your programs run faster and better.

ATARI BASIC Faster and Better begins with BASIC programming subroutines, "handlers" and "shells." Then it proceeds to machine language subroutines, magic memory techniques and BASIC overlays. Secrets of string manipulation, number crunching and sorting techniques are revealed, as well as controlled data entry. Keyboard trickery, video and sound routines, and useful utilities are explained in a clear style, with new solutions to common programming problems.

This book will free you to concentrate on your application. The subroutines, utilities and USR routines can be made a part of any program, making your BASIC program faster and better. There are complete listings for all programs. A library of four ATARI diskettes with all the major routines and programs in this book can be purchased separately, so you can put the programs to work immediately.

About the Author

Carl Evans has been writing on a professional basis since 1978, and is widely published in various technical and home computer magazines. His column "Tape Topics" and a technical help column ("Tangle Angles") have appeared in *ANTIC* magazine since 1982.

Carl is currently the manager of IJG's publications department. He also runs VERVAN, a software and documentation consulting firm that has developed an extensive series of machine-language utilities for the Atari computer.

He studied electronic engineering at the Georgia Institute of Technology and became involved with computers in 1971. Fortunately, for those of us wishing to know the secrets behind Atari's *magic*, Carl's "involvement" appears deep and lasting. A wizard has appeared, carrying a book of spells and incantations . . . .

ISBN 0-936200-29-4