THOMSON
COURSE TECHNOLOGY
Professional • Trade • Reference

WEB DOWNLOADS AVAILABLE

RETRO GAME PROGRAMMING

UNLEASHED FOR THE MASSES

EARL J. CAREY

Premier Press

SERIES EDITOR: ANDRÉ LAMOTHE, CEO, XTREME GAMES LLC

Retro Game Programming:

Unleashed for the Masses

Earl J. Carey

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

ISBN: 1-59200-906-9
Library of Congress Catalog Card Number: 2005921081
Printed in the United States of America

05 06 07 08 09 BH 10 9 8 7 6 5 4 3 2 1

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

Course PTR,
a division of Course Technology
25 Thomson Place
Boston, MA 02210
http://www.courseptr.com

Publisher and General Manager of Course PTR:
Stacy L. Hiquet

Associate Director of Marketing:
Sarah O'Donnell

Marketing Manager:
Heather Hurley

Manager of Editorial Services:
Heather Talbot

Senior Acquisitions Editor:
Emi Smith

Series Editor:
André LaMothe

Marketing Coordinator:
Jordan Casey

Project Editor:
Sandy Doell

Technical Reviewer:
Alex Varanese

PTR Editorial Services Coordinator:
Elizabeth Furbish

Interior Layout Tech:
Marian Hartsough

Cover Designer:
Mike Tanamachi

Indexer:
Sharon Shock

Proofreader:
Sara Gullion

Then she said, "I want you to love me as a poet loves his sorrowful thoughts. I want you to remember me as a traveler remembers a calm pool in which his image was reflected as he drank its water. I want you to remember me as a mother remembers her child that died before it saw the light, and I want you to remember me as a merciful king remembers a prisoner who died before his pardon reached him. I want you to be my companion, and I want you to visit my father and console him in his solitude because I shall be leaving him soon and shall be a stranger to him."

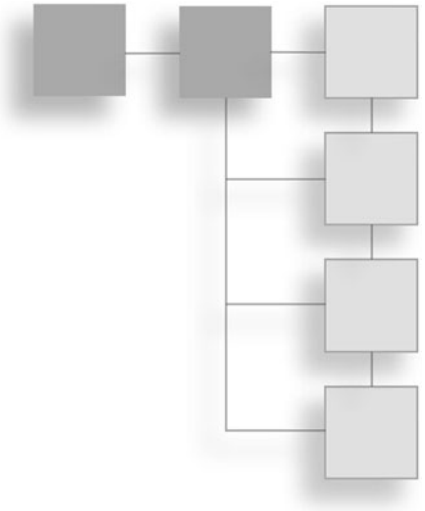
Kahlil Gibran, *Broken Wings*

To my mother who has passed away. This book, like all positive things I do, is dedicated to you and your memory.

"He was a genius—that is to say, a man who does superlatively and without obvious effort something that most people cannot do by the uttermost exertion of their abilities."

Robertson Davies, *Fifth Business*

This book is dedicated to all of the legends that made the video game industry, the computer industry, and indeed, the way of life we know today possible.



Acknowledgments

First I must acknowledge my mother, who died so long ago. For many years the loss of her tore me apart. While other children existed in a state of eternal bliss, my mind strained under the weight of the burden of trying to comprehend the fact that she would never see me, and I would never see her, again. Surely it was this that sobered me at such an early age and made me value every minute of every day. I was unable to find peace until I realized that as long as I lived my life to the fullest of my ability, she would not be sad. If I did something significant with my life, she would be happy. So I ran full speed ahead through life trying to accomplish something, anything, that my mother could be proud of. Only the faster I ran, the slower I seemed to move through life. After years of running and getting nowhere, I crashed and burned. There were many who loved me and wanted to help, but who can understand a motherless son?

I thank my Grandmother, Hazel Cooper; my Father, Earl Carey; and my Uncle Farion; my aunts; my uncles; my cousins; my sister and brother, Earnessa and Earlin Carey; my teachers; everyone who toiled with me and trained me over the years. I know now that it is not easy to raise a man. We did not always agree, but in the end, we all have to live on this Earth together. We may as well love each other and be done with it. I have resolved myself to do just that.

I thank my wife, Mitchlyn Carey. When I was in a dark place, she was able to bring me light. The two things I always wanted in life were to achieve greatness and to raise my family. The path to having my own family always seemed uncertain, because with all of the troubles of this world, I knew I had little control over finding someone I could trust to the extent required for marriage. You came and allowed me to find myself, and you helped to bring me from a very dark place. I love you and I appreciate you, especially for bearing

with me over the years as I moved from one project to the next, still trying to accomplish that one great elusive goal that I could not identify. Most of all, I thank you for my son, Zurial Earl Carey.

Zurial, what was I doing with my life before I had you? I do not even remember because it is impossible for me to envision a world without you. You give my life new meaning. Watching you grow amazes me. How could so pure an entity exist? I love you, and whatever positive thing it takes for me to ensure that you reach your full potential will be done.

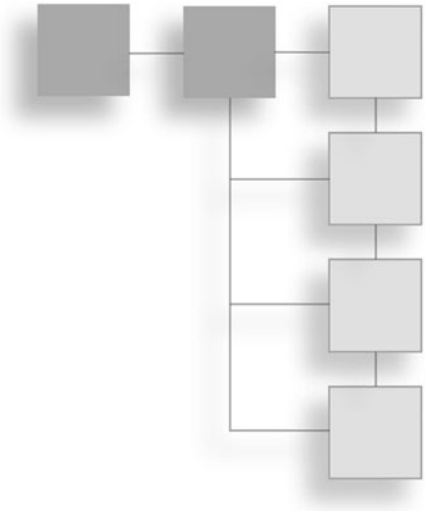
I need to thank my boss Kathy Ingraham for being so supportive of me in this venture. When I needed time off to work or to recover from a long weekend of labor or even to take a trip to California to go to conferences, I always had full support.

I need to thank André LaMothe on so many levels. Thanks for basically starting the whole game programming book industry. Like so many programmers around the world, I gained much knowledge from the books André wrote. This game programming series created an opportunity for my voice to be heard. He took a chance on me, and I will never forget it. Thank you.

I need to thank my project editor Sandy Doell. She was patient and stern at the same time. She kept me focused and made sure I got through author review. I need to thank Alex Varanese for helping me to keep my facts straight. I need to thank my acquisitions editor Emi Smith for her help. I need to thank Heather Hurley for helping me with the marketing of this book.

Special thanks goes out to Mr. Ralph Baer who assisted me with accuracy when writing the history section of this book.

Last, but by no means least, I need to thank all of the legends who created the game industry: Ralph Baer, Willy Higinbotham, Nolan Bushnell, Steve Jobs, Steve Wozniak . . . the list goes on. I thank everyone who made computers and their software. Everyone who grew up on video games and all who are now growing up on video games. All of you have helped to make the game industry what it is today.



About the Author

Earl J. Carey began programming on the TRS-80 color computer at the age of 5. He has created numerous C/C++, Visual Basic®, and assembly programs. He now leads a fulfilling career as a computer programmer and graphic artist and is currently the chief graphic artist/programmer of Capital City Marketing in Nassau, Bahamas. Carey recently delivered a lecture on Retro Game Programming at the Vintage Computer Festival 7.0. Visit his Web site at http://www.ristudios.net.

About the Series Editor

André LaMothe, CEO of Xtreme Games LLC and the creator of the XGameStation, has been involved in the computing industry for more than 27 years. He wrote his first game for the TRS-80 and has been hooked ever since! His experience includes 2D/3D graphics, AI research at NASA, compiler design, robotics, virtual reality, and telecommunications. His books are best sellers in the game programming genre and his experience is echoed in the Thomson Course Technology PTR *Game Development* books. You can contact André at ceo@nurve.net and www.xgamestation.com.

Letter from the Series Editor

Over 25 years ago in the mid-1970s, there was a "singularity" in the computer industry where in a single moment everything changed. This moment was, more or less, the introduction of the "computer" to the masses. Now, some historians will argue when this actually occurred. Some will say in 1974 when the Altair was released, others will argue it was the release of the Apple II in 1977. Still others will say that the creation of Atari and PONG in 1972 was the big bang. Whomever you tend to agree with more, there is no arguing that in a short period of time we had a "punctuated" evolution in the computing industry.

I was only a boy when this happened, but I can tell you it was the most exciting thing that I have ever been part of. Atari, for example, was the fastest growing company in history—period! People literally slept outside of Atari to try and get jobs there. And Apple Computer when it went public was the largest public offering in American history. Commodore Business Machines, when they acquired the Commodore computer, ended up selling more computers than anyone in history (at the time) making the Commodore C64 the world's best selling computer.

So what do all these companies and historic events have in common? Video games! For example, Nolan Bushnell, founder of Atari in 1972, wanted to create games; specifically, he wanted to create a cheap, high quality version of the *Spacewar!* game he had played while attending college. The results of this were *Computer Space* and the first steps of Atari. Atari was the quintessential, prototypical model of all the Silicon Valley companies to follow. Nolan Bushnell was the "rock star" of technology and games and the first Silicon Valley millionaire with the jet and the $25M in change to prove it from the sale of Atari. But wait; there's more. . . .

At the same time Atari was in its heyday in the late 1970s, a young programmer/tech named Steve Jobs was working there. Steve had a friend, Steve Wozniak, and together they would create Apple Computer. The interesting thing, however, is that the Apple computer and Steve Jobs' experience with customer satisfaction, marketing, and human factors all came from Atari. Steve Wozniak, the technical genius behind the Apple I/II, made the Apple simply to play games.

The stories go on and on, all of them intertwined, but all of them connected to video games in one way or another. Even the great duo, John Romero and John Carmack, were Apple programmers first; they turned to IBM PCs later.

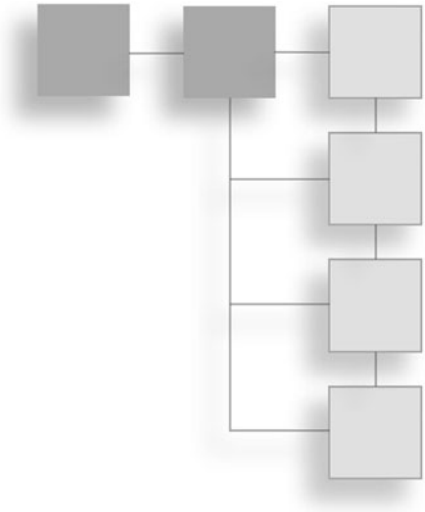
Retro game programming is not only fascinating from a technical standpoint, but the history and stories are even more fascinating to study. Entire empires were created because of video games! And the technology we have today has its roots in games, so studying this material and getting into the minds of the early hardware and software developers is a treat that everyone should indulge in. This book will introduce you to the brilliance of these early innovators, their machines, and their games.

With that, please enjoy *Retro Game Programming: Unleashed for the Masses* as your first step on this incredible journey of discovery.

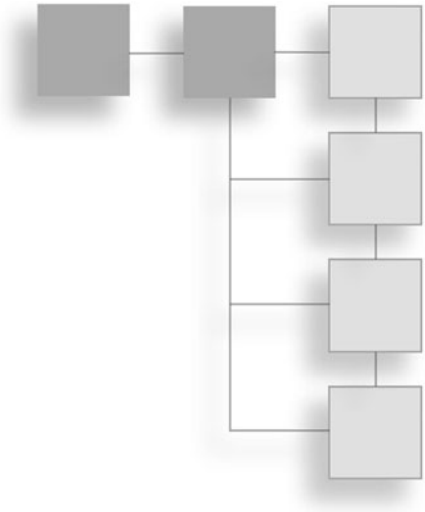
Sincerely,

André LaMothe
2005
Game Development Series Editor

Contents at a Glance



Contents



Introduction

If we value the pursuit of knowledge, we must be free to follow wherever that search may lead us. The free mind is not a barking dog, to be tethered on a ten-foot chain.

Adlai E. Stevenson, Jr. (1900–1965),
speech at the University of Wisconsin, Madison,
October 8, 1952

Twenty-one years ago, I read a book that changed my life. Today I hope to write a book that will change yours. This is not a self-help book or some form of new philosophy. This book is the gateway to the inner sanctums of game programming, past and present. Bold words, I know, but I believe in this book with my whole heart. To someone who has never touched a keyboard or written a single line of code, game development can seem daunting. In the old days, this was usually the result of a lack of information. Game programming books weren't available at your local bookstore, so it was very difficult to learn the skills needed to build games unless you were very intuitive and willing to dedicate a large amount of your time to trial and error.

Today, game programming is difficult to learn because there is almost too much information: DirectX, OpenGL, Vertex shaders, pixel shaders . . . the list goes on and on. Ironically, most information about modern computer systems is in the form of closely guarded trade secrets. Even if this were not the case, it would take a lifetime for the average person to master all that information. Finally, even if he could master it, the hardware would be obsolete long before he could use his knowledge!

In the 21 years that I have been programming, two things have not changed: Computers are based on binary logic, and the basic structure of games has not changed. It is these two facts that give me the courage to call this book the gateway to the game programming world past and present. Master the past to understand the present. The book I read 21 ago that changed my life, was the Users' Manual for the original TRS-80 color computer. It made computer programming easy to learn and formed the foundation for my entire programming career. The goal of this book is to make retro game programming easy for anyone to learn. After you are able to understand the underlying principles of retro game programming, it will be easier to understand the complexities of modern game programming.

By the end of this book, you will not only be building games, but you will have the foundation you need to understand how today's modern games work. If you still have questions, please feel free to contact me at info@retrogameprogrammingunleashed.com or at info@ristudios.net. I will help you in any way possible. You can also visit my Web site at www.ristudios.net for more information and a chance to interact with your fellow retro game programmers around the world.

The Web site for this book is retrogameprogrammingunleashed.com. There you will find many neat things, including the source code for the programs in this book, source code for even more retro games, bonus information, and links to many other sites of interest to the retro game programmer.

Computer programming has been very rewarding to me. Exactly 80.7695 percent of my life has revolved around either programming computers, fixing computers, or just making them do really "cool things." It is my hope that, through this book, you will find this journey as rewarding as I have.

The Significance of Retro Computer Systems

Every generation makes discoveries and innovations that are important. Every so often, a generation comes along and makes discoveries so profound that they change the way we think and live, and alter the very fabric of our lives. The innovation that was the catalyst for virtually all modern development over the past 20 years is the computer. The machines studied in this book are a part of a great legacy that should be preserved. Future generations need to know the role they played in the development of the computer industry, which in turn, has had an undeniable and absolute effect on society as a whole.

A wise man once said that, "The more things change, the more they remain the same." My grandfather used to tell me, "The only thing constant is change." When those two thoughts

merged in my mind, the result was the idea that "everything is constantly changing into another form of the same thing." For example, take the computer industry; new technologies come out every day that render ones only a little older obsolete. Yet the exact same principles are used as the foundation for both. You can take the principles of assembly language programming that you learned programming an Apple IIe and use them, with slight modifications and a memory map, to write code for an 8086, 286, 386, 486, and straight up to whatever happens to be the most advanced computer system in use as you read this book. While you may not use these systems to run your business, the principles that you will learn programming these machines will never be completely obsolete.

There is a big difference between a programmer and a normal person who reads the following in a computer manual in bold caps:

> **"THIS COMPUTER SYSTEM CANNOT PERFORM ANY OPERATION THAT IS NOT DESCRIBED IN THIS MANUAL."**

The "normal" person looks at the list to see if the features he needs are available. If they aren't, he moves on to another computer system. The programmer reads the same list, smiles to himself, takes a deep breath, and then spends the next sleepless week coding until he has created the features that the manual claimed were impossible.

Pushing computers beyond their limits is a key element in the spirit of the programmer. No other type of programming pushes personal computers harder than game programming. By learning game programming you will learn how to make use of every single piece of disk space, RAM, and silicon that the computer has.

It should be noted that game programming is, and always has been, a driving force behind the development of computer technology for the home user. No other type of software application pushes the computer to its limits the way video games do. Even today, when processors have long since crossed the multi-gigabyte milestone, computer games are pushing the boundaries of what a PC can do by fueling the development of ever more powerful and sophisticated video cards. The level of real-time photo-realism that is currently possible is staggering, and it is still growing. What is even more staggering is the fact that such technology is available to home users. "Necessity is the mother of invention," and no other application requires the same level of real-time 3D rendering and powerful processors that games do. Without games, I'm sure these technologies would still have been developed, but I think that such advanced technology might only have been available to universities or large corporations, just as mainframes were in the early days of computers. If there were no real need for individual users to have powerful desktop machines, they wouldn't exist.

Old School Meets New

For a long time, people have said that assembly language programming was dead. Every game programmer on the planet knew that this was absurd, yet that strange idea persisted. Occasionally though, something would happen to turn that idea on its head. When Intel released its new MMX technology, for instance, the only way to take advantage of the processor's full functionality was through the use of assembly language because these features had yet to be integrated into any of the high level languages! These features consist of 57 multimedia instructions that could perform functions normally and handle my video and sound cards, such as Digital Signal Processing (DSP). Programmers used assembly to create "MMX-enabled" software, which could use this new technology.

With the advent of Application Programming Interfaces (APIs), such as OpenGL and DirectX, many games today do not make use of assembly language programming! Instead of writing fast screen routines and other common functions needed to create a game, most programmers use the graphic functions found in the previously mentioned APIs (DirectX and OpenGL). Although many programmers might still use assembly language to program certain areas of the game or even to improve the speed and power of DirectX itself, computer processors are becoming faster every day, and high speed computer systems are becoming more inexpensive, so the need for such optimizations is, for the most part, unnecessary—with one exception—the video card.

The Atari 400 and 800 were the first computers that allowed you to write a program that was not designed to be executed by the processor but instead was designed to be executed by the video circuitry of the computer. In today's computer systems, this kind of technology has advanced greatly and is an essential part of modern graphics hardware. Vertex shaders and pixel shaders are the key to real-time photo-realistic games. The new bottleneck for game programmers is not so much getting the program itself to run faster but getting these smaller programs that are being executed by the GPU on the video card to run faster. Ironically, these programs are written in two languages: one resembling C, and another resembling assembly language.

You Really Can Learn to Program

I started programming when I was very young using the TRS-80 color computer. I could just leave you there with the impression that I was a very bright kid, but that is not the whole story. One of the biggest reasons I was able to start programming was that the programming manual that came with the machine was very well written. It was easy to read and made programming much easier. I am happy to say that after you have a firm grasp of programming principles, the path gets much easier and you can adapt to almost any programming language.

My goal in this book is to make retro programming as easy to understand as possible, while explaining the most advanced programming concepts in operation on these machines. If you follow along and apply yourself, you will be amazed at what you are able to do.

What this means for you is that the optimizations you must learn in order to write fast games on a retro game machine are the same kinds of optimizations that you must make in order to make these vertex and pixel shaders run faster. Age-old programming methods are the same ones used to perform today's most high tech programming!

Why not just learn to program vertex and pixel shaders on a PC? Well, for many people, this is an option, but for others, it is not. Today's computers, although more powerful, are more difficult to learn to program. These machines are a lot more complicated than retro game machines. There is much more that you need to know in order to write a game for a modern PC than you needed to know to write a game for a retro game machine.

To exacerbate the problem, there is an ever-increasing level of secrecy around today's hardware — "security by obscurity." Computer designers and manufacturers want to keep a competitive edge by making the inner workings of their hardware a closely guarded secret. When the computer industry was young, hobbyists and hackers were encouraged to experiment with computers. To that end, every single piece of information about the computer system was made public, including schematic diagrams of how the entire system was wired together! This information was often included in the Owners' Manual. Try finding it in the manual of any computer that you buy today — assuming that it even comes with a manual.

If you want to learn hard core programming, you have two choices: you can start on a modern PC where many things are mysteriously undocumented and you have to learn a new language. Or you can start on a retro game machine where everything about the computer system is laid out plainly, and the only thing that you have to learn is how to practice good programming. By choosing the latter and reading this book, you will have a firm foundation from which to learn how to program today's modern computer systems.

My Vision for This Book

I called this book *Retro Game Programming: Unleashed for the Masses*, but it could just as easily have been called *The Joy of Programming* after the famous line of cookbooks. The vision for this book is that even someone who has never touched a keyboard can learn how to program retro game machines, and people who are already able to program retro game machines will learn to do it better.

In writing this book, I am not trying to create blockbuster games. My goal is to make the concepts behind blockbuster retro games easy to understand so that you can create them. I want to make a cookbook of sorts. The last chapter contains recipes for games. Everything up to that point is designed to help you understand the ingredients used in those

recipes. This way, after you see how I have made my games, you can take those same recipes, modify them, and get started making games of your own.

You may want to go about reading this book in several different ways, but I will suggest two.

First of all, you might try reading the first part of the book first to understand how all this stuff works. Then you could read the chapters where we actually build games to see the concepts all put together. Next, start writing your own games.

I think the best way to use this book, however, is to set up your machine or emulator, fire up your assembly program, and load in the source for one of the games. Run the game and play it a few times. Next, just look at the code and see if you understand what is going on. Now read Chapter 10, "Putting It All Together: Building Games," to cement your knowledge of how the game was made and how it works. If you are unsure about a topic, then flip over to the chapter where that topic is discussed, read it, understand it, and then go back to deciphering the program. When you are done, you should know how to make your own arcade game!

Even though I gave you these suggestions, I would like for you to read this book the way that makes you feel comfortable. You know best what that is.

Now go wild, let your creativity run free, and produce the coolest video games possible.

CHAPTER 1

Bringing Your Retro Machine to Life

Setting Up Your TRS-80 Color Computer

The TRS-80 Color Computer (also known as the COCO) is a "deceptively" easy machine to work with. One of the great things about this machine is that you do not need to have a degree in computer engineering in order set it up. Furthermore, when the machine starts up it has everything you need to start programming right there onscreen in front of you. There are no diskettes to load or operating systems to worry about. There are operating systems such as Disk Basic and OS9 (not to be confused with Apple's operating system) and disks available for the color computer but you do not need to even look at these things until you are ready to.

Take a look at the photo of the back of the TRS-80 in Figure 1.1.

Figure 1.1 Photo of the back of the TRS-80COCO/ COCO2.

There are three switches and five ports. Starting at the left, the first thing you see is the reset switch. When you press this button and the machine is turned on it will reset the machine. What this means is that it will clear memory and all of the registers, basically placing the machine into the state that the machine was in when the computer first started. When the rest is completed you will see an OK onscreen.

Second from the left is the cassette port. This is where you plug your cassette player into the color computer to use as a storage device.

Third from the left is the serial port. This port allows you to connect all manner of devices to the computer. While it can be used to drive robots or for home automation, it also allows for more practical capabilities, such as using a modem to log into bulletin board services.

Next are two joystick ports. This is where we connect our joysticks in order to play arcade video games.

Sixth over from the left is the channel selection switch. This switch allows you to choose between using channels 3 or 4 on your TV screen to view the computer's output. The 7th element from the left on the back of the computer is the TV connection. This is where you will plug in the cable that connects your computer to the TV.

Finally the eighth element from the left is the power switch that turns the computer on and off. Over to the far right is the built-in power cord.

The beauty of this machine is that if all you want to do is jump in and start programming, we may completely ignore five of the eight elements on the back of the machine. As you advance and your knowledge of the machine expands, you can make use of the other elements. The greatest feature of this machine is that you can easily get started using its basic features without being intimidated by its more complex elements. As your knowledge grows, you can move on and make use of more and more of the computer's resources.

Figure 1.2 is a photo of an RF switch.

This is all you need to get started programming the TRS-80. Examine Figure 1.3.

As you can see from the diagram, all that you need to get started programming is to connect the computer to your TV using an RF switch and then plugging in the machine's power cord.

Turn on the machine by pressing the power switch and you are good to go.

Right now your screen should look like the screen shown in Figure 1.4.

Figure 1.2 Photo of the RF switch.

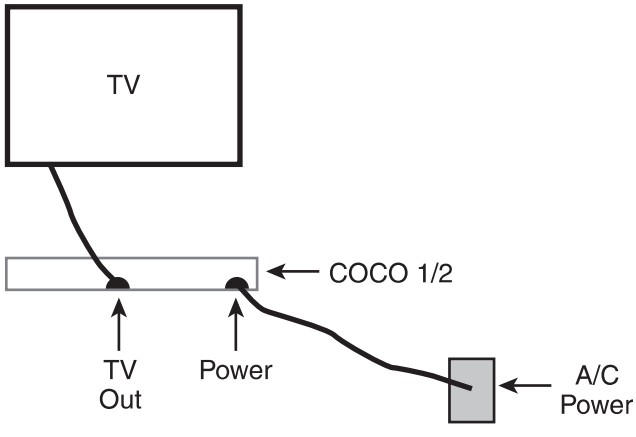


Figure 1.3 Diagram of basic TRS-80 color computer installation.

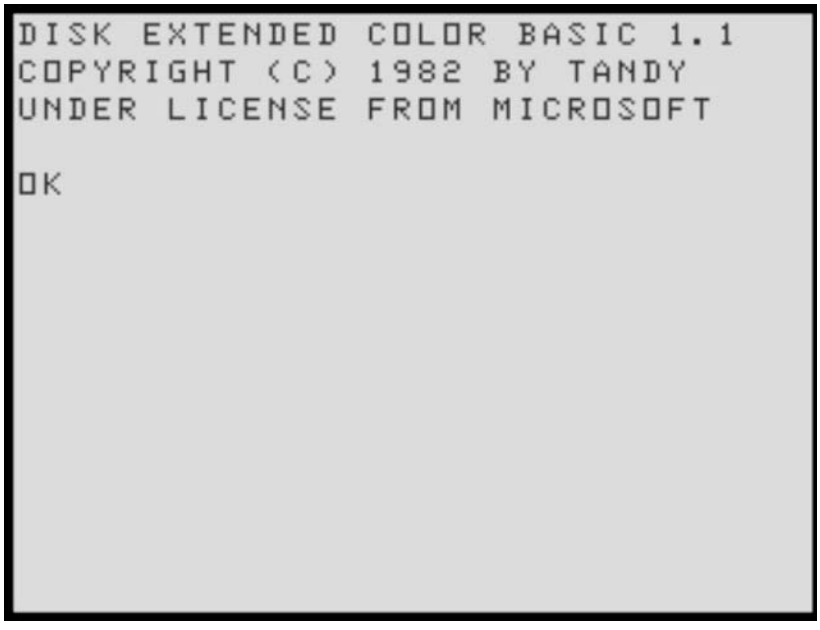


Figure 1.4 The TRS-80 boot screen.

You are now in extended color basic (or color basic if you are following along on a COCO1). The text onscreen tells you which version of basic you are using. Also note that you may have a color computer that has been upgraded to extended basic, in which case, you will know which system you have by looking at the case. (The COCO1 has a gray case while the COCO2 has a white case.)

Color Computer Storage Devices

You have two options for storage: disk drives or cassette tapes, discussed in the following sections.

Cassette Tapes

We will discuss cassette tapes first. Fortunately, the tape cassette is very easy to install. Figure 1.5 is an illustration of how to install the TRS-80's CTR-80A cassette recorder. There are other cassette recorders that may work, but the connection will be different, and there is no guarantee that they will function correctly.

One end of the cable has a single connection, pictured in Figure 1.6.

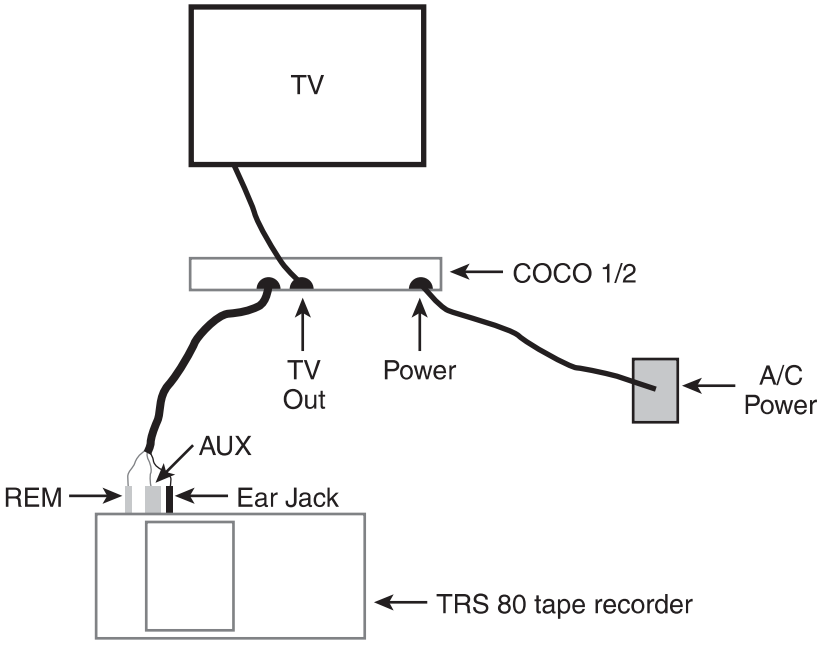


Figure 1.5 Illustration showing the installation of the cassette recorder.

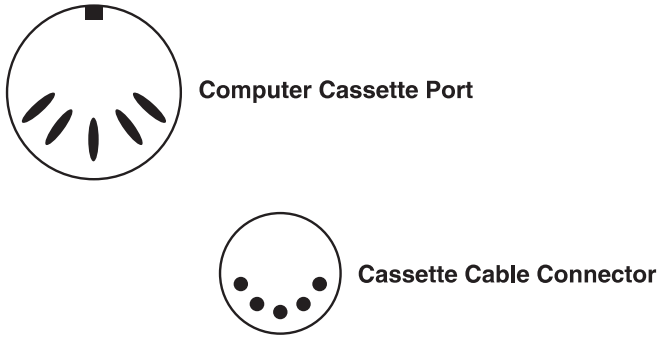


Figure 1.6 Illustration showing the connection that plugs into the computer.

Look at the U shaped pattern of this pin. The cassette port on the back of your TRS-80 has the same U-shaped pattern of indentations. Line the plug up next to your computer's cassette port so that both of their U-shaped patterns line up. Plug it in.

Next, take a look at the three plugs on the other end of the cable, illustrated in Figure 1.7.

Figure 1.7 Illustration of the other end of the cable.

The small gray plug connects to the REM jack, the large gray plug connects to the AUX jack, and the black plug connects to the ear jack.

Finally, plug the recorder power cord into the wall's power supply.

Disk Drives

Setting up the floppy disk drives is just as easy. Look at the diagram in Figure 1.8.

There are two cables coming out of the floppy disk drive. One looks like a power cord, and the other looks suspiciously like a game cartridge. This appearance is a hint as to how the floppy drive is connected. The cable that ends with an improvised cartridge case (shown in Figure 1.9) is plugged into the TRS-80's ROM drive on the right side of the computer.

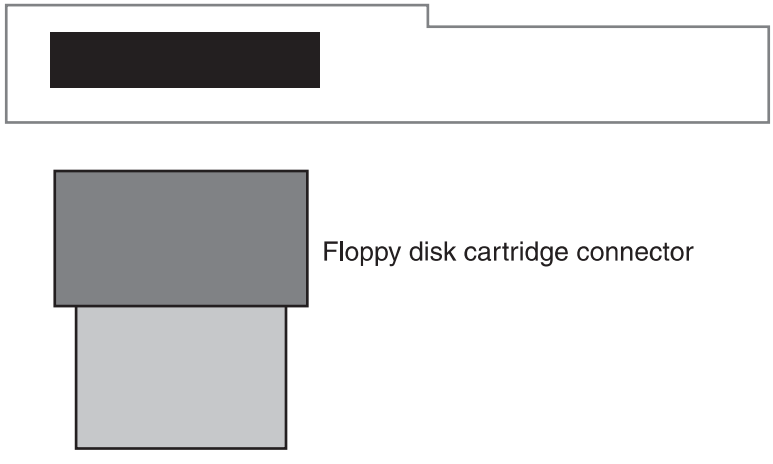


Figure 1.8 Diagram of the installation of the floppy disk drive.

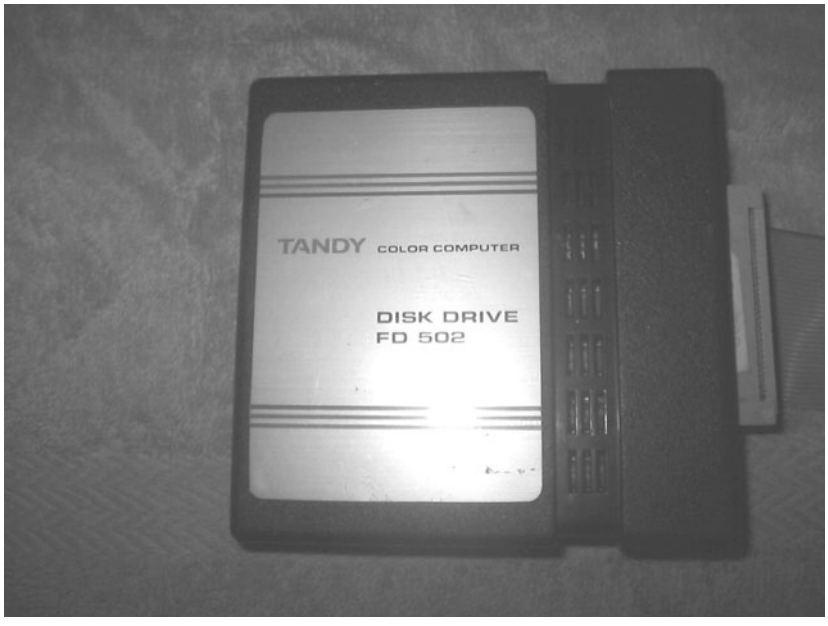


Figure 1.9 Photo of the ROM cartridge connector.

Next we plug the power cord into the wall socket. After turning the power switch on the back of the floppy disk into the on position, you are ready to go. When you turn on your computer with a disk drive connected, the Disk Basic operating system will automaticaly be loaded which will give you the ability to interact with the disk drive.

Setting Up Your Atari 400/800

Before you can use the Atari 800, you have to set it up. The exact cables that you need to install an Atari 800 vary slightly depending on whether you are connecting it to a TV or to a monitor. Following is a list of the devices that you will need to install your Atari.

Power supply (Figure 1.10)
Serial Cable (Figure 1.11)
Joy Stick (Figure 1.12)
Disk Drive (Figure 1.13 and 1.14)
RF Switch (Figure 1.15)
In Line connector (Figure 1.16)
Video Cable (Figure 1.17)

Figure 1.10 Power supply: You should have two of these, one for the Atari Machine itself and one for the disk drive.

Figure 1.11 Serial Cable: This is used to connect the computer system to the disk drive.

Figure 1.12 Joystick: Of course no video game system would be complete without one of these. You will use this to control all of the action on your Atari console.

Figure 1.13 Atari 810 disk drive front view: This is used to store your programs.

Figure 1.14 Atari 810 disk drive rear view: This is used to store programs.

Figure 1.15 Generic RF switch: This is used to connect your Atari to the television.

Figure 1.16 In-line connector: You will have to use this along with your RF Switch to connect the Atari to your TV.

Figure 1.17 Generic Video Cable: This is used to connect the Atari to a computer monitor.

Installation Instructions

Now, you have gathered all the parts needed to fire up the Atari 800, so let's bring this "baby" to life.

Installing the Power Supply

First things first; we have to give our machine some power. Look at the Figure 1.11 above for a picture of the power supply. Set the Atari before you with the keyboard facing you. Put your hand on the right side of the machine and turn that end of the Atari to face you. What you see should look like Figure 1.18.

Figure 1.18 Side view of the Atari 800.

Monitors and Receivers

At first glance a TV and a computer monitor may seem to be exactly the same. Indeed, today with the plasma screens being used for both television and computers, they often are. Plasma screens often come with enough input options to allow them to be used interchangeably. Traditionally, however, computer monitors have the advantage of a much higher resolution and provide a crisper image than television. The downside is that they usually do not have the circuitry needed to convert television signals to an image on the screen. A television set does not have the same high resolution of a computer monitor, but it does possess the circuitry to convert a television signal to an image on its screen.

When you connect the Atari to a computer monitor, it passes a standard composite signal to the monitor. A composite signal carries the data that determines the brightness and color of each point on the screen.

Things get a bit more involved when you connect your television to the Atari. The television was designed to receive a composite signal that has been mixed with an RF (radio frequency) signal. A composite signal has all of the information that is needed to display an image on the screen. The problem is that a composite signal is not strong enough to travel through the air.

If all we had were composite signals, broadcast television would be impossible. RF signals are strong enough to travel great distances through the air. These are the signals that are used on AM and FM radios. By combining RF signals with composite signals, we can send television images great distances through the air.

In order for your Atari to display images onscreen, it has to take its standard composite signal and combine it with an RF signal so that the telvision is used for receiving.

This technique, which is called *crowbar modulation*, was actually invented by Mr. Ralph Baer as a way to display the images from his games on television screens.

The connections that you see going from left to right are as follows:

- Monitor port: This is where we are going to plug our monitor into the computer.
- Serial port: This is where we connect the disk drive to the computer.
- Channel selection switch: You can use this to decide whether your Atari should function on channel 2 or 3 if you choose to use a TV to play your games.
- Power switch: This is the master switch, which is used to turn the Atari off and on.
- The power port: This is where the power supply is plugged in.

Be sure that the power switch on the power supply is in the off position. Pick up your power supply and examine both ends. You will notice that one end looks just like a regular drop cord. Plug this end into your wall socket or surge. In Figure 1.19, you will see a picture of the other end of the power supply. Plug this end into the power port of your Atari 800.

Figure 1.19 This side of the power supply plugs into the Atari's power port.

Connecting the Atari to a Monitor

You can use either a TV or a computer monitor to view the action on your Atari.

First, we will go through the steps for connecting your Atari to a computer monitor. Figure 1.20 pictures the monitor port on the Atari. Figure 0.21 is a picture of a generic video cable. One end of this cable separates into four separate cables; the other is round and cylindrical. It is this cylindrical end that we are interested in first. Look inside this end of the cable and make note of the pattern of the pins inside. Now, look at the pattern of the holes in the monitor port of the Atari. You will see that they both form a half circle as seen in Figures 1.20 and 1.21.

Now take this same end of the cable and hold it just below the monitor port, shown in Figure 1.21.

Turn the cable until the patterns of the port and the cable line up with each other. Now push the cable into the port. Connecting the other end of the cable to the video monitor is going to be tricky. You have to get the pinout for the generic video cable. This tells you which pin on the cylindrical end of the cable corresponds to which pin on the other side of the cable. This is very important for you to know when connecting any of these computers (except the Apple) to your monitor. Next, after finding this information, you will need to know the pinout information for the Atari 800's monitor port. This information is provided for you in Figure 1.22.

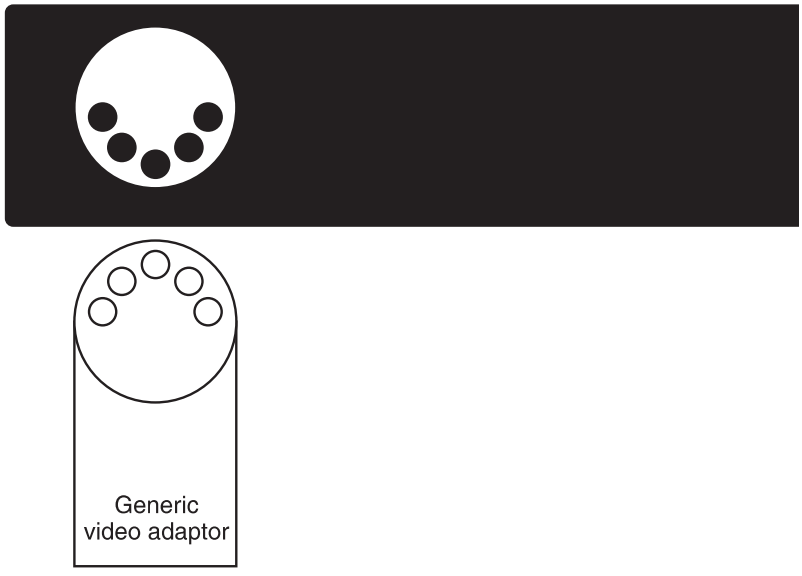


Figure 1.20 The Atari's monitor port and the generic video cable have matching patterns.

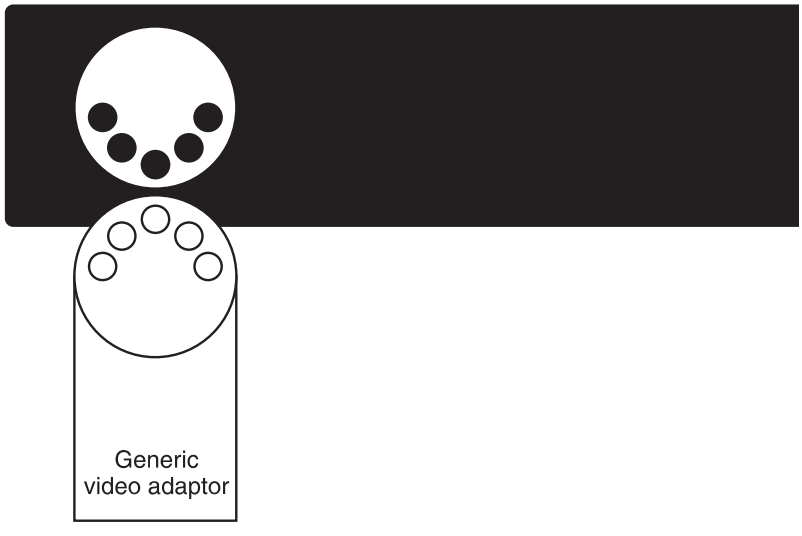


Figure 1.21 Be sure to match the patterns of the Atari's monitor port and the generic video cable.

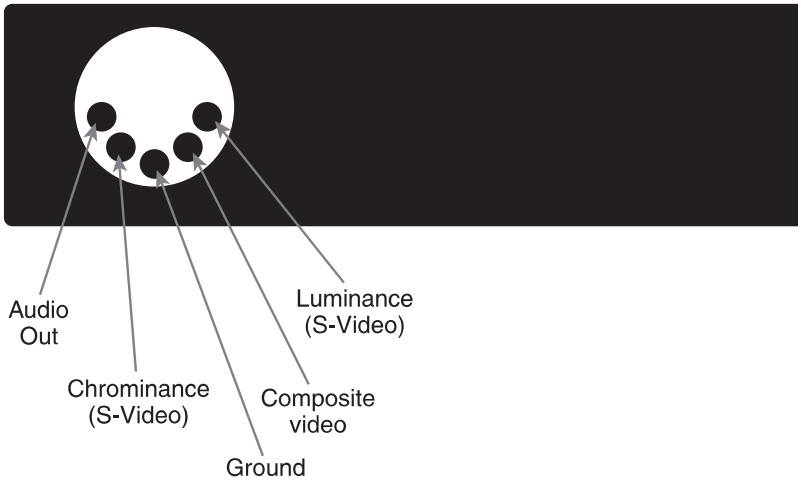


Figure 1.22 Pinout of Atari's monitor port.

Look at Figure 1.23 carefully. Make note of which pin is used to transmit *luma*, which transmits *chroma*, and which transmits *audio*. Now carefully pull out the monitor cable from the Atari and hold it as shown in Figure 1.23.

Note

Chroma is short for chrominance and refers to color. *Luma* is short for Luminance and refers to brightness. The Chroma line controls the colors on the screen while the Luma line controls the brightness of the colors on the screen. The *audio* line transmits the sounds that people playing your game are going to hear.

Make a note of which pin on the cable corresponds to the connection on the monitor port for Luma. Now make a note of which pin connects to the connection for Chroma and Audio. Now you have all of the information you need to hook up your monitor. This information is laid out in Figure 1.24.

On the back of your monitor, you will see connections labeled Luma, Chroma, and Audio. Use the pinout information that you found for your generic video cable to identify which cable is transmitting luma, audio, and chroma from the Atari and connect them to the appropriate port on the monitor.

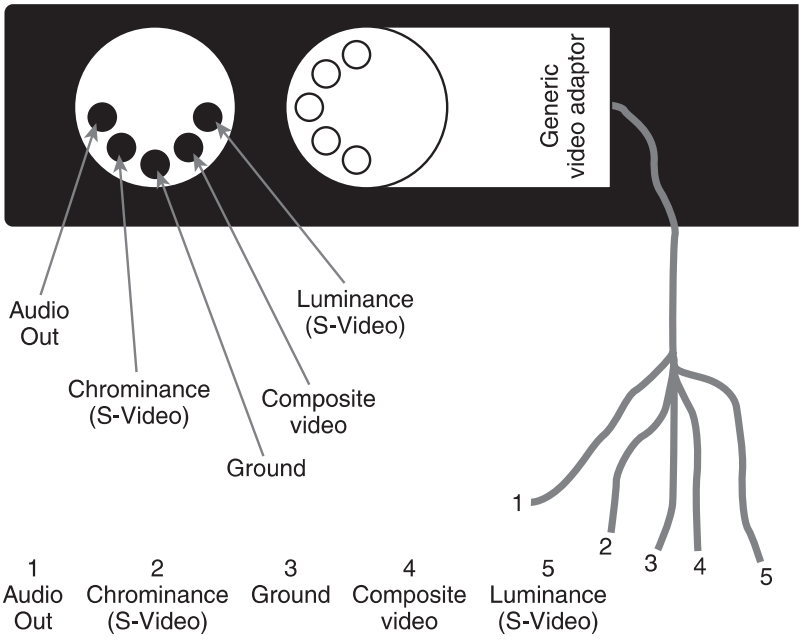


Figure 1.23 Make a note of how the pinout of your generic video cable matches up with the pinout of Atari's monitor port.

Pin Outs

	Audio out	Chrominance	Ground	Comp video	Luminance
Atari	1	2	3	4	5
Cable					
Monitor					

Figure 1.24 A diagram of all the information you need to connect your Atari to a computer monitor.

Connecting Your Atari to a TV

Most of you will be connecting your Atari to a TV though. For this configuration you will need three things:

- An RF switch
- An inline connector
- A TV

Located at the rear of the Atari you will find a built-in RF cable. The original Atari came with its own RF switch, which had a female input. For this reason, the built-in RF cable on the back of the Atari had a male connector (take a look at Figure 1.25 for an example of the differences between male and female connectors). Most RF switches today have a male connection that is designed to plug into a female socket at the back of the computer device you are working with. For this reason, you probably will not be able to directly connect your Atari to your RF switch unless you are lucky enough to find an original Atari RF switch.

This is where the inline connector comes into play. The inline connector has female connections on both sides. That helps us because we can connect the RF switch to one end and the Atari 800 to the other, as seen in Figure 1.26.

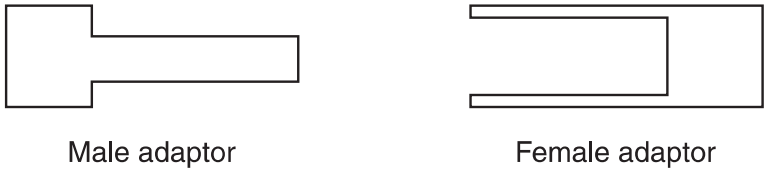


Figure 1.25 The relationship between male and female connectors.

Figure 1.26 You can use an inline connector to hook the Atari up to your TV using a modern RF switch.

It's a Boy!

Whether you are working with electronics, plumbing, or any number of other fields you will come across the terms male and female. These terms are usually found where we need to connect one device or cable to another, as in the case of connecting the generic video cable to the monitor port. Generally, one connection will be concave and the other will be convex. The convex connection will typically plug into the concave. As an example of this, think about when you plug your TV in to a wall socket. The plug from the TV is convex and is called male. The wall socket is concave and is called female. The male drop cord plugs into the female wall socket.

Installing Your Disk Drive

Now it is time to connect your disk drive. First, get your second power supply and plug the appropriate end into your wall socket or surge protector; plug the other end into the power socket on the disk drive. Next connect one end of your serial cable to the disk drive and the other end to your Atari as shown in Figure 1.27.

Now you must adjust the jumpers at the back of the disk drive so that the Atari will recognize this as disk one. Take a look back at Figure 1.14 to see a picture of the rear of the Atari 810 disk drive. You will notice that there are two identical I/O connectors. The disk has two connectors so that we can *daisy chain* a number of disks together. What this means is that we could actually string a number of disk drives together by connecting a serial cable from the Atari itself to the first disk drive by inserting the cable into one of the I/O connectors. We could then connect a second disk drive by running a serial cable

Figure 1.27 Both ends of the serial cable are identical, so don't worry about which end goes where. After it is properly oriented it will fit just fine.

from the second I/O connector to one of the I/O connectors on the second disk drive. This process can go on until you have connected a total of four disk drives as seen in Figure 1.28.

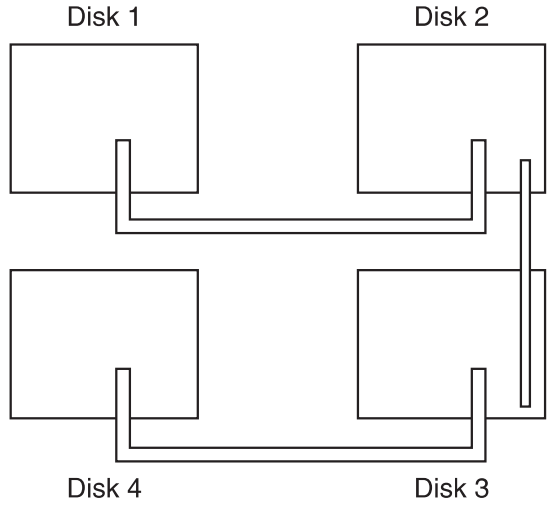


Figure 1.28 We can connect up to four disks together by using a daisy chain.

Once you have daisy chained several disk drives together, or even if you only have one disk drive, the computer needs to know which drive is which. For this reason, on the back of the Atari 810 disk drive there are two binary switches. These two switches can be adjusted to form any one of four combinations. Depending on the combination that these switches are set to on each disk drive, the Atari will be able to tell which is disk 1, 2, 3, or 4. These combinations can be seen in Figure 1.29. Figure 1.30 illustrates daisy chaining and the disk settings for each switch.

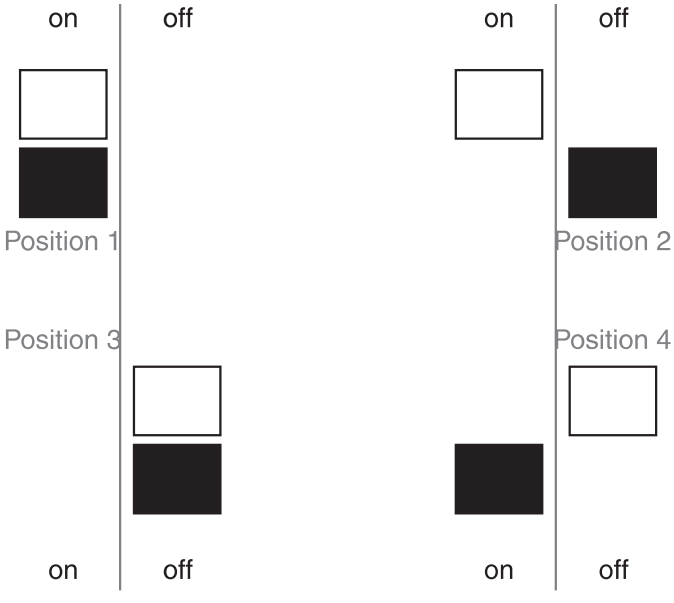


Figure 1.29 Set the two binary switches on the back of each disk drive so that the Atari can tell which disk drive is which.

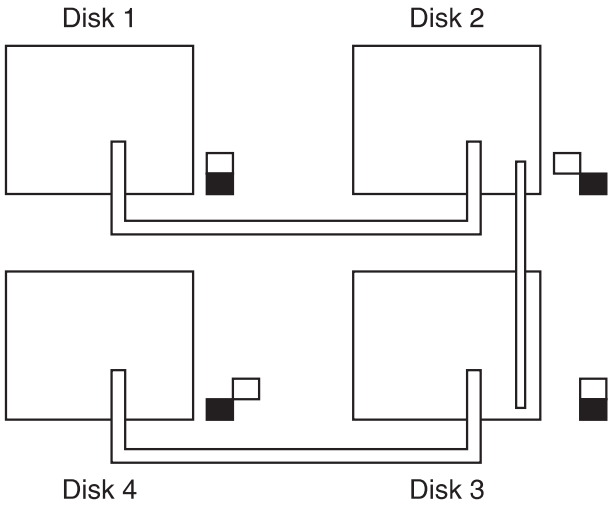


Figure 1.30 This diagram illustrates both daisy chaining as well as the appropriate switch settings for each disk in the daisy chain.

Connecting the Joystick

We are almost ready to rock and roll. There is just one more step; you must plug in the joystick. The diagram in Figure 1.31 demonstrates how to do this.

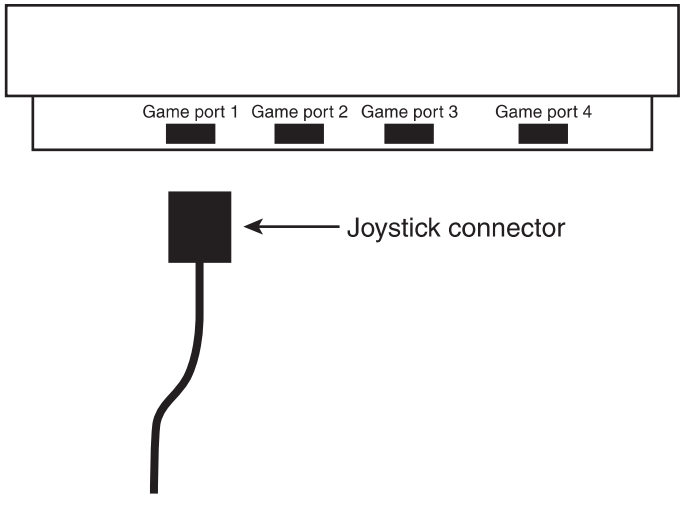


Figure 1.31 Connecting your joystick is easy.

Setting Up Your Commodore 64

Like the Atari and the Color Computer, the Commodore 64 is very easy to set up. Before you get started, make sure that you have the following items, pictured in Figures 1.32 through 1.36.

Figure 1.32 Commodore 64.

Figure 1.33 A power supply.

Figure 1.34 If you are using a monitor, you will need a video cable.

Figure 1.35 If you are using a TV, you need a TV switch box (sometimes called an RF switch).

Figure 1.36 You will need either a TV set or a vintage monitor.

When you have all of these items, you are ready to get started. Examine Figures 1.37 and 1.38.

Figure 1.37 Side panel of the Commodore 64.

Figure 1.38 Back panel of the Commodore 64.

If you do not have a dedicated monitor for your Commodore, you will have to use a TV set. Fortunately, this is no problem to do.

The first thing that you need to do is connect your computer to some form of video output. These steps will vary depending upon whether you are using a monitor or a television set. If you are using a TV, you will attach one end to your computer's TV connector and the other end to the back of your set. Figure 1.39 below illustrates.

Connecting your Commodore is just as easy; it just requires a bit more attention to detail. It is easy to tell which end goes into the computer. See Figure 1.40.

The other ends of the cable will vary slightly depending on the manufacturer. There will usually be a white or red connector that must be connected to the video input on your monitor. Your cable should come with a pinout. Compare it to Figure 1.41 to be sure that you are connecting the right cable to the right inputs.

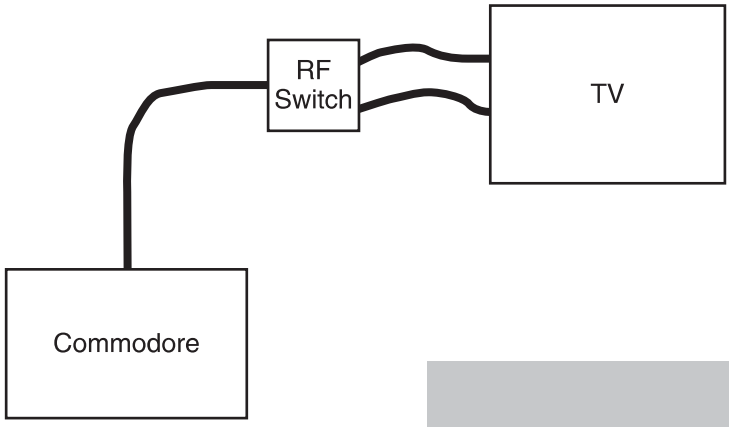


Figure 1.39 Here we see just how easy it is to connect our Commodore to our TV set.

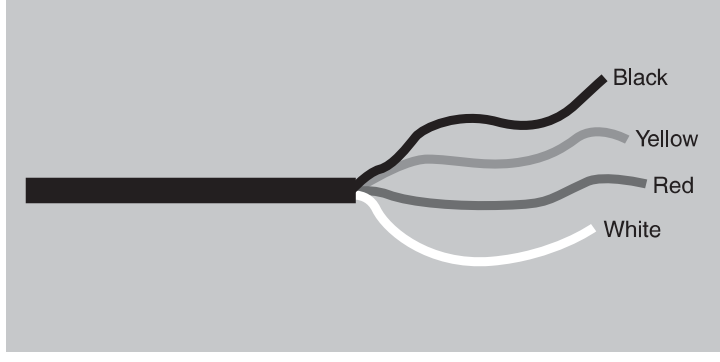


Figure 1.40 This end of the video cable gets connected to the computer.

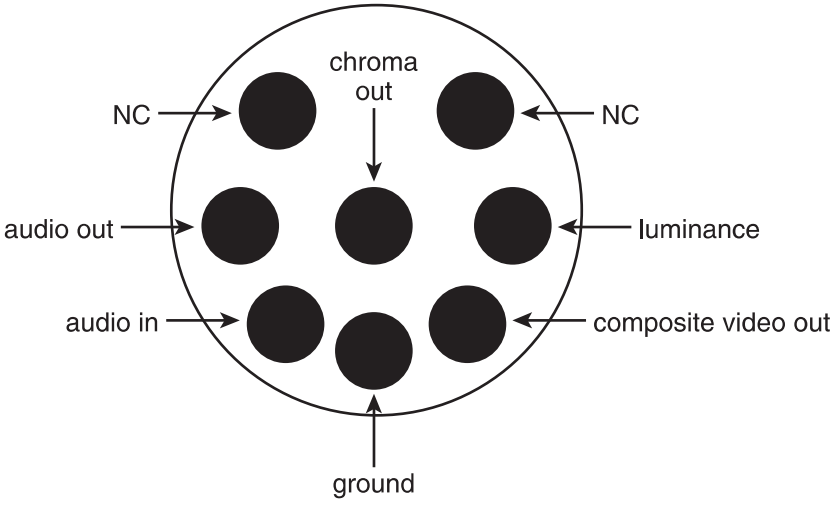


Figure 1.41 Use this pinout to make sure that you are placing the correct part of the cable.

After you have confirmed that you know where to put each cable (see Figure 1.42), you can connect your monitor.

Figure 1.42 Your array of cables.

Now all that you have to do is connect the power supply to your computer and you are almost ready to rock. There is just one more thing you need to do, and that is connect your disk drive.

Simply make sure that your computer is off and the disk drive is switched off; then connect the drive.

Setting Up Your Apple II

This machine comes in a few different flavors. The one that I am going to be using is the IIe Platinum Edition.

In order for you to set up your Apple II, you need the following items:

Retro Computer Monitor
Video Cable
Power Cable
Disk Drive
DOS Boot Disk

You can connect your Apple II using the diagram in Figure 1.43.

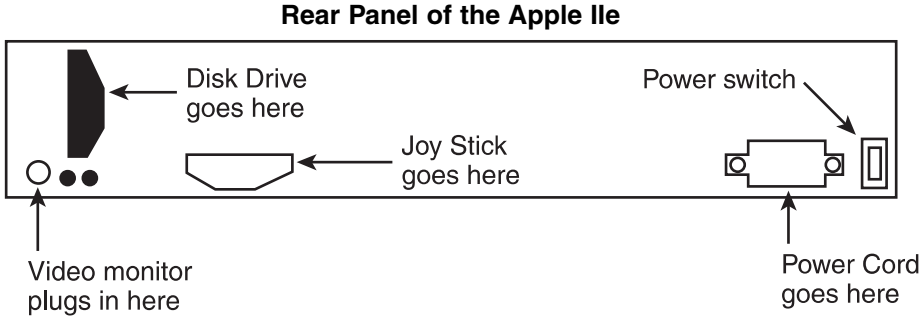


Figure 1.43 Basic configuration used to set up our Apple II.

Let's go through the process step by step. The first thing you need to do is to ensure that your power switch is in the off position and the power switch is disconnected. Also make sure that your computer monitor is turned off.

Now that you are sure that there is no power going to your computer, you can begin the process of installing the machine.

Connect one end of the video cable to the video port of the Apple II. Plug the other end into the chroma port of the video monitor.

Now connect the disk drive to your computer. Look at the diagram in Figure 1.43 and locate the disk port. Find this port on the back of your Apple II. This is where you are going to plug in your disk drive. Connect the disk cable to your computer and place the disk drive on the side of your computer or on top of it. Place your boot disk into the disk drive and close the door.

Connecting a joystick is usually optional, and you do not need one in order to use your Apple computer. If you are going to be playing games, however, then a joystick is a must. Find the joystick port on the back of your Apple computer. You can use Figure 1.43 as a guide.

Finally, double check to make sure that the power switch is in the off position and plug in the power cord. You are now ready to go; turn on your computer monitor and flip the power switch on the back of your computer into the on position. Your Apple II will now roar to life. Welcome to the world of the Apple II. You are now ready to get started working with your retro machine.

CHAPTER 2

Simply Complicated Game Programming

I want you to understand that game programming is easy and complicated at the same time. Look at the map shown in Figure 2.1.

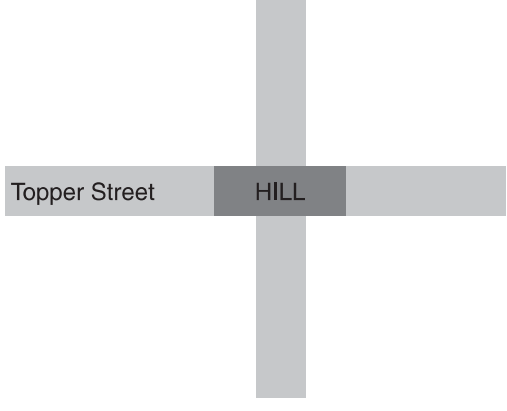


Figure 2.1 Map of a fictitious neighborhood.

If you wanted to direct someone to your house from point A, you could just say, "Go straight on Topper Street until you reach the big hill. Take a right and you are there." Easy. Not so easy, however, when you speak English and the other person speaks Greek. Or you speak Greek and she speaks Japanese. Communication becomes very difficult when you and the other party do not speak the same language. This is also true when that other party is your computer. In this example, you could also communicate directions to your house by drawing a map or using dictionaries and simple words and gestures to communicate. These methods will work, but ultimately they are too slow to be useful under a stressful situation where information needs to be communicated very quickly.

Programming in BASIC (Beginner's All-purpose Symbolic Instruction Code) is the equivalent of drawing pictures and maps or slowly using a human-to-machine dictionary to communicate with your computer. You speak in a human language, and the computer

speaks in binary. BASIC is designed to get you started communicating with your computer. You can tell it to print files, draw graphics, and perform really complicated procedures. You can have it do almost anything as long as you do not want to do anything really super fast. Because of the way BASIC is designed and the innate limitations of "High Level Languages" (computer languages that closely resemble the English language), programs written in BASIC run very slowly.

Imagine you are sitting right next to your best friend, who we will call Cindy. You tell her to go and pick up a ball on the opposite side of the room. Your friend gets up right away and does as you ask. Your instructions were carried out very quickly, just the way you like it.

Now imagine that you are still sitting next to your friend. There is another person in the room, who we will call Tom. Tom speaks a language you do not understand. Your friend knows how to speak a little of Tom's language. The only way for us to get Tom to go and get the ball would be for us to ask Cindy to ask Tom to go and get the ball. It takes time for you to ask Cindy to ask Tom to get the ball. It takes time for Cindy to remember how to say that in Tom's language. It takes time for Cindy to ask Tom to get the ball. It takes even more time for Tom to make sense of the translation before he actually gets the ball.

In the first example, we spoke directly to our friend and our instructions were carried out immediately. In the second example, we spoke to Tom using an interpreter (Cindy), which took much more time. The second example is very similar to the way that BASIC works. BASIC is usually designed in the form of an interpeted language. You type your program into a text editor and then you run your program. When you run your program, an interpreter that is usually found in memory reads your instructions and loosely interprets them into binary, which the computer can understand. The effect is similar to asking Cindy to tell Tom to pick up the ball for us. The job will get done, but it is going to take a lot longer than if we were able to tell Tom to pick up the ball directly.

Some flavors of BASIC do come in a compiled version. What this means is that rather than our BASIC program being interpreted and executed at the same time, the program is interpreted and stored on disk in the computer's native language where it can be executed at any time. This is the equivalent of having Cindy tape record the instructions for Tom to pick up the ball. We can play the instruction to Tom any time we wish, which would speed up the process a bit. The only problem is that these languages usually are not very optimized. What that means is that they will usually create more lines of binary instructions than we really need. If we need the computer to multiply the value of two variables, the computer may generate 10 lines of binary code when we really only need 5.

What this means is that it takes a longer time for the computer to read the program before it can actually carry out the instructions found in the program. Also some instructions take longer than others to execute. The BASIC compiler will not look to find the best

instructions to use in specific situations, which means that its choice of instructions will not be the best combination of instructions for our program.

To get the ultimate speed from your computer, you would have to learn binary as the pioneers of computer science did. Unfortunately, applications, especially games, are much more complex than those that ran on early machines, so writing them in binary is out of the question. We must limit ourselves to the next best thing, which is writing in assembly language. Assembly mangles human and binary into a language, which is just enough like English to be understood by humans and close enough to binary that it can be easily assembled accurately into binary.

Because BASIC is an introductory computer language, it is designed more in favor of being understood by humans. Its commands reflect human concepts: Print something onscreen, make a sound, draw a circle, and so on. Assembly language, on the other hand, is designed more in favor of computers. The commands reflect the way computers think: move bits of information from memory to a register or from a register to a register, test these bits, and so on.

At first, the way the computer thinks may seem a bit odd. It becomes easier to understand when you know one very important point. Anything that your computer does, no matter how complex it seems, no matter how intelligent it seems, as awesome as the latest 3D shooter looks, every single thing the computer does is done by moving bits from one location to another.

At its core, a computer is made up of millions of logic gates. These gates are given names such as *And Gates*, *Or Gates*, *XOR Gates*, and so on. These gates control the flow of binary data through the circuitry. While we call these circuits gates, they act more like switches. Each gate will have two or more inputs (places where binary data can enter the switch) and one exit. When there is no data coming in to the gates, the exit will usually be closed, which would mean that no binary data is leaving the gate.

In the case of an AND gate, the gate's exit will be opened (thus allowing for binary data to flow out of the gate) only when we have binary data coming into all of the inputs. An OR gate will open when either the first OR, the second OR, or any of the inputs receive data. An XOR or Exclusive OR gate will open when any one of the inputs receive data but will cut off if more than one of the inputs gets data. It is these basic logic functions that allow the computer to control the flow of binary data flowing through the machine.

These same logic gates are re-created in software using logical statements and are used to control the flow of the program. This allows us to give instructions like "If variable VB and a variable XC are greater than 50 then go to line 100."

While most of what the computer does is move bits of binary data from one location to another, it is binary logic that allows for the binary data to flow through the computer and

to be stored at specific locations. Logical statements control the order in which the instructions in our programs are executed. This allows us to apply some logic to the way in which the binary data is manipulated.

I am typing this book on a laptop computer. From my perception I am pressing keys on the keyboard and somehow characters are magically appearing onscreen. To the computer, on the other hand, every few seconds a random set of switches is being flipped. Each time these switches are flipped, a pattern of eight bits is placed into a part of the computer's memory (don't worry if you do not know what a bit is; I will explain this shortly). Several times a second, the computer moves that pattern of eight bits to the video memory where video circuitry puts it onscreen. This is a rather simplistic way of looking at how the computer works, but it does emphasize our point. Moving bits of information around performs most of the complex processes the computer does.

Once we learn how to effectively communicate with our computers, programming video games becomes very simple. We tell the computer to put an image onscreen, do some stuff in the background, and finally put a new image onscreen. We do this over and over again, and from the perspective of the player, a game is being played onscreen. You and I know better though. We know that it is really a very basic operation being repeated again and again. Everything else is just an illusion.

The tricky part comes in knowledge. We need to know exactly how to put images onscreen. We need to learn more about the "stuff" that we need to do in the background, and of course, because we know how to put an image onscreen we know how to put a new image onscreen. On the other hand, after we know how to do these three things we can build any game that our hearts desire.

Game Systems: Similar but Different

All game systems are the same! They all have some form of input. They all have some form of video output. They all have some form of sound output. They all speak some dialect of assembly language.

All game systems are different because they interact with these elements in slightly different ways.

When you begin with a new game system, you need a few different bits of information in order to work with it.

Assembly Dialect

In any computer system, the CPU (Central Processing Unit) is the brain of the computer. Different CPUs understand different dialects of assembly language. All of these dialects, however, are usually similar, and generally use the same classes of instructions.

As mentioned earlier, the majority of the work that computers do involves moving bits of data around the computer, so the largest *class* of instructions will always be Move instructions. These instructions move data from one memory location to the other, from memory to registers, from registers to memory, and from register to register.

Another class of instructions is the arithmetic instructions. These allow the computer to perform basic calculations such as addition, subtraction, multiplication, and division.

The next class of instructions are logical operators, which form the basis of all computer logic and come into play whenever the computer must make a decision.

Working hand in hand with the logical instructions are the branch instructions. These allow the program to jump back and forth from one group of instructions to another. This allows the program to change the way it operates depending on what happens while it is running.

Finally, each CPU tends to have a few miscellaneous instructions to assist you. These instructions are useful but do not fit into any of the above categories. An example of this is the NOP instruction. This instruction literally performs no operation. We use this to "pad" our program. When we run an assembly language program, it is loaded into memory and each line is given an address. Certain instructions, such as jump or logical instructions, need to know the address of each line so that they can properly control the flow of the program. When we add new lines of code to the program, we change the address of the lines in our program. To solve this problem, if you think that you may need to insert a few lines of code into your program at a later time, you can insert a few lines of NOP instructions. The CPU will just ignore them, but if we replace them with other instructions later on then new functionality can be added to the program without changing the address of the line in our program.

Apart from always using the same classes of instructions, all instructions are usually three-letter abbreviations for what the instructions actually do. For example, the Move instruction which is used to move binary data from one location to another in your computer's memory will usually be written as

```
Mov.
```

All of these concepts will become much clearer when we cover the chapter on assembly programming.

The Memory Map

After you learn how to speak your computer's language, you are going to have to find things in the computer's memory. You need to know, for example, where to find video memory if you want to display your graphics. You need to know where to go to read the computer's input and how to find sound registers. After you know the language of your computer and you have a memory map, you can do almost anything.

CPU, Bus, and Memory Characteristics

Finally, you will need to know how much disk space, RAM memory, and CPU speed you have available. These characteristics create the confines in which you will have to build your games. The slower the CPU, the slower your programs will run and the harder you will have to work to optimize your code to obtain a fast moving video game.

The size of RAM memory limits the overall size of your program.

The amount, or availability, of the disk space determines the size your program can be, as well as what other features you can use such as saving the highest scores of your players.

BASIC 101

At this time we want to focus completely on learning how to get started programming in BASIC.

You need to familiarize yourself with the environment that you will be working with, shown in the Figure 2.2.

Figure 2.2 Screen shot of the Atari 800 work area.

Notice the blue box with a black border. This constitutes the work area. It is here that you type your commands to the computer, and it is where the computer displays messages. The first thing you see in the work area is the word READY. This is just the Atari's way of letting you know that it's all set and ready to go. Just below this we see a light blue rectangle. This is what is called a cursor. When you press a key on the keyboard, the letter or number you press appears onscreen at the exact location where this cursor is located. It's time to introduce yourself to the computer. Type the following message and see what the computer does. Type: Hello my name is Earl. (You should replace my name with yours.) Your screen will look similar to the one shown in Figure 2.3.

What's this? The computer says you made an error. What did you do wrong? Did you spell your name right? Maybe you should have used a period? Well, the problem is you were simply speaking the wrong language. In order for you to make the computer follow your commands you must learn to talk in a way that the computer can understand.

The first thing we'll do is get the computer to print something. Type the following instructions at the command prompt and press Enter:

```
Print "hello I am an Atari 800"
```

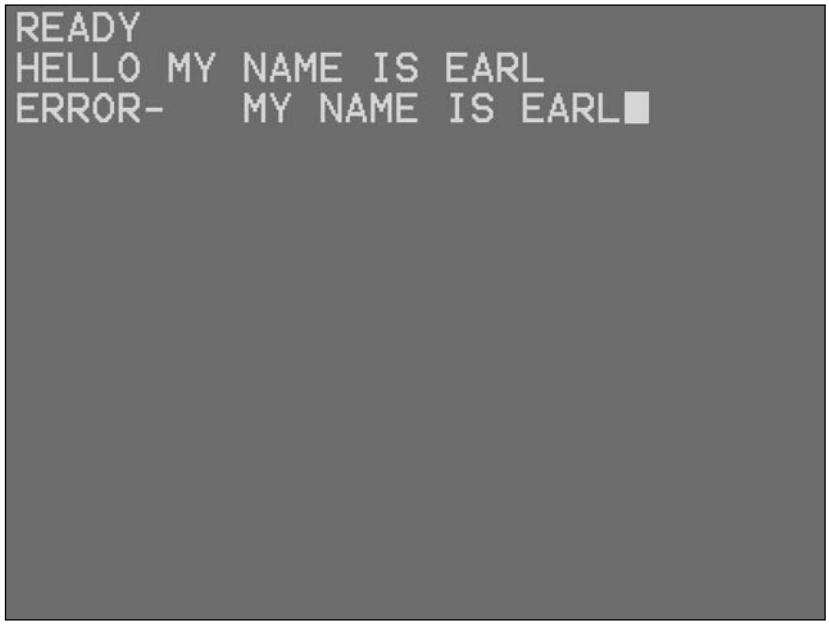


Figure 2.3 Our first try at introducing ourselves to the computer.

That's more like it; this time the computer actually behaved itself and did as we asked it. Generally, as long as you use the proper syntax and methodologies, the computer will follow your instructions. As you will see, 9 times out of 10, when the computer does not react to your code the way you expect it to, it is because you did not enter the correct instructions to accomplish the goals you want. As you learn more about the computer and how it works, you will be able to more powerfully and completely bend the computer to your will.

Figure 2.4 illustrates.

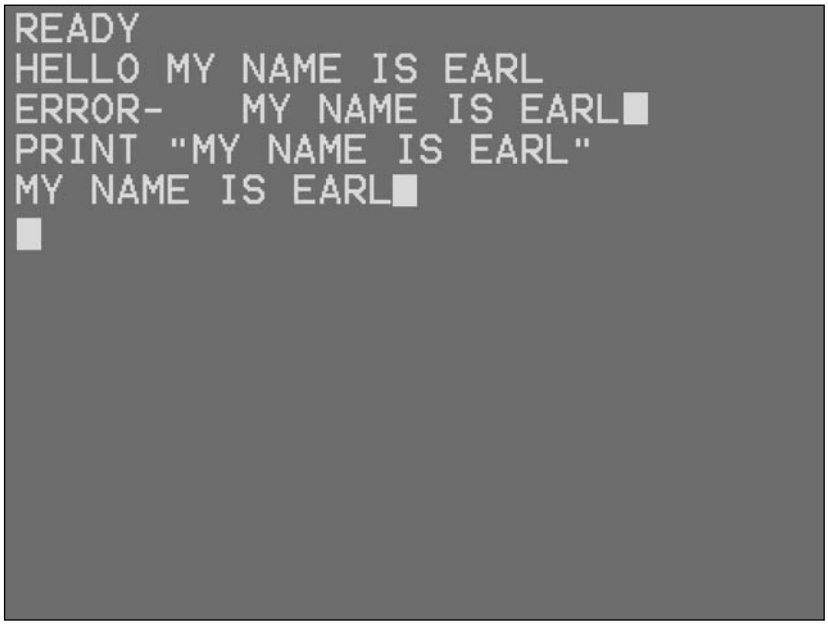


Figure 2.4 Screen shot of the computer obeying our commands.

Principles of BASIC

Your primary school teachers did not try to teach you every single word that you will ever see for the rest of your life. Instead they taught you how to use the alphabet and later how to use a dictionary. By learning these basic principles, you were empowered with the ability to understand words, which allowed you to form sentences that allow you to communicate with almost any person you will ever meet. I want to use the same concept here. I want to teach a few basic principles that you can use to communicate with any retro game machine using the BASIC language.

Using the principles detailed in the following sections, you should be able to create computer programs on any computer system equipped with the BASIC language.

The Variable Principle

The first principle that we will cover is the variable principle. It is important for you to understand that everything that is done with a computer boils down to data manipulation. From its very foundation, a computer is simply a collection of circuitry that is used to store, generate, and manipulate binary numbers. That is all a computer really does. The trick is that skilled programmers can cause the computer to manipulate those binary numbers in such a way as to give the computer the illusion of intelligence and to perform impressive technical feats. As an example, consider Figure 2.5, which is an image of the video display of the PC I am using to write this book. The monitor has numerous pixels, which are tiny dots that cover the entire screen. By changing the colors of these pixels, the monitor presents us with the illusion of an image onscreen. But what controls the color in those pixels? Well, for each and every pixel onscreen, there is a portion of Random Access Memory (RAM) that is referred to as video memory. Video memory is simply a large collection of . . . you guessed it . . . 0s and 1s. Each color that the computer is able to generate is represented by a particular combination of 0s and 1s.

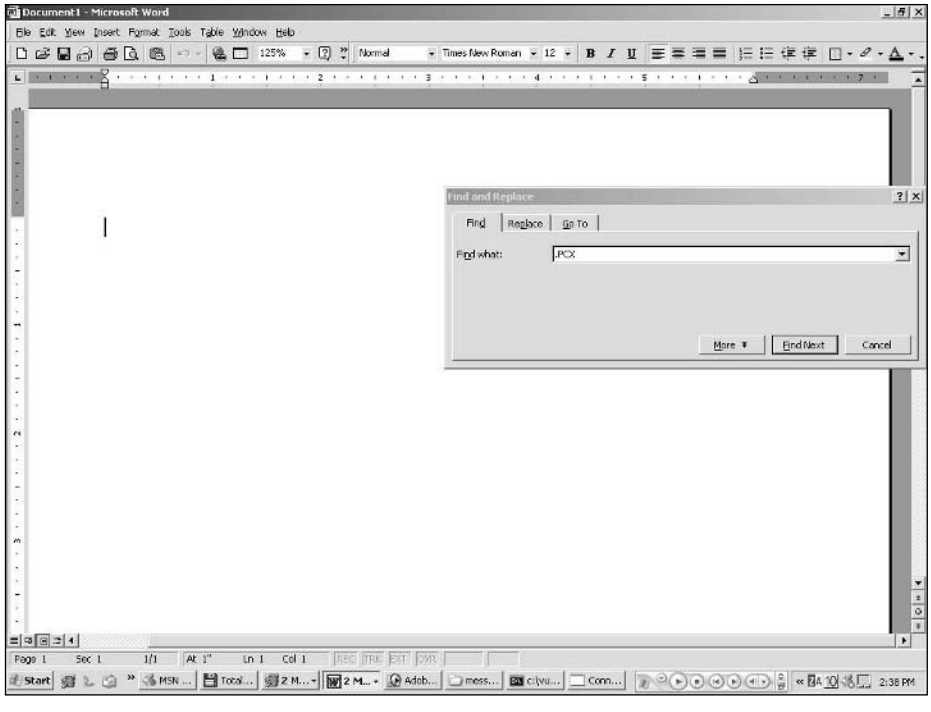


Figure 2.5 Screen shot of my computer taken as I write this book.

In order to generate an image on the monitor, you must set each pixel onscreen to a particular color, and the color of each pixel is determined by what combination of 0s and 1s are in a particular portion of video memory. Then all you have to do to put a picture onscreen is to fill video memory with a particular combination of 0s and 1s.

Anything the computer does, no matter how elaborate it seems, boils down to a simple manipulation of 0s and 1s. Of course, typing millions of 0s and 1s is not the most efficient way to create a computer program. The human brain simply is not designed to process information in exactly the same way as the computer. It is for this reason that computer languages such as BASIC were created. Rather than having to think in binary to communicate with computers we can think in a language that is much closer to English. After we have created our program in this "English-like" language a compiler or interpreter is used to convert our "English like" language into full-fledged binary.

Later we will discuss the raw manipulation of memory but now we will focus on our current principle, which is the variable principle.

The most basic form of data manipulation is based around a concept called a *variable.*

The variable principle states that:

> *"A variable is a container that you can store information in. Generally speaking as long as your computer does not lose power and you do not give it any commands that would cause it to alter the memory location where the variable is stored, or you specifically alter the value kept in the variable; then when you place a value into a variable it will always remain there and remain unchanged."*

Let's see this in action. Type the following instructions into your Atari 800 emulator (you should be running Atari Basic):

```
LET Q = 6
```

Your screen should look like the one shown in Figure 2.6.

You just created a variable called Q and placed a value of 6 inside it. The LET command tells the computer that you have some information you need to store and tells it to create a variable called Q and LET a value of 6 be put inside of that variable. But let us see if the computer really did do what we asked it to. Enter the following command:

```
PRINT Q
```

You have just given the computer the print command. This command has nothing to do with printing to an actual printer but instead it will print out whatever you ask it to on the video screen. In the case above we asked the computer to print out the contents of the variable named Q. Your screen should now look like that in Figure 2.7.

Figure 2.6 Creating a variable for the first time.

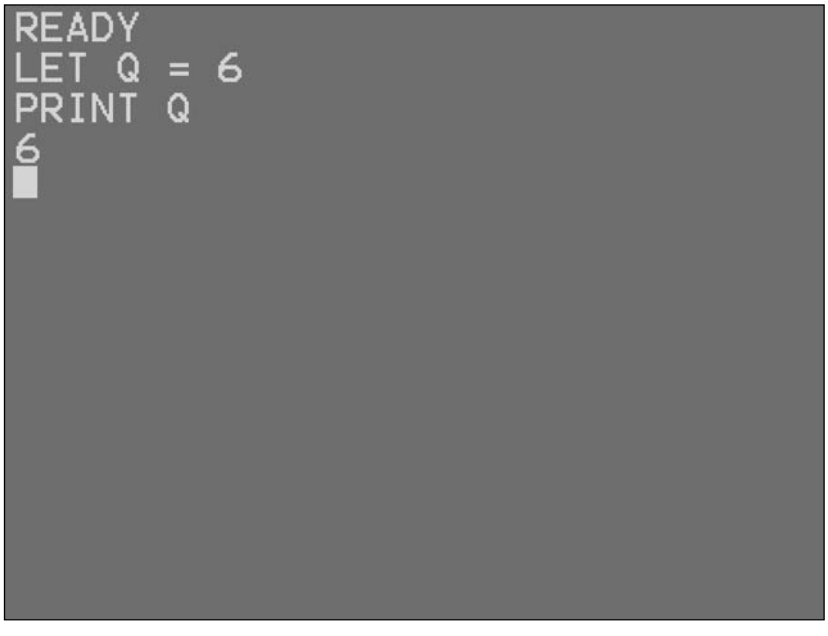


Figure 2.7 Printing a variable for the first time.

As you can see, the value 6 really was stored in the variable Q. Now take a few minutes and play with this concept. Experiment with creating variables and storing information in them, but for now only attempt to store numbers and not letters. You will see why in a few moments.

Earlier I told you not to try to store any letters in the variables you were creating and now you will see why. Enter the following command

```
GR.0
```

All that command will do is clear the screen. Now enter this command:

```
LET G = BN
```

Now the computer tells us that it is ready to attempt to print out the value of this command. Enter:

```
PRINT G
```

Your screen should look something like that shown in Figure 2.8.

But what went wrong? Why didn't the computer store our letters the same way it did the numbers? Well, the answer is that you have to ask the computer to store letters a bit differently than the way you ask it to store numbers. Here is the command to ask the computer to store letters:

```
DIM NAME (X)
```

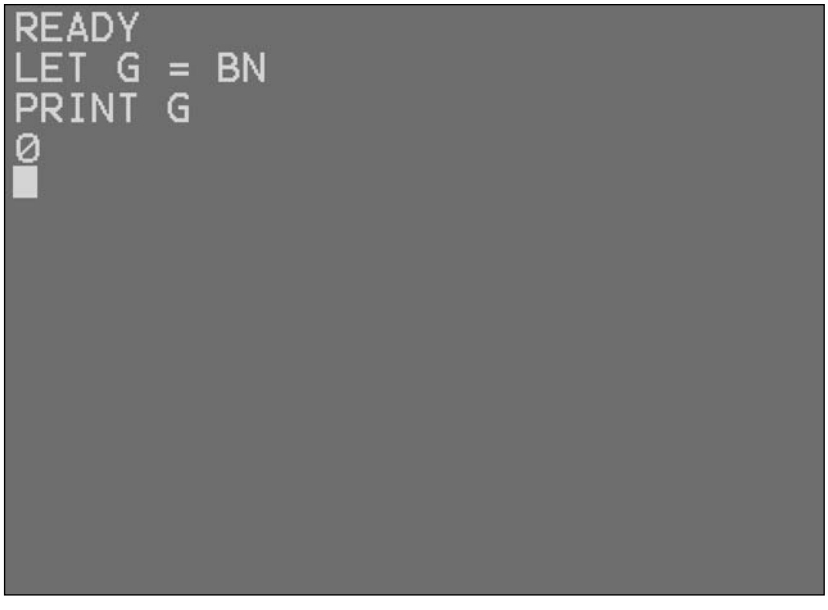


Figure 2.8 A failed attempt at printing a text stored in a generic variable.

In the above command line, DIM is the actual command. It is short for dimension. NAME is the portion of the command that you would use to give your variable a name. The X represents an integer value that tells the computer how many letters your variable should be able to hold. So to put it all together if we want to create a variable called `G$` that can be used to store five letters, we would declare it like this.

```
DIM G$(5)
```

Notice that we have added a dollar sign to the end of the name of our variable. This is to signify that this variable is a string variable. Any variable that you create to hold letters must have this dollar sign at the end of it. Now try this command for yourself; enter the following code:

```
DIM D$(5)
```

Next enter this line of code:

```
LET D$ = "QWERTY"
```

Now finally enter this line of code:

```
PRINT D$
```

Your screen should look similar to the one shown in Figure 2.9.

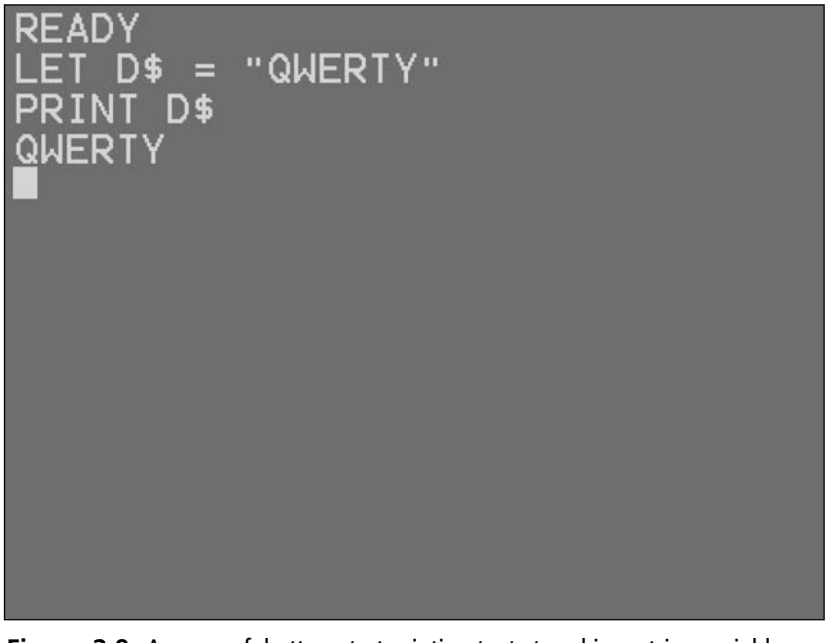


Figure 2.9 A successful attempt at printing text stored in a string variable.

This time the computer has indeed stored our letters in memory. Once again, experiment with creating different string variables. (In programming, the term *string* is used to describe anything that has to do with storing or manipulating letters and words. As a result, variables that hold letters are called string variables.) Experiment with printing them out to be sure that the computer has indeed saved them until you are satisfied that this stuff works.

The Input Principle

Up to this point in order to fill a variable with a value, programmers have specifically placed that value into the variable at design time. Now when we create our program and have it shipped off worldwide for people to use, we will not be there to specifically place values into the variables. When we are actually writing the instructions for our program, we call the time that we spend *design time*. When the computer starts executing our program it is called *runtime*. We need a way to get input from the user at runtime and have this information placed into the variable at runtime.

This is where the input principle comes into effect. The input principle may be stated as follows:

> "*Most flavors of basic provide instructions the computer can use to obtain information from the user at runtime. These instructions are usually able to also store the information they receive into a variable.*"

What time is runtime?

As are so many other terms in programming that seem so complicated when you first hear them, the terms runtime and design time are not very complicated concepts at all. The time you spend sitting down actually typing out the lines of code for your program is called design time. When you enter the `run` command, which will be covered later, and your program code begins to actually execute, this period of time is called runtime. Simple, isn't it? Over the years, I have on many occasions taught people how to use various aspects of the computer. The biggest problem that most people have to overcome is that the machine intimidates them. Many people think they will break the machine, or they think that the computer is so complicated they can never comprehend it. It is important that you do not get caught up with these beliefs. Nothing in life is really hard; there are just things you do not know how to do yet. These computers are more resilient than you may think, and they will not break as easily as you might imagine.

Let's see this principle in action. Enter the following code:

```
DIM L$(5)
INPUT L$
```

Your screen should look something like the one shown in Figure 2.10.

What you just did was create a string variable, as we have done so many times before. You gave that variable a length of 5 letters. What's new here is the `INPUT` statement. This statement tells the computer, "I need you to get some information from the user via the keyboard and place that information into a variable for me." As you can see from this example, we have asked the computer to store the information it gets from the user into a string variable we just created called `L$`.

The format for this command is as follows:

```
INPUT  (string variable)
```

`INPUT` is the actual command being given to the computer and (string variable) is where we enter the name of the variable we want the data to be stored in.

Type in five letters and press Enter. The computer should display READY as a sign that it has accepted the input and is ready to move on. Now as we have done so many times before, let's ask the computer to print out the contents of our variable. Enter:

```
PRINT L$
```

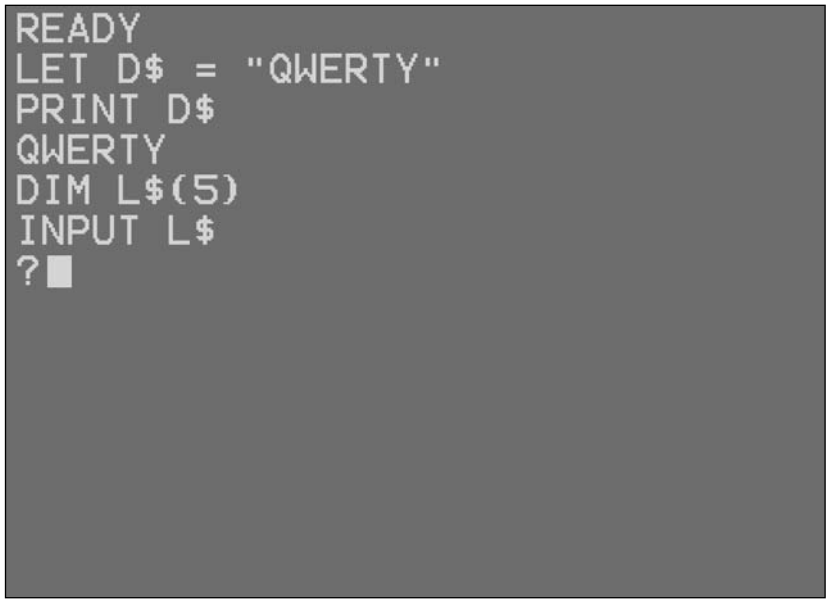


Figure 2.10 Example of using the INPUT command.

As you can see, the computer prints out the contents of the variable `L$`, which is the data that you just typed in.

Now experiment with this concept. Create as many different variables as you want, then use the `INPUT` command to place values into the variables you have created. Try creating string variables and numeric variables just for fun. Experiment with this concept until you are sure you completely understand how it works, and then move on.

Your computer's memory is probably pretty cluttered and unorganized because you have been creating so many variables and filling themn with random data. In order to clean things up a bit we will use the `NEW` command. What this command does is erase any variables or programs that may be stored in memory. In essence, your computer's memory will be just as empty as when you first turned it on. It is important for us to note that this command will not clear the screen as you might expect. In order to clear the screen, we will use the following command: `GR.0`. Now let's give these commands a try.

Enter the following code

```
NEW
GR.0
```

At this point your screen should look just like when you first turned it on, as shown in Figure 2.11.

Figure 2.11 The Atari display after clearing memory and the screen.

The Listing Principle

So far we have been giving the computer direct commands. We type the command and the computer follows our instructions immediately. This may be fine for learning simple concepts, but we have to find a way to store all the commands that we need to execute. In order to do this, we have to create a program listing. A program listing, as its name implies, is a list of all the commands that are used to make up our program. Now in order to make a program listing we have to enter our commands into the computer in the following format:

```
(Line Number1) (Command1)
(Line Number2) (Command2)
(Line Number3) (Command3)
(Line Number4) (Command4)
(Line Number5) (Command5)
```

The above listing would be used to store a program with four commands. Notice that before each command is a line number. All commands are executed in order from the one with the lowest line number to the highest. To make things clear, let us create our first program listing.

Enter the following code:

```
10 Print "Please enter your age"
20 let age = 0
30 input age
40 Print "your age is:"
50 Print age
```

As you have probably noticed, the commands that you entered did not execute immediately as earlier ones did. In fact, nothing seemed to happen. That is because all of the action so far has taken place behind the scenes. The commands that you entered were stored in the computer's memory. There are two things that we can do with this program now that it is in memory. First, we can have the computer display the program so that we can be sure we typed everything in correctly.

Now clear the screen using the command `GR.0`. This isn't strictly necessary, but if you do it, the screen does not become too cluttered. Now enter the following command:

```
LIST
```

As you can see, the computer just listed the entire program that you just wrote onscreen. It is at this time that we could make changes if something was wrong or if we just wanted to make some adjustments. Let's make a change to our program so that you can see how easy it is to do so. To make a change to the program , simply enter the line number for the

command you want to change and then enter the commands for that line the way you want them to work. To see this concept in action, enter the following code:

```
10 Print "Please enter your age now"
```

Again enter the `LIST` command. You can see that line 10 has been changed. The second thing we can do is have the computer execute the program. To do so, enter the following command:

```
RUN
```

Be sure to follow the onscreen instructions.

The computer runs each of your commands in order, first asking you to enter your age, creating a variable to hold your age, giving you the opportunity to enter your age, and then finally printing your age.

Congratulations. You are now an entry-level programmer.

Here is the listing principle:

> *"The computer must be given the precise order in which to execute commands. We give the computer this order by placing line numbers before each command. The computer executes commands starting from commands with the lowest number up to the highest."*

Short Test

Before we move on to the rest of the principles, you should take some time and apply them without any guidance.

1. Create a program that will ask the user for her first name, then her last name, then her age, and finally print out all of this information.
2. Make up your own program ideas from scratch where you get input from the user and then display this information onscreen.

Please do not skip these exercises. They will help you to cement the concepts covered so far in your mind. When you are sure that you have mastered these concepts, then you are ready to move on. Answers to the exercises can be found on the book's Web site.

The Math Principle

This principle states that:

> *"Every version of BASIC has the ability to perform basic and complex calculations, which can be used to solve math problems and manipulate data."*

Any version of BASIC that you use is capable of functioning as a giant calculator. You are familiar with standard mathematical expressions such as +, -, ×, –, and ÷. Now on our

BASIC calculator most of the expressions that you use remain the same with a few exceptions. Namely, we represent the multiplication sign with a * and we represent the division sign with a /. So if we wanted to know what 5×6 is, we would type the following command:

```
PRINT 5*6
```

This command tells the computer to print the results of 5×6.

The following code would give us the result of 100 divided by 20

```
PRINT 100/20
```

Now it is important for you to understand that the math principle works hand in hand with the variable principle. To see what I mean, let's take a look at the following example.

Enter the following code:

```
NEW
LET A = 5*6
PRINT   A
```

Your screen should look something like the one shown in Figure 2.12.

Figure 2.12 Example of storing the results of a mathematical operation into a variable.

As you can see, we are able to store the results of mathematical operations inside of variables. We can take this a step further by incorporating variables into the mathematical expression itself. To demonstrate this, enter the following code:

```
PRINT A/5
```

Your screen should now look like the one shown in Figure 2.13.

As you can see, we were able to take the variable we created earlier, divide it by five, and have the computer print the results. The use of the math principle to manipulate stored data is perhaps one of the most powerful features of any computer system.

Take some time to play with this concept by dividing, adding, and multiplying numbers and storing them into variables and printing them until you are sure about how this stuff works.

BASIC has what are called mathematical functions that give us even more power when dealing with mathematical problems. We will learn these functions later in the book as we study each machine in depth.

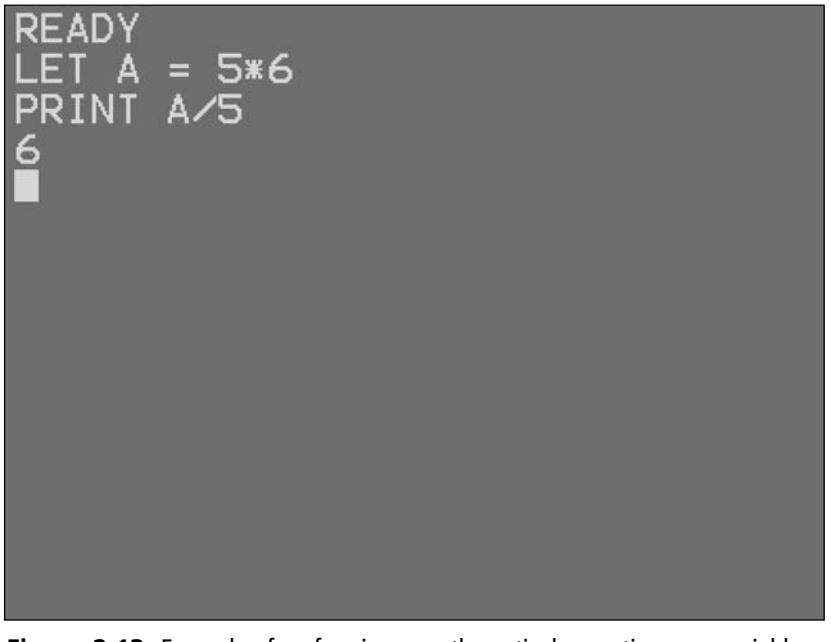


Figure 2.13 Example of performing a mathematical operation on a variable.

The Logic Principle

> *"The logic principle states that all decision making on the computer is based on binary true or false statements."*

All that this means is that if the computer is making a decision, it will say, "If this is true and this is true, then I will do this." To see this in action, write a small program that uses the `IF` command. The `IF` command uses a format like this:

```
IF  variable/value (operator) variable/value THEN
commands
END IF
```

`IF` is the command that lets the computer know that we need it to make a decision. `Variable/value` can be variables, numbers, or letters that you want to compare. `Operator` will be a mathematical symbol, such as the greater than(>), less than(<), or equal (=)signs. You will usually be able to find out which symbol to use by reading your `if` command as a sentence. So for example, if the variable PL holds the value of a player's health points and we need to make sure that the game ends when the player's health points reach zero we would say if PL is equal to 0 then the game is over. From our sentence, we see that we will be using the equal sign in our statement. When we write our game code this `if` statement would look something like this:

```
100 IF PL = 0 THEN
```

`THEN` is the next command used. This command cannot be used alone in and of itself and must be used along with the `IF` command. It tells the computer that if the results of the `IF` operation are true then the computer should execute the commands that follow.

The next command you have to be aware of is the `END IF` command. When we use the `IF` command, there are usually a precise number of lines of code that you want executed when the parameters of your `IF` command are true. In order for the computer to know exactly which lines of code you want executed, we normally enclose them between an `IF` command and an `END IF` command as seen in the example below.

It should be noted that not all retro machines support the `END IF` command. In these cases your entire `IF THEN` command has to be written on a single line.

```
IF A > B THEN PRINT "A IS BIGGER THAN B"
100 IF PL = 0 THEN
commands
commands
commands
200 END IF
```

This way the computer knows that if PL really is equal to 0, it should execute all of the lines between the IF and the END IF command. On some occasions, you may only have one command that you want executed as a result of the IF command; in such cases, the END IF command is not really needed and you can use the following format:

```
IF variable/value (operator) variable/value THEN commands
```

Try the following example:

Create two variables called A and B with the following code:

```
10 LET A = 5
20 LET B = 10
```

Now it's time for the IF command:

```
30 IF A < B THEN PRINT "B is bigger than A"
```

Enter the RUN command to execute the program.

As you can see the computer prints:

B is bigger than A

Because A is less than B. Now let's try changing the value of A to 20:

```
10 LET A = 20
```

Now run the program again. This time nothing happens. This is because A is no longer less than B, so the computer does not execute the commands after the THEN command.

The Screen Mode Principle

The screen mode principle states that:

"A given computer system may have a screen mode of one of two types: graphics screen modes or text screen modes. What makes each graphics screen mode different from other graphics screen modes is the amount of color that can be displayed, the resolution of the screen, and whether the screen displays text or graphics elements. The element that separates text modes from each other is the amount of colors that can be displayed, the number of characters that can fit horizontally across the screen, and the number of characters that can fit vertically onscreen."

When you want to present information to the user in the form of text, as you have done so far in this chapter, you need the computer to be in text mode. When you want to present information to the user in the form of graphics, as you shall learn to do shortly, you need the computer to be in graphics mode.

How do we change the graphics mode of the computer? Remember when you learned how to clear the screen? You used the command `GR.0`. The `GR.` command is actually an abbreviation for the command `GRAPHICS`. This command is used to set the graphics mode of the computer. When your computer starts up, it starts in text mode `0`. When you give a command to the computer to change the screen mode, everything that is onscreen is automatically erased even if you are setting the computer to a screen mode that it is already in. So when the computer starts up in text mode `0` and we give the graphics command to set the screen mode to `0`, the computer simply clears the screen.

Before you experiment with screen modes, you should know that there are also mixed screen modes that display text on parts of the screen and graphics over the rest of the screen. Graphics mode `1` is such a screen mode. Let us enter this screen mode now.

Type the following code:

```
GR.3
```

Your screen should look like the one shown in Figure 2.14.

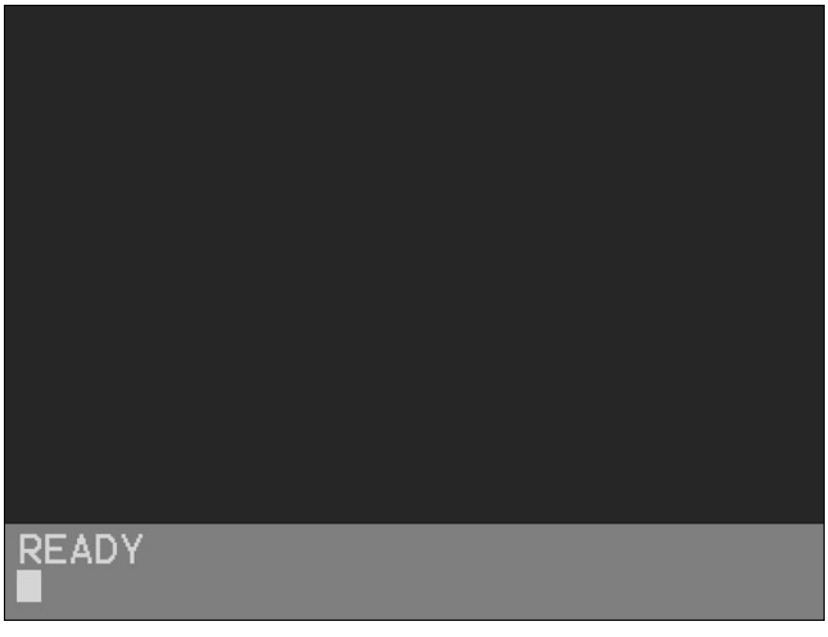


Figure 2.14 Screen shot of Graphics mode 3.

The blue area at the bottom of the screen is the text area. This is where you can type commands and get information from the computer. The top of this screen is the graphics area where we are able to draw pictures.

In the next section, you will learn how to create pictures under the graphics principle.

The Graphics Principle

The graphics principle states that:

> *"Each version of BASIC has built-in functions to facilitate the production of generic graphics, such as points, lines, and circles. Using these graphics elements, you can create complex or simple computer graphics."*

If you have been following along with the instructions in this chapter, you should be in graphics mode 3, which means that you have four text lines at the bottom of the screen, and the top of your screen is in graphics mode. If this is not the case, enter the following line of code:

```
Gr.3
```

The Color Command

There are color registers that can hold one of 15 different values. Each value represents a different color. When you execute a graphics command, the command draws a color based on the value that is in the selected color register. In the current graphics mode, you have access to four colors at a time. Color register 3 has a default value of blue, so if you want to use the color blue for drawing you would enter the following command.

```
COLOR 3
```

The command we have just used is called the `COLOR` command and is used to tell the Atari which color register it should use to draw with. (In our current graphics mode, we can draw using color registers 0–3.)

More information about using colors on the Atari is covered later on in Chapter 7, "Hacking the Video Buffer."

You would not see any immediate changes onscreen after executing the `COLOR` command, but if you use any graphics command now, the command will draw in the color blue. Let's test this idea and introduce the next graphics command, which is the `PLOT` command.

The PLOT Command

Before we can discuss the use of the `PLOT` command, we have to understand the way that the graphics screen is organized. Take a look at Figure 2.15 to see the way the graphics screen is organized.

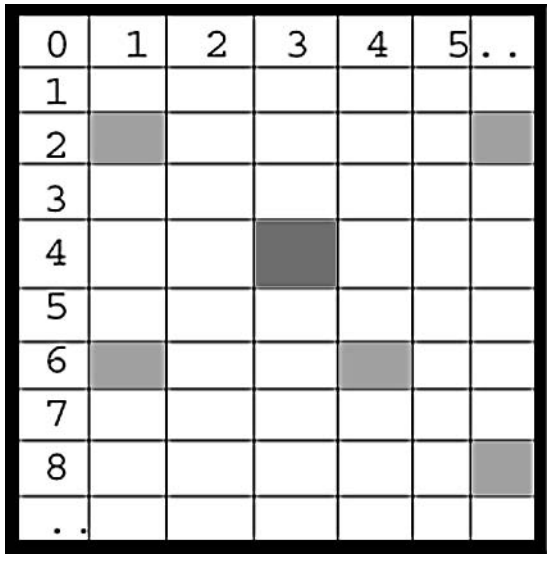


Figure 2.15 Diagram of the graphics screen layout.

As you can see, the screen is organized like a grid and we have a coordinate system that we can use to find any point onscreen. In our coordinate system the top-left corner represents the origin. If we give the computer the coordinates 0,0, the computer will focus its attention on the top-left corner of the screen. For every amount that you increment the x value of these coordinates, the computer will focus its attention one more pixel to the right. For every amount that you increment the y value of your coordinate system, the computer will focus its attention one pixel lower.

The `PLOT` command uses this coordinate system. When you call this command and provide a necessary pair of coordinates, it will plot a dot onscreen at the location of the coordinates that you give to this command. If the top-left corner of the screen is represented by the coordinates 0,0 and if we use the command

```
PLOT  0,0
```

our computer should plot a point in the top-left corner of the screen. Enter the command

```
PLOT 0,0
```

and let's see what happens. Your screen should look something like the one shown in Figure 2.16.

As we predicted, the computer plotted a pixel in the top-left corner of the screen. Now experiment with the plot command and try plotting points at different coordinates

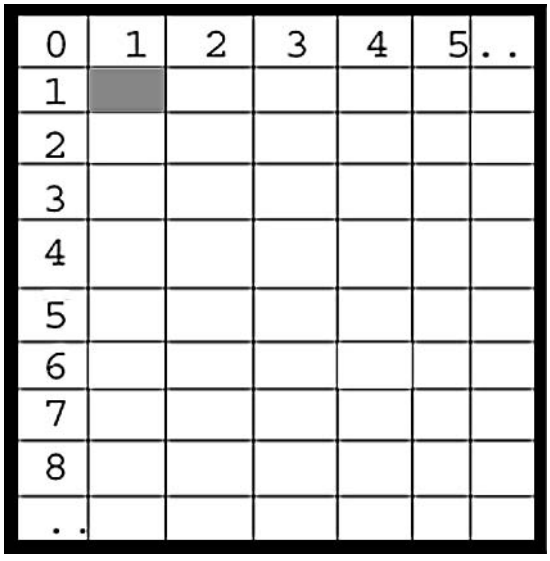


Figure 2.16 Here we use the `plot` command to place a dot in the top-left corner of the screen.

onscreen. Continue until you are sure that you fully understand the concept of how the coordinate system works. Keep in mind that the largest number that you can use for your x and y value will change depending on the video mode you are using. If you use anything larger, you will get an error message. Like most of the errors you will make when programming, this one is easily corrected. Just retype your commands using values that are within the limits of what the computer can print.

When you are done experimenting, enter the following command to clear the screen:

```
GR.3
```

The DRAWTO Command

When you plot a point using the `PLOT` command, the computer remembers the coordinates that were used. The `DRAWTO` command takes advantage of this. If you plot a pixel in the top-left corner of the screen and want to plot a line from there to the coordinate 7,7, you would have to give the computer the following command: `DRAWTO 7,7`. Let's give this a try by entering the following code.

```
GR.3
COLOR 3
PLOT 0,0
DRAWTO 7,7
```

Your screen should look like the one shown in Figure 2.17.

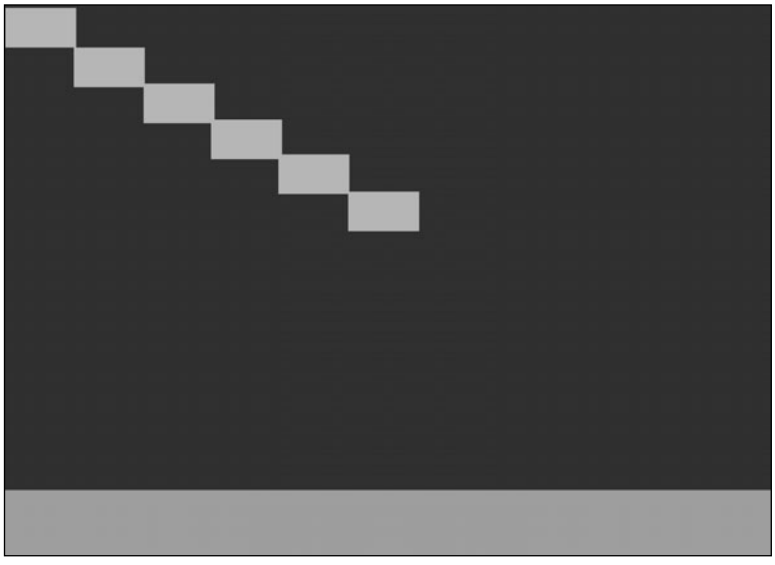


Figure 2.17 In this example we use the DRAWTO command to draw lines.

Now try the following command.

```
DRAWTO 7,0
```

Your screen should now look like the one shown in Figure 2.18.

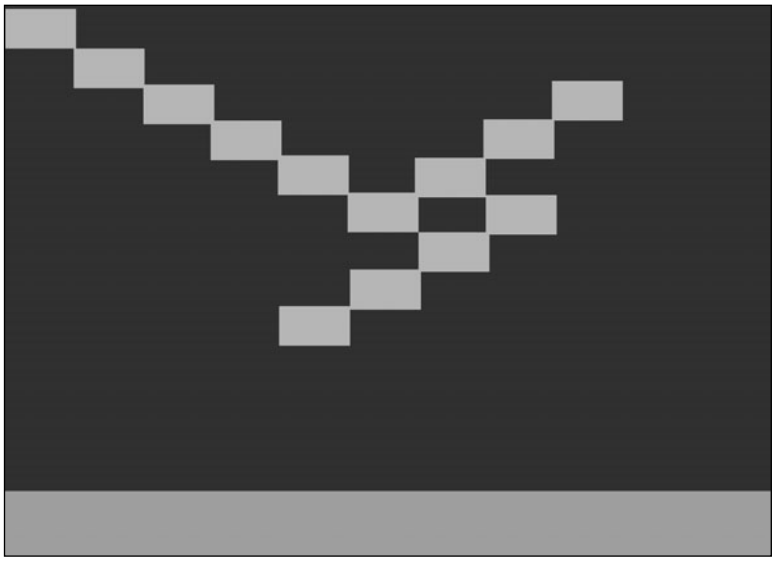


Figure 2.18 You can use the DRAWTO command as often as you wish to fill the screen with lines.

Experiment with this command a bit by entering the DRAWTO command a few times with different coordinates until you get the hang of it.

Now you have the hang of the graphics principle. Some versions of BASIC have more functions to draw circles and other graphics, but we will cover those when we are dealing with each machine in the following chapters.

The Branch Principle

This principle states:

> "*The ability of a computer to produce the appearance of intelligence is based on both the ability to make decisions and the ability to alter program flow as a result of such decisions. Branch commands are what enable the computer to change the program flow.*"

Earlier we discussed the Logic Principle and saw how we could cause the computer to execute a number of commands based on the results of an IF statement. While this does give us the ability to make our program slightly "intelligent," it is still greatly limited. Your program still starts from the first line of code and progresses toward the last line of code, after which it stops. It has one continuous program flow that never changes.

Now consider the exercise that you did under the previous principle. What if one person used your program and then someone else wanted to use it? The way your program is written now, after the first person used it, the next person would have to use the RUN command and rerun your program in order to use it. Your program will look a lot more professional if after each person uses it, the program would then ask if anyone else wanted to use it. If a user types Yes, the program will start over from the first line of code, and if the user enters No, the program will end.

This is a very simple example of how we can use branch commands to alter program flow and add another level of intelligence and interaction to your programs. The branch command that is most common is called GOTO. This command is used to tell the computer to go to a particular line. The format for this command looks like this.

```
GOTO line number
```

This command is very straightforward. Line number is the line number that you want the computer to jump to. If you use 5 as your line number then the computer will jump to the 5th line of your program and execute the instructions on that line before moving on to executing the remaining lines of your program. Let's see this in action. Type NEW and press enter to clear your computer's memory. Now enter the following program.

```
10 A = 100
20 B = RND(10)
30 PRINT  A " x " B " = " A*B
```

Make a note of the line number for the first line of code in your program. If you ever write a very long program and you are not sure what the line number is, use the `LIST` command to print out a listing of your program onscreen. Now add lines of code to your program to tell the user to enter yes to use your program again and no to exit your program. Your program should look something like this.

```
40 PRINT "WOULD YOU LIKE TO RUN THE PROGRAM AGAIN"
50 INPUT L$
```

Note the $ after the variable which indicates that this is a string variable.

Now add a new line to your program and on that line enter the following command:

```
If  (your variable) =  "yes" then GOTO first line number
```

Remember to replace your variable with the name of the string variable that you used to hold your yes or no answer. Substitute `first line number` with the line number of the first line in your program. Now run your program. If you did everything correctly, your program should run just like it did before, only this time rather than just ending, it will give the user a chance to start the whole program over or just let the program end as it did before. Your program should look like this.

```
10 A = 100
20 B = RND(10)
30 PRINT  A " x " B " = " A*B
40 PRINT "WOULD YOU LIKE TO RUNE THE PROGRAM AGAIN"
50 INPUT L$
60 IF L$ = "YES" THEN GOTO 10
70 PRINT "GOODBYE"
```

The Looping Principle

The Looping Principle states that:

> "*When a given task must be repeated again and again, it is best to incorporate that task into an infinite or a finite loop.*"

Let's take a look at an example of where we might need to use this principle. Suppose we have a game with a space ship in the center of the screen. When the player presses the button on his controller, his ship unleashes its "super-mega bad-mega big laser-cooled plasma cannon." In order to add to this effect, we may want our player's space ship to jump back a short distance and then move back into its original position in order to simulate a gun's recoil. Now suppose that we have two variables that hold the ship's position called *PX* and *PY*. These hold the ship's *x* and *y* values. So far everything is set up just as it was when we covered the input principle. Now let's throw in a twist. We will add a new

variable called *PR*, which initially will have a value of 0. From now on, when we plot the position of our players' ships, rather than using the following code as we did before

```
PLOT PX,PY
```

we will use this code:

```
PLOT PX,PY + PR
```

As you are now adding the value of *PR* to *PY*, so far this will have no real effect on your program because *PR* is set to 0. Now think about what would happen if *PR* were equal to 2 or 5 or 10. Because *PY* controls the vertical position of the ship onscreen and the larger *PY* is, the farther down on the screen our player's ship will appear, increasing the value of *PR* will cause the ship to be displayed farther down on the screen. Likewise, if you were to subtract *PR* from *PY*, your player's ship would move farther up on the screen. Assuming that the ship is facing upwards when it fire its cannons and you want to simulate the ship moving backwards and then back to its original position, all that you have to do is manipulate the value of *PR*. First increment its value so that your ship moves downward, and then slowly decrease the value of *PR* to move the ship back into its original position. It is the manipulation of *PR* that requires you to use the loop principle.

When you are creating a loop, you will generally use the commands called FOR and NEXT. The format for these commands is as follows:

```
100 FOR I  = 1 TO 10
110 Commands
120 NEXT I
```

The command line with the FOR command and the NEXT command form the beginning and end of our loop. Any number of commands can be placed between the FOR and NEXT commands. In the preceding example, the computer executes the given commands 10 times. This is because we have given the variable *I* a range of 1 to 10.

The first time the program reaches line 100, it returns a value of 1. The program then goes to line 110 and executes the commands on that line. When the program moves down to line 120, it jumps back to line 100 and the value of *I* is increased to 2. The computer moves to line 110, executes the command, moves down to line 120 where once again the NEXT command causes the program to jump back up to line 100 where *I* is increased to 3. This process is repeated again and again until the value of *I* reaches 10.

It should also be noted that *I*, like any other variable, can be used in our program. We noticed that each time the loop ran, the value of *I* increased. What if the PLOT command that we used earlier were placed between our FOR and NEXT commands? What if each time our loop runs, the value of *PR* is set to the value of *I*. Each loop would bring our player's ship 10 pixels down.

That code would look something like this:

```
100 if joy code
120 COLOR 0
130 PLOT PX,PY+PR
140 FOR I  = 1 TO 10
150 COLOR 0
160 PLOT PX,PY+PR
170 PR = I
180 PLOT PX,PY +PR
190 NEXT I
200 END IF
```

First, we enclose all of our code in an `IF` command that will only allow it to be activated when the player pushes the button on her joystick. Next, we activate our loop, which causes the player's ship to move down 10 pixels.

So far you can cause your ship to jump back 10 spaces to simulate the first portion of the recoil; now we need to have the ship move back to its original position.

You can do the same thing by creating another loop that is very like the last one, except this time you will set *I* to range from 10 to 1, which causes the program to count backwards from 10 to 1 and places the ship back in its original position. The complete list of commands that can be used to do this simple recoil animation is listed below.

```
if joy code
120 COLOR 0
130 PLOT PX,PY+PR
140 FOR I  = 1 TO 10
150 COLOR 0
160 PLOT PX,PY+PR
170 PR = I
180 PLOT PX,PY +PR
190 NEXT ISX
200 FOR I  = 10 TO 1
210 COLOR 0
220 PLOT PX,PY+PR
230 PR = I
240 PLOT PX,PY +PR
250 NEXT ISX
260 END IF
```

This page intentionally left blank

CHAPTER 3

The Early History of Video Games

History is the witness that testifies to the passing of time; it illumines reality, vitalizes memory, provides guidance in daily life, and brings us tidings of antiquity.

Cicero (106 BC–43 BC), *Pro Publio Sestio*

The task has been laid before me to be your guide as we travel down the mythical halls of legend and magic and sheer genius that together form the history of the video game industry. In this manuscript lies the secrets of the ages—the deep knowledge and science that was born of a will to create machines that did things never before seen and thought to be impossible. The history of the gaming industry is filled with drama and suspense, and from its inception, it was filled with controversy. But what great story isn't?

Build It and They Will Come!

Remember my telling you that the gaming industry is full of controversy? Well, the very first controversy is the question of who originally came up with the concept of building video games. The honor of being the first person to propose the idea of the video game goes to a 29-year-old TV engineer named Ralph Baer, pictured in Figure 3.1. In 1951, Sam Lackoff, who was the chief engineer at Loral, gave Mr. Baer and his collogue, Leo Beiser, an order: "Build the best TV set in the world!"

Figure 3.1 Ralph Baer.

That simple phrase fueled the start of the entire video game industry. Well, almost. . . . You see, during the development

of this ultimate television set, Ralph Baer proposed the idea of integrating video games with television sets. Unfortunately, this initiative was not put into effect at the time because his boss rejected the idea. In fact, the fruit of their labor, the television that was to be the "best TV in the world," never even went into production. This experience was not in vain, however, for the seed had been planted, and this seed grew into quite a tree indeed.

Noughts and Crosses

The first graphical computer game was actually built by a gentleman named A. S. Douglas in 1952 as a part of his doctorial thesis on "human-computer interaction." To illustrate his point, Douglas developed a tic-tac-toe–type game called Noughts and Crosses on an EDSAC vacuum tube computer. Figure 3.2 shows a simulated screen shot of Noughts and Crosses.

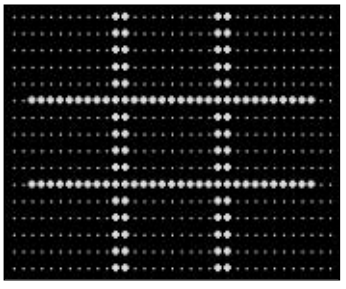


Figure 3.2 Simulated screen shot of Noughts and Crosses.

note

While the game Spacewar! is regarded by many as the first real graphical computer game, it should be noted that this game actually precedes Spacewar! by 10 years.

This machine stored programs and data on 32 mercury delay lines (or *long tanks*), which could hold 1,024 words and represented the display data as a matrix of 35×16 dots, which were displayed via one of three cathode ray tubes. The game allowed the player to choose who would go first, the player or the computer. The computer would then use special algorithms to attempt to win the game whenever possible.

Willy Higinbotham's Game

The next evolution in the gaming industry toke place in 1958. Every autumn Brookhaven National Laboratory, a U.S. nuclear research lab in Upton, New York, held a series of open houses to show people how "safe" working in a nuclear lab could be. William A. Higinbotham (Figure 3.3), a physicist working for Brookhaven, noticed that visitors to the lab seemed to be bored with displays made of simple photographs and static equipment.

Figure 3.3 Willy Higinbotham.

Determined to make visits to the lab a bit more interesting, he came up with the idea of creating a video game. At that time, the analog computers in use at Brookhaven were very good at doing two things: cryptography and plotting missile trajectories. Correctly guessing that nobody would find it fun to sit down and watch a computer deciphering cryptographic keys, Mr. Higinbotham opted to leverage the strength of an analog computer's ability to plot missile trajectories. His idea was to use an analog Donner computer to calculate the path of an imaginary ball, which could be displayed on an oscilloscope. The game had two controllers that were outfitted with a dial and a button as seen in Figures 3.4 and 3.5.

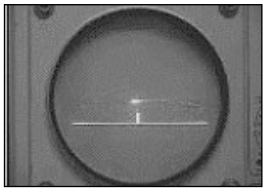


Figure 3.4 Picture of Willy Higinbotham's game in action.

Figure 3.5 Picture of game controls used for Mr. Higinbotham's game.

As the ball bounces off a horizontal line at the bottom of the screen (this line, of course, represented the ground), a player could use the dial on his controller to adjust the angle of the ball's trajectory (as long as it was his turn). Then he could press the button on his controller to hit the ball back over to his opponent's side of the screen. (There was also a reset button to put the ball on either side of the screen and make it ready to go back into play.) In the center, on the "ground," stood a small vertical line that represented the net. If a player hit the ball into the net, she lost the game. It was a brilliant idea, and after three weeks, Willy Higinbotham and technical specialist Robert V. Dvorak had built the first ever video game system, which was dubbed Tennis for Two, as seen in Figure 3.6.

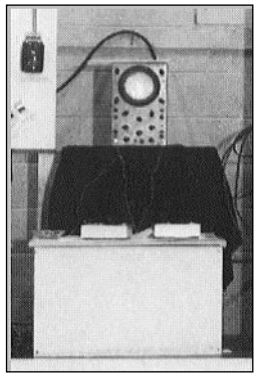


Figure 3.6 Picture of the first ever video game system.

In October 1958 in the Brookhaven National Laboratories gymnasium, Tennis for Two went on display for the first time and was a major hit. Despite the fact that this game was played on a five-inch oscilloscope screen that only had one color (phosphor green) and no score was tabulated, people still stood in line for

hours to play it. One year later Tennis for Two, also known as *tennis programming*, was back and bigger and better than ever for the 1959 open house. This time around, its features included the ability to adjust gravity to simulate playing tennis on other planets and a much bigger screen.

Unfortunately, this groundbreaking machine was short lived. In the 1950s, parts were very expensive and hard to come by. As a result, they often reused the same parts over and over, building one machine today, then dismantling it to harvest parts, which they would use to build another machine tomorrow. Such was the fate of the Tennis for Two video game system. Even more tragic is the fact that Higinbotham never made any attempt to patent or copyright his invention. Apparently, he thought the idea was so obvious and simplistic that it was not worth pursuing, a mistake that he would regret later.

Higinbotham was in no way aware of the dreaming of Ralph Baer or the works of A.S. Douglas, but to this day there is great controversy concerning which of these men deserves the right to be called the Father of the Video Game Industry. The concept of using a home TV set for playing games was not even in the remotest parts of either Higinbotham's or Douglas' minds. Ralph Baer is the first man to be credited with dreaming up the idea of using a standard home TV set to play a video game, A.S. Douglas was the first man to actually build an analog computer game, and Higinbotham was the first person to actually build an analog video game.

At this point, I should probably note that there is a technical difference between video games and computer games. A *video game* is a game that is designed to display games using a *Raster video display*, a.k.a., a TV. The video games designed by Ralph Baer between 1966 and 1968 and the commercial games based on his inventions in the seventies were not programmable in the sense of current video games but rather were hard-wired to perform one specific task: play the game that they were made to play. In the early days, these video games were vastly different from computer games, which could only be played on large mainframe computers that were the size of a small city. (Well, not really as big as a city, just a small house, but they were huge and cost millions of dollars to make.)

In a Land Far, Far Away . . .

Previous stages of the evolution of the electronic game were pretty straightforward. Ralph Baer was building a television set and said, "Hey you know what would be really cool? Let's integrate a video game with a TV set." A.S. Douglas was trying to obtain his doctorate degree, and Willy Higinbotham wanted to entertain guests in the laboratory he worked for. In each case, there was a simple cause and effect, of which only Baer's work led to a major development in the electronic gaming industry.

The next evolution, however, was a bit more complex, in part because technology was improving, but also because there were actually a number of different causes that all came together in one big gigantic melting pot. When the molten contents of this pot were poured into a mold and left to harden, the result was the groundbreaking video game called Spacewar!.

Spacewar!

Amazingly, if we want to identify the culprit who was the ultimate root and cause of the Spacewar! game, we would have to go all the way back to the 1920s. It was on a hot summer night in Washington D.C. that Edward Elmers Smith (better known as E.E., or Doc, Smith), his wife, and a few close friends were marinating in the slow heat of the Smith apartment. During the course of idle conversation, Smith mentioned how wonderful it would be if they were in the absolute zero temperatures of space. Needless to say, it was a fascinating topic, and the group spent the rest of the evening engaged in a conversation dominated by fantasies of outer space. Finally, Mrs. Garby convinced Mr. Smith that he should write a book about adventures in outer space.

Even though he initially refused the idea saying, "Got to have a love story to write a book, and I don't see how a love story would fit in with that kind of stuff," nevertheless, she was able to convince him to change his mind by agreeing to write the "love stuff" herself if he would write the "wild stuff." The rest is history. "The Skylark of Space," "Gray Lensman," and all the other works of "Doc" Smith became seeds that would take some 40 years to grow.

Long, Long Ago

Cut scene: Fade to a room filled with members of the world's first hardcore "hackers'" club, dubbed the Tech Model Railroad Club (TMRC). It is here that the seeds of E.E. Smith's space stories take root and begin to blossom into something truly beautiful. You see, the Tech Model Railroad Club members were major fans, sorry, scratch that, fanatics of the works of "Doc" Smith. In fact, these guys were also major movie lovers and for the life of them they could not see why there were not any movies based on the novels of "Doc" Smith. As a result, they spent hours every day daydreaming and fantasizing about heavy special effect-ridden movie sequences based on the works of the author they held so dear. Boy, oh, boy, what fantastic dreams they must have had! You see, "Doc" Smith wrote about major intergalactic starships and tremendous battles in space. This man was the grandfather of the entire Sci-Fi genre!

note

When most people hear the word hacker, they tend to think about a pimply-faced teenager defacing popular Web sites or some dark malevolent figure hacking into banks and moving money from one bank account to another. For many people, it would seem very odd to use the word "hacker" way back in 1961 to describe the Tech Model Railroad Club members. After all, the Internet wasn't even invented yet.

The truth is the current image of a hacker is really a modern invention. Oddly enough, the original definition of a hacker was "one who used an ax to carve furniture." In the computer world, the word hacker was used to describe anyone who was skilled with computers and had the ability to push a computer beyond what others thought possible. This is the noblest definition of the word "hacker." A word that was almost a badge of honor has been defiled to the point that it is now considered almost shameful. Today the words *hacker* and *criminal* can almost be used interchangeably.

Today dictionary.com gives the three following definitions of a hacker:

1. One who is proficient at using or programming a computer; a computer buff.
2. One who uses programming skills to gain illegal access to a computer network or file.
3. One who enthusiastically pursues a game or sport: *a weekend tennis hacker.*

Unfortunately for our hacker friends, it took a lot in those days to make a movie. You needed sound stages, explosions, and a lot of other things that they just simply had no access to, which meant they would not be making movies any time soon. As fate would have it, however, a new medium was about to make itself available to them through which they could create their own adventures in space.

The 1960s

Now, up to this point in time, all computers were huge, I mean, gigantic. These things were often literally the size of a small house! While things begin to get a little smaller due to transistorization, computers like the TX-0 mainframe were still pretty big. Then came the PDP-1 from Digital Equipment Corporation, which can be seen in Figure 3.7. The PDP-1 was groundbreaking in a number of ways, most notably that it was much smaller than its predecessors. Another key feature of this machine was that it was more user friendly than past computers had been. In order to use previous machines, you had to have a degree in electrical engineering. Furthermore, you virtually had to perform a séance every morning to get the machine to start up, and shutting it down was not any easier. With the PDP-1, all you had to do to turn it on was flip a switch, and you could just shut it off any time you wanted to without fear of irreparable damage to the machine. This was unheard of in 1961.

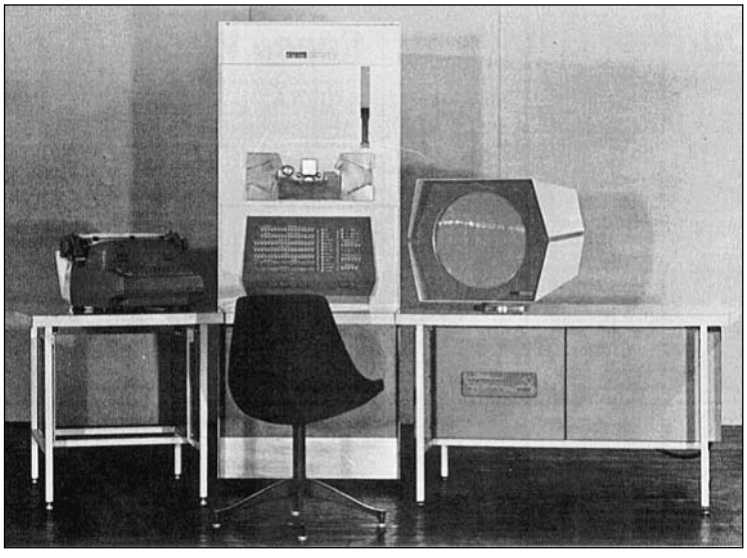


Figure 3.7 Digital Equipment Corporation's PDP-1.

Fascinating though these computers may be to computer geeks such as myself who love them for their inner beauty and charm, the average person takes very little pleasure in watching them. They are not much to look at. Despite their size, to the untrained eye they just look like a bunch of closets filled with tape recorders, glowing tubes, and lights that seem to hum for no apparent reason. In order for the average person to appreciate these machines you have to give them something to look at, like a cathode ray tube (CRT), more commonly known as a TV set.

You can take the most computer-illiterate person in the world and sit him in front of a TV set for hours on end without his losing interest once. So, it stands to reason that if you can make a computer do something with a CRT, people will be fascinated and amused by what the computer is doing for hours on end. The truth is that not only was it amazing how long people would remain captivated, but also how little the computer actually had to be doing in those days to capture their attention.

This phenomenon was demonstrated yearly at MIT's annual Open House Day using the screen of a computer called the Whirlwind. It was here that the first ever demo that utilized a CRT was built and demonstrated. What did it do? Well, it caused a dot to appear at the top of the screen. This dot then fell as though being pulled down by gravity to the bottom of the screen and bounced repeatedly, losing momentum each time, until finally it rolled off the screen. See Figure 3.8.

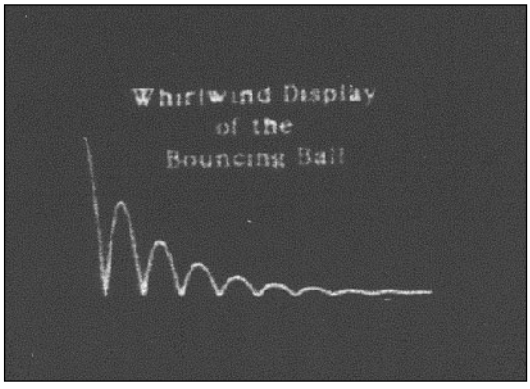


Figure 3.8 The infamous bouncing ball demo.

The TX-0's answer to the bouncing ball was a demo called "mouse in a maze," which was written by Douglas T. Ross and John E. Ward. This demo allowed the user to create a maze and place pieces of cheese throughout the maze. A virtual mouse would then navigate its way through the maze finding the pieces of cheese. Another cool version of the demo replaced the cheese with martinis. After the mouse found the first martini, it would stagger as it walked to find the other martinis. There was another very interesting demo available for the TX-0 called HAX, which displayed changing patterns on the display and emitted amusing sounds from the speaker. The display of patterns could be effected in real time by using two console switch registers.

The Saga Continues

The final demo that was available for the TX-0 was a game of Tic-Tac-Toe.

It is at this point that we must return our focus to the Tech Model Railroad Club. Several months before a PDP-1 was scheduled to be installed at their campus, Wayne Witanen, mathematician, early music buff, and mountain climber; Stephen R. (Slug) Russell, specialist in steam trains, trivia, and artificial intelligence; and J. Martin Graetz, self-described "man of no fixed talent who tended to act superior because he was already a published author," all came together to form a sort of committee to decide what they should do with it. You see, the demos for the TX-0 and the Whirlwind formed the foundation for building a great demo for the PDP-1. The bouncing ball was a pure demonstration. "The mouse in a maze" was unique in that it allowed the user to manipulate it and make it different every time it ran. HAX was advanced in that it was a graphics program that you could interact with in real time. And of course, the significance of Tic-Tac-Toe was that it was a game (simplistic as it may be, it was definitely a game).

To create a demo for the PDP-1, they quickly developed the criteria that they felt they must meet to create a great demo:

- It should *demonstrate* as many of the computer's resources as possible, and tax those resources to the limit.
- Within a consistent framework, it should be interesting, which means every run should be different.
- It should involve the onlooker in a pleasurable and active way; in short, it should be a game.

But wait, hmm, something was missing. What could it be? What was missing was the reason that the TMRC was so interested in this machine in the first place. They may not have been able to create a movie based on the works of E. E. Smith, but using this machine, they could produce a game based on the works of their beloved author. The group concluded that they would create a space game, which would revolve around two user-controlled space ships that would be equipped with some form of weapon, such as a ray, beam, or missile. Finally, to round out the initial features of the game, it was decided to implement a hyperspace feature that would allow a player to disappear when things got too sticky and reappear at a random location on the field.

Okay, so it was all set. They would build the game, but there was one problem. The PDP-1 was a very no-frills machine, and apart from a few diagnostics and utility programs, there were no tools to make the game with. As a result, before they could build the game, they had to build the tools to make the game with. Of course, this was no problem for our hacker friends; as a matter of fact, they liked it that way. Jack Dennis' MACRO assembler and Thomas Stockham's FLIT debugging program, were the first of their kind when they were created for the TX-0. Both of these programs were translated from TX-ish to PDP-ese. Steve Piner wrote a text editing program called *Expensive Typewriter*, which was the first word processing program ever created.

Wow, so they did it. The hackers had created the tools they needed to produce the video game. Now it was time for the real work to begin.

Before jumping in and building the beast called Spacewar!, the hackers cut their teeth creating a few smaller demo programs. First, they re-created the bouncing ball program on the new PDP-1. Next came *The Minskytron.* (Actually, its real name was *Tri-Pos: Three-Position Display*, but because it was created by Professor Marvin Minsky and "tron" was the cliché suffix of the early '60s, it was inevitable that rather than calling this program by its real name, it would be dubbed The Minskytron.) It displayed three dots, which bobbed and weaved and otherwise interacted with each other based on the values of a few initialization constants that were set via the console switches.

Now that preliminary work was out of the way, work could begin on building the actual *Spacewar!* game. During the first stage of development, our young programmers designed the *Wedge* and the *Needle*. These were the names given to the game's two space ships because of their shapes: one ship looked like a wedge of cheese, and the other looked like a sewing needle. It was decided that both ships should have starting positions located at diagonally opposite corners of the screen (or, in keeping with our space theme, two diagonally opposite quadrants) as seen in Figure 3.9.

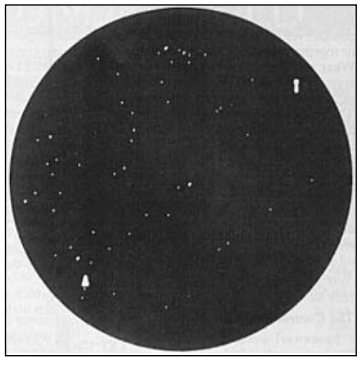


Figure 3.9 Opening round screen shot of Spacewar!

Already the game was somewhat playable and had very realistic game physics. When you fired the throttle, the ship would slowly accelerate until it got up to speed. In order to stop it, you had to turn around and fire your thrusters in the opposite direction. This use of "realistic" game physics definitely lent much to the feel of the game. The last feature that Russell, a.k.a. Slug, implemented at this stage of development was a random star generator. This was important because it was difficult to gauge relative motion with the ships on a plain black screen. At least with a basic star field background, your brain could compare the relative distance of the ship from the stars around it to recognize whether the ship was moving or standing still.

note

If you are reading this book, then you may have some interest in becoming a game hacker. It is important for you to understand that as a game hacker it is never good enough for you to just do something. Whatever you do must be done with style and class. Your program should either have elegant code or should carry out its function in an elegant way. Naturally, the ultimate of elegance is to write program code with style to produce a program that is beautiful. For this reason, a random star generator was not good enough for Spacewar!, and *Expensive Planetarium* had to be created, as we shall see.

The original control boxes looked something like that shown in Figure 3.10. The controls are

- Right-left rotation
- Acceleration (pulled back)
- Hyperspace (pulled forward)
- Torpedo button

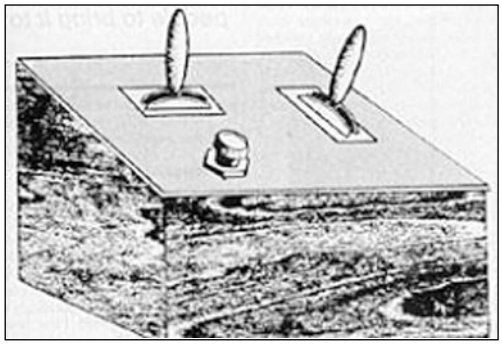


Figure 3.10 Drawing of the Spacewar! controllers.

And so the seeds of Spacewar! were sown, only they were less like seeds and more like a large stack of TNT on a short fuse that had just reached its end. The remainder of the development of Spacewar! did not occur in a series of steps and actions, but rather they took place as a flurry of activities all happening at the same time.

There were three things that kept the PDP-1 from being a viable game machine as it was.

- It would be very easy for a player intending to hit the torpedo button to hit the start button instead and "crash" the system.
- Because the screen was located off to the side of the machine's console, one player would be closer to the screen than the other and thus would have a visual advantage over the other player.
- The PDP-1's space was designed for one relatively calm systems operator, not two intergalactic space warriors locked in the thralls of combat.

Given the temperament of a game player in such cramped quarters (especially if he is losing), damage to the machine would be a constant risk. So what did our hackers do? They invented the first joystick, of course.

Alan Kotok and Robert A. Saunders marched off to the TMRC room and closed the door. When they emerged once again, they brought with them the first ever Spacewar! controllers. Compared to today's controllers they were relatively simplistic but very easy to use. The switch at the top of the box was used to turn the ship left or right. If you pushed the switch to the right and backwards, this would activate your thrusters while pushing the switch forward would activate your hyperspace drive. The button located at the bottom left of the controller was used to fire your weapon. Needless to say, the new controllers made the game much easier to play and therefore a lot more fun.

By February 1962, Spacewar! was playable. The game consisted of two ships, with a limited fuel supply, and an arsenal of "torpedoes" (these were essentially points of light fired from the nose of the ship). After you fired a torpedo, it would cruise along until it got near to a ship or its fuse ran out at which time it would explode.

note

Stephen R. Russell, a.k.a. Slug, gained his nickname from his friends because he had a tendency to procrastinate a lot. Towards the end of the year 1961, his excuse for not working actively on Spacewar! was "Oh, we don't have a sine-cosine routine, and, gee, I don't know how to write a sine-cosine routine. . . ." Alan Kotok came back from DEC headquarters with paper tapes saying, "All right, Russell, here's a sine-cosine routine; now what's your excuse?"

"Well," says Slug, "I looked around and I didn't find an excuse, so I had to settle down and do some figuring."

Do you remember my mentioning that for a hacker it is not enough to program; you must program with style? Well here is a quote from Martin Graetz that was published in the August 1981 issue of *Creative Computing* magazine:

> One of the forces driving the dedicated hacker is the quest for elegance. It is not sufficient to write programs that work. They must also be "elegant," either in code or in function—both, if possible. An elegant program does its job as fast as possible, or is as compact as possible, or is as clever as possible in taking advantage of the particular features of the machine in which it runs, and (finally) produces its results in an aesthetically pleasing form without compromising either the results or operation of other programs associated with it.

This quote says it all. At this time, Spacewar! was playable, but it was not yet elegant. One of the major drawbacks was considered to be the background. The random display of dots was just not considered appealing. It was at this point that Peter Samson wrote *Expensive Planetarium*. Based on *American Ephemeris* and *Nautical Almanac*, *Expensive Planetarium* was an accurate reproduction of the night sky between 22 ° N and 22 ° S. (This area includes the more popular constellations.) The realism of this facsimile of the night sky was further improved by firing each display point a set number of times, thus recreating the correct brightness of each star.

Spacewar! was growing in elegance, and it was fun to play—for a while. You see, any game that is purely a shoot 'em up type game is fun to play at first, but it quickly loses its appeal. So something had to be added to the game. Something that would add an element of strategy and force players to use more than just their motor skills and eye-hand coordination. Dan Edwards felt that some form of gravity was needed, and after Russell stated that he did not know how to program gravity calculations, Edwards sat down and programmed them himself.

The product of his labor presented itself as a blazing star located in the center of the screen. This star emitted a gravitational pull that could affect both ships no matter where they were onscreen. As long as a ship was not moving, it would be sucked into the sun. This feature worked well with the fact that each ship had a limited fuel supply. This meant that if you ran out of fuel, you would find yourself plummeting into this star. Needless to say, the use of strategy in the game was greatly improved.

note

When the "Heavy Star" was first implemented, the game lost its steady frame rate and begun to flicker. The shapes of the ships were stored in tables and were read from these tables on each game cycle. Dan Edwards realized that this way of doing things was overburdening the game engine and devised a small routine that would "compile the shape of the ship at the start of the game and draw the outline throughout the rest of the game cycle without consulting lookup tables." After this hack was implemented, Spacewar! regained its flicker-free frame rate.

The CBS Opening

There were some really cool things that could be done using the gravity of the Heavy Star. For example, a move called the "CBS Opening" (see Figure 3.11) became the standard way of beginning a match between skilled players. To perform this move, players turned slightly away from the Heavy Star and fired a short rocket burst that propelled the player's ship into an orbit around the Star. After both ships entered this orbit, they would turn and fire torpedo shots at one another.

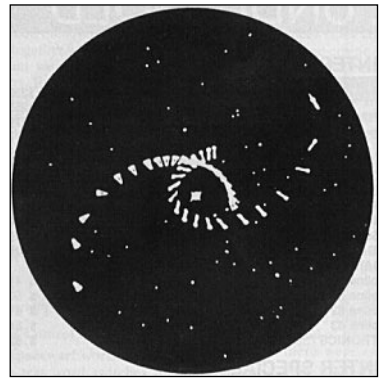


Figure 3.11 The CBS Opening.

note

Gravity calculations for the two ships were all that the program could handle. There simply was not enough power to apply gravity to each missile that was fired so the missiles were unaffected by gravity. The programmers simply made up a story to explain this away to the player. This is a very important point. The older machines' limited capabilities prevented you from implementing certain features that the player, being ignorant of game design, would assume should be there. So what do you do? You make the limitation a rule in the game. Most players will not question why rules were made, but if you just leave them hanging and they find out only after playing the game that they cannot do things they would expect to be able to do, then they will be very disappointed.

There was also another cool strategy that was actually the result of a bug in the game but added so much fun nobody bothered to change it. This maneuver involved plunging straight down the "gravity well" (in other words, flying almost directly toward the heavy star, also called the Sun) to gain more momentum than you were normally allowed by whipping around the Sun.

Spacewar! was now almost complete, and only one piece of the puzzle remained to be put in place: hyperspace. The basic idea behind hyperspace was that if you got into a jam, you could hit the hyperspace button, disappear for a few moments, and then reappear at some random location of the field. There was one large built-in problem with this feature though. Unskilled players could choose to jump in and out of hyperspace for the entire game, making the battle uninteresting and a bore to play.

This problem was solved in two ways. First, each player was allowed only three hyper jumps, so they could not abuse the feature. Second, the feature was made to be unreliable, which added an element of uncertainty. J. Martin Graetz, who was the creator of this feature, made up a story about "Mark One Hyper field Generators . . . hadn't done a thorough job of testing . . . rushed them to the fleet" to justify to the player why the hyperspace feature was unreliable.

The use of hyperspace was made even more perilous because after your ship's coordinates had been scrambled, you could reappear anywhere, including right on top of the Sun. Furthermore, your ship could reappear traveling at the same speed and trajectory as before it left, which meant that even if you did not land on the sun, you might still run into it or a missile. The final element of hyperspace was that this phenomenon would naturally displace the space surrounding the hyper jump. This effect was simulated by borrowing from an effect, pioneered in the Minskytron, which looked like a classical Bohr atom. (This symbol was an over-used cliché in the early 1960s to represent anything to do with science fiction.) Whenever a ship disappeared, it would leave behind a Hyperspace Minskytron signature. See Figure 3.12.

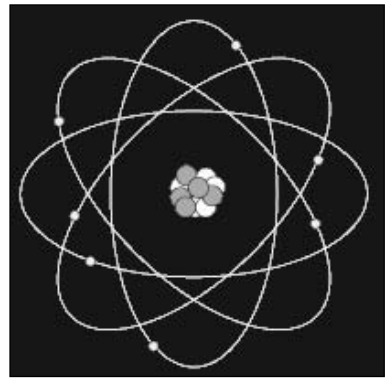


Figure 3.12 Diagram of a classical Bohr Atom.

Return of the Killer Pong

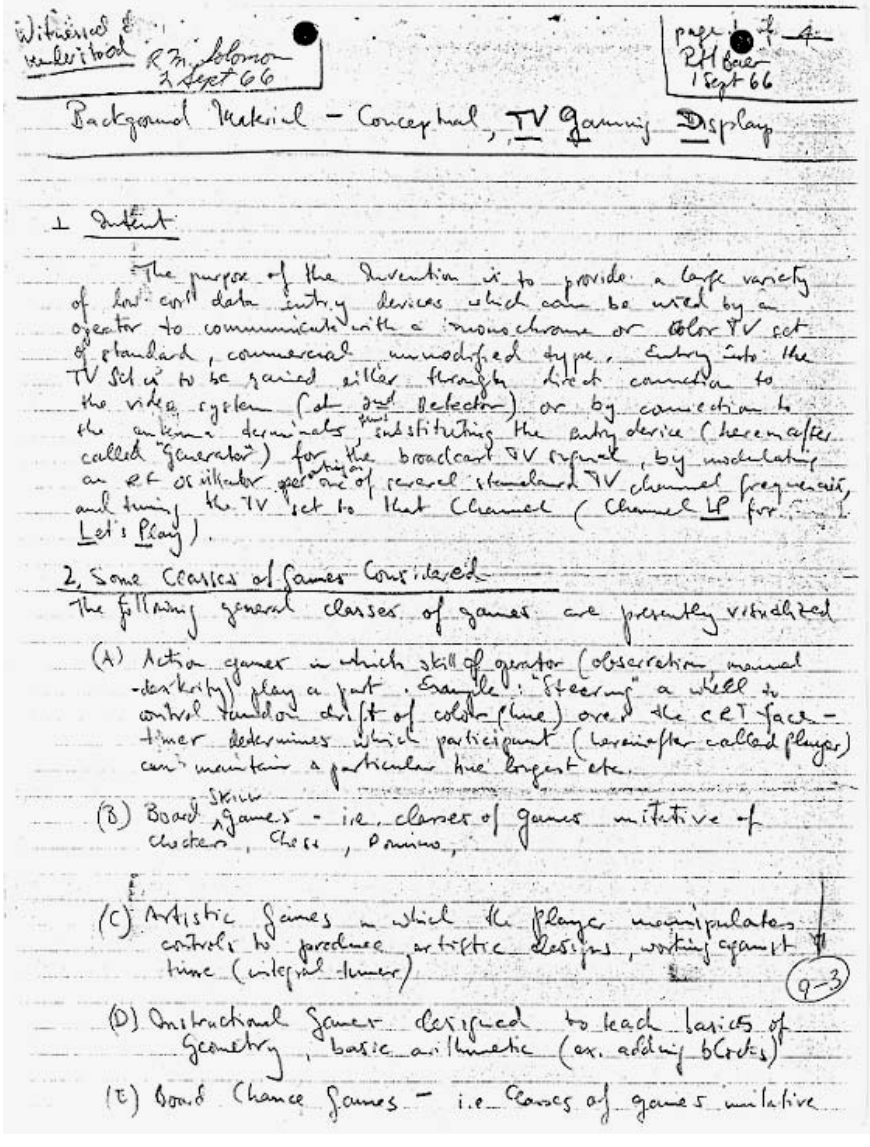
No dream of a hacker ever dies. Stop him from accomplishing his dreams today, and ten years from now he will still have a fire burning in him to complete what he started. Such was the case with Ralph Baer. Fifteen years after he originally came up with the idea of the video game, fifteen years after being stopped in his tracks by company brass at Loral from bringing his dream to reality, Ralph Baer was still thinking about building video games.

Of course, by then the first analog computer game had been built by William A. Higinbotham. However, Mr. Baer was not aware that this game had existed when he created his invention, and it wouldn't have made any difference if he had. Ralph Baer not only came up with a full-blown concept of playing video games on a home TV set, but documented this concept in a four-page document dated 1 September 1966. William A. Higinbotham also did not document his invention when he built it. Indeed, he never entertained the idea of making his analog computer game into a commercial product because this was manifestly impossible. For now, let us look at how Baer came to eventually convert his video game concepts into actual, working hardware.

The year was 1966, and a lot had changed. Ralph was no longer working for Loral. He was now chief engineer and manager of the equipment design division at Sanders Associates, a large military electronics development and manufacturing company that had absolutely nothing to do with television technology. That did not stop Mr. Baer, however. Fifteen years earlier, it had been his chief engineer who shot down his dream, but Mr. Baer was now the chief engineer and could have his subordinates do his bidding.

One day while Mr. Baer was sitting in a New York bus terminal waiting for a fellow engineer to arrive for a meeting, he started to jot down some notes concerning the concept of

building video games. Upon returning to his office in New Hampshire, he transcribed his handwritten notes into a four-page disclosure document. He asked Bob Solomon, one of the engineers in his division, to read, date, and sign the document to establish a legal record of when he created this design. Among other things, this paper documented categories of games, such as action games, board games, sports games, chase games, and more. These classifications are still used today. You can see an actual copy of the handwritten document in Figures 3.13 through 3.16.


Witnessed & understood R.M. Solomon 2 Sept 66

page 1 of 4 R.H. Baer 1 Sept 66

Background Material – Conceptual, TV Gaming Display

1 Intent

The purpose of the invention is to provide a large variety of low cost data entry devices which can be used by a operator to communicate with a monochrome or color TV set of standard, commercial unmodified type. Entry into the TV set is to be gained either through direct connection to the video system (at 2nd detector) or by connection to the antenna terminals, but substituting the entry device (hereinafter called "generator") for the broadcast TV signal, by modulating an RF oscillator operating on several standard TV channel frequencies, and tuning the TV set to that channel (Channel 4 for Let's Play).

2, Some Classes of Games Considered

The following general classes of games are presently visualized

(A) Action games in which skill of operator (observation, manual dexterity) play a part. Example: "Steering" a dot to control random drift of color (hue) over the CRT face – timer determines which participant (hereinafter called player) can maintain a particular hue longest etc.

(B) Board Skill games – i.e. classes of games imitative of checkers, chess, dominos,

(C) Artistic games in which the player manipulates controls to produce artistic designs, working against time (integral timer) 9-3

(D) Instructional games designed to teach basics of geometry, basic arithmetic (ex. adding blocks)

(E) Board Chance games – i.e. classes of games imitative

Figure 3.13 Page 1 of Baer's notes. *Notes and drawings on these pages courtesy of Mr. Ralph Baer.*

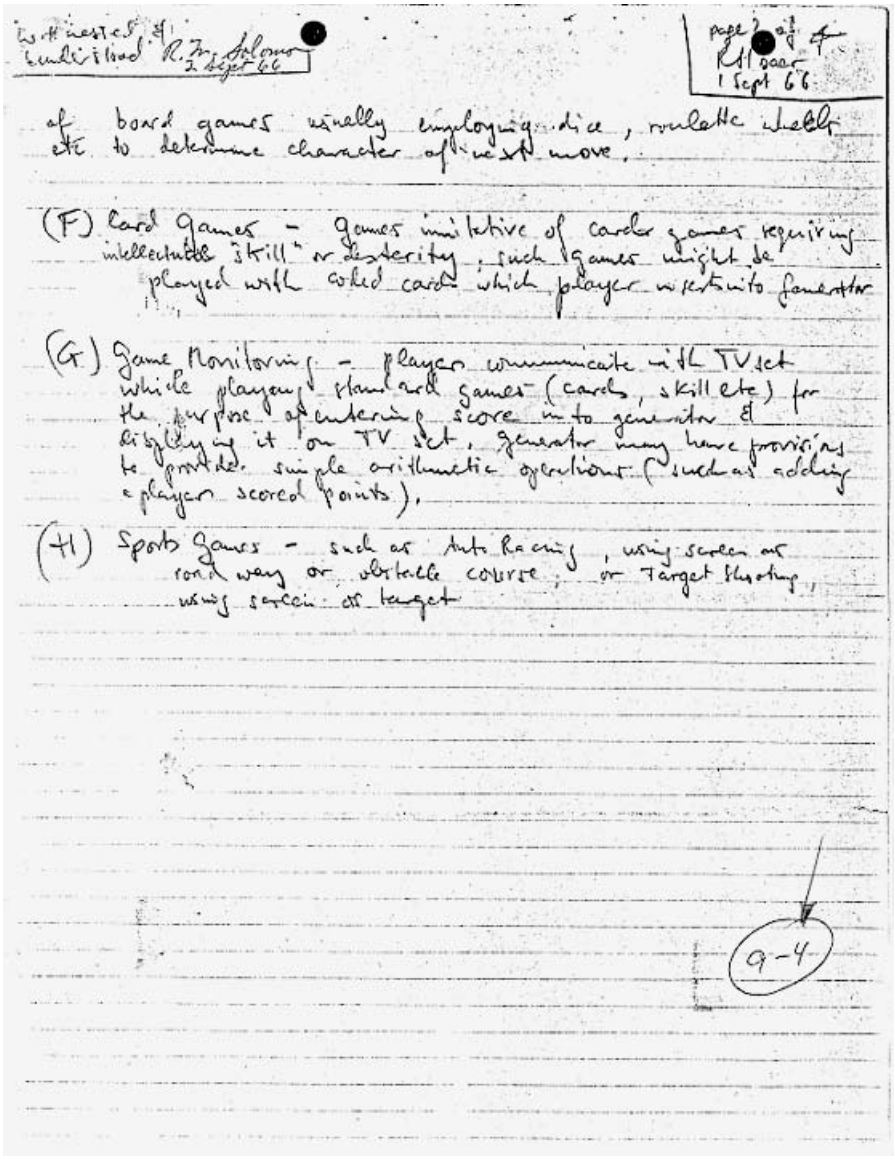
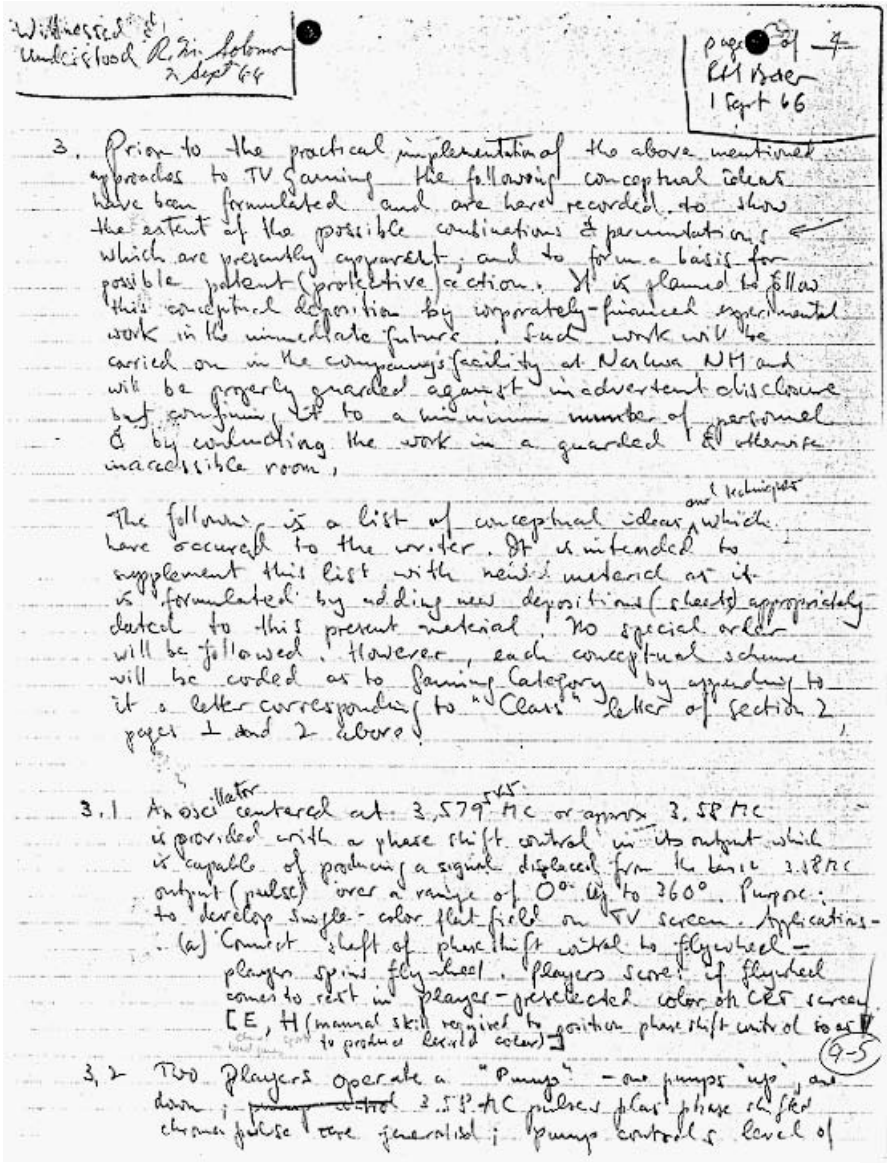


Figure 3.14 Page 2 of Baer's notes.



Witnessed & understood R. Baer 2 Sept 66

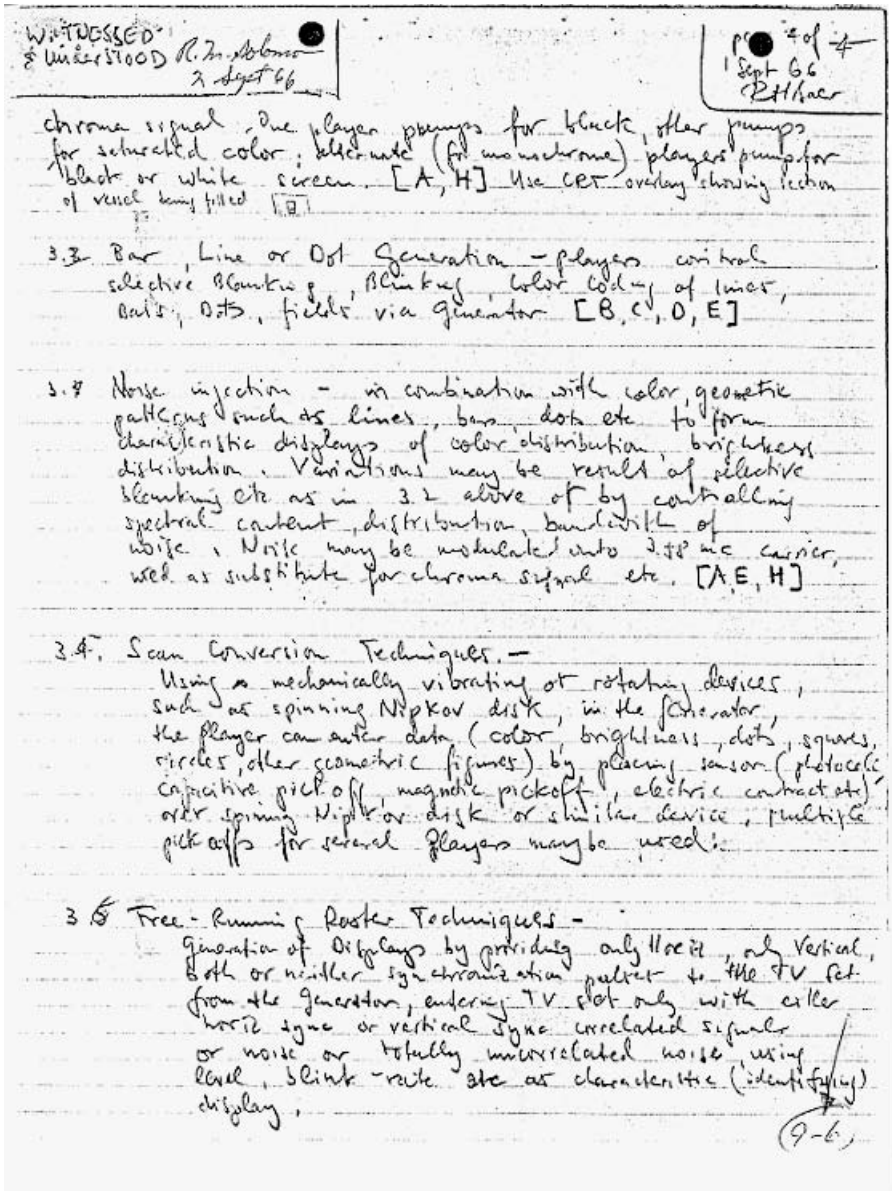
page 3 of 4
R.H. Baer
1 Sept 66

3. Prior to the practical implementation of the above mentioned approaches to TV gaming the following conceptual ideas have been formulated and are here recorded to show the extent of the possible combinations & permutations which are presently apparent, and to form a basis for possible patent (protective) action. It is planned to follow this conceptual disposition by corporately-financed experimental work in the immediate future. Such work will be carried on in the company's facility at Nashua NH and will be properly guarded against inadvertent disclosure by confining it to a minimum number of personnel & by conducting the work in a guarded & otherwise inaccessible room.

The following is a list of conceptual ideas and techniques which have occurred to the writer. It is intended to supplement this list with new material as it is formulated by adding new dispositions (sheets) appropriately dated to this present material. No special order will be followed. However, each conceptual scheme will be coded as to gaming category by appending to it a letter corresponding to "Class" letter of section 2, pages 1 and 2 above.

3.1 An oscillator centered at 3.579 MHz or approx 3.58 MHz is provided with a phase shift control in its output which is capable of producing a signal displaced from the basic 3.58 MHz output (pulse) over a range of 0° up to 360°. Purpose: to develop simple-color flat-field on TV screen. Applications - (a) Connect shaft of phase shift control to flywheel - player spins flywheel. Player's score if flywheel comes to rest in player-preselected color on CRT screen. [E, H (manual skill required to position phase shift control to accurately produce desired color)] (9-5)

3.2 Two players operate a "pump" - one pumps "up" and down; ~~pump control~~ 3.58 MHz pulses (var phase shifted) chroma pulse rate generated; pump control's level of



WITNESSED & understood R. H. Baer 2 Sept 66

p 4 of 4
1 Sept 66
RHBaer

chroma signal. One player pumps for black, other pumps for selected color; alternate (for monochrome) players pump for black or white screen. [A, H] Use CRT overlay showing section of vessel being filled [B]

3.3 Bar, Line or Dot Generation - players control selective blanking, blinking, color coding of lines, bars, dots, fields via generator [B, C, D, E]

3.4 Noise injection - in combination with color, geometric patterns such as lines, bars, dots etc to form characteristic displays of color distribution, brightness distribution. Variations may be result of selective blanking etc as in 3.2 above or by controlling spectral content, distribution, bandwidth of noise. Noise may be modulated onto 3.58 mc carrier, used as substitute for chroma signal etc. [A, E, H]

3.4 Scan Conversion Techniques. -
Using a mechanically vibrating or rotating devices, such as spinning Nipkov disk, in the generator, the player can enter data (color, brightness, dots, squares, circles, other geometric figures) by placing sensor (photocell, capacitive pick off, magnetic pickoff, electric contact etc) over spining Nipkov disk or similar device; multiple pick offs for several players may be used.

3.5 Free-Running Raster Techniques -
Generation of displays by providing only Horiz, only Vertical, both or neither synchronization pulses to the TV set from the generator, ending TV set only with either horiz sync or vertical sync correlated signals or noise or totally uncorrelated noise, using level, blank-rate etc as characteristic (identifying) displays.

(9-6)

Figure 3.16 Page 4 of Baer's notes.

Spot Generators

Five days later on September 6, 1966 Ralph drew up simplified schematics for what he called *spot generators,* as well as a method for using them to modulate a transmitter tuned to channels 3 or 4. This allowed the signal for the spots to be distributed to any TV set through the antenna cables. This diagram can be seen in Figure 3.17.

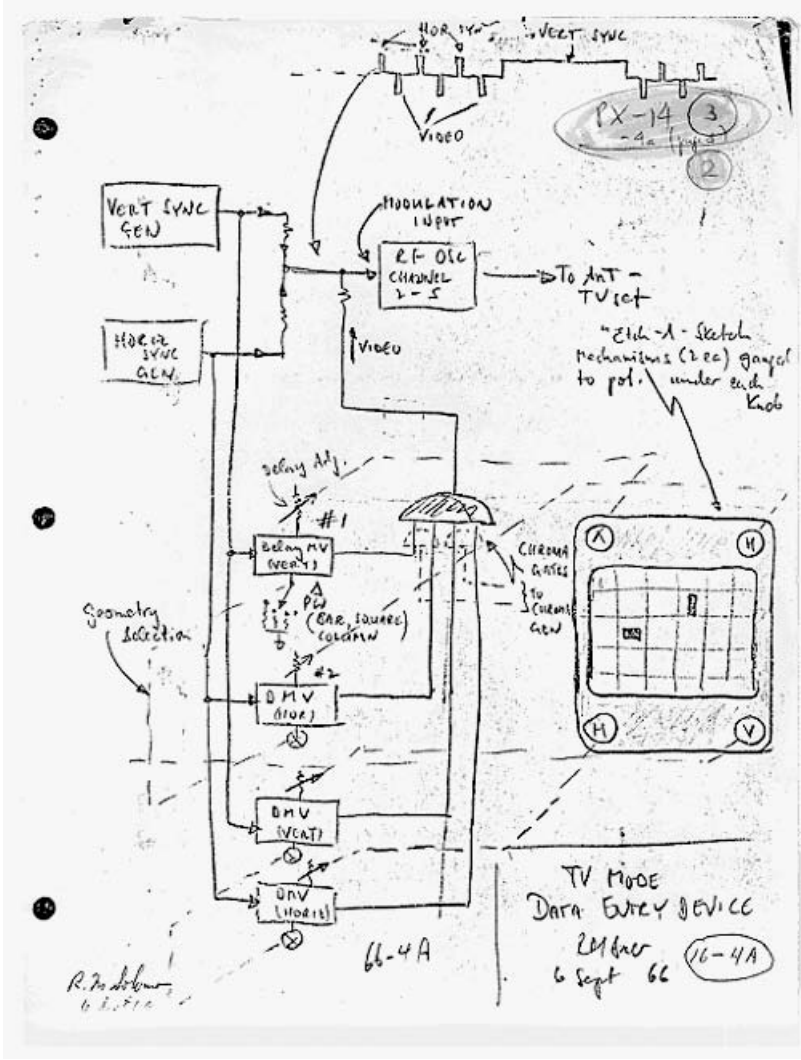


Figure 3.17 Hand drawn schematics for Baer's spot generators.

On October 20, Bob Tremblay completed the task of converting this basic diagram into an actual working prototype. What he came up with was rather primitive, but if you had enough imagination, you could pretend one spot was a fox and the other was a hound. The object of the "game" was for one player who was the hound to chase the player who was the fox until the hound caught the fox. Figure 3.18 shows a picture of the actual game while Figure 3.19 shows a picture of the world's first light gun.

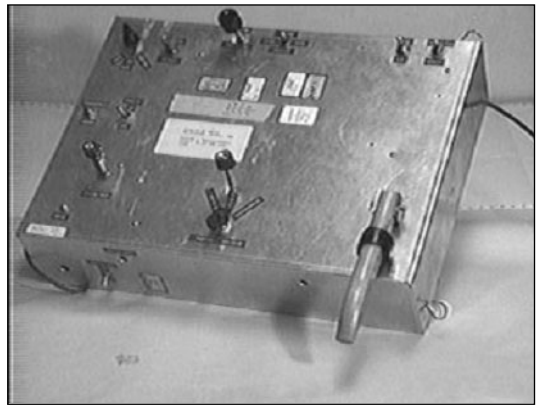


Figure 3.18 First ever two-person video game system (May 7,1967).

So they had a start, but even though Baer had accomplished his goal of creating a "game," he was still not finished. He still needed to follow the hacker's axiom and "do it with style." That said, in January of '67, Baer set technician Bill Harrison the task of creating the world's first multi-game system. This new system was truly bold and visionary. Previous video games were hard-wired to do one thing and one thing only: play the video game they were meant to play. Not only was Baer to create the first game system that played multiple video games, but he also introduced the revolutionary concept of the *light gun*.

Figure 3.19 World's first light gun.

The Home TV Game

Dubbed the "Home TV Game," the prototype was made ready for a presentation to the executives in the company. This would come in the form of a demonstration to Herbert Campman, who was the corporate director of research and development at Sanders. He loved the idea and approved the project for funding despite the fact that it had nothing to do with the military research his company specialized in.

The project was awarded a whopping $2,000 in funding. (Okay, so that's not whopping by today's standards, but back then it was a lot of money.) Additionally, a new engineer by the name of Bill Rusch was added to the unofficial team of game developers.

And what do you think was the very first game he built? Here's a clue: it features two paddles and a ball. You guessed it. He built *Pong*. Only it wasn't called *Pong* just yet. We will see a bit later how the infamous name *Pong* came into existence. For now, the game would be called *Catch*.

The new game sported a lot of high tech features, such as having the ball served from off the screen when a player missed it. (Once again, this was high tech by the standard of the day. Today, we take it for granted that all games have the latest 3D graphics, but back then just displaying more than two colors on the screen was a major accomplishment.) By the time Baer's invention would again be demonstrated, it boasted numerous games (including Ping-Pong, volleyball, handball, and several shooting games). There were also colored transparencies to place over the screen to represent the various play fields, and of course, their brand spanking new Light Gun.

All in all, Baer had accomplished a lot, but he still had much to do. He and his team continued improving their project, but they had one big problem: Sanders was a military company. Its daily business had nothing whatsoever to do with TVs. Sure, company execs thought that Baer's project was a cool idea, but they began to grumble and wonder exactly how Baer's project would profit the company. Baer had to move fast or risk having his idea axed by company brass. His first idea was for the games to be transmitted via cable TV networks. He worked out methods for accomplishing this, even going so far as to create a methodology for adding color to the play field, which would be broadcasted over the cable network. Unfortunately, at the end of the day, Baer's ideas were deemed not feasible.

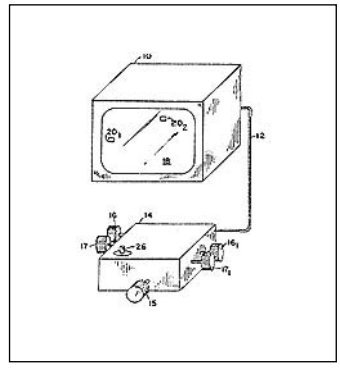


Figure 3.20 Drawing taken from Baer's patent application.

Baer was not deterred. On January 15, 1968, Ralph H. Baer filed for the first ever video game patent in history, patent number 480. See Figure 3.20.

In another effort to save his project, on October 1, 1968, Baer and his team demonstrated a complete switch-programmable video game unit that could play gun games, football, Ping-Pong, and volleyball. To represent the various play fields, transparencies were used to cover the screen and give it color. Now it was time for Baer to take his show on the road and make his project public. In January 1968, he, along with Lou Etlinger, who was Sanders' director of patents, began to invite all of the major TV makers to Sanders Associates' Nashua, New Hampshire plant to view a demonstration of the Home TV Game. Manufacturers invited included RCA, GE, Zenith, Sylvania, Magnavox, and Warwick (Sears). These companies were treated to a demonstration of Baer's most advanced prototype to date. A picture of this device can be seen in Figure 3.21.

Figure 3.21 Prototype of Baer's home video game system.

This device, which was the first fully programmable video game and included a joystick and light gun interface, was very impressive and indeed blew the crowd away. While everyone seemed to show interest, no one was willing to commit to the project. RCA went so far as to write a license agreement but quickly canceled it.

The demo Baer gave to prospective buyers was very impressive, and not everyone on RCA's team supported the decision not to move forward with Baer's idea. Bill Benders was one such individual, and although he was powerless to make Baer's idea a reality while at RCA, he would soon hold the position of Vice President at Magnavox. He was able to use his new and considerable influence at Magnavox to convince them of the virtue of investing in the home video game market.

The Odyssey

On July 17, 1970 at Magnavox's Ft. Wayne, Indiana plant, Ralph H. Baer and Lou Etlinger demonstrated their invention and blew the socks off of Magnavox's TV Marketing Division Vice President Derry Martin. On March 3, 1971, Magnavox licensed Baer's prototype (affectionately known as the "brown box"), all rights and patents related to the prototype, and all know-how related to the device. From March to September of 1971, Baer worked along with Magnavox engineers to produce the first commercially available video game. In March 1972, Magnavox dubbed Baer's invention the Odyssey. They presented the product to their dealers, and home video game systems were launched nationwide for the first time ever. See Figures 3.22 through 3.24.

The Odyssey retailed for $100, which was exactly $80.05 more than Baer had originally envisioned such games costing. Because Magnavox wanted to cut costs, the final release of the Odyssey was a somewhat downgraded version of Baer's original prototype. For example, unlike Baer's device, which used different colors for the play field, the Odyssey was strictly black and white.

Figure 3.22 The Odyssey was the first ever commercially available home video game system.

Figure 3.23 Magazine ad for the Odyssey.

Figure 3.24 Magazine ad for the Odyssey.

It had no sound at all. To make up for the lack of color, the Odyssey shipped with Mylar strips that could be placed over the screen to give the play field the appearance of different colors. For each game that shipped with the Odyssey, the appropriate strips could be used to give the screen the appearance of a tennis court or whatever playing field was appropriate for the game. The game shipped with 12 different cartridges that could be plugged into it to make it play 12 different games. The game also shipped with two controllers. Each controller had two knobs to control the vertical and horizontal position of the players "spot" and an "English" knob at the top of the controller to put a spin on the ball. Other item that shipped with the Odyssey included a pack of playing cards, poker chips, play money, a pair of dice, and a scoreboard to keep track of scores because the machine itself could not do so.

Magnavox did it! They were the first company to commercially produce home video game systems. But . . . yes, there is a but . . . their marketing strategy severely limited the flow of Odyssey systems and games off store shelves. First of all, because Magnavox makes TV sets and they were also the makers of this game system, people automatically assumed that you needed to use a Magnavox TV to play the game. This fact was further reinforced by Magnavox's own marketing campaigns and the fact that distribution of the Odyssey system and its game was limited to officially licensed Magnavox dealers. Fortunately, (with a little help from Frank Sinatra who was featured in a television commercial for the Odyssey),

Evolution of a Technology

Before there were movie cameras and the cinema, people would go to the theater and watch plays. After the movie camera was created, people used them to record plays, which would then be played in the cinema. So basically when you went to the movies, you were just watching a prerecorded play! Later people realized that they did not have to film the whole play in order. They could film the action one shot at a time. If there were mistakes, they could just retake the shot where the mistake was made. Someone said, "Hey, if I am filming a scene with a man in it, and I stop the camera, remove the man from the shot, and then continue shooting, I can make the man seem to disappear."

Although it is outside of the scope of this book to go over the history of the movie industry, I mention this topic to bring to your attention what often occurs when new technologies are introduced. It takes time for people to fully grasp exactly how to leverage the full force of any new technology. Full realization of the potential of the home video game system from a technological as well as a marketing standpoint was a gradual process, so it should be no surprise that so many blunders occurred so often in this industry, as they do in most industries.

Imagine what executives from Xerox must have thought years later when the idea of the "mouse" that they laughed at earlier became a staple of the computer industry.

they were still able to ship 100,000 units in their first year. Magnavox's great fortune was to be earned in the computer industry a few years later, only this fortune would be mined from the courtroom rather than from store shelves.

note

It should be noted that, although remarkable, the creation of the first commercially available game was not Baer's only accomplishment. He went on to create (and patent) numerous toys and other inventions, such as the first VCR-based Nested Data interactive TV gaming system. This was the Bike Max talking bicycle computer, a prototype to play games through the cable system, and the infamous Simon, which was actually based on an idea created by industry legend Nolan Bushnell.

The Syzygy

All right, it was finally here. The video game industry had been born. It was a long time coming and now that it was here, it was not to be given a very long childhood. The computer industry would experience a growth spurt and develop at a fast and furious pace largely due to the miniaturization of the machinery used to create and play games. Previously, games could only be played on large mainframe computers that cost millions of dollars and could only be afforded by large universities. (Indeed at this time only three universities in America could afford the PDP-1 and the expensive monitor needed to play the infamous Spacewar! computer game.) On the other hand, Ralph Baer had just completed a 15-year quest, which ended in the production of a video game console that was not only capable of playing multiple games, but more importantly, did not require million-dollar hardware to run and could actually use a player's home television set for display.

Have you ever watched one of those old Gothic movies where a sorcerer was trying to execute a spell but in order for everything to work, 10 planets had to be in precise alignment and this happened once every million years? And even then, he had to mix frogs' legs, eye of newt, and a number of other unique ingredients in order for his creation to come to life. Well, the computer industry must have been spawned by such a sorcerer, because a whole lot of things that had to line up for the industry to come to life somehow all lined up and fell into place. A prime example of this is that, among the hundreds of universities in the United States, only three could afford the expensive monitor needed to play the Spacewar! video game. One of the universities that had the ability to play it was the University of Utah, which happened to be the school attended by Nolan Bushnell, who just happened to become the Godfather of the arcade gaming industry.

What did I tell you? Are those planets lining up or what? Oh, yeah, by the way, I forgot to mention Nolan Bushnell was also the manager of an arcade in a Salt Lake City amusement park! Now with all these planets in alignment, the mood was right to stimulate the mind

of this 23-year-old college student and produce a vision in his mind that gave him the enlightenment to understand the commercial viability of a Spacewar! type video game that could be played on something other than a huge million dollar machine that the masses would never get to play.

They say the journey of a thousand miles starts with the first step. For Bushnell, his journey of four years started in 1970 with moving his daughter Brittia from her own room into her big sister's room so that he could convert hers into a workroom dedicated to building what would soon be called "Computer Space."

The next stop on his journey took place later in 1970 when he was employed by a company called Ampex in Sunnyvale, California. Ampex was in the business of making professional video tape recorders. He was paid the grand sum of $12,000 a year. It was here that he would join with Ted Dabney to not only build Computer Space, but also become a major force in the gaming industry. (That last part will come a bit later, as we shall see.) Filled with great ambition and confidence that he could make Computer Space a success, Bushnell boldly quit his job in 1971 to work on the project full time.

Fortunately, after all his labor, Bushnell was able to find Nutting Associates, a manufacturer who was willing to take a chance and build his dream machine. Bushnell became an employee of the firm. Unfortunately, the game did not sell well at all. It was a major disappointment for him as well as Nutting Associates, which had made a leap from its usual business of making coin-operated trivia games to try Bushnell's idea. But even though only 1,500 units were built, the experience was not lost on this young visionary—just as being shot down by his superiors at Loral did not stop Ralph Baer from accomplishing what he needed to do.

The whole ordeal was simply a learning experience. And what, you may ask, was the lesson? If you are going to make a game (especially one that is likely to end up in bars where people are relaxing and having fun, possibly even drunk, not looking for a challenge), you have to make the game as simple as possible, yet still fun and engrossing.

Simon Says

Do you remember the electronic video game called *Simon*? You know the one. Round circular disc with four colored buttons that flashed in random patterns and you had to repeat the pattern by pressing the buttons. Well, Baer invented that. The interesting thing is that he got the idea indirectly from Bushnell. In 1976 Mr. Baer was at a trade show where he saw a light and sound game called *Touch Me*, which was created by Atari. Three years later, Baer's game was successfully released by Milton Bradley. Interestingly enough, the patent that Ralph Baer got for the game actually cites the operating manual for *Touch Me*.

Now once again we have those planets I keep talking about positioning themselves in perfect alignment to facilitate the creation of the game industry we have today. In May 1972, the Magnavox Profit Caravan trade show at the Airport Marina Hotel in Burlingame, CA was featuring a demonstration of their new game system, the Magnavox Odyssey. I am sure you will never believe who showed up for the demo. Nutting Associates had caught wind of the demonstration and sent our beloved friend Nolan Bushnell and two other Nutting employees to check it out.

So Nolan went, he signed the guest book, he picked up the controls to play what essentially was Ralph Baer's Ping-Pong game, and he stayed there for half an hour playing the game. When he returned to Nutting Associates and was asked about the Odyssey, Bushnell replied "It's no Computer Space!"

Right as he was about the Ping-Pong game being" . . . no Computer Space," that simple game would have more of an impact on his life than he could ever have imagined that fateful day when he first played it.

I Am a Movie Star

The original Computer Space game may not have made it in the game business, but at least it made it to the big screen. The high tech case that housed Computer Space was so stylish that it was used as a prop in the 1974 sci-fi flick *Soylent Green*.

Many times, endings are nothing more than a new beginning, and such was the case for Bushnell when his relationship with Nutting Associates came to an end and he left (despite being offered a chance at creating another game) to form his own game company. He and partner Ted Dabney pooled their resources to obtain their starting capital. (Bushnell contributed $250 and Dabney contributed $250 for a grand total of $500, which, oddly, was the exact amount of money that they made in profits from selling Computer Space.)

Now here is the kicker. They decided to call their new company Syzygy (pronounced siser-gee). *Syzygy* is an astronomical expression used to describe the condition of the earth, moon, and sun all being in perfect alignment. (And you thought I was crazy with all this stuff about planets aligning.) By now, these two must have realized that they were becoming a part of something much bigger than themselves and much bigger than they had originally dreamed. Whatever the reason that they chose this name, it was apparently not meant to be; a roofing company had already registered that name.

Atari and Pong

Bushnell had a love for playing a Japanese game called GO. In this game, the equivalent of obtaining a check in chess is called *Atari*. On June 27, 1972, at the age of 29, Bushnell established his new company called Atari.

In one of the many twists of fate that are common in the game industry, when the time came for Bushnell to work on his first project under his new company's umbrella he ended

up hiring the very engineer (Al Alcorn) that Ampex had hired to replace him when he left there. Now, as I said, Bushnell was a visionary so, like most visionaries, he thought very big in everything he did. On this first endeavor, however, he decided to tone things down a bit and break in his new rookie with a simple game to test his abilities and ease him into game development. He decided that they should create a simple tennis game where the players controlled two paddles that could be used to bounce a ball back and forth. Hmmm, does that sound familiar to you? Doesn't it sound like the game Nolan Bushnell spent half an hour playing earlier that year at the Magnavox Profit Caravan? It was this similarity between the games that led to years of intense courtroom drama and controversy.

In an effort to encourage Al Alcorn to work intensely at developing the game, Bushnell told him that they already had a contract from General Electric to build the game. This contract, of course, did not exist and was simply a ruse he created as an incentive.

Even though Bushnell had toned down the intensity of what he wanted out of Atari's first game, he was still demanding. He even went so far as to request that Alcorn implement sound effects like roaring crowds, despite the fact that the technology to do such things was nowhere near being invented. Alcorn when to work. With no ROMs, and no microprocessors at his disposal, he had to hardwire the entire system using about 100 conventional logic integrated circuits. Unlike the Odyssey, this machine was designed to do one thing and one thing only. Fortunately, it did that one thing very well. This machine was made to play Pong, the name assigned to the game when Alcorn tried to describe the sound the ball made when it hit the paddle.

Pong probably had the shortest instruction manual in all of computer gaming history. The instructions read:

"Avoid missing ball for high score."

For some strange reason, corporations often seem determined to misunderstand the potential of new technologies. This fact was once again demonstrated when Bushnell tried to sell Chicago-based pinball giant Bally the Pong concept and they just completely blew him off. When he returned home, Bushnell installed the prototype machine in a bar named Andy Capps. That same evening he got a heated call from the bartender telling him that the machine was not working and he should "get the f#*@% thing out of here." Imagine Bushnell's surprise when he found out that his machine was not working because it was too full of quarters and the eager patrons who wanted to play the game simply could not push any more coins into the machine.

He decided then that Atari would build its own Pong machines. Bushnell set up shop in an old abandoned roller skating rink, and by the end of 1973, he had cranked out and sold 8,000 Pong units plus a few hundred Pong Doubles and Gotcha games. This was an extraordinary accomplishment especially when you consider that at this time in history a pinball machine was considered a hit if it sold 2,000 units in its entire production cycle.

Nolan thought a Pong game machine could average $25 dollars a week. Pong machines often actually generated as much as $100 dollars in quarters each week.

This "simple" game that Nolan Bushnell originally thought would just be a quick stepping-stone before moving on to bigger better games, this unit that cost $500 dollars to make and sold for $1,200, this game called Pong, would carry Atari for the next two years.

The Knockoff

In the summer of 1996, I attended a summer engineering academy at the University of Michigan. On one of our field trips, we visited a pharmaceutical company. They explained to us the long process that goes into making a new drug. Then they explained to us that they labor for years at a time and spend millions of dollars to manufacture a drug. However, once the patent runs out, other companies, rather than having to spend years of time and millions of dollars to produce a similar drug, only have to demonstrate that the drug they produce has the same chemical base as the original drug. That is how generic drugs come about.

A similar concept occurs in the gaming industry. One company labors and sacrifices and sheds blood, sweat, and tears just to produce a hit game. Then a million other companies simply produce knock-off imitations of the game's concept. This whole phenomenon got started in 1973 when everyone started producing "pong type" games. Even Nutting Associates, Bushnell's old employer, got in on the act.

The following are some of the games created in the Pong "mode."

Knockoffs Made by Other Companies

Games	Company
Eloping	Taito
Pong Tron	Sega
Pong Tron II	Sega
Tennis Tourney	Allied Leisure
TV Table Tennis	PMC
Hockey TV	Sega
Pro Hockey	Taito
TV Football	Chicago Coin
Paddle Ball	Williams
Soccer	Taito
TV Ping-Pong	Chicago Coin
Winner	Midway
Super Soccer	Allied Leisure
TV Ping-Pong	Amuntronics

Variations on Pong made by Atari

Dr. Pong
Pin Pong
Pong Cocktail
PONG Doubles
Puppy Pong
Quadrapong
Rebound
Spike
Super Pong

Despite all of the competition, however, Atari was still the clear market leader raking in $3.2 million in 1973. Unfortunately, despite its success, Atari appeared to be falling apart. Bushnell's long time friend and partner decided to leave the company and sold all his shares to Bushnell. Around the same time, a company called Kee Games, headed by Joe Keenan emerged as a major force to reckon with and became Atari's chief competitor. Several of Atari's key employees defected to this new company and from the outside it looked as if Atari might be looking at some darker days ahead.

Then it came—the game that made Kee Games look like it had Atari beat for sure: Tank! This game, created by Scott Bristow, was groundbreaking. Because of the rudimentary circuit being used, previous game machines had very limited graphics. (They could not display anything except blocks). Tank! changed all that by utilizing ROM chips in its design for the purpose of holding the graphics memory. This new and never-before-seen configuration gave Tank the ability to display much more complicated and intricate graphics than had ever been seen in a video game.

Remember we are talking about arcade video games here. At this point computer games were still run on very large, very expensive, mainframe computers that the general public had no access to. Sure, computer games would have had better graphics, but what mom would have been willing to remove everything from the living room and tear a hole in an outside wall just so her kids could get a mainframe computer in there to play Spacewar! on? Not to mention the expense.

Game play in Tank! consisted of two tanks in a maze. The two tanks were controlled by two players who tried to destroy each other's tank while avoiding land mines that were scattered over the playfield. Tank! became the best selling game of 1978, dealing a deathblow to Atari. Or did it? As it turns out, Kee Games was really a secret subsidiary of Atari. Why would Atari do a thing like that? What's with all the cloak and dagger covert corporate maneuvering?

Before there were video games, there were pinball machines. As video games started to take over the arcades, many of the old pinball doctrines and practices were still observed. One such practice was that when a game developer made a game and approached a distributor to distribute the game, that distributor would want exclusive rights to all games made by that company. So Atari might create a game this week called Billy Bob's Derby and sign a contract with Distributor A to distribute the game. Distributor A would make the company sign a contract stipulating that if Atari made a game next month called Super Sports Rally, Distributor A would be the only company that could distribute the new game. By creating a new company, one that had not signed contracts with any distributor and had no obligations to bind them to any distributor, and then creating a super hit game that every distributor wanted to get its hands on, Atari was able to break away from the old pinball practice and create distribution relationships with multiple distributors.

Once Kee Game had accomplished what it was created to accomplish, it once again merged with Atari. Joe Keenan took on the role of President at Atari, Steve Bristow became Atari's chief engineer and business went on as usual.

Games Created by Atari under the Kee Games Label

Name of the game	Year Created
Elimination	1973
Formula K	1974
Spike	1974
Twin Racer	1974
Crossfire	1975
Indy 800	1975
Tank II	1975
Flyball	1976
Quiz Show	1976
Sprint II	1976
Tank 8	1976
Drag Race	1977
Sprint 8	1977
Super Bug	1977
Ultra Tank	1978

Ultra Tank, which was a sequel to the original Tank! Game, was very cool because it was one of the first video games to allow a player to face off against a machine-controlled opponent. Not only that, but there was a version that would allow eight people to play against each other at the same time.

No one can deny that Pong, the arcade game, was a phenomenal smash hit. So it was no surprise that Bob Brown and Harold Lee (both employees of Atari) thought it would be a good idea to build a version of Pong that was designed to be played on a home TV set. They, along with Al Alcorn, the guy who built the original Pong arcade game, built this new system with the code name Darlene. (Darlene was an employee at Atari. It would soon come to be a tradition at Atari to name game systems after fellow female employees.)

Unfortunately, retailers were nervous about taking a chance with the machine. Magnavox's TV-based Odyssey game production and sales ended early in 1975 and there was no obvious sequel in sight, although Magnavox was about to come out with a new line. Still, the home videogame category had not yet been a blinding success for retailers, and they were afraid that Atari's Pong would suffer the same fate. As a result even though the game was created in 1974, it was not to be released until 1975 when they were able to make a deal with Tom Quinn, head purchaser for Sears sporting goods section. So Atari found a buyer for Darlene; that's the good news. The bad news was that there was no way in the world that they could produce the 150,000 units Sears ordered in the short space of time that Sears had given them to fulfill the order.

Bushnell needed to expand Atari's infrastructure big time. He was able to do so with the help of Don Valentine, who gave him a $10 million line of credit. Once again, Atari struck digital gold. There were reports of people waiting in line for hours before stores opened in order to plop down 100 bucks and get their own Pong game machine.

Big Business

And, yes, as you probably guessed, once again there was a massive digital gold rush as dozens upon dozens of manufacturers came out of nowhere with their own home version of Pong, trying to make their own fortunes. This time, however, it was much easier for others to mooch off Atari's idea with the aid of an invention developed by General Instrument. This invention came in the form of a microchip called the AY-3-8500 and dubbed *Pong on a chip*. It contained all of the essential circuitry needed to create a Pong video game. So, rather than spending large amounts of time developing the chip which constituted the internal workings of their own game machine as Atari had, all the other manufacturers had to do was follow directions, properly construct the external circuit that the chip needed, and, presto, they had an instant game of Pong.

Nothing is ever completely sure in the game industry, however, and success is just as much the result of luck as it is planning, skill, and strategy. As luck would have it, because so many companies were ordering the chip, it was impossible for General Instrument to meet all of the orders. And, as luck would have it, Ralph Baer had introduced Coleco management to the AY-3-8500 at G.I. As a result Coleco was the only company that was able to get their shipment of chips in time for the 1976 Christmas season. They sold their

machine for roughly half the price of Atari's game machine and in doing so increased their profits by 65 percent.

Lady luck was not only looking over the shoulders of Coleco. She was watching over Atari as well. Needless to say things were going well with Atari. Bushnell was happy and everyone who invested in his company was happy, so naturally people outside the company were taking notice and making efforts to see how they could be made happy too. One such company, or conglomerate, was Warner Communications, which became very interested in purchasing the now prosperous Atari. Bushnell's company was a hot commodity, and Warner was willing to pay a hot price, $28 million.

Kee Games and Other Maneuverings

After some prodding from his primary investor, Don Valentine, Bushnell decided to sell the company. For his efforts, he was able to secure $16 million for himself as well as the title of CEO. Joe Keenan was awarded the position of president. Now, Keenan was not just the president of an ordinary company; he was the president of a company that was about to take the video game industry, turn it on its ear and evolve a new creation, code named Stella. Stella would get Atari's groove back, once again knocking the entire gaming industry to its knees.

Atari was still very active generating its fair share of other games like Tank! Tank! was a breakthrough when it was released; it took video game graphics to new heights. In 1975 Atari used modified Tank! hardware to create the first ever video game with animated computer characters, Jaws. In another of its covert corporate operations, Shark Jaws, which was already produced under the Kee label for previously mentioned reasons, was also manufactured by Horror Game, which was of course also a subsidiary of Atari.

The reason for this latest tactical maneuver was that Atari was trying to cash in on the popular movie *Jaws*. Not only did Atari send out promotional material to prospective buyers urging them to cash in on the current popularity of sharks, but when they produced the cabinet they drew the word Jaws in big bold letters with shark written very small next to it. This popularity of course was a direct result of the movie *Jaws*. This little detail was, of course, left out of the promotional material for obvious reasons. So, technically the cabinet said Shark Jaws, but from a distance people would only see the word *JAWS* clearly and thus would associate the game with the movie.

Meanwhile, the rest of the video game industry was not resting on its laurels. Other companies were hard at work developing their own gaming systems. As a matter of fact, the gaming industry was booming with new players coming into the mix every year. This growth is plain to see; from 1971 through 1973, there where about 11 manufacturers. All of these companies together created a total of 30 video games. Over the next two years, video game production almost doubled. Between 1974 and 1975, 57 games were produced. In 1976, sales actually quadrupled with the production of 53 video games.

In order to keep things in perspective, we must mention the fact that not all of these were original games. In fact, most were actually Pong knockoffs, cashing in on Atari's success. At least one company, however, demonstrated that it had a mind of its own and could think for itself enough to create truly new and innovative game titles.

This company was Exidy. They created this name by combining the words *excellence in dynamics*. Years before the now famous Grand Theft Auto game series had a chance to take form in the minds of its developers, Exidy had already discovered the combination of freedom and carnage that gamers craved for. They proved that they knew what gamers wanted when they released their infamous game called "Death Race."

The idea of crossing the lines of what is taboo to intrigue gamers was first pioneered by Exidy. Perhaps their offering was not as advanced as Grand Theft Auto. There was no advanced AI, no 3D graphics, and no police chases. Nevertheless, this game was still very cool in its own gruesome way. You drove around the screen chasing after little running stick men. Whenever you successfully ran one of them down, they would scream and turn into a cross, which you now had to avoid running over or you would be the one turning into a cross.

Now while the game was obviously not as graphic as, say, *Mortal Kombat*, or as in-depth as *Grand Theft Auto*, it was graphic enough to get it yanked off the shelves. PTAs and other groups who were trying to save the children from the dangers of video games claimed that video games were causing delinquency, and Death Race was the poster child for a video game that would have a harmful effect on our children. (The artwork on the game's cabinet, created by Pat "Sleepy" Peak, did not help matters either; many found the graphics quite unsettling.) Exidy, however, was not deterred and despite the problems they had with their first release, which limited sales to a meager 500 units, the following year they released the sequel to their original title called Super Death Chase. This time they learned from their

In every industry there were always things that were considered taboo in that you just should not do them. For instance, in movies and TV, for many years it was just not done to show a couple sleeping in a double bed. Every married couple who shared a bedroom had twin beds with a bedside table between them. Much could also be written about the change in acceptable language over the years, but that too has been reflected in video game content.

In today's world restraints on behavior and language have slacked off a lot. Nowadays it almost seems that there is no more taboo. That is, until somebody really crosses the line and does stuff that shocks us all. Even though the standards change over the years, those who cross that ever-changing line generally gain much ridicule, much fame, and an ardent following. Such is the case with today's Grand Theft Auto series. The current installment, called "Vice City," contains options that you just know you're not supposed to be able to do in a video game.

mistakes; if people had a problem with them running people over and killing them in the game, they would kill the people before the game! In this sequel, the player chased after ghosts and skeletons. It seemed to work because this time there were no protests.

note

Death Race 2000

This game, created by the gifted game maker Howell Ivey, was inspired by the 1975 B movie of the same name, directed by Roger Corman.

Future Gods

Steve Jobs got started in the game industry working as a "game technician" who refined games and made them better as they came to the Los Gatos branch of Atari from the Grass Valley development labs. Back then, he made about $5 an hour, a far cry from his net worth today.

Our story begins on a late night at Atari. Steve Jobs and his friend Steve Wozniak are hard at work, or perhaps hard at play, on the latest Atari arcade machines. Steve Wozniak, future god of the Apple computer, is hooked; he loves playing arcade games and is amazed by their possibilities. He even goes so far as to create his own game. Steve Wozniak's game was fun to play and boasted antics such as displaying humorous messages on the screen when a ball was missed.

Steve Wozniak, also known as "Woz," soon got a chance to build his first commercial arcade game. The game was called "Breakout," and he designed it in 1976. Unlike most incarnations of Pong, which were two-player games and involved bouncing the ball back and forth across the screen, this game was different and involved using your paddle to bounce a ball against a row of blocks. Each time the ball hit the block it disappeared. The object of the game is to try and get all of the balls to disappear by hitting all of the blocks with the ball. Steve promised to have the job done in four days. This was a very bold promise. It was possible for Jobs to be this bold because he knew that he had a secret weapon. That weapon was Mr. Wozniak. In the true spirit of a computer hacker, he held down a full time job at Hewlett Packard while for four nights straight he worked on Breakout and completed the masterpiece on time.

With experience under his belt in the area of logic design and working with TV signals, he was now ready to take on the world. His first step was to create a computer version of Breakout. In another one of the major snafus of the computer industry, when Jobs and Wozniak presented their idea to Nolan Bushnell, he opted not to go along with the project. Even a demo of the system at Al Alcorn's house was not enough to get the project going. Bushnell's reluctance to take on the project probably stemmed from the fact that he was already dealing with major financial issues and was not ready to take on another risk.

At the end of the day, Bushnell directed the young inventors to Don Valentine, who in turn, directed them to Mike Markkula. It was at this point that Jobs decided to leave Atari and teamed up with his good friend Steve Wozniak to form Apple Computers. Together they would build the first Apple computer, which would spark the entire home computer industry.

The Birth of Vector Graphics

The cool thing about ideas is that you do not have to be a major corporation in order to have one as proven by a small El Cajon, California company named Cinematronics, which in 1977 created the first ever game to use vector graphics. Vector graphics are graphic displays that use lines and other geometric elements to draw the player's character and the environment.

This form of graphics became a huge hit. People loved it and even after people stopped making vector graphic games, collectors valued them greatly. It should be noted that all of the 3D games of today are vector graphic games, even though they are much more advanced.

Space Wars

Another one of the many coincidences that seemed to fall into place to allow the game industry to grow was Larry Rosenthal taking a tour of M.I.T. in 1968 and having a chance to play Stephen Russell's Spacewar!. Nine years later, he created his own version of Spacewar!. He tried for a while to market his game to several companies, including Atari, all of whom brushed him off, possibly because he demanded an unheard of at the time 50 percent of all of the game's profits.

Larry was finally successful when he managed to sell his game to . . . you guessed it . . . Cinematronics. He succeeded in retaining rights to his game, and they succeeded in purchasing a major hit. *Space Wars* would sell 30,000 units and remain among the top selling arcade games for three years straight. This would mark the start of a wonderful thing.

They would be responsible for some of the most interesting arcade games of their time such as:

Warrior, released in 1978
Rip Off, released in 1979
Tailgunner, released in 1979
Barrier, released in 1979
Star Castle, released in 1980
Armor Attack, released in 1980
Solar Quest, released in 1981
Starhawk, released in 1981
Cosmic Chasm, released in 1983

A New Age of Video Games

Up to this point in time creating video games was the equivalent of writing in stone. Once the game machine was built, you were stuck with whatever games were on the system when you bought it. This was great at first, but after you had mastered all the games on your machine, things could get boring.

This was true until 1976 when Fairchild Camera & Instruments released the Channel F video game machine. See Figures 3.25 through 3.27. This was the first programmable video game system ever to be released. When I say that this video game system was programmable, I mean that rather than all of the games being stored on the game machine itself, the video game programs were stored on yellow ROM cartridges. By changing cartridges, the player could change which game the Channel F gaming console would play. The first *video cart* available for this game console was called 4-in-1 because it had four games: Tic-Tac-Toe, Shooting Gallery, Doodle, and Quadra Doodle. For those not fortunate enough to be able to buy new video carts, the game console came with two games, hockey and tennis, built into the console itself.

Figure 3.25 A brochure for Fairchild's Channel F.

Revolutionary as this console was, it would not last long because it was outshone by the Atari VCS (Video Computer System).

Even though the Atari VCS outdid Channel F, it initially was not an overwhelming success. Most of the problems stemmed from within Atari itself. On the production end, there was a major problem with defective chips and cases. On the management end, the atmosphere inside of Atari was very laid back and relaxed. In 1978 Bushnell was pressured to leave Atari and Warner replaced the laid back atmosphere with a more rigid environment, one that was not as forgiving as the one Atari employees where accustomed to. In a very gutsy move, they also produced 800,000 VCS units, most of which became lost on warehouse shelves.

Figure 3.26 Channel F magazine ad.

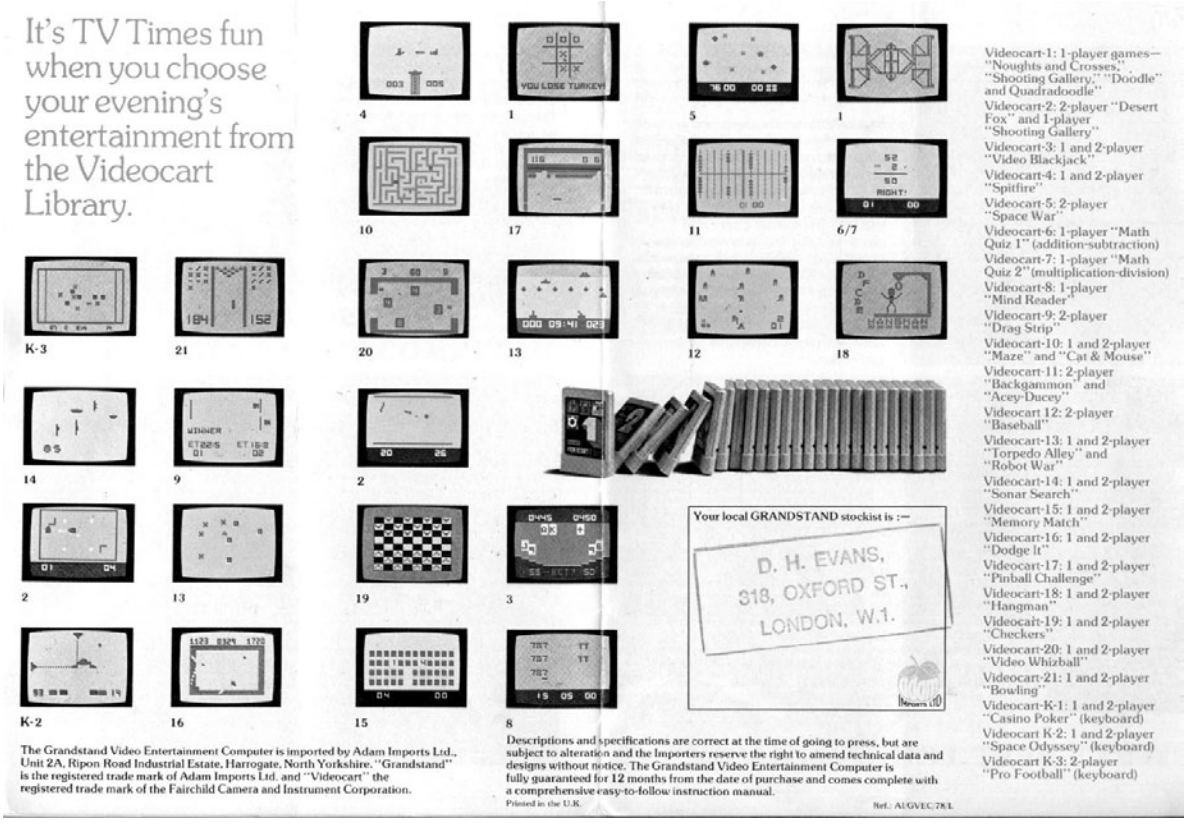


Figure 3.27 A British ad for Channel F, showing a number of games for the system.

Space Invaders

And then it happened. In 1980 Atari dropped a bomb on the gaming industry that would make all other technologies of the day useless and obsolete. Unlike most of Atari's accomplishments, this bomb was not necessarily a new creation and was not a work done by their own hand. The thing that would propel the VCS from a floundering machine to a mega success was a game called *Space Invaders*. When Atari licensed this game and released it for the VCS, people went crazy. They were hooked and could not free themselves from the overwhelming urge to play this game. Maybe it was the 112 different ways you could play the game. Maybe it was the intense two-player action. Maybe this game was just fun to play. The VCS machine began to sell so well that it went from being a drain on Warner's profits to producing over half of its profits. In the end, 200 games were produced for the machine by 40 manufacturers, and the VCS grossed over $25 million. Not a bad haul. See Figures 3.28 through 3.31.

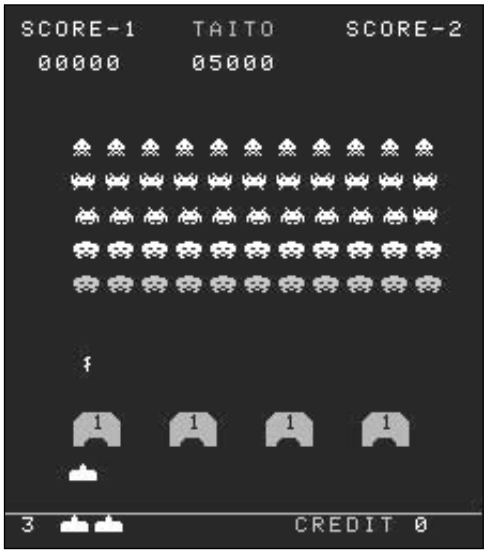


Figure 3.28 Screen shot of the mega hit game Space Invaders.

Figure 3.29 Atari VCS video game console.

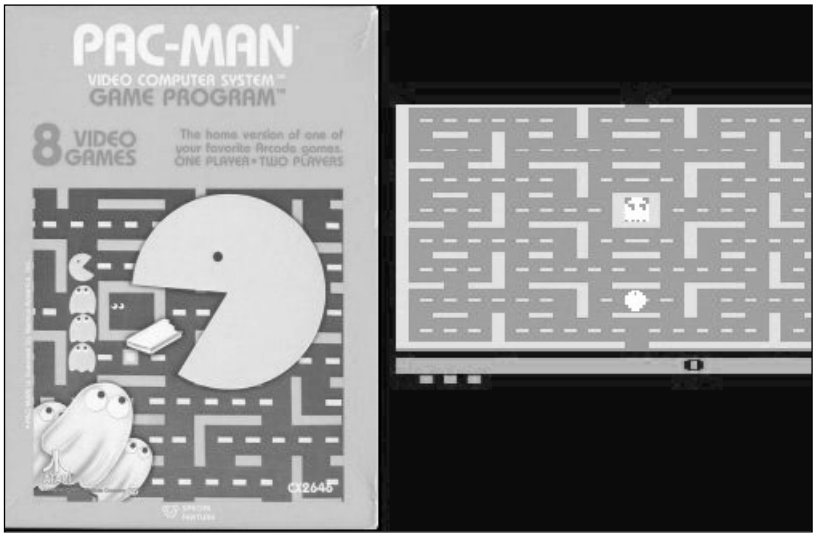


Figure 3.30 Pac-Man for the VCS.

Figure 3.31 Donkey Kong for the VCS.

Easter Eggs

Not everyone inside Atari was happy about their success. For example, designer Rick Mauer who was instrumental in porting *Space Invaders* over to the Atari was paid a total of $11,000 for his work. That is exactly 0.011 percent of the profit that the game made for Atari. Many other employees were unhappy over the policies of Atari, such as the practice of not giving credit to the actual programmers who made the games. Warren Robinette created a way around this with the first *Easter egg*. He actually hid his name inside of a game he created called Adventure for Gamers. If you did everything just right you could gain access to a hidden room where you would see the name of the author who created the game.

Third Party Software

Other programmers were not so subtle in expressing their resentment to not getting their due credits. In 1979, Larry Kaplan, Bob Whitehead, Alan Miller, and David Crane created their own video game company called Activision. They did not set out to make a competing game system but rather used their "inside" knowledge to become the first company to make third-party software for the Atari. They released a number of very good, very impressive games and because they placed the names of game authors prominently on the box, the game authors became minor celebrities, receiving in excess of 12,000 fan letters a week and even getting stopped in the streets for autographs.

In 1981, Imagic became the second company formed to create third-party content for the Atari. This time the founders of the new venture were former Atari and Mattel employees Bill Grubb, Brian Dougherty, and Denis Kobel. Denis Kobel was one of the first people to be employed by Atari. Other talented members of the company included Rob Fulop who

was the creator of a number of hits for the VCS such as Night Driver and the infamous Missile Command and Bob Smith who was the creator of Video Pinball, also for the VCS. Despite the talent and experience, the new company did not fare as well as its counterpart Activision. The new company released only 20 games, including the top selling Demon Attack before the great video game crash of 1983.

Activision survived the crash but just barely. At the same time the video game industry was crashing, the home computer market was booming. Activision changed its name to Mediagenic and released the first entertainment CD ROM in history, "The Manhole," which was designed for the Apple Macintosh.

The great crash of 1983 claimed many casualties, but a few of the more creative companies survived. Probably the most notable of these was Control Video Corporation. They had the ingenious idea of starting an online service called Gameline, which allowed gamers who owned the Atari VCS to download games over their phone lines. The company was founded by William F. Von Meister who had founded the world's first "online" service, called The Source. Players paid a one-time setup fee of $15 for the service. After that, they would pay either 10 cents a game or $1 an hour for each hour that they wanted to play a game.

William F. Von Meister had a much bigger design for his invention. He wanted to create a full-fledged Bulletin Board System with e-mail, home banking, news, and more. This did not pan out, because agreements could not be arranged with many top game developers. This meant that some of the best games that were set to be released on the service never saw the light of day. Finally, this venture bit the dust like so many other great ideas during the great video game crash.

During the crash, Von Meister was forced out of his own company and the company changed its name to Quantum Computer Services. This company would also suffer the fate of Control Video Corporation, but in 1989, AOL was born and became a billion dollar success. Unfortunately, until his death at the age of 53 in 1995 Von Meister would not reap one dime from this company, which was a direct result of his brainchild.

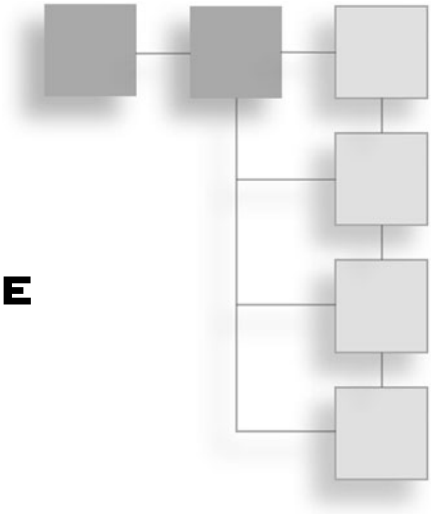
Conclusion

They say that history is written by those who survive to tell it. Indeed the game industry has its share of both survivors and casualties, all of whom have their opinions as to how the history of video games really evolved. This chapter has merely scratched the surface. The goal here was to give the newcomer to the world of retro game programming a glimpse into the early history of game design that laid the foundation for building the games that you will learn to program in this book.

Without Ralph Baer's contributions, there would probably be no game industry and without Steve Wozniak and Steve Jobs there would be no home computer market. At the very least, without these our computer and game industry would be nothing like what they are today. Every single person who built, designed, marketed, sold, and played a video game during these early years played a part in making the game and computer industry what they are today. Without the series of events you just read about, there would be no game industry today.

CHAPTER 4

Assembly Language



All truths are easy to understand once they are discovered; the point is to discover them.

Galileo Galilei (1564–1642)

They say that man fears what he does not understand and hates what he cannot conquer. That means that if you learn to understand something you will lose your fear of it. This allows you to conquer it and come to love it. This is the mystery behind the mastery of assembly language. The first thing that we must do is to learn to understand exactly what assembly language is.

Understanding Assembly Language

Every computer spends 99.99 percent of its time moving binary bits from one location to another. When you look at your screen, it seems to be one solid image. If you use a magnifying glass to take a closer look, you will see that this solid image is actually made up of a series of very small dots placed side by side to give the appearance of a solid object (see Figure 4.1).

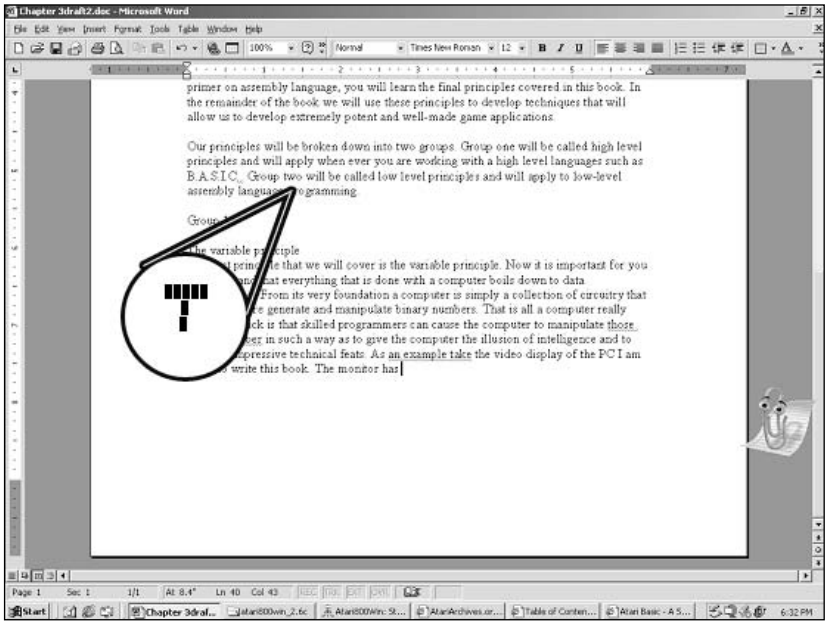


Figure 4.1 The screen is made up of many small dots placed side by side.

Moving Memory Around in Your Computer

When you type a letter or number on your keyboard, somehow the combination of dots on the screen mysteriously rearranges to form an image of the letter or number that you just pressed on the keyboard. How does this happen?

Each of those dots on the screen is stored in your computer's memory as a collection of bits. The area of memory where these bits are sorted is called *video memory*.

When you press a key on your keyboard, a group of eight bits used to represent that key is placed into a portion of memory called the *keyboard buffer*.

In order for a letter pressed on your keyboard to appear onscreen, the byte representing that character has to be moved from the keyboard buffer to the video buffer.

Before we can start moving data around, however, we need access to some information. We need to know where the keyboard buffer and the video buffer are stored in memory. We further need to know what part of video memory we must write to in order to put an image on a specific part of the screen.

As you will learn in Chapter 5, "A Game Graphics Primer," a memory map (see Figure 4.2) is used to tell us where the location of both video memory and the keyboard buffer are in the computer's memory.

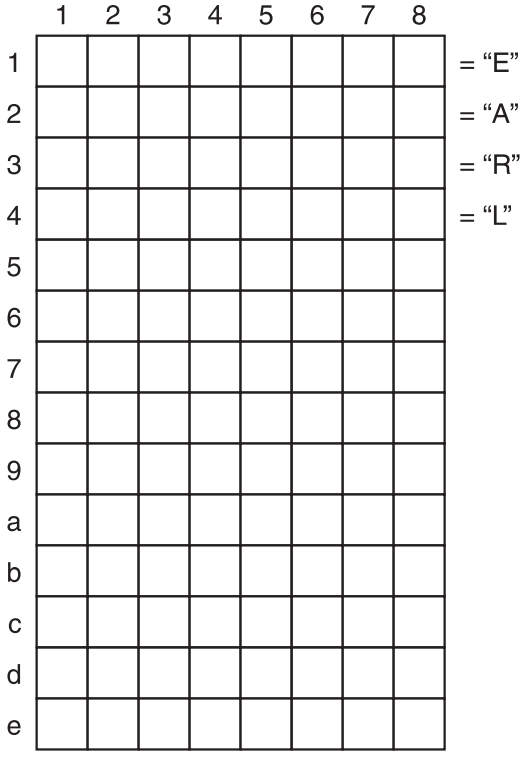


Figure 4.2 The memory map of an imaginary computer.

On most computer systems, the first byte of video memory represents the top-left corner of the screen. If you are going to place the key that we just pressed onto the top-left corner of your screen, you have to move the byte of data stored in the keyboard buffer to the first byte of video memory. (See Figure 4.3.)

If you could write a program to do this in plain English it would look something like this:

1. Read the data stored in the keyboard buffer (memory location xxx).
2. Store the data you just read into the first byte of video memory (memory location xxx).

This gives you a basic idea of what you have to do: read the data stored in the memory buffer and write it to the video buffer. This program, however, is not specific enough. What happens between your telling the computer to read the keyboard buffer and writing it to video memory? After the computer has read the keyboard buffer, it has to store that information somewhere until you give it the next instruction. Fortunately, every computer comes with built-in containers called *registers* that you can use to store information.

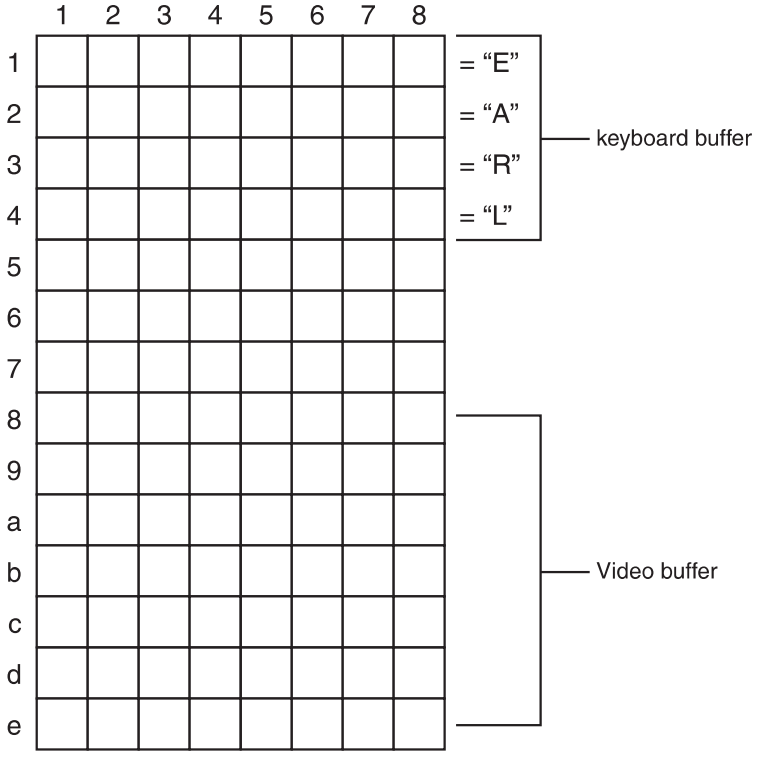


Figure 4.3 Move the data in the keyboard buffer to the first byte of the video buffer.

When you program in BASIC, you are introduced to the concept of a *variable*. A BASIC programmer creates these variables when he needs them, and he can destroy them at any time. Registers are like variables, except they are not created by the programmer and cannot be destroyed by the programmer. These register variables are built into the hardware of your computer's CPU. While we use variables in Assembly, it is much more common for us to work with registers. With this new knowledge you need to rewrite your program.

1. Move the data stored in the keyboard buffer (memory location xxx) to register A (also called the accumulator).
2. Move the data stored in the accumulator (register A) into the first byte of video memory (memory location xxx).

That's more like it; this program is very simple, and it does the job. There is still one problem with this though. We asked the computer to do two things, but we had to use 36 words. With this kind of ratio, if we had a program that asked the computer to perform 1,000 actions, we would have to write 18,000 words! This simply will not do. What if we

could reduce the number of words by using an abbreviation for these sentences? Look at the following program.

```
LDA     xxx
STA     xxx
```

Wow, we just eliminated 32 words! The first line starts with three letters. These letters (LDA) stand for Load Accumulator and basically tell the computer to store the memory address after the LDA instruction in the accumulator register. The second line starts with another three letters (STA) that stand for Store the accumulator. This instruction tells the computer to store the contents of the accumulator register into the memory location that follows this instruction.

Guess what? You just wrote your first assembly language program, and it wasn't all that complicated at all.

The important thing to remember is that if you can visualize what you want your program to do and write the program in plain English, then all you have to do is find the instructions that act as abbreviations for those sentences in your program. Eventually you will have the assembly language code that you need for your program.

There are many abbreviations found in the Assembly language. Each abbreviation is called an instruction. There are instructions that are used to move memory around, perform mathematical operations, and carry out other basic operations.

Most instructions are broken into two parts called the *Opcode* and the *Operand*. In the example above, for example, `LDA` would be the Opcode while `xxx` would be the Operand.

The opcode (which is short for operation code) tells the computer which operation to carry out. The operand tells the computer which memory location to perform that location on.

Within the small program that we wrote, the `LDA` command is the opcode that told the computer to move a memory location into the accumulator. `Xxx` is the operand that tells the computer which memory location to move to the accumulator. Likewise, the `STA` command is an opcode that tells the computer to store the accumulator in a memory location. `Xxx` is the operand that tells the computer where to store the data.

What you have learned so far is enough to give you a general idea of how Assembly works. We now have to flesh out this idea with specific details. The first detail we need to uncover is the concept of addressing.

Understanding Numbers and Math in Assembly Language

Once we have mastered the art of moving data around inside of our computer we must begin the task of learning to manipulate this data. We do this by performing a number of common mathematical operations of the data such as addition, subtraction, multiplication,

division, and others. These operations are performed in assembly the same way as we have performed them all of our lives with a few exceptions. These exceptions make Assembly based mathematics very scary and confusing until you get the hang of them. Once you understand how they work however you will find them to be very simple concepts.

It is now time for us to fully understand the way that your computer works with numbers.

Decimal, Binary and Hex Numbers

The numbering system that we have been taught since we were children is called decimal.

Decimal Numbering System

The decimal numbering system is also called *base 10* because it is composed of only 10 unique digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. (You are probably familiar with these concepts but read on anyway; it is important for us to review these points before we can move on.) Every single number that we can ever write in decimal is made using combinations of these 10 digits arranged in a horizontal line. So, for example, the number 6798 is made up of the numbers 6, 7, 9, and 8. Each of these numbers is placed in its own column. (See Figure 4.4.)

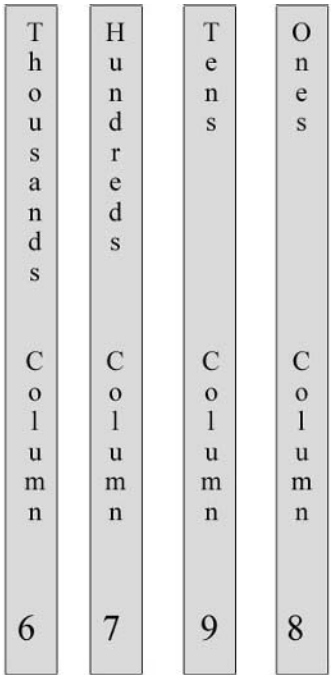
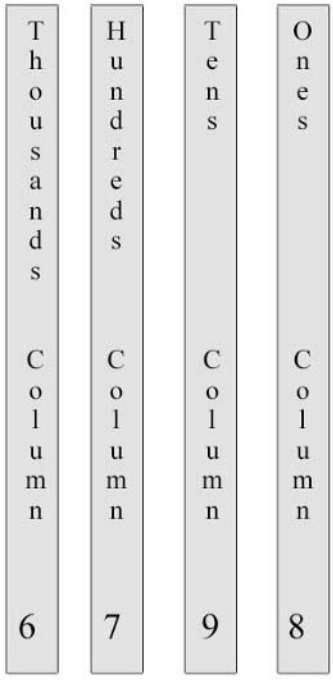


Figure 4.4 Illustration of each number in its own column.

Each column represents a multiple of 10, which is 10 times larger than the multiple to its right. (See Figure 4.5.)




```
6 X 1000   =   6000
7 X 100    =   700
9 X 10     =   90
8 X 1      =   1

6000 + 700 + 90 + 1 = 6,798
```

Figure 4.5 Illustrations of each number in its own column that is a multiple of 10.

This means that we can also represent this number like this.

$(6 \times 1000) + (7 \times 100) + (9 \times 10) + (8 \times 1) = 6798$

The equation above says that if we multiply 6 times 1000 we will get 6000. If we multiply 7 times, 100 we will get 700. If we multiply 9 times 10, we will get 90 and 8 times 1 = 8. Finally, 6000 + 700 + 90 + 8 = 6798.

This rule holds true with any number. (See Figure 4.6.)

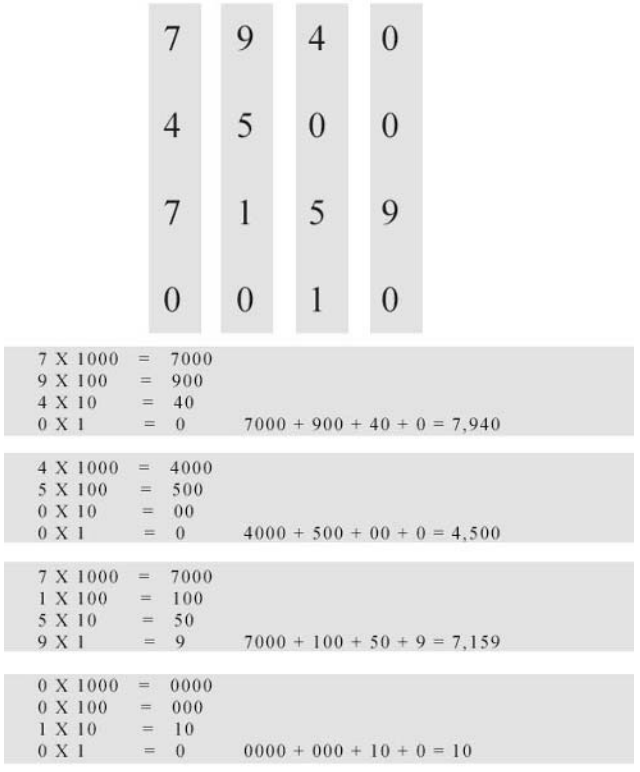


Figure 4.6 Illustration of some numbers and their breakdown.

Decimal Addition and Subtraction

Once we understand how the decimal system works we can not only identify decimal numbers, but we can also learn to add and subtract them.

$$\begin{array}{r} 10 \\ +10 \\ \hline 20 \end{array}$$

In the example above, the first column has two 0s, which are to be added together. This, of course, gives us 0, which we place at the bottom of the column.

The second column has two ones that are to be added together to equal 2, which we place at the bottom of the column. This gives us the number 20. $((2 \times 10) + 0)$.

$$\begin{array}{r} 11 \\ +19 \\ \hline 30 \end{array}$$

Here we have a 9 and a 1 in the first column. We know that 9 + 1 equals 10. The problem is that the number 10 has two digits and we can only place one digit in our column. We solve this problem by placing the 0 at the bottom of the first column and putting the 1 at the top of the second column.

```
  1
 11
+19
  0
```

We now have three 1s in the second column that we have to add together.

1 + 1 + 1 = 3.

We place 3 at the bottom of the first column.

```
  1
 11
+19
 30
```

This is how 11 plus 19 come together to equal 30.

A similar process takes place when we do subtraction. Let's see what happens when we subtract 11 from 22.

```
 22
–11
```

In the first column, we are subtracting 2 from 1. This leaves a value of 1, which we place at the bottom of the first column. The exact same thing happens in the second column, so we end up with a 1 at the bottom of each column. 22 –11 = 11.

We have to do things a bit differently to subtract 19 from 22.

```
 22
–19
```

In the first column, we need to subtract 9 from 2, which is something that we cannot do. To solve this problem, we borrow 1 from the second column and combine it with the two in the first column to make 12.

```
  1  12
 -1   9
```

Now we are subtracting 9 from 12, which we can do. The answer is 3, which we place at the bottom of the first column.

In the second column, we subtract 1 from 1, which equals 0. We place this value at the bottom of the first column. 22–19 equals 3.

Now that we have reviewed basic math in decimal, we can learn binary and learn to do these same functions in binary.

Binary Number System

Your computer does not understand decimal numbers. The computer only understands binary numbers. Binary numbers form a system of numbers called *base two*. It is called base two because the system only has two unique numbers: 0 and 1. Every single number that our computer will ever use, no matter how large, is expressed with a combination of these two numbers. This is not as impossible as it may sound and works in almost exactly the same way that decimal is able to represent an infinite amount of numbers using only 10 unique digits.

In decimal, the number 11111 breaks down as shown in Figure 4.7.

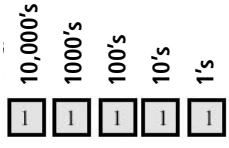


Figure 4.7 Illustration of the breakdown of the number 11111.

In binary, this exact same number breaks down differently. Whereas in decimal, each column is a multiple of 10 that is progressively one digit longer than the previous column, in binary each column is a multiple of 2, which is one power greater than the previous column. This means that the number 11111 will break down as shown in Figure 4.8.

2 to the 0 power equals 1. This means that the first column is equal to 1×1.

The second column represents 2 to the second power, which equals 2. Because we have a 1 in this column, the value of this column is 1 times 2, which equals 2.

The third column represents 2 to the third power, which equals 4. Because we have a 1 in this column, the value of the column is 1 times 4, which equals 4.

The fourth column represents 2 to the fourth power, which equals 8. Because we have a 1 in this column, the value of this column is 8.

The fifth column represents 2 to the fifth power, which equals 16. Because we have a 1 in this column the value of this column is 16.

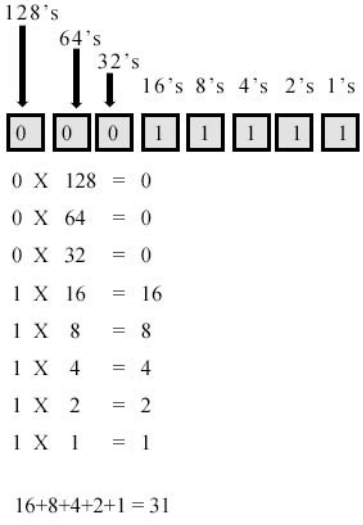


Figure 4.8 Illustration of the way 011111 breaks down in binary.

Finally, the sixth column represents 2 to the sixth power, which equals 32. Because we have a 0 in this column, the value of this column is 32 times 0, which equals 0.

In binary, the number 011111 equals 1 +2 +4 +8 +16 +0, which equals 31.Just as in decimal, this rule works with any binary number. (See Figure 4.9.)

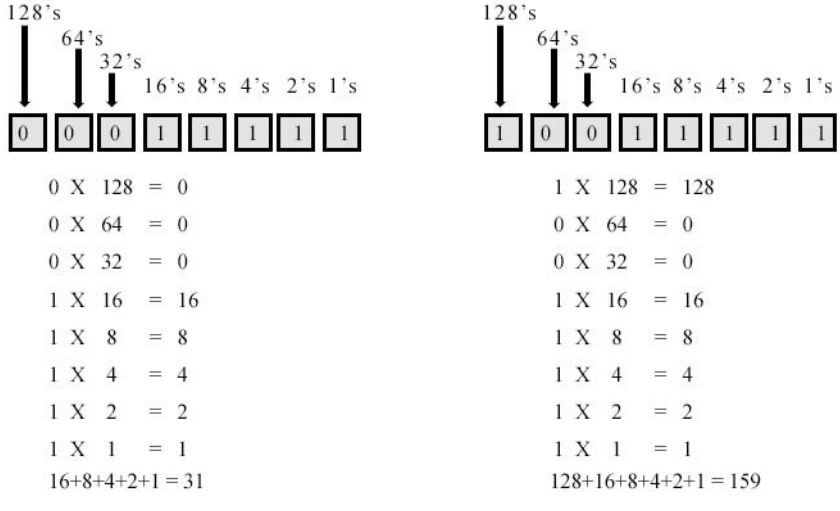


Figure 4.9 Illustration of various binary numbers broken down.

Binary Addition and Subtraction

Binary addition and subtraction work using the same principles as decimal addition and subtraction, which is why it was important for us to review them earlier. The only difference is that we only have two digits to work with. Let's look at our first example:

```
  01
+10
```

In the first column we are adding 1 + 0, which of course equals 1, which we place at the bottom of this column.

```
  01
+10
   1
```

In the second column we are also adding a 0 and a 1, so we place a 1 at the bottom of the second column.

```
  01
+10
  11
```

This gives us a value of 11, which, as we learned earlier, equals 3. As you can see, even though we are working in binary, 1 plus 2 still equals 3!

Now let's change things up a bit by adding 01 to 01.

```
  01
+01
```

In the first column, we are adding 1 + 1. In decimal, this would be equal to 2, which we would put at the bottom of the first column. In binary, 1 + 1 is still equal to 2 only we represent it as 10. We now carry the 1 just like we did in decimal and place the 0 at the bottom of the first column.

```
   1
  01
+01
   0
```

Now we are going to add 0 + 0 + 1, which of course equals 1, which we place at the bottom of the second column.

```
  01
+01
  10
```

Our final answer is 10, which is the binary equivalent of 2. Even in binary, even with a carry, 1 + 1 still equals 2!

The Hexadecimal System

Look at the two following numbers and tell me the difference between them in less than two seconds. Go.

```
01110000100011110001011111001101011110101010
01110000100011110001011110001101011110101010
```

Most people could not do what I just asked you to do and that includes me. After looking at it for a while, we would eventually see the difference. The fact is that humans do not think very well in binary. This poses a problem because while humans understand decimal, the computer cannot, and while the computer understands binary, humans do not. "What we've got here is a failure to communicate." In order to solve this we need to find some middle ground.

Those long numbers are made up of 44 digits. What if we could reduce them to only 11 digits? That would make our lives a whole lot easier; lets see how we can do this.

If we divide 44 by 11 we get the number 4. Lets break these numbers into 11 groups of 4.

```
0111 0000 1000 1111 0001 0111 1100 1101 0111 1010 1010
0111 0000 1000 1111 0001 0111 1000 1101 0111 1010 1010
```

If I asked you to instantly find the difference between these two numbers now, you would probably find it much more quickly so we know that we are moving in a positive direction.

In reality, however, we have not yet accomplished our goal because we are still thinking in binary. What we need to do is use this technique to create another base. A combination of 4 bits can be used to produce 16 different combinations of bits. That means the most logical base for us to work with would be a base of 16.

So far, the largest base that we have worked with is base 10, which had 10 unique digits (0 to 9), but this has six digits less than it would take to represent our base 16 numbering system. We need to find some substitutes. Whenever scientists run out of numbers, they

A Failure to Communicate

Just a bit of movie trivia: The quote I used above came from Stuart Rosenberg's 1967 film, *Cool Hand Luke*, which was one of the great American films of the 1960s and arguably Paul Newman's greatest performance. The words quoted were spoken by the captain of Road Prison 36, played by Strother Martin. The full quote is as follows.

"What we've got here is a failure to communicate. Some men you just can't reach, so you get what we had here last week, which is the way he wants it. Well, he gets it. And I don't like it any more than you men."

Paul Newman also uses this famous line later in the movie to mock the prison guards.

usually start using letters. We will do the same thing, so we will use the first 6 letters of the alphabet to represent the last six digits of our base 16 numbering system.

Base 2 systems use the digits 1 and 2. Base 10 systems use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Our new base 16 system uses the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f. This new base 16 system is called *hexadecimal* and is what we will be using while we work with Assembly.

Figure 4.10 illustrates decimal, hexadecimal, and binary equivalents.

Using this table, we can now convert the two large decimal numbers above to hexadecimal.

```
0111 0000 1000 1111 0001 0100 1100 1100 0111 1010 1010 = 708c17cc7aa
0111 0000 1000 1111 0001 0100 1000 1100 0111 1010 1010 = 708c17c87aa
```

As you can see, working with hexadecimal numbers is much easier than working with binary numbers.

Decimal Numbers	Hexadecimal Numbers
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14

Figure 4.10 Illustration of decimal hexadecimal equivalents.

Base 16 follows all of the patterns that we have used when working with base 2 and base 10. In every number base, we divide each digit into its own row moving from right to left. To find the value of the first column, we multiply the number in that column by the base number to the 0 power. So in base 2, the value in the column is multiplied by 2 to the 0 power. In base 10, the value in the column is multiplied by 10 to the 0 power. In base 16, the value in the column is multiplied by 16 to the 0 power. Because anything to the 0 power equals 1, the value of the first column will always be equal to the actual value in that column. Figure 4.11 shows the arrangement of the columns for base 2, 10, and 16.

Addressing Modes

As you just read, most instructions that you give to your computer are made of two parts: the opcode and the operand. We now know that the operand tells the computer what address we are working with. What we have not discussed is the fact that operands can use different kinds of addressing. Addressing modes are sort of like driving instructions. Look at the map in Figure 4.12.

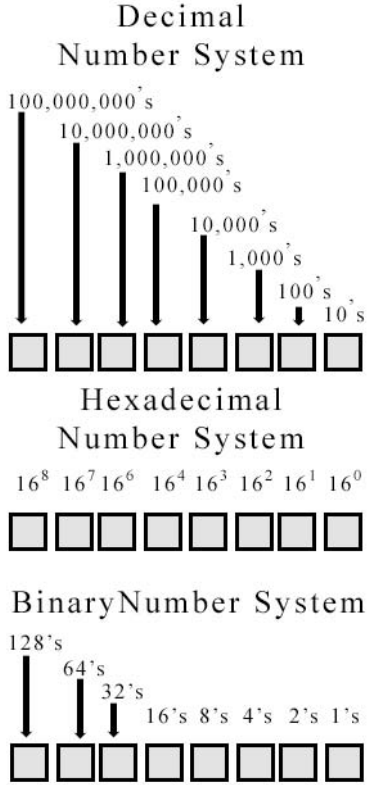


Figure 4.11 Table for conversions.

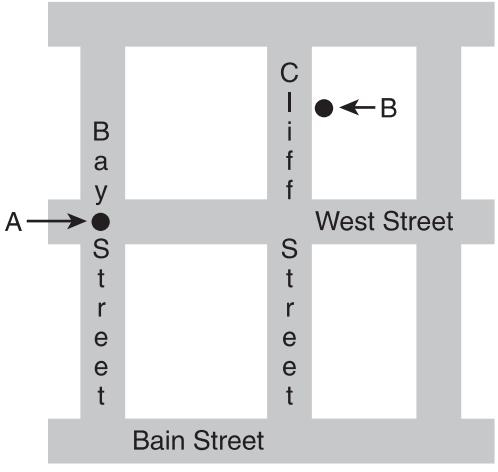


Figure 4.12 Small map of a city.

What if you were at point A on the map and you needed to get to point B? You are not familiar with the city, so you have to ask for directions. If you ask 13 different people for directions, you will probably get 13 different sets of directions that will all lead you to the same place. One person might say, "Go straight down West Street and take a left; you'll see it on the right." Another person may say go south on Bay Street, turn east on Bain Street, turn north on Cliff Street, and drive past West Street and you are there." Twelve other people may give you eight other routes that will take you to point B. You can think of these kinds of directions as *addressing modes.*

Using the LDA instruction (Load Accumulator) as an example, there are a few different ways to tell the computer which memory location to move into the accumulator. Look at the memory map in Figure 4.13 (our virtual city).

Here we see our familiar memory map showing the location of video memory and the keyboard buffer. In our example, we used the following command to load the keyboard buffer into the accumulator.

```
LDA xxx
```

This is one kind of addressing. In the same way that there was more than one way for someone to direct us from point A to point B, there are a number of ways for us to direct the computer to memory location xxx. Rather than giving the computer the exact instruction, we could have done this:

```
LDA xxx + x
```

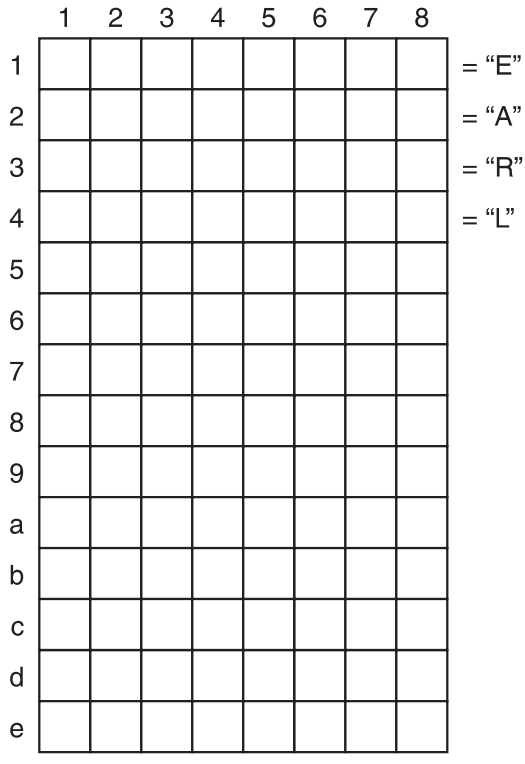


Figure 4.13 Memory map of a computer.

or this:

```
LDA xxx + y
```

or this:

```
LDA     xx
```

Each of these is a different kind of addressing, and each one is very useful. Before you can use these addressing modes, however, you need to understand how they work. There are 13 kinds of addressing modes available on retro game machines. These modes are discussed in the following sections.

Immediate Addressing

This is probably the most obvious form of addressing; if you want to add eight to the value held in the accumulator, then you will use 8 as the operand for the `ADC` command:

```
ADC #08
```

Notice the number sign that precedes the number 8. This tells the computer that you want it to use immediate addressing when interpreting the operand.

Absolute Addressing

The next most common form of addressing is called absolute addressing. With this form of addressing, rather than using the actual number that you want to add as the operand for your command, you use the address of a location that holds the number you want to add. So if a value of 8 is stored at the memory location `$2f23` and you want to add this value stored at this memory location, you would use this address as your operand:

```
ADC     $2f23
```

Notice that you did not place a number sign in front of the operand, and the operand contains four digits. This way the computer knows that it should use absolute addressing to interpret the operand, so rather than adding the value `$2f23` to what is in the accumulator, it will go to memory location `$2f23` where the value 8 is stored, and it will add that value to the accumulator.

Zero Page Addressing

When you use absolute addressing, you use a 16-bit memory address. Because you use the full 16 bits, you can use absolute addressing to access any part of the computer's memory space. There is a special portion of memory that you do not need to use a full 16-bit address to access. This area is called the *zero page*, and it is located between address `$00` and `$FF`. Because the largest memory address in the zero page is `$FF` and `$FF` can fit into a single byte, one byte is all that is needed to access any memory location in the zero page.

```
ADC $F0
```

Implied Addressing

Unlike other addressing modes, instructions that use implied addressing do not need to be given an operand.

An example of this kind of instruction is the `TAX` command. Unfortunately, this is not a command that is going to automatically do your taxes for you. `TAX` is actually short for Transfer A to X, and what this command does is move data stored in the A register to the X register. It will always move data from the A register to the X register, so we never need to give it an operand.

Accumulator Addressing

This is simply another form of implied addressing. The difference between this and regular applied addressing is that the target of this action is always the accumulator.

Indirect Absolute Addressing

This is the name of the addressing method used by the `JMP` command. The memory address that we give this command is used as a reference to the low part of a 16-bit address that contains the address of the next instruction we want to execute.

```
JMP ($1167)
```

In the preceding example, the computer would treat `$1167` as the low end of the address and use `$1168` as the high end of the address containing the instruction we want to execute. So if `$1167` contains `$EA` and `$1168` contains `$12`, then the next instruction executed will be `$12EA`.

Absolute Indexed Addressing

This form of addressing is interesting and can be useful for things like looping through a portion of memory. It does this by not only taking a 16-bit memory address, but also by taking the value held in either the x or the y register, and adding them together in order to get the final address:

```
LDA $E367,Y
```

So if you want to perform an action on the memory location `$E367` and the 10 bytes that follow it, you would do something similar to this:

```
LDA $E367
Perform processing of data in the accumulator
INY
```

This code between the `LDA` command and the `INY` command would include the code used to perform the operation that we want repeated 10 times and code that keeps track of the loop so that we can break out of the loop after 10 cycles.

Zero Page Indirect Addressing

This form of addressing works just like regular indirect addressing. The only difference between this and normal indirect addressing is that here we are referencing memory locations stored in the zero page. As with all forms of zero page addressing, we only need to use an 8-bit memory address to reference data as opposed to 16 bits. This means that performing zero page indirect addressing is faster than normal indirect addressing.

Indexed Indirect Addressing

```
LDA ($E4,Y)
```

This is an example of indexed indirect addressing. If we assume that the Y register contains a value of 4, then the `LDA` command would add `$E4 + 4` to get a value of `$E8` for the low end of the address that should move into the accumulator.

Indirect Indexed Addressing

This one is similar to indexed indirect addressing but is slightly trickier. Here is an example

```
LDA     ($B4), x
```

In this example, the computer first uses `$B4` as the low end of a memory address and would use `$B5` as the high end of the address. If `$B$` contains `$FF` and `$B5` contains `$A4`, then the computer will retrieve the value `$12F4`. This value will then be added to the value in `X`, so if `X` has a value of 3, then the final value the computer will use will be `$12F8`. This value will be used as the address of the data that will be placed into the accumulator.

Relative Addressing

This is the kind of addressing used by branch instructions. Unlike the other forms of addressing that give the computer a specific address to go to, when we use this form of addressing we tell the computer to move a certain distance away from the current instruction.

Working with the Stack

There will be times when you will need to preserve the contents of a memory location or a CPU register. The computer stack was made specifically for this purpose. Let's say your computer has three registers called `A`, `X`, and `Y`. I chose these letters because they are actually very common names given to registers.

What if you have very important information stored in these registers but need to use them to do something else without losing your important data? This is where the stack comes into play. You can push these registers onto the stack to save the contents of your register, use the registers for whatever you have to do, and then pull the original contents of the registers from the stack and back into the registers. As you can imagine, this is a very important technique and you will make use of it often.

Different CPUs will perform pushes and pulls differently. Some use a single push and pull command, which, when activated, will store all of the registers and flags to the stack and pull them off at the same time. Other CPUs require you to push each register or flag to the stack independently and pull them off of the stack again when you are ready to use them. When you push data onto the stack, you must remember that the first piece of data pushed onto the stack has to be the last piece of data that you pull from the stack.

Think of it as stacking several books one on top of the other on a table. The first book that you put on the table will be at the bottom of the stack, while the last book you put on the table will be on the top. If you start taking books off the stack one at a time, you have to take the last book you placed in the stack from the top, followed by the second to the last book you put on the table, and so on, until you have moved every book from the table. (See Figure 4.14.)

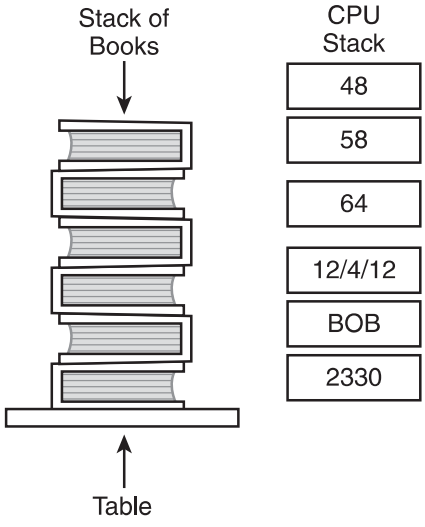


Figure 4.14 A stack of books compared to the computer's stack.

System Flags

System flags are similar to registers except that they only contain one bit. Depending on whether they are on or off, you can learn things about the state of your computer system, or you can find out whether certain events have occurred. For example, when you perform an addition, it is important for you to know if there was a carry. If a carry occurred, this flag would be set to 1 and if no carry occurred, the flag will be set to 0. Table 4.1 contains some common registers.

Table 4.1 Common System Flags

Name of flag	Action Taken
N	NEGATIVE: This is set if bit 7 of the accumulator is set.
V	OVERFLOW: Set after an addition of two numbers with the same sign or subtraction of two numbers with different signs if the result is larger than 127 or less than –128.
B	BRK Command: When an interrupt occurs if it was caused by a BRK command, then this flag is set. If the interrupt was an external interrupt, then this flag is not set.
D	DECIMAL MODE: This is set if the CPU is currently in Binary Decimal mode.
I	IRQ DISABLED: If mask able interrupts are disabled, then this is set.
Z	ZERO: If the result of any operation is zero, then this flag is set.
C	CARRY: This is used to tell us if a carry was produced after an addition or if a borrow was produced by a subtraction and to hold a bit after logical shift.

Logic and Branching Instructions

We have covered enough theory to perform the majority of actions you will do while programming in Assembly. You can move data from one location to the next using various addressing modes. You can add data, subtract data, and even multiply and divide data. Now you need to learn how to control if and when the computer does these things.

While there are many situations where you will have to use logic in your game, we will choose a very common yet simple example to understand how these decisions are made. What if you create a game that is very long? A player plays diligently and reaches the last level before he has to go to school. If you don't make special arrangements, that player will have to start playing the game all over again from the beginning when he returns. This would be a major turn off. So the question becomes what special arrangements should be made? The simplest solution is for you to give the player a password every time he reaches a new level. This password would allow the player to come home from school and continue playing right where he left off.

In order to implement this feature, you must have the player enter his password. Next you will have to see if this password matches the password for any of the levels in your game. If it does match a level, then you need to load that level so that the player can start playing. Here is a plain English list of the actions that need to be performed.

1. Get password from the player.
2. See if it matches the password for any of the levels.
3. If it matches the password of a level, load that level.
4. If it does not match, go back to number 1.

This is a pretty straightforward program. Once you get the password from the player, you test it and either load the appropriate level or ask the player for the password again. If the player continues to enter incorrect passwords, the computer will continue to ask for a new one until he enters a valid password.

This program is a *memory map* with the plain English program loaded into memory. (We are not interested in actual assembly instructions right now, just the concept of how logic and branch instructions work. It is for this reason that the memory map contains English instructions. This will never happen in an actual program.)

As you can see, this program is broken up into blocks of instructions. Each block is responsible for carrying out specific actions. One block of instructions gets the password from the user. Another block checks to see if the password matches the one required for any of the levels. Finally, several blocks are used to load specific levels. This program seems pretty organized, right? It should work, right? Wrong!

If you ran this program, it would get a password from the user, check to see if it matches the password of any of the levels, and then, whether it was a correct password or not, the

program would load every level in the game, one after another. This is because the computer will normally start executing the first command in the program and execute them all, one by one, going down the list until it has executed the last instruction.

This will not do. You need more control over your program. This is where logic and branch instructions come into play. They give you the control that you need over your program.

Let's look at some of the decisions you will have to make. First of all, the program should not automatically start checking passwords when the player presses Enter. First of all, the program needs to check to see if the player even entered a password and, if he did, whether the password was of the correct length. This means that you need to add another block to your program that tests to see if the player typed anything.

Here is how this will work. When the player enters a password, the password is placed into a memory location until you are ready for it. You will call this memory location the *password buffer*. Before the player enters a password, this memory location contains only zeros. Before you have the computer start testing passwords, you need to test this memory location to see if it contains a value greater than zero. If it does, you check for passwords; if not, you ask the player for a password again. Here is how this part of the program will work.

1. Get password from the player and store it in the password buffer.
2. Test to see if the password buffer has a value greater than zero.
3. If the password buffer equals zero then go to number 1.
4. If the password buffer is greater than zero, jump to memory location `xxx`.

From the memory map above, we see that location `xxx` is the portion of memory that contains the code to test for a correct password. If the password buffer equals zero, the player did not enter a password and he must be prompted to give the password again. If not, the program starts testing for passwords.

Next, we need to create the logic that is going to test the password to see if it is correct. The easiest way to do this is to go down the list of passwords and if the password matches that of any level, jump to that level. If you end up at the bottom of the list and no password matched, then the program jumps back to the block of memory with the code used to get a new password from the user. This part of your program would look like this:

1. If password equals to `qwedjg` then go to memory location `xxx`.
2. If password equals to `lgkjhn` then go to memory location `xxx`.
3. If password equals to `lfjgke` then go to memory location `xxx`.
4. If password equals to `vnghdu` then go to memory location `xxx`.
5. If password equals to `qogmsj` then go to memory location `xxx`.

6. If password equals to `cmgjke` then go to memory location `xxx`.
7. If password equals to `mbfhgk` then go to memory location `xxx`.
8. If password equals to `mchfke` then go to memory location `xxx`.
9. If password equals to `spzmlg` then go to memory location `xxx`.
10. If password equals to `qwedjg` then go to memory location `xxx`.
11. Go to `xxx`.

If the player's password matches any of the passwords on your list, the computer will jump to the memory location that contains the code to load the correct level. The only way that your program will ever reach line number 11 is if the player's password did not match any of the passwords for the levels. If that happens, you have the program automatically jump to the memory location with the code used to get a new password.

The final blocks of memory are out of the scope of this section so we will not go into them in detail. Suffice it to say that they are used to load a given level.

You now have all of the pieces that you need to add this password save feature to your program. As you can see, with the use of logic/branch instruction, we can intelligently add much functionality to our games.

Facing the Code

Do you remember the motto from the beginning of this chapter?

"Man fears what he does not understand and hates what he cannot conquer. That means that if you learn to understand something then you will lose your fear of it, which allows you to conquer it and come to love it. This is the mystery behind the mastery of assembly language."

You should now understand the basic idea of how assembly language programming works. Now that you understand assembly language you can face its code and lose your fear of it. In order for us to unlock the full power of the machines covered in this book we will have to use three flavors of assembly; 6502 assembly, sweet 16 assembly, and finally 6809 assembly.

We will now begin to look at actual assembly language instructions. There is nothing for you to fear here. Remember, when you need to make an assembly language program, you first write out the instructions you need to give to the computer and then you convert these instructions to assembly opcodes and operands, which are really just abbreviations for the sentences you just wrote. There are three basic things you will have to know about each assembly language environment; what registers are available, what the instruction set is, and how memory is arranged. Most 8-bit computers have similar registers so you should never really be completely lost about these. The instruction set is simply a list of

every command that the CPU recognizes. The computer's memory map will vary from computer system to computer system. The memory map for the machines used in this book can be found on the companion Web site to this book.

6502 Programming

The 6502 processor was a beautifully crafted work of art for its time and appears in many groundbreaking computer systems such as the Apple II, Commodore 64, and even the original Nintendo Entertainment System (NES for short).

CPU Registers

There are three registers on the 6502 microprocessor. These are the accumulator register, the x register, and the y register. The most important register is the accumulator. This is because the functions used for addition, subtraction, and handling comparisons are all designed to use this register automatically. The x and y registers are general purpose registers. They can be used to hold data or as the offset to a given memory location.

Instruction Set

In order for you to communicate with your computer, you need to know which instructions your computer can understand. Most of the instructions that you use to communicate with your computer fall into several different categories. Tables 4.2 through 4.15 list each instruction that you use to communicate with any computer using the 6502 processor. They are grouped by category for your convenience.

Table 4.2 Instructions to Load a Memory Location into a Register

Name of instruction	Action taken
LDA	Load the Accumulator
LDX	Load the x register
LDY	Load the y register

Table 4.3 Store Information in the Registers to a Memory Location

Name of command	Action taken
STA	Store the Accumulator
STX	Store the x register
STY	Store the y register

Table 4.4 Increment Instructions

Name of command	Action taken
INC	Increment memory by one
INX	Increment x by one
INY	Increment y by one

Table 4.5 Decrement Instructions

Name of command	Action taken
DEC	Decrement memory by one
DEX	Decrement x by one
DEY	Decrement y by one

Table 4.6 Branch Instructions

Name of instruction	Action taken
JMP	Jump to another location
BCC	Branch on carry clear
BCS	Branch on carry set
BEQ	Branch on equal to zero
BNE	Branch on not equal to zero
BMI	Branch on minus
BPL	Branch on plus
BVS	Branch on overflow set
BVC	Branch on overflow clear
CMP	Compare memory and accumulator
CPX	Compare memory and the x register
CPY	Compare memory and the Y register
BIT	Test bits

Table 4.7 Shift Rotate Instructions

Name of instruction	Action taken
ASL	Accumulator
LSR	Logical Shift Right
ROL	Rotate Left
ROR	Rotate Right

Table 4.8 Transfer Instructions

Name of instruction	Action taken
TAX	Transfer Accumulator to x
TAY	Transfer Accumulator to y
TXA	Transfer x to the accumulator
TYA	Transfer y to the accumulator

Table 4.9 Stack Manipulation Variables

Name of instruction	Action taken
TSX	Transfer the stack pointer to x
TXS	Transfer x to the stack pointer
PHA	Push accumulator on the stack
PHP	Push processor status onto the stack
PLA	Pull accumulator from the stack
PLP	Pull processor status from the stack

Table 4.10 Subroutine Instructions

Name of instruction	Action taken
JSR	Jump to subroutine
RTS	Return from subroutine
RTI	Return from interrupt

Table 4.11 Set/Reset Instructions

Name of command	Action taken
CLC	Clear carry flag
CLD	Clear Decimal mode
CLI	Clear interrupt disable
CLV	Clear overflow flag
SEC	Set carry
SED	Set decimal mode
SEI	Set interrupt disable

Table 4.12 Arithmetic Commands

Name of command	Action taken
ADC	Add memory and Carry to Accumulator
SBC	Subtract memory from accumulator with borrow

Table 4.13 Logic Commands

Name of command	Actions taken
And	Logically AND Memory with Accumulator
ORA	OR memory with accumulator

Table 4.14 Flag Commands

Name of command	Action taken
CLS	Clear carry flag
CLD	Clear decimal mode flag
CLV	Clear interrupt disable flag
CLF	Clear overflow flag
SEC	Set carry flag
SED	Set decimal mode flag
SEI	Set interrupt disable flag

Table 4.15 Miscellaneous Commands

Name of command	Action taken
NOP	No operation
BRK	Break

Sweet 16

Sweet 16 is really a remarkable accomplishment. Who better to explain it than the man who created it, Steve Wozniak, aka Woz? Here is an excerpt from "SWEET 16: The 6502 Dream Machine," *Byte Magazine*, November 1977.

"While writing Apple BASIC, I ran into the problem of manipulating the 16-bit pointer data and its arithmetic in an 8-bit machine. My solution to this problem of handling 16-bit data, notably pointers, with an 8-bit microprocessor was to implement a nonexistent 16-bit processor in software, interpreter fashion, which I refer to as SWEET16. SWEET16 contains sixteen internal 16-bit registers, actually the first 32 bytes in main memory, labeled R0 through R15. R0 is defined as the accumulator, R15 as the program counter, and R14 as a status register. R13 stores the result of all COMPARE operations for branch testing. The user accesses SWEET16 with a subroutine call to hexadecimal address F689. Bytes stored after the subroutine call are thereafter interpreted and executed by SWEET16. One of SWEET16's commands returns the user back to 6502 modes, even restoring the original register contents.

Implemented in only 300 bytes of code, SWEET16 has a very simple instruction set tailored to operations such as memory moves and stack manipulation. Most opcodes are only one byte long, but since she runs approximately ten times slower than equivalent 6502 code, SWEET16 should be employed only when code is at a premium or execution is not. As an example of her usefulness, I have estimated that about 1K byte could be weeded out of my 5K byte Apple-II BASIC interpreter with no observable performance degradation by selectively applying SWEET16."

For more information about SWEET16 visit the companion website for this book.

Instruction Set

Anyone can talk about something hard and make it sound hard. Some people even make things that are really easy complicated. Woz not only created language for a processor that does not physically exist, he also made the language very easy to understand and use. Sweet 16 has 16 register instructions and 13 non-register instructions.

Sweet 16's Register Instructions

All register instructions consist of two hexadecimal digits. The first digit is the opcode, and the second digit is the operand used to specify the register we are using. Similar commands are grouped together in the number line to make them easier to remember. (See Table 4.16.)

Table 4.16 Sweet 16's Register Instructions

Name of command	Instruction	Function
1	SET	Constant (Set)
2	LD	(Load)
3	ST	(Store)
4	LD	(Load Indirect)
5	ST	(Store Indirect)
6	LDD	(Load Double Indirect)
7	STD	(Store Double Indirect)
8	POP	(Pop Indirect)
9	STP	(Store POP Indirect)
a	ADD	(Add)
b	SUB	(Sub)
c	POPD	(Pop Double Indirect)
d	CPR	(Compare)
e	INR	(Increment)
f	DCR	(Decrement)

Sweet 16's Non-Register Instructions

With the exception of the `Break` command, the command used to return from subroutines, and the command used to return to 6502 assembly mode, these are all branch instructions used to cause the computer to jump to a given memory location. (See Table 4.17.)

Table 4.17 Sweet 16's Non-Register Instructions

Opcode	Command	Action taken
0	RTN	(Return to 6502 mode)
1	BR	(Branch always)
2	BNC	(Branch if No Carry)
3	BC	(Branch if Carry)
4	BP	(Branch if Plus)
5	BM	(Branch if Minus)
6	BZ	(Branch if Zero)
7	BNZ	(Branch if Nonzero)
8	BM1	(Branch if Minus 1)
9	BNM1	(Branch if Not Minus 1)
a	BK	(Break)
b	RS	(Return from Subroutine)
c	BS	(Branch to Subroutine)

6809 Programming

While the 6502 processor made its way into three of the four machines covered in this book, the 6809 is a superior processor. While virtual 16-bit registers had to be created to add 16-bit functionality to a 6502 system, the 6809 processor has this functionality built right in. This functionality comes in the form of two 8-bit accumulators that can be combined to form one 16-bit register. This processor also has two built-in 16-bit index registers and two stacks. Two stacks mean that the 6809 is capable of addressing modes, which are not available on the 6502.

The 6809 Processor

The heart of the TRS-80 Color Computer consists of three major components: the Motorola MC6809E microprocessor (which despite being an 8-bit computer has many 16-bit features, including two 16-bit index registers and two 16-bit stack pointers); the Motorola MC6847 VDG (video display generator); and the Motorola MC6883 SAM (synchronous address multiplexer).

Because of its versatile use of registers, the MC6809E was a very powerful processor for its time. First of all, this CPU has two accumulators, which can be used separately or combined to form a third register called D. Normally, the registers are used as holding registers or for data manipulation. When the CPU needs to perform any of its 16-bit operations such as additions, subtractions, loads, stores, and 8-bit by 8-bit multiplications, the D register is used.

Another very innovative and ingenious register that is unique to this CPU is the direct page register. On most computer systems, there is an area of memory called the zero page, which consists of the first 256 bytes of memory. This area is a special place because whereas every other part the computer's memory needs a 16-bit address to be referenced, an 8-bit address can be used to address any area of the zero page. This means that it takes less memory and time to perform operations on page 0. The MC6809E takes this concept to a whole new level by the use of the direct page register. With this register, any portion of memory can be treated like zero page memory! This is because when we operate an instruction that uses direct page addressing, the contents of the DP register is used to determine the high order byte as opposed to always defaulting to zero. The DP register by default will usually have a value of 0 and thus addresses the first page of memory. The contents of this register can be changed to place the "zero page" anywhere in memory.

The ability to combine our two accumulators to build a new 16-bit register is cool, but we also have access to four real 16-bit registers. First of all, we have the U and S registers, which are responsible for all stack operations. The S (system) register is used to keep track of the default system generated stack and is called whenever a push or pull function is executed. The U (user) register is provided to allow the programmer (that's us) to create his or her own stack! Very cool! Finally, we have the X and Y registers. These registers are primarily used as index registers, such as when we are performing indirect addressing. These registers do, however, have another very powerful feature—as index registers, allowing indexed addressing, and indirect addressing or indexed indirect addressing. This means that the MC6809E can function efficiently as a stack processor, allowing the microprocessor to support advanced graphics, high-level languages, and modular programming techniques.

As is the case on most processors, the program counter register is used to keep track of what instruction needs to be executed next. The TRS-80, however, provides us with a slight twist. This register can be adjusted to function as an indexed register; we can write programs that address the memory relative to the program counter. In order to take full advantage of program counter relative programming, we have to use relative branch instructions and Load Effective Address instructions (LEA for short). The relative branch instructions can be used in either 8-bit or 16-bit mode. If you use them in 8-bit mode, you

can reference any memory location within plus or minus 256 bytes of the program counter. If you use them in 16-bit mode, you can reference any part of memory. The LEA instructions can be used to store the address of a given memory location in a register where our program can use it. At first, there may seem to be no need for such functionality, but as we will see, this is the key to writing programs that can be placed anywhere in memory. You see, if all addressing is done relative to the program counter and the program counter is adjusted to wherever we place our program in memory, then the relative address of the lines of code in our program to the program counter will always be the same.

It should be noted that there are some limitations to relocatable code. This process makes the program 5 to 10 percent bigger and it takes 5 to 10 percent more time to execute than regular code. When deciding whether or not to use this kind of code, you have to weigh the pros and cons against what you are trying to do. They are best used for machine language utilities such as graphic routines and subroutines called by BASIC programs.

Finally, the condition register is used for branching instructions.

A key element of any video game, especially on retro machines, is timing. Fortunately for us, the MC6809E supports a number of interesting features that enable us to make things happen exactly when we want them to, thanks to a common element of all modern computer systems called the *interrupt*. Interrupts do just what their name implies. While the CPU is busy going about its business, an interrupt stops it from what it is doing and makes it do something else. When the new task is completed, the CPU goes back to what it was doing before. `!NMI`, `!FIRQ`, and `!IRQ` are interrupt vectors that hardware peripherals can use to make the computer give them attention. When these are called, the CPU will take a break from what it was doing in order to do whatever a peripheral device (disk drive, modem, or anything else that you have plugged into your computer) needs it to do. When the interrupt is done executing, the CPU will go back to what it was doing.

Other interrupts are also very helpful to us as game programmers, and they come in two varieties: normal interrupt requests and fast interrupt requests, both of which are very useful to us as programmers.

An example of how useful these interrupts are can be seen in the way the VDG keeps track of the vertical and horizontal sync. A cathode ray in the back of the screen starts drawing from the top-left corner of the screen and paints a line of the image across the screen until it reaches the top-right corner of the screen. Next, the ray is turned off and drops down one line and goes all the way back to the left of the screen. Then the ray is turned on and another line is drawn.

This pattern continues until the whole screen has been drawn and the ray is in the bottom-right corner of the screen. At this point, the ray is tuned off, moved back up to the top-left corner of the screen, and the whole process starts again. This is how the illusion of a picture is created. Only one point on the screen is ever actually lit at any given time. The picture that we see is an optical illusion, which we see because the points on the screen are all being lit so many times in one second that they appear to our eyes to be lit at the same time, and the screen seems to have a picture on it. Each line of the image that is drawn is called a *scan line*.

The period of time when the ray is turned off and moving from the right of the screen to the left is called a *horizontal blank*. The period of time when the ray is turned off and moving from the bottom right of the screen to the top left is called the *vertical blank*. Often, we will want to have some specific action taken when a specific portion of the screen is being drawn. In order to do this, we will need to know when horizontal and vertical blanks are happening. Fortunately for us, the VDG tracks these for us. Whenever a vertical or horizontal blank occurs, an interrupt is generated so that any necessary special code can be executed.

Interrupts are also used to auto-start read-only memory cartridges. Non-maskable interrupts are reserved for use by the expansion port and cannot be disabled or postponed.

There will be times when we will need to change the instructions stored in these interrupts. Ordinarily, we would not be able to do this because the interrupts are not a part of programmable memory. Thanks to the SAM chip, however, which we will discuss shortly, we can write to these areas because they are mapped on top of BASIC ROM memory. Each interrupt is actually nothing more than a 3-byte area of memory that is referenced whenever a specific event occurs. The 3-byte area begins with a 1-byte for the JMP op code, which tells the computer to jump to a location in programmable memory, which is stored in the other two bytes.

We can change what happens when a particular interrupt is called by changing this address from its default memory location containing system code to a new location calling the code that we want to have executed. Keep in mind that sometimes it can be very handy to have default system code do some of the work for us so you may want to have your code call the default code and then modify the results afterwards.

If it ain't broke, don't fix it (or don't reinvent the wheel)

In 1998, I became very interested in the Y2K problem. Not only did it pose an interesting intellectual challenge, but solving the problem also seemed to be a sure path to financial security. One year later, I had created a program I affectionately referred to as S.A.T.H.R. (Software Alternative To Hardware Replacement).

This story may be interesting to you for a few reasons, starting with the way the program used interrupts. You see, the problem was that after 1999, the hardware on many computers would start generating 1900 instead of 2000. Any DOS-based system, which means any operating system made by Microsoft at the time, used the same interrupts as their foundation, and all operating systems use the same BIOS interrupts for their foundations. This meant that any process that depended on interrupts could be hijacked and reconfigured on any Microsoft OS and, in theory, other operating systems, although this theory was not tested.

(I suspect that the program would have worked on Novell as well but this was never tested. The reason for my suspicion was that when you read over the interrupt list for DOS, you find a lot of references to interrupts that were designed for Novell!)

The great workhorse of all Microsoft systems at the time was interrupt 21. This interrupt handles most major functions such as opening and closing files, and getting and setting the date and time. By placing the value 2ah into the Ax register and calling Interrupt 21, the operating system and other software would cause the interrupt to read the date from the hardware and then tell the operating system or software what the date was. Of course, if the hardware fell victim to the Y2K bug, then this date would be wrong.

What my program did was replace interrupt 21. Because my program was a TSR (Terminate and Stay Resident) program, whenever interrupt 21 was called, my program was activated. If the date was requested, my program would call the original interrupt 21 so that it could do the work of obtaining the current date from the hardware. When the original interrupt had completed its work, my program would take over by running the date through a date filter that I invented, producing the correct date, which was passed on to the OS or application that needed the date.

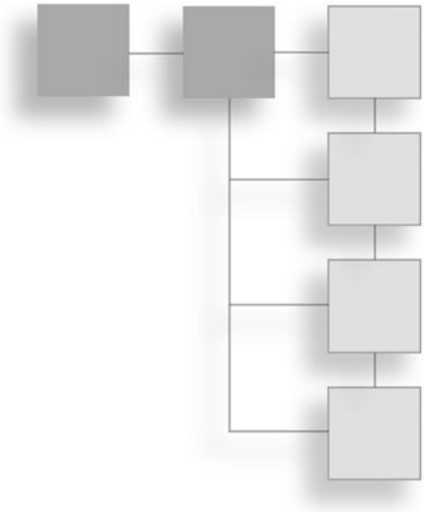
The point is that I could have completely hijacked the interrupt and read the date manually using BIOS interrupts and/or by reading the BIOS date locations directly, but for my purpose there was no need. As a hardcore programmer, I wanted to do this, but as a budding businessman who had to look at the whole picture, I was forced to resist the urge. My goal was to get a solid application working that was fast and, most of all, reliable. I knew the interrupt 21 service 2ah would always reliably obtain the current date from the hardware. What I needed to do was to make sure that my date filter would work just as reliably! The key thing to remember when you are optimizing games or any software is that your goal is not to optimize the whole application. Your goal is to optimize the areas where the speed and reliability will affect the user's comfort while using your application.

Conclusion

Like so many other things in life, assembly language programming can seem daunting at first, but with practice it becomes much easier. You can visit this book's companion Web site to download assemblers for all of the programs covered in this book as well as many tutorials and examples of machine language programs.

Once you master the art of assembly language programming, you will be able to build any game that you can imagine.

chapter 5



A Game Graphics Primer

Imagination is the beginning of creation. You imagine what you desire, you will what you imagine, and at last you create what you will.

George Bernard Shaw (1856–1950)

Many of the things that we enjoy in life are nothing more than illusions. As I write this chapter, I have a very large bottle of grape juice by my side. The juice looks like grape juice and tastes like grape juice, but it is not grape juice. Nevertheless, I like it. When I turn on the radio and listen to my favorite song, there isn't really a person in the box singing to me; it is simply another illusion. Finally, when I watch TV or play a video game, I am not looking into a real 3D environment; this is simply another optical illusion. The key to creating a good video game is to understand how these optical illusions work so that we can create a game world that appears to mimic reality.

While there is currently no way to implement the illusion of taste or smell in a video game, you can make very good use of optical and audio illusions and, to a more limited degree to fully immerse our players in the game. Always remember that reality is not as real as you think it is. There is a world around us that passes information to our sensory organs, which in turn passes this raw data to our unconscious mind where it is assembled to create our conscious perceptions of the world around us. The key to creating the illusion of there being an actual game on the player's monitor is not to re-create the world as you think that you see it, but rather to re-create the raw data that your unconscious mind takes in and converts into your conscious perception of reality. Oddly enough, once you learn a few new principles, it is actually easier to re-create this raw data than it is to re-create the world as you see it.

Note

There is a very interesting phenomenon that often occurs naturally when a player is truly immersed in a game. You see, the body is used to moving in order to navigate its environment. In the game world, however, our whole body does not have to move, only our fingers. Interestingly enough, when the mind is fully focused on the game, the player's body will move anyway. You have probably seen this before. When some people play Mario Brothers and they have to make a big jump, they often bounce their hands in the air as if this movement will give the character on the screen more momentum with which to jump. Likewise, when many people play a flight simulator or a driving game, they will lean as they turn curves or have similar reactions when they speed up or hit the brakes.

Color

Red, green, and blue are three very special colors. The reason that they are special is because if you mix these colors equally you will create the color white. Mixing these colors using various proportions, you will be able to create any color in existence. Most computer systems and color monitors work with color using a format called RGB. RGB stands for Red, Green, and Blue. This means that the colors on the screen are generated by mixing different proportions of red, green, and blue. On a retro game machine, you will not be able to create every possible combination of red, green, and blue. If you could, you would be able to generate any color known to man. Instead, you will normally have access to 16 colors or less, depending on the machine and graphics mode you are using. Because they are the primary colors, you will normally have access to red, green, and blue on any machine that you use.

What Makes a Picture?

Take a look at Figure 5.1. What do you see?

It seems like a meaningless arrangement of blocks, right? Now stand this book up so that you can look at it from a distance and slowly walk backwards until the hidden image becomes clear.

What just happened?

You just discovered one of the many talents that our brains possess that we not only use every day but we take for granted every day. This talent is called reconstruction, and it is

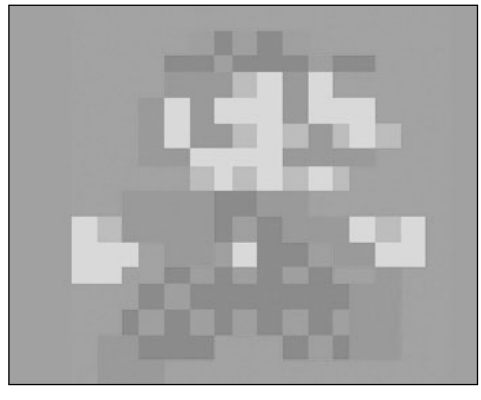


Figure 5.1 Seemingly random arrangement of blocks.

the reason your mind can make sense of the contents of a computer screen, works of art, and the text on this page! As long as our eyes are open, rays of light are constantly bouncing off of objects around us and entering our eyes as seen in Figure 5.2.

These images form an upside down picture on the retina of the eye similar to the way a picture is stored on the film of a camera. Figure 5.3 illustrates.

In a camera, the shutter is normally closed so that no light can enter the camera. When you press the button to take a picture, the shutter is opened for a fraction of a second to allow light through. This allows the image formed on the back of the camera to be saved on a portion of the camera's film. In movie cameras, the process is exactly the same except the shutter is opened and closed very rapidly so that as many as 24 or 30 or more pictures are taken every second. Examine Figure 5.4.

The human eye takes this process to the next level. The eye is always open, and there is always a picture focused on the back of the retina. This image is detected by optic nerves, called *rods and cones*, in the back of the eye that record the correct image and transmit this image to the brain via the optic nerve. The brain then makes sense of the image and presents your consciousness with the image that you eventually see.

Figure 5.5 is a rather simple diagram. There are more than just two or three rays of light entering our eyes; indeed, there is an uncountable number of light rays entering your eyes every fraction of a second, every day, all day, even as you read this book. Even more complex is the process that the brain has to go through to correctly interpret these

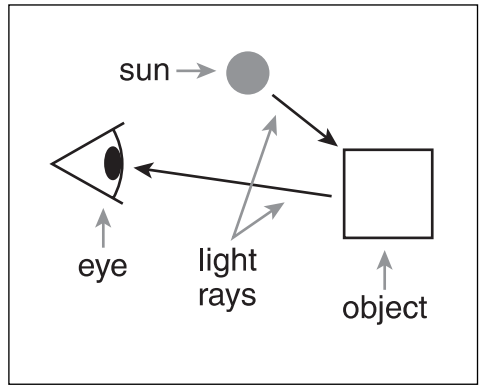


Figure 5.2 Light rays are constantly entering the human eye.

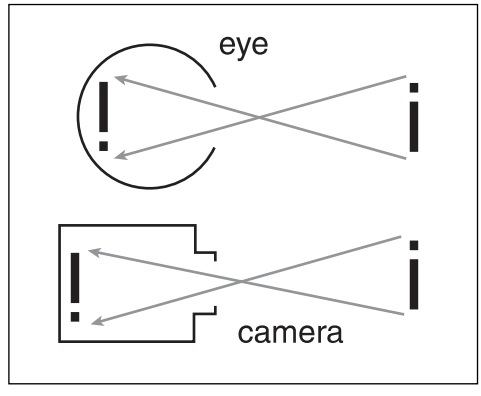


Figure 5.3 Comparison of the path of light in a camera and the human eye.

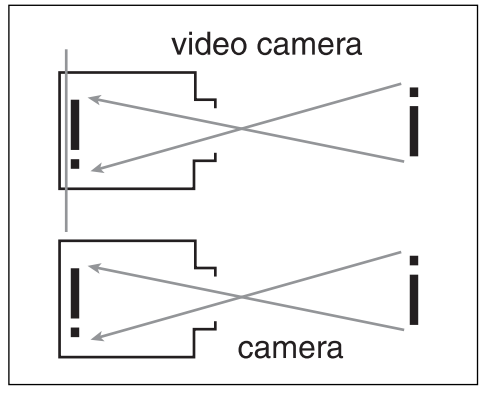


Figure 5.4 Illustration of how a camera and video camera work.

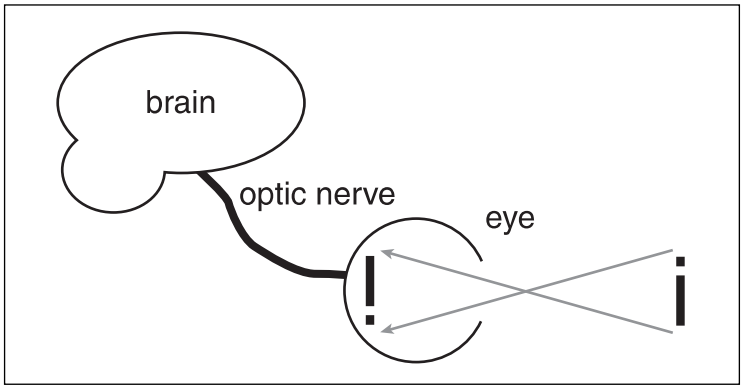


Figure 5.5 Illustration of how the eye and brain work together.

images of the outside world, which it obtains from the eye. While a full understanding of how the brain works is beyond the scope of this book, an understanding of image reconstruction is necessary for us to create believable images on the computer screen.

The first thing that we did in this chapter was to look at a strange, seemingly random, collection of blocks that had no meaning up close, but as you walked away from them, the image became clear.

This simple scientific experiment allowed us to see an example of one of the brain's many functions, as well as its primary limitation. That is if an image is broken down into a pattern of blocks and put in front of the human eye, the brain has the ability to reassemble those blocks into the original image provided the blocks are not too large. Figure 5.6 illustrates this.

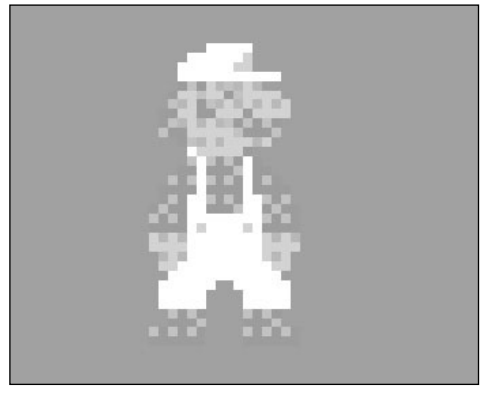


Figure 5.6 If an image is broken down into a series of "blocks" and placed in front of the human eye, the brain will reassemble the "blocks" into the original image as long as the blocks are not too large.

When you first looked closely at Figure 5.6, the "blocks" were too large for your brain to reassemble. As you moved farther away, the blocks appeared to become smaller until finally they were small enough for your brain to reassemble.

This exact phenomenon occurs when the player is looking at your game on the screen. If you were to draw that exact same pattern of blocks on a computer screen, it would be equally unrecognizable and it would become

equally clear as the player backs away from the screen. It is not practical, however, to ask the person playing your game to stand 12 to 50 feet a way from the screen in order to play. You have to find another way to make your blocks smaller. To solve this problem, most computer systems are able to display different video modes with different size blocks.

The size of the blocks on the screen is referred to as the resolution of the screen. Screen resolution, however, is not a measure in the way you may expect. We do not measure the height and width of the blocks on the screen to find the screen's resolution; instead, resolution is measured according to how many blocks can fit on the screen. If the blocks are very small, many blocks will fit on the screen. We call this a high-resolution display. If the blocks are larger, fewer blocks will fit on the screen. We call this a low-resolution display. See Figures 5.7 and 5.8.

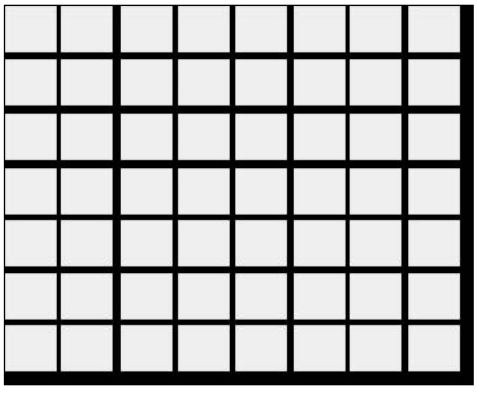


Figure 5.7 If the blocks on the screen are large, only a few blocks will fit on the screen. This is called a low-resolution display.

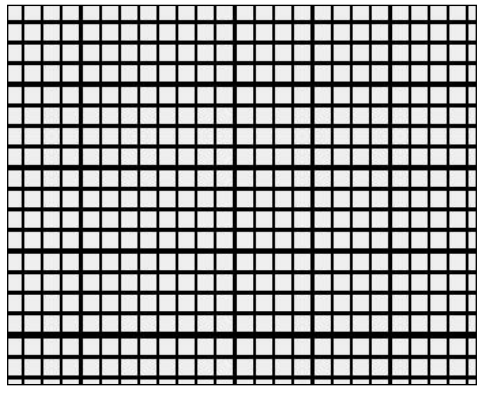


Figure 5.8 If the blocks on the screen are small, many blocks will fit on the screen. This is called a high-resolution display.

The higher the resolution, the smaller the blocks on the screen will be, which means that you can more easily create images on the screen that the brain will recognize. Before you can create images that will be displayed on a particular video system, you need to know exactly how many dots can fit on the screen in total as well as how they are arranged. For this reason, when we give the resolution of the screen we will say how many blocks can fit horizontally across the screen and how many blocks can fit vertically on the screen. If a screen can hold 100 blocks horizontally and 50 blocks vertically, then we would say that the screen has a resolution of 100×50. See Figure 5.9.

In a perfect world, you would always be able to use the highest resolution video mode that your computer can handle. We do not, however, live in a perfect world so things are not going to be that easy. The highest video mode on most retro game machines is actually black and white (or green and black or yellow and black depending on your monitor).

While you would be able to create the most detailed images that the machine can handle, black and white games are not as compelling as a game that fills the screen with color. On most retro game machines, the lowest resolution video mode will have the most color while the highest resolution video mode will have the least color. I admit, at first this does not seem to make much sense and actually goes against all logic. After all, if we want to make killer games we would want to have the highest resolution graphics display equipped with the most colors. Believe me, there is a method to this madness and all will become clear once you actually get started programming your game machines. For now, just accept this concept as a fact of life and take comfort in the knowledge that later on you will learn how to hack these game machines to produce larger amounts of colors than the manufacturer originally intended.

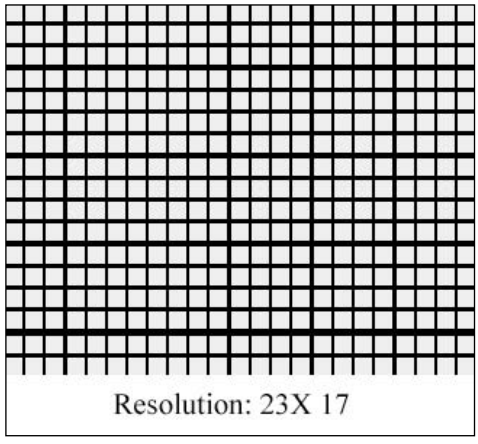


Figure 5.9 When we state the resolution of a computer screen we always state the number of blocks that can fit across the screen followed by the number of blocks that can fit vertically on the screen. A times sign (×) is placed between the numbers to separate them. When this is read, we say the screen has a resolution of "23 by 17."

Approximating Shapes with Limited Pixels

What is the image in Figure 5.10?

I bet you said a house.

What do you see in Figure 5.11?

I bet you said a smiley face.

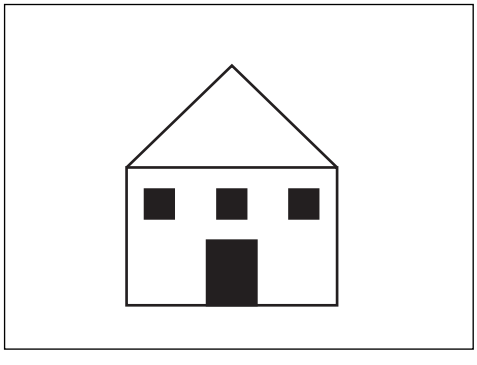


Figure 5.10 Boxed image of a house.

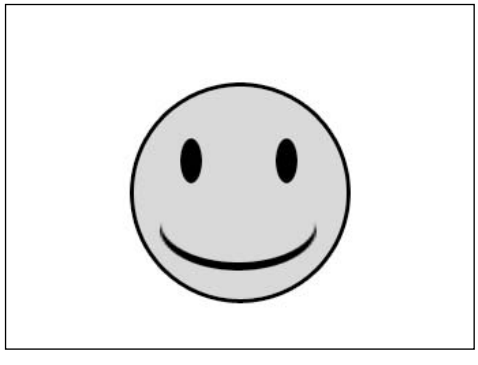


Figure 5.11 Smiley face.

Have you ever seen an actual house that looks like the one in Figure 5.10? Have you ever seen a person with a face that looks like the one in Figure 5.11?

I bet that you answered no to both of the last two questions. That begs the question why in the world did you automatically associate these two images with real world objects even though they look like no real world object that you have ever seen?

The answers to these questions form the heart of the knowledge that you need in order to create believable images on the computer screen.

Symbolism

Essentially, a symbol is an image that is used to represent an abstract concept or an idea. If you see the symbol found in Figure 5.12 somewhere, then you know that the area or facility is reserved for handicapped individuals.

Figure 5.12 Handicapped symbol.

No one has to spell it out for you. At a handicapped-parking area you will not find a long explanation telling you that this is a handicapped area and that if you are not handicapped then you should not park there. Instead all that you will see is a symbol and your brain will automatically kick in to tell you that this is a handicapped spot.

Symbolism is a very powerful tool in video game designs because symbols can convey large amounts of information to both the player's conscious and subconscious mind. Not all symbols, however, are equal.

A symbol has no effect if no one knows what it is. If you create a symbol that you know the meaning of but no one else does, the person playing your game will not receive important pieces of information that you are trying to convey to him. This will in most cases lead to confusion and a loss of interest in your game. On the other hand, if you use a symbol that you and your friends understand but no one else does, the game will be fun for you and your friends and no one else. When using symbolism in games, you want to use symbols that are universally agreed upon. If it is a must for you to create a new symbol, make sure that it is properly documented and easy to remember.

Visual Cues

Your mind is truly amazing. As you have seen, your mind works so well that you are not even aware of much of the work that it does because this work is done subconsciously. Another example of this is the way the mind will use visual cues to identify what an object

is. A good example of this is the creation of shadow animals. By turning off the lights and turning on a flashlight, you can position your hands in front of the flashlight in such a way that the shadow on the wall appears to be the shadow of an actual animal. You can see an example of this in Figure 5.13.

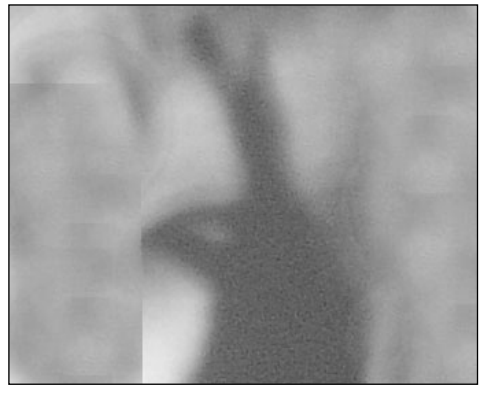


Figure 5.13 Shadow puppets made using a flashlight and hand gestures.

This shadow does not look like a living breathing animal, but the image "feels" like that of a dog. It seems to have a long snout, pointy ears and if the person making the shadows is very skillful it even seems to have a wagging tongue.

That is basically what we are going for when we create our game graphics. Even though we cannot produce a photographic rendering of the animal, we would like to have enough visual cues in our game objects to make our images "feel" like the object or creature that we are trying to represent.

Putting Them Together

Now that you understand the science and the psychology of how the brain takes in the world around it you need to look at how to use this information to create pictures players will recognize.

Time of Day

Our ability to accurately depict whether it is night or day varies depending on the game machine we are using and the level of detail and color usage that the machine makes available to us. On modern machines you can easily display many shades of blue and red and orange and combinations of these colors. This allows you to accurately recreate what a sky would look like at night, dawn, midnight, dusk, and every hour in between. Furthermore, most computers today are capable of displaying resolutions of 1280×1024 or more!

In the absence of such dazzling capabilities you must improvise, and it is here that symbolism and the visual cue come to the rescue.

There are three primary visual elements that tell us that it is nighttime: The sky is black, the moon is out, and there are stars in the sky. Element 1 can be reproduced on any computer system. If you do not put a color on the screen, the screen will be black. This means that all that you have to do is leave the area of the screen representing your game blank and you will automatically have a night sky! Elements 2 and 3 are still easy but a bit more involved. See Figures 5.14 and 5.15.

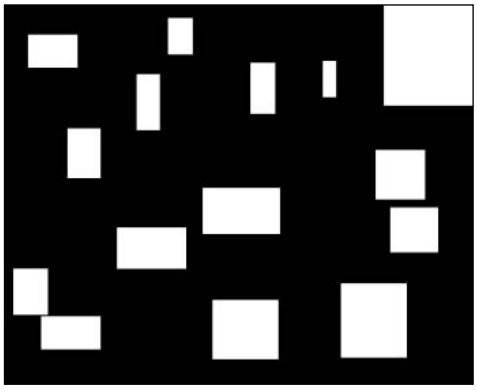


Figure 5.14 A night sky drawn in low-resolution mode.

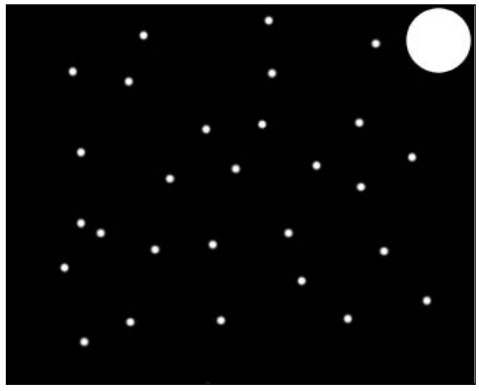


Figure 5.15 A night sky drawn in high-resolution mode.

Figure 5.14 looks like a deranged checkerboard, while Figure 5.15 is more easily recognizable as a night sky. As you can see if it is a must for us to create a game in low resolution mode, it is better to represent the night sky as a plain black screen as opposed to trying to fill the sky with stars. Because stars as well as the moon are usually white, you can use high-resolution modes to draw a convincing sky without worrying about any limitations in the number of colors available in a high-resolution mode.

There are basically two elements that tell us that it is definitely daytime: a blue sky and a sun in the sky.

Most retro game machines will allow you to use the color blue. This means that you should always be able to create a blue sky as long as the computer is connected to a color monitor or TV. To further add realism to the game, you can place a yellow ball in the sky to represent the sun. This allows for much more opportunities than you may think. We mentioned earlier that you cannot change the color of the sky to represent the shades of the atmosphere at various time of the day. You can, however, change the position of the sun to create the same effect. The sun in the top-right corner of the screen may represent the morning, the sun in the center of the screen may represent midday, and the sun in the far left of the screen may represent dusk. When the sun has been set for a period of time, you can paint the sky black, add stars and a moon, and presto, you have a night sky. You can repeat the same process with the moon so that the player can be aware of the passage of the night.

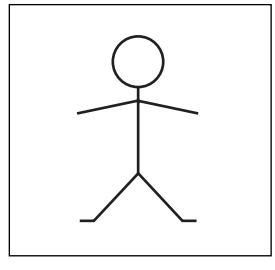


Figure 5.16 Stick figure of a man.

Human Anatomy

Okay, so the artwork in Figure 5.16 is not the best in the world. Nevertheless, you know what this image is. It's a stick figure of a

man. This image works partly because of visual cues and partly because of symbolism. The symbolism comes into play because you probably learned to draw like this as a child. Someone told you that this image represents a man, and this is the primary reason you are able to recognize this image. Visual cues also play a role in your understanding of this image.

Looking at Figure 5.17, you can see that the stick figure is jumping, running, and walking. A full understanding of how your mind recognizes what the man is doing is beyond the scope of this book (not however beyond the scope of my Web site www.retrogameprogrammingunleashed.com; check it out for more information), but suffice it to say that the position of his appendages as well as his orientation tells your mind what he is doing.

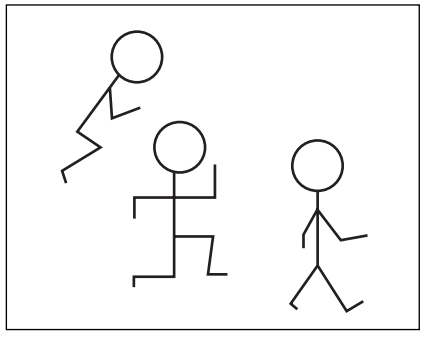


Figure 5.17 Stick figure of a man jumping, running, and walking.

As simplistic as this stick figure man in Figure 5.18 is, he is the foundation of virtually every character you will ever create. First of all, when you sit down to create a game, you enter a brainstorming session where you try to decide everything about the game, including what your player is able to do. Now you might as well be able to sit down and draw your character exactly as you want him to appear so you can create storyboards using the image of your final hero. Not everyone, however, has this ability. Furthermore, when you are brainstorming, you may know what the player's character will be able to do, but you may not know exactly what he will look like yet. Even if you are an artist and you know what your character will look like, you may not have time to create detailed storyboards. For all of these reasons, when you create your storyboards and decide the way your player moves, you will often want to use stick figures. Look at Figure 5.19.

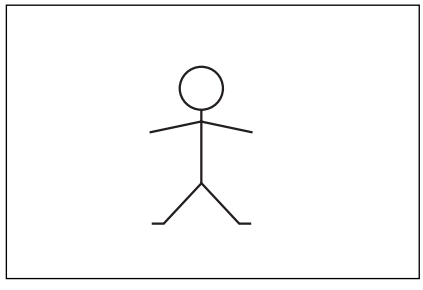


Figure 5.18 Stick figure of a man.

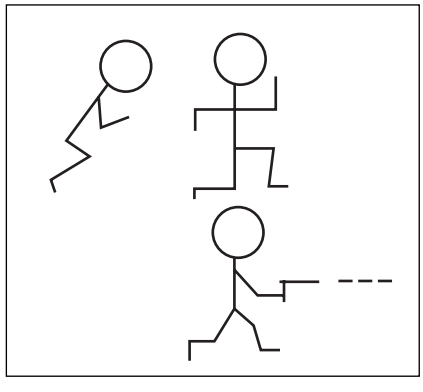


Figure 5.19 Stick figure of a man performing various game-related actions.

It's not pretty. In fact, it's pretty ugly, but it works for me when I need to visualize the various actions a player in my game will be able to do.

Dithering

At first glance, the image of a pipe in Figure 5.20 appears to be made up of three colors: white, gray, and black. Upon closer inspection, however, you can see that it is actually made up of only two colors, black and white. See Figure 5.21.

The technique that was used to give the impression of a third color is called dithering and involves creating a pattern of black and some other color (in this case, white) to create a lighter version of the color you are working with. This technique will be invaluable to you especially when you are working in high-resolution modes, which will usually only allow you to use black and white. This pattern is also useful when you need to apply shading to an image.

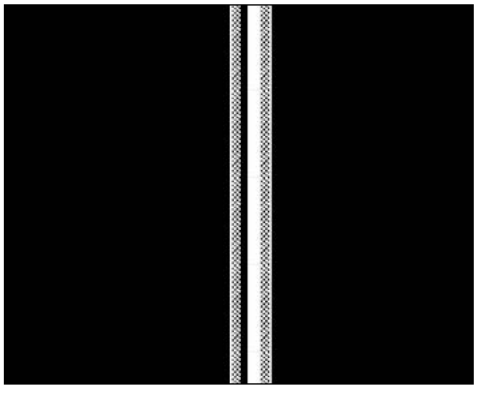


Figure 5.20 An image that seems to be made up of white, black, and gray.

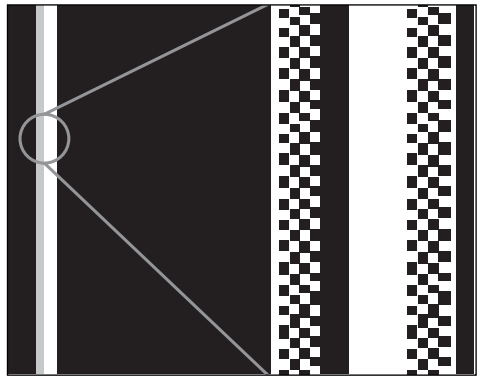


Figure 5.21 The pipe is really only made up of two colors: black and white.

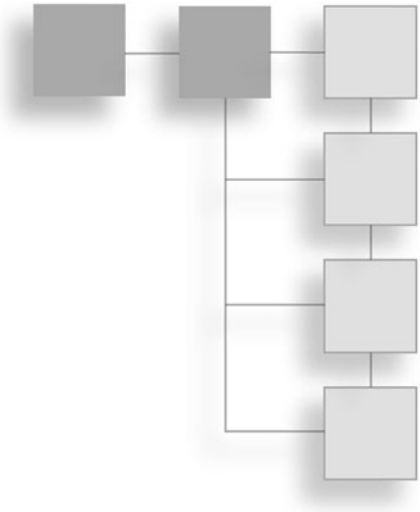
Conclusion

Creating great pixel art is very important if you are going to create a great-looking game. On modern day computer systems you have the luxury of using millions of pixels to draw an image, which means that you can create photo-realistic images. When programming with vintage computer system, you do not have the option of creating photo-realistic imagery. In order to make great imagery, you will need to first understand the way that the brain interprets the world around you. You have to understand the symbols and visual cues your brain uses to make assumptions about what it sees. Finally, after you have taken

all of this information into account, you will use the technical knowledge learned later in this book to put moving pictures on the screen. It is these moving pictures that will create the illusion of a game on the screen.

If you do not have an artist to do your artwork, then you should have enough information to understand the basics of how to design the graphics for your game.

CHAPTER 6



Setting the Video Mode

You cannot depend on your eyes when your imagination is out of focus.

Mark Twain (1835–1910)

You now know how to talk to your computer, and you know the concepts that go into drawing images on a computer screen. Let's recap in the simplest terms possible. There is a video screen, shown in Figure 6.2, and there is a video buffer, shown in Figure 6.1.

These two elements of the computer have a direct relationship that can take several forms. The most basic form is that shown in Figure 6.3.

```
0010000000100001O
1001000010000010O
1000000001001000O
0101000101010000O
1111110000010010O
1100000101010101O
```

Figure 6.1 Video buffer.

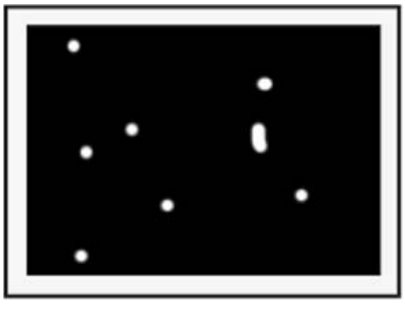
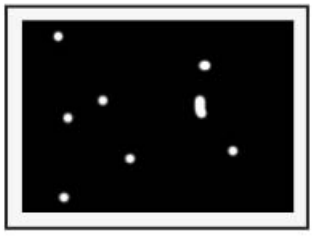


Figure 6.2 Video screen.



00100000001000010
10010000100000100
10000000010010000
01010001010100000
11111100000100101O
11000001010101010 1

Figure 6.3 Video buffer video screen relationship 1: Each bit of video memory controls one point on the screen.

In this example, each bit of video memory controls a point on the screen. If a bit is equal to 1, the point on the screen that it represents is turned on. If a bit is equal to 0, the point on the screen that it represents is turned off. This sort of video mode will always be black and white (or yellow and black or green and black, depending on your monitor). In order to display more colors on the screen, more than one bit has to be assigned to each point on the screen. A single bit on the screen can hold two values (0 and 1), so you can only assign one of two colors to any point on the screen.

If you use two bits to represent each point on the screen, then it is possible to display up to three colors. With three bits, seven colors. Finally, if you have used a single byte to represent the points on the screen, you will be able to paint a point on the screen in any of 256 unique colors. See Figure 6.4.

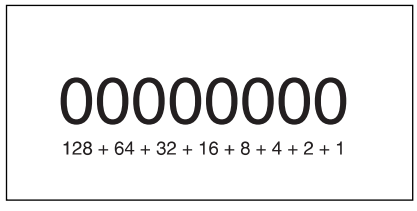


Figure 6.4 If you use eight bits to represent each point on the screen, you can display up to 256 unique colors.

What you need to learn now is how to control the number of bits used to control each bit on the screen and how to move images to and from the video display.

Setting the Video Mode

Like most things in life and in programming, setting your computer's video mode may seem confusing at first, but once you get the hang of it, you will find the actual process to be very easy. On most computer systems, this process simply involves placing a special combination of bits into one or more memory locations. On other machines, such as the Atari 400/800, things are a little more complicated than that but never get so complicated as to be beyond the grasp of an informed programmer's ability. The purpose of this chapter is to teach you how these systems work so that you may have full control over the graphic display of your chosen computer system. Let's get the ball rolling with the TRS-80 Color Computer.

Setting the Video Mode on the COCO

There are 15 unique graphics modes available to you: 2 alphanumeric, 5 semi-graphics, and 8 graphics modes. In the alphanumeric modes, placing a byte into video memory will cause a letter or number to appear on the screen. In semi-graphics mode, placing a byte of data into the video buffer will cause a letter, a number, or a pattern of blocks to appear on the screen. In a purely graphics mode, placing a byte of information on the screen will cause a pattern of blocks to appear on the video screen.

There are two devices that control what video mode your computer is in. These are the Video Display Generator (VDG) and the Synchronous Address Multiplexer (SAM).

The VDG controls the way video memory is interpreted. It is this chip that decides whether a byte will be interpreted as a character or as a pattern of bits. The SAM chip has full control over the way video memory is managed. Because the SAM chip is in charge of all memory management, it has to be made aware of what video mode is being used by the VDG chip in order to manage video memory correctly. If the VDG and SAM chip are set to work with two different video modes, strange things will happen—and some cool things can happen as well. Setting these two microchips to different video modes is the key to unlocking new video modes, such as those that put full bitmapped graphics and text on the screen at the same time.

As powerful as the VDG is, it has one major limitation. There is a limit to how much memory this chip can read at one time, which means that it cannot read the entire video buffer at one time. The problem you have with the VDG not being able to view the whole of video memory is also alleviated because the SAM is able to provide access to the full screen. Because this configuration allows the VDG to page through video memory 512 bytes at a time, page flipping, which is so vital for animation, is possible.

When you want to set the computer into a particular video mode, you have to place a special pattern of bits into the VDG register and the SAM register. Figure 6.5 shows these patterns.

Here is an example of the code you would use to set your video display into a 128 × 192 color graphics mode.

```
Lda     %00000111
Anda     $ff22
Ora     #%11101000
Sta     $ff22
Lda     #%11000000
Anda     $ffc0
Ora      #%00101001
Sta      $ffc0
```

MC6883 DISPLAY MODE REGISTERS			PIA REGISTER BITS HEX ADDRESS (FF22)								DATA BITS		ALPHA/GRAPHIC MODE SELECTED
V_2	V_1	V_0	7	6	5	4	3	2	1	0	7	6	
0	0	0	0	X	X	0	CSS	N	N	N	0	0	Alphanumerics
0	0	0	0	X	X	0	CSS	N	N	N	0	1	Alphanumerics Inverted
0	0	0	0	X	X	0	X	N	N	N	1	X	Semigraphics—4
0	0	0	0	X	X	1	CSS	N	N	N	1	X	Semigraphics—6
0	1	0	0	X	X	0	X	N	N	N	1	X	Semigraphics—8
1	0	0	0	X	X	0	X	N	N	N	1	X	Semigraphics—12
1	1	0	0	X	X	0	X	N	N	N	1	X	Semigraphics—24
0	0	1	1	0	0	0	CSS	N	N	N	X	X	64 x 64 Color Graphics
0	0	1	1	0	0	1	CSS	N	N	N	X	X	128 x 64 Graphics
0	1	0	1	0	1	0	CSS	N	N	N	X	X	128 x 64 Color Graphics
0	1	1	1	0	1	1	CSS	N	N	N	X	X	128 x 96 Graphics
1	0	0	1	1	0	0	CSS	N	N	N	X	X	128 x 96 Color Graphics
1	0	1	1	1	0	1	CSS	N	N	N	X	X	128 x 192 Graphics
1	1	0	1	1	1	0	CSS	N	N	N	X	X	128 x 192 Color Graphics
1	1	0	1	1	1	1	CSS	N	N	N	X	X	256 x 192 Graphics

X = DON'T CARE
N = DO NOT CHANGE

Figure 6.5 The patterns of bits needed to place your COCO into specific video modes.

Looking at Figure 6.5, you can see that in order to place the video mode to a 128 × 192 color graphics mode, you need to put a pattern of 11101000 into the VDG register and a pattern of 00101001 into the SAM register. There is just one problem. Both the VDG register and the SAM registers are used for things other than setting video modes. Because of this, there are bits in these registers that you are not allowed to change. If you just place the pattern of bits that you need into these registers, you will change the values of bits that you are not supposed to change.

In the VDG register, you are not supposed to change bits 0, 1, or 2. See Figure 6.6.

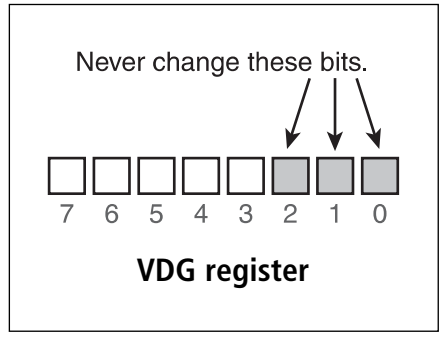


Figure 6.6 Never change the value of the first three bits of the VDG register.

You need to do some clever programming to get around this problem. Specifically, you are going to be using the AND and OR commands. Figure 6.7 contains the truth table for the AND command.

And truth table.		
bit 1	bit 2	result
0	0	0
1	1	1
0	1	0
1	0	0

Figure 6.7 The truth table for the AND command.

If you AND two bits together and either of those bits is a zero the result will be zero. This allows you to perform something called *bit masking*. See Figure 6.8.

In this example, you AND bits 3 to 7 with zeros. This causes bits 3 to 7 to be carried down as zeros. You AND bits 0, 1, and 2 with ones. This means that if the value of any of these bits is one, it will be carried down as a one. If it is a zero, it will be carried down as a zero. The end result is that the first three bits are preserved while the other five bits are cleared. You will then store this pattern somewhere until you need it. Now it is time to use the OR command. Figure 6.9 contains its truth table.

When two bits are ORed together and either or both of the bits is equal to one, the result will be one. Figure 6.10 demonstrates how you can use this to your benefit.

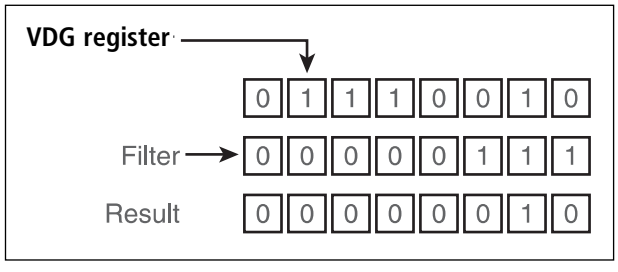


Figure 6.8 The results of ANDing the VDG with a specific pattern to preserve the first three bits.

OR truth table.		
bit 1	bit 2	result
0	0	0
1	1	1
0	1	1
1	0	1

Figure 6.9 The OR truth table.

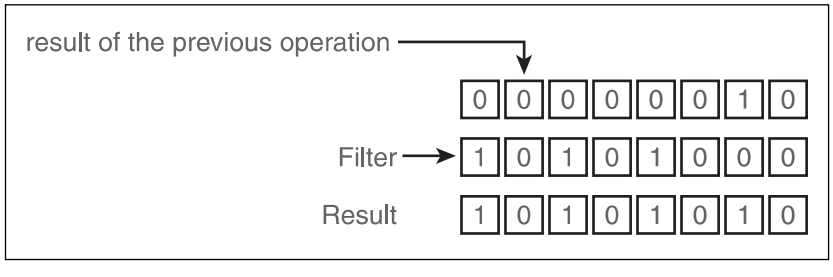


Figure 6.10 Here is the result of ORing the results of the previous operation with a specific pattern to set the VDG register into the video mode you need.

Here you OR the first three bits with 0. This is to ensure that you do not change these bits. If any of these bits is equal to zero, it will be carried down as a zero. If they are set to 1, the result will be 1. Bits 3 to 7 were all set to zero in the previous operation. You OR these bits by the pattern you need to set the video mode that you want (in this case a 128 × 192 color graphics display). This is how you set the video mode to the VDG register, and it is what you do in the first four lines of code above.

The first line fills the accumulator register with the pattern you need to "mask" the VDG register.

```
lda         %00000111
```

The next line ANDs the value in the accumulator with the contents of memory location ff2, which is the address of the VDG register. The results are stored in the accumulator. At this point, the accumulator contains the original value of the first three bits of the VDG register while the other five bits are set to 0.

```
anda         $ff22
```

Line three ORs the contents of the accumulator with the pattern you need in order to set the video mode of the VDG.

```
ora         #%11101000
```

Finally, line four stores the results of the ANDs and ORs back into the VDG register, completing the first step needed to set the video mode.

```
sta         $ff22
```

And that's it. You just set the VDG chip to the video mode that you want; now all that you have to do is set the SAM chip. To do this, you actually do the same thing as before, only this time you access a different memory location, you use a different pattern for the mask, and you OR with a different pattern. Let's see how this works.

This time around, you will use the pattern 11000000 for your mask. This will preserve the last two bits while blanking the first six and storing them into the accumulator. Here is the code that does this:

```
lda         #%11000000
anda        $ffc0
```

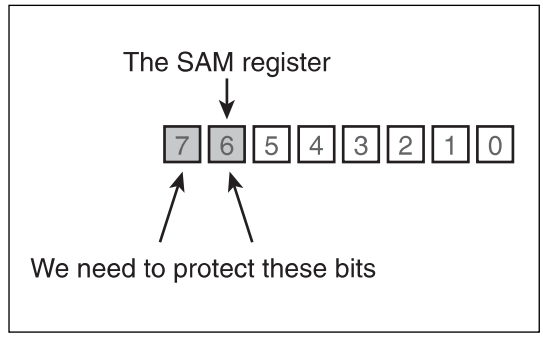


Figure 6.11 Set the first six bits to the value that you need, but do not change the last two bits.

Next, you will OR the value stored in the accumulator with the pattern needed to set the SAM chip to the video mode that you need. Setting this register is a bit tricky at first though. Think about how the light switches in your house work. You flip them in one direction to turn the light on and in another direction to turn it back off again. The same concept applies in setting the video mode of the SAM. There are three switches that you need to turn off or on in a particular pattern to get the desired result. Each switch is made up of two bits. Placing a 1 in the odd bit turns the switch off and placing a 1 in the even bit turns the switch on. See Figure 6.12.

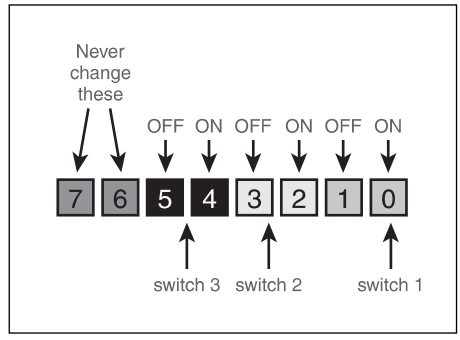


Figure 6.12 The SAM register contains switches that are turned off by placing a 1 into the odd bit and turned on by placing a 1 into the even bit.

You can see in Figure 6.12 that you need to turn switch 1 off, and turn switches 2 and 3 on. To do that, place the pattern shown in Figure 6.13 into the SAM register.

Here is the code that will OR the correct pattern into the register and store the results of your operation back into the SAM register.

```
ora         #%00101001
sta         $ffc0
```

That's it. You now know how to set the video mode of the COCO to any mode that you want. All that you have to do now when you want to set a given mode is look at the Figure 6.12 to see the pattern you need and modify the code that you have just written.

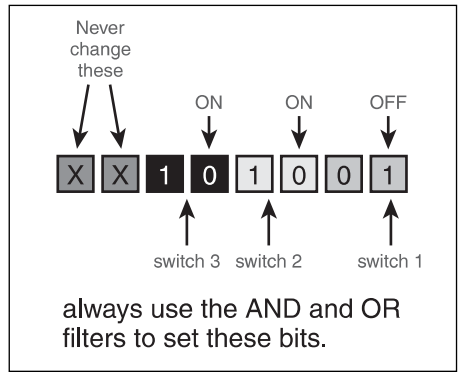


Figure 6.13 Use this pattern in order to set the SAM to the video mode you desire.

Setting the Video Mode on the Apple II

In the Apple II, there are three classes of video displays: text displays, low-resolution displays, and finally, high-resolution displays. In addition to these dedicated modes, it is also possible for you to create mixed modes that display text in a window at the bottom of the screen and either low- or high-resolution modes on the top of the screen. This gives you a combination of five possible screen modes.

- Full screen text mode
- Full screen low-resolution mode
- Full screen high-resolution mode
- Mixed text mode and low-resolution mode
- Mixed text mode and high-resolution mode

Setting these video modes is very straightforward and is similar to the way you programmed the SAM chip in the COCO with a few differences.

In the Apple II, there are eight soft switches. In your house, you have light switches. You flip them one way to turn the light on and another way to turn the light off. With a soft switch, there is one memory location that represents the off position and another that represents the on position. All that you need to "flip" a switch is to read from or write to the memory location that represents the on or off address of a given switch.

Figure 6.14 contains a list of the on and off addresses for the eight soft switches used to control the video mode of the Apple II.

Location		
Hex	Decimal	Description
$C050	49232	DISPLAY A GRAPHICS MODE
$C051	49233	DISPLAY TEXT MODE
$C052	49234	DISPLAY ALL TEXT OR ALL GRAPHICS
$C053	49235	MIX TEXT AND GRAPHICS MODE
$C054	49236	DISPLAY PRIMARY PAGE
$C055	49237	DISPLAY SECONDARY PAGE
$C056	49238	DISPLAY LO-RES GRAPHICS MODE
$C057	49239	DISPLAY HI-RES GRAPHICS MODE

Figure 6.14 These are the on and off addresses of the eight soft switches used to control the video mode of the computer.

These switches are used to generate the five different distinct video modes available to you as the programmer. These modes are listed in Figure 6.15.

Here is an example of code that will give you a full screen of high-resolution graphics.

```
LDA     $C054
LDA     $C057
LDA     $C052
LDA     $C050
```

Line one flips the soft switch used to activate the primary page.

Line two flips the soft switch used to activate high-resolution mode.

Line three flips the soft switch used to decide between a full screen and a mixed mode, setting it to display the whole page in one mode.

Finally, the last line activates graphics mode.

PRIMARY PAGE		SECONDARY PAGE
SCREEN	SWITCHES	SWITCHES
ALL TEXT	$C054 $C051	$C055 $C051
ALL LOW RES GRAPHICS	$C054 $C056 $C052 $C050	$C055 $C056 $C052 $C050
ALL HIGH RES GRAPHICS	$C054 $C057 $C052 $C050	$C055 $C057 $C052 $C050
MIXED TEXT AND LOW RES	$C054 $C056 $C053 $C050	$C055 $C056 $C053 $C050
MIXED TEXT AND HI RES	$C054 $C057 $C053 $C050	$C055 $C057 $C053 $C050

Figure 6.15 The five distinct combinations of video modes and the combination of switches used to activate them.

All that you have to do to activate a given video mode is to reference Figure 6.15 to see the switches you need to flip and modify the four codes above to reference those locations.

Setting the Video Mode on the Atari 400/800

So far, setting video modes has been very easy. In the COCO, all that you had to do was place specific patterns in two memory locations. On the Apple II, you flipped soft switches. Setting the video mode on the Atari is a whole other ballgame. It is still easy once you understand it, but there is a lot more information for you to cover. In the following sections, you will learn about a concept called the *display list*.

Screen Modes

Atari has always been a very innovative company. One key area where the Atari proved itself to be beyond its years in technical wizardry was in the way it handled its video display. As was discussed earlier in this book, screen memory is just like any other piece of memory in the game machine except that it is there that you store the binary data that is interpreted and translated to produce the illusion of images on the computer screen. Most computers are hard-wired to look at a specific portion of RAM and to always use that specific area of RAM for video memory. Furthermore, on most machines if, for example, you wanted to change the color of the screen from white to red, you would have to loop through all of the computer's video memory and change the data byte that represents each pixel on the screen

from a byte holding the binary combination that is used to represent white to a byte holding the binary combination that represents red. Such was the case for most computers, but once again, fortunately for us, the Atari was not like most computers.

First of all, even by today's standards, video cards that have their own processors are considered high tech. Very recently, vertex and pixel shaders were all the rage. They were considered breakthroughs in technology because they gave the programmer the ability to write small programs that could be made to run on the video card itself independently of the computer's main processor. If by today's standards, such technology is considered amazing, it is unimaginable to me what words could be used to describe Atari implementing such technology over 20 years ago. In my mind, that is the equivalent of a caveman building his own double-barreled shotgun to go hunting dinosaurs. The level of vision and ingenuity that must have been circulating through the veins of Atari employees is unbelievable.

With regards to dealing with screen modes and video data in general, all of this innovation was made possible by, and came in the form of, ANTIC. *ANTIC* was the name given to the full-fledged microprocessor that the Atari uses to manipulate video data. This processor, along with the CTIA/GTIA chip that was used to control the colors that would be used to display video data, gave the Atari a level of flexibility never before seen in any video game machine. Just as programs can be written to actually run on modern video cards, so did the Atari have the ability to write programs specifically designed to be run on the ANTIC. This microprocessor had its own language with its own instruction set and its own data area. This was a true microprocessor in every sense of the word.

When you write a program that is to be run on the ANTIC chip, this program is called a display list for reasons that will become apparent shortly. With your display list, you have the ability to do three basic things.

- You can tell the computer where to find video memory.
- You can give the computer information about what video mode (or modes) to use to display video data.
- You can specify any special video options that you want the computer to use, such as fine scrolling.

Notice the second thing that you can do with the display list. As this statement implies, it is possible for you to mix graphics modes on the Atari computer. If you were using any other computer, when we discuss the concept of a video mode, we would be discussing the concept of changing the entire video screen into one video mode or the other. As we have stated, however, we are not dealing with any other computer; we are dealing with the Atari; and so we must change our entire way of thinking. Until now when you thought of the screen, you thought of it as one big image that was displayed using one screen mode or the other. In the world of Atari, the video display is actually made up of a number of

what we call mode lines stacked one on top of the other. A mode line is a horizontal strip that extends straight across the screen from one side to the next and is *x* number of horizontal scan lines wide. The entire video display is made of a number of mode lines stacked on top of the other, as shown in Figure 6.16.

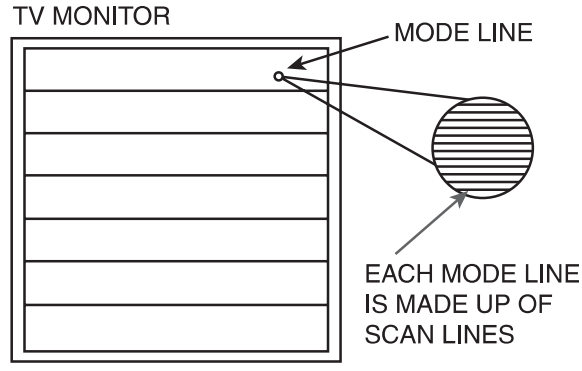


Figure 6.16 The screen mode of the Atari is made up of a number of mode lines.

You are probably thinking, "Why did they have to make things so complicated? What could you have possibly gained by this new way of doing things other than a colossal headache and a need for large supplies of aspirin?" Well, the answer is simple; each mode line can be set to a different video mode. In case you were not as blown away by this fact as I was when I first learned of it, let me explain why this is so impressive. Basically, this feature gives you the ability, for example, to create a game that has a text mode at the top of the screen, a high-resolution screen mode in the middle of the screen for displaying the game action and a low-resolution display at the bottom of the screen that can be used to display warning lights and signals to the player. This is just one of a billion uses for this feature. For now, just know that you can mix multiple graphics modes together when need be so there is no reason to limit your imagination when you are thinking about how you should lay out your video game.

One last thing that probably needs clearing up with regards to mode lines is that the height of a mode line depends on which graphics mode you set that mode line to. As we stated earlier, a mode line is *x* horizontal scan lines high. You probably wondered what that means. Well, in the back of your TV or monitor is a cathode ray tube that constantly streams electromagnetic rays and bombards those rays against the display screen. Special devices in the back of the TV are able to control what part of the screen is bombarded by rays.

Two key elements come together to give the illusion of an image on the computer screen. First, by changing the intensity of the ray, different colors can be obtained and generated on the video display. Second and most importantly for the current discussion, is the way the ray is moved across the screen to paint the picture.

The ray starts at the top-left corner of the screen and moves horizontally from the left to the top-right corner of the screen. The ray then moves back to the left of the screen and down one pixel. The ray then moves horizontally over to the right side of the screen and repeats the above mentioned pattern until it reaches the bottom of the display, at which time the ray is moved to the top left corner of the screen, and the whole process is repeated again.

Because the monitor updates the display this way so many times per second, the monitor is able to give the illusion that the whole screen is being lit up at the same time and, hence, is able to give the illusion of an image on the display screen. Each sweep of the ray across the video display is called a *horizontal scan line*. This is the fundamental measurement of height when measuring things on the screen. We call the period of time when the ray is not switched on and is moving from the right of the screen to the left of the screen and one scan line down a *vertical blank*. The time when the ray is turned off and is moving from the bottom right corner of the screen to the top left corner of the screen is called a *horizontal blank*. It takes 16,684 microseconds to draw the whole screen, the vertical blank takes roughly 1,400 microseconds, the horizontal blank takes 14 microseconds, and a single horizontal line takes 14 microseconds to draw.

You also need to know how to measure horizontal distance on the display screen. The horizontal unit of measurement is the *color clock*. The actual video display is 228 color clocks wide, but only 176 of these color clocks are actually visible. So this is generally the number used when measuring the width of the full screen. Keep in mind that it is possible to generate twice this resolution by working with what are called *half clocks*, in which case you would have a resolution of 352 pixels. This method is not often used, however, because it results in unusual color effects called *artifacts*. While the effects can sometimes be interesting, they can also produce rather undesirable effects.

That was probably a bit more information than you expected to get in order to learn about horizontal scan lines. All of that information will come in very handy though as you learn to bend the Atari to your will. Try to keep definitions such as horizontal scan line and vertical blank in your head because you will be using them a lot in your thought process.

Each mode line will vary in height, based on what video mode you set it to. Table 6.1 contains the pertinent information you need to know about mode lines of various graphic modes.

TABLE 6.1 Mode Lines and Their Scan Lines

ANTIC MODE LINE	Number Of Scan lines Produced
2	8
3	10
4	8
5	16
6	8
7	16
8	8
9	4
a	4
b	2
c	1
d	2
e	1
f	1

The ANTIC instruction set provides you with four types of instructions. The first type of instructions is called a map mode instruction. These cause the ANTIC to create a mode line that is used to display pixel data (in other words, it is designed to display pictures and not text). As you know, the video mode determines how data stored in video memory is interpreted and displayed. The Atari also has four color registers, which means that in any given map mode, each pixel can be up to any one of four different colors. In Figure 6.17 you can see that each register is represented by one of four binary patterns in a four-color graphics mode.

COLOR REGISTER VALUES		
0	0	REGISTER 1
1	1	REGISTER 2
0	1	REGISTER 3
1	0	REGISTER 4

Figure 6.17 Makeup of the color registers in a four-color graphics mode.

When in this mode, the computer interprets computer data by identifying how many colors are used in the given mode, what binary pattern is used to represent each color in the given mode, and how many pixels are represented in each byte of data. With this information the computer can correctly interpret video data and draw pictures on the screen. See Figure 6.18. Listed on the companion Web site of this book is a graph of which binary patterns represent which color registers in a given video mode as well as how many pixels are represented in each byte.

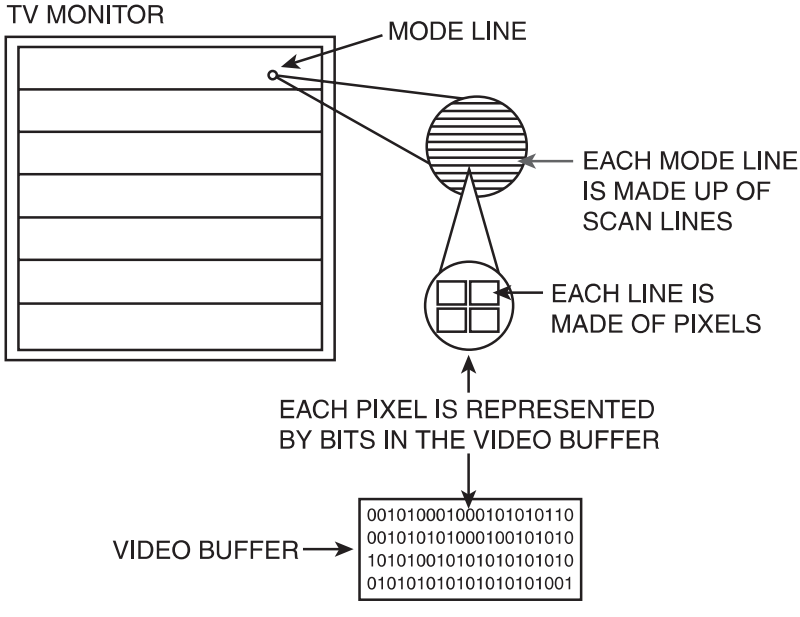


Figure 6.18 How video memory is interpreted in video mode.

The next kind of instruction, which is called a character mode instruction, creates a mode line used for displaying text characters. See Figure 6.19. While in character mode, the computer interprets video data by assuming that each byte of information represents a binary pattern that represents a letter of the alphabet, a number, or some other graphic character.

"Blank line instruction" is the name given to instructions that cause the computer to display a mode line (or lines) that does not display text or pixels but rather displays a solid bar of the background color. The eight blank line instructions that exist are used to tell the computer to display one to eight blank lines. See Figure 6.20.

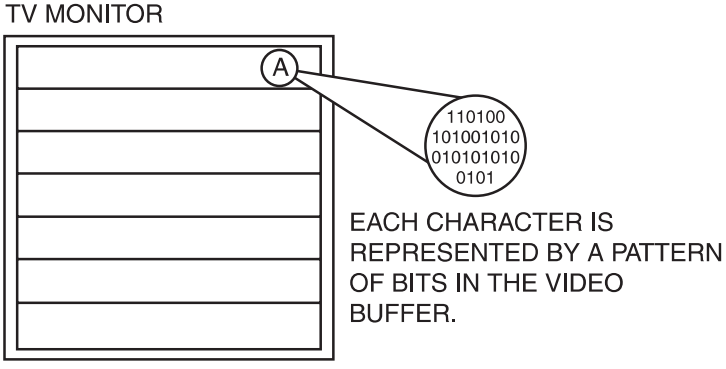


Figure 6.19 How video memory is interpreted in text mode.

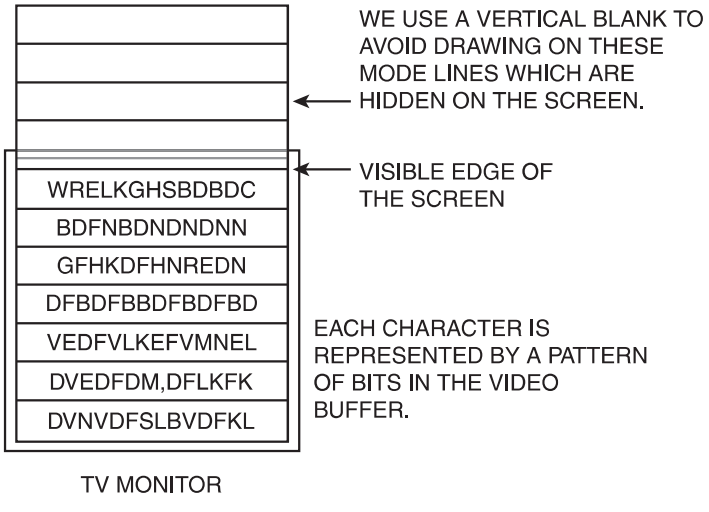
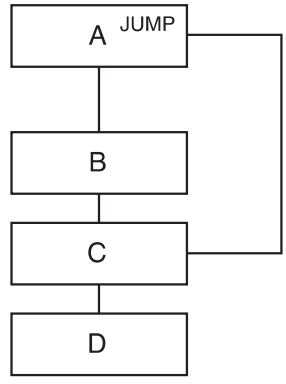


Figure 6.20 How blank line instructions are interpreted.

The final class of instruction are the jump instructions, which are used to reload the ANTIC program counter (a feature which is very handy and will be discussed in detail a bit later) and come in two distinctive flavors. The first arises out of the limitation of the ANTIC that prevents it from jumping over a boundary of 1k. If the display list program must jump over this boundary, you must use the `JMP` instruction to jump over this boundary. See Figure 6.21.



THE JUMP COMMAND CAUSES THE COMPUTER TO SKIP TO LOCATION C AND CONTINUE TO READ MEMORY LOCATIONS SEQUENTIALLY.

Figure 6.21 Normally, the computer would move sequentially through memory, but when it encounters a `jump` command, it will skip over some memory to get to a given location.

To fully understand the importance of the next instruction, it is necessary to get ahead of ourselves for a second and take a superficial look at the structure of the display list program. In your program, there is a portion where you provide a list of any combination of map, character, or blank line instructions to set up your display the way you want it. As you recall about how the monitor or TV creates the illusion of a picture on the screen, you have a ray that traces back and forth down the screen painting the picture.

You may also remember that each mode line on your screen is made up of a number of horizontal scan lines. When the cathode ray has completed drawing the screen, it jumps back to the top of the screen and starts the whole process again.

Now stop and think about this for a minute. If your screen were 20 mode lines high, you would have to write 20 lines in your display list program to set these mode lines to the correct video mode. Now the cathode ray is back up at the top of the screen ready to start drawing again, and it is relying on the ANTIC to correctly interpret video memory and generate the correct screen information to be displayed.

The ANTIC, in turn, is dependent on the display list program to know what model lines to use and what is the correct way to interpret screen memory. The problem is that you cycled through the 20 mode line instructions the first time you drew the screen and now the ANTIC is expecting you to give it another 20 mode line instructions. As a mater of fact the ANTIC will expect a new set of 20 mode line instructions each time the screen is drawn. Given that you use the exact same instruction every time to draw the screen, this would be a ludicrous method of programming the display list.

This is where the JVB jump instruction comes into play. You can use this command to wait for the vertical blank and then jump back up to the top of the display list program. This way, as the monitor's display is constantly drawing itself from top to bottom and then jumping back up to the top to start again, the display list program will also provide ANTIC with the instructions it needs. It then jumps back up to the top in an endless loop that runs like clockwork to ensure the computer always correctly displays the contents of video memory.

While they are not really instructions, there are four special options that you can use when programming the ANTIC. The first of these options is the display list interrupt (DLI), which is an unbelievably powerful feature. In order to understand the way this works, let's take a look at the following example.

You have a space invaders type game that you are creating. You want to use four colors to draw the different types of aliens. You also want to use a fifth color for your hero along with three other colors to draw the status display for your game. That gives you a total of eight colors that you want to use as seen in Figure 6.22.

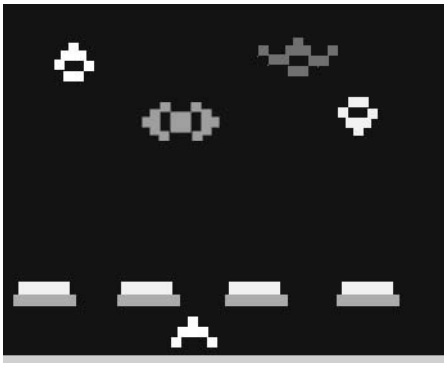


Figure 6.22 A theoretical game that uses eight colors at one time.

Now I know you are thinking that the computer only can display four colors at a time. There is no way that I can use eight colors. Well, you are half right. The computer can only display four different colors at a time, but what if you could change the four colors that the computer uses halfway down the screen? What if you could draw the top half of the screen, which holds the aliens, using four colors and then switch the four colors that you are using just before you draw the hero and the status display. This is what the *display list interrupt* allows you to do. Even though you have not gone into the actual structure of the display list program in detail, you have the basic idea of how the program works. You know that for each mode line on the screen, there is a corresponding mode line instruction in the display list program. What you have to do is design the layout of the game screen. You have to then decide exactly what half of the screen should use the first four colors and which part of the screen should use the last four colors as seen in Figure 6.23.

What you have to do now is find the line in the display list program that draws the last line on the screen using the first four colors. You are going to set the flag for the display list interrupt on this line. Ideally, this will cause the computer to stop and generate a display list interrupt.

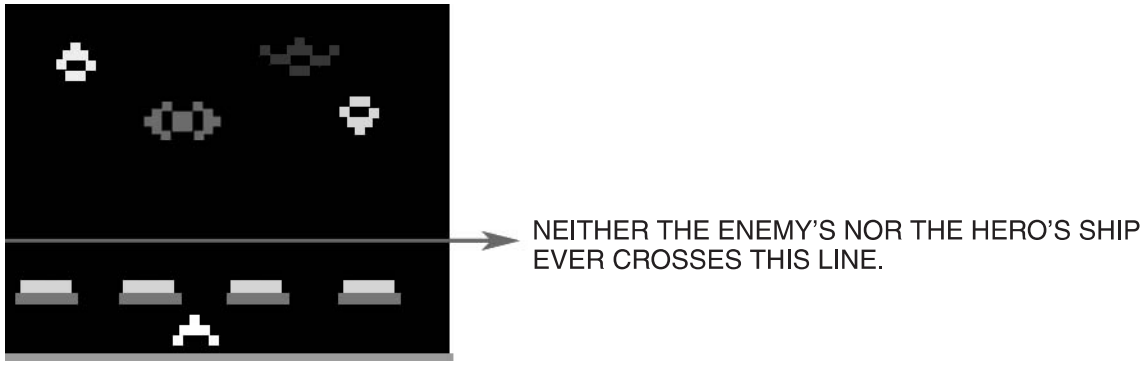


Figure 6.23 You have to plan which parts of the screen should have which colors. And use this to decide where you want to implement the change in which colors the computer uses.

Now I think we should pause here for a minute for a brief description on exactly what an interrupt is. Imagine that you are busy doing your homework or balancing your bank book or something else that requires your full attention. You have a 2-month-old baby that you are taking care of. If the baby starts to cry, you would stop what you are doing, find out what is wrong with the child and if, for example, the problem is that the baby needs a bottle, you would give the baby the bottle, burp her, and then go back to what you were doing before you were interrupted.

This is essentiality what happens when you call an interrupt on the computer system. The computer microprocessor is like you when you are busy working, and an interrupt is like the baby crying. The 6502 microprocessor, which is generally busy doing some very important task, will hear the display interrupt and respond to it just as you responded to that child. And, just as you went back to doing your work when you were done, the processor will go back to doing its normal work after it has finished responding to your interrupt. See Figure 6.24.

Building Display Lists

Finally, we get down to the point of this whole chapter, actually building display list interrupts. Atari display lists and display list interrupts offer an unbelievable amount of power to the programmer. The first step to unleashing this power is to create a simple display list interrupt.

Do you remember the way you organized the code when you were programming using BASIC? Your code looked something like this:

```
10 GR.3
20 COLOR 3
30  PLOT 3,3
```

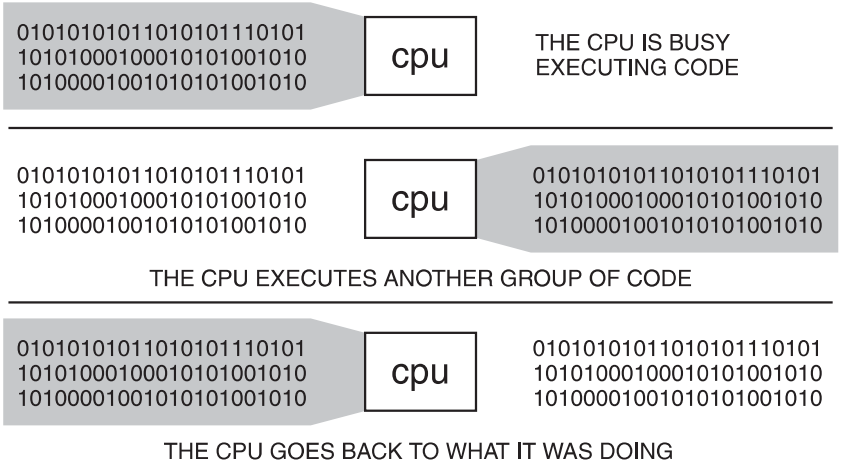


Figure 6.24 Just like you are able to stop what you are doing, attend to another problem, and then go back to what you were doing, you can interrupt the computer and have it perform some task, then go back to what it was doing.

You see that you used one line number for each command, and each command has its own line number to allow you to identify it. This also helps the computer to always know where a given command starts and ends. When you are creating the display list interrupt, you do things a bit differently. First of all, you do not use any line numbers. The ANTIC will automatically start reading the display list from the first byte of the list. Earlier, we discussed the various kinds of commands, such as character and map mode lines, `jump` commands, and so on. Most commands, including `mode line` commands, only occupy one byte. When the ANTIC comes across one of those commands, it knows that the command only uses one byte and after it executes the command in that byte, it simply has to move forward one byte in order to get to the next command.

`Jump` commands are a bit different. They occupy three bytes. The first byte is the actual command, and the next two bytes form the operand for the command; namely, they give the address for the ANTIC to jump to. When the computer comes across such commands, it knows that before it executes the command it has to read the two bytes following the command so that it knows exactly where it has to jump to; then it knows that the next byte will always be a new command for it to execute. This may sound a bit confusing, but once you get the hang of it, you will see that it really is not so bad. Figure 6.25 will help to better illustrate this point.

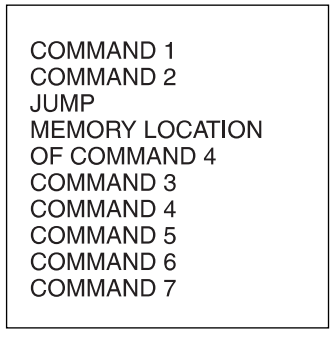


Figure 6.25 The computer can always figure out which command is next without the use of a line number.

Have you ever been watching a television program and realized that a part of the picture seemed to be cut off. Not much, just a small piece of the picture is missing; this is because of something called *over scan*. You see, the borders around the television screen actually cover a small portion of the cathode ray tube. This can be seen in Figure 6.26.

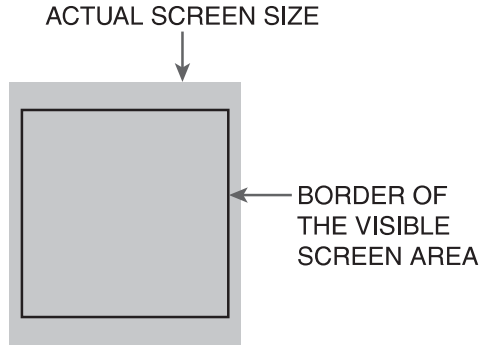


Figure 6.26 The border around the screen actually cuts off a small amount of the screen.

As you can see, the border around the screen actually covers the top 24 scan lines of the display. For this reason, if you started drawing mode lines from the first line of the display list, the top of the display will be cut off. Take a look at Figure 6.27 to see the relationship between the display list and the screen display.

The question is, how in the world do you prevent the screen from being cut off by the television's border? The answer is surprisingly simple because those blank line commands come to the rescue. You will make the first three lines of the program *blank eight line instructions*. Each of these commands will draw eight blank scan lines at the top of the screen. $3 \times 8 = 24$ so this means that the only thing that will ever be drawn under the border on the top of the screen are blank scan lines. Figure 6.28 illustrates this point.

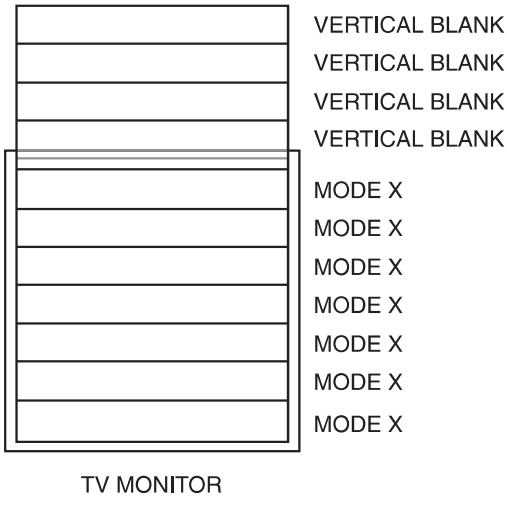


Figure 6.27 Here is a look at how your display list compares to your actual video display.

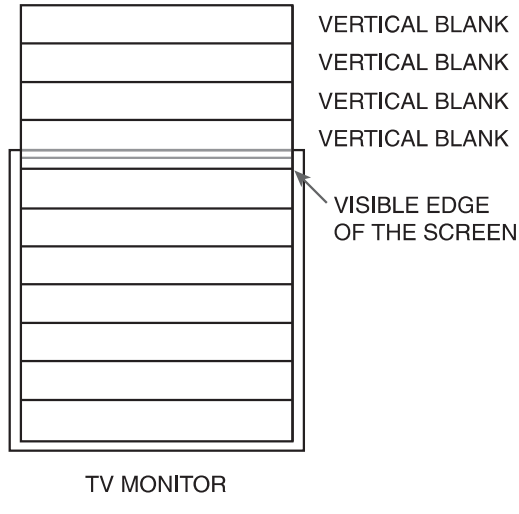


Figure 6.28 Using blank line commands, you can be sure that the display will never be cut off of the screen.

Once you are sure that the display will not be cut off by the top of the screen, it is time for you to instruct the Atari as to where to find video memory. This is done via the LMS command. LMS stands for load memory scan, and this command occupies four bits. All of the ANTIC mode line commands occupy no more than four bits. As we stated before, the display list usually reserves one byte per command. Usually, when you write an ANTIC mode command into the display list, the high bit of the command is left set to zero and the lower bit is set to the value representing the ANTIC mode line that you are working with. The fourth line of the display list is unique from every other line in the display list. On this line, you issue the first of the mode line commands, but unlike every other ANTIC mode command that you issue, you will not set the high end of the byte to zero. On the fourth line of the display list, you set the low byte to the value of the ANTIC command, but you will also set the high byte to the value that represents the load memory scan command. This way you can kill two birds with one stone. After this command is executed, two things happen. The computer knows exactly which mode type the first mode line of the display list will be, and the ANTIC now knows that the following two bytes contain the address of the memory location of the beginning of video memory. The first byte holds the low end of the memory location address, and the second byte holds the high end of that same location. So if video memory is located at 7C20, the 20, which is the low end of the address, would be placed in the first byte after the LMS command, while 7C, which is the high end of the address, is placed into the second byte after the LMS command. This concept can be seen in Figure 6.29.

Next comes what can be either the easiest or the hardest part of writing any display list. This can be the easiest part of writing the display list because all you have to do is decide which video mode lines you would like to work with and insert the corresponding ANTIC video mode command. The reason that it can be the hardest thing is because you

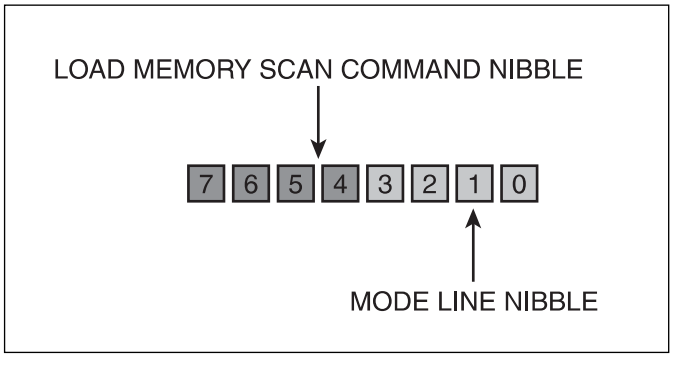


Figure 6.29 Because the load memory scan command uses only four bits, and the mode line commands each use four bits, you can use a single byte to hold each command.

have to plan exactly how the screen is going to be organized and how the display list should be organized. The best way to approach this, like so many other aspects of computer programming, is usually to start off using a little bit of pen and paper. First draw out a sample of the way that you want your screen to look. My drawing can be seen in Figure 6.30.

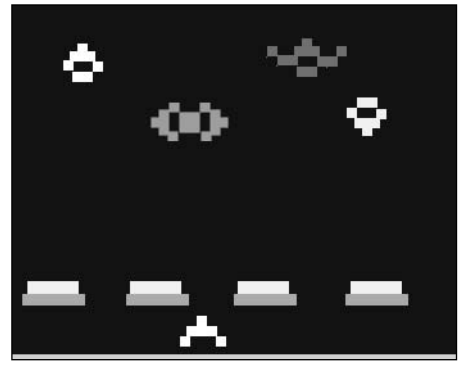


Figure 6.30 Before you create the display list, you have to plan the way you want the screen to work.

Next you have to decide on exactly what type of mode line is needed to represent each part of the screen. Now in this planning phase, it is important for you to keep one thing in mind. Each mode line that you create uses a certain amount of scan lines. When you add up all of the scan lines created by the mode lines, the total amount of scan lines can be no greater than 192. This is because the television screen is only 192 scan lines high. It is okay for the display list to have less than 192 scan lines, but you should never allow your display list to grow beyond 192 scan lines or else strange things will start to happen. Table 6.2 shows the correlation between mode lines and the number of scan lines that they produce.

What you see is, among other things, a list of the correlation between ANTIC mode lines and the amount of scan lines they produce. Armed with this information, you can now properly plan the construction of the display list. All that you need to do now is decide on how you want the screen to be arranged. For our first example, we will start off very simple. We will simulate graphics mode 2. Basically, all you need to do is create a display list where each mode line corresponds to ANTIC mode 2.

So let's get started. First, you have to insert the three blank eight mode line instructions like this.

```
70
70
70
```

Remember that when a byte in the display list contains 70, the ANTIC will interpret this as a command to insert eight blank lines. Now you have to simultaneously insert the load memory scan command and insert your first mode line command. To do this, insert a 4 into the high end of the next byte and an F into the lower end of that same byte, like so.

```
4F
```

Now you have to tell the ANTIC exactly where to find the start of video memory. Assuming that video memory is located at `7C20`, you draw a line between the `C` and the `2` to divide this address into two parts. The first part of the address is called the high end of the

TABLE 6.2 More Mode lines and Their Scan Lines

ANTIC Mode Line Number	BASIC Mode Numbers	Number of Scan Lines Produced	Number of Colors Available	Pixels per Mode Line	Bytes per Line	Bytes per Screen
2	0	8	2	40	40	960
3	none	10	2	40	40	760
4	none	8	4	40	40	960
5	none	16	4	40	40	480
6	1	8	5	20	20	480
7	2	16	5	20	20	240
8	3	8	4	40	10	240
9	4	4	2	80	10	480
a	5	4	4	80	20	960
b	6	2	2	160	20	1920
c	none	1	2	160	20	3840
d	7	2	4	160	40	3840
e	none	1	4	160	40	7680
f	8	1	2	320	40	7680

address, and the last half of the address is called the low end. Each end of the address takes up one byte. When inserting this address into the display list, you must first insert the lower end of the address and then the higher end of the address. When you insert the starting address of video memory it would look like this.

```
20
7C
```

Now what you have to do is insert all of the mode lines. The question is exactly how many mode lines do you need to insert. You know that you cannot insert more than 192 scan lines in to the display list and every time you add an ANTIC mode 2 mode line command, you automatically insert eight scan lines. This means that if you divide 192 by 8, you will know exactly how many mode lines you need to insert into the display list. This is true for any mode line that you want to insert; if all of the mode lines are going to be the same, you can just divide the number of scan lines that will be generated for each mode line by 192. The answer will be the maximum amount of mode lines you can enter in to the display list.

So going back to our example, 192 divided by 8 gives you a total of 24; what this means is that in order to fill the screen, you will have to insert a total of 24 ANTIC mode 2 line commands. You have already inserted the first mode line command in the same byte that you used to insert the load memory scan command, so you only have to insert the remaining 23 mode line commands.

```
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
```

All that's left for you to do now is a little bit of housekeeping. You have to be sure that when the ANTIC reaches to the end of the display list, it will automatically jump back up to the top of the display list, and most importantly, you have to be sure that the ANTIC jumps back up to the top of the display list at exactly the same time that the monitor is executing a vertical blank. To do this, you will use the JVB command to wait until the vertical blank occurs and then jump back up to the top of the display list. In code, the JVB command looks like this.

```
41
```

Whenever the ANTIC comes across a byte that contains a 41, it knows that the next two bytes hold the address of the top of the display list and when the monitor executes a vertical blank, the ANTIC will automatically jump to this address.

As you saw when you inserted the load memory scan command, when you enter an address into the display list, you must insert the low end of the address first and then the high end of the address. That means that if the starting address of the display list is 7BE0 then you would insert the display list like this:

```
E0
7B
```

Now your completed display list looks like this:

```
70
70
70
42
20
7C
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
24
```

```
24
24
24
24
24
24
24
24
41
E0
7B
```

What I have just given you is the basic theory behind creating a display list. Before you indulge in the actual mechanics of creating the display list, there is one more topic that we need to cover: the display list interrupts.

The irony is that even though this is one of the most powerful features on the Atari and indeed probably the most powerful feature of all the game machines covered in this book, it is completely and utterly useless all by itself. It only has value when used in conjunction with other features of the Atari such as player missile graphics. In this way, the display list is more of a catalyst that makes other features of the Atari more powerful. Now as you can imagine all of this power does not come easy. This is going to be the first real test of your ability to apply everything you learned about assembly language earlier. You are also going to have to get the hang of some very complex and intricate timing mechanisms.

Before you read any further, you should grasp one concept. There is nothing that you cannot do. There are only things that you do not know how to do yet. You can learn to do any of the things you do not know how to do if you just relax, take your time, and apply yourself. You are about to see a lot of numbers and figures. Don't be overwhelmed by them. Focus on the main body of the text first until you understand the big picture. Once you understand the main body of text, the tables and charts that you see will become much more clear and easier to understand.

How Does the Display List Interrupt Work?

By now you understand the way images are drawn on the screen starting from top to bottom. You have also been told that the Atari has certain limitations. For instance, the most colors that can be displayed by the Atari at any one time is four colors. With that in mind, think about this: the monitor itself has no limit to how many colors it can display. Whatever colors the Atari sends it, it will display. What if the Atari starts off displaying a number of horizontal bars with the colors red, blue, green and yellow? What if when the monitor is halfway finished drawing the screen, the Atari started sending horizontal bars with the colors orange, white, black, and gray? The Atari would still have worked within

its own limitation of only being able to produce four colors at one time, but by changing which four colors it was using halfway through the process of drawing the screen, the monitor itself would have a display of eight colors! That's a pretty great trick; the only question is how in the world are you going to pull it off? See Figure 6.31.

The first question that you have to answer is to find out exactly when the display has reached the center of the screen. (Actually the concept you are using can be used to change the color at any point on the screen. You are just using the center of the screen in this example for simplicity's sake.) The answer to that question is that you already know. You know because you wrote the display list that is being used to generate the display. When the ANTIC reaches the halfway point of the list, it is drawing the center of the screen. See Figure 6.32.

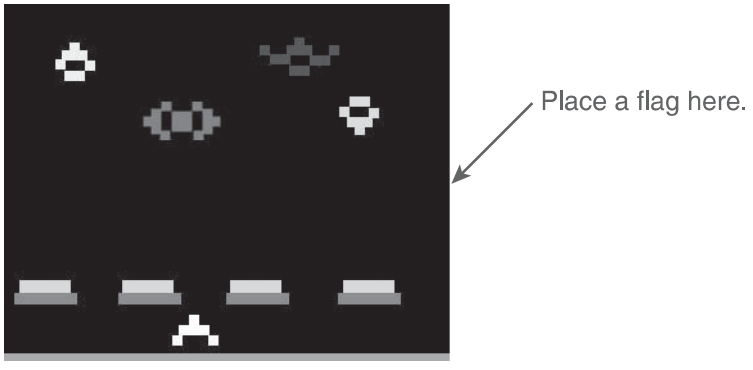


Figure 6.31 Theoretical diagram of the Atari displaying eight colors at one time on the screen.

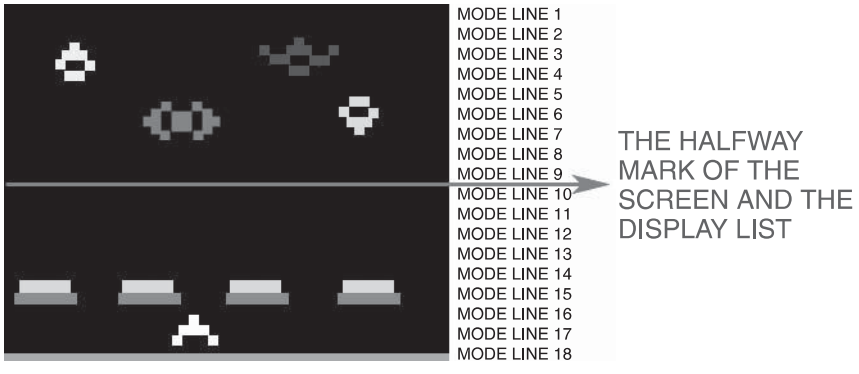


Figure 6.32 When the ANTIC has reached the halfway mark of the display list, it is at the halfway mark of the screen.

So the problem is half solved, but you still need to solve the rest of the puzzle. To do so, you must find a way to let the Atari know that you have reached the center of the screen and tell it to hurry up and change the colors it is using for its display now before the monitor draws the rest of the screen. See Figure 6.33.

This is precisely what a display list interrupt does. You insert this command into the middle of the display list. When the ANTIC reaches the line that contains the interrupt, it will interrupt the 6502 CPU of the Atari in whatever it is doing, and tell it to execute whatever commands you need executed. The CPU will then go back to whatever important work it was doing, the ANTIC will continue processing the display list, and the monitor will go back to drawing its image. Everything will go back to normal with one big difference. The color registers used to draw the display have now been changed to use four different colors. See Figures 6.34 and 6.35.

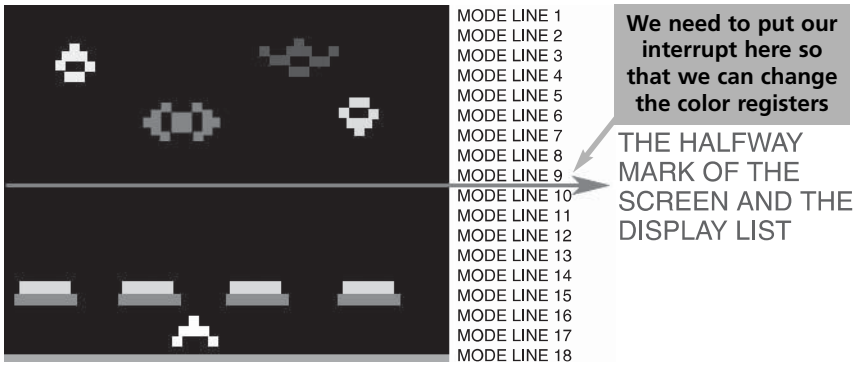


Figure 6.33 You need a way to tell the Atari that the monitor is halfway down the screen and have it change the colors that it is using.

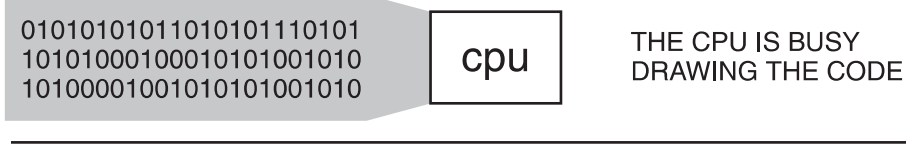


Figure 6.34 Normal operation of the computer when drawing display.

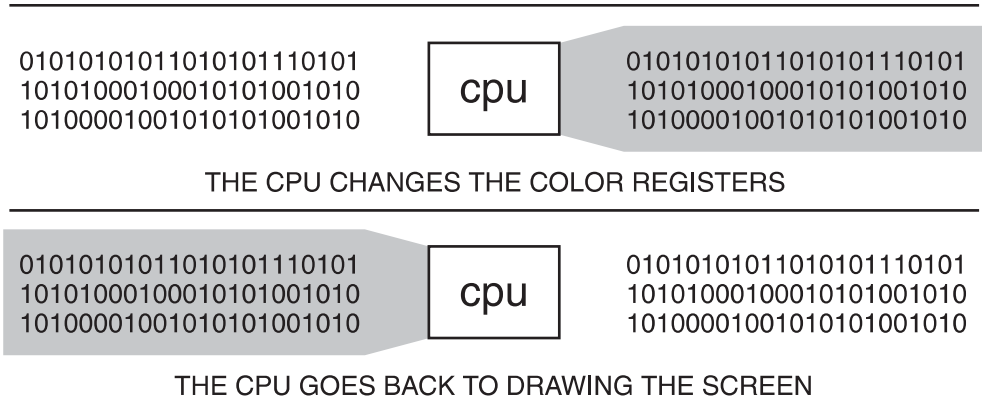


Figure 6.35 Operation of the computer when a display list interrupt is executed.

Timing Considerations

There is a pretty cool movie I like called *The Rock*, which stars Sean Connery and Nicolas Cage. There is a scene in the movie where Sean Connery has to break into Alcatraz by rolling down a corridor that was about 6 feet wide, 2 feet high, and very, very long. The catch is that this tunnel is part of a furnace, so there are huge streams of fire shooting across the corridor one after the other. In order for Sean Connery to make it through the tunnel alive, he has to time the jets of fire correctly and move only within the time frame so that he is not ever engulfed in flames.

What does the movie have to do with display list interrupts? If Connery made a wrong move in the movie, his character would have been toast. If you do not time everything correctly when working with the display list interrupt, the display will be toast. Let's take a closer look at the timing involved in creating a display list interrupt.

When you decide on exactly which mode line you will insert the display list interrupt into, you must keep in mind that the interrupt will not be called the second that the ANTIC reaches this mode line. The interrupt will be called after that line has been drawn and while the monitor is in the process of a horizontal blank. In a perfect world, this would be great because the color change would take place off the screen, giving you an even line with four colors above it and another four colors beneath it. It is not a perfect world, however, and the time that passes between the interrupt being called and the service routine that you write to change the colors is longer than the period of time during which the monitor is executing the vertical blank. This means that the color change will actually take place while the monitor is in the middle of drawing the next line on the screen as seen in Figure 6.36.

Figure 6.36 The color change takes place in the middle of the screen, producing a very unpleasant effect.

You need a way to solve this problem. The best way to solve it would be for the CPU to wait until the monitor was once again performing a horizontal blank before it changed the color registers. Fortunately for us, there is an assembly language command that will make the CPU do just that. This command is called `WSYNC`. This is short for "wait for horizontal sync." When this command is executed, the CPU basically freezes until the monitor is performing a horizontal blank. This way the color change takes place off the screen and you obtain the desired smooth line with the first four colors on top and the new colors on the bottom. See Figure 6.37.

If you were just an ordinary programmer creating ordinary programs, that would be all you need to know about timing. You are not an ordinary programmer though; you are a game programmer producing hard core games, which means that you are going to have to get much deeper into the inner workings of display list interrupts if you intend to pull off some of the cool stunts that you intend to pull off.

Figure 6.37 By freezing the CPU for a period of time, you can have the color change take place off the screen.

note

Most people nowadays know what a screen saver is. But not everyone knows why they exist. Believe it or not, they were not originally created to entertain us or even so that my son can have something fun to look at while Daddy is not typing. The reason screen savers were created was because if you leave a computer for a very long period of time without using it and the exact same image stays on the screen for that whole time, that image can actually be burned into the monitor.

An example of this could be seen on any PC that ran Lotus 1-2-3 software, which was an early spreadsheet program. When you were using this program, there would always be a bar extending horizontally across the screen and another extending vertically down the screen. After working with this program for an extended period of time, when you turned off your computer monitor, there would still be a "ghost" image of these bars on the screen. Screen savers were created so that if you left your computer unattended for a long period of time, the image on your screen would start to change constantly and so one image would not be burned into your monitor.

With the limited resources of computers from this era, the luxury of screen savers was not an option, so on the Atari 800, the engineers did the next best thing. If you leave your Atari turned on and do not touch any keys or use the joystick for nine minutes, the computer will enter what is called Attract Mode. When this happens, all of the colors on the screen will be made less bright, and they will change randomly to prevent the current image from being burned into the screen. See Figure 6.38. I mention all of this about Attract mode because when you start messing around with the display list interrupt, you stop this feature from working, which means that you have to include code in the display list interrupt service to make sure that this function still takes place. And that means that you have one more thing to worry about when you get deep into the timing mechanisms of the display list interrupt.

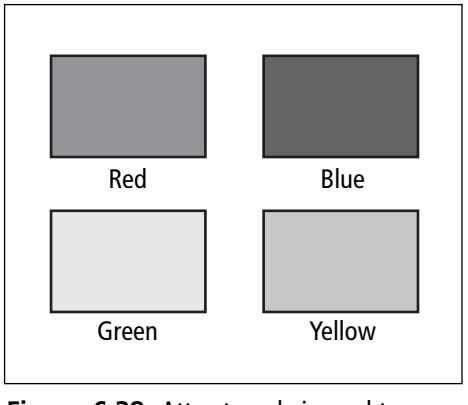


Figure 6.38 Attract mode is used to prevent the computer from burning an image into the screen.

There are three crucial periods of time that you must be concerned with when you are creating a display list interrupt. See Figure 6.39.

- The period of time between when the ANTIC first encounters the display list interrupt and when you actually use the WSYNC command.
- The period of time from when the WSYNC command is first called to the time that the electron beam first reappears on the screen.
- The period of time after the electron beam has reappeared on the screen to the time that the DLI service is completed.

note

DMA stands for dynamic memory access.

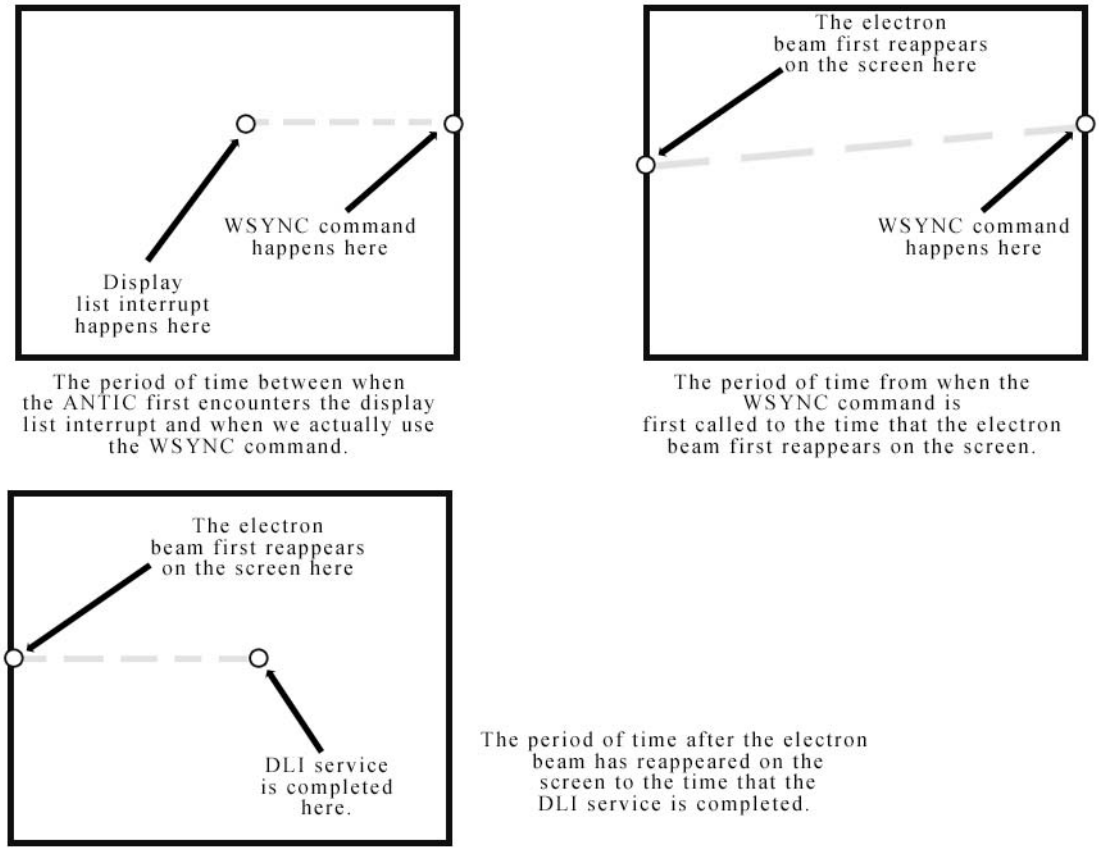


Figure 6.39 The three crucial timing phases involved in executing a display list interrupt.

In the real world, Greenwich meridian is the foundation of all time and every other time zone in the world is measured relative to this time line. In the computer, processor clock cycles are the Greenwich meridian by which all timing in the computer is measured.

When you execute a display list interrupt, every single thing that you do in regards to the first stage of this interrupt must take place in a time frame of 114 clock cycles. Why? Because this is how long it takes for the monitor to draw a horizontal scan line.

Let's take a look at the sequence of events that have to occur during these 114 clock cycles. First, it takes a whole 8 cycles just for the 6502 processor to be alerted that a display list interrupt has occurred and another 8 to 14 cycles before the processor can respond to the interrupt. Then it takes 11 machine cycles before the operating system will actually give control over to the routing that you wrote for the interrupt. That's 33 cycles already used up plus 3 cycles that will be stolen by DMA, and you have not yet even executed the first line of code in the display interrupt routine.

To further add to your already steadily growing headache, the `WSYNC` command cannot be executed any later than the 100th cycle. All of this combined with the fact that the DMA will now steal another 9 cycles means that you have a grand total of 55 cycles in which you must execute and complete the first phase of the DLI execution.

Unfortunately, you will not always be able to use all 55 of these cycles because the only way to have access to all of these cycles would be if you were displaying a blank line. This is because character mode and map mode instructions each consume one cycle for each byte that they the use in video memory. So, in a worst case scenario, such as one where you are using a BASIC mode 8, which uses 40 bytes of video memory per line, you would lose a whole 40 clock cycles. You would only have a total of 15 cycles to execute the display list interrupt.

Phase two covers a time frame of approximately 27 clock cycles. However, after you account for the loss of 5 cycles if you use player missile graphics, 1 cycle for the display instruction, 2 cycles if you use an `LMS` command, and of course, 1 to 2 cycles for our old friend `MR DMA`, you are left with a range of 17 to 26 cycles to execute phase two of the DLI.

The Keyboard and DLI Timing

There is one more very tedious timing problem that involves the keyboard. The problem is that every time a key on the keyboard is pressed, the computer beeps. No big deal, right? What in the world could this possibly have to do with the display list interrupt?

The problem arises because the timing for that beep is accomplished by the use of a few `STA WSYNC` commands. As you recall, whenever this command is executed, the 6502 processor freezes until a horizontal blank occurs. This screws up the timing of the DLI interrupt and causes the colors on the screen to jump downward for a fraction of a second. The easiest way to solve this problem is to shut off all input from the keyboard. There are other options available to you that we will look at a bit later.

Multiple Display List Interrupts

As you can see from the extreme time constraints that are imposed upon the display list interrupt, it may often be quite impossible for you to accomplish everything that you want to do in one display interrupt. For this reason, it will sometimes be necessary for you to execute more than one DLI to accomplish your goals.

There is one very big gotcha that you have to overcome in order to implement multiple DLIs. There is only one interrupt vector, and you are going to need more than one interrupt routine.

note

You know that when the CPU is busy performing an operation, an interrupt can cause the CPU to momentarily move from what it is doing and perform some other operation. When this happens, the CPU needs to know the memory location of the net line of code that it has to execute. This is where a display list interrupt comes into play.

An interrupt vector in a memory location points to the next line of code that the CPU needs to execute once an interrupt has been activated. Let's say that you have an interrupt vector stored at memory location `$10`. Whenever an interrupt is executed, the computer goes to memory location `$10` to find the location of the next memory location that it needs to execute.

Because you only have one interrupt vector, you need a way to execute the correct code each time an interrupt is called. There are a number of options that you can use to solve this problem. Most solutions, however, would require you to add more code to phase one and two of the display list interrupts execution, which would further detract from the amount of time that you have to actually execute the display list routine.

A better solution would be for you to find a way to handle this little problem in phase three of execution, where you are not overburdened by time constraints. As it turns out, there is a very straightforward way for you to do just that.

You see, the DLI vector is stored at the memory location `$200, $201`. That is, this location holds the address that the computer should jump to in order to find the display list routine. What if, after the display list interrupt had completed its main task and had entered phase three, you had it store the value held at address `$200, $201` and then write the address of the next DLI routine into that same memory location. The next time a DLI was called, it would go to the location of the second routine and execute its code. After this routine had done its primary job and had entered phase three, if you have a third routine, you could place the address of this routine into memory location `$200, $201` so that it could be executed next, or you could have it return the original address to that location so that it would execute the first interrupt routine. Using this technique, you could cycle through as many different routines as needed to get the job done. See Figure 6.40.

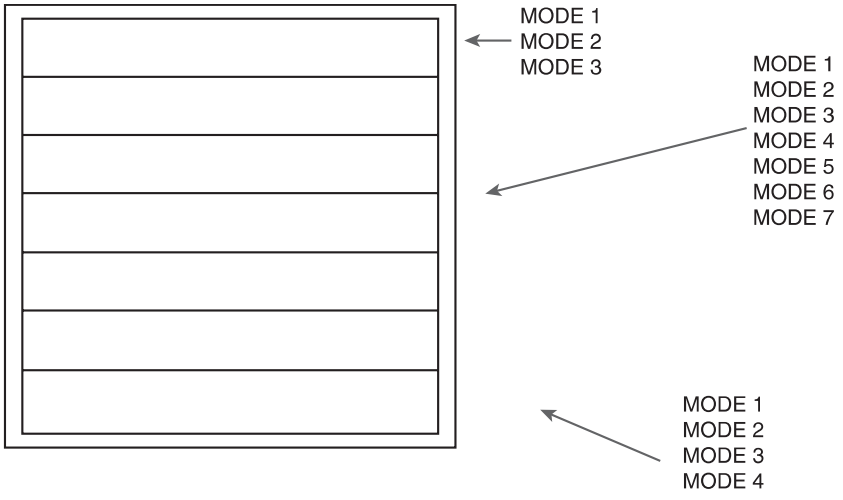


Figure 6.40 Once you know how to cycle through various routines, you can synchronize your routines to be executed at specific parts of the screen.

Placing the Text Window at the Top of the Screen

The first thing that you are going to do is create some custom display modes using your newfound knowledge of display lists. The first game that you created in the video game primer had a text window at the bottom of the screen. As you might have guessed, the next time you build this game, you are going to use this area to display the game's score, player status, and other such information. There is only one problem. The normal place for such information to be placed is at the very top of the screen. Of course, you can put this information anywhere on the screen that you wish, but as game programmers, the mere fact that the computer is designed to display text only at the bottom of the screen by default is enough to make you loathe the very idea of creating a game with text at the bottom of the screen. Blech. That would leave a very bad taste in our mouths, so let's start by creating a basic program that would allow you to place the text window at the top of the screen.

caution

Please, always be very careful whenever you are entering lines of code. Remember if you already have a program in your computer's memory and you enter test code like the code in the sidebar without first saving your program and executing a `NEW` command to clear your memory, the new program that you type will become a part of the old program that is in memory. So remember to save your work and clear memory before you enter any test code. Also remember to enter a new command when you are done with your test code so that memory is free for you to enter your next program. In you do not do this, strange things may start to happen.

Cute Little Tricks My Atari Taught Me.

You do not always have to manipulate the computer's display list to achieve the effect that you want. For example, normally when you enter graphics mode 3, there is a text window on the screen. What if you do not want that window to be there? What if you want the whole screen to be in graphics mode without a text window? The solution is surprisingly simple; all you have to do is add 16 to the number that you use in the graphics command. That is to say that if you would normally use the following command to enter graphics mode 3:

```
10 GRAPHICS 3
```

In order to display this graphics mode with no text number, you would use this command:

```
10 GRAPHICS 3 +16
```

which is really the same as entering

```
10 GRAPHICS 19
```

Now before you go and try this, remember that this does not work well in immediate mode. If you just type GRAPHICS 19 in immediate mode you will just see the screen flicker and return to graphics mode 0.

In order to see this work, type out the following short program:

```
10 GRAPHICS 19
20 GOTO 10
```

Press the Break button on your Atari or the Pause/Break button on your PC if you are using the emulator to stop this program from running after you are satisfied that it works.

tip

In order to save your work, use the following command:

```
SAVE "D1:FILENAME.BAS"
```

This will save your work to floppy disk one. Of course, you would replace filename with whatever name you wish to call your file, and if you were saving to disk 2 or 3 or any other disk, you would change the number after D to whatever drive letter you are using.

Create a Generic Display List

Since you have already started working with graphics mode 3, you will continue using that as the basis of the program you will create. This will not be a game in itself but rather an exercise for you to put all of this knowledge that you now have to practical use.

Before you get started, be sure to clear your computer's memory by using the `NEW` command. Change your display to a blank graphics mode 0 display by entering the following command:

```
GRAPHICS 0
or
GR.0
```

Now you are ready to rock and roll. Here is the first line of code for your program:

```
10 GRAPHICS 3
```

As you recall, this command will tell the computer to switch to graphics mode 3. Even though it is possible for you to write the display list completely from scratch, it is often easier, especially in the beginning, to start the display list by manipulating the display list that the computer has created for us. This is true for a number of reasons, such as the fact that the computer places the display list into memory for us, which means that the list will be stored in a relatively "safe" portion of memory. By safe, I mean that it is less likely that the computer will overwrite the display list with some other information.

Find the Location of Your Display List in Memory

Now that you have gotten the computer to create a generic display list for us, the first thing you need to do is glean some information from the display list that is already in memory. This first thing that you need to find is the location of the display list in memory. Fortunately, this task is much easier than it may seem. When the computer creates a new graphics mode, it does so by creating a generic display list. Once this list is created, the computer stores the location of the display list at memory locations `$0230` and `$0231`. `$0230` holds the low byte of the address and `$0231` holds the high byte of the address. So, if you read the data in these memory locations, you will know how to find the display list. Here is the code to do just that:

```
20 DL=PEEK(560)+256*PEEK(561)
```

The first thing that you will probably notice about this line of code is that neither the address `$0230` or `$0231` are found anywhere in the code. What's wrong? Did you forget to put it there? No, the answer is that both of these addresses really are there; it is just that they are written in binary format.

Do this from the Start menu on your computer; choose Programs (or all programs if you are using XP), Accessories, and finally Calculator. See Figure 6.41. Select View from the panel at the top of the calculator's window and make sure that Scientific is selected. Now make sure that the HEX option is selected.

Type the address that holds the low byte of the address of the display list.

Now click on the Dec option.

This will change the hexadecimal address that you just entered into its decimal equivalent. As you can see, the equivalent of `$0230` is 560, which is the value that you used in the first `PEEK` command in the line of code. What that means is that you are actually reading in the low byte of the address of the display list!

If you repeat the same procedure using the hexadecimal address of the high byte of the display list (`$0231`), you will find that its decimal equivalent is 561. This means that the second `peek` command reads in the high byte of the address holding the display list.

So basically what this line of code does is multiply the high byte of the address by 256 and add it to the low byte of the address. This is just a little formality that you have to carry out to reconstruct the address of the display list. Once the address is reconstructed, it is stored inside of the variable DL.

As you will soon see, you can now use this variable `DL` to reference any part of the display list and read or manipulate it in any way that you want.

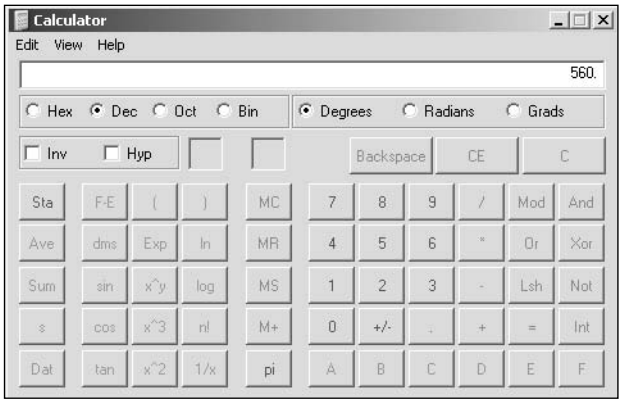


Figure 6.41 The Windows calculator in Scientific mode.

Find the Start of Video Memory

Take a look at the following list of a mode 3 display:

```
112
112
112
72
112
158
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
8
66
96
159
2
2
2
65
78
158
```

We have already discussed the nature of the display list, so you should understand the basics behind how this one works. We do, however, have to take a look at a few points that make this display list different from the ones we have looked at before. We will point out the difference as we move along.

Look at line 4 of the display list above. You know from experience that this line is a combination of a load memory scan command and the first mode line instruction. You can find out what mode line is held in this line by subtracting 64 from it. (The number for this line is obtained by adding 64, which signifies an LMS command and the number for the mode line that you want displayed as the first mode line of your screen.) 72 – 64 = 8, so you know that this line holds the first mode line instruction on the screen. You also know that the following two bytes contain the address of video memory. The first byte after the LMS command is the low byte of the address of video memory, while the second byte holds the high byte of the address.

You need a way to read these specific memory locations and save them. This is where the DL variable comes into play. You see, if you were to use the following command in the program:

```
PEEK (DL)
```

it would return a value of 112, which is the value of the first byte in the display list. This represents the first blank eight-line instruction. If you were to use the POKE command, you could actually change the value located in the first byte of the memory location. This is the basic principle behind how you are going to both glean information from the current display list and write the new display list.

You may have noticed a slight problem. You have a way to read and manipulate the first byte of the display list, but how are you going to manipulate the rest of the display list? Well, like so many other things you have done so far that looked complicated until you actually did it and found out how easy it is, the answer is really very simple. If you PEEK or POKE to memory location DL you will get the first byte of the display list, but if you PEEK or POKE to memory location DL+1, you can reference the second byte of the display list. PEEKing or POKEing to memory location DL+2 will reference the third byte of the display list. Do you see a pattern here? You can use indirect addressing to reference any part of the display list that you want. You may have also noticed that the number that you add to DL is always 1 number less than the number of the byte you are actually trying to reference.

Let's put this theory to the test by reading the address of video memory, which is located immediately after the load memories scan command.

The first three bytes of the display list are the blank eight mode line instructions. The fourth byte of the list is the actual LMS command. This means that the fifth byte will hold the low byte of the address of video memory while the sixth byte holds the high byte of the address of video memory. So to reference and read the low byte of the address of video memory, you would have to use the following command:

```
PEEK (DL+4)
```

To find the high byte, you will have to use this command:

```
PEEK(DL+5)
```

Here is the code that you will use to read in the memory location of video memory and store it into a pair of variables.

```
30 LMSLB=PEEK(DL+4)
40 LMSHB=PEEK(DL+5)
```

Line 30 above loads the low byte of the address of video memory and stores it into the variable `LMSLB`. Line 40 saves the high byte of the same address and stores it into the variable `LMSHB`. See Figure 6.42.

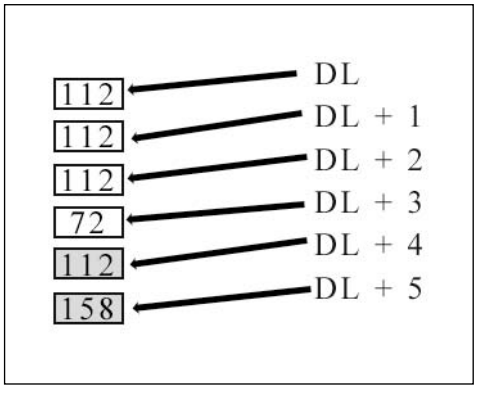


Figure 6.42 You can use `DL+4` and `DL+5` to reference the low and high byte of the address of video memory.

Text Editor Memory

There are two ways for the computer to get information to put on the screen. The first way, and the way that we have discussed the most up to this point, is to read video memory, translate it, and draw it on the screen. There is, however, another way for the computer to get information to place on the screen, which is to reference text editor memory. You see, video memory basically stays the same until you either use an operating system graphics command (e.g., `PLOT` or `DRAWTO`) to manually manipulate the memory location or change the graphics mode of the screen (i.e., create a new display list).

What this means is that if the only way for the computer to get information to display on the screen is from video memory, then you could bang on the computer's keyboard all day long and nothing that you typed would ever appear on the screen. This is because when you type something on the computer screen, the information that you type goes into a special computer buffer that has absolutely nothing to do with video memory. See Figure 6.43.

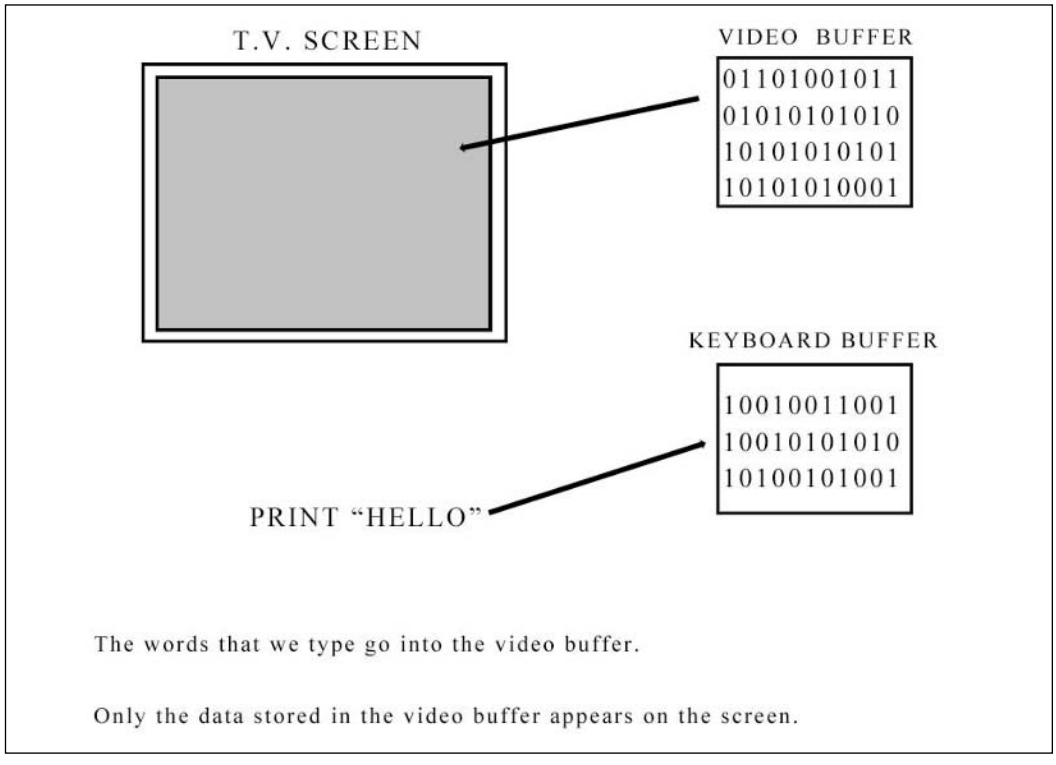


Figure 6.43 Normally, the things you type on the keyboard would never appear on the screen because it ends up in a completely different portion of memory than video memory.

Everything you type on the screen and every message that the computer wants you to see ends up in what is called *text editor memory*. You write your own custom display list, and you want to use text lines that will display text as you type it. (This will be graphics mode 0.) You will have to direct the computer to use text editor memory before it starts displaying the ANTIC mode 2 mode lines. This concept is seen in Figure 6.44.

Take a look back at the display list in the preceding section; near the end, you will see a 66. This is a command used to instruct the computer to start using memory at the location stored in the following two bytes to get the information to be displayed on the screen. This memory location is the start of text editor memory.

What you need to do is store the 27th and 28th byte of the display list so that you know how to find it if you decide to use it later. In order to reference the 27th and 28th byte of the display list, you will use the indirect addresses `D+26` and `DL+27`, respectively.

Here is the code to do just that:

```
50 TXTL=PEEK(DL+26)
60 TXTH=PEEK(DL+27)
```

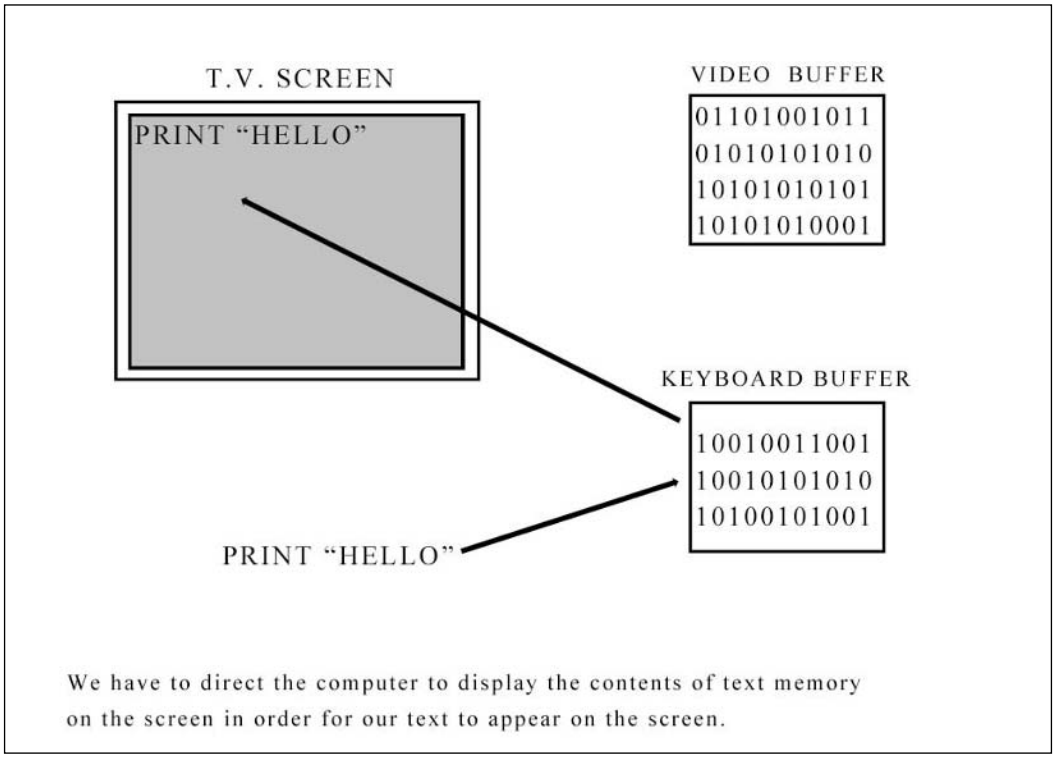


Figure 6.44 You have to direct the computer to use text editor memory before you give it the ANTIC mode 2 instructions so that what you type will appear on the screen.

Reading and Changing the Address of the Top of the Display List

The very first thing that you did when you started working with the display list was to find the location of the start of the display list. This was easy because the computer stored this information at memory location `$0230` low and `$0231` high.

Even though you can read the location of the display list from this location, you cannot change the location of the top of the display list by writing to this address location. This is because this is only a shadow address and not the actual copy of the address the display list will use to jump up to the top of the display list. This concept is demonstrated in Figure 6.45.

What this means is that if you want to change the address that the display list jumps to when the screen is executing a vertical blank, you need to change bytes 32 and 33 of the display list. Things can get kind of crazy once you really get going manipulating the dis-

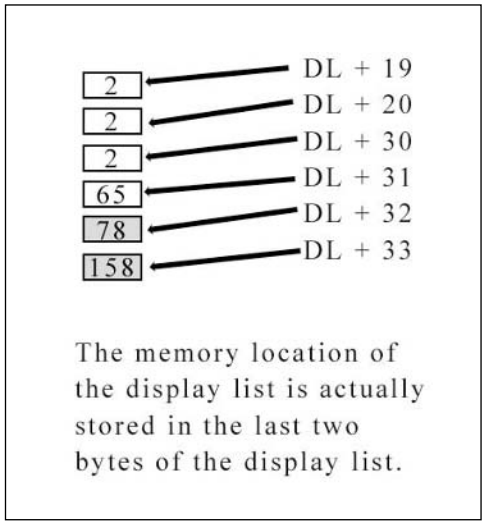


Figure 6.45 Memory address $0230 and $0231 holds a shadow copy of the location of the top of the display list. When the ANTIC chip is looking for the address to use to jump back over to the top of the screen, it does not look at the shadow copy of the address but instead uses an address that is stored in the last two bytes of the display list.

play list. For this reason, the first thing that I would like to do is create two variables and store the actual address of the top of the display list as it is stored in bytes 32 and 33 of the display list. An example of this can be seen bellow.

```
70 LSTL = PEEK(DL+31)
80 LSTH = PEEK(DL+32)
```

tip

You could just PEEK at addresses 560 and 561 to get the shadow copy of the address of the display list, but when things get hectic, it could be easy to have your program start doing strange things. This is because you are trying to find the display list at a location stored in your shadow address after you have already changed the actual address of the display list.

Table 6.3 Mode 3 Display List Information

Portion of the display list	Location in the display list	Value used to reference this item in the display list	Variable used to store data
Shadow copy of the location of the display list	Not found in display list. Low byte: memory location	Not referenced in the display list.	DL
Location of video memory	$0230 High byte: Memory location 40231	Low byte: DL+4 High byte: DL+5	Low byte: LMSLB High byte: LMSHB
Location of text editor memory	Low byte: 5 High byte: 6	Low byte: DL+25 High byte: DL+26	Low byte: TXTL High byte: TXTH
Actual location of the display list	Low byte: 26 High byte 27	Low byte: DL+31 High byte: DL+32	Low byte: LSTL High byte: LSTH
Load memory scan command	Low byte: 32 High byte: 33	DL+ 3	None
	Byte 4		

Creating Your New Display List

You now have all of the information that you need to create your own display list. You know a safe place to store the display list (the same location in memory where the operating system had stored the original mode 3 display list), you know where video memory is stored, and finally, you know the current address of text editor memory.

caution

You may be tempted to find out the actual memory location of video memory, text editor memory, and/or the location of the display list and hard code this location into your code. Never, ever, do this. You see, depending on a number of variables (the most important being the amount of memory the computer has) the location of these values will change from computer to computer. This means that if you hard code these values in your game, it may work just fine on your computer but then just completely bomb when someone else tries to run that same code on her computer.

Let's now put this vast amount of theory we have been studying for the past two chapters to use and complete the task of creating the display list.

The Load Memory Scan Instruction

Another benefit of using a predefined display list as the starting point of your own is that the first three bytes are already created for us. That is, of course the three blank eight mode line instructions. The fourth byte of the display list is the load memory scan, the first of the mode line commands. To calculate the correct value to place in this line, use the following formula.

64 (which is the value that signifies the `LMS` command)
+ The ANTIC mode number you want for the first line of the display

You want the first four rows of the screen to be displayed in ANTIC mode 2, which means that the first of the mode lines in the display list has to be ANTIC MODE 2. And that means that you have to add 64 +2 to obtain the correct value for the fourth byte of the display list. The correct value for you to use is 66. Looking at Table 6.3, you see that you have to use the equation `DL+3` in order to reference the fourth byte of the display list, which is the `LMS` command. You alter the `LMS` instruction like this:

```
90 POKE DL+3, 64 + 2
```

You could have just used this code:

```
90 POKE DL+3, 66
```

You can type out this instruction in whatever way is most comfortable to you. I would suggest that while you are just starting out that you use the first method just so you remember how you obtained the value for the display list.

Originally, the load memory scan command in this display list pointed toward video memory. For our example, however, you want to point the computer toward text editor memory so that whatever you type will end up on the top of the screen. Looking once again at Table 6.3, you see that you have stored the address of text editor memory in the variables TXTL and TXTH (the low byte and high byte of the address of text editor memory). What you need to do is replace bytes 5 and 6 with the high and low byte of the text editor memory. Here is the code:

```
100 POKE DL+4,TXTL
110 POKE DL+5,TXTH
```

Inserting the Remaining ANTIC Mode 2 Lines

The next three bytes of the display list contain the instructions to display the next three ANTIC 2 mode lines of the display list. Remember you already inserted the command for the first mode line on the screen when you executed the load memory scan command.

To reference these bytes you will use the equations `DL+6`, `DL+7`, and `DL+8`, and `POKE` a value of `2`.

```
120 POKE DL+6,2
130 POKE DL+7,2
140 POKE DL+8,2
```

A Look at What You Have So Far

The following listing is the result of your work so far:

```
5 REM code listing RG091EC.bas
10 GRAPHICS 3
20 DL=PEEK(560)+256*PEEK(561)
30 LMSLB=PEEK(DL+4)
40 LMSHB=PEEK(DL+5)
50 TXTL=PEEK(DL+26)
60 TXTH=PEEK(DL+27)
70 LSTL = PEEK(DL+31)
80 LSTH = PEEK(DL+32)
90 POKE DL+3, 64 + 2
100 POKE DL+4,TXTL
110 POKE DL+5,TXTH
120 POKE DL+6,2
130 POKE DL+7,2
140 POKE DL+8,2
```

Your screen should look something like that shown in Figure 6.46.

Figure 6.46 You should now have text lines at both the top and bottom of the screen. Whatever you type on the keyboard shows up on both the top and bottom of the screen.

The screen looks quite strange, doesn't it? You have text lines at the top and bottom of the screen. What's more, whatever you type appears at both the top and bottom of the screen. What's going on? The display list originally had four ANTIC mode 2 lines at the bottom of the list. Also, originally, just before these text mode lines, the list had a command for the ANTIC to start using text editor memory to fill these four lines with text.

What you have done is to place four ANTIC mode 2 instructions at the top of the display list and instructed it to fill these four lines with information from text editor memory as well. So you see what is happening on the screen is not all that strange after all. While this is a pretty cool little trick, you are not done yet. You still have to remove the four ANTIC mode 2 commands from the bottom of the screen, plus you have another problem: to try to use the `PLOT` command to draw a dot on the screen.

```
PLOT 6,6
```

You will get the following error message:

```
ERROR- 133
```

You see, when you start messing around with the display list, it becomes a lot more complicated to actually create the graphics for your displays. With some cases that prove to be exceptions, you can no longer use the standard graphics commands provided to you by the operating system. Before you take a look at displaying graphics in your custom display modes, let's complete the display list.

Switching Back to Video Memory

Now while you want to see the text that you type displayed in the top four lines of the screen, you do not (and cannot) want to have this information used for the display in the graphics portion of the screen. This means that you have to direct the ANTIC to once again start using video memory so that the graphic part of this screen can be filled with just graphics and not garbage caused from trying to convert text data to video data.

To accomplish this goal, you must execute another load memory scan instruction. This time, rather than giving it the address of text editor memory as its operand, you will give it the address of video memory.

```
150  POKE DL+9,64+8
160  POKE DL+10,LMSLB
170  POKE DL+11,LMSLB
```

As you can see, line 150 executes a load memory scan command. This time, however, you add 8 to the `LMS` command because you want the next line as well as all the remaining mode lines of the display to be ANTIC mode instructions so that you can draw the graphics for whatever game you are creating.

Lines 160 and 170 POKE the address of video memory immediately after the new LMS command as its operands so that the computer will begin using video memory to fill the rest of the screen with graphical information.

Inserting the Remaining Mode Lines

Now that the computer has once again been instructed to start using video memory, you have to be sure that the remainder of the mode lines in the display are graphic mode lines and not text mode lines. The question is exactly how many mode lines do you need to insert into your list?

Well, to try and figure this out, let's take a look at what we already know about this problem. Each mode line that you add to the display list adds a given number of horizontal scan lines to the screen. You know that you cannot add more than 192 horizontal scan lines to the display. You know that each of the ANTIC mode 2 lines that you have created has added 8 horizontal scan lines to the display. You have added 4 of these ANTIC 2 mode lines so you have already added a total of 32 scan lines to the display, leaving you with a total of 160 scan lines to fill.

Each ANTIC mode 8 line instruction that you add to the list will add another 8 lines to the display. If you divide 160 by 8, you will get your answer as to how many more scan lines need to be inserted. You need to insert 20 ANTIC mode 8 instructions to your list to complete the display.

This is an example of a time when a for loop comes in handy. You could, of course, write 20 lines of code, each one poking an ANTIC mode 8 instruction into the list, but you are a budding hacker. So you have to do this with style. Observe the following code.

```
180  FOR I = DL+12 TO DL+30
190  POKE I,8
200  NEXT I
```

This is not rocket science but it is effective. You create a for loop. In this loop, the value of I starts with a value of DL+12. This corresponds to the 13th byte of the display list. The loop will keep going and incrementing the value of I until it reaches a value of DL+30. This value corresponds to the 31st byte of the display list. With each loop that is executed, the computer is given a command to POKE the number 8 at the location stored in I during that particular iteration of the loop.

The net effect of this code is that the computer will continue looping until it fills the next 20 lines of the display list with ANTIC mode 8 instructions.

Polishing Off the Display List

You have one more little formality to get out of the way before you can consider your display list to be done. The last thing that you need to do before you can sign off on this list

is to add a `Jump on Sync` instruction that will cause the computer to jump to the top of the display list every time the screen is redrawn.

```
210  POKE  DL+31,65
220  POKE  DL+32,LSTL
230  POKE  DL+32,LSTH
```

Line 210 is the `jump on sync` command. Lines 220 and 230 give this instruction the address of the top of the display list, which you stored earlier so that it knows exactly where to jump to.

And Then There Was Light

You have just completed your first display list. Now let's put it to some use with a simple application. Add the following lines of code to your program:

```
240  COLOR 1
250  PLOT 0,0
260  DRAWTO 39,0
270  DRAWTO 39,19
280  DRAWTO 0,19
290  DRAWTO 0,0
```

Load the listing for program RG0902.bas or type in the code from this program and run it. As you can see, the program draws a border around the graphics area of the screen and the text window is located definitely at the top of the screen.

A More Advanced Display List

That display list does not exactly take full advantage of much of the power of the display list. It was just a taste for you to cut your teeth writing your first display list programs. Now let's use the power of display lists to do something a bit more practical.

Do you remember the game we created in the BASIC primer? All of the graphics were done in graphics mode 3. Let's spice things up a bit and take a look at how you could use display lists to improve the overall feel of that game.

You can use a combination of mode lines to enhance the game for the player. You can use a high-resolution graphics mode for the main game area, which gives you the option of a much more attractive playing field. Beneath the main playing field, you can have text mode display the tiles for the status display and score. Finally, you have another high-resolution mode at the bottom of the screen that can be used to display the number of lives the player has left.

Creating a Generic Display List

In the display list you created above, you started off by first giving the computer a command to switch the display to graphics mode 3, which it did by creating its own display list. You then gleaned all the information that you needed from the display list that the computer had already provided for you and used it to create your own custom display list. You are going to use this same basic concept to generate the new display list but with a few changes.

First of all, graphics mode 3 only requires 240 bytes of memory to draw the whole screen. A screen drawn in graphics mode 7 with no text window requires 3840 bytes of data. What this means is that if you tried to create your new advanced display list based on graphics mode 3, the computer would not generate enough memory to hold either display list of the video memory that you will need for your display. To get around this you have to examine the display you are trying to create and identify what kind of mode line is going to take up the most space in the list and hence take up the most memory.

A display list occupies some amount of screen memory. The larger the list, the more memory it needs. If you know how much memory your custom display list is going to need, it makes sense that you should first give the computer the command to generate a display list for a standard graphic mode that uses the same amount of memory that your custom display list will need. This gives the computer the headache of allocating memory and carrying out other housekeeping issues that need to be addressed.

The second little twist that you need to be aware of is that as you begin to work with larger and larger display lists, such as modes 7 and 8, you face the problem of display lists that are larger than 4K in length. The problem is that the ANTIC cannot address more that 4K of memory at a time. So each time the computer is about to reach the 4K mark, you have to execute a new `LMS` command to reset it and give I the address of the remainder of the display list. You have to be careful when you manipulate the original display list; you have to be really careful not to write over the second `LMS` command or strange things will start to happen to your display.

For your new display list, you will set the computer to mode 7 without any text lines and work from there.

Here are the first few lines of the program:

```
10 GRAPHICS 7 +16
20 DL=PEEK(560)+256*PEEK(561)
30 LMSLB=PEEK(DL+4)
40 LMSHB=PEEK(DL+5)
```

It is almost exactly the same as the previous program except that this time you have not stored the location of text editor memory because this display list does not use it. Also you

have not stored the address of the top of the display list. Just to show you that you can always use different methods to do the same thing, this time you have referenced the shadow locations to identify the location of the list. The only catch to this method, as I have said before, is that if you change the address of the display list for some reason, you will have to update the shadow registers with its location.

Also note that you have added 16 to the number you gave to the graphics command. This will cause the graphics mode to be generated without any text lines at the bottom of the screen.

Inserting Text Mode Lines

Now you have to calculate exactly where in the display list you need to insert the instruction of the ANTIC mode 6 commands. This will cause this portion of the screen to display text as opposed to graphics.

You want to insert your text line about four lines up from the bottom of the screen so that you have enough room to draw the graphics for the display at the bottom of the screen.

You have to do a few calculations at this point. First of all, you should know something that you need to be aware of when you are mixing graphic mode lines. You have to mix the modes in a particular combination. First, you have to examine the mode lines in the list to see which mode line uses up the most RAM per line. In our case, graphics mode 7 (ANTIC mode d) uses up the most memory per line, taking up 40 bytes. Graphics mode 1 lines only use up 20 bytes per line. In order for your display to work correctly, you are going to have to combine all of the mode lines that are smaller than the mode line that uses up the most memory in groups that equal the same amount of bytes as the largest mode line in your list.

Let me explain. The display list is currently made up entirely of ANTIC mode d lines. If you want to insert an ANTIC mode 6 line, you cannot simply insert one ANTIC mode line between two ANTIC mode d commands. You have to always insert two mode lines together or multiples of two model lines together. First you must identify the mode line that uses the most bytes. Next each mode line that you insert into the list that is smaller than the largest mode line must be inserted in such a way as to be sure the total amount of bytes used by the group equals the same as the amount of memory used by the largest mode line in the display.

Every time you insert a pair of ANTIC mode 6 lines you add a total of 16 more scan lines to the display. Each ANTIC mode d line that you insert in to the list adds two scan lines to the display. If you are going to insert four ANTIC mode d lines to the bottom of the screen that means a total of eight scan lines will be added to the screen. This means that you have to start inserting the ANTIC mode 6 lines 24 scan lines from the bottom of the screen.

The screen has a total of 192 scan lines, and each ANTIC mode d line uses up two scan lines. We discovered earlier that if we divide the total number of scan lines on the screen by the number of scan lines used by each mode line in the display list, we can find out how many mode line instructions are in the list. That said, 192 divided by 2 gives you 96 ANTIC mode lines all together in the display list.

`DL` references the first byte of the display list, which is one of the blank mode 8 commands. `DL+1` and `DL+2` reference the other two blank 8-mode instructions. `DL+3` is the load memory scan instruction, which also includes the first ANTIC mode d command. `DL+4` and `DL+5` reference the low and high bytes of the load memory scan command.

It is not until you reach `DL+6` that you reach the second of the 84 mode line commands. (Remember the first one was found at memory location `DL+3` along with the `LMS` command.) Keep in mind that you have to insert 84 mode lines before you insert the modified mode lines. Remember too that you have already added the first command and there must be 83 commands between the last address of the `LMS` command and the point where you will make your insertion.

By adding 83 to `DL+5` (the address of the high byte of the `LMS` command), you realize that in order to reference the point in the list where you insert the first ANTIC mode you have to use the equation `DL+89` to `POKE` the first ANTIC mode 6 command into the list and `DL+90` to `POKE` the second ANTIC mode 6 command.

```
50  POKE DL+89
60  POKE DL+90
```

The rest of the mode lines on the screen remain the same, and you have neither overwritten the portion of the display list that gives the `jump on sync` command nor changed the location of the display list, which means that the rest of the display list remains the same and does not have to be changed.

What's Next?

Try running the program. You can find it on the companion Web site for this book. It is listing number RG0903.bas. Nothing much happens; in fact, all it seems to do is blink the screen and bounce back to graphics mode 0. Let's improve this program to make it do something useful.

Drawing the Display for the Top Graphics Mode 7 Displays

Add this line of code to the program:

```
5000 GOTO 5000
```

The purpose of this line is to pause the display so you can see what you draw on the screen before the screen switches back to graphics mode 0. As soon as your program is finished, the screen will immediately switch back to graphics mode 0. You have placed this code on line 5000 so that you do not accidentally overwrite it as you build this program.

Let's put some color on the screen. Add the following lines of code to the program:

```
70  COLOR 1
80  PLOT 0,0
90  DRAWTO 159,0
100  DRAWTO 159,83
110  DRAWTO 0,83
120  DRAWTO 0,0
```

You have seen these commands before. Line 70 changes the color that you were using to the value of color register 1, which is by default orange. Lines 80 to 120 draw a border around the top half of the screen.

Adding Text to ANTIC Mode 6 Lines

The next thing that you want to do is to place the player's score and the heading for the number of lives he has left in the text mode lines.

You have a slight problem when writing to a custom display list. To examine this problem, let's take a look at what normally happens when you try to print test lines on the screen.

The computer keeps track of which graphics mode it is using as well as how many lines are needed in order to display a line in that graphics mode. Here is why that is important: If the computer is displaying a graphics mode that uses 40 bytes and needs to display a letter on line 5, it is easy for it to calculate the correct memory location to store that data for the letter it has to show.

The computer knows that it has to multiply 5 times 40, which equals 200. This tells the computer that it has to draw that letter 200 bytes from the beginning of the video memory. Usually this works perfectly. That is, of course, because usually all of the mode lines are the same size. The problem is that when we start mixing mode lines in our custom display list that are all different sizes, there is no easy way for the computer to find the start of each line 1.

You see that each line occupies 40 bytes of memory. So the computer knows that every time it has drawn 40 bytes of memory, it has completed drawing a line and is ready to start writing the next line of the screen.

The same concept occurs when the computer is working with a graphics mode.

Each line on the graphics mode 7 display also uses 40 bytes of memory. Now in whatever graphics mode you are in, if you give the computer a command to plot a point (or draw a letter) at a particular line on the screen, the computer has to have a standardized method of calculating exactly where on the screen you want it to draw the point.

If you tell the computer, for example, that you want it to draw a point on row 5 of the screen in graphics mode 7, it will multiply 5 by the number of bytes in each line, which in this case is 40, to give you a value of 200. That way, the computer knows that in order to manipulate a point on line 5, it has to move forward 200 bytes in screen memory.

Now you are ready to understand why you have a problem when you are trying to write to a mixed mode display list. In order for the computer to correctly find the line video memory for the line number you want it to reference, it has to multiply the line number that you give it by the number of bytes used per line. The problem is that this only works if all of the bytes in the display list are the same!

Fortunately, like pretty much any problem that you are going to run into in your career as a programmer, there is a way around this problem. You can solve this problem by manipulating the data stored in three memory locations: namely, memory locations 87, 88, and 89. You see, the computer stores the current graphics mode that you are in at memory location 87. Also memory locations 88 and 89 store the low and high bytes of screen memory.

The computer wants to know what mode the computer is in and the number of bytes it has to multiply by to reference whatever line you need it to reference. The first thing that you are going to have to do is adjust the value held in memory location to match the kind of mode line that you will be working with. In the case of your program, you would change this to a value of `D`. This will tell the computer that you are working with graphics mode 1. Next, you are going to have to adjust the values in memory locations 88 and 86. What you are going to do is trick the computer into thinking that video memory actually starts at the mode 1 line.

Here is the code you are going to use to both "trick" the computer into thinking it is purely in mode 1 and to adjust the location of video memory.

```
130   POKE  87,1
140   SCRN=PEEK(88)+PEEK(89)*256
150   SCRN=SCRN+ 3360 +1
160   POKE 88,SCRN-(INT(SCRN/256)*256)
170   POKE 89,INT(SCRN/256)
```

Let's look over this code and examine exactly what it does. First of all, line 130 is the line of code that is going to actually trick the computer into thinking that you are purely in graphics mode 1. It does this by poking I into memory location 87 so that when the computer does its calculations, it will assume that it is purely in graphics mode 1.Next line 140 takes the location of the top of video memory and stores it in the variable SCRN.

Now you have to figure out in which portion of video memory the graphics mode 1 line is located so that you can see that as the top of the video memory. As it turns out, this is really quite easily done.

You know that 84 graphics mode lines, each one utilizing 40 bytes, are drawn before you get to graphics mode 1 line, which means that a total of 3360 bytes are used up in video memory before you get to this line.

If you add 3360 to scan, that will give you the address of the last byte of the last line just before the video memory for the mode line you want to work with. That means that 3360 +1 must reference the first byte of the graphics mode 1 line.

Lines 150 to 170 add 3360+1 to the address stored in `SCRN`, and then it `POKE`s this new address back into memory locations 88 and 86.

It is now possible for you to correctly print to the graphics mode 1 line using the `PRINT #6` command.

Writing DLI Interrupts

The next step for us to take, of course, is to actually start creating our display list interrupts. This will be the first area in this book where you will actually begin to use assembly language programming. For your convenience and to keep this chapter in focus, each computer that we are studying in this book has an area on the companion Web site with the assembler programs you have the option of using. You will find instructions on how to set them up and how to create your first program.

My personal favorite program is SYNASSEMBLER. This is a very cool program with a number of great features, the best of which for the purpose of learning to get up and running in assembly language programming, is the fact that this program is designed to work almost like Atari BASIC does. This means that a lot of the commands that you have already become accustomed to using such as `LIST`, `NEW`, `SAVE`, `LOAD`, and so forth will work in SYNASSEMBLER, usually in exactly the same way.

After you have read the assembly language primer and have chosen and learned how to setup and use the assembler of your choice, you will be ready to move on and create your first of many display list interrupts.

Writing a Display List

To refresh our memory, let's discuss what exactly happens when you cause a display list interrupt vector.

You call an interrupt display vector by setting bit 7 of the byte holding the mode line instruction representing the row on the screen where you want the interrupt to take place.

When the ANTIC chip reaches this mode line, it will draw out the mode line and then look at the NMIEN register to see if its enabled bit is set; if so, it pulls the NMI line of the 6502 to low. After this point, the ANTIC no longer has anything to do with the interrupt. All further processing is done via the 6502 processor.

Because the processor sees that its NMI (which stands by the way for Non Mask able Interrupt) line has been set to low, it executes an NMI interrupt by calling an interrupt service routine, which is stored in the operating system.

tip

Remember, an interrupt is a signal that is given to one of the computer processors to tell it to stop what it is doing so that it can execute a list of commands and then go back to what it was doing. An interrupt service routine is the list of instructions that the computer is asked to execute.

This routine simply verifies that it was indeed a DLI that was called; then it goes to address $0200, $0201 (low and high bytes, respectively) to find the location of the code you want to have executed.

Wow, once again you see the Atari really has its hands full before it even gets as far as executing your code.

Writing the Code for Your Display List Interrupt

Remember the example we gave earlier about you doing some homework or doing your taxes when a baby starts crying? We observed that you would stop what you were doing, go and assist the baby with whatever was ailing it (a dirty diaper, thirst, or hunger), and then go back to what you were doing before. Seems rather easy, right, and you do this or something like it almost every day? But what if after you are done helping the baby, you could not remember what you were originally doing? It would be impossible for you to resume your work because you have no idea what your work is any more. It's the same thing with the computer. When we interrupt the computer from completing a task, it is impossible for it to resume what it was doing unless it can remember what it was doing.

Guarding the Computer's Memory

There are three basic steps that you must take to ensure that the 6502 processor is able to remember, and hence return to performing, whatever task it was doing before you interrupted it.

First, you have to save the contents of the accumulator register, which you are going to be using as well as the contents of the processor's status flags. If you do not do this, when the

computer is done with your interrupt service routing and goes back to its original task, it will crash because the register, and flags for that matter, will be set incorrectly.

There are two commands that will assist you in this venture. First on the list is the `PHA` command. This command pushes the contents of the accumulator onto the stack for safekeeping. The second instruction that you need to be aware of is the PHP instruction, which pushes the value of the processor flags onto the stack.

You are now free to write the code for your interrupt service routine. Once you have completed writing the code for the DLI, you have to perform the second step of allowing the computer to "remember" what it was doing before you interrupted it. This set, as you might imagine, involves pulling the values that were original in the accumulator and in the processor's flags from the stack and back into their original locations. The instructions to do this are PLP to restore the values of the processor flags and PLA to restore the value of the accumulator.

Finally, the third step to allow the computer to remember what it was doing is to tell the computer exactly where the code is located that it was working on before it changed its focus to working on the new request. The command that allows you to do this is `RTI`. You see, before the computer got as far as executing out interrupt service routine, it saved the location of the code that it was working on to the stack. When you execute the `RTI` command, it retrieves this address from the stack and instructs the computer to go to this location and continue its original work.

tip

The 6502 only has built-in instructions to store and restore the accumulator and the processor flags, but these are not the only instructions that you may have to protect. You need a way to protect the X and Y register as well. Luckily for us, there is a way to do this using existing instructions. You see the PHP and PLP can only store and retrieve the accumulator, but there is no limit to how many times it can store or retrieve the accumulator! Look at the following code.

```
PHA
TXA
PHA
TYA
PHA
PLA
TAY
PLA
TAX
PLA
```

tip

You executed the PHA instruction three times, pushing the contents of the accumulator onto the stack three times. The first time, you stored the original contents of the accumulator. The second time, you filled the accumulator with the contents of the X register and thus stored the contents of the X register, and the third time, you filled the contents of the Y register to the accumulator and stored the values of the Y register. When it is time for you to restore the registers, you will simply do the opposite.

tip

Remember, pulling and pushing from the stack is sort of like adding and taking away from a stack of books. If I put down four books named a, b, c, and d, I cannot just move the book on the bottom without disrupting all of the other books. I have to start removing books from the top of the stack first. This means that the last book that I add to the stack has to be the first one that I take off, and the second to the last book has to be the second one that I take off, and so on. It is the same with the computer's stack. The first thing that you pull off of the stack has to be the last thing that you put on the stack. The last thing that you put on the stack was the Y register, so when you execute the first PLA instruction, it will actually pull the original value of the Y register and store it in the accumulator. This is why our next instruction is the TAY instruction, which will transfer the contents of the accumulator to the Y register, thus completing the task of restoring the Y register.

Following this same pattern, you are able to pull the next value from the stack, which is the original value of the X register and place it into the accumulator from where the TAX command is able to move it the X register where it belongs. Finally, you execute the final PLA instruction, which restores the accumulator to its original value.

Table 6.4 Instructions to Return the Computer to Its Pre-Interrupt State

Instruction	Action performed
PHA	Pushes the accumulator to the stack.
PHP	Pushes the processor registers.
PLA	Pulls the accumulator from the stack.
PLP	Pulls the processor flags from the stack.
RTI	Pulls the address of the code the computer needs to execute to resume what it was doing before you interrupted it from the stack and instructs the computer to go to that location and resume executing code.

Writing the Actual Heart of the Display List Interrupt

It is time for you to cut your teeth writing your very first display lists. For this first example, we will keep things abundantly simple. All that you are going to do is change the background color of the screen display. Choose which assembler you are going to use and follow the steps given to you on this book's Web site to obtain the file and to set up and use it. You will be given the pros and cons of each option and a suggested assembler, but at the end of the day, the choice is yours as to which one you will use.

The first thing that you need to do is to preserve the values of your registers.

```
PHA
TXA
PHA
TYA
PHA
```

This is the exact same code as used in the last tip. It uses the `PHA` command to store the values of each of the computer's registers and the processor flags. If you are not going to use a particular register, it is advisable for you not to push and pull this register from the stack unnecessarily, as it uses up clock cycles each time and, as you may recall, you do not have very many clock cycles to spare.

In BASIC, you would use a command called `SETCOLOR` to change the colors of the color registers. In Assembly, you do not have this luxury, however, and must do this manually. Another topic that will be covered in detail in the next chapter is the existence of four-color registers. You have used these before in a few of the demo programs. The background of the screen is always drawn using color register 0. The reason that the background of most graphics screens default to black and graphics mode 0 defaults to blue is that color register 0 in most graphics modes defaults to black while in graphics mode 0 it defaults to blue. What this means is that in order for you to change the value of the background color of the graphics 0 screen, you are going to have to change the value of color register 0.

As it turns out, this is a rather easy task to accomplish. You see, all the color registers are locations in memory hold a given color value. Change the color value at that memory location, and you change the register. Change the register, and you change the colors that are being used to draw the screen.

Here are the memory locations of the first four of the computer's nine color registers. The other five will be discussed in Chapter 7, "Hacking the Video Buffer."

Table 6.5 Memory Locations of the Computer's First Four Color Registers

Register	Hex Address	Decimal Address
Color register 0	D016	53270
Color register 1	D017	53271
Color register 2	D018	53272
Color register 3	D019	53273

Depending on which GTIA chip you have in your computer, you will have access to a total of either 128 or 256 colors. Whatever the color range on your computer, every number between 0 and the largest number in the color range of your computer represents a different color available to you.

So if, for example, you wanted to change the background of the text screen to pink, you would have to place a value of `$58` into color register 0. Given what you have just read, that means to turn the background of the screen to white, you would place a value of `$58` into memory location `$D016`. Here are the assembly language instructions to do just that.

```
LDA    #$58
STX    WSYNC
STA    $D016
```

This code loads the color value that you want to use into the color register and then stores it at the memory location that represents color register 0. Between the loading of the color value into the register and the storing of the new color value, you notice a rather strange looking instruction. What this does is take the contents of the x register and place it into the WSYNC register. The value of x in this example is irrelevant. You see, whenever you make use of this register, no matter what you are doing with it, this will cause the computer to freeze until the computer has finished drawing whatever is the current horizontal scan line. This way the color change takes place between lines giving you a very smooth display.

Finally, you insert the code to restore the computer's registers to their original value and return the computer back to what it was doing.

```
PLA
TAY
PLA
TAX
PLA
RTI
```

Converting Assembly Language Code to Decimal

Before you can actually use this code, it is necessary to convert it into a decimal format that can be poked into the computer's memory. Here is the program listing; let's convert it to decimal starting with the code that preserves the 6502 registers.

```
PHA
TXA
PHA
TYA
PHA
LDA     #$58
STX     WSYNC
STA     $D016
PLA
TAY
PLA
TAX
PLA
RTI
```

Table 6.6 First Part of Code Converted to Decimal

Opcode	Decimal version of the code
PHA	72
TXA	138
PHA	72
TYA	152
PHA	72

Next you create the portion of the code that actually does the work.

Table 6.7 The Heart of the Interrupt Program

Opcode	Decimal version of the code
LDA	169
#$58	88
STX	150
WSYNC	10,212
STA	149
$D016	22,208

Finally, you convert the code that will be used to restore the computer's register.

Table 6.8 Decimal version of the code used to restore registers

Opcode	Decimal version of the code
PLA	104
TAY	168
PLA	104
TAX	170
PLA	104
RTI	64

Inserting the Display List into Memory

The first thing you have to do is decide exactly what part of your screen you want to have a pink background. You then have to find the line of your display list where you must start drawing the pink background.

Assume that you are using ANTIC mode 2 and you want to start drawing pink on line 10. You know that you have to put your interrupt on the line before where the color is going to be drawn. This means that you need your interrupt to be triggered on the 10th visible mode line.

You know that the first 3 mode lines are blank mode lines, the 4th, 5th, and 6th lines are used by the load memory scan command. This means that the 7th mode line is responsible for drawing the top line on the screen. You still need to count down another 10 mode lines. This means that you have to insert your interrupt in mode line 16.

You have to modify the 16th byte of your display list. You know from past experience that in order to reference this byte of the display list, you will need the following equation DL+15.

This is the line that you are going to have to modify in order to insert your interrupt. You perform the actual act of inserting the interrupt by setting the DLI bit, which is bit 7, to 1. When the ANTIC encounters this mode line, it will know that it has to trigger a display list interrupt. This can be accomplished by adding 2 (our ANTIC mode 2 command) + 128 (the value that will simply set bit 7 of this byte) together and inserting the result, which is 130, into this byte.

Here is the code to implement everything that we have discussed so far.

```
100     DL=PEEK(560)+256*PEEK(561)
110     POKE DL+15,130
```

The next step is to write the code that will insert your code into the computer's memory. The best way for you to do this is with a simple `for` loop that will read each byte of the code and `poke` it into memory.

```
120     FOR I = 1 TO 19
130     READ A
140     POKE 1536+1,A
150     NEXT I
```

Once again, pretty simple. This code reads each byte of the interrupt service routine and pokes it into a free spot in the computer's memory.

Now all you have to do is store the program in a few data statements and do a bit of housekeeping, and your program will be good to go.

```
160     DATA 72,138,72,152,72,169
170     DATA 88,150,10,212,149
180     DATA 22,208,104,168,104
190     DATA 170,104,64
200     POKE 512,0
210     POKE 5134,6
220     POKE 54286,192
```

Lines 160 to 190 store the DLI service routing in your program listing. Lines 200 and 210 redirect the DLI interrupt vector to point to your program. Finally, line 220 activates the DLI feature on the computer.

Fine Scrolling

What's your favorite sport? Football, basketball, golf, soccer? Whatever it is, imagine the large field or court that this game is played on. Then imagine this sport being played inside of your tub. Sounds silly, right? Well, the truth is that as a human being your imagination is huge. Game programming simply gives you an avenue to express this imaginative creativity, and the truth is that it is just as silly for your favorite sports team to play the championship game in your bathtub as it is to think that all of your overflowing bubbling imagination and innovation and genius is going to always be able to fit on a single screen of game play.

When you graduate, as you will shortly, from the world of pong type incarnate games and create your first intergalactic space shooter, you will find that the universe you are going to create is going to be much too big to fit on one screen. This is where scrolling comes into play. Basically, what this technique will do is allow you to create a universe almost as large as you want and then display a portion of this immense world onto the screen at one time. Rather than moving the player around the screen, you will use the screen as a window that moves around the universe, allowing you to explore your world. See Figure 6.47.

Ever since I started writing about the Atari, I have been raving about the vision and advanced technology for its day that went into building this awesome machine. Here the Atari shines brightly once again. On most other computers in order to implement any type of scrolling, you have to move the entire contents of video memory in order to implement scrolling. See Figure 6.48.

On the Atari, very crude scrolling can be accomplished by simply manipulating two bytes, namely the address that the load memory instruction inside of the display list uses. Adding the size of a single line to the address used by the `LMS` command will cause the screen to move upward while subtracting the same amount will move it downward. See Figure 6.49.

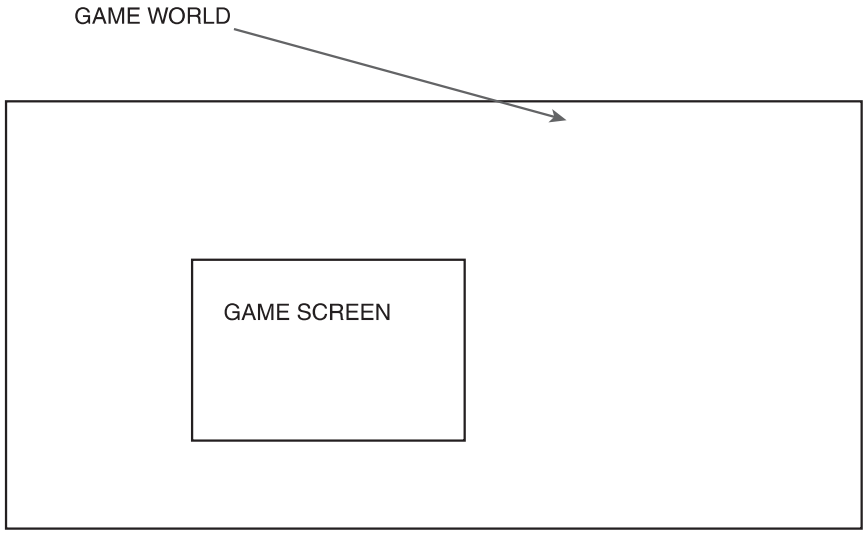


Figure 6.47 You will use the screen that you can move over your universe.

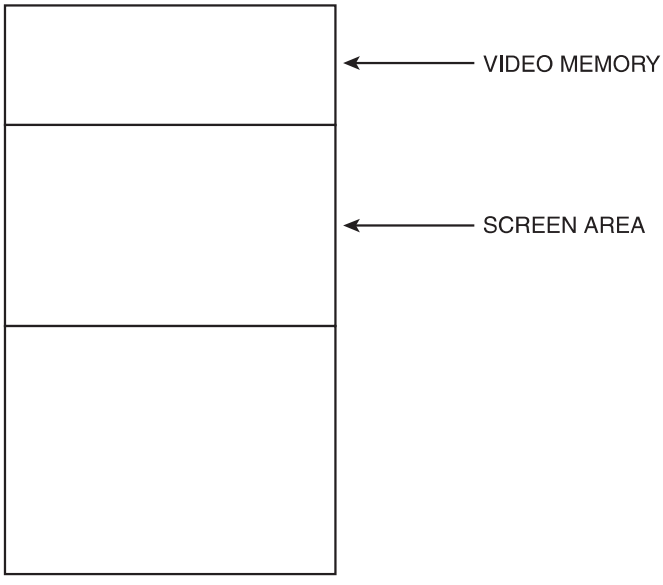


Figure 6.48 The traditional way of implementing scrolling.

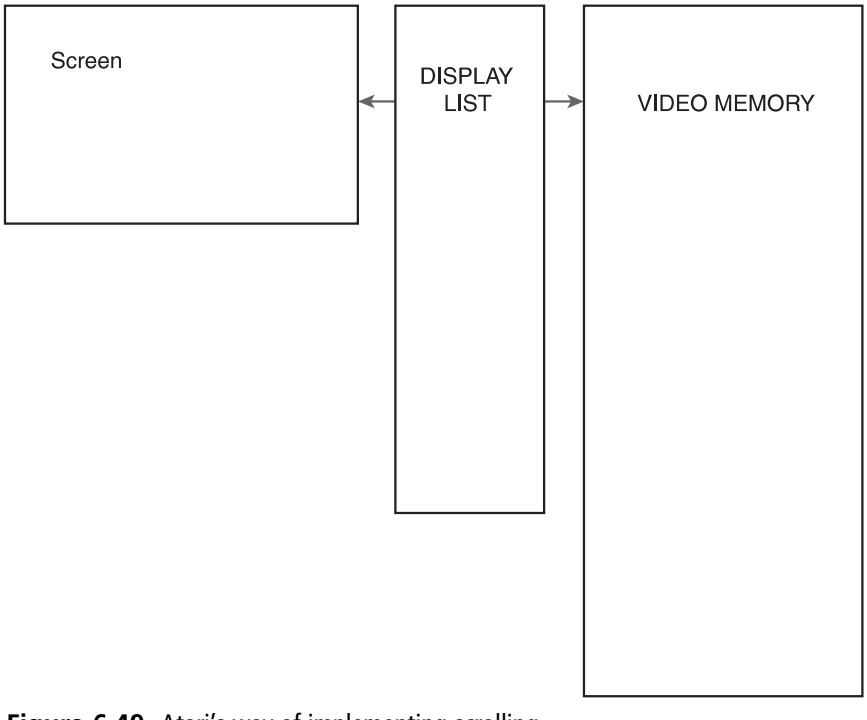


Figure 6.49 Atari's way of implementing scrolling.

As you can see, the Atari uses a much more advanced method of scrolling than other machines, and we have not even explored advanced features such as fine scrolling. So let's look at the way the computer implements scrolling in more detail.

Course Scrolling

Take a look at the following program:

```
10      GR.0
20      DL=PEEK(560)+256*PEEK(561)
30      LMSL=DL+4
40      LMSH=DL+5
50      SCRNL=0
60      SCRNH=0
70      SCRNL=SCRNL+40:REM Next line
80      IF SCRNL<256 THEN GOTO 120:REM Overflow?
90      SCRNL=SCRNL-256:REM Yes, adjust pointer
100     SCRNH=SCRNH+1
110     IF SCRNH=256 THEN END
120     POKE LMSL,SCRNL
130     POKE LMSH,SCRNH
140     FOR I = 1 TO 100
150     NEXT I
160     GOTO 70
```

The first four lines are nothing new. You ensure that the computer is in graphics mode 0 and the screen is clear in line 10.

Lines 20, 30, and 40 first give you a pointer to the display list and save the original address of video memory that was used by the load memory scan address.

Lines 50 and 60 declare the variables that you are going to use to store the high and low bytes of the value that you are going to add to the `LMS` address to perform the scroll.

You are in graphics mode 0. In this graphics mode, each line use up 40 bytes of video memory. For this reason, you add 40 to `SCRNL`, which you are going to be adding to the low byte of the `LMS` command. Each time the program loops back to line 70, you add 40 to `SCRNL` making it 40 bytes bigger; that way, each time the program loops around, the screen will scroll up by one screen line.

Lines 80 to 110 are used to be sure that you pan through memory correctly. As you recall, memory is organized into pages. Each page consists of 256 bytes.

If you are scrolling through the computer's memory from page one, you first increment through all 256 bytes of page 1, switch to page 2 and increment through all 256 of its bytes, switch to page 3, and so on until you have looped through all of the pages in the memory.

The high byte of the address used by the LMS represents the page that you are referencing. The low byte represents the byte within that page that you are trying to reference.

Lines 80 to 110 make sure that you pan through the pages of memory correctly. If SCRNL is less than 256, the program will loop to line 120, which will update the address used by the LMS, and thus implement the loop. If SCRN is more than 256, then lines 90 to 110 are executed.

First of all, if SCRNL is more than 256, you have crossed over into another page of memory. Line 90 subtracts 256 from SCRNL, which means that you are now referencing the beginning of the page. Line 100 increases SCRNH. This means that you are now referencing the beginning of the next page of memory.

Finally, if SCRNH is equal to 256 it means that you have crossed over into the last page of memory so you end the program.

Horizontal Course Scrolling

Horizontal scrolling is a bit more complicated then vertical scrolling. Take a look at Figure 6.50 to understand why horizontal scrolling is more complicated than vertical scrolling.

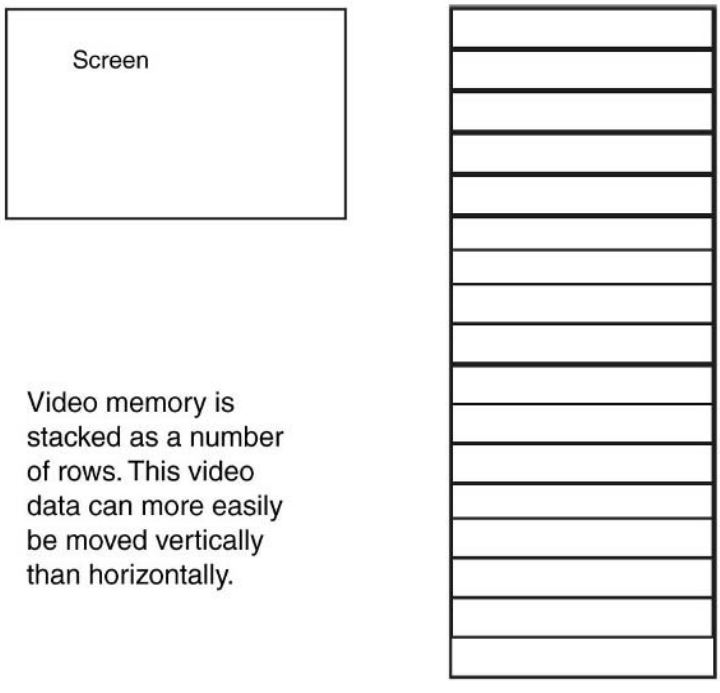


Figure 6.50 Understanding the way that video memory is organized is important to understanding why horizontal scrolling is more complicated than vertical scrolling.

Figure 6.51 illustrates the solution to this problem.

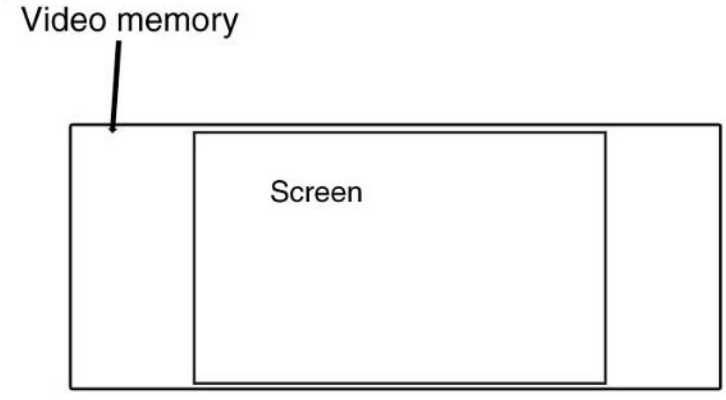


Figure 6.51 The solution to your program is to expand the video memory until it stretches beyond the width of your screen.

Now that you understand the general concept of course horizontal scrolling, it is time for a practical example. Let's examine the following code:

```
20 POKE 1536,112:REM 8 blank lines
30 POKE 1537,112:REM 8 blank lines
40 POKE 1538,112:REM 8 blank lines
50 FOR 1=1 TO 12:REM Loop to put in display list
60 POKE 1536+3*1,71:REM BASIC mode 2 with LMS set
70 POKE 1536+3*1+1,0:REM Low byte of LMS operand
80 POKE 1536+3*1+2,1:REM High byte of LMS operand
90 NEXT I
100 POKE 1575,65:REM ANTIC JVB instruction
110 POKE 1576,0:REM Display list starts at $0600
120 POKE 1577,6
130 REM tell ANTIC where display list is
140 POKE 560.0
150 POKE 561,6
160 REM now scroll horizontally
170 FOR 1=0 TO 235:REM Loop through LMS low bytes
175 REM we use 235 --- not 255 --- because screen width is 20 characters
180 FOR J=1 TO 12:REM for each mode line
190 POKE 1536+3*J+1,1:REM Put in new LMS low byte
200 NEXT J
210 NEXT 1
220 GOTO 170:REM Endless loop
```

Fine Scrolling

Fine scrolling is basically the same as course scrolling but with two major exceptions. First of all, fine scrolling is much, much easier than course scrolling and, second and probably the most obvious, is that this version scrolls smoothly.

As you recall from the above example, you scrolled the text on the screen either horizontally one character at a time or vertically one line at a time.

With fine scrolling, you are able to scroll horizontally or vertically by a portion of a character at a time as can be seen in Figure 6.52.

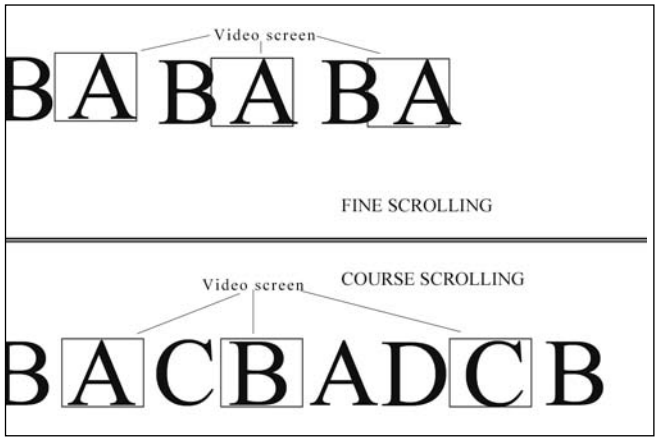


Figure 6.52 Fine scrolling compared to course scrolling.

Amazingly, as we have said before, despite the fact that fine scrolling is a more sophisticated way of performing scrolling, it is 100 times easier to implement. You see there are two registers related to fine scrolling, a horizontal register and a vertical register. All that you have to do to implement fine scrolling is to set the fine scrolling enabled bit of each of the mode lines that you want scrolled and then place a value inside of either the vertical or horizontal register, which represents the amount of clock ticks or scan lines you want the screen scrolled.

Let's take a look at a program that implements simple fine scrolling:

```
10 HSCROL=54276
20 VSCROL=54277
30 GRAPHICS 0:LIST
40 DL=PEEK(560)+256*PEEK(561)
50 FOR O = DL+6 TO DL+28
60 POKE O,50:NEXT O
70 FOR I=0 TO 7
```

```
80 POKE VSCROL,I
90 GOSUB 200
100 NEXT I
110 FOR J=0 TO 3
120 POKE HSCROL,X
130 GOSUB 200
140 NEXT j
150 END
200 FOR k=1 TO 200
210 NEXT K:RETURN
```

Lines 10 and 20 create two variables, which hold the decimal memory address of the horizontal and vertical fine scrolling registers.

Line 30 makes sure that you are in graphics mode 0 and then prints a copy of the program to the screen to make sure that you have something to scroll.

Line 40 creates the usual reference to the display list.

Lines 50 and 60 create a `for` loop that modifies the mode lines of the screen by enabling the fine scroll enabled bit of each of the mode lines on the screen.

Lines 70 to 100 create a `for` loop that increments the value of the vertical register thus scrolling slowly up the screen.

Lines 110 to 140 create a loop that increments the value of the horizontal scroll register thus scrolling the screen to the left.

Now it should be noted that both loops reference line 200, which creates a `for` loop that does nothing but waste time. Indeed, that is the whole point of this line of code. You use it to create a delay to slow down the scrolling of the screen so that you can actually see the scroll. At the end of this loop is a `RETURN` command; this causes the program to jump back to the `I for` loop or the `J for` loop, depending on which one called it.

The final result is that this program will first scroll the screen vertically and then horizontally. As you can see, implementing fine scrolling is exceptionally easy. There are two little pitfalls that you need to keep in mind though. First there is a limit to the distance that fine scrolling can scroll the screen.

In order to circumvent this problem, you have to implement a combination of fine scrolling and course scrolling.

The last little pitfall is that even though fine scrolling is implemented, the text at the bottom of the screen will appear to pop onto the screen as opposed to smoothly scrolling onto the screen. This can be solved by not setting the fine scrolling bit on the last mode line of the screen.

Setting the Video Mode on the Commodore 64

As you can see, setting the graphics mode on the Atari 400/800 is a very involved task. If you just take your time and go over this chapter piece by piece you will find that it is really quite easy. Fortunately, setting the video mode on the Commodore 64 is a lot easier.

Graphics in the Commodore 64 are generated by the Vic-II chip. This chip is programmed by manipulating the contents of its 47 control registers. 34 of these registers are used exclusively for working with sprite.

Graphic Mode VIC-II Register Address Bit to Set.

Table 6.9 Bit manipulation needed to set the C64 into its various video modes

Graphic Mode	Bit to Set	VIC-II Register Address
Multi-color mode	4	53270
Extended background color mode	6	53265
Standard bit-mapped graphics	5	53265
Multi-color bit map mode	5	53270

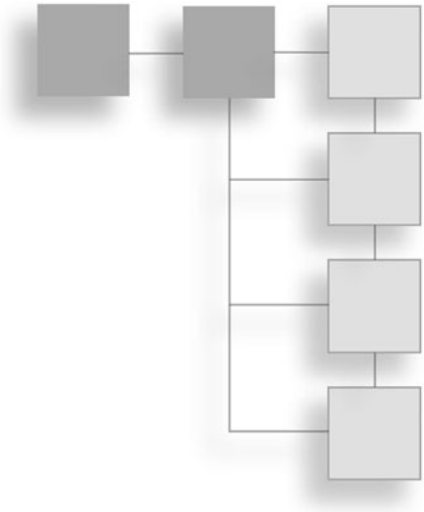
When all of these graphic modes are turned off, the computer is in Standard Character mode. This is also the default mode when your computer boots up.

Conclusion

You will hear me say it often. Retro game programming is easy. You put some graphics on the screen, get some player input, throw in some AI, rinse, and repeat. You now know 50 percent of what you need to know in order to accomplish the first goal of putting some graphics on the screen. You know how to set the video mode so that its contents become visible. Now in the following chapter you are going to learn the basics of how to place imagery on the screen.

This page intentionally left blank

CHAPTER 7



Hacking the Video Buffer

I'm tired of all this nonsense about beauty being only skin-deep. That's deep enough. What do you want, an adorable pancreas?

Jean Kerr

There are a few mantras that I believe in with all of my heart when it comes to game programming, and at the foundation of them all is this: "All that any computer can ever do is move and manipulate binary data and make decisions based on that data." As I have said before, once you control the flow of binary data, you control the machine.

At this point in your retro game programming career, you have almost all of the skills you need to do just that. You have learned how to communicate with your computer, and you have learned about the kind of graphic images you are going to want your computer to produce. Chapter 6, "Setting the Video Mode," has taught you how to place the computer into a receptive state so that it's ready for whatever graphics you want to put on the screen.

In this chapter, you will learn how to control the flow of data to the video screen. Within limitations, you draw images on your computer screen in exactly the same manner no matter which machine you are programming. The only thing that changes is the exact language that you use to do so and the finer details of the video mode you are in.

Identify the Characteristics of the Current Graphics Mode

The computer is turned on, the assembler is fired up, and all of the code is written to place the computer into the correct graphics mode you need in order to build your game. Now what? Well, the first thing that you need to do is figure out exactly where in the computer's memory, video memory starts and ends. This will let us know exactly what part of the computer's memory you will need to manipulate to draw images on the screen. Fortunately for us, most computers have default locations and sizes of video memory when they are set to specific video modes. Figure 7.1 shows some of the default locations and sizes for retro machines.

I often talk to people about the mind-expanding effects of retro game programming. This is often the result of working within the confines of the resources available. Working under these extreme conditions, there will often be a few "gotchas" waiting in the shadows to get you. One such "gotcha" involves the ability of many retro game systems to change among different colors as they draw horizontal lines of imagery across the screen. Let me explain. You already know that placing patterns of bits into the video buffer will cause dots to appear on the screen. You know that in certain video modes you may use a pattern of two, four, or eight bits to give each image on the screen a specific color.

Here is where the gotcha comes into play. You see, it takes time for your computer to change from drawing one color to the next. If you want to draw a blue line on top of a green line, there is no problem. After the blue line has been drawn, the computer can change colors during the horizontal blank. On the other hand, what if you want to draw a single horizontal line that starts off blue and then changes to green halfway across the screen? There is no horizontal blank between colors that will give us time to process the color change. As a result, strange things can happen in the middle of the screen where you change colors. In order to deal with this problem, certain conventions have been established on most retro game machines to help us to handle that situation. The two most common conventions are

- On some machines, placing two dots right next to each other will cause both dots to appear white no matter what color value you have assigned to the individual dots.
- On some machines, even and odd columns use two different groups of colors.

The first convention is easy. You see, on some computers the line you talked about earlier, which you wanted to be half green and half blue, would actually be drawn as all white! That's just a quick and dirty method of allowing us to guarantee what will appear on the screen as opposed to wondering what will happen on the screen between the blue and green dots. This may seem strange at first but once you get the hang of working with this phenomenon, it will not be as big a problem as it may first appear to be.

Computer	Screen	Page	Start(HEX)	Size
Apple II	Text/lo-Res	Primary	$400	1,023 Kilo Bytes
		Secondary	$800	1,023 Kilo Bytes
	Hi-Res	Primary	$2000	8,191 Kilo Bytes
		Secondary	$4000	8,191 Kilo Bytes
Commodore 64	General	N/A	$400	999 bytes
	Color	N/A	$D800	999 bytes
Atari 400/800	Graphic 0	N/A	$400	993 bytes
	Graphic 1	N/A	$400	513 bytes
	Graphic 2	N/A	$400	261 bytes
	Graphic 3	N/A	$400	273 bytes
	Graphic 4	N/A	$400	537 bytes
	Graphic 5	N/A	$400	1017 bytes
	Graphic 6	N/A	$400	2025 bytes
	Graphic 7	N/A	$400	3945 bytes
	Graphic 8	N/A	$400	7900 bytes
	Graphic 9	N/A	$400	7900 bytes
	Graphic 10	N/A	$400	7900 bytes
	Graphic 11	N/A	$400	7900 bytes
Color Computer 2	Alphanumeric	N/A	$400	512 bytes
	Alphanum Inv	N/A	$400	512 bytes
	Semigraphic4	N/A	$400	512 bytes
	Semigraphic6	N/A	$400	512 bytes
	Semigraphic8	N/A	$400	2048 bytes
	Semigraphic12	N/A	$400	3072 bytes
	Semigraphic24	N/A	$400	6144 bytes
	64 X 64	N/A	$400	1024 bytes
	128 X 64	N/A	$400	1024 bytes
	128 X 64	N/A	$400	2048 bytes
	128 X 96	N/A	$400	1536 bytes
	128 X 96	N/A	$400	3072 bytes
	128 X 192	N/A	$400	3072 bytes
	128 X 192	N/A	$400	6144 bytes
	256 X 192	N/A	$400	6144 bytes

Figure 7.1 Default location and sizes of the video buffer on retro machines in various graphics modes.

The second convention can also complicate our programming lives until you really get used to it. You see, another way of guaranteeing what will happen on the screen is to optimize the computer's ability to change from one group of colors to the next. As an example on the Apple II series of computers (with the exception of those using Revision 0 Apple boards), while in high resolution graphics mode, any dot drawn in an even column can be black, violet, or blue while dots placed in odd columns are drawn as either black, green, or red (on Revision 0 Apple II boards the colors blue and red are unavailable). Once again this convention takes some getting used to, but once you get the hang of it, it will not seem so bad.

Video Buffer Hacking 101

There are a few basic things that you need to be able to do with video memory now that you know where it is and how the information placed inside of it is going to be interpreted.

- We need to know how to clear the screen either to black or to some background color.
- We need to know how to take bytes of data stored elsewhere and place them into video memory.
- We need to be able to move data from video memory and store it somewhere else.

These three skills are the fundamental building blocks of computer graphics and animation. (Remember my mantra "all that any computer can do is move and manipulate bits of binary data.) Clearing the screen makes sure that you do not have a bunch of garbage on the screen. For some games you can use the exact same code in order to place a background color on the screen. Once you have the screen clear, all computer graphics and animation is done using the second two skills: moving data out of the video buffer to another location, moving data from another location into the video buffer.

You already know how to clear the computer's video screen. In order to clear the computer's video screen you need to set every bit of the video buffer with zeros. You already know how to set a byte to zero. You load the accumulator with zeros and then you store those zeros into a given byte that you want to clear.

```
LDA    #0000    LOAD THE ACCUMULATOR WITH ZEROS
STA    $400     STORE THE ZEROS IN THE BYTE OF VIDEO MEMORY WE WISH TO CLEAR.
```

If you only needed to clear one or two bytes to clear the screen, that would be all the information you needed. Unfortunately, there are so many bytes to clear that it would be far too tedious for us to manually clear each byte of the video buffer. Fortunately for us, Assembly makes it very easy for us to clear large amounts of memory. You simply need to loop through the video buffer clearing all of its bits.

The first thing that you need to do in order to clear an area of memory is to store both the starting location of the memory you need to clear as well as the number of bytes you intend to clear. Here is an example of this using 6809 assembly language. As you learned in Chapter 4, "Assembly Language Programming," the X and Y registers are real 16-bit registers.

```
Ldx    #1024     point to top of screen
Ldy    #512      set # of bytes to clear
```

The `LDX` command stores a decimal value of 1024 into the 16-bit X register. In this example, this is the location of the start of video memory. The `LDY` command loads a decimal value of 512 into the 16-bit Y register. In this example, this is the number of bytes you wish to clear (set to zero) in your video buffer. This acts as sort of an ad hoc clock counter.

Next you will use the LDD command to place zeros into the D register. Remember the D register is a register made by combining the 8-bit A and B registers into one 16-bit register. If you reference the A or B register after storing data in the D register, the A register will have the high part of the data, and the B register will have the low part of the data.

```
Ldd    #0000     use 2 bytes of  0
```

Now here is where the real work gets done. You have stored two bytes of zeros in the D register. Now you are going to store those zeros in the video buffer. To clear your screen you need to fill the video buffer with zeros, two bytes at a time. You need to clear two bytes, then move to the following two bytes and clear those, and so on. To do this, you are going to use an interesting form of indirect addressing. Imagine that you had a loop that kept clearing the byte stored at address 1024. You could run that loop for as long as you want and all that would happen is that the same two bytes would be cleared over and over. Now imagine that you stored the address 1024 in the X register and executed the same loop, clearing the address stored in X. You would still only clear the same two bytes.

Now imagine that same loop, only this time, every time you clear those same two bytes, you increase the address stored in X by 2. The next time around, a new area of the video buffer would be cleared. If you kept running this loop, you would eventually clear the video buffer! You need to clear 1024 bytes, and you are going to clear these bytes two at a time, which means that you need to repeat this loop 512 times (1024 ÷ 2).

That seems like a lot of work to do but 6809 assembly is going to allow us to do all of that using only one instruction.

```
Loop      std       ,x++   clear the 2 characters
```

Be sure to note that this line of code begins with a label named "loop." This is so you can jump back to this line later on.

Now that you have cleared two bytes of memory and advanced to the next two bytes of memory, you need to subtract two for your counter (the Y register). You will do this using the LEAY instruction and another form of indirect addressing.

```
Leay    -2,y    subtract them from count
```

This instruction will load Y with the original value of Y minus 2. What you want is for this program to keep looping and clearing memory until the Y register is equal to 0, which would mean that the proper amount of memory has been cleared. Fortunately, there is a system flag called the *zero flag* and a group of assembly commands referred to as branch commands that assist us with this goal. Whenever the results of an operation, such as placing a value in a register (in this case the Y register), results in a zero, the zero flag is set to one. In the line above you subtracted two from the Y register and placed the value into the Y register. If the result of this operation were zero, the result would cause the zero flag to be set to 1. If this happens, you know that you have cleared the required amount of memory.

To help you with your task, you will use the BNE instruction. BNE is short for Branch if Not Equal to zero.

```
    Bne    loop    count not 0, so repeat
```

When the code above is executed, it will check to see if the zero flag is set to 1. If it is not, then you have not cleared enough memory and it will cause the program to jump back up to the line that starts with the label "loop." The program will once again clear two bytes of memory, move to the next two bytes, and subtract two from the counter.

If the zero flag is set to one, this instruction will do nothing and the rest of the program will be executed. Here is the complete code to clear portions of video memory. (Actually this can be used to clear any part of memory and is actually a similar process to the way you are about to learn to draw sprites.)

```
          ****************************clear the screen
cls     ldx     #1024     point to top of screen
        ldy     #512      set # of bytes to clear
        ldd     #0000     use 2 bytes of 0's
loop    std     ,x++      clear the 2 characters
        leay    -2,y      subtract them from count
        bne     loop      count not 0, so repeat
          ****************************end clear screen
```

Remember that the same principle applies when you want to fill the screen with a color or a given character or pattern. All that you have to do is place the binary pattern for your character or pattern or color into the video buffer instead of zeros. The same principle also applies when you are using sweet 16 on the Apple II or 6502 assembly on any com-

puter system that uses it. Here is an example of the same clear screen procedure written in both 6502 and sweet 16 assembly.

Sweet 16 assembly example of clearing the screen.

```
        SET     R5    $A034     ;Start of video memory
        SET     R4    9         ;
        SUB     R0              ;Zero ACC
LOOP2   ST      @R5             ;Clear a mem byte
        DCR     R4              ;Decrement count
        BNZ     LOOP2           ;Loop until Zero
```

6502 assembly example of clearing the screen.

```
START   LDA     LDA #$0
        LDY     #$0
        LDX     #$400
LOOP    STA     $400,Y
        INY
DEX
BNE     LOOP
```

As you can see, clearing the video buffer is usually quite easy. It is easy when video memory is arranged to flow line by line as seen in Figure 7.2.

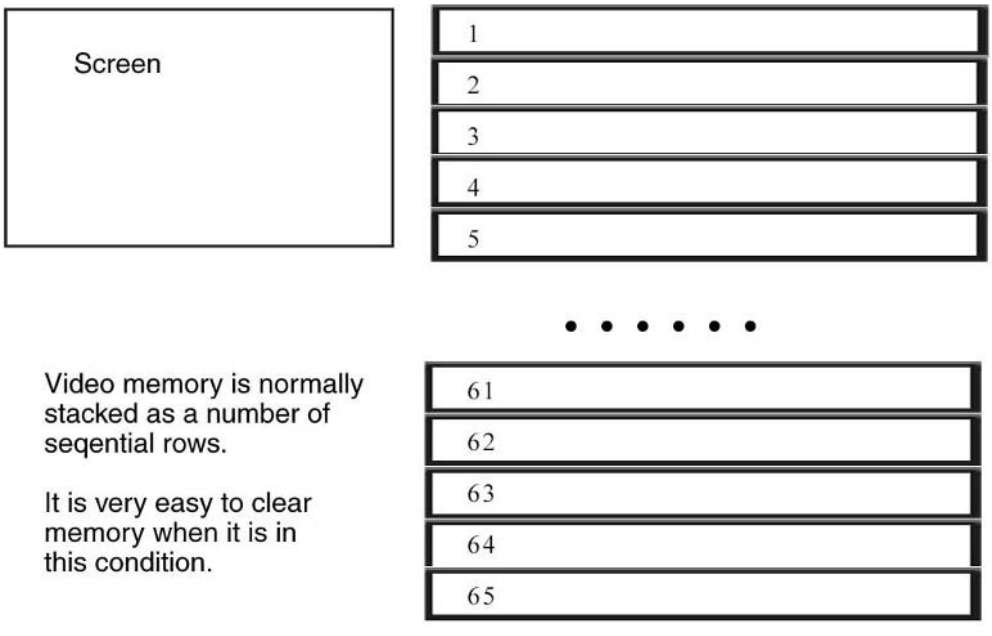


Figure 7.2 When video memory is arranged line by line it is easy to clear.

Some computers complicate things a bit by arranging video memory differently as shown in Figure 7.3.

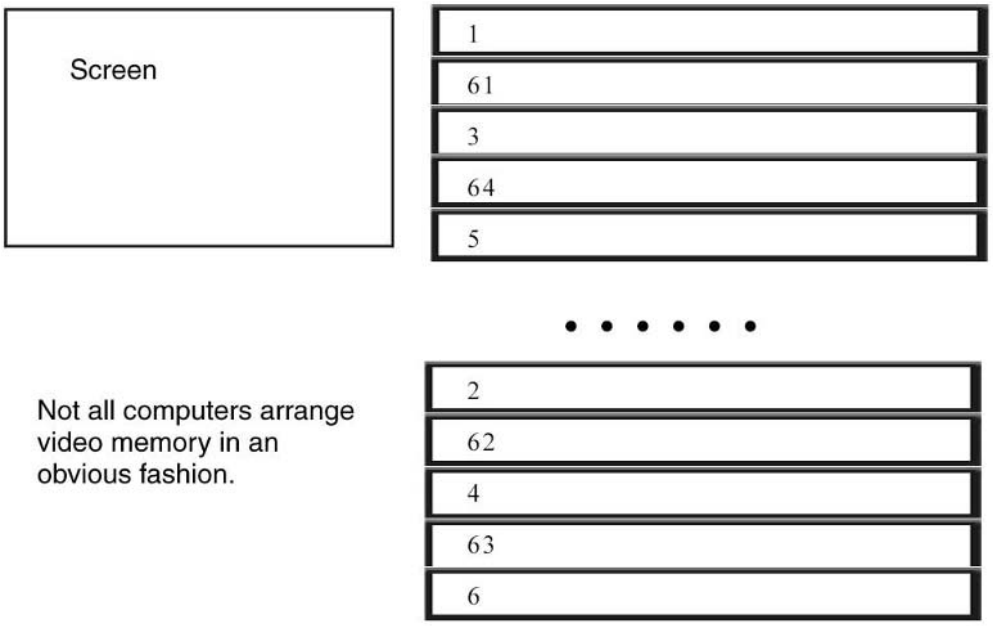


Figure 7.3 Not all computers arrange video memory in an obvious fashion.

In these situations, clearing the video memory is still easy and still works using the same principle. You simply have to make a few adjustments to your code. Using the knowledge you have gained so far, let us look at the following code, which is designed to set an Apple II computer to low resolution mode and clear the screen. If you type this code into your assembler and run it, you will see that the screen is cleared.

```
START   LDA   #$0
        STA   $C054
        STA   $C052
        STA   $C056
        STA   $C050
        LDY   #$0
        LDX   #$79
LOOP    STA   $400,Y
        STA   $480,Y
        STA   $500,Y
        STA   $580,Y
        STA   $600,Y
```

```
        STA     $680,Y
        STA     $700,Y
        STA     $780,Y
        INY
        DEX
        BNE     LOOP
PSTOP   JMP     PSTOP
```

Take a look at the code above and Figure 7.3 and try to see if you understand what is going on.

As you can see, video memory in the Apple II is arranged in a very interesting pattern. Figure 7.4 makes the pattern much clearer.

Each byte represents two vertically stacked blocks. Each row is 40 blocks wide (0 to 39), so if you start at $400 for example and clear 40 bytes of memory, you will clear a line on the top of the screen. If you clear another 40 bytes you will clear another line about midway down the screen. Clearing another 40 bytes will clear another line closer to the bottom of the screen. Clearing another 40 bytes will start clearing the screen again starting with the row following the one that you first cleared.

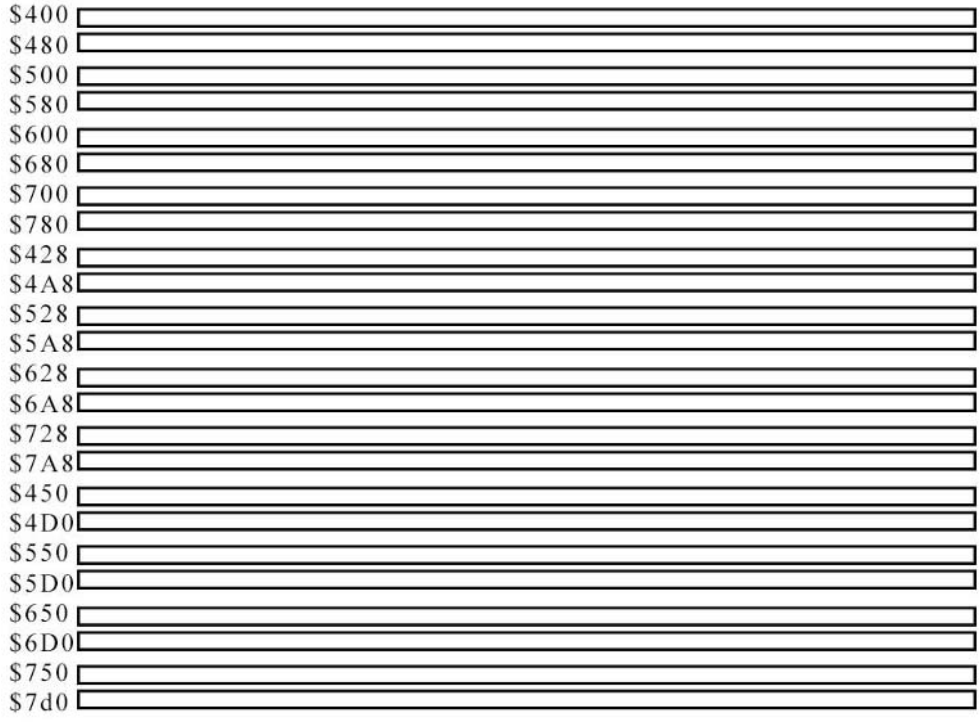


Figure 7.4 Video memory in the Apple II is arranged in a very interesting pattern.

Our screen is 24 bytes high. This would normally mean that it would take 24 loops in order to clear the screen. You, however, are going to reduce this number to 8. You do this by writing exactly the same code that you wrote before with two exceptions. First of all, rather than setting your counter to clear one line, you will set it to clear three lines. Next, rather than just using one STA instruction to start clearing from the beginning of video memory, you will use the memory locations for the start of the top 8 rows of the screen. Performing this loop will first clear the top 8 lines of code followed by the middle lines of the screen and finally the bottom rows of the screen.

Placing Data in the Video Buffer

Programming techniques are almost always built on other techniques. This makes it a bit easier to learn because once you understand one technique you can use that knowledge to understand more complex ideas. The skill you just learned for clearing your computer's display actually forms the basis for almost all of the graphic work that you are going to do to make your game.

When you cleared the screen, you placed zeros into one memory location, moved to the next location, put zeros there, and on and on until you had cleared the display. See Figure 7.5.

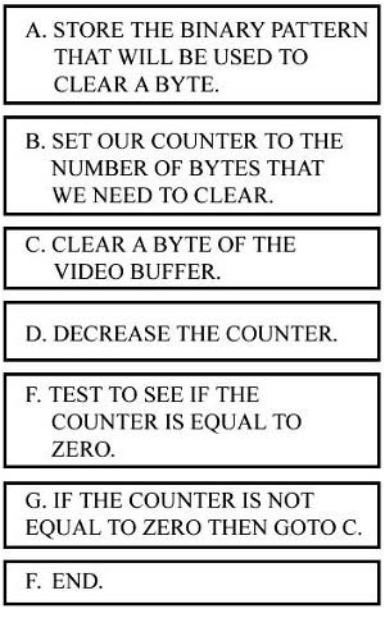


Figure 7.5 A flow diagram of how you cleared the screen.

If you had a picture of the background for your game stored somewhere in memory or on disk, you could use the exact code that cleared the screen above to paint that picture on the screen. All that you have to do is add some additional code so that instead of always storing zero into the memory locations of the video buffer, you can take bytes from your background, which is stored in memory, and place them into the appropriate byte on the screen.

Let's take a look at some code to do just that.

```
START    LDY    #$0
         LDX    #$400
LOOP     LDA    $300,Y
STA      $400,Y
         INY
DEX
BNE      LOOP
```

As you can see, this is exactly the same code that you used earlier in order to clear the screen with the addition of one line: `LDA $300,y`. `$300` can be any location in memory where you have images stored for the background of your video display. Every time this loop is run, not only will the program loop through video memory storing data in every byte of the video display, but it will also loop through the memory locations holding your background image, retrieving the bytes of data that are stored in the video buffer.

This technique works, but it is not very efficient. It takes a whole lot of memory to store a background at some arbitrary location in memory. On a retro game machine, extra video memory is usually in short supply and is therefore a commodity you cannot afford to use haphazardly. As you saw in Chapter 5, "A game Graphics Primer," it is much more efficient to use some form of tiling system. A tiling system takes up much less memory and allows you to build backgrounds *on the fly*. These tiling systems also work on the same principles as before only with a few more modifications.

Let's say that you are in a high-resolution graphics mode where one bit equals one point on the screen. You want to draw tiles that are 32 bits wide and 32 bits high. That's 128 bytes of data for each tile. Let's say that you have that data stored starting at memory location `$200` and video memory starts at location `$400`. You want to draw a tile in the top left hand corner of the screen.

What if you made this quick change to your code?

```
START    LDY    #$0
         LDX    #$80
LOOP     LDA    $200,Y
STA      $400,Y
         INY
DEX
BNE      LOOP
```

This code would draw your sprite on the screen but not the way you want it to. Rather than drawing the sprite as a number of rows stacked on top of each other, this would draw them as a number of rows laid side by side.

What you need is a way to tell the computer to jump to a new row on the screen every time it has completed drawing a row of your sprite. Let's modify your code a bit.

```
START   LDY     #$0
        LDX     #$0
LOOP    LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        CLC
        TYA
        ADC     $27
        TAY
        TXA
CMP     $80
BNE     LOOP
LOP2    JMP     LOP2
```

This code is going to draw your sprite on the screen the way you want it. Let's take a look at what this code does.

```
START   LDY     #$0
        LDX     #$0
```

We are going to use the X and Y registers as counters. The first two lines of code sets these counters to zero.

```
LOOP    LDA     $200,x
STA     $400,Y
```

The next two lines are the heart of this program. The first line reads a byte from your sprite using indexed addressing which allows us to treat the sprite memory location as a kind of array. The second line also uses indexed addressing to store the byte of the sprite you just retrieved into the video buffer.

```
        INY
        INX
```

Instructions to increment both your X and Y registers (the counter) causes the next byte of your sprite to be placed into the next screen location when the `LDA` and `STA` commands used above are executed again. Because each row of your sprite is four bytes wide (32 bits), you will execute this combination of code four times.

```
LOOP    LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
```

Now you have the same old problem again. If you just keep on reading from the sprite and placing data into the buffer, you are going to end up with the sprite being draw as 32 rows drawn side by side as opposed to stacked on one another as you would like.

This is where the next few lines of code come into play.

```
CLC
        TYA
        ADC     $27
        TAY
```

If this code looks familiar the reason is that this code is exactly the same as the addition code you studied in Chapter 4. Let's look at what it does in the context of your program.

Each row is 39 bytes wide. The first four bytes are used by a row of your sprite. If you move ahead 36 (24 hex) bytes and start writing the next row of your sprite, that row will be drawn exactly where you want it! It is for this reason that you add the hexadecimal value for 36 to the Y register. (Remember the Y register controls the location where each byte will be written in the video buffer.) Taking this action will cause the next row to be drawn directly beneath the previous row.

All you need to know now is how to find out if you have any more rows to draw; if you have more rows to draw, you simply jump back up to the row labeled loop.

```
TXA
CMP     $80
BNE     LOOP
```

The first line of code transfers the contents of the X register to the accumulator. The second line of code compares the contents of the accumulator (which you just copied from the X register) to `$80`. If they are equal, you know that you have drawn the entire sprite.

We can use the same code to clear a tile from the screen as well. All you have to do is make some minor modifications to the code.

```
STAR   LDY      #$0
       LDX      #$0
LOO    LDA      #$0
STA    $400,Y
       INY
       INX
       STA      $400,Y
       INY
       INX
       STA      $400,Y
       INY
       INX
       STA      $400,Y
       INY
       INX
```

```
        CLC
        TYA
        ADC     $27
        TAY
        TXA
CMP     $80
BNE     LOOP
LOP2    JMP     LOP2
```

As you can see, the code above does exactly the same thing, only this time rather than reading sprite data to place in the video buffer, you simply fill the accumulator with zeros at the beginning of the code. The code will now fill a 32 by 32 square area of video memory with zeros.

If you need to save a tile from the screen and save to an another part of the computer's memory, you can use the same code with another small change added.

```
START   LDY     #$0
        LDX     #$0
LOOP    LDA     $400,Y
        STA     $200,x
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        LDA     $200,x
STA     $400,Y
        INY
        INX
        CLC
        TYA
        ADC     $27
        TAY
        TXA
CMP     $80
BNE     LOOP
LOP2    JMP     LOP2
```

This time your code is going to read from the screen and save to a given memory location.

There is still one problem in all of the code written above. Look at the code and see if you can tell what it is. All of the code references a specific byte of video memory, which means that you cannot move the tile around the screen. This is an easy problem to fix. All you have to do is create a variable in your assembler and use that variable to reference the portion of video memory you want to use. Here is an example.

```
SPRITE .DA      $400
START   LDY      #$0
        LDX      #$0
LOOP    LDA      $200,x
STA     SPRITE,Y
        INY
        INX
        LDA      $200,x
STA     SPRITE,Y
        INY
        INX
        LDA      $200,x
STA     SPRITE,Y
        INY
        INX
        LDA      $200,x
STA     SPRITE,Y
        INY
        INX
        CLC
        TYA
        ADC      $27
        TAY
        TXA
CMP     $80
BNE     LOOP
LOP2    JMP      LOP2
```

Page Flipping

The skills you have just learned are enough for you to perform basic animation. You can draw a sprite on the screen, then erase it and draw it somewhere else to give the illusion of animation. If there is a background on the screen, you can save the background before

drawing your sprite. This way you can simply redraw the background over the sprite to clear the screen while preserving the background.

If all you needed to do was move one image at a time on the screen, then I would be able to end this chapter right here. Few, if any, arcade games will ever have you animating just one object on the screen, however, which means you are going to have to learn one more important skill. When you have to draw multiple characters on the screen, it takes a while for images to be erased and then redrawn. This will lead to flickering and will completely destroy the illusion of a game, not to mention really irritate the player. This is where page flipping comes into play.

In *page flipping*, as the name implies, you actually use two video buffers. Here is how it works. You set up two video buffers but only display one. While one buffer is being displayed, you draw your images in the hidden video buffer. Then, all at once, you switch the buffer that is being displayed. Now you draw to the other video buffer. When you are done, you can switch video buffers again. This way, one whole image is drawn on the screen and there is no flickering.

There is another technique that is very similar called *double buffering*. In this technique you have one buffer that is always shown and one buffer that is always hidden. You do all of your drawing to the hidden buffer and then copy the contents of the hidden buffer to the visible buffer where they can be seen.

Fortunately, most retro video game systems make this little trick very easy to use.

On the Apple II, you set your video mode so that you can simply adjust the combination of soft switches you use to choose between the primary and secondary video buffer.

On the Atari you can alter your display list on the fly causing the computer to point to any video buffer you would like to have displayed at a given time.

On the Color Computer, you have to manipulate the SAM chip.

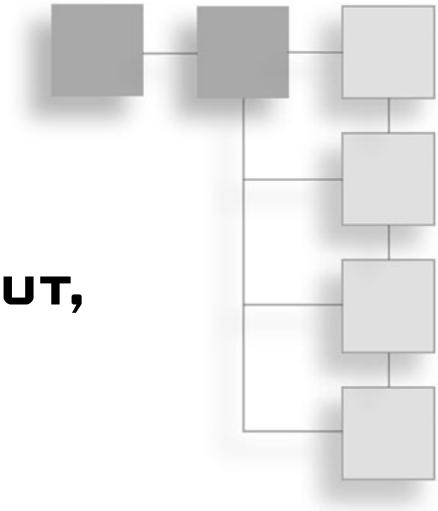
The Commodore 64 can be adjusted to point to one of the four quarters of the computer's 64 kilobytes of memory. In order to adjust the video buffer of this machine, you need to place the location of the video buffer you want to display into the computer's register stored at memory location `$D018`. Then you tell the computer to point to the memory quadrant that contains your video buffer.

There are numerous examples of both page flipping and double buffering on the companion Web site to this book. I suggest that you download the examples for your retro machines to see how they work. Each example has a detailed tutorial to explain each line of code used in the program.

Conclusion

In the past three chapters, you have learned the basics of how computer graphics are generated followed with a more detailed account of how to set the video mode and finally how to place images on the screen. Good graphics have always been an important part of making an enjoyable game. With practice, you will be surprised at what you can do. Now is a good time to go to this book's Web site and download some of the graphics demos. Experiment with them and try making some of your own. When you are sure that you have mastered creating game graphics, you will be ready to move on to the following chapters, which will teach you the final lessons you need to learn to create good retro games.

CHAPTER 8



Adding Player Input, Physics, and AI

Men fear thought as they fear nothing else on earth—more than ruin—more even than death. . . . Thought is subversive and revolutionary, destructive and terrible; thought is merciless to privilege, established institutions, and comfortable habit. Thought looks into the pit of hell and is not afraid. Thought is great and swift and free, the light of the world, and the chief glory of man.

Bertrand Russell (1872–1970)

Earlier in this book, I told you that making a video game is very easy once you know how. You draw something on the screen, do some stuff in the background, "rinse and repeat." It is here that we learn what "doing some stuff in the background" means. Basically, in a retro game, there are four things that will be done in the background:

1. Getting input from the player
2. Running the computer's artificial intelligence (or AI) algorithms
3. Modeling game physics
4. Managing game states.

These procedures can be as simple or as complicated as the computer system you are programming on and the nature of your game demands. In most cases, we will try to keep things as simple as we can for the sake of writing faster, more optimized, code.

Creating Your Computer's Intelligence

No game would be fun if all of the player's enemies just sat in one spot until they got shot. Your player wants a challenge and he should get one. For this reason, we must be sure that the enemies in our game appear to think about what they want to do. It really doesn't take that much to make an enemy character appear to be intelligent. For an example, if an enemy sees the player and the enemy is stronger than the player, it should attack the player. If, on the other hand, an enemy sees the player and the player is stronger, then the enemy should do the smart thing and run away. It's not rocket science, but it does make the player's experience a lot more challenging and adds much more depth to the game.

As a child, I wrote these algorithms all the time even though I never saw them in any book. It was not until later when I read my first actual game book that I saw these were "real" algorithms. My first thought was, "Wow! I was actually on the right track. Cool!" The reason I was able to come up with these algorithms on my own is not because I am some super genius but simply because these algorithms follow the basic laws of common sense! This fact makes them very easy to learn.

Tracking Algorithms

Tracking plays a very important part in any game. Whenever an enemy fires a missile or chases after our player some form of tracking algorithm is being used. A tracking algorithm has to be aware of two game objects; a source object and a target. In the case of a missile that is fired at our player, the missile is the source object and the player is the target. With every game cycle the source object will be moved closer to the target object until eventually the source object reaches its target. In this case the missile will continue to get closer to the player until it hits the player. Figure 8.1 illustrates this concept.

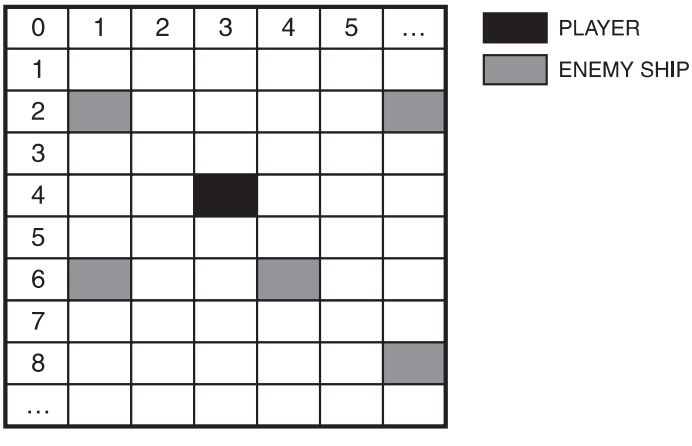


Figure 8.1 Screen coordinate system with the player and several enemies on the screen.

Looks like our player is in a lot of trouble. The player is completely surrounded by enemies. As players, we should be very afraid. As game programmers, we know that the player is really in no danger. Why? Because we have not yet written the code that will allow those ships to track down our player. Right now, all that those ships can do is sit there and look pretty.

We can easily think in a common sense sort of way and figure out exactly what we need to do. First let's look at what we know.

- We know that each enemy has its own X and Y positions that determine where it appears on the screen.
- We know that the top-left corner of the screen is the origin of the screen, which means that a sprite whose position is 0,0 will appear there.
- We know that the higher the value of a sprite's y value, the lower it will be on the screen and vice versa.
- We also know that the larger its x components are, the farther to the right of the screen it will appear and vice versa.

Armed with this information, let's take a look at enemy number one in Figure 8.1. As you can see, it is to the left of our enemy. We can infer that this enemy's X component is less than that of our player's. We can also infer that if we increase this enemy's X component, then that enemy will move further to the right and thus closer to our player.

If we look at enemy number two, we can see that it is located to the right of our player, which means that its X component must be greater that that of our player. If we want this enemy to move closer to our player, we are going to have to decrease its X component.

Enemy three is above our player, so we need to increase its Y components in order to move it closer to our player. Inversely, enemy number four is beneath our player, so we will have to decrease its Y component in order to move it closer to the player.

From the example above, we can easily assume how to implement a tracking system to allow any enemy or enemy missile to track our player. Our algorithm will look something like this.

1. Compare the X component of the player and the enemy object.
2. If the X component of the enemy object is less than that of the player, increase the X component of the Enemy.
3. If the enemy's X component is greater than that of the player, decrease the enemy's X component.
4. Compare the Y component of the player and the enemy object.
5. If the Y component of the enemy object is less than that of the player, increase the Y component of the enemy.

6. If the enemy's Y component is greater than that of the player, decrease the enemy's Y component.

As you can see, this is not a very difficult algorithm. All of the other algorithms are just as easy once you come to understand them.

Now that you know the theory of how this program works, let's look at some actual code.

Evasion Algorithms

This is another important ability that your player's enemies must possess, which is the sense to know how to run away. The good news is that if you read and understood the previous section, then you have all of the information that you need in order to implement evasion (think about it). In the last section, we compared the X and Y components of the enemy and the player and either increased or decreased the value of the enemy's coordinate components in order to move them closer to the player. All that you have to do now is the opposite. You will compare the coordinate components of both the player and his enemy; then where you increased the value of a component in the previous example, you will decrease it here, and vice versa.

Better Tracking and Evasion Algorithms

The tracking and evasion algorithms we developed earlier work well for many situations, but they are not suitable for all situations because they are not very realistic. Planes and cars and motorbikes do not make perfect 90 degree turns at full speed. Vehicles moving at high rates of speed tend to turn in arcs. The reasons for this will be discussed in detail when we discuss game physics at the end of the chapter. For now, we will jump the gun a little and examine how to use this fact in our tracking algorithm. Examine Figure 8.2.

What this concept means for us as game programmers is that we are going to have to rethink our tracking and evasion algorithm. Look at Figure 8.3.

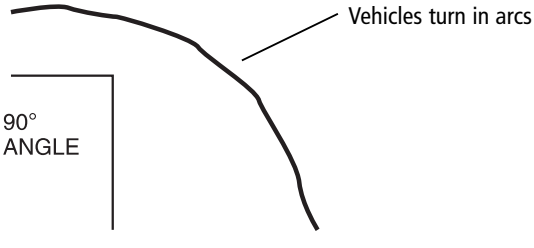


Figure 8.2 Fast moving vehicles do not turn at acute 90-degree angles but rather turn on arcs.

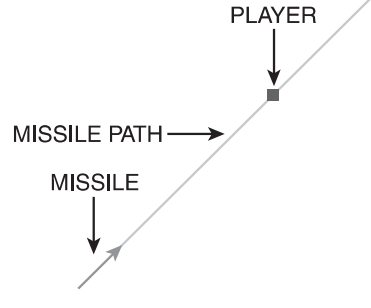


Figure 8.3 A missile has been fired at our player who is standing still in the center of the screen.

Because our player is standing still, he can easily move in any direction. The missile that has just been fired does not enjoy this same freedom because it is already traveling in a specific direction toward our player. Suppose our player moves down a few pixels out of the way. See Figure 8.4.

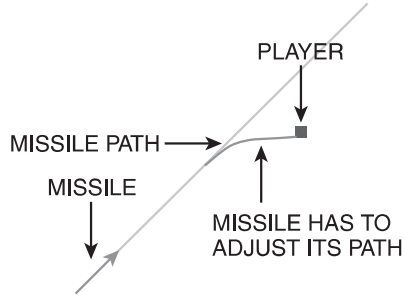


Figure 8.4 Our player moves to dodge the missile.

In the real world, this missile cannot just turn around suddenly and point directly at the player. What this missile could do, however, is begin to make a turn in the player's direction. In order for the missile to travel in the direction it is supposed to travel, it is moved a certain number of pixels to the right and a certain number of pixels up with each form of animation within certain bounds of error.

If we keep incrementing the position of the missile at this constant rate, the missile will follow its original path. Now what if we were to increase the rate at which the missile is moved to the right while keeping its upward momentum constant? The answer is that the path of the missile will change. If we were to adjust the rate of change just right, the missile could be made to arc in the direction of the player and possibly hit him. An added benefit is that this gives the player a chance to actually escape. Although we do want the game to be a challenge to the player, if the game becomes impossible to win, it will be no fun to play at all.

Patterns

Having some enemy characters pursue the player all the time is the most basic form of intelligence. The next step up is to give the enemy the ability to patrol a given area or to use some form of strategy to fight the player. Look at the game in Figure 8.5.

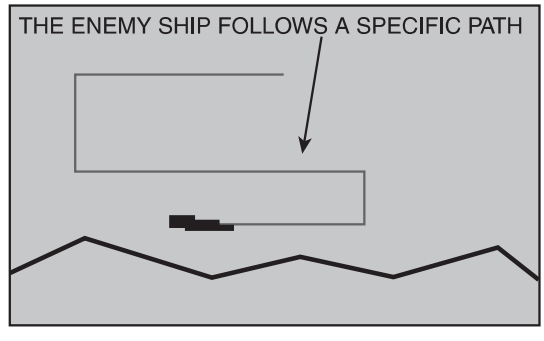


Figure 8.5 Enemy ships do not just sit in one spot waiting for the player to come near. Instead they are patrolling, looking for the player.

Those ships are actively out on patrol searching for our player character. The easiest kind of pattern is for the player's ship to simply fly back and forth over a given area. All patterns, however, do not have to be so simple. See Figure 8.6.

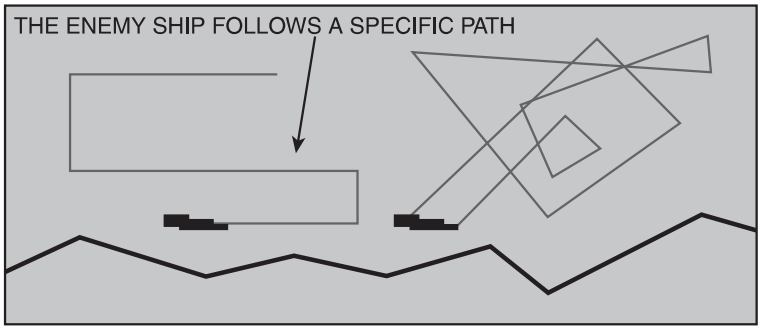


Figure 8.6 The patterns the enemy uses to patrol a given area can be as complicated or as simple as we want them to be.

They can actually be very complex and do things like fly to the top-left quarter of the screen and patrol that area. If the enemy sees the player's ship, then the enemy engages the player. If not, move to the lower right quarter of the screen. Fire random shots just in case a player is hiding out of range of the radar.

Patterns can also play an important role in battle. You see, although it may be acceptable for lower level minions to only be capable of charging directly at the player, higher level bosses should show a bit more class. They should seem to think about what they are doing. Perhaps they zoom around the screen very fast so that the player cannot aim at them, or they stop and release a barrage of attacks. Really, it is up to the creativity of the programmer to produce interesting patterns for the enemy AI to follow.

Random Movement

Completely random and chaotic movement also has its place in the world of arcade games. This kind of movement is often found in games such as space shooters where enemy ships will often come barreling toward the player at full speed with completely unpredictable movements. This adds another element to the challenge of the game because the player cannot predict what will happen next and so must very quickly find some way to either dodge the enemy or get the enemy in their sights so that they can blast them away.

At the heart of random movement is a random number generator. This kind of code generates a random number between 0 and 10. We can make a decision, based on which number comes up, to determine the direction that our player should travel. The next step up from completely raw chaotic movement is the chaotic selection of predefined patterns. Earlier we discussed using patterns that make the enemy appear to think about what he is doing. If all that we do is use patterns, then no matter how complex those patterns are, the player will eventually figure out what the patterns are and how to counteract them to win

the game. We can make our enemy characters seem even more intelligent by having them choose from a selection of a few patterns. This way just when the player thinks that he has figured the computer out, we can throw in a curve that takes him completely by surprise.

Fuzzy Logic

Fuzzy logic is a term coined in 1965 by Lotfi Zadeh, a professor at the University of California at Berkeley. At the time, it formed a revolutionary new approach to data processing. Until then, the only form of logic that was used in the computer industry was binary logic. *Binary logic* can be described as a black and white form of reasoning. Everything is right or wrong, good or bad; you have to turn left or right. While this form of logic was able to solve most kinds of problems, it did not take into account the gray areas of the world that we live in. Fuzzy logic was an attempt at embracing the gray areas of the world that we live in and thus move computers closer to "thinking" the way humans do.

There is often very little "logic" to the way we as humans do things or even why we do things for that matter. Why do we choose to eat the meat off our plate before the rice, or the other way around? Why would you choose to buy a red car as opposed to a yellow one? There is no black and white logical answer to these questions.

The simplest way to think of fuzzy logic is to consider that we will add a level of uncertainty to the kind of logic we have grown used to. Given the same set of inputs, our fuzzy logic code will not always produce the same results. If you ask three people the same question, you are likely to get three different answers. Even asking the same person the same question three times will not guarantee that you get the same answer three times. So it is with fuzzy logic.

Fuzzy logic is actually no less precise than binary logic; it is simply better suited for making decisions based on inherently imprecise concepts.

The reason that fuzzy logic is able to be so flexible is that, unlike binary logic, which represents all of its data as zeros and ones, fuzzy logic can store data as any number between 0 and 1. This means that fuzzy logic data can be represented as 0, 0.1, 0.9786, 0.539, or 1.

Binary data can be used to identify whether something is hot or cold and make a decision based on that information. Fuzzy logic, on the other hand, can determine whether an object is warm, kind of hot, hot, very hot, or scalding, and make decisions based on this information.

One very exciting prospect of fuzzy logic is the ability to create in a game, artificial intelligence (AI), which can cause the game to learn from watching the way a player plays. By keeping track of what a player does in a given situation, for example, the computer, before it attacks the player, can decide whether the player is likely to dodge a particular attack, very likely to dodge it, or not very likely to dodge the attack at all. Based on the outcome, the computer may opt to use a different attack or not to attack the player at all.

Reading Player Input

Giving a computer the ability to think allows us to create a rich environment that is able to react to the actions our player makes as he plays the game. There's just one problem. We haven't yet examined the concepts that will allow the player to create actions in our world. In order for the game to actually be a game, there has to be some way for the player to affect what is happening in the game's world. The most common methods the player uses to affect the game world are typing on the keyboard and manipulating the joysticks.

Here we are going to learn how to retrieve player input from the keyboard and joysticks of the computers covered in this book.

The keyboard was being used to control the action in video games long before the invention of the joystick. The exact method used to find out which key has been pressed is almost exactly the same on all of our game machines and the way we read the keyboard on modern day PCs.

Most computer systems have what is called a *keyboard buffer*. See Figure 8.7. The keyboard buffer is a specific memory location used to store whatever key has been pressed on the keyboard until we have a chance to use it.

All keyboard buffers work in exactly the same way. The only thing that really changes is the size of the keyboard buffer, which, in turn, controls how many keystrokes can be stored there. In the worst-case scenario, you will have one byte of data available for your keyboard buffer. In this kind of computer system, you will only be able to store the last keystroke typed. If you are lucky, you will have a few bytes to use as your keyboard buffer. In this situation, you will be able to store the last few keys the player pressed in the keyboard buffer.

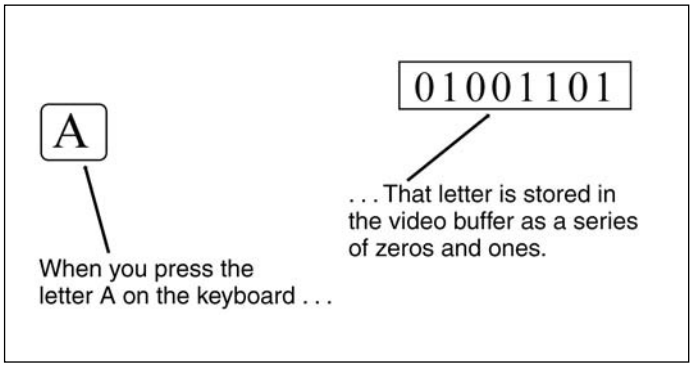


Figure 8.7 Most computer systems have a special memory location called a keyboard buffer that is used to store what is typed on the keyboard.

The first thing that you need to do to read the keyboard of your favorite vintage computer is to visit the companion Web site for this book and find out the size and location of your keyboard buffer. Once you have this information, you have a few options for reading the keyboard buffer. From BASIC, we would use the `PEEK` command.

```
A = PEEK (6502)
```

The `PEEK` command reads the contents of a byte of memory and stores it into a variable. In this case, the byte of data that we read will represent a key that was pressed and stored into the keyboard buffer.

If you are working in assembly language, you would use the `LDA` command to store the byte of the keyboard buffer in the Accumulator, followed by `STA` command that will allow you to store that byte to some other memory location where you can manipulate it.

```
LDA     $100
STA     $5A9
```

As you learned when we studied the history of computer games, the joystick was actually invented by the Tech Model Railroad Club when they needed a better way for people to play their computer game called *SpaceWar!*. The joystick has been a staple of the video game industry ever since.

Reading the position of the joystick is actually very similar to the way that we read the last key pressed on the keyboard. There will be a special memory location in the computer's memory that holds the data that lets us know what position the joystick is in. All that we have to do is read that memory location and translate the data correctly. A list of the memory locations where joystick data is stored can be found on this book's companion Web site.

Modeling Game Physics

We're almost there! You actually have all of the tools that you need in order to build a working game. All that you need to do now is add a few finishing touches to polish the game off and make things operate in a very realistic manner.

If you throw a ball up in the air, it is not going to keep flying until it reaches outer space. Instead, gravity is going to ensure that the ball always comes back down to Earth. If you bounce a ball the height it bounces to depends on the amount of air in the ball. Not everything that is round like a ball will bounce either. Some things will simply smash to the ground while others, such as a balloon filled with helium, will rise high into the sky. The point I want to make is that in the real world there are a number of very important forces that act on all objects. If we are going to include an object in our game and want that object to behave the way it does in the real world, we must understand the forces of physics that affect that object and find some way to include those forces in our game.

There are three forces that will apply to almost every game you make. These forces are thrust, friction, and gravity. *Thrust* is the force that causes objects in our games to speed up. *Friction* is the opposite of thrust and works to slow down objects in our game. *Gravity* is the reason that what goes up must come down.

Thrust

Newton's First Law of Physics states that "An object at rest tends to stay at rest, and an object in motion tends to stay in motion with the same speed and in the same direction unless acted upon by an unbalanced force."

In plain English, the first part of this law means that if a ball is sitting still on a desk, that ball will continue to sit still until someone pushes it or the wind blows it or some other outside force causes it to move. Thrust is that force that causes the ball to move.

Friction

The exact opposite of thrust is friction. When an object is moving through space, a force called friction works to slow it down. Different mediums create different amounts of friction. Water slows a vehicle down a lot more than air, so if we create a vehicle that can fly in the air and in the water, the force of friction should slow the vehicle down more when it is in water than when it is in the air.

Gravity

The force of gravity rounds out our trio of forces. Newton's Law of Gravity states that "Every object in the universe attracts every other object with a force directed along the line of centers for the two objects that is proportional to the product of their masses and inversely proportional to the square of the separation between the two objects."

This definition expands upon most people's concept of gravity. When most people think of the word gravity, they tend to think about the force that pulls objects closer to the Earth. This is only partly right. Actually, every object in the universe draws other objects toward it like a magnet as seen in Figure 8.8.

As Figure 8.8 also demonstrates, the larger the object, the stronger it is able to pull objects toward it. This is why we are able to sense the pull of gravity from the Earth and not from other objects, such as your desk or this book or even other people!

The other limiting factor of gravity is distance. The farther we are away from an object, the less its pull of gravity is on us. That is why astronauts in space are able to float around when they travel far away from the Earth, while we are still held solidly to the ground.

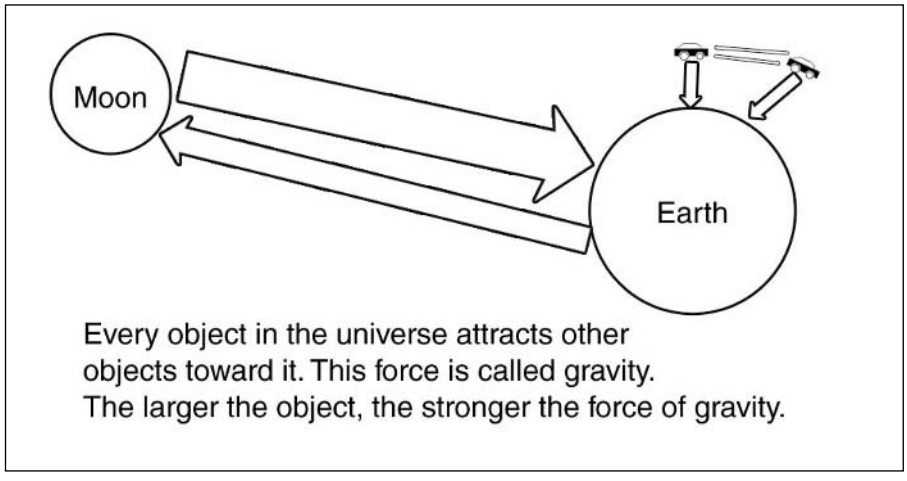


Figure 8.8 Every object in the universe attracts every other object towards it.

Putting All the Forces Together

Very rarely will an object in the real world be subject to only one force. Most objects will be affected by all three of the forces that we just discussed. We need to learn how to apply these forces to an object on the screen. Take into consideration Figure 8.9.

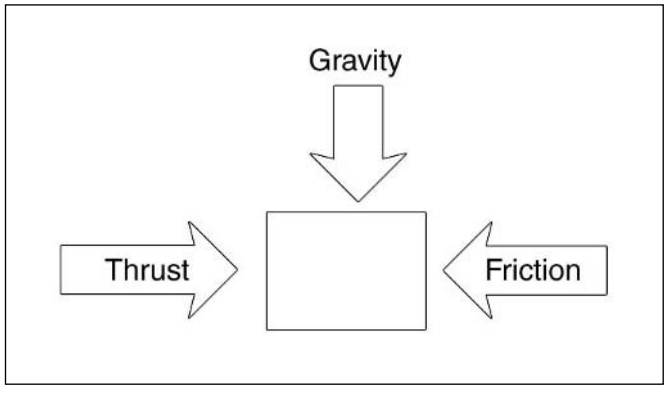


Figure 8.9 Most objects are affected by at least three forces.

The first thing that we have to do is assign X and Y values to our ships. See Figure 8.10. In this case let's use a value of 20 for both variables to put the ship in the middle of the screen.

```
LET X = 20
LET Y = 20
```

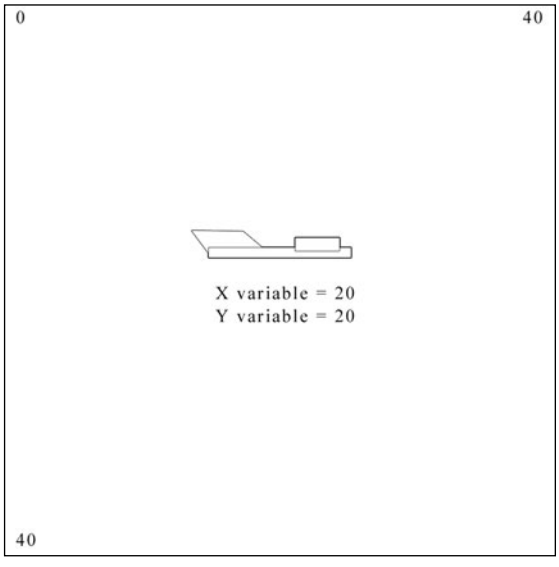


Figure 8.10 We have to give both X and Y variables to the ship.

Right now, the ship is at rest because we are not applying any forces to it. With each cycle, our ship is going to either stand still, or it is going to move in a given direction. During each game cycle, we have to change the values of the X and Y variables to make the ship move in the direction that we want.

Let's define our forces. We will declare a variable called `TF` to represent forward thrust, `TU` for upward thrust, `F` for friction and `G` for gravity. We will assign some values to these variables, but we are not going to use them in the game cycle.

```
LET  TF   =  3
LET  TU   =  3
LET  F    =  1
LET  G    =  2
```

We will start adding these forces to the game cycle, one by one, until the ship behaves naturally. We will begin with thrust. See Figure 8.11. When the player moves his joystick to the right, the `TF` variable will be set to three. When the joystick is moved to the left, that same variable will be set to negative three. When the player moves the joystick up, the `TU` variable will be set to minus three, causing the ship to move up. When the joystick is moved down, the `TU` variable is set to three, which causes the ship to move down. When

the joystick is not being moved up or down, TU will be set to zero. When it is not moving left or right, the TF variable will be set to zero.

In order for this to work, we would add the TU variable to the Y variable in every game cycle. We will also add the TF variable to the X variable.

Right now, the ship will move when we move the joystick, and continue moving in the last direction it was directed to move when we stop moving the joystick.

Next, we need to apply the concept of friction to our game. See Figure 8.12. The force of friction will always resist the movement of an object through space. So if the force of thrust is three, we are going to need a negative value for friction to counteract its force.

Without the force of friction, the ship would never stop moving. With the force of thrust active in our game, the ship will slow to a stop whenever we let go of our joystick.

The next force that we need to bring into the equation is gravity. In the context of most games that we create, the bottom of our screen is going to be the ground and gravity will be the force that pulls objects downward. Some games will be different, such as the first ever video game Spacewar!, which had a star in the center of the screen that acted as the center of gravity. In the case of this game, all objects were attracted to the center of the screen.

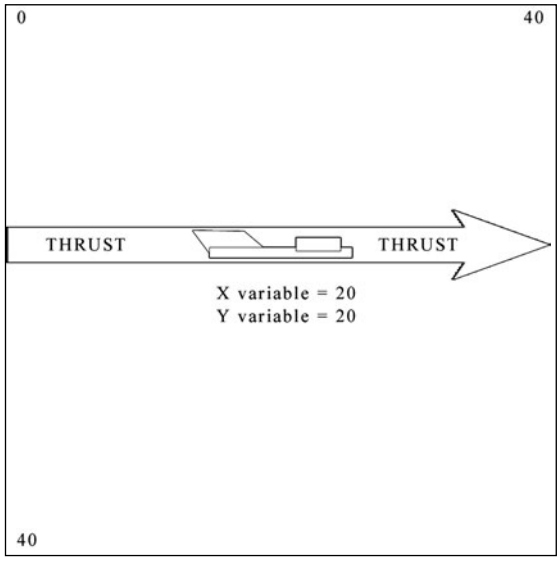


Figure 8.11 The force of thrust causes the ship to move.

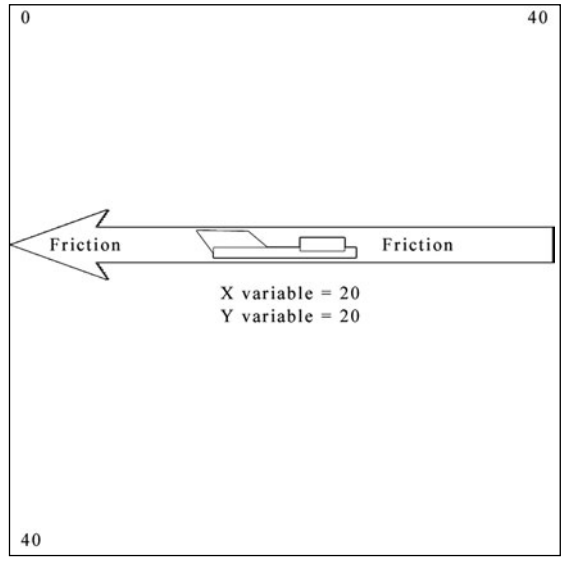


Figure 8.12 The force of friction resists the movement of an object through space.

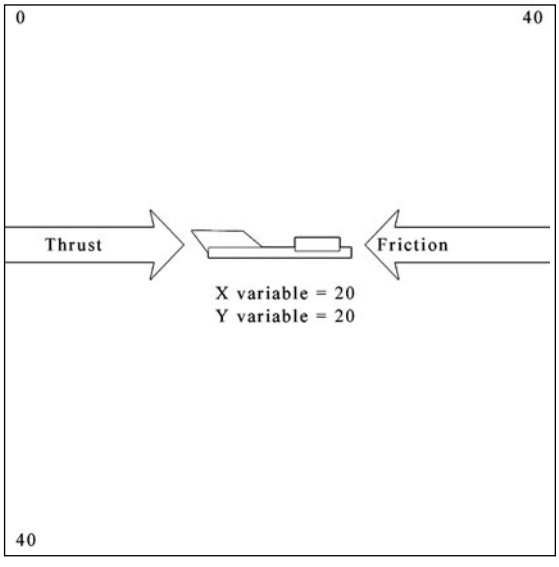
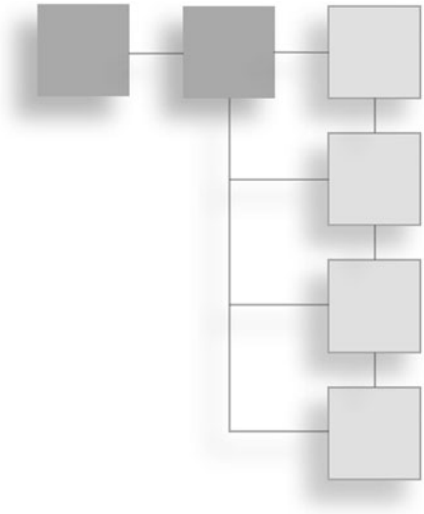


Figure 8.13 The force of friction will counteract the force of thrust.

Conclusion

Game input and physics are important to allow the player to control his character and have that character explore a world that functions the way the player would naturally expect. The information covered in this chapter has familiarized you with the concepts needed to understand the complex issues that go in to building a convincing game environment. We are now one step closer to being ready to build our own video games. At this time, I would suggest going to the companion Web site for this book and looking over the examples you will find there.

CHAPTER 9



Sound Effects

Excuses are the tools of incompetence built upon monuments of nothingness and those who so often use them seldom amount to anything.

Unknown

How Sound Works in the Real World

There are many tried and true analogies used to explain the way sound travels through the air. The example that I will use here is that of a pond with a perfectly still surface. Picture in your mind a small pond that is perfectly still. Now imagine if you threw a stone into the center of the pond. There is going to be a splash where the stone enters the water. Within seconds, ripples will extend from the point of impact and spread over the entire pond.

Sound travels through the air in the same way, only rather than a stone, the ripples are caused by vibrations and rather than a pond, the waves travel through the air. These vibrations can be caused by any number of sources; the speakers in a radio, a car alarm, a musical instrument. Even the keys on my keyboard make a noise as I type this book. These waves are what we call sine waves and look similar to the diagram in Figure 9.1.

As you can see the wave goes up, then down, then up, then down. Each time the wave goes up and then down, it is said to have completed a *cycle*. The distance from the start of one cycle to its end is called the *wavelength* and the height of the wave is called its *amplitude*.

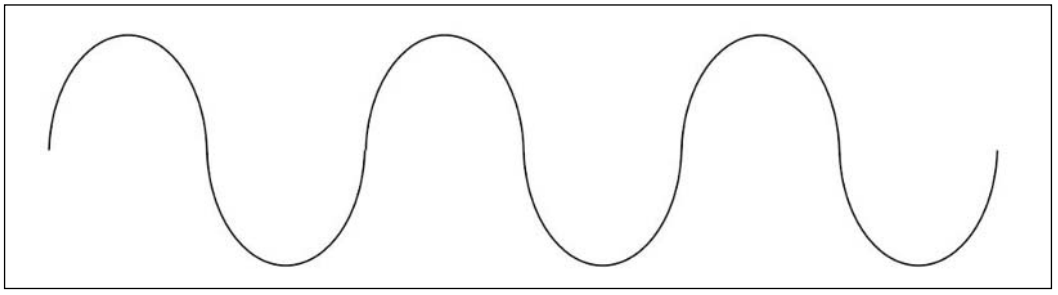


Figure 9.1 Sound travels as sine waves.

The unit of measurement for sound is the *hertz*. This is a measurement of the number of cycles that occur in a second. The shorthand for this unit is Hz. It will often be used with several different prefixes that you are already familiar with. Examples of two prefixes are M (for mega or one million). You are already familiar with this term and its use in measuring disk space. Another prefix is K (for kilo or 1000). Most retro machines measure their memory in kilobytes. In the example above, rather than writing 20,000 Hz we would write 20 KHz.

Whenever an object vibrates, it compresses and expands the air molecules around it. These compressions and expansions spread through the air in every direction in the form of waves. When these sound waves reach our ears, they cause tiny hairs in our ears to vibrate. Finally, if these are between 20 and 20,000 cycles per second (20 to 20,000 hertz or 20Hz to 20,000Hz), then these vibrations are converted to signals, which are sent to our brain where they are interpreted as sounds we recognize. Humans cannot hear any sound with a frequency outside of the 20 to 20,000 hertz range. Other creatures have much greater ranges of hearing. A dog can hear sound in the range of 67 to 45,000Hz; cats can hear from 45 to 64,000Hz; a cow can hear from 23 to 35,000Hz; and finally, a porpoise can hear in the range of 75 to 150,000Hz. This is how dog whistles work. Your pets can hear the sound and be trained to react to it in a certain way. Humans cannot hear the sound, so it does not irritate anyone.

The speed of sound is not always constant. It changes depending on the density of the substance that it is traveling through. The closer the molecules are packed together the less time it takes for one molecule to bounce against another; therefore, the wave travels faster.

Solids are packed the most densely, followed by liquids and, finally, gasses. As a result, if you tap on a desk, the sound will travel from one end of that desk to another before it gets to your ears. If the same sound is made in the air and in a body of water at the same time, the sound will travel farther more quickly through the water than in the air. At sea level in dry air, sound will travel at roughly 770 miles per hour. Despite what you have seen in endless Sci-Fi movies, sound will not travel in outer space.

Scientists are able to find the wavelength of a sound by dividing the speed of a sound by its frequency.

Speed ÷ frequency = wavelength

Sounds with lower frequencies tend to have longer wavelengths and produce a lower pitch. Inversely, sounds with higher frequencies usually have a higher pitch and shorter wavelengths.

note

People often say that there are no molecules in space. In reality, there are molecules in space, and indeed, there is actually wind in space! The molecules are simply spaced very far apart, and the wind is not as dense as what we have here on earth. This wind, which is sometimes considered the "interplanetary medium," is estimated to fluctuate between 5 particles/cm3 and 100 particles/cm3 and has a temperature of 100,000 Kelvin. This wind is created by a dense stream of charged particles emitted by the sun at roughly 450 km/sec. These winds have very slight effects on the paths of spacecraft and combine with much higher energy particles ejected by solar flares to cause radio interference, power surges, and even the Aurora Borealis here on Earth.

Mimicking Real World Sounds on a Retro Game Machine

Often a problem may seem overwhelming. As a programmer it is important for you to learn to break a problem down into its simplest parts. These smaller parts are usually much more manageable. The problem that we have to solve is how we are going to get our digital retro computer system to play analog sound.

The only tool that we are guaranteed to have in every machine we program is a built-in speaker and some form of digital soft switch that we can use to make the speaker click.

So what we end up with is a basic sound element: the beep. Just as we use pixels to make up an approximation of a picture, we can use a beep to make an approximation of a sound or even music. Our brains will fill in the blanks enough to figure out what sound we are trying to make. The context of the game also has a lot to do with how the player perceives a sound. The same sound may be interpreted as a laser or a shooting star or a crash depending on the kind of game, what's happening in the game, and what you tell the player in the user's manual about that sound.

We know that we can vary the sound that is generated by adjusting the frequency of beeps generated by the computer. Look at the following algorithm.

```
1 Beep computer
2 goto 1
```

This algorithm generates a very high-pitched tone. What if we did this?

```
1 beep computer
2 delay computer by x microseconds
3 goto 1
```

This time we generate a different frequency. By changing the value of x, we are able to generate sounds of different frequencies.

If we repeat this process several times using different values for x we will be able to create a basic melody.

Computers with Special Sound Hardware

You will not always be limited to working directly with the speaker in order to generate sound. Some retro game machines actually have very complex sound hardware built into them. The Commodore 64, for example, was so sophisticated that after the computer was no longer in production, sound cards for PCs were being created using not only its technology but also the very same chips that were in the machine.

Before we take a look at how these systems work, we need to cover a bit more sound theory. You know that every sound can be played at a particular volume. The problem that we have is that sound does not just appear out of silence, play, and then go back to silence. There is a process that occurs, and if we ignore this process, then our sounds will be "flat."

The volume of the sounds we generate will actually increase until it is higher than the volume that we need to produce. The frequency then decreases until it reaches the volume that we want. The volume stays where we want it for a while and then it decreases back to zero, or silence. Each of these four phases actually has a scientific name. These names are labeled in Figure 9.2.

The period of time when the volume is rising to a point above our desired sound is called the attack. When the volume is falling to the level that we want it is called decay. While the volume is where we want it is called the sustain and finally the period where the volume goes back to zero is called the release.

The benefit of computers with specialized sound hardware is that we have more control over attack, decay, sustain, and release.

The Commodore 64

When we were working directly with the speaker, the basic sound element that we had to work with was the beep. We adjusted the frequencies used to generate the beeps to make different sounds. This time around, we have a bit more control over how we are going to be generating sound. First of all, we have three "voices" to work with. When we were

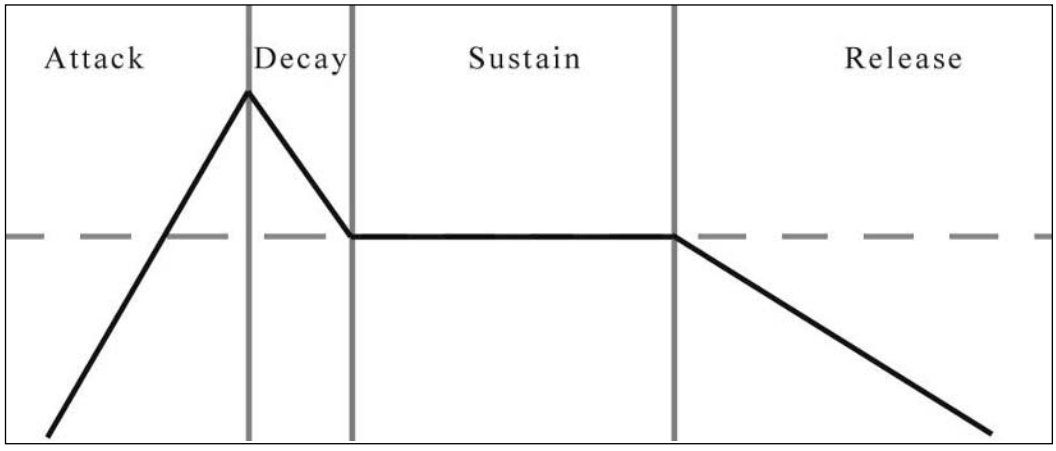


Figure 9.2 Sounds in the real world are made up of four parts: attack, decay, sustain, and release.

working directly with the computer, we could effectively only generate one sound at any given time. When we work with advanced sound hardware, we have the ability to control more than one voice, all of which can be played at the same time. Each voice has a number of properties that we can control by writing to specific memory locations.

Sound on the Commodore 64 is generated by the 6581 Sid chip. This chip has three synthesizer voices, which means that it can play three sounds at once. We are able to control attack, decay, sustain, and release on each channel by writing to specific memory locations. Examine Figure 9.3.

As you can see, the lowest four bits of the first register control the decay, while the highest bits of that same byte control attack. Likewise the lower four bits of the second register control release, while the highest four bits control sustain.

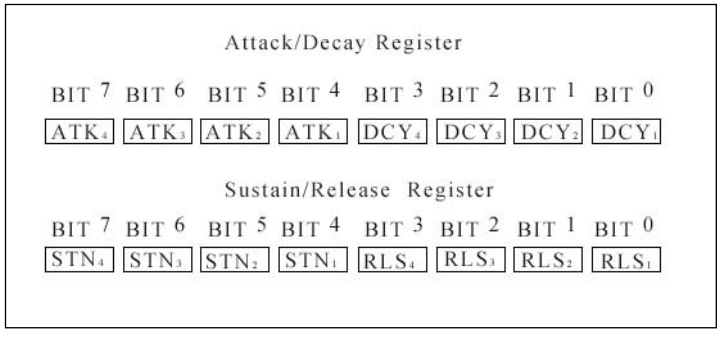


Figure 9.3 Here is how the registers that control attack, decay, sustain, and release are arranged.

We can set the values for attack, decay, sustain, and release to values 0 through 15. These values correspond to a pre-established amount of time as can be seen in Table 9.1.

Table 9.1 Attack, decay, sustain, and release Timing

Value	Attack Rate (time/Cycle)	Decay/Release Rate (time/cycle)
0	2 ms	6 ms
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	240 ms
8	100 ms	300 ms
9	250 ms	750 ms
10	500 ms	1.5 s
11	800 ms	2.4 s
12	1 s	3 s
13	3 s	9 s
14	5 s	15 s
15	8 s	24 s

The address of the relevant registers can be found in Table 9.2.

Table 9.2 Location of the relevant registers.

Voice	Memory Address(hex)	Type
1	05	Attack/Decay
1	06	Sustain/Release
2	0C	Attack/Decay
2	OD	Sustain/Release
3	13	Attack/Decay
3	14	Sustain/Release

The lower three bits of SID register 24 control the overall value of sound from the computers.

The Atari 400/800

At your disposal is the advanced sound chip called POKEY. This bad boy supports four independent voice channels. This means that we can play four separate sounds simultaneously and completely independent of each other. Each channel has its very own frequency register that can be used to determine the note being played, a control register to keep track of the volume, and the noise added to that channel. There are also a lot of other features, which allow you to add some cool effects such as high and low pass filters. So let's get started.

Basic Sound Command

Probably the best and easiest place to start in our discussion of sound is the Atari sound command. Here is the basic format for using this command.

```
SOUND voice, pitch, distortion, loudness
```

We have four separate sound channels at our disposal. See Table 9.3. The voice option allows us to choose which of the four voice channels we are going to use. A value of 0 will reference channel 1, a value of 1 references channel 2, a value of 2 references channel 3, and of course, a value of 3 references channel 4.

Table 9.3 The Atari Has Four Voice Channels

Voice channel	Reference number
1	0
2	1
3	2
4	3

The pitch option controls the frequency of the sound being produced and can range in value from 0 to 255. The clock produces a steady stream of pulses. If we set our pitch option to 255, then all of those clock pulses would go to the speaker. We would hear the highest pitch that the computer can generate. If we set this pitch to zero, then almost none of the clock pulses would make it to the speaker. We would hear the lowest frequency that our computer can generate.

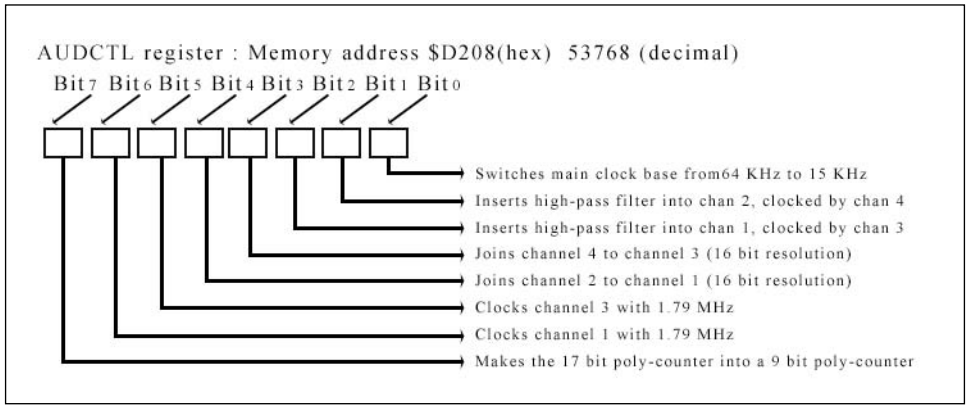


Figure 9.4 The AUDCTL register controls the global settings for all of the sound channels.

In the real world, sound is sometimes distorted by various natural phenomena that occur in nature. Often, we will want to recreate this in our games. The distortion option of the sound command allows us to do just that. By selectively removing some pulses from the sound wave being produced, the computer can actually simulate distortion that occurs naturally. If we set this value to 10 or 14, the computer will generate pure tones. Using other even numbered values will introduce different amounts of noise to the tone being produced.

The final option used by the sound command sets the volume of the tone that we are creating. This can be set to any value from 0 (which will be completely silent) to 15 (which is the loudest). In order to get the best quality sound, it is suggested that you do not allow the sum of the volume of all of the channels to equal more than 32 because this can over modulate the audio's output and cause a degradation in the quality of the sound produced.

For all its power, there is one major disappointing element lacking in the sound command, and that is an option to control how long the sound is produced. Once the sound command is used to start generating a tone, the computer will continue generating that tone until another sound command is used to either shut off the sound or change the nature of the tone. For this reason, we are forced to resort to the rather crude method of using `FOR…NEXT` loops to control the timing of sound in our game.

Even though numbers can vary depending on runtime conditions, an empty `FOR…NEXT` loop can be executed about 450 times per second. Furthermore, an empty loop, where I = 1 to 225, takes approximately 1/2 second to execute; thus, we can use a number of `for` loops whose counters are either 225 or multiples or factors of 225 to generate the timing for our musical score or sound effect. A few examples of this can be seen in the following

code samples, which can be used to generate several common sound effects found in arcade games.

Sample for loop that simulates a pistol being fired:

```
10 FOR I=10 TO 4 STEP -0.25
20 SOUND 0,10,0,I
30 NEXT I
10 FOR I=15 TO 0 STEP -0.5
20 SOUND 0,20,2,I
30 NEXT I
```

Here we see an example of mixing a number of different sound commands and timing FOR NEXT loops to generate the sound of a falling object:

```
10 FOR I=30 TO 200 STEP 3    REM START FOR LOOP
20 SOUND 0,I,10,I/25         REM PLAY SOUND BASED ON I
30 FOR J=1 TO I/10:NEXT J    REM ADD A DELAY
40 NEXT I                    REM END LOOP
50 SOUND 0,20,0,14           REM PLAY SOUND
60 SOUND 1,255,10,15         REM PLAY SOUND
70 FOR K=1 TO 150:NEXT J     REM ADD DELAY
80 SOUND 1,0,0,0             PLAY SOUND
```

Now if the only thing in the world that our program did was play sound, then this would be no problem. Unfortunately, our programs are games and, hence, will be doing much more than playing sounds, which means that we have a very big problem. Imagine that we have a number of FOR loops and sound commands that are combined to create a nice background sound for our computer. The timing for this musical score is based solely on the timing set up by the for loops. But if the computer is locked up in these FOR loops, it has no opportunity to execute code used for the actual playing of the game. On the other hand, we can create a game loop and integrate the FOR NEXT loops for the sound inside of this main game loop. The sound will play and the game will play, but at one time, the timing between two sound commands may be half a second while at another, it may be ¼ of a second and yet another time one second. Obviously, this is unacceptable. There has to be a more reliable way to implement the sound for our game.

If we can execute the code to play our sound during the vertical blank we will be able to play our sound even while the game is playing without any adverse effects on the timing of our sound or our game. The only possible way to accomplish this is to write an assembly language program that will be used in conjunction with our BASIC program in order to give us the power we need over the POKEY chip in order to unleash the full potential of the Atari's sound hardware.

Assembly Sound Programming

There are eight registers that are used to control sound when we are working in assembly language. These registers are as follows:

```
AUDF1=$D000   AUDC1=$D001
AUDF2=$D002   AUDC2=$D003
AUDF3=$D004   AUDC3=$D005
AUDF4=$D006   AUDC4=$D007
```

There are four AUDF registers, which control the frequency on the sound for each available voice channel and can have a value ranging from `$00` to `$FF`. Four AUDC registers control both the volume and distortion of the sound channels.

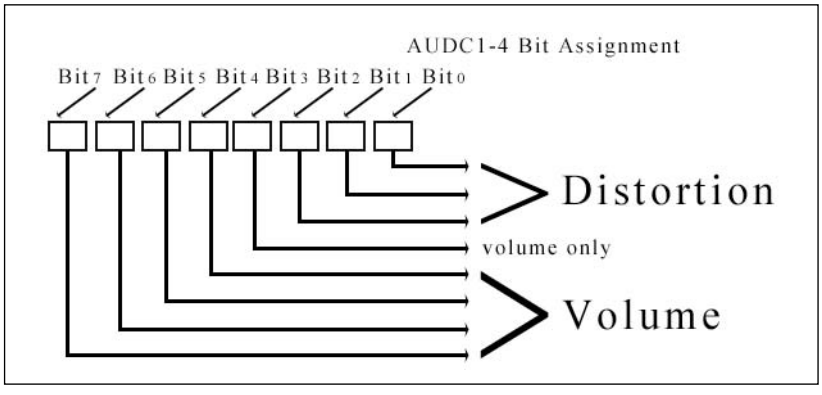


Figure 9.5 Here is the make-up of a general AUDC register.

In addition, there is also an `AUDCTL` register, which can be used to control the global settings of the Atari's sound hardware.

The frequency of our tone is based largely on the clock frequency. By setting bit 0 of the `AUDCTL` register, we are able to switch the base clock between a rate of 60 and 15 kHz.

By setting bits 1 and 2 of the `AUDCTL` register, we are able to implement a high pass filter. This is a basic effect that is used a lot in stereo systems but can also have some use in creating certain sound effects in our games. Basically, what this feature does is use one channel as a rule to measure all of the other channels. So if, for example, channel one is playing at a given volume, the only way we will hear any of the other channels is if they are playing a sound equal to or louder than that being played by channel one.

Bits three and four are used to control a very interesting and powerful feature of the `AUDCTL` register, the ability to combine two voice channels to form one channel with a larger frequency range. Whereas previously we could only use a frequency value between 0 and

255, we would now have the ability to enter a frequency value between 0 and 65,535, which means we can generate the tones and notes for our musical score with a great deal more precision.

Take another look at the general makeup of an AUDC register. As you can see, three bits are reserved to control distortion of sound, but we are not sure exactly how this works. Let's examine this concept.

We know how the tones are created on the Atari. The base clock generates a base frequency. The Atari then uses the value that we give it to divide the base frequency by to give us the final tone that will actually be heard by the person playing our game. When we introduce the concept of distortion into our channel, the Atari generates an irregular frequency in which random pulses will be missing. Both the tone and the irregular wave are used as inputs for a comparator circuit. Basically, a comparator circuit is one that will only generate a pulse when both of the inputs are receiving a pulse.

As you can see, the net effect is that some of the pulses from our tone will not be heard. This is how distortion is implemented on the Atari. Just as the system clock is the basis for the creation of the frequency of our tones, the polynomial register is the foundation of the timing (or lack of timing) for distortion on the Atari. Bit seven of the AUDC register is used to switch the polynomial counter from between a 17-bit and a 9-bit counter.

Now that we understand the basics about the POKEY registers that are available to us, let's revisit the problem of playing music and sound effects on the Atari with our interfering with the timing of either our game or the musical score.

We need to find a point in time when the computer is free and not doing anything in particular. Sounds impossible, right? From what we have seen so far about the Atari, it does not seem possible that there is any time when the computer is not busy. In reality, there is such a time. As a matter of fact, this happens 60 times every second. You see, 60 times per second the screen is redrawn. After each time the screen is redrawn, a vertical blank occurs, and each time a vertical blank occurs, it generates a vertical blank interrupt.

Normally, this interrupt is used to do nothing more than perform a few housekeeping tasks, such as maintaining timers and other tasks that need regular updating. We have the power to take over this interrupt and execute any code that we wish every time this interrupt occurs. What this means is that we can write a small group of code and have it executed consistently 60 times every second! This is the key to our problem.

Let us suppose that we want to play some invigorating music to get the player's blood pumping as he gets ready to fight the final boss in our game.

Rather than using a number of commands and timing mechanisms to generate the music, we would pre-calculate the notes that we are going to be using as well as the length of time each note will be played and store them in a table. We would then create an interrupt

service routine, which will be executed each time a vertical blank interrupt is called. This program will simply read the current note that is to be played from the table and play it. Sixty times per second, we can update the sound that is being played on any channel at any time.

This method of playing our music has absolutely no effect on the speed and accuracy of our main game code. In fact, the main game program is not even "aware" that the code playing our music even exists. Furthermore, because of the consistency with which this code is called and executed, we are 100 percent guaranteed that the timing in our music will always be the same and always be correct.

Once we have designed the code to play our music and have created a lookup table with our musical notes, we are ready to install it as an interrupt service routine. The procedure is really pretty straightforward. First, we have to place our program and its table into the computer's memory. Remember that interrupt I told you about? The interrupt service routine for this interrupt is called `XITVBL` and the address of this interrupt is stored at memory location `$224` low byte, `$225` high byte. Before we replace `XITVBL` with our code, we have a way of calling this interrupt. We still want this interrupt to be called after our code is executed so that all the housekeeping that it is supposed to do is still maintained. By making sure that the last command in our interrupt service routine is `JMP $E462`, we ensure that at the very time the vertical blank interrupt is called, our code is executed, thus playing our music, performing housekeeping, and finally, giving control back to the main program.

Next we store the low byte of the address in the Y register and the high byte in the X register and execute the following command:

```
JSR     $E45C
```

This command makes a call to the `SETVBV` function, which takes the address of our interrupt service routine (which we store in the X and Y register) and sets that as the address that will be called when a vertical blank interrupt is executed.

Conclusion

Sound programming is an important element of our video games. At first it seems unlikely that we are going to be able to use our digital computer to produce analog sound. Knowledge of this skill rounds off the techniques that you need to build your own retro games. Like all chapters in this book, tutorials, source code, and demo programs can be found on the book's Web site in order. The more you experiment with the code and see it in action, the better you will be at programming both sound effects and musical scores for your games.

CHAPTER 10

Putting It All Together: Building Games

> CHRIS: The bane and blessing of human nature. That old cat killer, curiosity. Something so deeply embedded in our psyches that it screams to us from ancient myths of Pandora. Eve. Lot's wife.
>
> JOEL: Eve lost Paradise, Lot's wife was turned into a pillar of salt. Knowledge doesn't come cheap, my friend.
>
> CHRIS: Good or bad, curiosity is woven into our DNA like tonsils or like the opposable thumb. It's the fire under the ass of the human experience!
>
> Jeffrey Vlaming, *Northern Exposure, The Final Frontier, 1992*

Great! You made it. I'm happy to see that you have gotten this far! The majority of this book has been geared toward providing you with the ingredients needed to build a retro video game. This chapter is different in that it focuses on teaching you to take those ingredients and put them together in a playable recipe.

There is nothing really complicated in this chapter. If you have made it through all of the other chapters in this book, then you can make it through this one. All you will do here is create the rules and guidelines that you need so that you can apply your newfound knowledge to any retro computer system you choose.

The Universal Game Structure

When your computer starts up, it is not a clean slate ready for you to write on. It is more like a chalkboard that has chalk and paint and pencil markings all over it. Before you can do anything useful with it, you have to clean it up and make sure it is well organized. You saw an example of this when you learned to set, and later to clear, your computer's video buffer. When you set your computer to a high resolution, for example, the screen was full of random dots of different colors. Even if you drew an image on the screen, it would be hard for the player to recognize it among all of the chaos surrounding it. After you clear the screen, however, it becomes much easier for a player to see what is going on. It is for this reason that every computer game starts with what is referred to as *initialization code*.

Initialization

This is always the first part of any game's code. It is here that you clear the screen, load any graphics you need from the disk, and set up your variables. After this code is executed, you are ready to enter the next phase of your game, which is called the *game loop*.

The Game Loop

If there is one idea I have repeated more than anything else in this book it is that all any game ever really does is draw an image on the screen, manipulate some binary data and then draw another image on the screen. That phrase sums up in its entirety what the game loop does. The game loop is the name given to the actual code responsible for performing these actions.

Cleanup

On a game console, all the player ever does is play games. Usually, for them to change games, they have to turn the computer off and put a new game in. That means the games do not have to worry about the state that they leave the computer's memory in. On a personal computer, however, you do not have that luxury because the player uses his computer for more than just playing games. After he chooses to exit your game, he may want to use a word processor or perhaps even play another game. In a perfect world, the next application that runs will clean up the memory address that it wants to use. This is not a perfect world, and some applications that the player is using will just assume that all is well with the system. It is for this reason that you have to always be sure to leave the computer in exactly the same state that you found it. This is what the cleanup code does. It clears any memory that you may have used and corrects any changes you may have made to the OS or the environment, including the screen mode.

In many ways, retro game programming is simply putting pieces of a puzzle together to form a picture. You know the elements of a game (graphics, sound, AI, etc). You know the order that they have to be used, and you have an idea in your head concerning the kind of game that you want to make. The trick is in knowing how to put the pieces together. Let's take a look at some basic concepts for making simple retro games. Once you can do this you can build some more complex games.

Programming Text-Based Games

The first question that anyone who has never played a text game and grew up in a Quake 3 world may ask is "What's so special about a text-based game?" Today's games live or die by their graphics. How can you be impressed by a game that has absolutely no graphics at all? Well, the answer to that question is that no computer game system available today is able to create the kind of graphics found in those old text-based games. The reason for this is that today's games depend on inferior technology. Games today depend on silicon and megabytes and RAM and ROM. They use video cards made by companies like, Nvidia™, and ATI™. Text-based video games, on the other hand, are powered by imagination and are rendered by the human mind!

Before there were computers and, hence, computer games there were books. Before there were books, there were storytellers. The storyteller was perhaps the first entertainer. A skilled narrator could make a listener feel like he was in some far away land fighting mythical dragons or conquering lands, winning the love of a fair maiden, or even just floating along on a cloud. When the storyteller told a single story, she was really telling a multitude of stories because each person who heard her pictured himself living out the adventures told in the story. Countless dreams have been the result of a good tale told the day before.

In time, books were created, allowing the storyteller's tale to travel to countries she had never seen and whisper into the ears of people she had never met. Millions of people could now read a story that once would only have been heard by a handful. The advancements in printing technology made this possible, but for a long time there was still one major limitation. Interactivity!

You could read a book from cover to cover and create your own adventures in your imagination. Then you could sit down and use your imagination to create your own versions of the adventure, but you could not actively modify the story of the book as you read it. If a character in the book has a choice to take the left or the right path and the author has the character take the left path, then every time you read that book, the character will take the left path.

This limitation was overcome eventually through interactive books. These were very interesting documents because the story was not meant to be read from cover to cover. Instead if, for example, the book reaches the point where a character has to choose between the left or the right path, a note would be placed at the bottom of the page telling you to turn to a specific page if you want the character to turn left and to another page if you want him to turn right. What this meant was that the rest of the story that you read would be different depending on which choice you have the character make. Not only did this make the story much more interesting and enjoyable, but it also added to the reread value of the book. You could read the same book 10 times and read a different story each time!

Text-based games are the next technological step up from these "interactive books" and were perfect for early computer systems that were still a very long way from producing a photo-realistic video display. Most of these games had you exploring an abandoned mansion or a dark dank cave. These environments were the most common for two reasons. First of all, the most natural genre of games to translate into text-based games were the old dungeon crawling board games, such as *Dungeons and Dragons*. Second, because of the way that these games operate, it is easier to create a modular environment such as a house that has defined rooms than it is to create an abstract wide open expansive environment such as a desert or a forest.

What Is a Text-Based Game?

As the name implies, a text-based game is a game that uses text instead of graphics to create the gaming environment.

A computer game can be either turn-based or real-time. All text-based games were turn-based. An example of a real-time game is *QUAKE*™ 1 through 3. In these games, all characters in the game are moving simultaneously. While you are shooting at a player, that player can shoot back at you. An example of a turn-based game is the *Final Fantasy*™ series. In these games, if there are three characters on the screen fighting each other, they each have to take their turn fighting. First, you will attack, then your team member will attack, and finally, the enemy will attack. The last man standing wins.

The combination of being text-based and turn-based meant that these games could be played even on the most modest of game systems. Graphic-based games do not share this luxury. It takes many times the effort and resources to recreate the imagery described figuratively in a large text-based game literally as a graphical adventure. A text-based game can use generic instructions and does not need special access to memory, which means that they can run exactly the same on multiple game systems. Graphics-based games use machine-specific commands and reference machine-specific memory locations, which means that they are not platform portable like graphical games.

As you can see, a text-based game is really an electronic interactive book that can be enjoyed on almost any computer system. Now that you know what a text-based game is, let's learn how to make one.

Building Your First Text-Based Game

Building a text-based game starts the same way building any game, including the most advanced 3D animated games, or the biggest blockbuster movie, begins. You write a good story. Although there are some obvious exceptions to this point, such as *Tetris*, where a story is of little or no importance, if a game does not have a good story line, it is not going to be very compelling to play. Let's create a story line for your first text-based games.

The Story

"You are Tom Smith, and the fate of the world is in your hands. You are the only person in the world skilled enough to fight legions of ghosts and goblins that are threatening to invade the whole world starting with your town. You have killed almost all of the ghouls and you are ready to go after the head honcho. He, along with the strongest ghost he could find, are holed up in Old Man Peterson's house planning their next move. It is up to you to stop them.

You are standing at the front door of the house. What will you do?"

Our goal is to paint a picture for the player as well as to elicit an emotional response. You want the player to be filled with a sense of trepidation, yet at the same time you want him to feel that there is a very important mission that he must fulfill. After all, the entire world is depending on him.

The Lay of the Land

Our story is the gateway to the world your game is being played in. It places the player in the correct frame of mind to play your game. It is important for us to maintain this illusion by creating a believable game environment. Whatever emotional response you are able to elicit from the player after he has read your story should be maintained as the player plays the game. In graphics-based games, this is done by placing graphic images on the screen that look as much like the imaginary environment as possible. In text-based games, the player's environment is created through the use of figurative and descriptive language. This gives text-based games a definite advantage over other games of the era.

A game designer could envision a creepy old haunted mansion and draw an image on the screen. The image could convey the point that the player is in a dark haunted mansion and draw it on the screen, but that picture (especially with the limited graphics functionality

available) may not fully capture what the designer was thinking when he created the game. On the other hand, through the clever use of words, the game designer could render in your mind's eye a much more detailed image of what he was seeing as he wrote the game.

Before you can start putting together your potent, passionate, image generating prose, however, you have to take a collective look at the world that you want your player to live in and then create a map of this world.

Let's think. What would a haunted house full of ghost and ghouls look like?

Well, it is a house, so you can assume that this house will have a few basic rooms: living room, dining room, bedroom, bath room, and kitchen. To add a bit of variety to the game, you will add a study and a small corridor filled with art.

The rooms have to be arranged in some logical fashion. The arrangement of the rooms of the house is called a floor plan and can be created in a number of ways. You can use the floor plan from your house or office, you can look at real floor plans from architectural magazines, or Web sites, or you can create a floor plan from your imagination. Figure 10.1 shows the floor plan for the game that you are creating.

You know that the game will take place in a house, you know that the house is haunted, and you have a general idea in your mind of what the house will look like. Now you have to let your mind wander a bit farther.

Close your eyes and imagine yourself standing outside the front door of your haunted house. Imagine the look, feel, and smell of your environment.

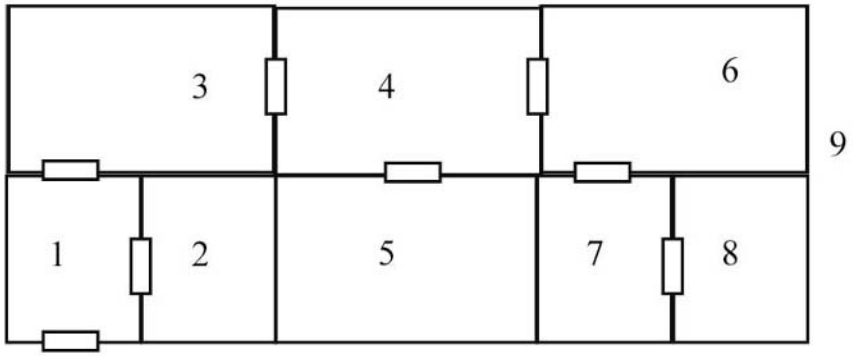


Figure 10.1 Floor plan for your first text-based game.

Look around, take in everything you see, hear, smell, and touch. Now imagine yourself as you open the front door. It creaks a little as you open it . . . you hope that the noise has not alerted any of the uninvited guests inside . . . a black crow flies overhead, and you shiver as a cold breeze blows out of the house and kisses your cheek.

It's too dark to see anything, so you light the kerosene lamp in your hand . . . the lamp casts strange shadows on the walls . . . you almost drop it as the reflection from your lamp passes over an old jacket hanging on a rack casting a shadow that looks more like a dragon than what it actually is.

It's unusually dusty in here for a house that was fully occupied until yesterday. A strange, bittersweet stench fills the air, and you follow a trail of rancid smoke trailing from the kitchen.

Apparently, someone left a pot of soup on the stove and forgot to turn off the burners. As you look inside the pot, you see the charred remains of bones, human bones . . . this is going to be a long night.

Despite the smoke, this room is impeccably clean. There is no dust, no dirt, and apart from the rancid odor and smoke filling your nostrils, there is nothing to see here . . . time to explore the rest of this house. You leave the kitchen, and you are once again standing in the living room. To the north, you see a door and you walk through it. This is obviously the dining area. There is a large table big enough to seat at least 10 people. Candles are still burning though they are producing remarkably little light. Five of the 10 places have plates of half eaten food. The other five have their place mats neatly set aside.

Whoosh.

A cold breeze has just blown out all of the candles. In the dim light, you see strange figures dancing along the wall, and at first, you think that they are just more strange shadows from the light of your lamp. Then you realize the breeze has blown out your lamp as well. Whatever is dancing on that wall in front of you must be real!

As suddenly as they came, the images have vanished. Were they real or were they a figment of your imagination? You will, unfortunately, know soon enough. It's time to move on.

A door to the east of the room takes you into a long corridor. The walls are filled with priceless paintings. Hold on, isn't that the rare painting that was just stolen from the museum? It's worth well over two billion dollars. What in the world is it doing here?

You do not notice it, but the eyes of many of the paintings are actually moving. They are turning to watch you as you walk by. The statues that you have passed have now moved and are making threatening gestures toward you.

This is not a good place to be right now. Let's explore the rest of the house.

A large glass door has brought us into the study. There are papers on the desk . . . strange papers written in a language that you do not understand. They seem to be diagrams and maps . . . maps of the town . . . maps of the whole country . . . maps of the globe and all of the electronic communication and transport for the entire world. What is going on here?

Wait, something is wrong, but you can't quite put your finger on it. That's it; there are footprints in the dust on the ground . . . footprints that are not yours . . . footprints that were not there when you came in!

Very strange. You walk through the bedroom door.

The only thing that separates this room from a hotel room is that there is no mint on the pillow.

That and the stench coming from the bathroom door. Someone must have used it recently. But what is that strange light coming from under the door?

You go to investigate, but as soon as you open the door there is silence. A bright light . . . you fall to the floor. You have not won the fight on this night."

The above narration is an example of a mental journey that you should take to help you design your game. At this stage you did not describe any monsters or even any elements of the game play. You simply think of the environment that you want to build. After you have a very good understanding of what the environment is going to be like, you can carry out the same exercise again but in more detail.

Picture yourself standing outside the front door, only this time, write down what you "see" on paper (or type it in a word processor). This will be the description that you give the player when he first starts the game.

Next, picture yourself in the living room. Describe it to yourself in great detail, and once again, write this description down. This is the description that you will give to the player when he enters the living room.

Go through the steps of describing each of the rooms in this manner.

As you describe the rooms, there is a kind of middle ground that you need to walk. You see, if you do not give enough detail in your descriptions, it will be hard for the player to picture the room in the same way that you see it in your mind's eye. If you give too much detail, there is little room for the player to use his own imagination.

Apart from the correct use of detail, you have to be sure to be as creative as possible. Think of clever traps and pitfalls that you can create for your player to figure out. For example, the owner of the house may be a rock climber and have a room where he keeps all of his rock climbing supplies. The player may need to go to this room and find a rope before he can climb through a trapdoor that is well hidden beneath a bed in one of the other rooms into a strange, underground cavern.

You may create rooms where the floor automatically falls apart, hurling the player down a bottomless pit. No one says that your map has to have only one floor. You can incorporate stairs and elevators into your game design. Most of all, be as creative as you can possibly be when you describe the room so that the player feels that he is really in your world and can actually see what you see when you think about the game.

Creating Things That Go Bump in the Night

What would a game be without something or someone to blow up, shoot at, or otherwise have war with. Now that you have made the game environment compelling to play, you need to focus on making it exciting. The key to this is not to limit your creativity to the description of the rooms but to allow this creativity to overflow in the design and development of your monsters, ghosts, and whatever other villains run rampant in your game.

It should be noted that the adversaries in your game do not have to be strange creatures from another world. In fact, your game does not even have to be violent. You can create a game based on the old *Clue*™ board game where there are a number of people in the house and a murder, or perhaps some other crime or mystery, has occurred, and the job of the player is to explore the house finding clues until the player is able to solve the crime.

In the game that you are creating, all of the protagonists are ghosts, goblins, Frankenstein, or some similar otherworldly character. Most players have probably watched a *Dracula* movie or some other film of the horror genre. When you mention a monster, their minds probably race back to the picture of that monster that they saw in those movies. You can use this to your advantage. You do not necessarily have to go into as much detail to describe a werewolf as you would if you were describing one to someone who had never seen a werewolf movie. All that you have to do is give enough details for the player to recall the details they remember from the last werewolf film that they watched. In this way, you can focus more on what the werewolf is doing as opposed to what it looks like. Your description of the werewolf itself could focus on specific images that will stick in the player's mind, such as an arm hanging out of its jaws.

To add a bit of variety to your game, you should create at least five different monsters:

1. A ghost
2. A walking skeleton
3. Frankenstein
4. A werewolf
5. A strange three-headed fire-breathing creature who is the leader of these monsters

You have a number of choices for implementing your monsters as well as the way you choose to represent them in your game world. Generally speaking, you will have to give the creature a name, you must create a value that represents how strong the monster is,

and you must assign a value that describes how ferocious the creature is. While the first two values may seem obvious, the last one may need more explaining. See Table 10.1.

Table 10.1 Variables You Must Track for Each Monster

Monster name	Strength	Ferocity
A ghost	100	30
A walking skeleton	100	40
Frankenstein	110	50
A werewolf	110	60
Three-headed monsters	150	100

You need to give the monster a name so that you are able to identify it and so that the player knows which monster he is facing. This is important to help him plan his strategy. If he is fighting a werewolf, he is going to have to use a silver bullet. On the other hand, if he is fighting a ghost, the bullet will be useless and he needs to use the holy water. When he is facing the last boss, he may need to use all of his weapons to defeat him.

The strength of the monster tells us how much life energy the creature has left. The stronger the monster, the more you have to hurt it to destroy it. Some monsters are stronger than others, and this helps to add to strategy. If your energy is low, you may take a chance fighting a ghost, but you may choose to run from a werewolf, which would be much harder to kill. You also have a strength meter, which will be taken into consideration in the battle. If you are very strong, the blows you inflict with your weapon will be more powerful and inflict more damage than they will if you are weak. Likewise, a monster with little strength will inflict less injury to you than a beast that is fully charged and ready to fight.

The third element used to describe your monsters is the ferocity of the creature. This is a meter of how aggressive each creature is. As an example, the ferocity meter of a sheep might be a 1 or a 0, while the ferocity of a lion would be 100. This number will be used to establish how likely a creature is to attack you. The lower the ferocity rating, the more likely you will be able to run away from it. If you try to run away from a more ferocious creature, it will charge and attack you before you reach the door. This factor will also influence how likely a creature is to win a fight. The more ferocious the creature, the more aggressively it will fight and the more damage it will inflict.

It should be noted here that your player's character also has variables that are used to describe it. First of all, there is the player's name. You could just give your player a name

and leave it at that. This, however, lessens the player's feeling of being a part of the game. You want to totally engulf the player in the game. An important element of this kind of game design is to have the player enter his name and use his own name during the actual game play. If you have to tell the player that he has lost the game, you can say, "Sorry, Tom, but you just lost the game," as opposed to just saying "game over." You can also have the characters of the game interact with the player using his first name; "Come on, Tom, surely you must know that I am smarter than that."

I mentioned that there is a strength variable that can be used to keep track of how much health energy you have left. I also mentioned that there is a ferocity meter to tell us how aggressive a monster will be. The equivalent of a ferocity meter for your character is the fear meter. Like its counterpart, this variable will help to determine how likely it is for the player to win a fight with a monster. If a player's fear meter is zero, he is very courageous and will fight more aggressively. If, on the other hand, his fear meter is 100, he is scared out of his wits.

This can affect game play in a number of ways. For example, if a player is afraid, then when a command is given to attack, he may drop his weapon or be much more likely to miss his target. If the command is given to run, he may fall down. The initial value for the fear variable can be set in a number of ways. It could be set according to the difficulty setting of the game. If the game is at the easiest level, the fear variable could be set to zero, and if the difficulty is set to the hardest, the fear variable may be set to 100.

Another option would be to automatically set the fear factor to zero and increase it depending on the events of the game. The fear factor could change according to the type of creature the player is facing. Perhaps the stronger the ferocity factor of the monster the player is fighting, the higher the player's fear factor will be. Also the fear factor can be set to zero and increase every time the player does something that could be considered cowardly such as running away from a fight.

Getting back to the villains in your game, they all range in ferocity from about 30 to 100. They are also listed in order from least ferocious to most ferocious.

1. The ghost (ferocity level 30)
2. A walking skeleton (ferocity level 40)
3. Frankenstein (ferocity level 50)
4. A werewolf (ferocity level 60)
5. Strange three-headed fire-breathing creature who is the leader of these monsters (ferocity level 100)

So you have your hero and you have your hero's foes and a great story that gives them all the reason in the world to fight. Now all that you need to do is give them the tools that

they need in order to do so. You have to figure out how you are going to take that action-filled fight scene that is going on in your head and put it on the computer screen. The first thing that you have to remember is that you are creating a turn-based game, so there is not going to be a whole lot of button mashing going on. The player is going to enter a room and come face to face with a vile villain. He will be asked that all encompassing question:

"What do you want to do?"

The player can choose to run, to defend, or to attack. The attack can be of a magical nature, such as using an amulet or casting a spell, or the attack can be of a physical nature, such as swinging a sword or even fighting barehanded if need be. After the player makes his move, the computer takes over and decides what happens next. You have quite a few options for allowing the computer to determine the result of the battle. The easiest method, of course, would be to have the computer just randomly decide whether you or the monster wins. While this method is easy to implement, it is not very realistic and makes the game more a game of chance than one that rewards players for their skills.

The next step up is to take into account the nature of the hero and his current state of mind as well as the nature and demeanor of the monster. For example, if a player's fear level is very high, he may be twice as likely to lose the fight. If his strength is low, he may be four times more likely to lose. If he is weak and the monster he is fighting is strong, he may be 20 times more likely to lose, and so on.

With this system, it is in the player's best interest to play smart and stay healthy as long as he can because if he doesn't he will never beat the harder bosses later on. This adds to the overall strategy of the game. There are other elements that can add to this strategy. For example, if a player is holding a special amulet, it may give him courage and make him much more likely to win. This encourages exploring and rewards the players who take the time to look under every last rock in your game.

This is a very important point; the player needs to feel like he has gained something when he survives and explores and spends countless hours playing your game. Furthermore, he needs to always feel that there is something else he needs to achieve and he absolutely has to explore at least 10 more rooms before he goes to bed.

Another creative twist that you can add to your game to make it a bit more challenging is to have weapons that only work on certain creatures. So, for example, you would not be able to use a sword or an axe to attack a ghost. In order to defeat any apparitions that may appear, the player may have to use holy water, a magic spell, or possess a certain amulet. You can tie this technique in with your fear system. If a player uses the wrong weapon against a creature you can cause the player's character to panic and drastically increase his fear factor.

You have a multitude of choices. Which ones will you take? It would not be practical for us to cover every possible combination of fight system elements in this book. What you can do, however, is choose a combination here and add it to your game. You can experiment later and try mixing and matching these elements to see which works best for your game. Don't forget to upload your creations to www.ristudios.net for the whole world to see.

In order to get your fighting system to work, you need to give a weight to all of the factors that will affect the outcome of the fight, namely your player's fear, the enemy's ferocity, your player's strength, the monster's strength, and whether or not the player is holding a magic amulet. See Figure 10.2.

You have two variables; one represents the fear rating of the battle while the other is the randomly generated number that decides the outcome of the battle. Let's say that you use the command `RND(100)` to produce the randomly generated result of your battle. If you call this variable `OC` (for outcome), then when you are done, this variable will have a random number between 0 and 100. If you wanted to write the simplest code possible, you could just say that if the number is over 50, the enemy will win, and if the number is less than 50, the player wins. You do not want to do this however, so in addition to the `OC` variable, you will create a variable called `FF` (for fear factor).

If `OC` is less than `FF`, the player will win, while if `OC` is greater than `FF`, the enemy will win. You can set the value of `FF` to 50. This would mean that 50 percent of the time the player will win and 50 percent of the time the enemy will win. Now you have to allow the four factors that affect the outcome of the battle by changing the value of `FF`.

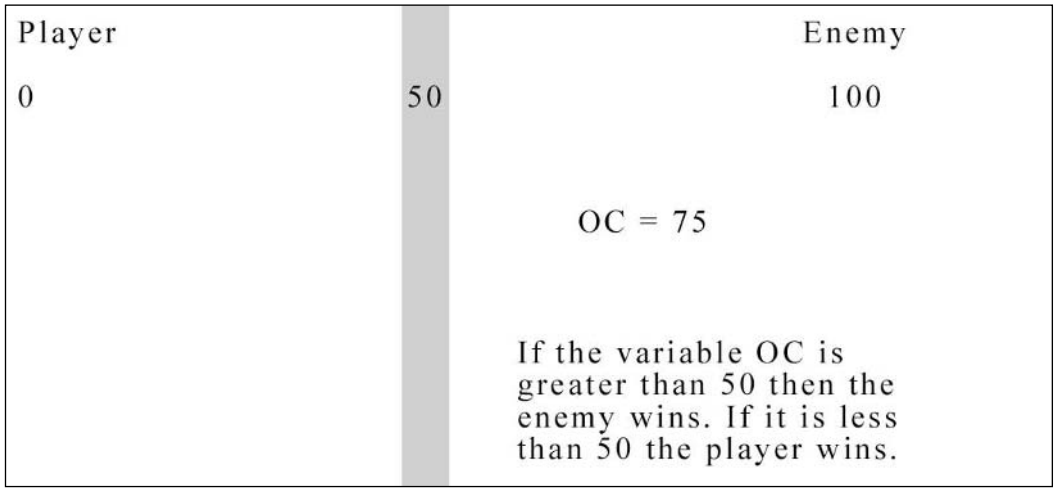


Figure 10.2 Weights of the elements of the fear factor.

First you subtract the fear of the player from FF. See Figure 10.3.

This will decrease the value of FF, making it more likely that the value of OC will be greater than FF and hence more likely that the enemy will win.

Next you add the player's strength to this variable.

This will increase the value of FF and make it more likely that the player will win.

Next you subtract the creature's ferocity from FF as well as the enemy's strength, which will increase the odds of the enemy winning.

You now have the theory in place for a working fighting system. All that you have to do now is determine whether you want to apply this on a micro or a macro level. What I mean is that you can use this system to either play out the entire fight or on a turn-by-turn basis. If you allowed this to determine the results of the whole battle, the player would decide to either attack or defend and, depending on the results of your calculations, the computer can go into auto mode where it narrates a battle sequence where the player

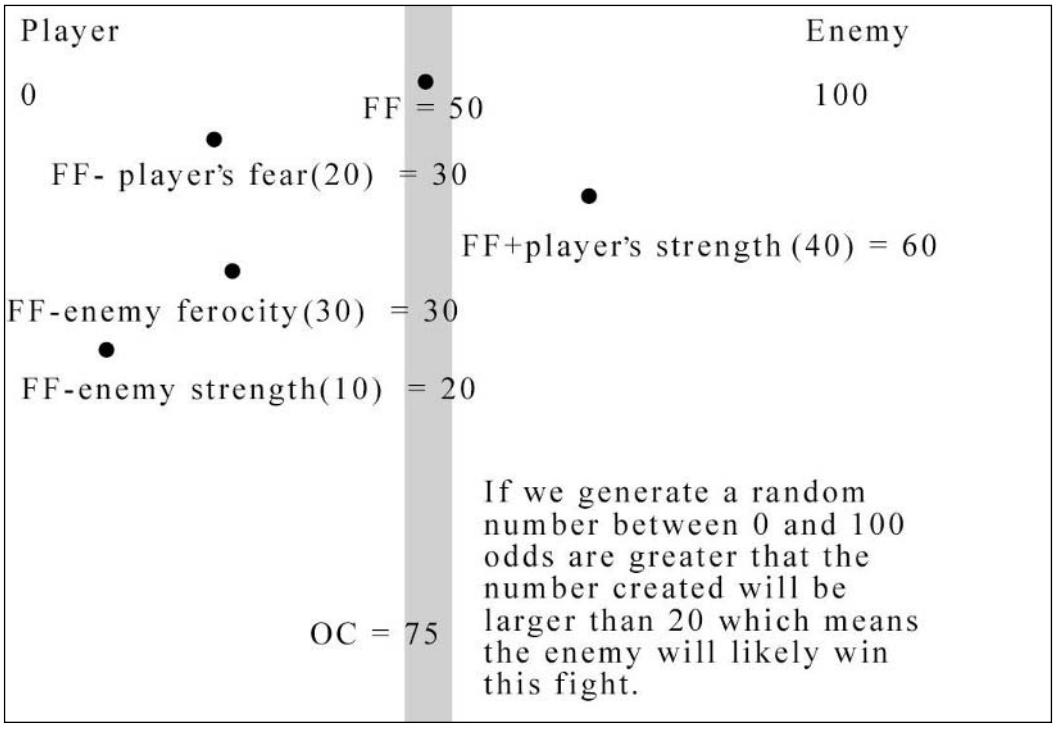


Figure 10.3 Subtract the fear of the player from FF. Then add the player's strength to the FF variable. Finally we subtract the creature's ferocity from FF as well as the enemy's strength, which will increase the odds of the enemy winning.

either wins or loses. On a turn-by-turn basis, you can use this system to effect how powerful each attack is in a given round. So you will make an attack or a defensive movement and the enemy will do the same. Your battle system will decide who will be hurt, the player or his enemy. Whoever gets hurt in this round will be weaker in the next round until someone loses completely.

Personally, I do not like it when the computer goes off and completes a whole battle. I like to feel as though I am a part of what is happening. Not everyone shares my preference and it is for this reason that whenever possible you should build your game in a way that gives the player the option of how he wants his battles to be played out. (You should give the player other options as well, such as difficulty settings and the ability to set the initial value of his fear factor and strength, whenever possible.)

Tools of the Trade

There is usually more than just running and gunning in any good adventure game. Text-based games are no different. A great deal of the fun of these games is the exploration. Exploration is not fun if you do not find something useful. A thing is not useful if you cannot use it in some meaningful way. What this means is that you not only have to think of things the player can add to his inventory but also you have to understand exactly how to incorporate these elements into game play.

What's the point in the player having a torch if all of the rooms in the game are lit? To further integrate the torch into the game, you can make some or all of the rooms in your game dark. This way, if the player has no torch, he can see nothing. On the other hand, with a torch he can see the room clearly. To further add to the challenge of the game and further encourage the player to explore, you can set a time limit for how long a torch can last. This way the player cannot simply find one torch and use it for the whole game; he has to constantly explore the game world to find more. Another level of detail can be added by having the torch fade over time. When it is fully lit the player can "see" everything in the room. When the torch is half lit, the player can see half of the items in the room, and so on.

An important note here is that you have to handle time differently in a turn-based game as opposed to a real-time game. In a real-time game, if you want a torch to last for 10 minutes, you simply set a timer for 10 minutes, and when the timer reaches 0, you put the torch out. You cannot do this in a turn-based game. If you have not put a timer on how long the player has to make a move, then he could conceivably wait for a minute, an hour, or two years before he makes a move.

In turn-based games you have to measure time in turns. Each time the player or an enemy makes a move, you assume that a certain amount of time has passed. You can have a torch that is good for 10 turns, for example, and after 10 turns, the torch will go out. The

amount of "time" that passes before the torch goes out can be set by the difficulty setting, or you can let the player choose this value when the game starts.

There is a very thin line here. There is a difference between rewarding the player and irritating him. If a player can just play your game without much exploring and still win, that is okay. It makes your game accessible to more players. On the other hand, if a player who does explore finds tools that make playing the game more fun or just easier, that is a reward for his efforts. If, on the other hand, there are numerous elements that a player must have in order to play the game and this results in a lot of backtracking, then you have a problem. There are few things that a player hates more than backtracking, especially when it is a regular part of the game. Be aware of this phenomenon. Make your games challenging but not irritating.

But what in the world are you going to litter about the game world for the player to find? There are some regular staples that you almost cannot do without:

1. Health
2. Magic
3. Maps
4. Talismans
5. Treasure

There are also a number of proven methods for placing these items throughout your world and having the player find them. The absolute easiest method is to randomly fill the rooms with a single item. When the player enters the room, he automatically acquires that item. This solution is easy to program but not very realistic. When you walk into a room, items do not automatically jump into your pocket. You have to decide which item in the room you want to pick up and make a conscious effort to pick it up and put in your pocket.

This adds a new level of strategy to the game. Your player will only be able to carry a limited number of items and so he will have to choose wisely when deciding whether or not to pick up each item. For example, if a player can only hold one more item and he has no torches and must pass through a dark room, it would not be wise for him to pick up a magical item. Instead, he may explore to find a torch before moving on and make a note of the room so that he can come back to that room later if he needs some magic.

The next level of complexity for your game is to have items in an enemy's possession instead of just lying around. This way, the player has to actually defeat his opponents and then the item is his reward. This also forces the player to fight through the game instead of being able to always run away from a fight.

Creating a Language for Your Game

Could you imagine yourself going through life without knowing any specific language? How could you communicate with others? How could you let your ideas be known?

Not a very pleasant thought. It's just as unpleasant a thought to plop the player in the middle of your game world with absolutely no form of communication with the world around him.

Communication can occur on a number of different levels. In most text-based games, such as the one you will build shortly, the computer will communicate with the player to find out what he wants the character to do. In more advanced games, you may have other friendly characters who the player must talk to in order to find out information or simply to give them orders to do something.

The foundation of communication is language. The computer does not understand English, and you cannot expect the player to learn the BASIC programming language in order to play your game. You have to create your own language that falls somewhere in between.

Your language should be easy to learn and easier to remember. You do not want the player struggling to remember the command he needs in order to use his magic amulet when a 12-foot tall werewolf is about to bite his head off! Your language also should not be overly comprehensive; the player will not be describing any sunsets, or writing epic poems in your game (unless, of course, that is the point of your game). All the player needs to do is move around the game world, pick up items, use those items, and fight so you only need to create a language that allows the player to carry out these actions. Adding any more elements to your language will constitute wasted energy. You also want to be sure that the player does not have to enter the full word of the command that he wants to execute. This way, when the heat is on, the player can simply enter F instead of the full word, FIGHT, in order to enter battle with an enemy. That said, let us take a look at your new programming language.

```
N-NORTH
S-SOUTH
E-EAST
W-WEST
U-UP
D-DOWN
P-PICKUP
M-MAGIC
F-FIGHT
R-RUN
```

If the player enters one of the directional commands (north, south, east, west), the game will check to see if there is a door in that part of the room, and if there is, the player will go through that door.

Up and down work in basically the same way. If there is a stairway or a ladder or an elevator in the room, the player will climb either up or down to another floor.

If the player enters a room and there is treasure or other useful items lying around, the player will use the pickup command to pick it up.

The magic command will be used to cast spells that can either aid the player's health or help to defeat an enemy.

Finally you come to quite possibly the two most important words in the game. Fight and run. These are the two options that the player will have whenever he encounters an enemy. As you will see later, in order to discourage the player from cowardly behavior, the run command will not always work. Eighty percent of the time when the player runs, the command will not work and the player will be forced to fight. The fear factor for the player will be adversely affected each time he chooses to run. If the run command does work, the player will exit the room through the same door he entered.

The key to making your game as interesting as possible is to incorporate as much strategy as you can. The more elements that the player has to manage, the more satisfying game play is going to be. When the player enters the fight command, a menu will appear asking the player which weapon he wants to attack with. This menu will list the weapons that the player is currently carrying. When the player selects an item, that weapon will be used to attack the enemy.

Likewise, when a player uses the magic command, he will also be presented with a menu where he can choose which spell he would like to use. Options would normally include teleportation, restoration of health, and various attacks on the enemy.

Writing the Code for Your Game

Now that you have the theory of how a text-based game works, it is time for us to get down to business and start writing some code. It should be noted that the choice of commands used in the program you are about to write will work on all of the retro game programming machines covered in this book.

Mapping Out Your Program

Many great sculptors will tell you that when they first start working on a project they have no idea what it will look like when they are done. They will tell you that the sculpture is

already there in the raw material and it is up to them to find it. They start with the general shape of what they want and then they chip away until a masterpiece is created. A similar process takes place as you write your video games. You are looking at a blank screen with nothing but a prompt from your editor, compiler, or interpreter. You know the general idea of the game you want to build. Now all that's left to do is create your generic shape and then chip away until you have a masterpiece.

The General Shape of Your Game

You already know that a number of things must happen for your game to work; you have to initialize all of the variables and clear the screen; you have to describe the room the player is in, including any monsters; you must give the player a chance to react to the environment; you must process what the player has done; and, finally, you must repeat this procedure continually until the player is either dead or has won the game.

Although you have a vague idea of how to write this program, you will not get a good idea of how to implement the specific game features until you take these general ideas and create a skeleton on which you can build your game.

```
B    1000 : REM INITIALIZE GAME
     GOSUB 3000 : REM
```

This code will not run.

The following sections explain how to use those two lines of nonworking code to expand upon and create your game. Notice the colon in the middle of each line. That's an important factor in the BASIC language, which will also be covered in the following sections.

Mapping Out Your World

What you need is a simple method of storing your game map in the computer's memory. As it turns out, this is not very difficult to do. There is a handful of information that you need to store in order to represent each room of your game. All that you need to do is create a few arrays that you can use to store this list of items. The word *array* is new, but like most topics covered in this book, it is not very complicated. Look at this line of code:

```
A = 5
```

You have seen this before. Basically you created a variable called `A` and stored the number 5 in it. Now look at this.

```
Dim A(4)
```

This is new; you have added two brackets with the number 4 in the middle. This rather innocent looking change is actually a very powerful new twist on the way you store your

information. Previously, if you wanted to store the numbers 1, 2, 3, and 4, you would have had to create four variables:

```
A = 1
B = 2
C = 3
D = 4
```

Using an array gives us a much simpler method of storing lists of information.

```
A(1) = 1
A(2) = 2
A(3) = 3
A(4) = 4
```

What you just did was to make use of the `A(4)` variable that you created earlier. When you executed the `Dim A(4)` command, you made a new variable that was exactly like all of the other variables that you have made, with one exception. This variable can hold more than one value. The number found in brackets determines how many separate values the variable can hold. In this case, the variable was set up to hold four elements. Each value stored in the variable is given a number. If you want to read what that value is or store a new value there, you have to write the variable name followed by an open bracket, followed by the number representing the value that you want to work with, followed by a closed bracket. This way, a single variable called `A(4)` can act as four separate variables called `A(1)`, `A(2)`, `A(3)`, and `A(4)`. See Figure 10.4.

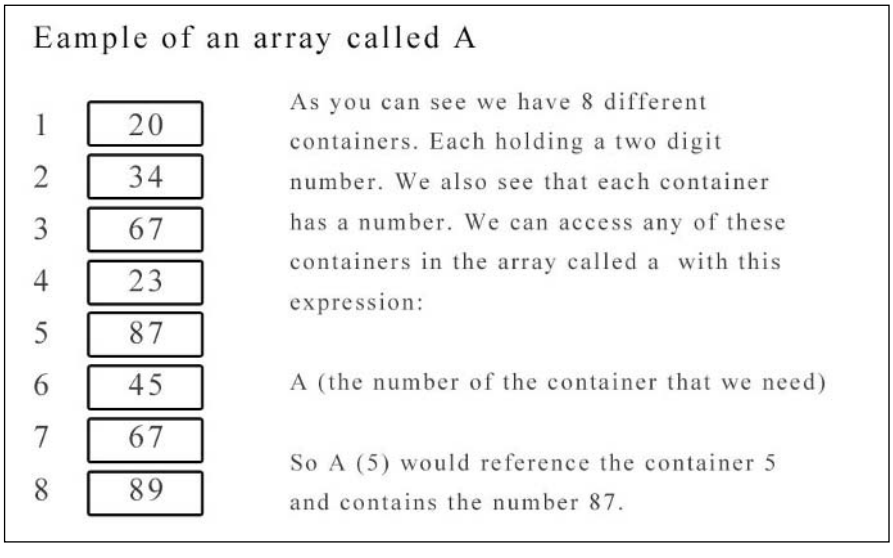


Figure 10.4 You can think of an array as a container that can hold data in a number of separate compartments.

Using arrays makes it easier for you to work with lists of information, saving time and lines of code. As an example, let's say you need to check to see if any of five different variables has a value greater than 5. If so, you need to jump to line 250 of your program. Here is how you would do this without a variable array.

```
10 IF A > 5 THEN GOTO 250
20 IF B > 5 THEN GOTO 250
30 IF C > 5 THEN GOTO 250
40 IF D > 5 THEN GOTO 250
50 IF E > 5 THEN GOTO 250
```

It just took five lines of code to do what you had to do. On the other hand, look at the same code written using an array.

```
10 FOR I = 1 TO 5
20 IF A(I) > 5 THEN GOTO 250
30 NEXT I
```

You just did the same thing as above using only three lines of code!

It gets better.

The array that you just used is called a *single dimension array* because it uses only one number to determine exactly which of the values contained in an array you would like to reference. When you use two or more numbers to select items in the array, things get a lot more interesting. For the purpose of your game, you will be using a number of two-dimensional arrays. It will be much easier to understand this concept when you are able to see the practical use of a two-dimensional array. See Figure 10.5.

The house that your game takes place in has eight rooms as well as a ninth "room," which is used to represent outdoors. Each room (except, of course, the ninth room representing outside) has four walls. Each wall has the potential to hold one door. When the player wants to move in any of the four possible directions, the computer has to know if there is a door or some other similar facility in that direction, which the player can use to exit the room and, if so, which room that door leads to.

Figure 10.6 illustrates the concept of a two-dimensional array. The array is declared like this:

```
DIM A(8,4)
```

This array holds 32 separate values (8×4).

These values are divided into eight rows of four columns.

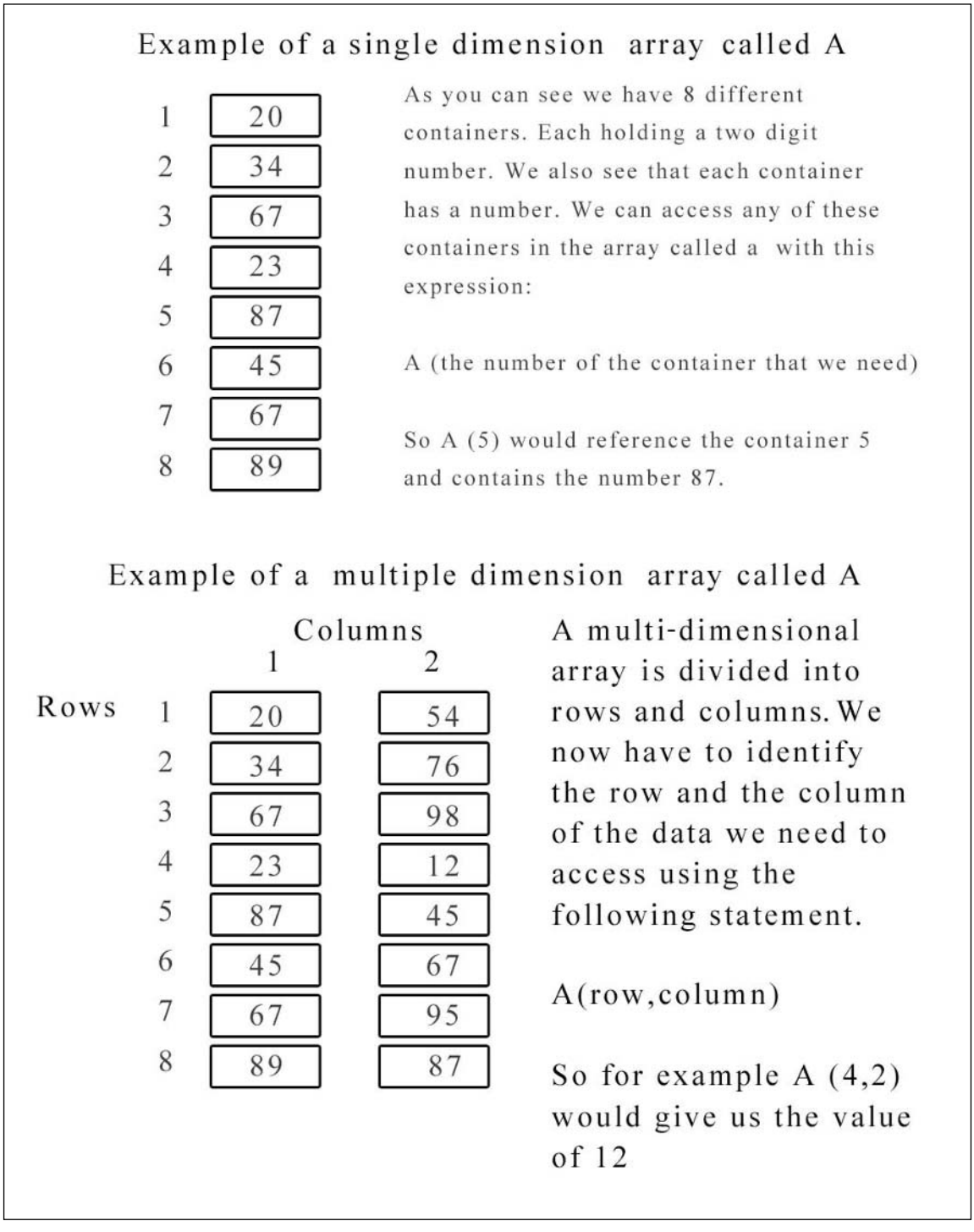


Figure 10.5 Single and multiple dimension arrays work differently.

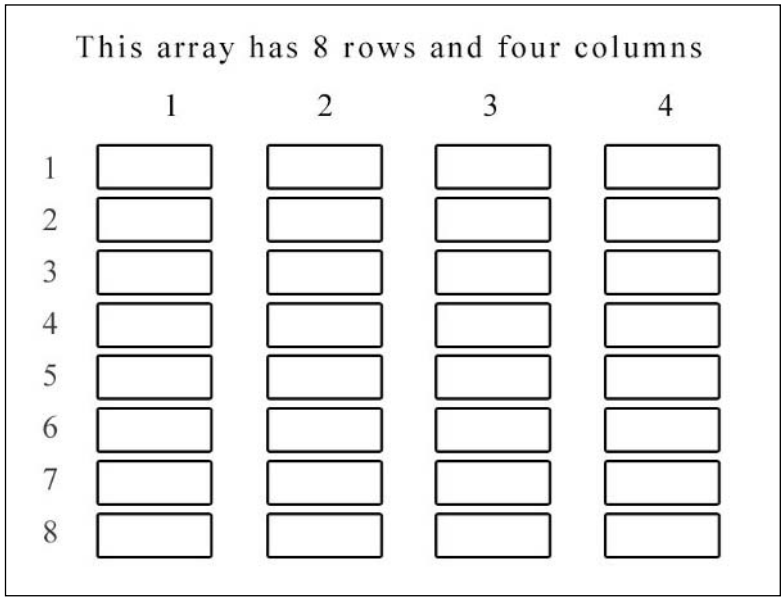


Figure 10.6 Illustration of eight groups of four values in your array.

Each group represents a room, and each value for that room represents a wall. The letters n, s, e, and w stand for north, south, east, and west. See Figure 10.7. Under each of these letters, you will find a number between 0 and 9.

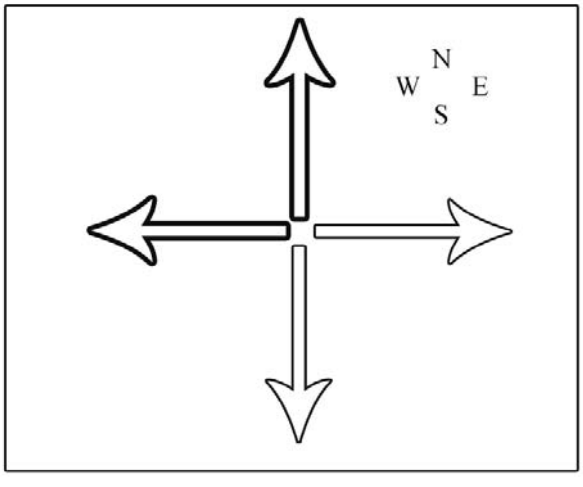


Figure 10.7 The player can move in 4 directions.

When the player is standing in a room, he can move either north, south, east, or west.

If you assume that the player is standing in room one, then he is standing in the entertainment room. He can move north into the dining room, south to go outside, or east to go to the kitchen. The player cannot move to the west because there is no door. You can easily create a short program to store this information in the computer.

```
10 Dim A(8,4)
20 A(1,1) = 3
30 A(1,2) = 9
40 A(1,3) = 2
50 A(1,4) = 0
```

In line 10, this short program creates a two-dimensional array to represent the house. In line 20, you reference room 1 and wall 1 (a room number, wall number), which represents the north wall. You store the number 3 because the door on the north wall of this room leads to room 3, which is the dining room. Next, in line 30 you reference room 1, wall 2, which is the south wall. You store the number 9 here because the south wall leads to room 9, which in your game represents outside. In line 40, you reference room 1, wall 3, which is the east wall, and you store a value of 2 because this door leads to the kitchen. Finally, line 50 references room 1, wall 4, which is the west wall. Because there is no door you store a value of 0 here.

Each room in the game can be represented in this manner.

You now have a way to store the map of your world in the computer, but you still have one problem. The method that you just used was very awkward and cumbersome. There's got to be a better way to do this. Take a look at the following code:

```
10  DIM A(9,4)
20  FOR B = 1 TO 9
30  FOR C = 1 TO 4
40  READ A(B,C)
50  NEXT C
60  NEXT B
70  DATA  3,9,2,0
80  DATA  0,0,0,1
90  DATA  0,1,4,0
100 DATA  0,5,6,3
110 DATA  4,0,0,0
120 DATA  0,7,0,4
130 DATA  6,0,8,0
140 DATA  0,0,0,7
150 DATA  1,0,0,0
```

This code is a lot neater. Look at it for a minute and see if anything seems familiar. The data statements found in lines 70 through 150 are an exact replica of the table you used to store the map of your house! This not only makes it a lot easier to enter your game's map into the computer, it also makes it easier for you to see if you have made a mistake.

note

Data statements use two commands: `READ` and `DATA`. These commands are very useful for storing and retrieving data.

Line 70 shows how the `DATA` command is used. Following each command is a list of numbers separated by commas. Each number represents a piece of data that you need for your game.

```
70  DATA  3,9,2,0
80  DATA  0,0,0,1
90  DATA  0,1,4,0
100 DATA  0,5,6,3
110 DATA  4,0,0,0
120 DATA  0,7,0,4
130 DATA  6,0,8,0
140 DATA  0,0,0,7
150 DATA  1,0,0,0
```

Every time you execute the `READ` command, a number will be pulled from this list of numbers stored in a variable that you specify. The `READ` command is used like this:

```
READ variable
```

So in line 40, you are able to use the `READ` command to fill your two-dimensional array.

```
40  READ A(B,C)
```

Lines 10 through 60 should look slightly familiar. You have seen all of these commands before but never in this particular configuration. What you are looking at is called a *nested loop*. This is another way of saying that you are looking at a loop within a loop.

Let's say that you want to put doors on all of the walls in room 2 of your house. You can use a simple `for next` loop to accomplish your goal.

```
10 FOR I = 1 TO 4
20 A(2,I) = 18
30 NEXT I
```

In this example, all of the walls in room 2 would lead to room 18, as would be the case if room 2 was a smaller room located at the center of room 18. See Figure 10.8.

Let's say that you had eight rooms, each having two doors, one in the north and one in the south. All of these doors lead to room 18, as would be the case if you had a row of eight stalls located at the center of a larger room numbered 18. See Figure 10.9.

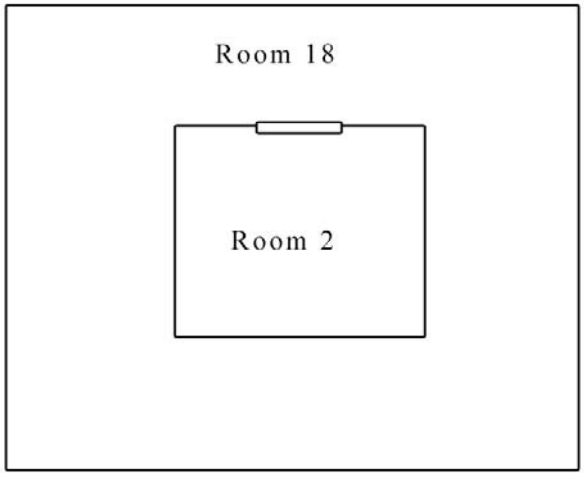


Figure 10.8 Illustration of a small room 2 at the center of room 18.

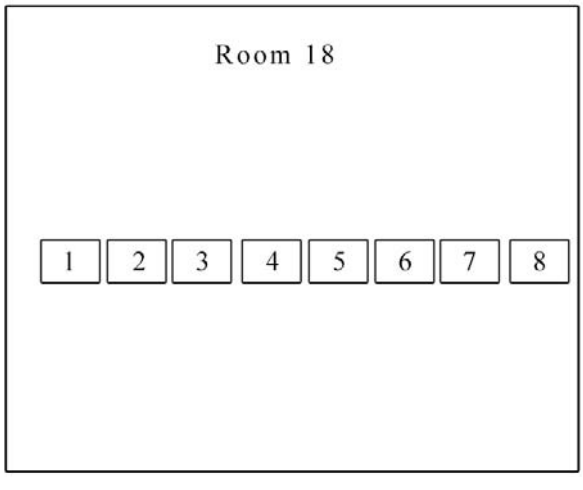


Figure 10.9 Illustration of eight small rooms at the center of room 18.

This calls for a slightly more complex technique. That technique is called a nested loop. Basically, what you want to do with your nested loop in this situation is:

1. To start in room 1, create two doors each leading to room 18.
2. Move on to room 2 and make two doors that each lead to room 18.
3. This continues until you have created all of the doors for all eight of your rooms.

Laid out as a flow diagram, this process would look the diagram you see in Figure 10.10.

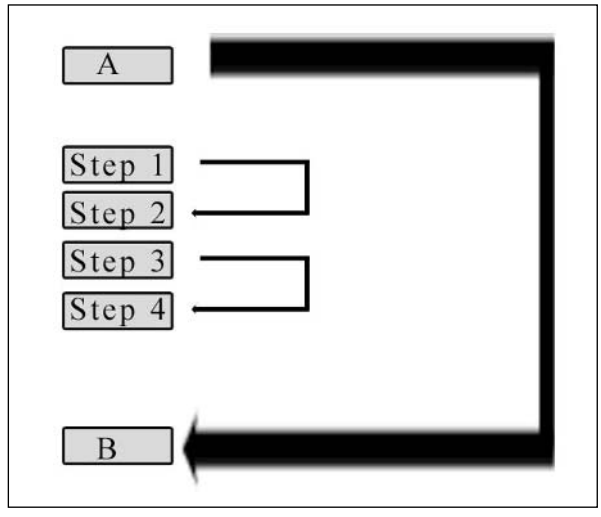


Figure 10.10 Illustration of your nested loop shown as a flow diagram.

You know what will happen if you create the following code:

```
10  FOR I = 1 TO 8
…
…
…
50 NEXT I
```

Whatever code is located between lines 10 and 50 is going to be executed eight times. What if you put this code between lines 10 and 50?

```
10 FOR I  = 1 TO 8
20 FOR Y = 1 TO 2
30 A(I,Y) = 18
40 NEXT Y
50 NEXT I
```

The `for` loop found in lines 20 to 40 will be repeated eight times.

With a nested loop, each time your Y loop is executed, the value for `I` will be increased by 1, so rooms 1 to 8 will be fitted with two doors both leading to room 18.

Now let's take another look at your nested loop in your program and see what it does.

```
10  DIM A(9,4)
20  FOR B = 1 TO 9
30  FOR C = 1 TO 4
40  READ A(B,C)
```

```
50  NEXT C
60  NEXT B
70  DATA  3,9,2,0
80  DATA  0,0,0,1
90  DATA  0,1,4,0
100 DATA  0,5,6,3
110 DATA  4,0,0,0
120 DATA  0,7,0,4
130 DATA  6,0,8,0
140 DATA  0,0,0,7
150 DATA  1,0,0,0
```

Lines 70 through 150 place 32 numbers into memory in a straight line. See Figure 10.11.

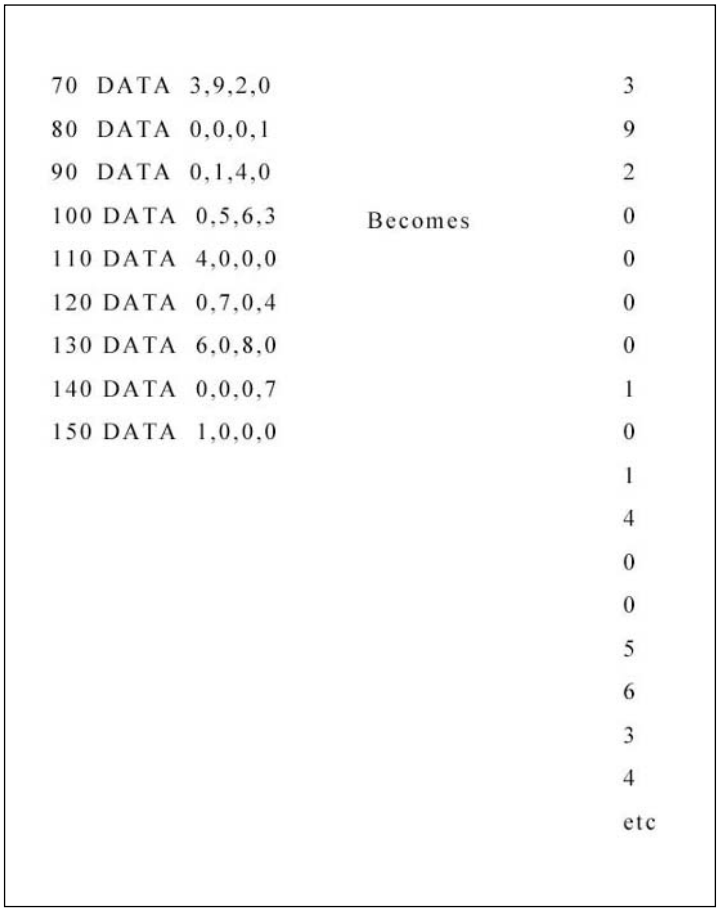


Figure 10.11 Illustration of the numbers in memory.

The `READ` command takes one parameter.

```
READ variable
```

When you execute the `READ` command, the first number in your list is moved into whatever variable you indicated. If you execute the `READ` command four times, then the first four numbers in your list will be moved from your list and into variables.

Looking at the data lines, you see that you have eight lines containing four numbers each. Each line represents a room, and each of the numbers represents a wall in that room. What you want to do is read the first four numbers and store them into the portion of your array that represents room 1. Next, you want to read the following four numbers and save them in the array as another room. You need to repeat this loop until each room is assigned a wall. That job is taken care of by your nested loop.

Jumping from Text-Based Games to Graphics-Based Games

The key to creating any graphics-based game is to understand the game elements that we have been discussing throughout this book. You now know how to clear the computer's screen, draw sprites, and add game logic. All that you have to do now is apply them in the same kind of pattern you used to create the text-based games, mixed with a little imagination to make a great game.

Conclusion

You have now come to the end of this book but not the end of your journey. You now understand all of the features and principles needed to build retro games. The next step is for you to log on to www.ristudios.net and visit the code and tutorial repository there. You will find basic and advanced source code and tutorials for each of the machines covered in this book. Also be sure to visit the Web site for this book at www.retrogame-programmingunleashed.com for all the code in the book, links to other great retro programming sites, and much more.

Before you read this book, the data on this Web site would have been useless to you. Now that you have mastered the principles of retro game programming, however, you can not only understand the code in this repository, but you can use it as the basis for your own games. You are now ready to push these machines to their limits.

Have fun.

Live long and prosper.

You are now a part of the official worldwide retro game programming community, and there is no limit to what you can do. Be sure to check out André LeMothe's X Game Station at www.xgamestation.com. There you will not only learn to build your own game, but you will also be able to learn to build your own video game system!

INDEX

Symbols and Numerics

A

B

D

E

F

G

H

I

M

N

S

T

U

V

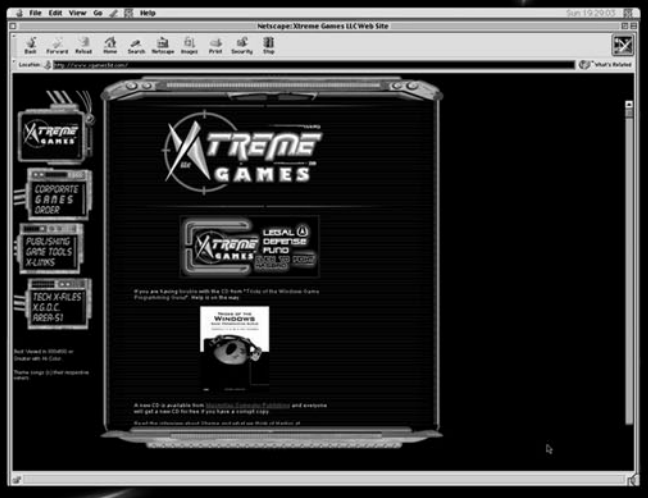
W

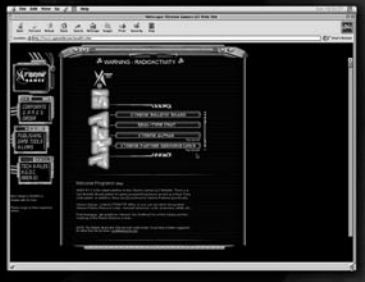
X

Y

Z

TAKE YOUR GAME TO THE XTREME!





Xtreme Games LLC was founded to help small game developers around the world create and publish their games on the commercial market. Xtreme Games helps younger developers break into the field of game programming by insulating them from complex legal and business issues. Xtreme Games has hundreds of developers around the world. If you're interested in becoming one of them, visit us at **www.xgamestation.com.**

www.xgamestation.com

XTREME GAMES LLC