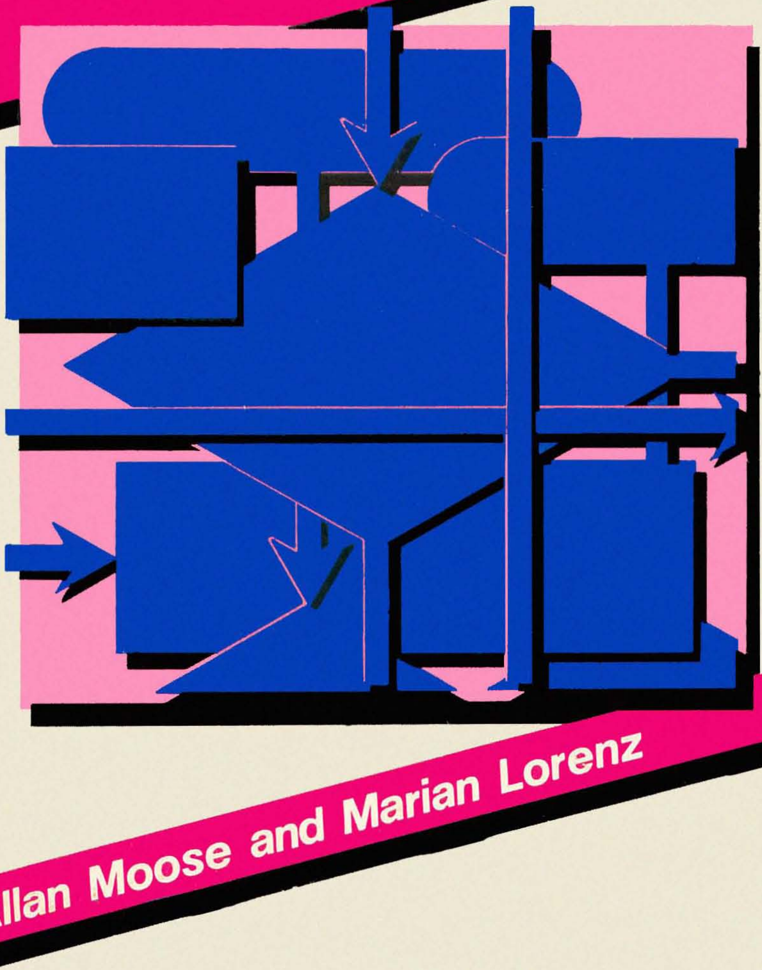
WEBER SOFTWARE SERIES

ATARI

Assembly Language Programmer's Guide



Allan Moose and Marian Lorenz

Atari Assembly Language Programmer's Guide

by

Allan E. Moose

Marian J. Lorenz

Weber Systems, Inc.
Chesterland, Ohio

Published by:
Weber Systems, Inc.
8437 Mayfield Road
Chesterland, Ohio 44026
(216)729-2858

For information on translations and book distributors outside of the United States, please contact WSI at the above address.

Atari Assembly Language Programmer's Guide

Contents

List of Boxes

List of Figures

List of Tables

Preface

Atari Assembly Language Programmer's Guide is written for the person who has had little or no experience with 6502 machine language (machine language and assembly language are used interchangeably) but who would like to incorporate machine language subroutines into BASIC programs as well as for the intermediate programmer. The preparation and organization of the book assumes that the reader has a working knowledge of BASIC as the book is designed to build upon that foundation. By a working knowledge we mean that the reader is familiar with FOR-NEXT Loops, PEEKS and POKES, setting up strings, etc.

As you leaf through the book, you will notice that there is a good deal of material included besides 6502 assembly language. The reason is that we want the book to be useful for readers who want to tap the unique graphics and sound capabilities of Atari Home Computers. Since many of these features are available only through machine language, we thought it appropriate to devote two chapters to graphics and sound. These chapters serve to give you the background necessary to put your new machine language programming abilities to maximum use.

The concepts behind much of the material presented in these two chapters is scattered throughout technical material published by Atari, Inc. such as *De Re Atari* and *The Technical Reference Notes* or magazines such as *Compute!*, *Antic*, *Analog*, *Byte*, and the now defunct *Micro*. However, the organization and presentation is our own. Where different sources have been in conflict, we have done our best to sort out the differences and present the most accurate information available.

Wherever appropriate we have suggested short exercises and practice problems to help you develop your skills and understanding. Throughout the book we have illustrated programming instructions and concepts with subroutines that will make something happen on the screen. Many of these subroutines can be used, or modified for use, in your own programs. When you examine these subroutines it may occur to you that they could be improved, or the tast programmed differently. Generally, there is more than one way to program any given job. A good way to learn is to modify someone else's program. So feel free to experiment! Active involvement is the best way to learn.

A final note to people who have some acquaintance with other books on assembly language programming. You will notice that in this book there is a marked lack of hexadecimal numbers. Machine language subroutines that are called from BASIC must be written in decimal numbers. Therefore, we believe that beginners to machine language programming will experience less confusion at the start if they use the more familiar decimal numbers wherever possible, rather than having to continually switch back and forth between hex and decimal. Once you are familiar with the instruction codes and programming techniques you may want to acquire an assembler editor cartridge and use hexadecimal numbers. But then you are no longer the novice or intermediate programmer for whom this book was written.

Introduction

It has become traditional to begin a book or article written for machine language novices with the question: "Why machine language?" The answer to the question is two-fold. First, the traditional answers. Then, more specifically, why Atari owners will find it advantageous to learn machine language programming.

Traditional Answer Number One: SPEED

High level languages such as BASIC, PILOT, and LOGO are relatively easy to learn and are convenient to use for programming. But, for sheer speed nothing surpasses machine language. The reason for this can be readily seen. The 6502 processor used by the Atari operates at a rate of 1.79 million cycles per second. Depending on the command, machine language instructions take between two and seven cycles to execute. A ten step machine language subroutine that averages four cycles per step will take .0000235 seconds to execute. In more concrete terms: A program to redefine the Atari character set and display a picture made from the new characters that takes fourteen seconds to execute in BASIC can be executed in about half a second in machine language. As a general rule, machine language programs run ten or more times faster that similar BASIC programs. Speed of this nature is important in games, programming for real time situations, or in educational programs where the pupil should not have to wait for the computer to respond.

Traditional Answer Number Two: "KNOWLEDGE IS POWER"

This aphorism first appeared in a brief treatise called, The Art of War, written by Sun Tzu around the fourth century B.C. Tzu proposed that the application of the principle is broader than the waging of war. Similarily, while learning to program in machine language you will gain invaluable information about the internal workings of your Atari. This knowledge is absolutely essential to getting the most from your computer. Furthermore, it will enable you to "program smarter" —more efficiently and more creatively.

Traditional Answer Number Three: FLEXIBILITY

If you program solely in a higher level language, such as those mentioned above, you are to a large extent limited to the commands chosen by the language authors. When you program in machine language the potential of what you do is largely determined by your imagination, experience and ability. For example, Atari BASIC has no RENUMBER command, but it is possible to write a machine language routine to renumber for you.

Traditional Answer Number Four: INTERFACING

If you decide you want to become a hardware hacker and use your Atari to control the temperature and humidity of your greenhouse, control household security, or use it for some other "real time" application, chances are that you will need to write a machine language control program.

More importantly an Atari Home Computer owner should learn machine language because *learning 6502 machine language will allow* *you to get the most out of your Atari's sound and graphics capabilities.* The designers of Atari Home Computers built an excellent machine with probably the best sound and graphics capabilities of any microcomputer presently on the market. Because of these unique features, one can justify calling it a second generation microcomputer. Many of its features such as display list interrupts, vertical blank subroutines and dynamic music are not accessible from BASIC. Others, such as, player-missle movement, page flipping, and scrolling, although accessible from BASIC, are most satisfactorily implemented by machine language subroutines.

Through machine language the programmer can access many of the internal registers used in sound and graphics generation. There any many things that this accessibility will allow you to do. With machine language subroutines you can play music while a display is being created, or generate sounds that imitate musical instruments. Machine language is absolutely essential for dynamic sound production. In creating graphics, machine language allows you to change colors on the fly, use different character sets on different parts of the same screen, achieve smooth animation, and smooth scrolling, to mention a few applications.

One of the first problems that confronts a beginner is the profusion of buzzwords, acronyms and unfamiliar number systems. Vectors, pointers, MSB's, LSB's, AUDC1, POKMSK, binary numbers, hex numbers and other arcania slow down the learning process. To ease the way, chapter 1 begins with a discussion of the decimal, binary, and hexadecimal number system. The computer works with binary numbers, experienced assembly language programmers work with hexadecimal numbers, and the rest of the world uses decimal numbers. In writing programs and reading other people's programs, you will need to be adept at converting from one system to another. Chapter 1 will help you hone your skills in this area.

The discussion of the number systems is followed by an overview of the 6502 architecture and its operation. Usually a beginning programmer will not need to worry about counting machine cycles or other such details. However, a general knowledge of the inner workings of the central processor unit (CPU) is an important aid to visualizing what the various machine language instructions do.

There are two ways to learn to program: one can jump in and start programming, learning new instructions along the way, or one can survey the instruction set before beginning to write programs. We have opted for the second choice. Therefore an overview of the 6502 instruction set is presented in chapter 2. Since there are 151 different instruction codes that can be used with the 6502, we feel that it is advantageous to impose structure on this mass of information. Instructions have been grouped according to function because we believe that this will simplify the mastery of the codes and help you organize the parts into a meaningful whole.

Many of the programming examples will make use of the Atari's special graphics capabilities. Thus, you will be able to see immediately what each program accomplishes. Furthermore, you will be able to incorporate the subroutines into your own programs. Chapter 3 is a comprehensive description of Atari graphics including display lists, redefining characters, and player/missile graphics. Chapter 3 is designed to present graphics fundamentals and to be used as a reference aid in your later programming. Additionally, the topics in this chapter are approached in such a way as to introduce some of the concepts underlying machine language. For example, in treating ANTIC as a microprocessor we stress how bit patterns determine the processor's instructions.

Actual programming begins in chapter 4 with short simple examples. Several examples are display list interrupts — an Atari feature accessible only through machine language. Other examples include moving players vertically and redefining character sets. Ready to run programs are provided to illustrate programming concepts and techniques. Here and there short programming exercises are suggested to give you immediate feedback on your understanding of these concepts.

By the end of chapter four you should have a pretty good grasp of the instruction set; you'll be writing short routines and making things happen. In addition, while learning about Atari graphics, you will become familiar with memory organization in the Atari. With this background in hand, you will be prepared to explore some more of the Atari's special capabilities such as vertical blank interrupts, POKEY and sound generation. These topics and their applications are covered in chapters 5 and 6.

As mentioned before, one of our goals has been to write a book that includes useful reference material. For this reason we've included boxes, charts, tables, and appendices that cover all the necessary fundamentals. This material will serve as a handy resource when programming. Two of the appendices deserve special mention. Appendix C is a disassembler written in BASIC. This program allows you to input the decimal numbers representing a machine language routine: it will return the assembly language listing. This will be helpful in taking apart and understanding someone else's machine language routines. Appendix B is an assembler written in BASIC. This program will be useful in writing longer machine language programs.

Programming examples are designed to be directly applicable to your use and are designed to illustrate general principles rather than programming tricks. The programs in this book have been tested and run on Atari 800's, 800XL's and the new 65XE and 130 XE computers. Some of the programs assume that you have 48K of RAM available. They are easy to modify. Ready to use machine language subroutines are included. These routines have been chosen to be useful to those readers who want to work within the Atari off-the-shelf capabilities as opposed to system programming or arithmetic routines. The materials and techniques presented in this book should provide the reader with the necessary background to be comfortable with assembly language and therefore write more sophisticated programs.

In summary, *Atari Assembly Language Programmer's Guide* presents the fundamentals of machine language programming as well as the techniques for establishing and operating machine language programs called from BASIC. It is, therefore, in one sense a textbook and in another sense a practical handbook. Among its purposes is to help the reader enjoy the best of both languages-BASIC and Machine Language by providing the necessary tools.

1

Number Systems and Hardware

Introduction

Computers use binary numbers, professional assembly language programmers use hexadecimal numbers, and the rest of the world uses decimal numbers. When starting to learn assembly language programming it is necessary to begin by learning how to convert from one number system to another. Among the reasons for learning this skill is that many machine language routines you will come across are written using hexadecimal numbers. If you want to call these routines from BASIC it will be necessary to convert the hexadecimal numbers to decimal. The first section of this chapter will discuss the decimal, binary, and hexadecimal number systems and the representation of characters and numbers by number codes.

In many situations assembly language programmers do not require a detailed knowledge of computer hardware in order to write programs that work. However, it is necessary to know some basic facts about the 6502 processor, the ATARI's special sound and graphics chips, and memory organization to aid you in visualizing how to organize and write more effective machine language routines. These topics will be covered in the second section of this chapter.

The Decimal Number System

The primary distinguishing feature of a number system is its radix or base. The base of a number system is equal to the number of digits or characters used. The decimal system, base 10, uses the ten digits: 0,1,2,3,4,5,6,7,8,9. The binary system uses two digits: 0 and 1, while the hexadecimal system uses sixteen characters: the digits 0 through 9 and the letters A,B,C,D,E,F. These number systems are called positional or weighted systems because each digit or character has a value assigned to it according to its position. Consider the decimal number 1729. The weights assigned to the individual digits are successive powers of 10. Table 1-1 illustrates the concept of weighted position:

Table 1-1. Number System Weighted Position

POSITION NAME	THOUSANDS	HUNDREDS	TENS	UNITS
Powers of 10	10^3	10^2	10^1	10^0
Weight	1000	100	10	1

$$1729 = (1 \times 1000) + (7 \times 100) + (2 \times 100) + (9 \times 1)$$

The Binary System

It is apparent from Table 1-1 that in the decimal system counting is done in powers of 10. On the other hand, in the binary system, counting is done in powers of 2. Table 1-2 gives the first sixteen powers of 2:

Table 1-2. Binary system powers of two

Power	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight in decimal	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Binary numbers are expressed as sequences of ones and zeros. For example, let's look at the binary number 11010010. This number can be related to its decimal equivalent as follows:

Power of 2:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight in decimal:	128	64	32	16	8	4	2	1
Binary Number:	1	1	0	1	0	0	1	0

Conversion
to decimal: (1x128)+(1x64)+(0x32)+(1x16)+(0x8)+(0x4)+(1x2)+(0x1) = 210

Each digit in a binary number is called a bit. A group of 8 bits is called a byte. The Atari computer represents numbers and characters in bytes. The individual bits in a byte are sometimes referred to by their position with the letter D and a subscript as in:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	0	1	0	0	1	0

The left most bit of a byte (D_7) is called the Most Significant Bit (MSB) because it has the greatest weight or value. The right most bit (D_0) is the Least Significant Bit (LSB) because it has the smallest weight.

There are several ways to convert binary numbers to decimal. One method is to write down the powers of two as we did for 11010010 and add together the weights wherever a 1 appears. Another scheme is to write down the number and use the recursive rule:

(previous result) x base + (next digit) = result

Examples to Illustrate Recursive Rule

EXAMPLE 1:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	0	1	0	0	1	0

Starting with D_7, there is no previous result so:

[previous result=0]x2 + [next digit(D_7=1)] = 1

[1x2] + [next digit(D_6=1)] = 3

[3x2] + [next digit(D_5=0)] = 6

[6x2] + [next digit(D_4=1)] = 13

[13x2] + [D_3 (=0)] = 26

[26x2] + [D_2 (=0)] = 52

[52x2] + [D_1 (=1)] = 105

[105x2] + 0 = 210 decimal equivalent

Example 2:

128	64	32	16	8	4	2	1
0	1	1	0	1	0	1	0

(0x2) + 0 = 0

(0x2) + 1 = 1

(1x2) + 1 = 3

(3x2) + 0 = 6

(6x2) + 1 = 13

(13x2) + 0 = 26

(26x2) + 1 = 53

(53x2) + 0 = 106 decimal equivalent

Notice that the middle column of numbers is a sequence of 1's and 0's that matches the binary number. With a little practice, it is possible to make the conversion shorter and do it in your head or with a calculator.

MSB							LSB
1	0	1	1	1	0	1	0
1	2	5	11	23	46	93	186 (Decimal)

In each case the number in the second row was obtained by multiplying the previous result by two and adding the digit above it. Remember that the 'previous result' for the MSB is always zero. The recursive rule of conversion has the advantage that it can also be used to convert hexadecimal numbers into decimal.

Another way to convert binary numbers to decimal numbers is to be consciously aware of digit patterns and number combinations. For example, in decimal the binary number 11111111 is 255. The binary number 11111110 is 255 minus 1 or 254. The number 11111101 is 255 minus 2 (2^1 is missing) or 253. Similarly 11000000 is 192 in decimal and so 11000001 is easily seen to be 193. As you work with binary numbers, more and more of these combinations will become familiar to you. Taking note of them can save a lot of calculations in converting from one system to another.

As an example of where you will use conversion from binary to decimal, to produce a pure note on the Atari, it is necessary to store the bit pattern 10100000 at one of four memory locations. Using any one of the conversion techniques discussed, you can easily convert this binary number to its decimal equivalent, 160.

The largest number that can be represented in binary with 8 bits is 255 decimal. With assembly language it will be necessary to represent numbers as large as 65535. This is accomplished by using two bytes for a total of 16 bits. Referring back to Table 1-2 you can see that the largest number that can be expressed with two bytes is:

$$\overbrace{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}^{\text{Byte 1}}\qquad\overbrace{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}^{\text{Byte 2}}$$

$$\underbrace{32768+16384+8192+4096+2048+1024+412+256}_{\text{Byte 1}}+\underbrace{128+64+32+16+8+4+2+1}_{\text{Byte 2}} = 65355$$

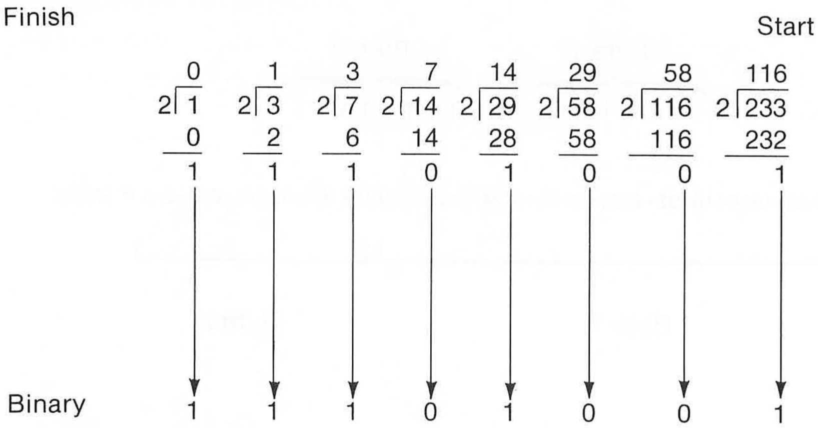
The byte labeled as byte 1 is often called the Most Significant Byte (MSB) or Hi-Byte. Byte 2 is called the Least Significant Byte (LSB) or Lo-Byte. In order to avoid confusion with the most significant and least significant *bits*, hereafter we will refer to bytes 1 and 2 as the High-Byte and Lo-Byte respectively.

Situations in which you have to convert decimal numbers greater than 255 to binary are rare. We will illustrate two ways to do the conversion with 8 bit numbers. The principles are the same for larger numbers. The first method is to use Table 1-2 and subtract powers of 2, recording a 1 for each power used and a 0 for those powers you don't use. For example, to convert 233 to binary proceed as follows:

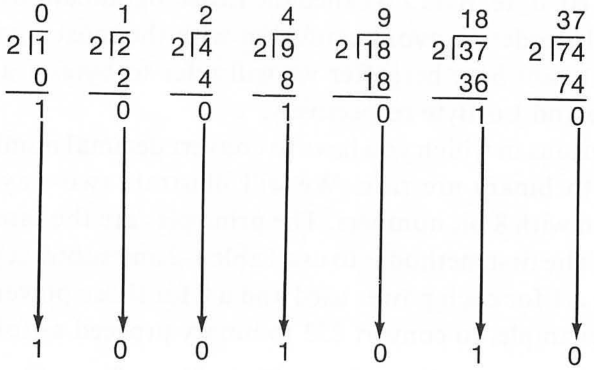
Power of 2:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Value:	128	64	32	16	8	4	2	1
Computation:	233	105	41	9	9	1	1	1
	-123	-64	-32	-16	-8	-4	-2	-1
	105	41	9	X	1	X	X	1
Binary Equi:	1	1	1	0	1	0	0	1

The second way to convert decimal numbers to binary is to repeatedly divide by two and record remainders. Starting with the number, divide by 2 (see example below). If there is no remainder, record a 0 as the LSB. Now divide the quotient and record the remainder as the next bit. Continue until the quotient is zero.

Example 1 using 233:

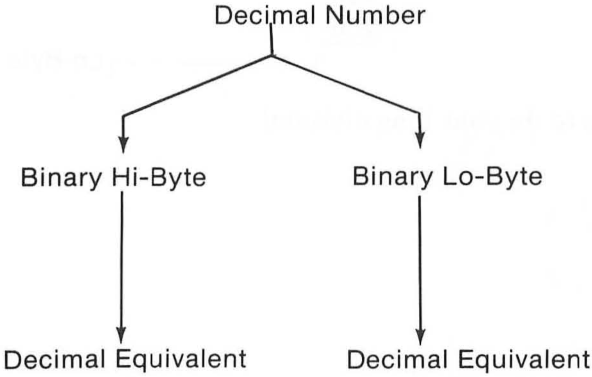


Example 2 using 74:

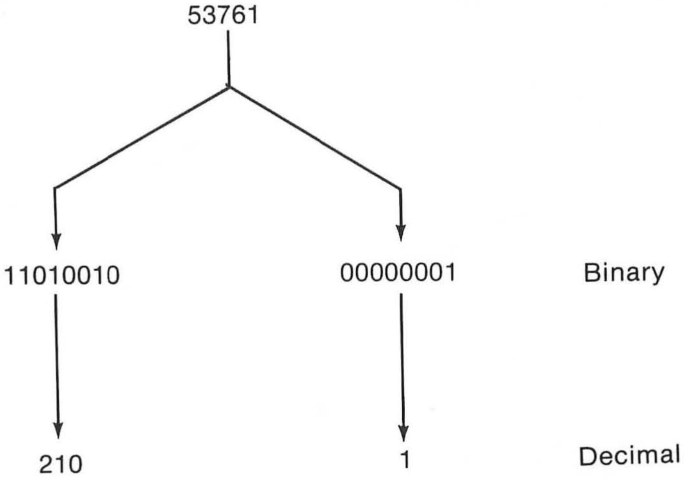


In this case you "pad" the binary to 8 bits and write it as 01001010.

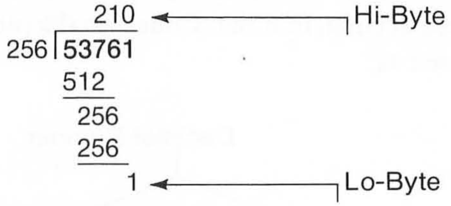
One of the most frequent calculations you will have is to convert a decimal number into its Hi-Byte, Lo-Byte form where each byte is written as a decimal number. Conceptually (not computationally!) this is expressed as:



An example using 53761 (one of those sound registers again):



Computationally this calculation is relatively simple: Divide the decimal number by 256. The quotient is the Hi-Byte. The remainder is the Lo-Byte.



Here is a place to do your long division!

The Hexadecimal System

Use of the hexadecimal number system grew out of a need for programmers to have a convenient way to write and think about binary numbers without carrying around long sequences of 1's and 0's. In writing machine language routines to be called from BASIC, you will use decimal numbers. However, the hexadecimal system is so thoroughly entrenched in use that it is important to become familiar with it as a part of your background knowledge. Hexadecimal numbers take advantage of the fact that a byte is two groups of four digits. A four bit number (sometimes called a nibble) can represent any number from zero to fifteen. The digits 0 to 9 are used as in the decimal number system, but there are no single position symbols for the equivalents to 10,11,12,13,14, and 15. The letters A,B,C,D,E, and F are used so that hexadecimal numbers can be reproduced by a printer. Table 1-3 compares the decimal, binary and hexadecimal systems.

Table 1-3. Decimal, binary, and hexadecimal systems.

DECIMAL	BINARY	HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Conversion of a binary number into hexadecimal is straight forward. Separate it into its groups of four bits and write the hex equivalent of each group:

1 0 1 0 1 1 1 1 = AF

or

0 0 1 0 1 0 1 1 = 2B

Conversion from hexadecimal to binary is equally simple:

A9 = 1 0 1 0 1 0 0 1
↑ ↑
A 9

Conversion from hexadecimal to decimal can be done using the recursive rule [previous result*base]+[next digit]=result rule.

For example:

A9

[previous result(0)x16] + A(=10) = 10

(10 x 16) + 9 = 169

For two digit numbers like this you'll shorten it to (16x10) + 9 = 169. But for numbers such as D201 the rule is helpful:

D201

[previous result(=0)] + D(=13) = 13

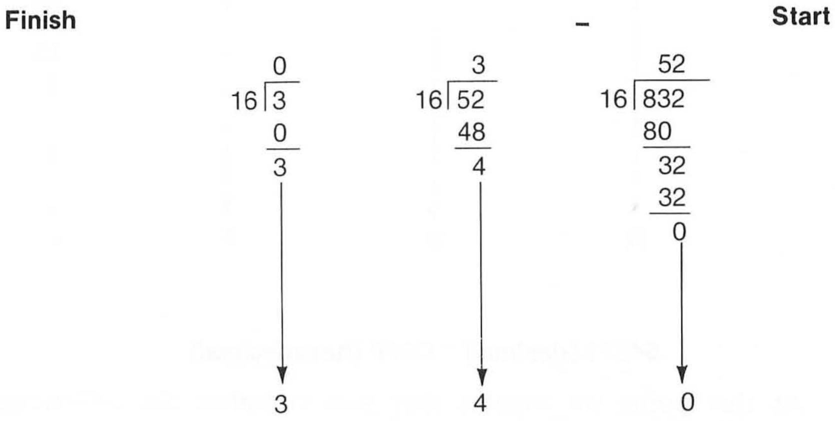
(13 x 16) + 2 = 210

(210 x 16) + 0 = 3360

(3360 x 16) + 1 = 53761

Conversion of decimal numbers into hexadecimal numbers can be accomplished by the repeated division procedure used earlier for conversion from decimal to binary. We'll illustrate the process with some examples:

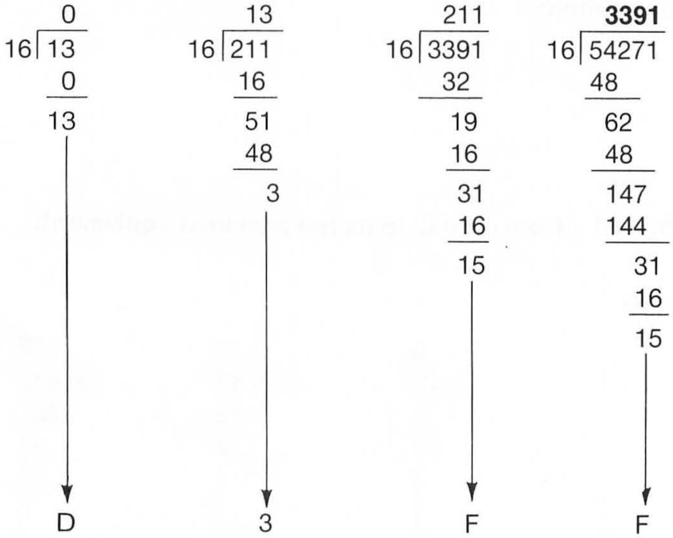
Example 1 - Convert 832 to its hexadecimal equivalent:



832 (decimal) = 340 (hexadecimal)

Example 2 Convert 54271 to its hexadecimal equivalent:

Finish **Start**



54271 (decimal) = D3FF (hexadecimal)

At this point we suggest that you complete the following conversions for practice. These practice problems have been chosen from numbers that you will work with later on and are useful memory locations or assembly language codes.

A. Convert from decimal to Hi-Byte, Lo-Byte form:

1. 1536
2. 53768
3. 53762
4. 53249
5. 54282
6. 54286
7. 58466

B. Convert from hexadecimal to decimal:

1. 22F
2. 230
3. 26F
4. 2F4
5. D407
6. E45F
7. D20E

C. Convert from decimal to binary:

1. 255
2. 198
3. 228
4. 195
5. 248
6. 219
7. 63

Codes

As well as using the digits zero and one to represent numbers in binary form, sequences of 0's and 1's can be used as codes to represent characters, special symbols, and instructions to peripheral devices. Such codes permit communication between the computer's central processor unit (CPU) and the keyboard, TV screen, and printer.

One of the most common codes used in computers is the ASCII Code (American Standard Code for Information Interchange). The ATARI uses a modified version of this code, called ATASCII. The two major differences between ASCII and ATASCII are that the former uses only 7 bits for each character or instruction code while ATASCII uses 8 bits and the ATASCII code includes many special graphics symbols. In fact ATASCII makes use of all the numbers 0 through 255 while ASCII does not. This property of the ATASCII code will be very important later for it will allow you to store machine language routines as strings in BASIC. Appendix E lists the ATASCII code with its corresponding symbols and keystrokes. This listing is an important resource in your work.

Another code that you will come across in assembly language programming is Binary Coded Decimal, or BCD. The idea of BCD is to use binary numbers to represent each digit of a decimal number. Thus, the decimal number 469 is represented by:

0 1 0 0	0 1 1 0	1 0 0 1
4	6	9

The 6502 has instructions that will allow it to perform arithmetic operations using either pure binary numbers or binary coded decimal numbers. This is an advantage to programmers who are writing systems and arithmetic routines. Table 1-4 lists the decimal digits and their binary codes.

Table 1-4. Decimal/Binary equivalents

Decimal Digit	Binary Code
0	0000
1	0001
2	0011
3	0010
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The Central Processor Unit

Unlike many other microcomputers, the Atari uses two processors, the CPU and ANTIC. The central processor (CPU) is the 6502 and it handles all data transfers, arithmetic and logic calculations. ANTIC (Alpha-Numeric Television Interface Chip) is responsible for the production of the TV display. In addition to these processors there are three large scale integrated circuit chips that support the CPU and ANTIC in communicating with the keyboard, TV screen and other peripherals. They are POKEY and GTIA (CTIA in pre Dec. 1981 ATARI'S) and PIA.

The 6502 is an 8-bit processor. This means that it handles data in one byte units. The 6502 communicates with memory and peripherals over three groups of wires called busses. Data travels in and out of the CPU over an eight line data bus. Since there is one wire for each bit, a whole byte is transferred from one place to another at once. This is called parallel data transfer. The 6502 is a memory mapped microprocessor which means it treats memory and peripherals on an equal basis. *Reading* a byte of data in memory is, to the CPU, the same operation as reading a byte of data sent out from a peripheral device, such as the keyboard. Likewise *writing* or sending, a byte of data to memory or a peripheral are equivalent operations. Both the CPU and the peripherals or memory have to agree on whether a given data transfer is to be a *read* or a *write* operation. Therefore, there is a control bus that carries *read/write* and other control signals from the CPU to external devices and memory. Computer memory and peripheral input/output locations are identified by number. The 6502 uses a 16 bit address bus and consequently can identify 2^{16} or 65356 different memory locations.

Internally the microprocessor has an arithmetic logic unit where additions, subtractions and logical operations take place. There is a control unit that decodes instructions and shifts data to and from memory. Of primary importance to us are six special internal data storage locations or registers. Five of these are 8 bit registers. The sixth is a 16 bit register. The five 8 bit registers are the:

1. Accumulator
2. X-Register
3. Y-Register
4. Processor Status Register
5. Stack Pointer.

The remaining register is the 16 bit program counter.

The functions of the registers are as follows:

1. The Accumulator: This is the busiest register in the CPU. It is the only register where arithmetic and logic operations can be performed. Data transfer from one memory location to another usually goes through the accumulator.

2. The X and Y Registers: These two registers are referred to as index registers because one of their primary functions is to serve as a kind of subscript or index used in addressing memory locations. The contents of the X and Y-registers can be incremented or decremented one unit at a time so they serve as very natural loop counters. In addition the X and Y-registers can be used to transfer data between the CPU and memory.

3. The Status Register: This register contains seven usable bits. Two of the bits are control bits. The remaining five are status flags The status flags provide information on the result of a previously executed instruction (usually the preceding instruction). The 6502 can be programmed to test the condition of each of these flags. Based on the results of these tests, the 6502 can choose between two possible sequences of instructions. The locations, labels, and functions of these bits are described in Box 1.

4. The Stack Pointer: The stack is a special storage area in memory at locations 256 to 511 (Page 1 of memory). The stack works as a last-in/first-out storage area analogous to a stack of plates in a kitchen cabinet. It is used to store information necessary to perform subroutine calls and interrupts correctly. On power-up or after a reset, the stack begins at address 511, which in Hi-Byte/Lo-Byte form is 01,255. The stack pointer will, at this time, contain the byte 255, as data is stored or removed on the stack, the stack pointer is incremented or decremented so that it always gives the address of the next available stack location.

5. The Program Counter: Machine language instructions are stored in memory in order by address. The program counter insures that instructions are performed in the proper sequence. At any instant the program counter contains the address of the next byte of the program to be read.

BOX 1

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
N	V		B	D	I	Z	C

Note: If a flag is set, then there is a Logical 1 in that particular bit of the status register. If a flag is clear, then there is a Logical 0 in that particular bit of the status register.

C = Carry Flag. This bit is set when an addition, shift, or rotate generates a carry. It is cleared when a subtraction or compare produces a borrow.

Z = Zero Flag. This bit is set when the accumulator, the index registers or a memory location contain all zeros as the result of an instruction such as increment, decrement or the arithmetic and logical instructions.

I = Interrupt Flag. This bit is set whenever an interrupt in the normal processing occurs or when a Break (BRK) instruction is executed. It is cleared whenever a Return from Interrupt (RTI) instruction is executed.

D = Decimal Flag. This flag is used to signal the processor that addition and subtraction are to be performed in the decimal mode using BCD.

B = Break Flag. This flag is set, along with the I flag whenever a BRK instruction is executed. It is cleared following an RTI instruction.

V = Overflow Flag. This flag is set when an addition or subtraction produces a result greater than 127 or less than -128. It is used in applications involving signed numbers.

N = Negative Flag. This flag indicates whether or not the result of a signed arithmetic operation produced a negative result.

Box 1. Status flags

CPU Operation

A machine language program is nothing more than a series of numbers stored as bytes in memory. When a computer is executing a program it goes through a fundamental sequence that is repeated over and over again. To execute a program, the CPU must *fetch* a byte, *decode* its meaning, and *execute* the instruction. This fetch-decode-execute sequence is repeated at very high speed until the program is finished. The rate at which operations occur within the CPU, and their sequence, is governed by a system clock that sends out electrical pulses at the rate of 1.79 million cycles per second. One cycle per second is also called one Hertz so you will often see this written as 1.79 MHz.

Instructions to the CPU are either one, two, or three bytes long. A three byte instruction consists of a numeric code followed by two address bytes. A typical two byte instruction is one code byte followed by a number. A single byte instruction is simply a code byte to do some task, for example, Return from Subroutine (RTS). CPU instructions take from two to seven clock cycles to execute with most taking three to four cycles. Occasionally it is necessary to time the length of a subroutine in order to insure that it will be completed within a specified time limit. Timing considerations will come up later when we discuss display list interrupts.

Other Processors and Chips

By itself a CPU is limited in its capabilities for communicating with the outside world. The inclusion of four support chips (integrated circuits) in ATARI computers provide a great deal of flexibility in programming graphics and sound. Since a detailed description of how to use these support chips is given in later chapters, we will provide only an overview of their functions here.

1. ANTIC: The primary function of ANTIC is to fetch data from memory and display it on the TV screen. ANTIC does this independently of the CPU by sending a HALT signal that effectively disconnects the CPU from the address and data busses. ANTIC is then free to use these busses to access the memory that it needs. This process is called Direct Memory Access (DMA). ANTIC also controls the non-maskable interrupts of the processor. A Non-Maskable Interrupt (NMI) is just what its name implies - a signal to the CPU, which it cannot ignore, to stop its current operations and go to another program. These interrupts are useful in both sound and graphics programming.

2. GTIA: is a special television interface adaptor chip that works in conjunction with ANTIC. Its primary purpose is the direct control of the TV display. Thus it is responsible for controlling color and luminance, playfield and player missile graphics, collisions and priority.

3. POKEY: is a digital I/O chip that has a variety of functions. One of the foremost is sound generation. It is the keeper of the registers that control the frequency and type of sound output. In addition POKEY takes care of transmission of data from the keyboard and from the serial communications port. Serial data transfer differs from the parallel data transfer mentioned earlier in that data is transmitted one bit at a time. Other functions of POKEY will be described in Chapter Six.

4. PIA: is the Peripheral Interface Adaptor chip with the primary responsibility for controlling data to and from the joystick ports. It is used with any peripheral, such as paddles, joysticks, keypads, or the Koala pad, that plugs into these ports.

Memory Organization: An Overview

The 6502 can address 65356 different memory locations and treats Random Access Memory (RAM), Read Only Memory (ROM), and peripherals in the same manner - as memory locations. A useful concept in discussing memory is the idea of a page. A page in memory is 256 bytes long. If a memory address is given in Hi-Byte/Lo-Byte form, the Hi-Byte is the page number and the Lo-Byte is the address within the page. To illustrate, in binary form, the first byte of page six has the address 00000110,00000000. In decimal Hi-Byte/Lo-Byte form this is written as 6,0. As a single decimal number this is memory location 1536. To find the page number of a memory address written as a single decimal number divide that number by 256. The whole number quotient is the page. The whole number remainder - not the decimal fraction - is the address within the page. Occasionally we will refer to "whole page boundaries" or "1K and 2K boundaries". A memory location is said to lie on a one page boundary if its address is evenly divisible by 256. A memory address lies on a 1K or 2K boundary if its address is divisible by 1024(1K) or 2048(2K) respectively.

In the Atari computers, certain pages of memory are set aside for use by the operating system (OS) and BASIC, while others are used primarily for hardware registers. Which page is chosen for a particular use depends in part on the design of the 6502 microprocessor and in part on choices made by the Atari's designers. Two pages are especially important to the 6502. These are page zero and page one. Page zero is important because it can be accessed by the processor faster and more easily than any other page. Page one is important because the 6502 uses it as the stack. Box 2 gives a summary of memory allocation. A more detailed memory map is given in Appendix D.

BOX 2

LOCATION	USE
65535 to 55296	Used for the operating system and Arithmetic Routines
55295 to 53248	Hardware Registers ANTIC 54272 — 54783 PIA 54016 — 54271 POKEY 53760 — 54015 GTIA 53248 — 53503
53247 to 40960	4K unused memory BASIC or Left Cartridge

Box 2. Memory overview

40959 to 32768	Used for screen memory and Display Lists. Amount used depends on graphics mode
32767 to	User program RAM. Location of the bottom depends on presence or absence of DOS and other factors
1791 to 1536	Page Six. Unused by OS or BASIC. May be used to store machine language routines
1535 to 1152	RAM used by BASIC and Arithmetic routines
1151 to 512	OS RAM. Contains shadow registers used to update hardware registers during vertical blank
511 to 256	STACK Page 1
255 to 128	BASIC Page Zero
127 to 0	OS Page Zero

Box 2. (cont.)

Machine Language Programming: Some Comments

Many of the machine language programming concepts that will be discussed in later chapters are illustrated by sound and graphics applications. Machine language programs used in sound and graphics usually do one or more of three basic operations:

1. Move massive amounts of data (500 to 1000 bytes) from one place to another. An example is a routine to redefine the character set that may move 512 or 1024 bytes.

2. Change the values in one or more hardware registers at intervals in either space or time. The values placed in the registers often come from a table. To illustrate, a table driven display list interrupt routine changes color values at screen (space) intervals. While a music program changes notes in sound registers at timed intervals.

3. Increment or decrement one or more hardware registers or memory locations. Horizontal or vertical scrolling illustrates this operation.

In implementing the above routines, the principles used are very similar. One makes use of the accumulator, the X-register and the Y-register. Of these, the X and Y registers are most commonly used as counting or indexing registers to either keep track of how many times we've cycled through a routine or to successively locate items of data in a table.

2

OVERVIEW OF 6502 INSTRUCTIONS

Introduction

Table 2-1 is a list of the fifty-six different instruction names for the 6502 CPU. Each instruction name has been coded into a three letter mnemonic that is suggestive of the task to be carried out. Roughly half of these instructions perform simple workman-like jobs such as transferring the contents of one register into another, incrementing or decrementing a register, and setting or clearing a bit. Approximately half of the remaining instructions manipulate data by transfering it to and from memory, or comparing a register with the contents of memory.

One of the most important features of the 6502 is the number of options available for specifying the location of the byte to be manipulated. There are thirteen different addressing modes. This is several more than are available on other common 8-bit microprocessors and provides for greater flexibility in programming the 6502. The fifty-six basic instructions in combination with the thirteen addressing modes yields a total of 151 different instructions available to the programmer.

Table 2.1. 6502 Microprocessor instruction set

ADC	Add memory to Accumulator with carry
AND	Logical "AND" of memory with Accumulator
ASL	Shift left one bit (Accumulator or memory)
BCC	Branch or carry clear
BCS	Branch or carry set
BEQ	Branch on result equal to zero
BIT	Test bits in memory with Accumulator
BMI	Branch on result minus
BNE	Branch on result not equal to zero
BPL	Branch on result plus
BRK	Force Break
BVC	Branch on overflow clear
BVS	Branch on overflow set
CLC	Clear the carry flag
CLD	Clear decimal mode
CLI	Clear the interrupt disable bit
CLV	Clear the overflow flag
CMP	Compare memory and Accumulator
CPX	Compare memory and X-Register
CPY	Compare memory and Y-Register
DEC	Decrement memory by one

Table 2.1. (cont.)

DEX	Decrement X-Register by one
DEY	Decrement Y-Register by one
EOR	Logical "Exclusive-OR", memory with Accumulator
INC	Increment memory by one
INX	Increment X-Register by one
INY	Increment Y-Register by one
JMP	Jump to new location
JSR	Jump to subroutine
LDA	Load the Accumulator
LDX	Load the X-Register
LDY	Load the Y-Register
LSR	Shift right one bit (Accumulator or memory)
NOP	No operation
ORA	Logical "OR", Memory with Accumulator
PHA	Push Accumulator onto stack
PHP	Push Processor Status Register onto stack
PLA	Pull value from stack into Accumulator
PLP	Pull value from stack into Processor Status
ROL	Rotate one bit left (Accumulator or Memory)
ROR	Rotate one bit right (Accumulator or Memory)
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract memory from Accumulator with borrow
SEC	Set carry flag
SED	Set decimal mode

Table 2.1. (cont.)

SEI	Set interrupt disable
STA	Store Accumulator in memory
STX	Store X-Register in memory
STY	Store Y-Register in memory
TAX	Transfer Accumulator to X-Register
TAY	Transfer Accumulator to Y-Register
TSX	Transfer Stack Pointer to X-Register
TXA	Transfer X-Register to Accumulator
TXS	Transfer X-Register to Stack Pointer
TYA	Transfer Y-Register to Accumulator

The most effective approach to learning these instructions is to group them according to their function. The first part of this chapter will give brief descriptions of the instructions. The second part of the chapter will discuss the different addressing modes.

Instructions by Function

1. Load and Store Instructions: Since the 6502 processor has a memory-oriented design, the most fundamental operations involve transferring information into and out of memory. All such transfers can be made using either the Accumulator, the X-register, or the Y-register. Loading consists of copying the information from memory into one of the three registers. This is a non-destructive operation in that the byte in memory is not altered by the load instruction. The three load instructions are:

LDA	Load Accumulator with memory
LDX	Load X-register with memory
LDY	Load Y-register with memory

2. Register Transfer Operations: There are six one-byte operations that transfer data between the registers. These are:

TAX	Transfer Accumulator to X-register
TXA	Transfer X-register to Accumulator
TAY	Transfer Accumulator to Y-register
TYA	Transfer Y-register to Accumulator
TSX	Transfer Stack Pointer to X-register
TXS	Transfer X-register to Stack Pointer

Of these six instructions, the first four will be the most useful. Examples of their use will be demonstrated with display list interrupt routines in Chapter Four.

3. Increment and Decrement Instructions: One of the functions of the X and Y registers is to serve as general purpose counters. In addition it is often desirable to set aside a memory location as a counter. Counters are useful in accessing consecutive memory locations or keeping track of the number of passes through a loop. The increment and decrement instructions are:

INX	Increment X-register by one
DEX	Decrement X-register by one
INY	Increment Y-register by one
DEY	Decrement Y-register by one
INC	Increment memory by one
DEC	Decrement memory by one

4. Compare and Branch: Compare instructions are commonly used to determine if a register or memory location that is being used as a counter has reached a certain value. The three compare instructions are:

CPX	Compare X-register and memory[1]
CPY	Compare Y-register and memory
CMP	Compare Accumulator and memory

[1] Compare instructions can also compare the contents of the register with the number immediately following the instruction code.

The compare instructions subtract the contents of memory from the register but does not save the result. The indications of the result are the conditions of the three status flags: N (negative), Z (zero), and C (carry). The condition of these flags indicate whether the register contents are less than, equal to, or greater than the contents of memory location.

A very natural follow-up to a compare instruction is to make a decision on the basis of the comparison. A common decision is whether or not to branch to another part of the program. There are eight branch instructions. Two of these,

BEQ Branch on Result Equal to zero

and

BNE Branch on Result Not Equal to zero

work very nicely with the compare instructions when you want to execute a loop. The remaining six branch instructions are:

BCC	Branch on Carry Clear (carry = 0)
BCS	Branch on Carry Set (carry = 1)
BMI	Branch on Minus (Negative (N) = 1)
BPL	Branch on Plus (Negative (N) = 0)
BVS	Branch on Overflow Set (overflow (v) = 1)
BVC	Branch on Overflow Clear (overflow (v) = 0)

5. Jump and Return: The branch instructions are called conditional because they cause a change in a program's normal sequential flow only if some condition is met. The jump and return instructions cause unconditional changes in program flow. There are three of these instructions:

JMP Jump to a specified memory location
JSR Jump to a subroutine
RTS Return from subroutine

6. Interrupt Instructions: Interrupts are signals to the processor from another chip or peripheral requesting the processor's attention. There are two types of interrupts; Non-Maskable Interrupts (NMI) and Interrupt Requests (IRQ). Whether or not the processor responds to an IRQ depends on the IRQ disable bit (I) in the processor status register. If the I bit is clear (ie. equal to zero), then the external interrupt will be serviced. If the I bit is set (ie. equal to 1), the processor will ignore the interrupt request. The instructions to set and clear this bit are:

SEI Set Interrupt Disable Bit
CLI Clear Interrupt Disable Bit

Interrupt requests cause the processor to go to a subroutine that services the interrupting device. Such a subroutine must end with:

RTI Return from Interrupt

7. Stack Operations: The stack is used as a temporary storage place for register contents, parameters, and return addresses needed to get back from a subroutine. The instructions to transfer data to and from the stack are:

PHA Push Accumulator on stack
PLA Pull Accumulator from stack
PHP Push Processor Status on stack
PLP Pull Processor Status from stack

The instructions involving the accumulator will be useful in the programs in later chapters. Notice that there are no instructions to move the contents of the X or Y registers directly to the stack. Therefore, data from these registers must be transferred through the accumulator to the stack. PHA is a 'copying' instruction. It does not destroy the contents of the accumulator. On the other hand, PLA *removes* the byte from the stack.

8. Arithmetic Instructions: It is not our intention to discuss how to write routines to add, subtract, multiply and divide numbers. Routines to do this are covered thoroughly in many other books. However, you may occasionally find it useful to add or subtract a pair of numbers. The 6502 can add or subtract numbers in a binary form or binary coded decimal form. The form used is controlled by the two instructions:

SED Set Decimal mode

and

CLD Clear Decimal mode

Of course, SED causes the CPU to work in the decimal mode while CLD directs the CPU to act in the binary mode.

Addition can be completed with or without a carry occuring as part of the result. Similarly subtraction can be performed with or without borrowing. Unlike some other processors, the 6502 only has instructions for addition with a carry and subtraction with borrow. The digit to be 'borrowed' is contributed by the carry flag of the status register. Before an addition the carry flag should be cleared. Before a subtraction the carry flag should be set. The relevant instructions are:

CLC Clear Carry
ADC Add with Carry
SEC Set Carry
SBC Subtract with Carry as Borrow

Other instructions used in arithmetic routines are:

CLV	Clear Overflow (V) flag. The overflow flag is used in signed arithmetic routines.
ASL	Accumulator Shift Left
LSR	Logical Shift Right
ROL	Rotate Left
ROR	Rotate Right

These last instructions are commonly used in multiplying and dividing numbers. They may also be used to perform serial-to-parallel and parallel-to-serial conversions when the CPU is communicating with serial oriented peripherals.

9. Logical and Miscellaneous Instructions: Situations may arise in which you wish to test certain bits rather than a whole byte. Or, you may wish to set or clear certain bits. The logical instructions will allow you to do this. They are:

AND	And memory with Accumulator
EOR	Exclusive Or memory with Accumulator
ORA	Or memory with Accumulator

The AND instruction is primarily used to mask out (set to zero) certain bits in the accumulator. The EOR instruction is primarily used to determine which bits differ between the accumulator and memory. Finally, the ORA instruction is used to set certain bits.In their simplest form, the logical operations AND, ORA, and EOR produce a single bit result after a comparision of two input bits. The possible results are summarized in Box 3.

Box 3

AND

Input Bit #1	1	1	0	0
Input Bit #2	1	0	1	0
Result	1	0	0	0

OR

Input Bit #1	1	1	0	0
Input Bit #2	1	0	1	0
Result	1	1	1	0

EOR

Input Bit #1	1	1	0	0
Input Bit #2	1	0	1	0
Result	0	1	1	0

The 6502's Logical Operations perform eight separate comparisons, one for each pair of bits D_7 through D_0. For example 10110101 ANDed with 11101110 produces:

Input #1	1	0	1	1	0	1	0	1
Input #2	1	1	1	0	1	1	1	0
Result	1	0	1	0	0	1	0	0

Box 3. Logical AND, Logical OR, Exclusive OR

There are three instructions that we have classified as miscellaneous. They are:

NOP No Operation
BRK Break
BIT Test Bits in memory with Accumulator

A NOP instruction causes the processor to do nothing at all. NOP can be used to reserve space in a program under development, to make the processor pause for a few machine cycles, or to replace instructions that have been removed without requiring all of the branch and jump addresses to be changed. BRK is commonly used in debugging during the early stages of program development. It causes the processor to execute an interrupt sequence after which you can check what your program has accomplished to that point.

The BIT instruction will test a bit in memory by ANDing it with the accumulator. The command does not alter either the accumulator or memory, but records information in the status register as follows:

N flag is the original value of bit 7 of the memory byte.
V flag is the original value of bit 6 of the memory byte.
Z flag is set if the AND operation generates a zero.

BIT is a special purpose instruction that is used in communication between the PIA and the 6502. It is used mainly in programs that are written as part of hardware interfacing.

Addressing Modes

Each of the 151 instructions has its own numeric operation code (op-code, for short). Most instructions consist of one byte of op-code plus a one or two byte operand. The op-code tells the CPU what task is to be performed and the mode of addressing used. The operand may be data, information pertinent to the location of the next instruction to be executed, or may refer to the location where data is to be found or placed.

The instructions that make up a program are located sequentially in memory. The CPU recognizes a byte as an op-code or operand through the combined efforts of the internal decoding logic and the program counter. At the start of a machine language routine, the program counter contains the address of the first op-code. The processor fetches this byte into the decoding section and at the same time the program counter is incremented to the address of the next byte in memory. Once the op-code has been decoded, the CPU will know how to interpret the next byte it fetches. Thus, how a particular byte is interpreted is context dependent. One common cause of program failure is a branch that is executed and the byte branched to is not a valid op-code, but is the operand of some other instruction.

As mentioned earlier, there are thirteen addressing modes. These modes can be divided into two groups; the seven basic modes:

1. **Immediate**
2. **Absolute**
3. **Zero Page**
4. **Indirect**
5. **Implied**
6. **Relative**
7. **Accumulator**

and six modes that are a combination of indexed addressing and one of the basic modes:

8. **Absolute X-indexed**
9. **Absolute Y-indexed**
10. **Zero Page X-indexed**
11. **Zero Page Y-indexed**
12. **Indirect indexed**
13. **Indexed indirect**

Immediate Addressing Mode: The immediate addressing mode takes its operand from the memory location immediately following the op-code. Therefore, it is a two-byte instruction; one byte of op-code followed by a one byte operand.

Absolute Addressing Mode: In the absolute addressing mode, the two bytes following the op-code give the memory address from which the CPU is to fetch data to be operated on, or where a byte of data is to be stored. The absolute addressing mode has the following format:

First byte	Second byte	Third Byte
Op-code	Lo-Byte of addr.	Hi-Byte of addr.

This 'reverse' form of writing the memory address takes a bit of adjusting to at first, but rapidly becomes second nature.

Zero Page Addressing: This is a form of absolute addressing in which the CPU 'knows' that the Hi-Byte of the memory location is page zero. All that is needed in addition to the op-code is a single byte to specify the particular memory location in page zero. The advantage of page zero addressing is speed. A two byte instruction can be processed more rapidly than the three byte instructions required by other addressing modes. Unfortunately, the only zero page locations normally available to the Atari user are 203 through 209.

Indirect Addressing: Indirect addressing without indexing applies only to the JMP instruction. The idea of indirect addressing is that an intermediate storage area is used to hold the actual address that will be used by the instruction. This can be clarified with an example:

```
JMP 128 06
```

This instruction sends the CPU to memory location 1664 (06,128) for the Lo-Byte of the effective address. The Hi-Byte of the effective address is in the next memory location, 1665 (06,129). The advantage of indirect addressing is that it allows a fixed instruction sequence to go to different memory locations simply by changing the values in the immediate storage area.

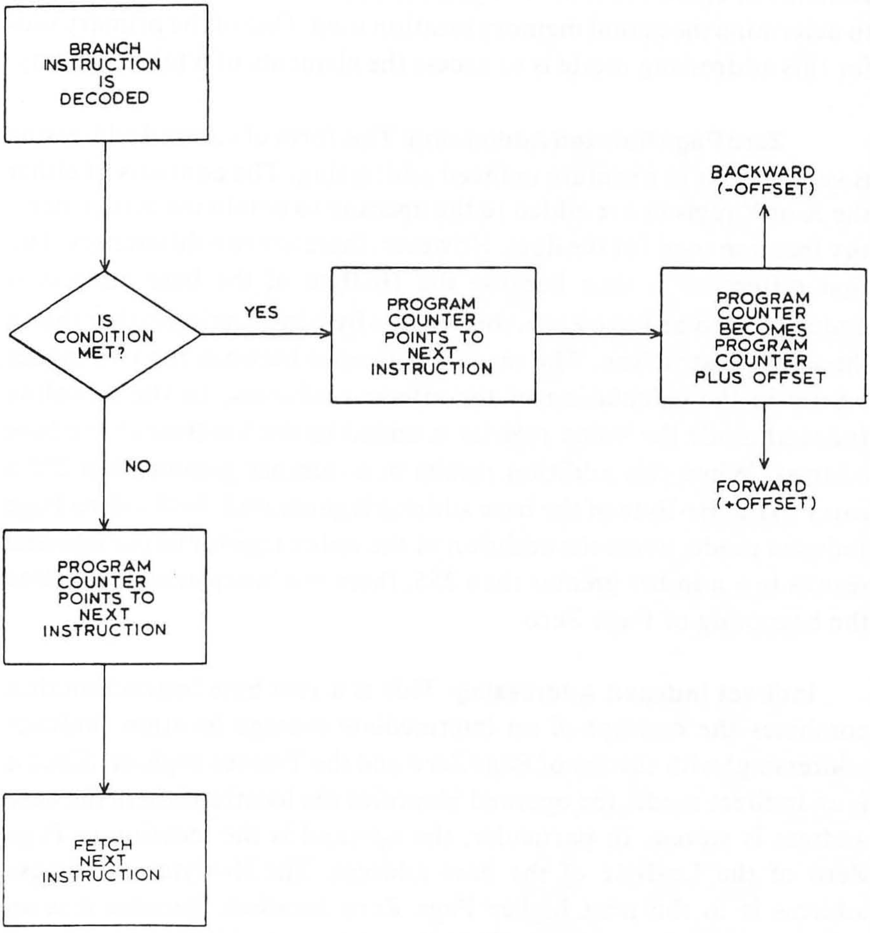
Implied Addressing: Many instructions involve operations internal to the CPU itself. These are simple tasks such as incrementing a register or data transfer between registers. Since the registers involved are internal to the 6502, they have no assigned address. Both the source and destination address is implied in the instruction. For example, TXA (Transfer X-register to Accumulator).

Relative Addressing: Relative addressing is used with branch instructions. The effective address is calculated with respect to the location of the *op-code* following the branch instruction. Flow chart 2-1 summarizes what takes place.

The offset of a branch instruction is specified by a *single* byte operand. This implies that the numbers 0 to 255 are used to represent both forward and backward offsets. Technically, the offset is interpreted as a twos complement signed number. Essentially what this means is that the numbers 1 to 127 represent forward branches and the numbers 255 to 128 represent backward branches. Forward branches should be no problem to figure out - you simply count forward beginning with zero. For backward branches you count back from 256. Thus, an offset of minus one is represented by 255, an offset of minus seven is represented by 249.

Accumulator: Accumulator addressing is an implied type of addressing that is unique to the four instructions that shift and rotate the contents of the accumulator.

FLOW CHART 2-1



Absolute Indexed Addressing: There are two forms of this addressing mode: Absolute X-indexed and Absolute Y-indexed. Both modes function in the same manner. Remember that in absolute addressing the two bytes following the op-code specify the address of the data to be manipulated. In absolute indexed addressing, the contents of either the X or Y register are added to an absolute address to determine the actual memory location used. One of the primary uses for this addressing mode is to access the elements of a table or array.

Zero Page Indexed Addressing: This form of indexed addressing is very similar to absolute indexed addressing. The contents of either the X or Y register are added to the operand to obtain the actual memory location used for the data. However, there are two differences. The first difference is that because the Hi-Byte of the base address is understood to be Page Zero, this is a two byte instruction rather than a three byte instruction. The second difference between the two modes occurs in the calculation of the effective address. In the Absolute Indexed mode the index register is added to the Lo-Byte of the base address. When this addition results in a number greater than 255 a carry to the Hi-Byte of the base address is generated. In the Zero Page Indexed mode, when the addition of the index register to the operand results in a number greater than 255, there is a 'wrap around' back to the beginning of Page Zero.

Indirect Indexed Addressing: This is a two byte instruction that combines the concept of an intermediate storage location (indirect addressing) with the use of Page Zero and the Y-index register. Since it is an indirect mode, the operand identifies the location where the base address is stored. In particular, the operand is the location in Page Zero of the Lo-Byte of the base address. The Hi-Byte of the base address is in the next higher Page Zero location. Because it is an indexed mode, the contents of the Y-register are added to the value of the base address to obtain an effective address used by the instruction.

Indexed Indirect Addressing: In the previous mode, the index was added to the value in memory to determine the location of the data. However, in Indexed Indirect Addressing, the operand plus the index determines the intermediate location where the effective address is stored. Indexed Indirect Addressing is a two byte instruction in which the X-index register is added to the operand to give the location in Page Zero of the Lo-Byte of the effective address. The Hi-Byte of the actual address is stored in the next higher Page Zero location.

3

Atari Graphics

Introduction

The video portion of the Atari Home Computer system was developed to be compatible with the functioning of an ordinary TV set. For example, the system clock was designed to have a frequency that is a multiple of a fundamental TV frequency. This allows CPU interrupts during horizontal and vertical blanks to be easily implemented. The compatibility of the TV and the computer is an important feature of Atari graphics. In addition, the Atari system is unique among home computers because it uses a second microprocessor (ANTIC) to control the TV display. Since there is such an intimate connection between the Atari system and the TV set, we shall begin this chapter with a description of how a TV operates, with the remainder of the chapter devoted to an in depth discussion of ANTIC and Atari graphics.

TV Operation

The picture on a TV screen is made up of many small picture elements, or pixels. A TV picture is produced by the interaction of a modulated beam of electrons with phosphors on the screen. At the rear of the TV is an electron gun that produces a narrow beam of electrons. In the beginning of the sequence that forms a picture, the beam is aimed above the upper left hand corner of the TV screen (see figure 3-1). The beam sweeps from left to right across the face of the screen in 64 μ sec. (a microsecond is a unit of time equal to one millionth of a second). A single horizontal sweep of the electron beam across the screen is called a scan line. When the electron beam reaches the end of the scan line it is shut off briefly and the electron gun is re-aimed at the left side of the screen, but slightly lower down. The period of time that the electron gun is turned off is called the *horizontal blank* (14 μ sec.).

This horizontal scanning process is repeated until a picture is built up line by line. The complete sequence from top to bottom is called a frame and sixty complete frames are drawn per second. In a normal TV picture received from a broadcast station there are 525 scan lines per frame in an arrangement called interlacing. The Atari system does not use interlacing and there are 262 scan lines from top to bottom. In actuality, the electron beam scanning starts slightly above and ends slightly below the visible portion of the TV screen. Similarly, it extends slightly to the left and slightly to the right of the visible screen. This overscanning prevents unsightly borders for normal TV pictures, and must be taken into consideration in computer displays.

Vertical positioning on the screen is measured in scan lines. Horizontal positioning on the screen is measured in units called color clocks. There are two machine cycles per color clock and 228 color clocks per scan line. Display dimensions including overscan are 262 scan lines by 228 color clocks. To prevent loss of information due to overscanning, the normal Atari display uses 192 scan lines by 160 color clocks.

When the electron beam reaches the end of the last scan line it is shut off and the electron gun is re-aimed at the upper left hand corner of the screen. This period of time in which the beam is off is called the

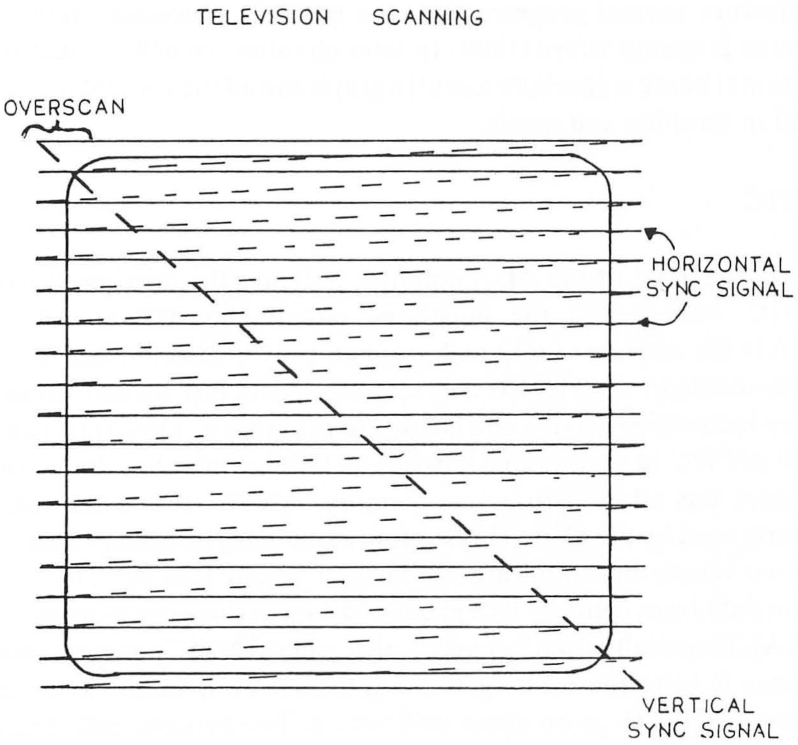


Figure 3-1. Television scanning

vertical blank (1400 μ sec.). The horizontal and vertical blanks are important events in the Atari system. The *display hardware* generates horizontal and vertical synchronization pulses that are used by the TV and can also be used to signal interrupts to the CPU. The Atari system uses these synchronization signals to give programmers an opportunity to interrupt normal program flow and have the processor carry out machine language subroutines. In later chapters we will see that the horizontal blank is specially useful in graphics and the vertical blank is useful in scrolling and music.

ANTIC

At the heart of the Atari graphics system is the microprocessor, ANTIC. Along with the integrated circuit chip GTIA, ANTIC controls the display of text and graphics on the screen. Since it is a microprocessor, ANTIC has control lines, a data bus, an address bus, and an instruction set that can be used to program it. The control lines allow ANTIC to communicate with the CPU, while the address bus and data bus allow it to access memory. ANTIC shares the RAM memory used by the 6502. This sharing of memory by both processors has two interesting implications. First, it means that ANTIC must obtain data from memory by a process known as direct memory access (DMA). Essentially what happens is that when ANTIC needs access to memory it halts the CPU, gets the information it needs, and then allows the CPU to go on about its business. This process, called cycle stealing, slows down the CPU's execution speed. The second implication is that the CPU can modify the sections of memory used by ANTIC. This, of course, is precisely the idea behind creating dynamic graphics -ANTIC handles the details of generating the TV display while the CPU changes the data ANTIC uses.

ANTIC's program is called a *display list*. The section of memory used by ANTIC to determine what to display on the screen is called *screen memory*. The remainder of this section of the chapter is devoted to a discussion of ANTIC's instruction set and writing display lists that provide custom graphics.

Although ANTIC, like the 6502, has a 16 line address bus, it has limitations in addressing a display list and screen memory. ANTIC has two registers that act as program counters. There is a display list counter for accessing the display list and a memory scan counter for accessing screen memory (recall that a program counter holds the address of the byte to be fetched in the fetch-decode-execute sequence). The two counters are each 16 bits wide, but do not function as full 16 bit counters. The upper six bits of the display list counter are fixed, leaving bits D_0 to D_9 to act as the program counter. Restricting the counter to ten bits means that a display list cannot cross a 1K boundary unless a jump instruction is used. This is because the largest decimal number that can be represented with ten bits is 1023. If the display list starts on a 1K boundary, that is, if the starting address of the display list is divisible by 1024 with no remainder, it usually will not be a problem. Most display lists are short - less than a hundred bytes.

While the upper six bits of the display list counter are fixed, only the upper four bits of the memory scan counter are fixed. This leaves bits D_0 to D_{11} to function as the counter. Since the largest decimal number that can be represented with twelve bits is 4095, ANTIC cannot access screen memory that crosses a 4K boundary without a special instruction.

ANTIC's instructions can be grouped as:

1. Display mode instructions
   a. Character mode
   b. Map mode
2. Blank line instructions
3. Jump instructions
   a. Jump during vertical blank (JVB)
   b. Jump to a new memory address (JMP)

In addition to these instructions, there are a number of special options available. These are load memory scan (LMS), display list interrupts (DLI's), and scrolling.

ANTIC combines the TV's scan lines into groups known as mode lines. Each mode line is made up of one to sixteen scan lines depending on the graphics mode. There are two types of graphics modes - character mode and map mode. Character mode instructions cause ANTIC to display a mode line with alpha-numeric or character graphics in it. Each byte in screen memory is the internal code of the character to be displayed. Map mode instructions cause ANTIC to display solid color pixels.

Blank line instructions cause ANTIC to display one to eight scan lines in the background color. These instructions are most commonly used to allow for TV overscan.

Jump instructions are analogous to the BASIC GOTO command, except that you specify the memory address to go to, not a line number. When the program is run the address specified is loaded into the display list counter and consequently starts the fetch-decode-execute sequence at the new memory location.

Instructions for any 8 bit microprocessor such as ANTIC are coded as binary numbers. We shall examine ANTIC's instruction byte in detail to see how to derive the decimal code for each instruction and option. The ANTIC instruction byte can be represented as:

D_7 D_6 D_5 D_4	D_3 D_2 D_1 D_0
instruction option nibble	display mode nibble

Bits D_3 to D_0 are used to determine the display mode and bits D_7 to D_4 select the special options - DLI, LMS and scrolling. Table 3-1 gives the display mode corresponding to each of the possible bit patterns.

Notice in Table 3-1 there are nine display modes listed as BASIC modes and five display modes listed as ANTIC modes. The BASIC modes are accessible with the BASIC GRAPHICS command. In the 400/800 the ANTIC modes are accessible only by creating your own display list for ANTIC to follow. In the XL and XE series the OS supports the ANTIC Modes.

Table 3-1. Display mode according to bit patterns

Lower Nibble D_3	D_2	D_1	D_0	Decimal	Display Mode
0	0	1	0	2	Character - BASIC Mode 0
0	0	1	1	3	Character - ANTIC Mode 3
0	1	0	0	4	Character - ANTIC Mode 4
0	1	0	1	5	Character - ANTIC Mode 5
0	1	1	0	6	Character - BASIC Mode 1
0	1	1	1	7	Character - BASIC Mode 2
1	0	0	0	8	Map mode - BASIC Mode 3
1	0	0	1	9	Map mode - BASIC Mode 4
1	0	1	0	10	Map mode - BASIC Mode 5
1	0	1	1	11	Map mode - BASIC Mode 6
1	1	0	0	12	Map mode - ANTIC Mode 12
1	1	0	1	13	Map mode - BASIC Mode 7
1	1	1	0	14	Map mode - BASIC Mode 14
1	1	1	1	15	Map mode - BASIC Mode 8

Note: The supporting system of the 'XL' and 'XE' series computers supports ANTIC Modes.

To use				
To use	ANTIC 4	call a Graphics	12	command
	ANTIC 5		13	
	ANTIC 12		14	
	ANTIC 14		15	

Bits D_7 to D_4 select the special options as follows:

1. **Bit D_7** is used for display list interrupts. If D_7 is set (equal to 1) and also bit 7 of memory location 54286 (Non-Maskable Interrupt Enable, NMIEN) is set, the processor is interrupted during the horizontal blank.

2. **Bit D_6** is used for the Load Memory Scan (LMS) option. When this bit is set, the next two bytes in the display list will be loaded into the memory scan counter as the address of screen memory. As with the 6502, the address bytes must be written in Lo-Byte/Hi-Byte order.

3. **Bit D_5** if set, enables vertical fine scrolling.
4. **Bit D_4** if set, enables horizontal fine scrolling.

To demonstrate these concepts, consider these examples. Suppose you need an instruction for ANTIC that enables a DLI and horizontal scrolling in BASIC mode 7. Then the instruction code is:

Weight	128	64	32	16	8	4	2	1
Bit	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Binary	1	0	0	1	1	1	0	1
DLI	Horizontal Scrolling				Basic mode 7			

Using the techniques in chapter one you can convert this into its decimal equivalent: 128+16+8+4+1=157. The decimal value is what you will use in your display list.

As another example, suppose you need an LMS in BASIC Graphics 2. The bit pattern is:

Weight	128	64	32	16	8	4	2	1
Bit	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Binary	0	1	0	0	0	1	1	1
		LMS			BASIC Mode 2			

The decimal equivalent of this binary number is 64+4+2+1=71. However, because of the LMS option, this is no longer a single byte instruction. The instruction code, 71, must be followed by two address bytes giving the location of screen memory.

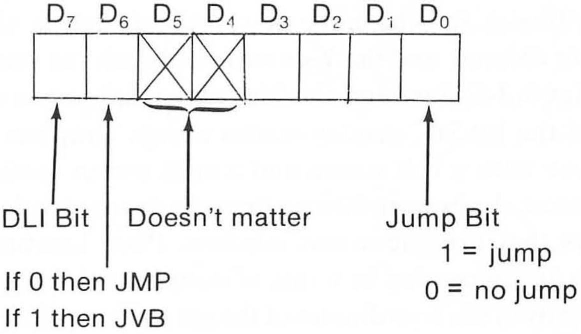
The remaining ANTIC instructions, blank line and jump instructions, are less complicated than display mode instructions. When bits D_3 to D_0 are all zero the instruction byte is identified as a blank line instruction. Then bits D_7 to D_4 determine the number of blank scan lines as listed in Table 3-2.

Table 3-2. D_0 to D_7 correlation to blank lines

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Decimal Value	NumberofBlankLines
0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	16	2
0	0	1	0	0	0	0	0	32	3
0	0	1	1	0	0	0	0	48	4
0	1	0	0	0	0	0	0	64	5
0	1	0	1	0	0	0	0	80	6
0	1	1	0	0	0	0	0	96	7
0	1	1	1	0	0	0	0	112	8

As in display mode instructions, bit 7 of the blank line instruction is used for display list interrupts.

The two jump instructions are jump during vertical blank (JVB) and jump to a new address (JMP). The first jump instruction reloads the display list counter with the address of the first instruction of the display list. As its name implies the loading occurs during the vertical blank. The JVB instruction is the last instruction in a display list and causes ANTIC to execute an endless loop that reads the display list each time a frame is drawn on the TV. The JMP instruction enables ANTIC to cross a 1K boundry in a display list. The instruction format for jumps is:



Thus a JVB without the DLI option has the code 65 and the JMP instruction without the DLI option has the numeric code 01.

Display Modes

An important feature of Atari graphics is the ease with which a programmer can mix graphics modes on the screen by writing a custom display list. Before we discuss how to construct a custom display list, it is important to have an understanding of Atari display modes. First, we shall describe the modes available from BASIC. Then we will describe the ANTIC display modes.

The fundamental structure of the TV display is 192 scan lines vertically and 160 color clocks horizontally. The basic differences between the display modes are how this structure is organized into pixels, and the colors available. There are three character modes and six map modes accessible from BASIC. These are BASIC Modes 0, 1, 2 and 3, 4, 5, 6, 7, 8, respectively. Atari Computers with GTIA support three additional graphics modes (9,10,11) that are enhancements of Graphics 8. As an example of the different ways to organize the basic structure, Graphics 0 uses pixels that are 8 scan lines by 4 color clocks, while Graphics 2 uses pixels that are 16 scan lines by 8 color clocks. Consequently, Graphics 2 pixels are twice as high and twice as wide as Graphics 0 pixels. Additionally, Graphics 0 has two colors and Graphics 2 has five colors available.

The location of pixels on the screen is conveniently described by X-Y coordinates in which the X-coordinate labels the horizontal position, or column and the Y-coordinate labels the vertical position, or row. Figure 3-2 illustrates this idea with a full screen in Graphics 2.

All of the BASIC display modes except Graphics 0 and GTIA Modes have both a full screen and a split screen version. In a split screen version, the bottom 32 scan lines are devoted to four Graphics 0 mode lines that provide a text window. Pixel location in the text window, when expressed in terms of coordinates, is best thought of independently of the coordinates of the graphics mode above it. Figure 3-3 shows Graphics 3 with a text window to emphasize this point. In figure 3-3, the upper lefthand pixel of the text window is labeled 0,0, just as if it were at the top of the screen. This idea of locating pixels within a particular group of mode lines independently of the other graphics modes on the screen is useful with custom display lists and mixed modes.

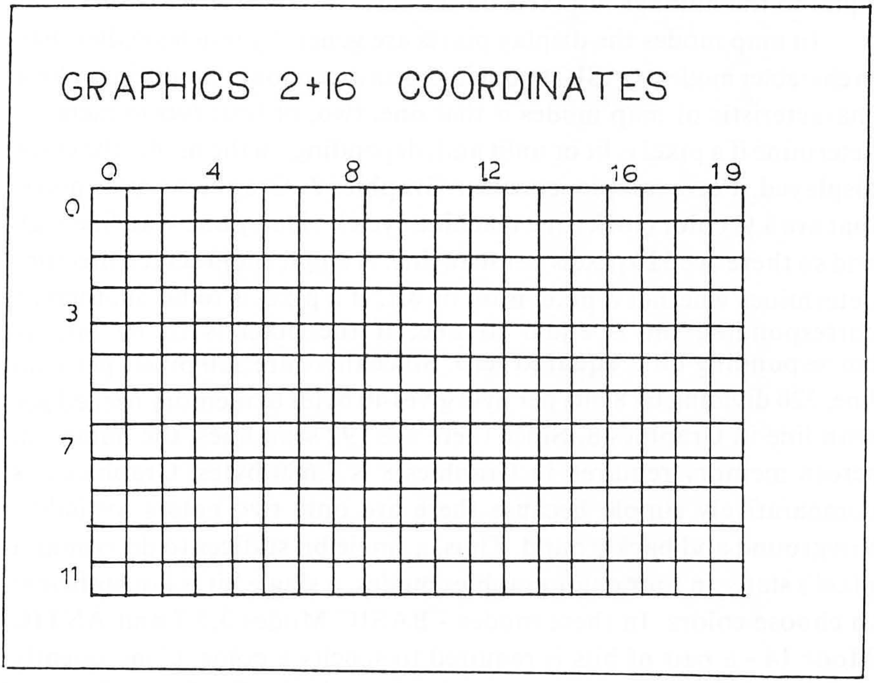


Figure 3-2. X-Y coordinates

Another factor that distinguishes the various display modes is the amount of screen memory required for each mode. In general, character modes require much less memory than map modes. This is due to the difference in how memory is used to determine screen display in the two types of modes. Character modes use one byte of screen memory per pixel, no matter what size the pixel is. It follows that there is a one-to-one correspondence between the pixels on the screen and the locations in screen memory. The bytes stored in memory are the internal codes for the characters to be displayed on the screen. Although the pixels on the screen are organized two dimensionally, the bytes in memory are organized linearly. Referring back to figure 3-2, corresponding to the first row of pixels on the screen are the first twenty bytes in screen memory. Location 0,0 corresponds to the first byte; location 0,19 to the 20th byte. Corresponding to the second row of pixels is the next twenty bytes in screen memory, and so on. As a result, the minimum screen memory needed for a character mode is equal to the number of pixels on the screen.

In map modes the display pixels are generally much smaller than in character modes (BASIC Graphics 3 is an exception). The distinguishing characteristic of map modes is that one, two, or four *bits* in memory determine if a pixel is lit or unlit and, depending on the mode, the color displayed. For example, consider Graphics 8. Graphics 8 uses pixels that are a ½ color clock (one machine cycle) wide by one scan line high and so there are 320 pixels per scan line. A single bit in screen memory determines whether a pixel is on or off. If a pixel is to be on, then its corresponding bit is equal to one; if the pixel is to be off, its corresponding bit is equal to zero. Since there are 320 pixels per scan line, 320 dividing by 8 bits per byte gives 40 bytes of memory needed per scan line in Graphics 8. Since there are 192 scan lines, the minimum screen memory required in Graphics 8 is 7,680 bytes. Graphics 8 is comparatively simple because there are only two colors available, foreground and background. Thus, a single bit suffices to determine a pixel's state. In four color graphics modes, a single bit is not sufficient to choose colors. In these modes - BASIC Modes 3,5,7 and ANTIC Mode 14 - a pair of bits is required to specify a color. Consequently each byte of screen memory encodes four pixels. The four color map modes use larger pixels than Graphics 8 to keep the screen memory requirements within reasonable limits.

Basic Modes 4 and 6 are similar to Graphics 8. In these two color map modes, a bit value of 1 selects the foreground color from color register 0, while a bit value of 0 selects the background color from color register 4. These map modes are especially memory efficient since each byte of screen memory encodes 8 pixels and the pixels are 4 scan lines by 2 color clocks (mode 4) or 2 scan lines by 1 color clock (mode 6).

The ANTIC display modes differ from the BASIC modes (in the 400/800 series) in that they are not available through a simple GRAPHICS command. The BASIC GRAPHICS command causes the operating system to generate an appropriate display list for the

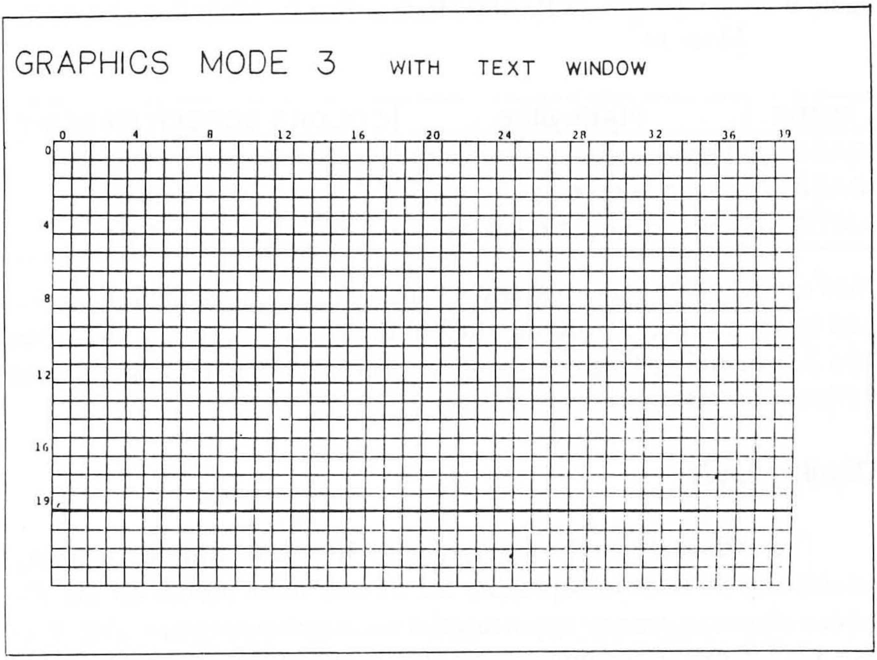


Figure 3-3. Graphics Mode 3 with text window

Graphics mode called. In order to use the ANTIC modes in the 400/800 series, you must construct your own display list. Although it entails more work to use the ANTIC modes, the compensation is that they offer a number of features not available in the BASIC modes. ANTIC Mode 3 is a character mode that allows descenders on lower case letters. ANTIC Modes 4 and 5 are four color character modes. As with the four color map modes discussed earlier, pairs of bits determine the colors used. We defer the more complete description of these modes until we have discussed character sets. ANTIC Mode 14 is a four color map mode with pixels that are one scan line high and one color clock wide. As a result, the vertical resolution is equivalent to BASIC Mode 8, but the horizontal resolution is only half as great.

It is worthwhile to compare BASIC Mode 8, BASIC Mode 7, and ANTIC Mode 14 in order to see the relationship between pixel size, colors available, and screen memory:

Table 3-4. Comparison of BASIC Mode 8, BASIC Mode 7, and ANTIC Mode 14

MODE	PIXEL SIZE	COLORS	SCREEN MEMORY
BASIC 8	1 scan line x ½ color clock	2	7680
BASIC 7	2 scan lines x 1 color clock	4	3840
ANTIC 14	1 scan line x 1 color clock	4	7680

You can see that there are trade offs made between resolution, colors, and screen memory. Consideration of these factors is important when you are planning large or complex programs that may use several different colorful screen displays.

Display Lists

The Atari Home Computer, with its many different display modes and its built in capability for mixing these modes on the TV screen allows you many opportunities for creative programming. You can mix character and map modes almost at will. The way to do this is to write your own custom display list. There are two ways to proceed when creating a display list. First, you can modify one that is accessible from BASIC. Second, you can create your own display list from scratch, store it in memory, and tell the computer to use it.

When planning a custom display list, there are two categories of items to take into account. The first category relates to the overall organization of the display you wish to create. These factors are: the types of modes you wish to use, the colors available, and special features such as scrolling and display list interrupts. The second category of factors to consider are of primary importance in actually constructing the display list. These are: the number of scan lines per mode line, the number of memory bytes per mode line, and the total screen memory needed. Since these factors are essential in planning a display list, Table 3-4 summarizes this information.

Table 3-4. Essential information when planning a display list

MODE NUMBER BASIC	ANTIC	COL/ROW with TEXT WINDOW	COL/ROW no TEXT WINDOW	SCAN LINES per MODE LINE	SCRN RAM per MODE LINE	MINIMUM TOTAL SCRN MEM
0	2	-	40 x 24	8	40	960
-	3	-	40 x 19	10	40	760
-	4	-	40 x 24	8	40	960
-	5	-	40 x 12	16	40	480
1	6	20 x 20	20 x 24	8	20	480
2	7	20 x 10	20 x 12	16	20	240
3	8	40 x 20	40 x 24	8	10	240
4	9	80 x 40	80 x 48	4	10	480
5	10	80 x 40	80 x 48	4	20	960
6	11	160 x 80	160 x 96	2	20	1920
-	12	-	160 x192	1	20	3840
7	13	160 x 80	160 x 96	2	40	3840
-	14	-	160 x 192	1	40	7680
8	15	320 x 160	320 x 192	1	40	7680

Prior to discussing the steps necessary to create a display list, it will be helpful to examine a display list available from BASIC. Figure 3-4 is a Graphics 2 display list.

Antic executes the display list program sixty times each second, once each time a TV frame is drawn. Certain features of display lists generated by the OS are consistent among all the graphics modes. These are: Blank line instructions, LMS instructions, and the JVB instruction. Examine the Graphics 2 display list. You will see that the first three bytes tell ANTIC to display 24 blank scan lines at the top of the screen to allow for TV overscan. The fourth byte of the display list serves a dual function. First, bit six is set. Therefore this byte includes the LMS option. Second, the lower four bits tell ANTIC to display the first mode line of Basic Mode 2. Each instruction in a display list that includes the LMS option must be followed by a two byte address that tells ANTIC where the screen memory is located. These bytes are

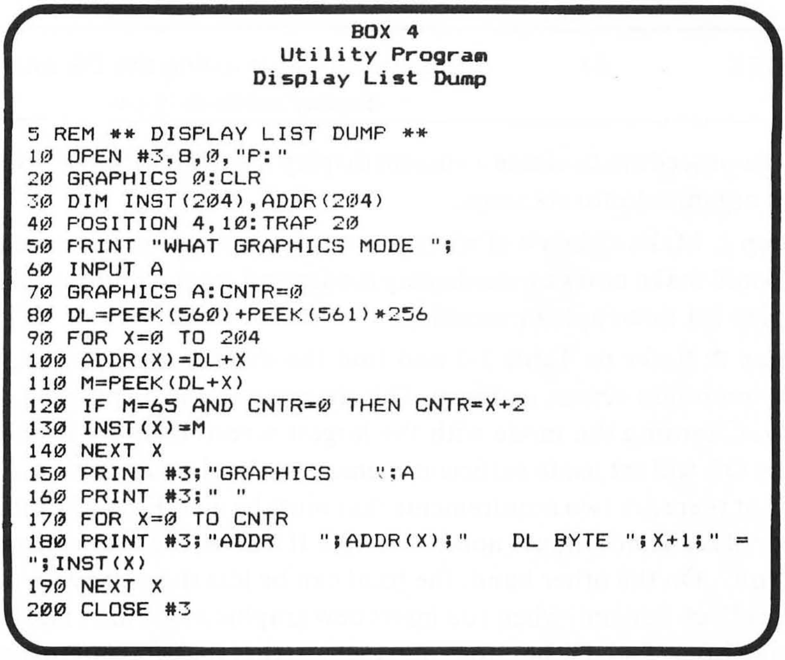
DL Byte	Decimal Value	Binary	
1	112		24
2	112	0111 0000	Blank
3	112		Lines
●	●	●	●
4	71	0100 0111	LMS and GR. 2
5	112		Lo-Byte
6	158		Hi-Byte
●	●	●	●
7	7		
8	7		
9	7		
10	7		
11	7		BASIC
12	7	0000 0111	MODE
13	7		2
14	7		
15	7		
16	7		
17	7		
●	●	●	●
18	65	0100 0001	JVB
19	92		Lo-Byte
20	158		Hi-Byte

Figure 3-4. Graphics display list

written in Lo-Byte/ Hi-Byte order. In this particular display list, which was generated by an Atari with 48 K of memory, the screen memory started at location 112 of page 158. The next eleven bytes in the display list are BASIC Mode 2 instructions. Including byte 4, there are a total of twelve Mode 2 lines, each consisting of 16 scan lines for a total of 192 horizontal scan lines from top to bottom on the TV screen. The last three bytes of the display list are the JVB instruction. 65 is the JVB op code. The next two bytes are the operand, in this case the Lo-Byte/ Hi-Byte of the address of the first byte in the display list.

The addresses following the JVB and LMS op codes allow you to infer how the OS has positioned the display list and screen memory within the computer's memory. According to the JVB instruction, the display list starts at 40540 (158,92). If you count the number of bytes in the display list you will see that the last one is in memory location 40559 (158,111). Checking the LMS instruction, we see that the next byte in memory, 40560 (158,112), is the start of screen memory. In the Atari the OS locates screen memory immediately following the display list. The exact addresses where the OS positions the display list and screen memory are dependent on the amount of system memory and graphics mode.

Box 4 is a short BASIC program that will allow you to print out display lists for each of the BASIC display modes. It is useful to have these printouts for reference when planning a custom display list.



BOX 4
Utility Program
Display List Dump

```
5 REM ** DISPLAY LIST DUMP **
10 OPEN #3,8,0,"P:"
20 GRAPHICS 0:CLR
30 DIM INST(204),ADDR(204)
40 POSITION 4,10:TRAP 20
50 PRINT "WHAT GRAPHICS MODE ";
60 INPUT A
70 GRAPHICS A:CNTR=0
80 DL=PEEK(560)+PEEK(561)*256
90 FOR X=0 TO 204
100 ADDR(X)=DL+X
110 M=PEEK(DL+X)
120 IF M=65 AND CNTR=0 THEN CNTR=X+2
130 INST(X)=M
140 NEXT X
150 PRINT #3;"GRAPHICS   ";A
160 PRINT #3;" "
170 FOR X=0 TO CNTR
180 PRINT #3;"ADDR    ";ADDR(X);"    DL BYTE ";X+1;" =
";INST(X)
190 NEXT X
200 CLOSE #3
```

Box 4.

We have mentioned that there are two ways to proceed in developing your own display list: (1) from within the framework of a display list provided by BASIC, or (2) start from scratch. We shall describe both methods. In each case there are certain memory locations of crucial importance. These are listed in Table 3-5.

Table 3-5. Important memory locations when developing a display list

LABEL	LOCATION	FUNCTION
SDLSTL	560,561	560 Lo-Byte of DL Address 561 Hi-Byte of DL Address
SAVMSC	88,89	88 Lo-Byte of start of screen memory 89 Hi-Byte of start of screen memory
DINDEX	87	Contains the value telling the OS what display mode is in use

The procedure to create a custom display list from within BASIC can be organized into six steps:

Step 1: Make a sketch of what you want to appear on the screen. You should make notes on the display modes and special options such as display list interrupts or scrolling.

Step 2: Refer to Table 3-3 and find the display mode with the largest minimum screen memory. This determines the display list to modify. Choosing the mode with the largest screen memory insures that the OS will set aside sufficient memory to hold your display. At this point there are two requirements that must be met. First, the total number of scan lines should not exceed 192. If it does, the screen image may "roll". On the other hand, the total can be less than 192 with no adverse effect. Second, when you insert new graphic mode lines into an existing display list, it is best to group them so that the total number of bytes per group is a whole multiple of the bytes per mode line in the display list being modified. The best way to keep track of these things is to make a drawing like Figure 3-5.

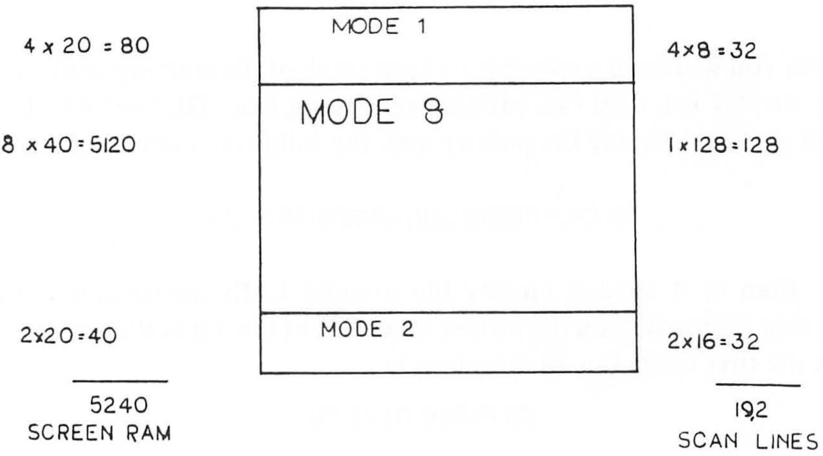


Figure 3-5. Depiction of graphic mode lines

The example in Figure 3-5 modifies a Graphics 8 display list. Each line of Graphics 8 requires 40 bytes of RAM. At the top there are four lines of Mode 1 requiring 80 bytes of screen memory, an integral multiple of 40. Similarly, at the bottom there are two lines of Mode 2, each requiring 20 bytes of screen memory for a total of 40 bytes. Matching up the byte requirements between inserted lines and existing lines is one way to insure that text and graphics will appear where you want them. The reason for all this calculating of byte requirements is that there is potential for conflict between what ANTIC does and what the OS thinks ANTIC is doing. When you start out with a Graphics 8 command, the OS will assume that there are 40 bytes per mode line in screen memory. However, when ANTIC reads the display list and encounters the first Mode 1 instruction it will display only 20 bytes of what the OS thought was a 40 byte line. Including a second line of Mode 1 will keep things synchronized. Later we will see that there is another way to work around this potential conflict by changing SAVMSC, DINDEX and their associated hardware registers.

Step 3: Begin writing your BASIC program with a GRAPHICS command calling the display list you are going to modify. In the example of Figure 3-5 we would have:

```
10 GRAPHICS 8
```

Next you will need a variable to keep track of the starting address of the display list. Call this variable something like "DL" or "START" and peek the display list pointer with the following command:

```
20 DL=PEEK(560)+PEEK(561)*256
```

Step 4: If needed modify the original LMS instruction in the display list to give you the proper mode line at the top of the screen. To get the first mode line of Graphics 1:

```
30 POKE DL+3,70
```

Step 5: Modify the remainder of the display list. Here is where the printout of the original display list is handy because you can count bytes in the original to figure out where to put your new instructions. The Graphics 8 display list is shown in figure 3-6.

First, we want three more Graphics 1 lines at the top of the screen. This is accomplished with:

```
40 POKE DL+6,6:POKE DL+7,6:POKE DL+8,6
```

which will replace the three graphics 8 instructions following the LMS address bytes.

In order to place the Mode 2 lines we count 128 Graphics 8 lines down the display list. This example illustrates something important about Graphics 8 display lists. Recall that the memory scan counter cannot cross a 4K boundary and Graphics 8 needs 7680 bytes of screen memory. Consequently screen memory is broken up into two blocks. ANTIC is sent to the first block of screen memory by the first LMS instruction in address 32851, and is then sent to the second block of screen memory by the second LMS instruction in address 32947. The need to "jump the 4K boundary" occurs only in the Graphics 8 Mode. Care should be exercised that neither the second LMS instruction nor its operand are accidentally clobbered by inserted mode lines. Also the two address bytes must be allowed for in counting where you will insert mode lines near the bottom of the screen. Taking all of this into account we have line 50:

ADDRESS	DL BYTE	INSTRUCTION
32848	1	112
32849	2	112
32850	3	112
32851	4	79
32852	5	80
32853	6	129
32854	7	15
32855	8	15
32856	9	15
32857	10	15
	85 Bytes omitted	
32943	96	15
32944	97	15
32945	98	15
32946	99	15
32947	100	79
32948	101	0
32949	102	144
32950	103	15
32951	104	15
32952	105	15
	60 Bytes omitted	
33013	166	15
33014	167	15
33015	168	66
33016	169	96
33017	170	159
33018	171	2
33019	172	2
33020	173	2
33021	174	65
33022	175	80
33023	176	128

Figure 3-6. Graphics 8 display list

```
50 POKE DL+139,7:POKE DL+140,7
```

Step 6: Once the changes in mode lines are complete, finish off with a JVB followed by the Lo-Byte/Hi-Byte of the return address:

```
60 POKE DL+141,65
70 POKE DL+142,PEEK(560)
80 POKE DL+143,PEEK(561)
```

Note that after these values are put into the display list any bytes remaining from the original display list will not be used.

Displaying characters or graphics on a modified screen involves telling the OS how to interpret the data in screen memory. It would not work to tell the computer to display a character in Mode 1 if the computer thinks it is using Graphics 8. The register DINDEX (location 87) tells the OS which display mode is in use. Accordingly, to print in either the Mode 1 or Mode 2 portions of the above display list it is first necessary to POKE 87,1 or POKE 87,2 respectively. In all cases the number POKEd is the BASIC display mode number.

A second complication arises when the OS positions text or graphics on the screen. This occurs because positioning is done by counting bytes from the start of screen memory. The OS does its calculation on the basis of the size of screen memory associated with the display mode value stored in location 87. With a custom display list, it is possible for total screen memory to be considerably longer than the mode the OS is using. This disparity can cause the dreaded "cursor out of range" error message as well as trouble positioning material on the screen. Fortunately the cure for this problem is fairly simple. Before creating a display on the screen, change the pointer to the top of screen memory (SAVMSC) to coincide with the start of the mode section where you want the display to appear. This means that you temporarily treat the upper left hand pixel of that mode as being position 0,0 and place your display within that mode section in the usual manner. This technique also eliminates the trial and error method of positioning things on the screen.

For example, suppose we had printed something in the Graphics 1 section of the screen and now wanted to display a geometric design in the Graphics 8 section. The program would read as follows:

1. Tell the OS what mode to use;

```
POKE 87,8
```

2. Locate current top of screen address;

```
TPSCRN=PEEK(88)+PEEK(89)*256
```

3. Next, offset the variable TPSCRN by the number of memory bytes for the Mode 1 lines plus 1 (4 lines * 20 bytes per line + 1 = 81);

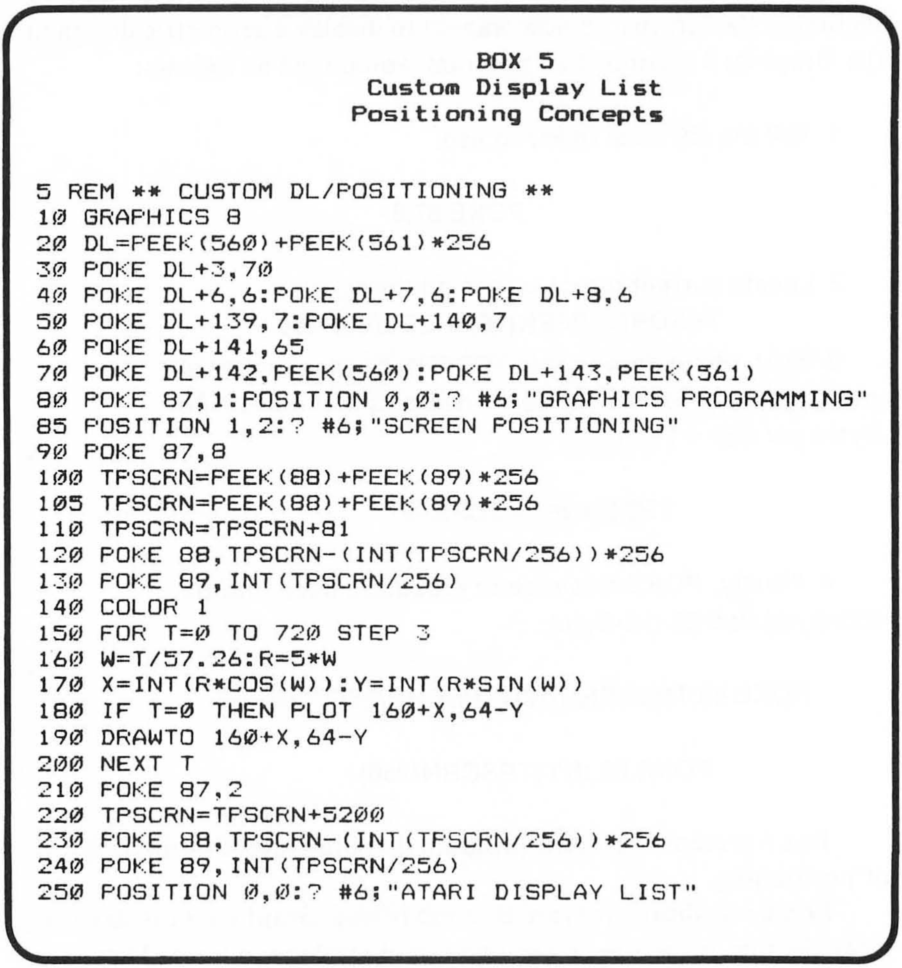
```
TPSCRN=TPSCRN+81
```

4. Finally, POKE this memory location back into 88 (Lo-Byte) and 89 (Hi-Byte);

```
POKE 88,TPSCRN-(INT(TPSCRN/256)*256)

POKE 89,INT(TPSCRN/256)
```

Box 5 presents a BASIC program that illustrates this method of positioning.

Box 6 is a short program that modifies a Graphics 8 display list. Near the top of the screen are two Graphics 2 mode lines. These are followed by some Graphics 8 lines and then some Graphics mode 1 lines. When you type in and run this program it will give you a feeling for the difference in the sizes of mode lines made up of 8 and 16 scan lines. More importantly, however, this program can serve as the focal point for some important exercises: (a) figure out the number of mode 8 scan lines at the top of the screen by writing out the first dozen or so bytes of the display list; (b) move the Graphics 1 mode lines around the screen to get a feeling for the placement on the screen and the position of the instruction in the display list; (c) deliberately try to overwrite the



BOX 5
Custom Display List
Positioning Concepts

```
5 REM ** CUSTOM DL/POSITIONING **
10 GRAPHICS 8
20 DL=PEEK(560)+PEEK(561)*256
30 POKE DL+3,70
40 POKE DL+6,6:POKE DL+7,6:POKE DL+8,6
50 POKE DL+139,7:POKE DL+140,7
60 POKE DL+141,65
70 POKE DL+142,PEEK(560):POKE DL+143,PEEK(561)
80 POKE 87,1:POSITION 0,0:? #6;"GRAPHICS PROGRAMMING"
85 POSITION 1,2:? #6;"SCREEN POSITIONING"
90 POKE 87,8
100 TPSCRN=PEEK(88)+PEEK(89)*256
105 TPSCRN=PEEK(88)+PEEK(89)*256
110 TPSCRN=TPSCRN+81
120 POKE 88,TPSCRN-(INT(TPSCRN/256))*256
130 POKE 89,INT(TPSCRN/256)
140 COLOR 1
150 FOR T=0 TO 720 STEP 3
160 W=T/57.26:R=5*W
170 X=INT(R*COS(W)):Y=INT(R*SIN(W))
180 IF T=0 THEN PLOT 160+X,64-Y
190 DRAWTO 160+X,64-Y
200 NEXT T
210 POKE 87,2
220 TPSCRN=TPSCRN+5200
230 POKE 88,TPSCRN-(INT(TPSCRN/256))*256
240 POKE 89,INT(TPSCRN/256)
250 POSITION 0,0:? #6;"ATARI DISPLAY LIST"
```

Box 5. Custom Display List Positioning Concepts

second LMS byte in the Graphics 8 display list. See what happens! Does it affect PLOT's and DRAWTO's? Deliberately creating programs with 'bugs' and studying the results can be a great help in later program debugging; (d) change the address bytes for the first block of screen memory to page zero of memory. One thing you should see is the real time clock in action.

BOX 6

```
5 REM ** MODIFIED DISPLAY LIST **
10 GRAPHICS 8
20 DL=PEEK(560)+PEEK(561)*256
30 POKE DL+10,7:POKE DL+11,7
40 POKE DL+24,6:POKE DL+25,6
50 POKE DL+122,6:POKE DL+123,6
60 POKE DL+136,65
70 POKE DL+137,PEEK(560)
80 POKE DL+138,PEEK(561)
```

Box 6. Modified Graphics 8 Display List

Creating your own display list from scratch can seem easier than modifying a display list provided by BASIC, because you are starting with a clean slate. The first concern is where to store the display list and its screen memory so that they won't be overwritten by BASIC. The OS solves this problem by storing them between the addresses pointed to by MEMTOP (741,742) and RAMTOP (106). MEMTOP is the pointer to the last free byte available to BASIC. RAMTOP points to the dividing line between RAM and the high memory address used for the BASIC cartridge, GTIA, POKEY, and so on. The value in RAMTOP is always expressed in pages (multiples of 256), and in a 48K Atari is 160, corresponding to memory address 40960.

Providing a place in memory that is safe from being overwritten by BASIC is a problem that occurs whenever you want to use special features such as redefined character sets, player/missile graphics, machine language subroutines, or when creating your own display list. There are several solutions to the problem. One solution we shall frequently use is to lower RAMTOP by a BASIC statement such as:

```
POKE(106),PEEK(106)- # of pages to reserve
```

or its machine language equivalent which is:

```
LDA RAMTOP minus the number of pages to reserve

STA 106
```

The space in memory between the old RAMTOP value and the new one is essentially safe. Usually, if you are working from BASIC, it is a good idea to follow the change in RAMTOP with a GRAPHICS or a CLOSE #6,OPEN#6,8,0,"S:" sequence. This will update MEMTOP and insure that any display list created later in the program by the OS will not override data in your reserved area. You should be aware that any clear screen commands or text window scrolling that occurs will clear some memory beyond RAMTOP - up to 800 bytes for text window scrolling. Therefore, to avoid memory conflicts it is fortuitous to leave a buffer zone between RAMTOP and your display list, or other special programs.

As with modifying a BASIC display list, writing a display list from scratch should be approached in a sequential manner.

Step 1: Figure out how you want to present the screen. Be sure to: (a) Allow for 24 blank lines at the top of the screen. (b) Plan what mode lines you are using and their position on the screen. (c) Take into account special options. For example, with scrolling you may want every mode line used to have the LMS option. Figure 3-7 is a rather complicated example that we have provided simply to give you a feeling for the calculations involved in this and succeeding steps.

As a practical example we will construct an ANTIC Mode 4 display list. After making a drawing similar to Figure 3-7 and taking into account special option, make a rough display list such as this:

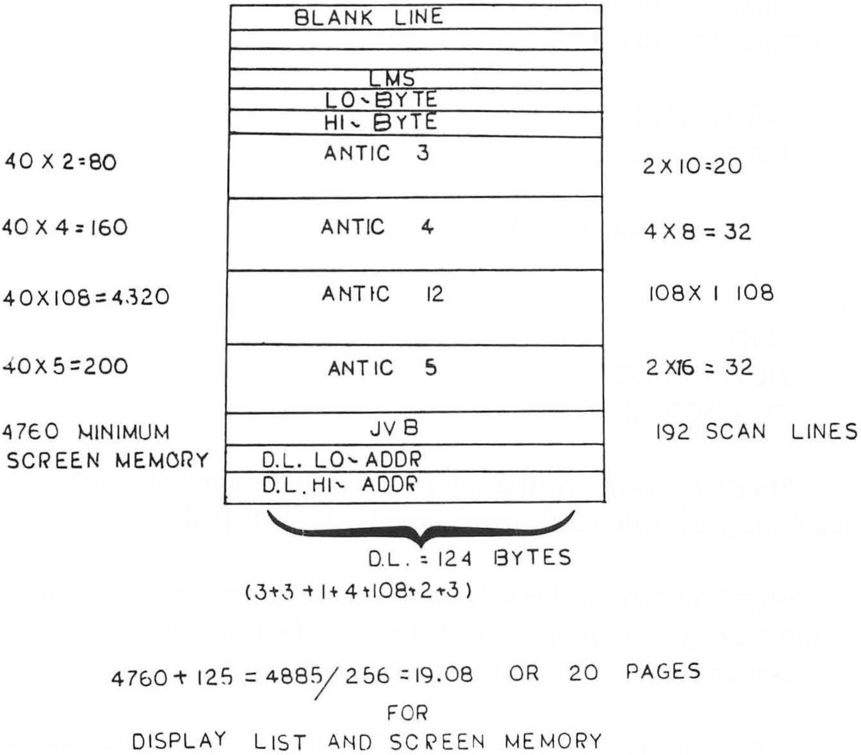


Figure 3-7. Display List and Screen Memory

```
     112
     112  24 blank lines
     112

     LMS
  SCRN MEM ADDR-LO
  SCRN MEM ADDR-HI

    MODE LINE
    MODE LINE
       .
       .    23 Mode 4 lines
       .
       .
     JVB
    DL ADDR-LO
    DL ADDR-HI
```

Step 2: From the draft display list, count the number of bytes that the display list will use. In our example, 3+3+23+3=32.

Step 3: Determine the amount of screen memory needed from the number of bytes required per mode line. The total number of bytes equals the amount of screen memory. (40 bytes * 24 mode lines = 960)

Step 4: The results of steps 2 and 3 determine the number of pages needed for both the display list and screen memory (32+960=992/256 = 3.8 or 4 pages). If you anticipate using a clear screen command or text window scrolling later in the program, leave a buffer between RAMTOP and your display list.

Step 5: Decide upon the relative position, in your reserved area, of the display list and screen memory. Although the OS locates screen memory immediately after the display list, this is not mandatory.

In the example of Box 7, we construct an ANTIC Mode 4 display list for the 400/800 series. The display list itself needs 32 bytes. The screen memory needed is 960 bytes. If we put the display list on its own page (wasteful of memory!), allow four pages for screen memory, and allow three pages as a buffer above RAMTOP, we can plan the positioning as:

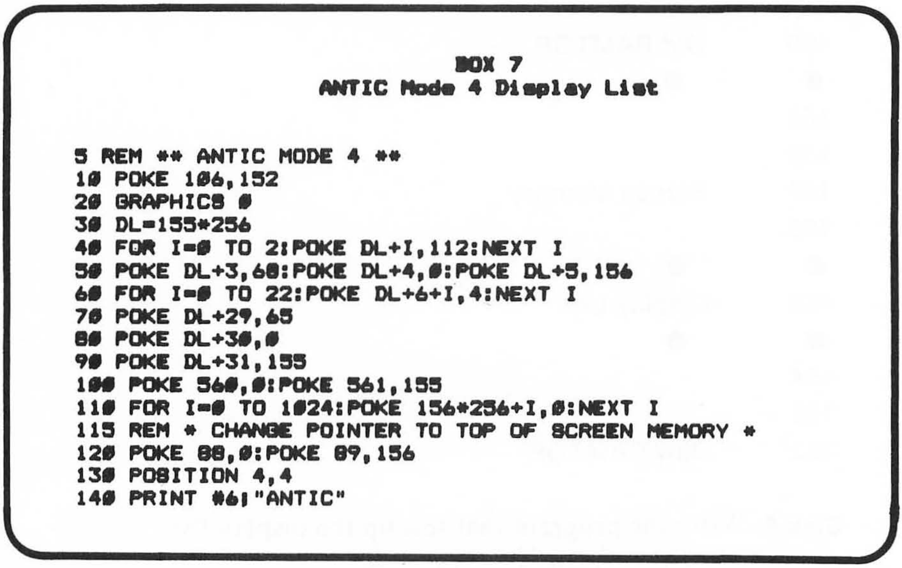
PAGE	CONTENTS
160	Old RAMTOP
●	●
159	
158	
157	Screen Memory
156	
●	●
155	Display List
●	●
154	
153	
152	New RAMTOP

Step 6: Write the program that sets up the display list.

The program in Box 7 is written in this particular form for pedagogical reasons. First of all, Lines 10,30,50,90, and 100 contain explicit references to page numbers so that you can clearly see the correspondence between the planning steps, such as the chart in Step 5, and the actual program. Because of the explicit page references, the program as written is for an Atari with 48K. It needs modification to be transportable to 16K or 32K machines. The modifications are simple. Replace Line 10 with:

```
POKE 106,PEEK(106)-8
```

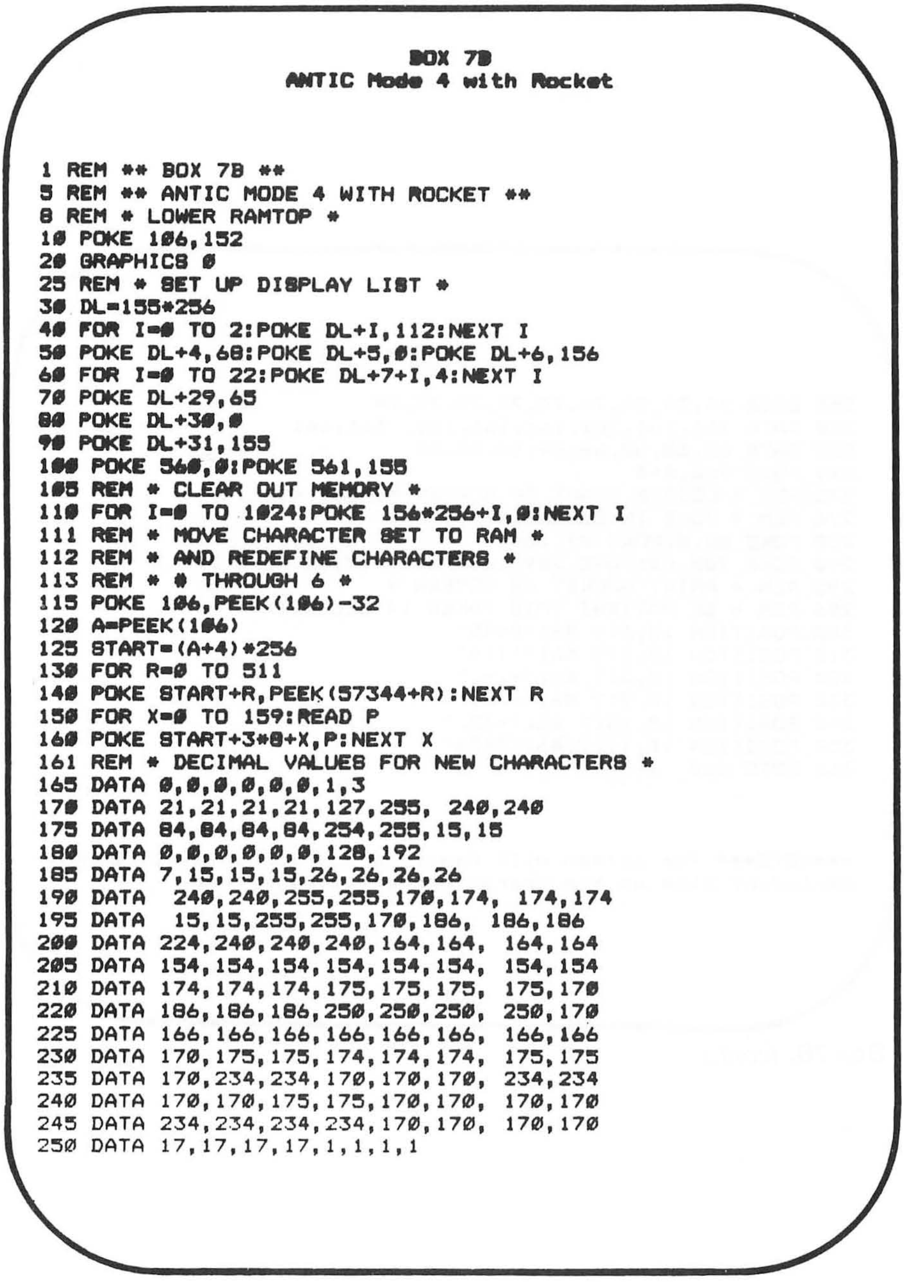
Also replace each number 155 with (PEEK(106)+3) and each number 156 with (PEEK(106)+4).



```
BOX 7
ANTIC Mode 4 Display List

5 REM ** ANTIC MODE 4 **
10 POKE 106,152
20 GRAPHICS 0
30 DL=155*256
40 FOR I=0 TO 2:POKE DL+I,112:NEXT I
50 POKE DL+3,68:POKE DL+4,0:POKE DL+5,156
60 FOR I=0 TO 22:POKE DL+6+I,4:NEXT I
70 POKE DL+29,65
80 POKE DL+30,0
90 POKE DL+31,155
100 POKE 560,0:POKE 561,155
110 FOR I=0 TO 1024:POKE 156*256+I,0:NEXT I
115 REM * CHANGE POINTER TO TOP OF SCREEN MEMORY *
120 POKE 88,0:POKE 89,156
130 POSITION 4,4
140 PRINT #6;"ANTIC"
```

Box 7. ANTIC Mode 4 Display List

The program also makes another point about reserving a safe place by lowering RAMTOP. When you run the program you will see that included in the space we have reserved was the screen memory used by the computer when the program was typed in! Line 110 allows you the fun of watching this section of screen memory being cleaned out! Two conclusions can be drawn. First, before using a section of RAM as screen memory, you may want to clear it out by filling it with zeros. Second, BASIC is very S-L-O-W at this job. In the next chapter we'll write a machine language routine for this purpose.



```
BOX 7B
ANTIC Mode 4 with Rocket

1 REM ** BOX 7B **
5 REM ** ANTIC MODE 4 WITH ROCKET **
8 REM * LOWER RAMTOP *
10 POKE 106,152
20 GRAPHICS 0
25 REM * SET UP DISPLAY LIST *
30 DL=155*256
40 FOR I=0 TO 2:POKE DL+I,112:NEXT I
50 POKE DL+4,68:POKE DL+5,0:POKE DL+6,156
60 FOR I=0 TO 22:POKE DL+7+I,4:NEXT I
70 POKE DL+29,65
80 POKE DL+30,0
90 POKE DL+31,155
100 POKE 560,0:POKE 561,155
105 REM * CLEAR OUT MEMORY *
110 FOR I=0 TO 1024:POKE 156*256+I,0:NEXT I
111 REM * MOVE CHARACTER SET TO RAM *
112 REM * AND REDEFINE CHARACTERS *
113 REM * 0 THROUGH 6 *
115 POKE 106,PEEK(106)-32
120 A=PEEK(106)
125 START=(A+4)*256
130 FOR R=0 TO 511
140 POKE START+R,PEEK(57344+R):NEXT R
150 FOR X=0 TO 159:READ P
160 POKE START+3*8+X,P:NEXT X
161 REM * DECIMAL VALUES FOR NEW CHARACTERS *
165 DATA 0,0,0,0,0,0,1,3
170 DATA 21,21,21,21,127,255, 240,240
175 DATA 84,84,84,84,254,255,15,15
180 DATA 0,0,0,0,0,0,128,192
185 DATA 7,15,15,15,26,26,26,26
190 DATA  240,240,255,255,170,174, 174,174
195 DATA  15,15,255,255,170,186, 186,186
200 DATA 224,240,240,240,164,164, 164,164
205 DATA 154,154,154,154,154,154, 154,154
210 DATA 174,174,174,175,175,175, 175,170
220 DATA 186,186,186,250,250,250, 250,170
225 DATA 166,166,166,166,166,166, 166,166
230 DATA 170,175,175,174,174,174, 175,175
235 DATA 170,234,234,170,170,170, 234,234
240 DATA 170,170,175,175,170,170, 170,170
245 DATA 234,234,234,234,170,170, 170,170
250 DATA 17,17,17,17,1,1,1,1
```

Box 7B. ANTIC Mode 4 with Rocket

```
255 DATA 74,74,74,74,74,74,74,74,74
260 DATA 161,161,161,161,161,161, 161,161
265 DATA 68,68,68,68,64,64,64,64
270 POKE 756,A+4
275 REM * LOCATE START OF SCREEN MEMORY *
276 REM * POKE IN COLORS *
280 POKE 88,0:POKE 89,156
290 POKE 708,60:POKE 709,168:POKE 710,88:POKE 712,8
295 REM * PRINT ROCKET ON SCREEN *
296 REM * BE PATIENT THIS TAKES 14 SECONDS *
300 POSITION 10,6:? #6;"#$%&"
310 POSITION 10,7:? #6;"'()*"
320 POSITION 10,8:? #6;"+,-."
330 POSITION 10,9:? #6;"+/0."
340 POSITION 10,10:? #6;"+12."
350 POSITION 10,11:? #6;"3456"
360 GOTO 360
```

NOTE The screen will remain black for a fairly long period of time as the character set is redefined.

Box 7B. *(cont.)*

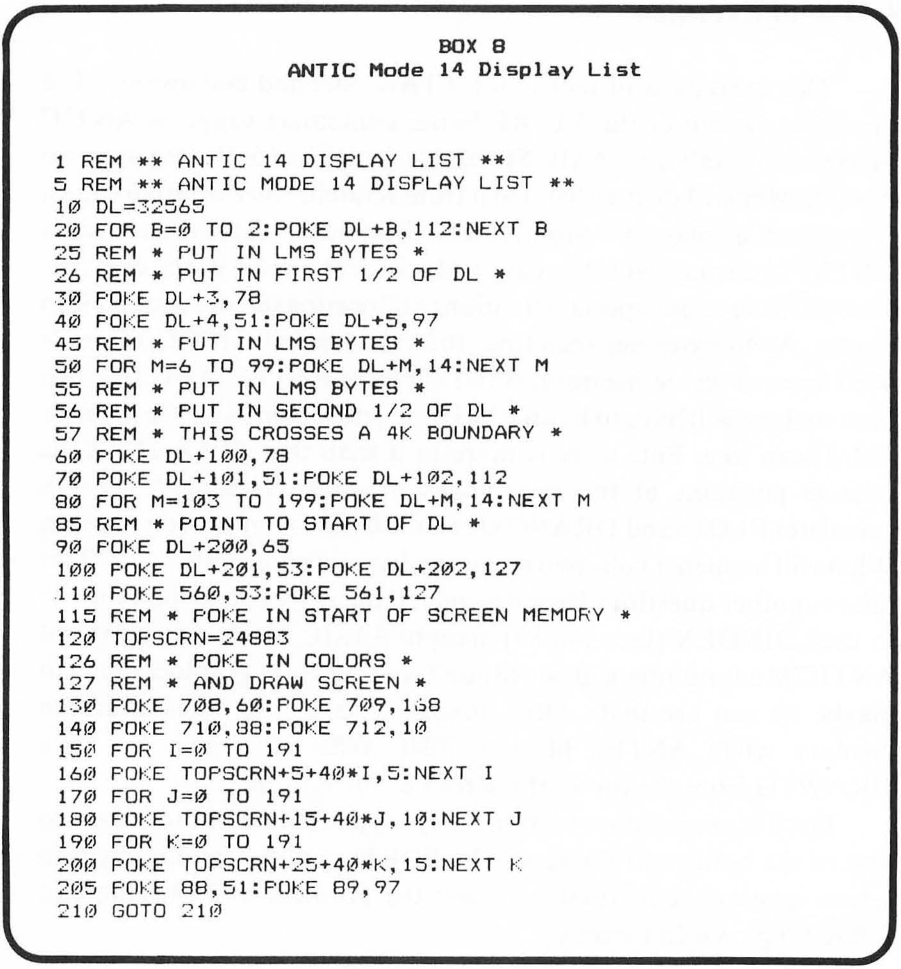
A Useful Exercise

This exercise is primarily for ATARI 400 and 800 owners. The operating system of the XL/XE Series computers supports ANTIC Mode 14 through the BASIC Statement Graphics 15. Write your own ANTIC Mode 14 display list. Do it from scratch, NOT by modifying a Graphics 8 display list. Modifying a Graphics 8 display list into an ANTIC 14 display list is too easy and misses the point of the exercise. You will have to pay special attention to allocating screen memory into blocks. At 40 bytes per scan line, 102 scan lines of ANTIC 14 needs 4080 bytes of screen memory. A full screen of ANTIC 14 has 192 scan lines so there will have to be an LMS instruction somewhere before the 103rd scan line. But, there is more to it than that. What about the relative positions of the two blocks of screen memory? The OS calculates PLOTs and DRAWTOs on the basis of screen memory size. What will happen if your two memory blocks are not contiguous? This raises another question: How do you tell the OS what graphics mode to use? DINDEX (location 87) accepts BASIC Mode numbers, not ANTIC Mode numbers. Basic Mode 7 is a four color graphics mode so maybe we can use that.... But, BASIC 7 uses 3840 bytes of screen memory while ANTIC 14 uses 7680. What does that do to a DRAWTO from the top of the screen to the bottom?

Box 8 is one solution to the display list problem that shows how to plot to the bottom of the screen by POKEing numbers directly into screen memory, but doesn't answer the problem of PLOTing and DRAWing on a full screen.

Page Flipping

From the knowledge that you have accumulated at this point, the concept behind page flipping should be easy to grasp and almost as easy to implement. The intent of page flipping is to reserve several different sections of RAM for screen memory, each with its own display, and 'flip' from one section to another simply by changing the address bytes of an LMS instruction. One can flip whole screens or parts of screens depending on where the LMS instruction is placed in the display list. This technique is useful for animation or providing a



BOX 8
ANTIC Mode 14 Display List

```
1 REM ** ANTIC 14 DISPLAY LIST **
5 REM ** ANTIC MODE 14 DISPLAY LIST **
10 DL=32565
20 FOR B=0 TO 2:POKE DL+B,112:NEXT B
25 REM * PUT IN LMS BYTES *
26 REM * PUT IN FIRST 1/2 OF DL *
30 POKE DL+3,78
40 POKE DL+4,51:POKE DL+5,97
45 REM * PUT IN LMS BYTES *
50 FOR M=6 TO 99:POKE DL+M,14:NEXT M
55 REM * PUT IN LMS BYTES *
56 REM * PUT IN SECOND 1/2 OF DL *
57 REM * THIS CROSSES A 4K BOUNDARY *
60 POKE DL+100,78
70 POKE DL+101,51:POKE DL+102,112
80 FOR M=103 TO 199:POKE DL+M,14:NEXT M
85 REM * POINT TO START OF DL *
90 POKE DL+200,65
100 POKE DL+201,53:POKE DL+202,127
110 POKE 560,53:POKE 561,127
115 REM * POKE IN START OF SCREEN MEMORY *
120 TOPSCRN=24883
126 REM * POKE IN COLORS *
127 REM * AND DRAW SCREEN *
130 POKE 708,60:POKE 709,168
140 POKE 710,88:POKE 712,10
150 FOR I=0 TO 191
160 POKE TOPSCRN+5+40*I,5:NEXT I
170 FOR J=0 TO 191
180 POKE TOPSCRN+15+40*J,10:NEXT J
190 FOR K=0 TO 191
200 POKE TOPSCRN+25+40*K,15:NEXT K
205 POKE 88,51:POKE 89,97
210 GOTO 210
```

Box 8. ANTIC Mode 14 Display List

variety of backgrounds upon which player/missile action can take place.

There are a few things to consider concerning page flipping. First, every screen uses memory even when it is not being displayed. Second, you probably wouldn't use it with Graphics 8. At nearly 8K of RAM per screen you can use up memory in a hurry! Consequently, page flipping is used most often with the more memory efficient character modes. Third, transitions between screens occur most smoothly if the LMS address bytes are changed during the vertical blank. We've included a program in Box 9-just to take some of the mystery out of the process! This program uses Graphics 2 and flips screen memory from page zero to the BASIC cartridge.

BOX 9

```
5 REM ** PAGE FLIPPING **
10 GRAPHICS 2
20 DL=PEEK(560)+PEEK(561)*256
30 POKE DL+4,0:POKE DL+5,0
40 FOR I=1 TO 100:NEXT I
50 POKE DL+5,192
60 FOR I=1 TO 100:NEXT I
70 GOTO 30
```

Box 9. Page Flipping

An additional comment is appropriate here. Many other home computers set aside a limited number of static blocks of RAM for screen memory. In principle, with the Atari Home Computer you can use any section of RAM as screen memory. Flexibility such as this allows you more room for creative approaches to programming. As an example, one intriguing idea is that it is possible to store both your display list or screen memory in strings, allowing you to use Atari BASIC's string handling routines to change displays.

Color

Another facet that distinguishes the Atari Home Computer from other popular computer systems is the greater number of colors available. One reason for the larger color selection is that in an Atari one can choose a luminance and a hue to produce a color rather than simply specify just a color number. Luminance can be thought of as regulating the intensity of the TV's electron beam which in turn produces variations in brightness thereby permitting a variety of shades of the 16 basic hues. There are eight choices of luminance and sixteen hues which means that, in principle, there are 128 different color choices. In actuality, some hue-luminance combinations may look pretty much the same.

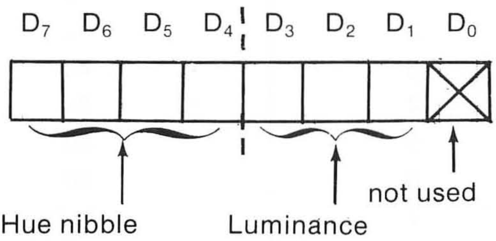
The Atari Home Computer gives you more than extra colors with which to work. It gives you more options as to where and how to display color. This flexibility is provided, in part, by nine color registers that exist in two incarnations: as OS shadow registers and as hardware registers. The colors you see while a program is running are generated from information stored in the hardware registers. During each vertical blank, the OS updates the hardware register values using the data stored in the shadow registers. A crisp transition is insured when a color is changed by a BASIC SETCOLOR command, or a POKE to a shadow register because this transition actually occurs when the screen is blank. Besides giving crisp color changes, the existence of two complete sets of color registers is crucial to the implementation of display list interrupt color changes.

Four of the nine color registers are devoted to players. The remaining five registers are used with playfield graphics. The shadow and hardware addresses of the color registers are listed in Table 3-6. The hardware addresses are listed in Hi-Byte/Lo-Byte form, as well as decimal, for convenience when you are writing machine language routines.

Table 3-6. Color register addresses

FUNCTION	OPERATING SYSTEM		HARDWARE			
	LABEL	ADDRESS	LABEL	ADDRESS	HI-BYTE	LO-BYTE
Player 0	PCOLR0	704	COLPM0	53266	208	18
Player 1	PCLOR1	705	COLPM1	53267	208	19
Player 2	PCOLR2	706	COLPM2	53268	208	20
Player 3	PCOLR3	707	COLPM3	53269	208	21
Playfield 0	COLOR0	708	COLPF0	53270	208	22
Playfield 1	COLOR1	709	COLPF1	53271	208	23
Playfield 2	COLOR2	710	COLPF2	53272	208	24
Playfield 3	COLOR3	711	COLPF3	53273	208	25
Backround	COLOR4	712	COLBK	53274	208	26

By now you realize that all control registers in a computer system perform different functions according to the bit pattern that has been set. The color registers are no exception to this general rule. The color register format is:



Taken as a group of four bits, the hue nibble can have any value between 0 and 15. Corresponding to these numbers are the hues listed in Table 3-7. To POKE, or store a hue value in the left four bits of an 8 bit color register it is first necessary to multiply the decimal value (0 to 15) by 16 (see Table 3-7). Luminance is determined by bits D_1 to D_3 of the color register. Since bit D_0 is not used, luminance is effectively determined by the even numbers 0 through 14 with fourteen being the highest luminance and therefore the brightest. To set these bits in a color register simply add the luminance value to the appropriate hue number:

COLOR NUMBER = (VALUE IN COLUMN 3) + LUMINANCE

Upon power up the operating system sets default colors in the playfield registers 708 through 712. The player color registers 704 through 707 are all set to zero (black). Of course, the playfield registers can be changed with the BASIC SETCOLOR command while player color registers must be changed with a POKE or a machine language store instruction. Table 3-8 lists all display modes, the number of colors available and the default colors. Tables 3-8 and 3-4 are useful for planning custom display lists.

Table 3-7. Hues

HUE	NIBBLE VALUE	POKE OR STORE VALUE
Black	0	0
Rust	1	16
Red-orange	2	32
Dark-orange	3	48
Red	4	64
Lavender	5	80
Light purple	6	96
Purple blue	7	112
Medium blue	8	128
Dark blue	9	144
Blue grey	10	160
Olive green	11	176
Green	12	192
Yellow green	13	208
Orange green	14	224
Orange	15	240

Table 3-8. Display modes with available number of colors and default colors

NUMBER OF COLORS	DISPLAY MODES	DEFAULT COLORS	SHADOW REGISTERS	NOTES
1 Hue	BASIC 0,8	Light Blue	709	Regist. determ.
2 Luminances	ANTIC 3	Dark blue	710	Background
		Black	712	Border
TWO	BASIC 4,6	Orange	708	COLOR 1
COLORS	ANTIC 12	Black	712	Background
				(COLOR 0)
FOUR	BASIC 4,6	Orange	708	COLOR 1
	3,5,7	Lt Green	709	COLOR 2
COLORS		Blue	710	COLOR 3
	ANTIC 14	Black	712	Background
				(COLOR 0)
	BASIC 1,2	Orange	708	BASIC 1&2 color
FIVE		Lt Green	709	
	ANTIC 4,5	Blue	710	
COLORS		Red	711	
		Black	712	

GTIA Modes

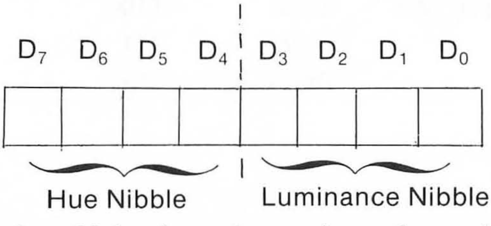
Graphics Modes 9,10 and 11 are enhancements of Graphics 8 that give extended color choices. Briefly, Graphics 9 allows 16 luminances of one hue; Graphics 10 allows the selection of nine colors; Graphics 11 allows 16 hues with one luminance. These Modes are referred to as GTIA Modes because their appearance on the screen is determined, not by an instruction in ANTIC's display list, but by the setting of bits D_6 and D_7 in the GTIA hardware register PRIOR (53275).

Bit: D_7	D_6	Mode
0	0	Determined by ANTIC's DL
0	1	Mode 9
1	0	Mode 10
1	1	Mode 11

The remaining bits, D_0 to D_5, of PRIOR are used in player/missile graphics and their function will be described later in this chapter.

The display list used by ANTIC with the GTIA Modes is a full screen Graphics 8 display list. The difference between the GTIA Modes, individually and in comparison with Graphics 8 is how the data in screen memory is used. In Modes 9,10 and 11, GTIA uses four bits of data for each pixel. Using four bits increases the information that can be transferred from memory to the screen. However, to keep memory requirements within reasonable limits something had to be given up, and in this case it is horizontal resolution. The pixels in the GTIA Modes are one scan line high and two color clocks (four machine cycles) wide. Consequently, the display resolution is 80 by 192.

Now, let's see how these modes work. Recall that a color register has the following structure:



In Graphics 9, which gives the option of one hue and sixteen luminances, the hue nibble of color register 712 (hardware equivalent 53274) is fixed and the luminance value can be changed. This mode differs from all other display modes in that all bits in the luminance nibble are used. From BASIC the luminance is changed with a COLOR N statement, where N is a number from 0 to 15. If you are working from machine language, however, it is sometimes useful to have a memory location in which to store a value that will determine the color displayed. The following demonstration program illustrates

```
                    BOX 10
5 REM ** POKEING IN COLORS **
10 GRAPHICS 9
20 POKE 712,96
30 FOR J=0 TO 3:FOR I=0 TO 15
40 COLOR I
50 PLOT I+16*J,0:DRAWTO I+16*J,30
60 NEXT I:NEXT J
70 FOR J=0 TO 3:FOR I=0 TO 15
80 POKE 200,I
90 PLOT I+16*J,40:DRAWTO I+16*J,70
100 NEXT I:NEXT J
110 GOTO 110
```

Box 10. POKEing in Colors

that POKEing the numbers 0 to 15 into memory address 200 achieves the same effect as COLOR N.

When setting the hue values in color register 712 for Mode 9, you want to be sure that bits D_0 - D_3 are left as zeros. From BASIC that is handled automatically with the command SETCOLOR 4,HUE 0. When using a POKE or a machine language store, you will want to check the bit pattern of the number being stored. The reason for this is based on how the value in the color register is combined with the pixel data in screen memory to get the final display color number. The display color number is arrived at by a logical ORing of the value in register 712 with pixel data. For example:

Register 712	1 1 0 1 0 0 0 0	Hue 13 Dk Green
Pixel Data	0 1 1 0	Luminance 6
	1 1 0 1 0 1 1 0	Display color #

But suppose the value stored in 712 inadvertently had some extra bits:

register 712	1 1 0 1 1 1 0 1	Hue 13 Dk Green
Pixel Data	0 1 1 0	Luminance 6
	1 1 0 1 1 1 1 1	Display Color #

Because of the way a logical OR works (see Box 3), the final luminance value is 15, not 6 as originally desired.

Graphics 11 works analogously to Graphics 9 except that now the luminance value is taken from Bits D_1 - D_3 of color register 712. Since D_0 is not used, there are only eight luminances. The hue value used in plotting is specified with a COLOR N, or POKE 200,N statement, where N is a number from 0 to 15. Table 3-7 lists colors and their corresponding number. Again the final color number is obtained by a logical ORing of the hardware register and the pixel data. This time, the color register should be set up with zeros in the left hand nibble so that the ORing doesn't modify the final color data.

Graphics 10 makes use of all nine color registers 704 - 712 (53266 -53274). The color number is derived in the usual manner:

```
COLOR NO. = HUE*16+LUMINANCE
```

Registers 708 through 712 can be set with either the SETCOLOR command or a POKE. Color in registers 704 through 708 must be set with POKE statements. Colors to PLOT or DRAW with can be chosen with COLOR N or POKE 200,N. However, now N is restricted to 0 to 8. Zero selects 704, one selects 705, and so on. Numbers from 9 to 15 will select one of the lower value color registers.

The GTIA Graphics Modes will run very much the same way as other display modes. This means that you can use standard graphics commands, player/missiles, and the full set of ANTIC options.

A Digression

When one gets involved in programming in BASIC or another higher level language it is easy to lose sight of what's going on at the machine level when a sequence such as,

```
10 COLOR 3
20 PLOT 0,0:DR. 40,40
```

is executed. Suppose you are working in Graphics 10. Then the above two lines will cause the execution of a number of subroutines that will store the bit pattern 0101 into screen memory in such a way that when the screen memory is accessed by ANTIC, a colored line is displayed diagonally on the screen. In the other display modes how the data in screen memory is interpreted differs from the GTIA Modes but the idea is the same. Screen memory contains information that is read and interpreted by ANTIC and GTIA. At this point we would like to remind you that there is absolutely no reason why you have to rely entirely on BASIC commands such as PRINT, PLOT, DRAWTO, and the OS to place data into screen memory. The two programs in Box 11 illustrate this point. Program A draws a diagonal line with PLOT and DRAWTO. Program B uses a FOR NEXT Loop to put the data that generates the line directly into screen memory.

If you think back to the exercise we proposed with the ANTIC 14 display list, there is a problem using the OS's routines for PLOT and DRAWTO. POKEing 87,7 gives you four color graphics, but limits you to using only half the screen. The above discussion provides a clue to one approach to using ANTIC display modes; write routines that place display data directly into screen memory.

BOX 11

```
10 GRAPHICS 10
20 POKE 704,144:POKE 707,88
30 COLOR 3
40 PLOT 0,0:DRAWTO 40,40
50 GOTO 50
```

Box 11. Method of Displaying

BOX 11B

```
10 GRAPHICS 10
20 POKE 704,144:POKE 707,88
30 START=PEEK(88)+PEEK(89)*256
40 FOR I=0 TO 39
50 POKE START+41*I,3:NEXT I
60 GOTO 60
```

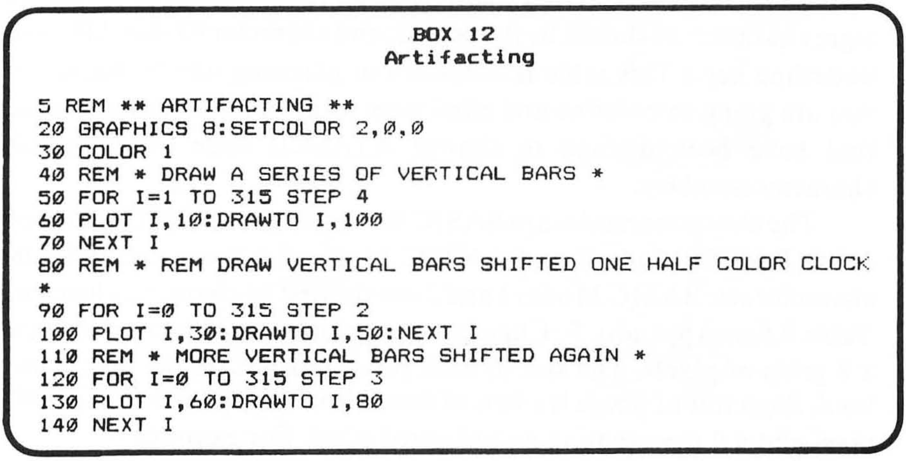
Box 11B. Method of Displaying

Artifacting

Artifacting is a method of putting color on the TV screen that depends on three things: (1) how the TV produces color; (2) how the human eye interprets combinations of color; and (3) the two to one relationship between machine cycles and color clocks in the display modes BASIC 0 , 8 and ANTIC 3.

To produce color, the inside of a color television screen is coated with an array of dots that glow in red, green, and blue when struck by electrons. By controlling the brightness of each dot, it is possible to produce any desired color. At normal viewing distances, the dots are too small for the human eye to perceive individually and so the colors appear to merge into a single image.

The signal sent to a color television set is called a composite video signal. It consists of horizontal and vertical synchronizing pulses, brightness information (luminance), and a 3.58 MHz "subcarrier" that contains color information (chrominance). This 3.58 MHz frequency is a standard value designed into color TV circuitry. Incidentally, if you divide 3.58 MHz by 2, the result is 1.79 MHz, the frequency of the Atari CPU. In the composite video signal, the luminance is the primary signal. Whenever the luminance changes it forces a phase shift or timing change in the color signal. The phase or timing of the chrominance signal is crucial to determining the color displayed. If the luminance is changed on a whole color clock boundary, the color is unaffected. However, if the luminance is changed on a half color clock boundary, it will affect the colors. This is the reason that artifacting is used in those modes that have one color, two luminances and a pixel width of one-half color clock. Box 12 is a short program that illustrates artifacting.



```
BOX 12
Artifacting

5 REM ** ARTIFACTING **
20 GRAPHICS 8:SETCOLOR 2,0,0
30 COLOR 1
40 REM * DRAW A SERIES OF VERTICAL BARS *
50 FOR I=1 TO 315 STEP 4
60 PLOT I,10:DRAWTO I,100
70 NEXT I
80 REM * REM DRAW VERTICAL BARS SHIFTED ONE HALF COLOR CLOCK
*
90 FOR I=0 TO 315 STEP 2
100 PLOT I,30:DRAWTO I,50:NEXT I
110 REM * MORE VERTICAL BARS SHIFTED AGAIN *
120 FOR I=0 TO 315 STEP 3
130 PLOT I,60:DRAWTO I,80
140 NEXT I
```

Box 12. Artifacting

Character Set Graphics

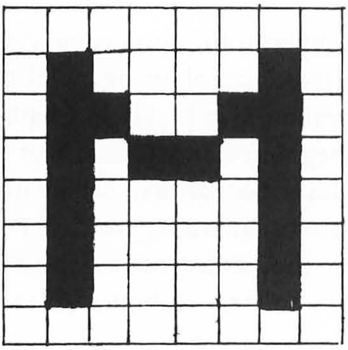
Among the many reasons for the superior graphics capabilities of Atari Home Computers is the relative ease of using redefined characters. Redefined characters are invaluable in playfield graphics for games, simulations, and utilities. They may be used in any of the BASIC or ANTIC text modes to draw blueprints, schematics, scientific symbols, icons, maps, trees, ... almost anything you may need to add realism and interest to a program. Another attractive feature of character set graphics is that it allows you to create a high resolution type screen with much less memory.

A character set is the table of data the Atari Home Computer uses to define each letter or shape that it displays in text modes. Each character is represented by a sequence of eight bytes. The Atari stores 128 characters in ROM at locations 57344 to 58367. Each character and each inverse video character has been assigned a number from 0 to 255 (see Appendix E). These numbers are the ATASCII Codes. However the characters are not stored in ROM in ATASCII order. The order in which the characters are stored in ROM is listed in Table 9.6 of the *Atari BASIC Reference Manual* and here in Appendix F. We have referred to this ordering of the characters as the internal character code. (There are three misprints in our edition of the BASIC

Reference Manual. Internal character number 13 should be the minus sign; character 14 should be the period; and character 63 should be the underline key.) This table is necessary in planning which characters you are going to redefine and eliminates using complicated formulas that have been devised to change ATASCII code into internal character numbers.

The character modes are BASIC Modes 0,1,2 and ANTIC Modes 3,4,5. BASIC Mode 0 and ANTIC Modes 3,4,5 use the full 128 character set. BASIC Modes 1 and 2 use the first 64 characters listed in Table 9.6 or Appendix F. Characters are plotted on the screen using 8 x 8 grids of pixels. The size of each pixel depends on the text mode used. Each row of pixels is a byte of data with a 1 representing a lighted pixel and a 0 representing an unlighted pixel. For example:

CHARACTER



DATA BYTES

0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	0
0	1	1	0	0	1	1	0
0	1	0	1	1	0	1	0
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0
0	0	0	0	0	0	0	0

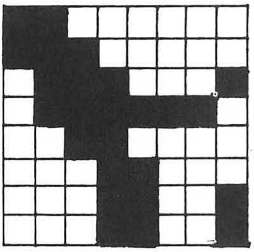
Figure 3-8. Lighted and unlighted pixels

Now you can see why each character occupies eight bytes in memory. There is one byte for each row of pixels. The spacing between lines of print on the screen is made by leaving the top and bottom rows of the grid unlit. There are spaces between characters only if you construct them that way. Thus, you may build up large pictures by combining characters.

There are four basic steps to follow when redefining characters. These are:

1. Construct and define your characters on an 8 x 8 grid. For large pictures use graph paper to sketch out the complete set of characters that make up your picture.
2. Move the standard character set from ROM to RAM.
3. Revise the relocated standard set by POKEing in you new data.
4. Print the redefined characters on the screen.

Step 1: Construct and define your characters. Figure 3-9 is an example of a character that is used in the program in Box 14.



1 1 0 0 0 0 0 0	128+64		= 192
1 1 1 0 0 0 0 0	128+64+32		= 224
0 1 1 1 0 0 0 1	64+32+16	+ 1	= 112
0 1 1 1 1 1 1 0	64+32+16+8+4+2		= 126
0 0 1 1 0 0 0 0	32+16		= 48
0 0 0 1 1 0 0 0	16+8		= 24
0 0 0 1 1 0 0 1	16+8	+ 1	= 24
0 0 0 1 1 0 0 1	16+8	+ 1	= 24

Figure 3.9. Example character

The sequence used to construct a character can be seen from this figure. The character is sketched on graph paper. Then each lit pixel is indicated by a 1, each unlit pixel by a 0. The resulting eight bits are treated as a binary number which is converted into a decimal number. The decimal numbers are the data that define the character and will be stored sequentially in memory starting with the top byte.

When you have to redefine a large number of characters, converting the binary numbers to decimal numbers is a tedious job. The program in Box 13 is a simple utility that generates a character's data numbers and print them on the screen or printer. The program displays an 8x8 grid of "O's." Starting with the upper left 'O,' a question mark is displayed. If you want the pixel at the location of the question mark lit, type 'Y' and an 'X' is displayed. If you do not want the pixel lit, type 'N' and the 'O' is erased. After running through a grid you may edit and then print out the data.

Step 2: Move the standard character set from ROM to RAM. Strictly speaking this step is necessary only if you intend to use some of the standard characters. In BASIC this is done by PEEKing each ROM location from 57344 to 58367 (in BASIC Modes 1 and 2 from 57855 to 58367) and POKEing the value into a RAM location. Before doing this you must reserve a safe place in memory. As we discussed in the section on display lists one way to secure a safe location is to lower RAMTOP.

BOX 13
Utility Program
BASIC Character Generator

```
5 REM ** CHARACTER GENERATOR **
10 DIM D(8)
20 OPEN #2,4,0,"K:"
30 GRAPHICS 2
35 REM * DISPLAY GRID *
40 FOR J=1 TO 8:FOR I=6 TO 13
50 POSITION I,J:? #6;"O"
60 NEXT I:NEXT J
70 PRINT "TYPE Y FOR PIXEL LIT"
80 PRINT
90 PRINT "TYPE N FOR PIXEL OFF"
100 FOR J=1 TO 8:FOR I=6 TO 13
110 POSITION I,J:? #6;"?"
120 REM * GET USER'S CHOICE AND COMPUTE DATA NUMBERS *
130 GET #2,CHOICE
140 IF CHOICE=ASC("Y") THEN GOTO 560
150 IF CHOICE=ASC("N") THEN POSITION I,J:? #6;" ":GOTO 170
160 GOTO 120
170 NEXT I
180 D(J)=DECICODE
190 DECICODE=0
200 NEXT J
210 REM * GIVE USER A CHANCE TO EDIT *
220 PRINT :PRINT
230 PRINT "EDIT LINE (Y,N)"
240 GET #2,EDIT
250 IF EDIT=ASC("Y") THEN GOTO 610
260 IF EDIT=ASC("N") THEN GOTO 290
270 GOTO 240
280 REM * GIVE USER AN OUTPUT OPTION *
290 PRINT :PRINT
```

(Continued on next page)

```
300 PRINT "PRINT TO SCREEN OR PRINTER (P,S)"
310 GET #2,OPTION
320 IF OPTION=ASC("P") THEN GOTO 400
330 IF OPTION=ASC("S") THEN GOTO 350
340 GOTO 300
350 PRINT :FOR P=1 TO 8:? D(P);",";
360 NEXT P
370 GOTO 480
380 REM * OUTPUT TO EPSON PRINTER *
390 REM * YOUR FORMAT MAY DIFFER *
400 OPEN #3,8,0,"P:"
410 FOR P=1 TO 8
420 PRINT #3;D(P);
430 PRINT #3;"  ";
440 NEXT P
450 PRINT #3
460 CLOSE #3
470 PRINT :PRINT
480 PRINT :PRINT
490 PRINT " 'C' TO CONT. 'Q' TO QUIT"
500 GET #2,OPTION
510 IF OPTION=ASC("C") THEN GOTO 30
520 IF OPTION=ASC("Q") THEN GOTO 540
530 GOTO 490
540 END
550 REM * ALGORITHM TO COMPUTE DATA NUMBERS *
560 POWER=INT((2^(13-I))+0.1):POSITION I,J:? #6;"X"
570 DECICODE=DECICODE+POWER
580 IF FLAG=1 THEN GOTO 710
590 GOTO 170
600 REM * EDITING ROUTINE *
610 TRAP 610:PRINT "TYPE LINE NUMBER (1-8) RETURN"
620 INPUT J
630 FLAG=1
640 TRAP 40000
650 FOR I=6 TO 13:POSITION I,J:? #6;"O":NEXT I
660 FOR I=6 TO 13:POSITION I,J:? #6;"?"
670 GET #2,CHOICE
680 IF CHOICE=ASC("Y") THEN GOTO 560
690 IF CHOICE=ASC("N") THEN POSITION I,J:? #6;" ":GOTO 710
700 GOTO 670
710 NEXT I
720 FLAG=0
730 D(J)=DECICODE
740 FLAG=0:DECICODE=0
750 GOTO 230
```

BOX 13 continued

In the program in Box 14, lines 10 through 70 lower RAMTOP 8 pages and move 64 characters from ROM into RAM starting at a location 4 pages above RAMTOP. This leaves a 1K buffer between RAMTOP and the character set. There is one general rule to observe: *The character set must begin at the start of a memory page.*

Step 3: Redefine the character set. Choose the characters listed in Appendix E that you are changing and POKE in new data numbers. In Box 14, lines 80 through 230 change characters 3 through 15 (# through /). The redefined characters will be identified with the internal code and the corresponding symbol of the character they replace. When choosing characters to modify, it makes sense to choose a continuous sequence for ease in programming.

Step 4: Displaying the characters on the screen. The new character set will not be displayed on the screen if the OS is not aware of its existence. Memory locations 756 (CHBAS - shadow register) and 54281 (the corresponding hardware register) store the page number of the start of the character set currently in use. Switch character sets by POKEing the appropriate address in 756. This instruction must come after the GRAPHICS command that sets up the screen. Each time a GRAPHICS command is executed, the OS sets 756 back to the ROM character set. Once CHBAS is changed new characters may be printed to the screen using the standard symbols as their names or POKEing their internal code numbers directly into screen memory.

When you type in the program in Box 14 and run it you will discover that using BASIC to redefine a character set is slow. The process can be speeded up immensely by using short machine language routines to move and redefine the character set and by storing the new character set data as a string. We will explain how to do this in the next chapter.

ANTIC Modes 4 and 5 (BASIC 12 and 13 in XL/XE Series)

Antic Modes 4 and 5 are four color character modes specifically designed to be used with redefined character sets. As we previously discovered, if you increase color options then you have to give something up. Here, once again, it is resolution. BASIC Mode 0 characters are 8 x8 grids of pixels; ANTIC Mode 4 characters are 4 x 8

```
5 REM ** REDEFINED CHARACTERS **
10 POKE 106,PEEK(106)-8
20 GRAPHICS 2+16
30 A=PEEK(106)
40 START=(A+4)*256
50 FOR R=0 TO 511
60 POKE START+R,PEEK(57344+R)
70 NEXT R
80 FOR X=0 TO 103:READ P
90 POKE START+3*8+X,P
100 NEXT X
110 DATA 192,224,113,126,48,24,25,25
120 DATA 0,126,129,0,0,0,129,129
130 DATA 3,7,30,126,12,24,152,152
140 DATA 8,4,2,1,0,0,63,0
150 DATA 49,96,192,192,192, 163,242,164
160 DATA 0,0,0,0,24,255,24,24
170 DATA 140,6,3,3,3,197,79,69
180 DATA 16,32,64,128,0,0,252,0
190 DATA 0,1,2,4,8,0,0,0
200 DATA 192,128,64,32,16,15,0,0
210 DATA 24,24,24,126,129,0,0,0
220 DATA 3,1,2,4,8,240,0,0
230 DATA 0,128,64,32,16,0,0,0
240 POKE 756,A+4
250 POSITION 9,1:? #6;"#$%"
260 POSITION 8,2:? #6;"&'()*"
270 POSITION 8,3:? #6;"+,-./"
280 GOTO 280
```

Box 14. Redefining a character set using BASIC

grids of pixels. On the screen, Mode 4 pixels are twice as wide as Graphics 0 pixels so the characters come out to be the same size. ANTIC Mode 5 pixels are the same width but twice as high (16 scan lines) as Mode 4 pixels. In Mode 4 and 5 the standard characters are unreadable.

Each pixel is assigned a pair of bits that determine the color register used to display it. Color register selection is made according to Table 3-9.

Table 3-9. Color register selection

Bit Pair	**Internal Char.** **Codes 0 - 127** Normal Video	**Internal Char.** **Codes 128 - 127** Inverse Video
00	COLBAK 712	COLBAK 712
01	PFO 708	PFO 708
10	PF1 709	PF1 709
11	PF2 710	PF3 711

Storing the data in memory and putting ANTIC Mode 4 and 5 characters on the screen follows the same procedure described previously. However, constructing ANTIC Mode 4 and 5 characters is a slightly different task than redefining character sets for the other modes.

To define your ANTIC Mode 4 characters, you will, of course, want to sketch your picture in color first. Then assign colors to the color registers listed in Table 3-8. Generating character data then becomes a matter of coloring in a character and translating from color to binary number to decimal number. For example suppose we want to design this character as part of a space ship:

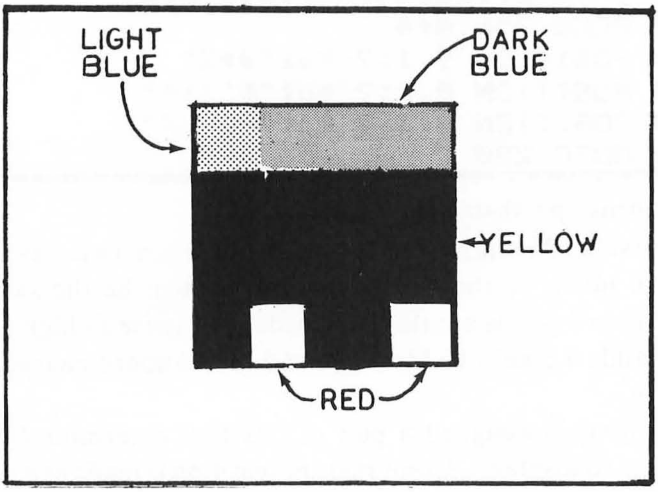


Figure 3-10. Designing a character

We'll make the register assignments as:

00	COLBAK	712	LT. BLUE
01	PF0	708	DARK BLUE
10	PF1	709	YELLOW
11	PF2	710	RED

00	01	01	01	00010101	21
00	01	01	01	00010101	21
10	10	10	10	10101010	170
10	10	10	10	10101010	170
10	10	10	10	10101010	170
10	10	10	10	10101010	170
10	11	10	11	10111011	187
10	11	10	11	10111011	187

Figure 3-11. Register assignments

At best, this is a tedious job so we have included a simple character editor in Box 15.

Player Missile Graphics

Probably more has been written about Atari's Player Missile (PM) graphics than any other feature of the Atari system. Certainly a good portion of the reason for this is that PM graphics is essential for high quality animation in games, and games are fun! Another part of the reason for interest in PM graphics is that you can do things with PMs that you can't do as easily with character set or map mode graphics. Consequently, there is a small increase in the complexity of the ideas involved and the number of concepts to be explored and this also generates more articles.

Player Missile graphics is, to a great extent, independent of other display graphics. You can use PMs in animation and games, but you can also use PMs to enhance character set and map mode graphics. We urge you to approach the topic of PM graphics receptive to the idea that you can use PM for more than games. We will begin by discussing the nature and characteristics of PM graphics. Then we will describe how to set them up and what you can do with them.

The reason PM graphics is independent of playfield graphics is that data for PM graphics is separate from the data used for character or map mode graphics. You can think of ANTIC and GTIA as taking information from two different sections of memory and from two different sets of color/control registers and combining it into one video signal. In a sense the PM graphics information is superimposed on the signal that generates the playfield. There is a fundamental difference between screen memory organization, player memory organization, and how the two are related to pixels on the TV screen.

Earlier in the chapter we said that in Graphics 2, the first 20 bytes in screen memory correspond to the first row of characters (pixels) on the screen, the second 20 bytes in screen memory to the second row of pixels, and so on. This describes the mapping of the two dimensional screen array into a linear memory array. With PM graphics, a linear array of either 128 or 256 memory bytes is mapped into a vertical column eight pixels wide on the screen. There are four players, labeled

BOX 15
Utility Program
ANTIC Character Generator

```
20 REM ** MULTICOLORED CHARACTER EDITOR **
30 REM * TO USE THIS PROGRAM YOU SHOULD *
40 REM * HAVE A COLORED SKETCH FROM *
50 REM * WHICH TO WORK *
60 PRINT "}"
70 DIM DN(8)
80 OPEN #2,4,0,"K:"
90 PRINT
100 PRINT "INPUT COLOR SELECTIONS"
110 PRINT
120 PRINT "COLOR NO. = 16*HUE+LUMINANCE"
130 PRINT
140 PRINT "COLOR NUMBER TO GO WITH 00 IS"
150 INPUT A
160 PRINT
170 PRINT "COLOR NUMBER TO GO WITH 01 IS"
180 INPUT B
190 PRINT
200 PRINT "COLOR NUMBER TO GO WITH 10 IS"
210 INPUT C
220 PRINT
230 PRINT "COLOR NUMBER TO GO WITH 11 IS"
240 INPUT D
250 REM * DISPLAY COLOR CHOICES ON THE CHOSEN BACKGROUND *
255 REM * COLOR CHOICES WILL AFFECT TEXT WINDOW  LEGIBILITY
260 GRAPHICS 7
270 POKE 712,A:POKE 708,B:POKE 709,C:POKE 710,D
280 COLOR 1:Q=5:GOSUB 5000
290 COLOR 2:Q=45:GOSUB 5000
300 COLOR 3:Q=85:GOSUB 5000
310 PRINT "TYPE E TO EDIT"
320 PRINT
330 PRINT "TYPE G TO GO ON"
340 GET #2,OPTION
350 IF OPTION=ASC("E") THEN 140
360 IF OPTION=ASC("G") THEN 390
370 GOTO 340
380 REM * DISPLAY GRID *
390 GRAPHICS 2
400 FOR J=1 TO 8:FOR I=6 TO 13
410 POSITION I,J:? #6;"'"
420 NEXT I:NEXT J
430 PRINT "TYPE 0 OR 1 IN"
440 PRINT
```

Box 15. Utility Program ANTIC Character Generator

```
450 PRINT "RESPONSE TO  '?' ."
460 FOR J=1 TO 8:FOR I=6 TO 13
470 POSITION I,J:? #6;"?"
480 REM * GET USER'S CHOICE AND COMPUTE DATA NUMBERS *
490 GET #2,CHOICE
500 IF CHOICE=ASC("1") THEN 920
510 IF CHOICE=ASC("0") THEN POSITION I,J:? #6;"0":GOTO 530
520 GOTO 260
530 NEXT I
540 DN(J)=DECICODE
550 DECICODE=0
560 NEXT J
570 REM * GIVE USER A CHANCE TO EDIT *
580 PRINT :PRINT
590 PRINT "EDIT LINE ?  (Y/N) "
600 GET #2,EDIT
610 IF EDIT=ASC("Y") THEN 970
620 IF EDIT=ASC("N") THEN 650
630 GOTO 600
640 REM * GIVE USER AN OUTPUT OPTION *
650 PRINT :PRINT
660 PRINT "PRINT TO SCREEN OR PRINTER ? (P/S)"
670 GET #2,CHOICE
680 IF CHOICE=ASC("P") THEN 760
690 IF CHOICE=ASC("S") THEN 710
700 GOTO 670
710 PRINT :FOR P=1 TO 8:PRINT DN(P);",";
720 NEXT P
730 GOTO 840
740 REM * OUTPUT TO EPSON PRINTER *
750 REM * YOUR FORMAT MAY DIFFER *
760 OPEN #3,8,0,"P:"
770 FOR P=1 TO 8
780 PRINT #3;DN(P);
790 PRINT #3;"  ";
800 NEXT P
810 PRINT #3
820 CLOSE #3
830 PRINT :PRINT
840 PRINT :PRINT
850 PRINT "C TO CONT. OR Q TO QUIT"
860 GET #2,OPTION
870 IF OPTION=ASC("C") THEN 390
880 IF OPTION=ASC("Q") THEN 900
890 GOTO 860
900 END
910 REM ALGORITHM TO COMPUTE DATA NUMBERS
920 POWER=INT((2^(13-I))+0.1):POSITION I,J:? #6;"1"
```

Box 15. *(cont.)*

```
930 DECICODE=DECICODE+POWER
940 IF FLAG=1 THEN 1070
950 GOTO 530
960 REM EDITING ROUTINE
970 TRAP 970:PRINT "TYPE LINE NUMBER (1-8)"
980 INPUT J
990 FLAG=1
1000 TRAP 40000
1010 FOR I=6 TO 13:POSITION I,J:? #6;"‘":NEXT I
1020 FOR I=6 TO 13:POSITION I,J:? #6;"?"
1030 GET #2,CHOICE
1040 IF CHOICE=ASC("1") THEN 920
1050 IF CHOICE=ASC("0") THEN POSITION I,J:? #6;"0":GOTO 1070
1060 GOTO 1030
1070 NEXT I
1080 FLAG=0
1090 DN(J)=DECICODE
1100 FLAG=0:DECICODE=0
1110 GOTO 590
1120 REM
1130 REM
5000 FOR I=1 TO 10
5010 PLOT Q,50+I:DRAWTO Q+5,50+I
5020 NEXT I
5030 RETURN
```

Box 15. *(cont.)*

P0 - P3 and four missiles, M0 - M3. Missiles are columns that are two pixels wide. Each missile is associated with the player of the same number by the simple technique of having both of them the same color. The four missiles can be combined to form a fifth player. In order to combine the four missiles into a player the bit D_4 of PRIOR (53275) must be set. This may be done by POKEing 16 into its shadow register at memory location 623. This sets the "fifth player enable bit". However, using the fifth player from BASIC is clumsy since each missile retains its own width and horizontal position. Consequently, moving the fifth player horizontally entails four POKEs, one for each missile. The only feasible way to utilize this option is to program the movement in machine language.

Organization of player memory so that it represents a column on the screen greatly simplifies movement of an image from one place to another. Horizontal positions can be changed with a simple POKE or STA. Vertical positions can be changed with a short, efficient machine language routine. The result is that animation can be accomplished more smoothly and more rapidly than if you were moving bytes through the normal screen memory.

PM graphics works like the normal BASIC character set graphics in that bit mapping is used to display a pattern. As usual, "1" equals pixel on and "0" equals pixel off. Player and missile pixel sizes can be varied. There are two choices of vertical resolution: one scan line, (256 byte player) or two scan lines, (128 byte player). Normal horizontal resolution of the players is eight color clocks. There is the option to set individual players to widths of sixteen or thirty-two color clocks. The widths of all missiles is set the same: either two, four, or eight color clocks. Colors for each player and its associated missile are taken from the four color registers 53266 - 53269 which are shadowed at 704 through 707.

Part of the complexity of PM graphics revolves around the fact that there are over thirty registers or memory locations that can be used to implement various options. In addition, some of these registers have dual functions. To impose some order on this chaos, we have prepared two tables that list the registers and their functions. Table 3-10 lists the general PM graphics control registers. Table 3-11 lists the hardware registers that serve dual functions.

Table 3-10. Player Missile Graphics Control Registers

Player Missile Graphics **CONTROL REGISTERS**	
SDMCTL 559 (shadow)	**DMACTL 54272 (hardware)**
Direct Memory Access (DMA) enable Bit assignments are as follows:	
D_1 D_0 Playfield Options:	0 0 no playfield 0 1 narrow playfield 1 0 standard playfield 1 1 wide playfield
D_2 Missile DMA	0 = disable 1 = enable
D_3 Player DMA	0 = disable 1 = enable
D_4 P/M Resolution	0 = 2 scan lines (default) 1 = 1 scan line
D_5 DMA Enable	0 = shuts ANTIC off/speed up CPU 1 = DMA enable
D_6 D_7 Unused	
GPRIOR 623 (shadow)	**PRIOR 53275 (hardware)**
This register selects which parts of the screen display will have priority and allows several other options. Bit assignments are as follows :	

continued on next page

BIT	**HIGHEST PRIORITY ——→ LOWEST**
D_0, if set:	Player 0-3/Playfield 0-3/Background
D_1, if set:	Player 0-1/Playfield 0-3/Player 2-3/Background
D_2, if set:	Playfield 0-3/Player 0-3/Background
D_3, if set:	Playfield 0-1/Player 0-3/Playfield 2-3/Background

Only one of the above bits should be set at one time.

D_4, if set:	Four missiles combine to a fifth player
D_5, if set:	Overlaps of players give a third color
D_6 D_7	Enable GTIA Modes

COLOR

704-707	OS shadow color registers for Player 0 - Player 4
53266-53269	Hardware color registers for Player 0 - Player 4

GRACTL 53277 **Graphics ConTrol.**

To turn on missiles set bit D_0

To turn on players set bit D_1

This register is also used with trigger inputs.

PM BASE 54297

This location, in pages, of the Player/Missile shape data. See discussion in text on how player missile memory is organized.

Table 3-11. Dual Function Hardware Registers

Dual Function Hardware Registers

LABEL (W)=write	DECIMAL (R)=read	HEX	FUNCTION
HPOSPO	53248	D000	(W) horizontal position of player 0
MOPF			(R) missile-playfield collision
HPOS1	53249	D001	(W) horizontal position of player 1
M1PF			(R) missile-playfield collision
HPOSP2	53250	D002	(W) horizontal position of player 2
M2PF			(R) missile-playfield collision
HPOSP3	53251	D003	(W) horizontal position of player 3
M3PF			(R) missile-playfield collision
HPOSMO	53252	D004	(W) horizontal position of missile 0
POPF			(R) player to playfield collision
HPOSM1	53253	D005	(W) horizontal position of missile 1
P1PF			(R) player to playfield collision
HPOSM2	53254	D006	(W) horizontal position of missile 2
P2PF			(R) player to playfield collision

(cont. on following page)

HPOSM3	53255	D007	(W) horizontal position of missile 3
P3PF			(R) player to playfield collision
MOPL	53256	D008	(R) missile 0 to player collision
SIZEPO			(W) size of player 0
M1PL	53257	D007	(R) missile 1 to player collision
SIZEP1			(W) size of player 1
M2PL	53258	D00A	(R) missile 2 to player collision
SIZEP2			(W) size of player 2
M3PL	53259	D00B	(R) missile 3 to player collision
SIZEP3			(W) size of player 3
POPL	53260	D00C	(R) player 0 to player collision
SIZEM			(W) size for all missiles
GRAFPO	53261	D00D	(W) graphics shape for player 0
P1PL			(R) player 1 to player collisions
GRAPFP1	53262	D00E	(W) graphics shape for player 1
P2PL			(R) player 2 to player collisions

(cont. on following page)

GRAPFP2	53263	D00F	(W) graphics shape for player 2
P3PL			(R) player 3 to player collisions
GRAPFP3	53264	D010	(W) graphics shape for player 3
TRIG0 (644)			(R) joystick trigger 0
GRAPFPM	53265	D011	(W) graphics for all missiles
TRIG1			(R) joystick trigger 1
COLPM0	53266	D012	(704) color/luminence of player/missile 0
TRIG2 (646)			(R) joystick trigger 2
COLPM1	53267	D013	(705) color/luminance of player/missile 1
TRIG3 (647)			(R) joystick 3 trigger

The procedure for setting up players to use in a program is quite similar to that for setting up character graphics. In general terms:

1. Choose your colors and resolution.
2. Design your player or missile and represent it as data bytes.
3. Set aside a location in memory for the player or missile data and store the data bytes.
4. Tell ANTIC and GTIA to start the display and where to place it on the screen.

Suppose we want to design a light bulb to be player zero. Its color will be pink, we'll use normal width pixels and two line resolution. Referring to Tables 3-10 and 3-11 we note that we'll have to keep in mind to:

POKE 704,88 To set player's color to pink
POKE 559,88 To set Bit D_1 (normal playfield)
Bit D_3 (player DMA enable)
Bit D_5 (screen memory DMA enable)
POKE 53277,2 To enable GRACTL
POKE 53256,0 To set the width to 8 color clocks.

Sketching the lightbulb and calculating the data numbers is the same procedure as with character sets. The major difference between the two is that you can have many more data bytes per player than per character.

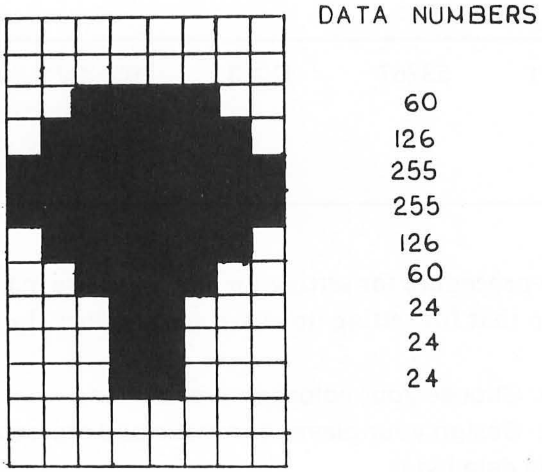


Figure 3-12. Lightbulb character

Memory organization for Player Missile RAM is quite different from anything we have run into thus far and reflects the internal design of Antic. For double line resolution PM graphics, you need 1K bytes set aside and the starting location, called PM BASE, must be on a 1K boundary. For single line resolution, you need 2K bytes set aside and PM BASE must start on a 2K boundary. Within the block of PM memory there is an unused section. This memory may be used for other things such as machine language player-movement routines. Figure 3-13 is a detailed memory map of the PM RAM.

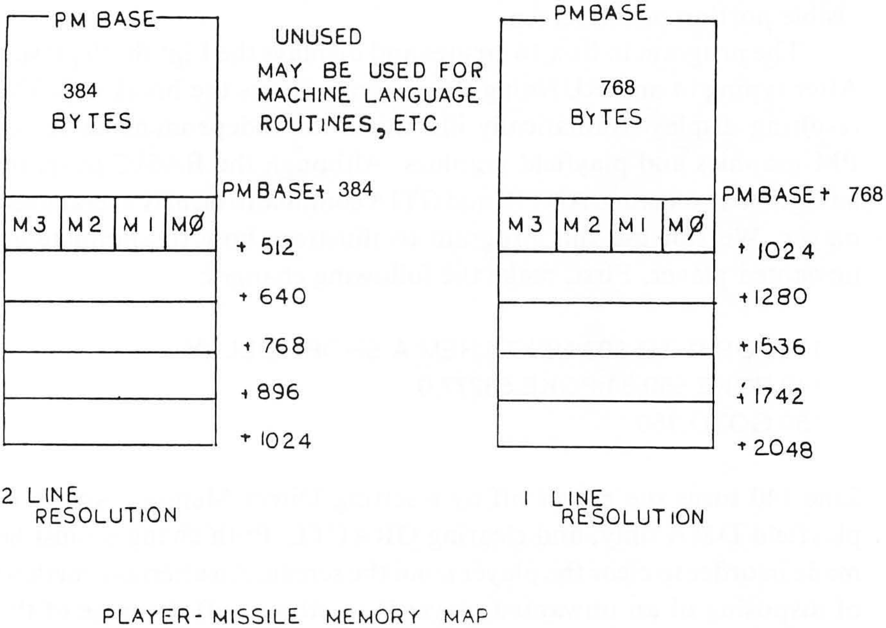


Figure 3-13. Player-Missile memory map

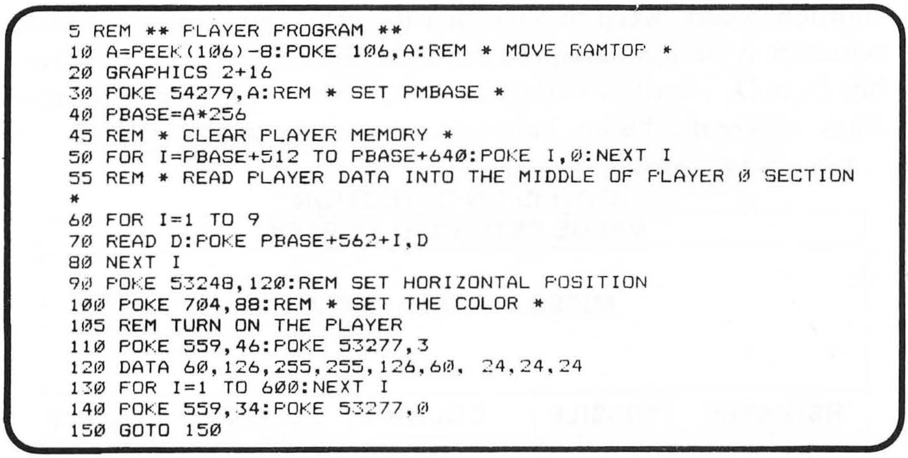
Before we put all of the above information together into a simple demonstration program, let's look at how players are positioned on the screen. Horizontal positions of players and missiles are determined by POKEing or storing values from 0 to 255 in registers 53248 to 53255. The number POKEd in determines the color clock used for the left hand edge of the player. Players are visible in the approximate range of 47 to 208. But, due to differences in individual TV sets it might be best to allow a smaller range, say 55 to 200. Since a normal playfield is 160 color clocks wide, this implies that it is possible to store a player "in the wings waiting to come on stage". Vertical positioning of a player depends on the positioning of the image within the RAM set aside for that player. Consider our lightbulb. When storing the data numbers within the PM RAM we will first clear the RAM to all zeros so no extraneous images appear. We can then place the 9 data bytes anywhere in the section of RAM set aside for Player 0 (see Figure 3-13). The lightbulb's position on the screen will correspond roughly to the position of the data bytes in RAM. Note that, because of TV overscan, it is possible to locate a player or missile above or below the visible portion of the screen.

The program in Box 16 creates and displays the Lightbulb player. After typing in and RUNning the program, press the break key. The resulting display dramatically illustrates the independent nature of PM graphics and playfield graphics. Although the BASIC program has stopped executing, ANTIC and GTIA continue to display a scrambled player. We can use this program to illustrate how to eliminate an unwanted player. First, make the following changes:

```
130 FOR I1 TO 600:NEXT I:REM A SHORT DELAY
140 POKE 559,34:POKE 53277,0
150 GOTO 150
```

Line 140 turns the player off by resetting Direct Memory Access to playfield DMA only, and clearing GRACTL. Both changes must be made in order to clear the player from the screen. An alternate method of disposing of an unwanted player is to store it off the edge of the screen. Try changing line 140 to:

```
POKE 53248,0
```



```
5 REM ** PLAYER PROGRAM **
10 A=PEEK(106)-8:POKE 106,A:REM * MOVE RAMTOP *
20 GRAPHICS 2+16
30 POKE 54279,A:REM * SET PMBASE *
40 PBASE=A*256
45 REM * CLEAR PLAYER MEMORY *
50 FOR I=PBASE+512 TO PBASE+640:POKE I,0:NEXT I
55 REM * READ PLAYER DATA INTO THE MIDDLE OF PLAYER 0 SECTION
*
60 FOR I=1 TO 9
70 READ D:POKE PBASE+562+I,D
80 NEXT I
90 POKE 53248,120:REM SET HORIZONTAL POSITION
100 POKE 704,88:REM * SET THE COLOR *
105 REM TURN ON THE PLAYER
110 POKE 559,46:POKE 53277,3
120 DATA 60,126,255,255,126,60, 24,24,24
130 FOR I=1 TO 600:NEXT I
140 POKE 559,34:POKE 53277,0
150 GOTO 150
```

Box 16. Lightbulb player missile concepts

Collisions and Priority

The last topics we will consider in this chapter are collisions and priority. A collision occurs when you have instructed GTIA to overwrite one object on the screen with another. Objects refer to players, missiles, and playfields. Playfields can be either character graphics or map mode graphics displays. A collision is any overwriting of one object by another in which at least one screen pixel is overwritten. Two objects touching do not constitute a collision. There are four types of collisions available to the programmer. These are: (1) player to player, (2) player to playfield, (3) missile to playfield, (4) missile to player.

The Atari Home Computer provides sixteen hardware registers to monitor the fifty-two possible collisions. From table 3-11 we see that they are the dual function hardware registers 53248 through 53263. The various types of collision cause certain bits (D_0 to D_3) of these registers to be set. If a bit is set a particular collision has occured. These are read only registers consequently they can only be cleared by storing a number in the "Hit Clear Register", 53278 (HITCLR). The OS does not automatically clear the collision registers. As a result it must be done by the programmer so that the program can continue checking for collisions.

Table 3-12 gives the bits set and the values returned when a collision occurs. Keep in mind that the value returned by a PEEK statement is the decimal equivalent of the binary number expressed by bits D_0 to D_3 . Because of this, the value returned will depend on how many *different* collisions have occured since the last 'hit clear'.

Table 3-12 . Collision Detection

COLLISION DETECTION

VALUE RETURNED BY PEEK

MISSILE TO PLAYFIELD

REGISTER	MISSILE	COLOR 1	COLOR 2	COLOR 3
53248	M0	1	2	4
53249	M1	1	2	4
53250	M2	1	2	4
53251	M3	1	2	4

PLAYER TO PLAYFIELD

	PLAYER			
53252	P0	1	2	4
53253	P1	1	2	4
53254	P2	1	2	4
53255	P3	1	2	4

MISSILE TO PLAYER

	MISSILE	P0	P1	P2	P3
53256	M0	1	2	4	8
53257	M1	1	2	4	8
53258	M2	1	2	4	8
53259	M3	1	2	4	8

(cont. on following page)

PLAYER TO PLAYER

	PLAYER				
53260	P0	0	2	4	8
53261	P1	1	0	4	8
53262	P2	1	2	0	8
53263	P3	1	2	4	0

Another facet of collision detection concerns collisions between players. This type of collision results in bits being set in two registers. For example, a collision between Player 0 and Player 1 sets bit D_1 in register 53260 and bit D_0 in register 53261. Suppose player 3 collides with player 2 - the value in register 53263 is 4 and the value in register 53262 is 8. When two players collide with each other both registers have a number written in them. Another aspect of reading collision registers is that if player 2 collides with more than one player then the value in 53263 will be the sum of the collisions. For example:

```
Player 2 hits Player 1 = 2
Player 2 hits Player 3 = 8
                 53263 = 10
```

It is evident that collision detection gives you several programming options. If you only need to know that a collision has occured, it is sufficient to set HITCLR and test the appropriate register to see if it is greater than zero. On the other hand if you need to know exactly which object has been in a collision, then individual bits must be tested with a logical AND.

The program in Box 17 illustrates collision detection. In this program the lightbulb drawn in the previous box falls onto a row of M's at the bottom of the screen. As the bulb descends, the value in the collision register is displayed in the text window allowing you to see and hear the program in action. After typing in the program and running it, remove line 140 and run it again. As the bulb descends through the row of M's and on into the text window you will see the collision register values change. These changes are a result of the color registers, ie. the playfield registers, used to display the * and the text window.

This illustrates another aspect of collision detection - the values returned depend upon the color registers *being used* (registers 708-712) but not on the color values in the registers. Recall that in a sense playfields are synonymous with color registers. Thus a multicolored character in ANTIC Mode 4 or 55 could return a different value depending on which part of the character was overwritten.

BOX 17

```
1 REM ** COLLISION **
5 PRINT CHR$(125)
10 DIM MOVE$(21):DOWN=ADR(MOVE$)
15 A=PEEK(106)-8:POKE 106,A
20 POKE 54279,A
30 PB=A*256
35 FOR CM=PB+512 TO PB+640:POKE CM,0:NEXT CM
40 X=120:Y=20
50 FOR P=1 TO 9:READ D:POKE PB+512+Y+P,D:NEXT P
55 DATA 60,126,255,255,126,60, 24,24,24
60 POKE 53248,X
70 FOR I=DOWN TO DOWN+20
75 READ B:POKE I,B:NEXT I
80 DATA 104,104,133,204,104,133,203
85 DATA 160,20,177,203,200,145,203
90 DATA 136,136,192,255,208,245,96
100 GRAPHICS 2:POKE 704,88
105 POSITION 0,8:? #6;"MMMMMMMMMMMMMMMMMMM"
110 POSITION 0,9:? #6;"*******************"
120 POKE 559,46:POKE 53277,3
125 POKE 53278,255
130 ST=USR(DOWN,PB+511+Y):Y=Y+1
135 PRINT PEEK(53252)
140 IF PEEK(53252)=4 THEN GOTO 150
145 FOR I=1 TO 25:NEXT I:GOTO 125
150 ? PEEK(53252):SOUND 1,20,0,14:SOUND 2,255,10,15
165 FOR I=1 TO 300:NEXT I
170 SOUND 1,0,0,0:SOUND 2,0,0,0
```

Box 17. Collision

When you have one or more objects on the screen, you may want to hide one behind the other. If one of the objects is moving, this can add a three dimensional quality to the picture. The Atari operating system has a register at 623 (GPRIOR) that is the shadow register for PRIOR at 53275 which controls display priority among the players and playfields. As you can see from table 3-10, PRIOR controls several unrelated functions. However the lower four bits D_0 to D_3 control display priority. On power-up bit 0 is set and players have priority over the playfields and backround. The backround always has the lowest priority. Setting bit 1 POKE 623,2) gives players 0 and 1 priority over playfields *and* over players 2 and 3, but the playfields have priority over players 2 and 3. The remaining priorities may be found in table 3-13. The bits in PRIOR are mutually exclusive . This means, theoretically, you can only set one of the bits D_0 to D_3. In a mutually exclusive situation, turning on more than one bit causes the bits to be in opposition. When the priority bits are opposed and any of the objects displayed overlap the display turns black in the overlapping area.

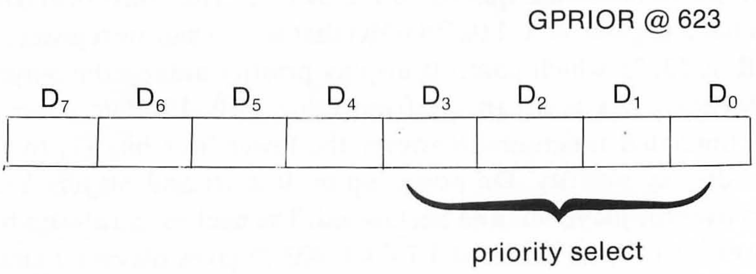
Finally, if you have established priority between players and playfields, the collision registers respond in the usual manner. Hence, when two objects are on the screen at the same time and place a collision will occur whether or not you see it and this is duly recorded by the collision registers.

To see priority in action add the following lines to the program in box 17:

```
102 POSITION 0,4:PRINT #6;“####################”
103 POKE 623,8
```

When you run the program with this addition you will see that the playfield #'s have priority over the lightbulb so that the bulb appears to pass behind the #'s.

Table 3-13. Priority



BIT	Set by POKE 623,	PRIORITY CONTROL
0	1	All players have priority over all playfields
1	2	Players 0 and 1 have priority over playfields, which have priority over players 2 and 3
2	4	All playfields have priority over all players
3	8	Playfields 0 and 1 have priority over all players, which have priority over, Playfields 2 and 3

Within the group of players, lower numbered players have priority over higher numbered players.

A lower numbered playfield has priority over higher numbered playfields.

4

Getting Started in Machine Language Programming

Introduction

After the previous three chapter's lengthy introduction to the fundamentals of machine language and Atari graphics, it is time to get started on some programming examples using machine language subroutines. There are three ways in which such a subroutine can be integrated into a BASIC program: (1) Flags set in one or more hardware registers can cause the CPU to jump to a short subroutine; (2) A machine language routine can be called by a BASIC USR command; (3) Through a process called vector stealing, a subroutine can be added to the normal tasks carried out by the operating system during the vertical blank. The first of these three methods, flag setting, is used by display list interrupt routines. The BASIC USR command is useful for all sorts of routines such as moving players, clearing sections of memory, and redefining characters sets. Using vector stealing to insert machine language routines into the vertical blank is valuable in fine scrolling and music. We shall begin the discussion of actual machine language programming with a number of display list interrupt routines and follow this with examples of programs carried out using the BASIC USR command. Routines that execute during the vertical blank will be discussed in chapter six.

Display List Interrupts

The display list interrupt (DLI) is a nice place to begin experimenting with the 6502 instructions introduced in chapter two and writing machine language subroutines. With just a little bit of work you can obtain very colorful results that give immediate feedback. Furthermore, from a few simple programs you will learn to use quite a few different instructions and gain an understanding of several important programming concepts. Although the examples stress color changes, DLIs are useful for many other applications, some of which we will mention as we go along.

Box 18 is the first DLI program. The task performed is simple -the background color of a full screen Graphics 2 display is changed halfway down the screen from light blue to dark blue. The steps for preparing such a program are:

(1) Plan what you want to do on the screen.
(2) Write the DLI service routine
(3) Set up the program and instructions that invoke the routine.

Before we look at the machine language routine in Box 18, let's review what will happen within the computer. Setting bit D_7 of a display list instruction equal to 1 signals ANTIC to regard that instruction as a request for a DLI. ANTIC then checks NMIEN (Non-Maskable Interrupt ENable) at memory location 54286. If bits D_6 and D_7 of this register are set, then ANTIC will relay a NMI signal to the CPU. In the Atari Home Computer there are three sources for a NMI. These are the vertical blank interrupt, the DLI, or the Reset Key. The NMI signal tells the CPU to execute a sequence of instructions to determine the origin of the signal. Once it has found out that the interrupt is a DLI, the CPU goes to memory locations 512 (Lo-Byte) and 513 (Hi-Byte) for the address of the DLI service routine.

The task of our routine is to change the number in the hardware register 53274 from 152 to 146. To do this we will need to use the Accumulator. In this situation, prior to executing our service routine, the CPU was busy carrying out some other program. Consequently, we should anticipate that the Accumulator and the X and Y registers will be holding values that will be needed after our routine is completed. Therefore, the first thing that the DLI routines must do is to save the values in any of the CPU registers it will use on the stack. The last thing it must do before returning from the interrupt is to recall these values from the stack.

With this in mind, look at the machine language program in line 30 of Box 18.

```
                    BOX 18
            Display List Interrupt
               One Color Change

5 REM ** SIMPLE DISPLAY LIST INTERRUPT **
10 REM * READ ML PROGRAM INTO PAGE SIX *
20 FOR I=0 TO 10:READ ML:POKE 1536+I,ML:NEXT I
30 DATA 72,169,146,141,10,212,141,26, 208.104,64
40 REM * SET UP SCREEN AND SET DLI BIT IN DISPLAY LIST *
50 PRINT CHR$(125)
60 GRAPHICS 2+16:POKE 712,152
70 DL=PEEK(560)+PEEK(561)*256
80 POKE DL+10,7+128
90 REM * TURN ON DLI *
100 POKE 512,0:POKE 513,6
110 POKE 54286.192
120 GOTO 120
```

BOX 18. Display List Interrupt One Color Change

Written out in mnemonics it is as follows:

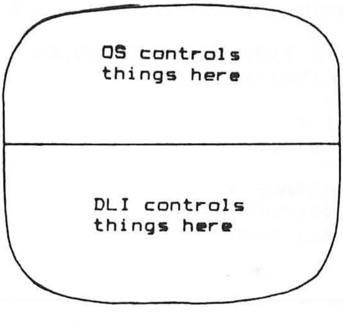
MNEMONIC	DECIMAL VALUE	FUNCTION
PHA	72	Push Accumulator to stack
LDA #146	169,146	Load the Accumulator with 146
STA WSYNC	141,10,212	Synchronize DLI with screen display
STA ADDR	141,26,208	Store 146 at 53274
PLA	104	Retrive previous accumulator value from stack
RTI	64	Return from interrupt

The program first saves the value of the accumulator by pushing it on the stack. It then loads the accumulator with the color number 146 and follows with STA WSYNC to synchronize the color change to the horizontal blank process. The STA command (opcode value 141) is in the absolute addressing mode. The numbers 10,212 are the Lo-Byte / Hi-Byte form of the register WSYNC (54282). 54282 is a memory location assigned to ANTIC. An interesting sidelight concerning this register is that apparently it does not matter what value is stored there. The STA operation itself is sufficient to do what is needed for synchronization. Following STA WSYNC, the instruction STA ADDR puts the color number 146 into the hardware register for the background color. This sequence makes it apparent that STA is a nondestructive command, ie., the number remains in the accumulator after it is stored in memory. Once again, this STA uses absolute addressing. Finally, the accumulator retrieves its original value from the stack and the processor returns from the interrupt.

Once the machine language program has been read into memory (line 20) it is necessary to set up the proper conditions to have it implemented. This consists of setting the DLI bit (D_7) in the display list instruction of the mode line before the one at which you want the change to occur. Then you must tell the OS where to find the DLI service routine by putting its starting address in 512 and 513. The final job is to set bits 6 and 7 of hardware register 54286 by POKEing in 192. A DLI may be turned off by a POKE 54286,64.

In the previous chapter we discussed the problem of providing a safe place for display lists, player-missile memory, and character sets. It is of prime importance to store machine language routines where they are not going to be overwritten by BASIC. Nothing makes a program crash faster than a machine language routine missing an RTS or some other vital instruction because it was overwritten with another part of the program. One option for storing machine language routines is to use page six (1536-1791) of memory. It has been pretty well documented that some cassette I/O can write over the first half of page six (1536-1664). Consequently, if you are using cassette storage, only page six locations 1665-1791 are safe. If you use disk storage all of page six is safe. Later we will discuss other techniques for storing or setting aside a safe place for machine language routines.

Display list interrupts are most advantageously used in situations where there is an OS shadow register associated with a hardware register. The reason for this is that during each vertical blank, the OS uses the value in the shadow register to update the corresponding hardware register. With a DLI you can change the number in the hardware register. Using color as an example, you effectively partition the screen so that the

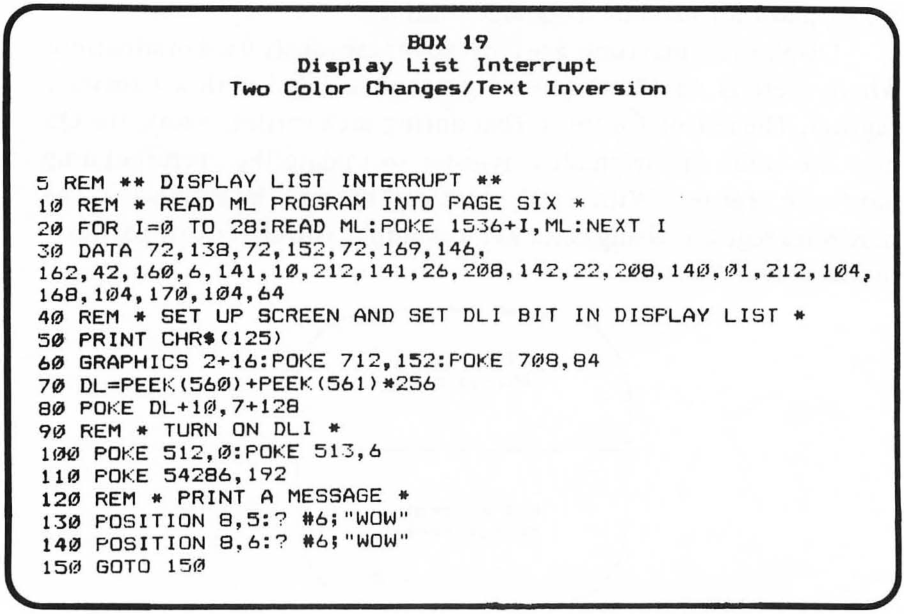


Of course, one doesn't have to split the screen in half. You can place a DLI on any or all mode lines. This means that you do not have to rely entirely on shadow register/hardware register pairs in using DLIs. With the inclusion of a DLI in one of the three blank line instructions at the top of ANTIC's display list, any change made in a system register while the electron beam is partway down the screen can be restored before the start of the drawing of a new display.

The program in Box 18 makes a single color change at a single location on the screen and serves to illustrate the use of PHA, LDA, STA, PLA, and RTI. There are three good ways to build on this simple program. One is to make up to three color changes during one interrupt routine by using the accumulator, the X and Y registers. A second course is to use multiple DLI routines. The final method is to use DLIs to access a table of color values.

The program in Box 19 makes changes in two color registers and one control register. It illustrates the use of several commands: TXA, TYA, LDX, LDY, STX, STY, TAY, and TAX.

Box 19. Display List Interrupt Two Color Changes/Text Inversion



BOX 19
Display List Interrupt
Two Color Changes/Text Inversion

```
5 REM ** DISPLAY LIST INTERRUPT **
10 REM * READ ML PROGRAM INTO PAGE SIX *
20 FOR I=0 TO 28:READ ML:POKE 1536+I,ML:NEXT I
30 DATA 72,138,72,152,72,169,146,
162,42,160,6,141,10,212,141,26,208,142,22,208,140,01,212,104,
168,104,170,104,64
40 REM * SET UP SCREEN AND SET DLI BIT IN DISPLAY LIST *
50 PRINT CHR$(125)
60 GRAPHICS 2+16:POKE 712,152:POKE 708,84
70 DL=PEEK(560)+PEEK(561)*256
80 POKE DL+10,7+128
90 REM * TURN ON DLI *
100 POKE 512,0:POKE 513,6
110 POKE 54286,192
120 REM * PRINT A MESSAGE *
130 POSITION 8,5:? #6;"WOW"
140 POSITION 8,6:? #6;"WOW"
150 GOTO 150
```

The machine language routine in Box 19 has four sections:
Section I: *Save the Accumulator, the X-register and the Y-register onto the stack.*

MNEMONIC DECIMAL VALUE FUNCTION

PHA................72.........Save accumulator on stack

TXA...............138.........Transfer X-register to accumulator

PHA................72.........Save this value, too

TYA...............152.........Transfer Y-register to accumulator

PHA................72.........Save Y-register, too

Section II: *Load the registers.*

LDA #146........169,146.......Load accumulator with the number for medium blue

LDX #42.........162,42........Load X-register with the number for yellow

LDY #6..........160,6.........Load Y-register with #6

Section III: *Wait for the horizontal blank. Store colors.*

STA WYSNC......141,10,212......Wait for horizontal blank

STA Color1......141,26,208......Medium blue to background register

STX Color2......142,22,208......Yellow to foreground register

STY Control......140,01,212......Change character mode control

Section IV: *Restore register, return from interrupt.*

PLA............104............Recall top value on stack
into accumulator

TAY............168............Transfer it to Y register

PLA............104............Recall next value on stack

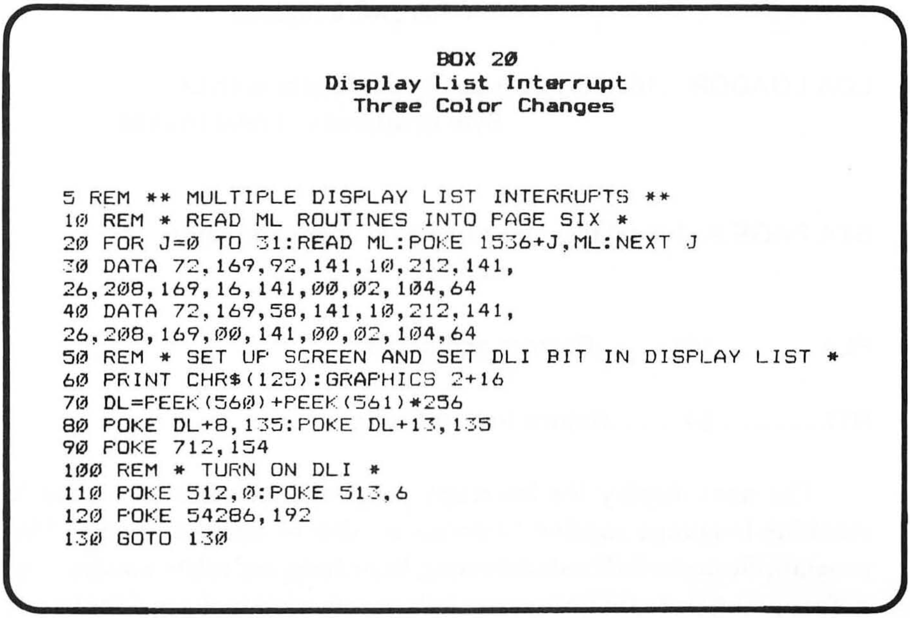
TAX............170............Transfer it to X register

PLA............104............Recall original accumulator
value

RTI.............64............Return from subroutine

Observe that the order in which the values are recalled from the stack by the PLA statements in Section IV is the reverse of the order in which they were pushed onto the stack by the PHA commands in Section I. This illustrates the last-in-first-out nature of the stack. You should also note that there are no instructions that pull values from the stack directly into the X and Y registers. Rather, one must restore these registers in a two step process involving the accumulator and the transfer instructions TAX, TAY. As a final comment on this program, observe that the load instructions use the immediate mode of addressing and the store instructions use the absolute mode of addressing.

Suppose that you plan a screen display using several DLIs, each doing a different job. You write the service routines and store them in page six. But how do you tell the CPU where to find the proper service routine for each interrupt when there is only one place to store a starting address? The solution is to have each service routine put the starting address for the next routine into 512 and 513. Box 20 is a very simple program illustrating linking one routine to the next. The program changes the background color of a Graphics 18 screen twice, first from light blue to pink and then to gold.



```
BOX 20
Display List Interrupt
Three Color Changes


5 REM ** MULTIPLE DISPLAY LIST INTERRUPTS **
10 REM * READ ML ROUTINES INTO PAGE SIX *
20 FOR J=0 TO 31:READ ML:POKE 1536+J,ML:NEXT J
30 DATA 72,169,92,141,10,212,141,
26,208,169,16,141,00,02,104,64
40 DATA 72,169,58,141,10,212,141,
26,208,169,00,141,00,02,104,64
50 REM * SET UP SCREEN AND SET DLI BIT IN DISPLAY LIST *
60 PRINT CHR$(125):GRAPHICS 2+16
70 DL=PEEK(560)+PEEK(561)*256
80 POKE DL+8,135:POKE DL+13,135
90 POKE 712,154
100 REM * TURN ON DLI *
110 POKE 512,0:POKE 513,6
120 POKE 54286,192
130 GOTO 130
```

BOX 20. Display List Interrupt Three Color Changes

The two service routines are in the data lines 30 and 40. With some counting you can see that the first routine is stored from 1536 through 1551 and the second from 1552 through 1568. Therefore, the starting address for the first routine in Lo-Byte/Hi-Byte order is 00, 06; for the second routine, 16, 06. The structure of the two routines is the same. The assembly listing of the first routine follows. You should write out the listing for the second routine for practice.

MNEMONIC DECIMAL VALUE FUNCTION

PHA...............72..........Push accumulator to stack

LDA Color......169,92.........Load accumulator with pink

STA WSYNC....141,10,212.......Wait for horizontal blank

STA COLREG...141,26,208.......Store value in accumulator in hardware register

LDA LOADDR....169,16..........Load accumulator with Lo-Byte of address of next routine

STA PAGE 2...141,00,02........Store value in accumulator at 512

PLA..............104..........Restore accumulator

RTI...............64..........Return from interrupt.

The next display list interrupt program discussed uses a single machine language routine to access a table of color numbers. This program illustrates indexed addressing, branching, and relative addressing. Before going on to that program, let's review what we have done so far.

The programs in Boxes 18, 19, and 20 have used these machine language commands:

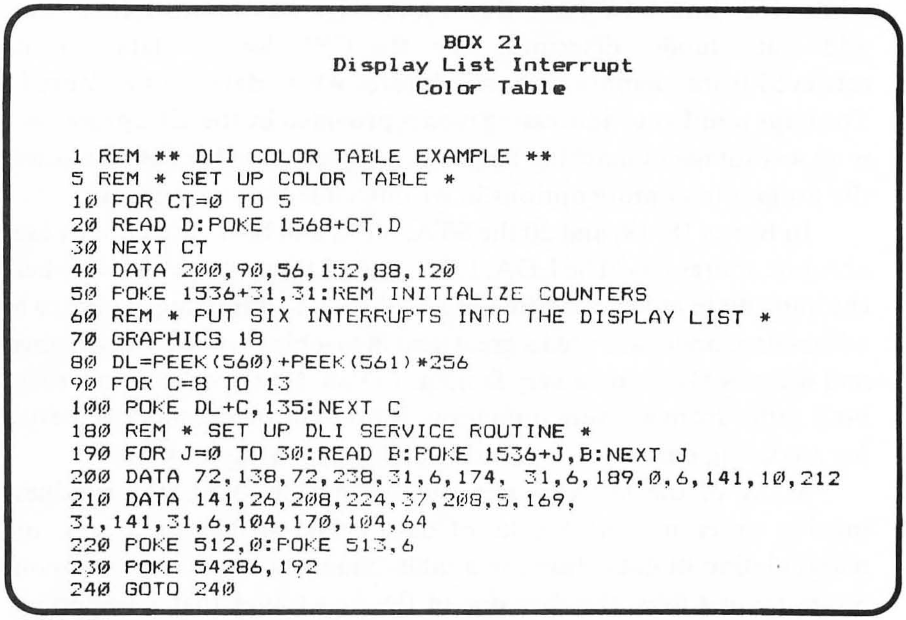
Column I	**Column II**
PHA	LDA
PLA	LDX
TAX	LDY
TAY	STA
TXA	STX
TYA	STY
RTI	

The machine language instructions in column I all use the implied mode of addressing, that is, the instruction itself indicates the source and destination of the byte being moved. However, the instructions in column II each have several different addressing modes. For example, if you look up LDA in Appendix G, you see that it has eight different addressing modes; LDX and LDY each have five addressing modes; while STX and STY have three, and STA has seven. Recall that addressing modes determine how the CPU locates data that is retrieved from memory or how it locates where data is to be stored. The large number of addressing modes provided by the 6502 processor is an advantage in machine language programming because it allows the programmer more options in writing code for a given task.

In Boxes 18, 19, and 20 the STA, STX, and STY instructions use absolute addressing. The LDA, LDX, and LDY instructions use either the immediate mode of addressing or absolute addressing. These two addressing modes are used a great deal in machine language programs and will rapidly become very familiar to you. On the other hand, they both suffer from a severe limitation. Neither of these modes is useful for retrieving data from, or storing data in, an array or table.

Many of the tasks carried out by machine language routines involve either moving blocks of data called strings or arrays, or manipulation of data stored in a table. Since these are very common programming jobs, they are one of the first things that a beginning assembly language programmer should master. Both arrays and tables are stored in contiguous memory locations. Very often what is required is to move a block of data from one location to another or to access the items in a table in sequential order. Since the 6502 is an 8 bit processor, these manipulations occur one byte at a time. As a consequence, the program structure usually is a loop that cycles as many times as there elements in the array.

The DLI routine in Box 21 illustrates the basic elements needed to repeatedly loop through a table. In the program, six display list interrupts are written into a Graphics 2 display list. At each interrupt, the routine loads a color number from a table of values and puts it into the background color register. In order to do this properly, the program needs a way to keep track of its position in the color table and needs to determine when it has reached the end of the color table. The first task is handled by using a pointer and indexed addressing. The second by a compare and branch sequence.



```
                          BOX 21
                  Display List Interrupt
                        Color Table

1 REM ** DLI COLOR TABLE EXAMPLE **
5 REM * SET UP COLOR TABLE *
10 FOR CT=0 TO 5
20 READ D:POKE 1568+CT,D
30 NEXT CT
40 DATA 200,90,56,152,88,120
50 POKE 1536+31,31:REM INITIALIZE COUNTERS
60 REM * PUT SIX INTERRUPTS INTO THE DISPLAY LIST *
70 GRAPHICS 18
80 DL=PEEK(560)+PEEK(561)*256
90 FOR C=8 TO 13
100 POKE DL+C,135:NEXT C
180 REM * SET UP DLI SERVICE ROUTINE *
190 FOR J=0 TO 30:READ B:POKE 1536+J,B:NEXT J
200 DATA 72,138,72,238,31,6,174, 31,6,189,0,6,141,10,212
210 DATA 141,26,208,224,37,208,5,169,
31,141,31,6,104,170,104,64
220 POKE 512,0:POKE 513,6
230 POKE 54286,192
240 GOTO 240
```

BOX 21. Display List Interrupt Color Table

The service routine, pointer, and color table have been loaded into page six as follows:

- 31 bytes of machine language routine at locations 1536 through 1566.
- The pointer at location 1567.
- 6 bytes of the color table at 1568 through 1573.

The service routine may be split into sections as follows:

Section I: *Save registers to stack and increment the pointer.*

MNEMONIC	DECIMAL VALUE	FUNCTION
PHA	72	Push accumulator to stack
TXA	138	Transfer X-register to accumulator

PHA	72	Save X-register on stack
INC POINTER	238,31,6	Increment Pointer

This last instruction is carried out each time the interrupt is used. The pointer was initialized to 31 in line 50 of the program so that when the first DLI is encountered, this instruction will increment the pointer to 32, the Lo-Byte of the first color number's address.

Section II: *Load the X-register. Get the color value.*

LDX POINTER	174,31,6	Load the X-register with the value in pointer
LDA ADDR,X	189,0,6	Load accumulator with the value in the addr following using the X-register as an index

When the LDA command is executed, the zero in 0,6 is added to the value in the X-register to get the Lo-Byte of the address of the color number in the table.

Section III: *Change the color register*

STA WSYNC	141,10,212	Wait for the horizontal blank
STA COLREG	141,26,208	Store color in color register

Section IV: *Test for end of the table*

CPX NUM	224,37	Compare the value in the X-register with 37
BNE END	208,5	Branch if not equal to zero

If the result of the CPX command is not equal to zero, the program jumps ahead to restore the X-register and then returns from the interrupt, thus by-passing the next two instructions.

Section V:

```
LDA RESET       169,31     Load the accumulator with 31

STA POINTER    141,31,6    Store 31 to reset pointer
```

Section VI: *Exit the routine sequence*

```
END PLA           104      Pull top of stack into the accumulator

TAX               170      Transfer accumulator to X-register

PLA               104      Restore accumulator

RTI                64      Return from interrupt
```

Here is how the program works. Each time a DLI is encountered the accumulator and X-register values are stored on the stack. The pointer is incremented and this value is placed in the X-register. The accumulator is then loaded with a color value at an address, the Lo-Byte of which is the value in the X-register. After this value has been stored in the background color register, the program tests to see if the address used was that of the last color value. If not, the routine ends for that interrupt. If it is the last color, then the pointer is reset before the routine ends.

This program shows that when dealing with a block of data or a table there are three things that must be done: (1) A counter, or index register must be initialized. (2) The index register or counter must be incremented (or decremented) as the program works through the data. (3) The register or counter must be tested for the end of the block. These tasks are common to all programs that are moving or manipulating a block of data. The exact implementation depends on the job to be done and the programmer's inclination. For example, in this program, where each pass made through the loop by the processor is separated from every other pass with some other processor activity, an independent counter kept in memory was required. When a block of data is being manipulated all at once, that is, when the loop is cycled through continuously without interruption then it is often possible to use one of the index registers as a counter. The next program we will discuss does this.

The test to see if a loop has been executed the proper number of times can be performed in several ways. Using CPX or CPY followed by a BEQ or BNE is one method. But if you study the branching process you can easily invent other ways to test for the end of an array. Branch instructions occur in response to the state of certain bits in the processor status register (see Box 1). The instructions BEQ and BNE take action according to whether the zero flag is set or reset. BMI and BPL take action according to whether the N flag is set or reset. The flags in the status register are controlled by certain of the 6502's instructions. Appendix F lists those instructions that affect status flags.

For example, referring to Appendix F, we see that DEX changes the N flag and the Z flag. Therefore you could set up the X-register as a counter and decrement it down to zero and use the BEQ to branch out of the loop.

Program Listing

Before proceeding with the development of programs let's briefly consider some conventions used for writing assembly language programs. If you are familiar with more than one higher level language, such as BASIC, PASCAL, and FORTRAN, then you know that each language has rules as to how to construct a program - line numbers, special punctuation, and things of this nature. Strictly speaking, a machine language program is nothing more than a sequence of bytes in the computer's memory and so the format of how to write out the program on paper is pretty much left to personal choice. In many of the programs that follow we will use a format for listing programs that is similar to that used by The Atari Assembler Editor Cartridge and many other assembler programs. In this format there are six columns or fields as follows:

Label	**Opcode Mnemonic**	**Operand Mnemonic**	**Numeric Opcode**	**Numeric Operand**	**Comment**

Optionally, a seventh field could be used, the line number preceeding the label field. There are some other conventions regarding notation and indexing that we shall adhere to. They are:

- Hexadecimal numbers will be preceeded with $

- Immediate operands will be preceeded with #

- Absolute jndexed operands will be indicated by ————————,X or —————————,Y

- Nonindexed indirect will be indicated by parenthesis ie. JMP($6000)

- Indexed indirect will be indicated by (——————,X) ie. INC($99,X)

- Indirect indexed, which uses the Y-register only will be indicated by (——————),Y. ie. LDA ($2B),Y

- Indexed page zero will be indicated by——————,X or —————————,Y but the number ——— will, of course, be less than 256

USR

The USR command is one of the handiest ways to integrate machine language subroutines into a BASIC program. The command is structured so that parameters can easily be passed from the BASIC program to the subroutine. The USR command has the format:

```
DUMMY=USR(ADDR,parameter 1,parameter 2...)
```

DUMMY stands for 'dummy variable' which implies that it is necessary for the format of the command, but that no useful value is returned. ADDR is either the decimal value of the address where the subroutine is stored, or is an expression that evaluates to the address. Parameters are passed to the subroutine via the stack. The stack structure for a USR command is:

TOP OF STACK N - the number of parameters (may be 0)
Hi-Byte of parameter 1
Lo-Byte of parameter 1
Hi-Byte of parameter 2
Lo-Byte of parameter 2

■
■
■

BOTTOM OF STACK Lo-Byte of return address

Hi-Byte of return address

The stack structure forces at least one PLA instruction in every routine called by a USR. If there are parameters, "N" must be removed with a PLA before the parameters can be accessed. If there are no parameters, then N is zero and must be removed before the return address can be accessed.

Strings

The USR and Atari BASIC's string handling capabilities provide a good way to store machine language routines safely within a BASIC program. The idea is to translate the decimal numbers representing the subroutine into ATASCII characters and store these in a string. As we'll see shortly, the routine can be easily addressed by USR.

Storing machine language routines in strings has several advantages. First, it avoids memory management problems by turning the job over to BASIC. Second, the length of the machine language program is not limited as it is with page six storage. Third, the string method of storage is more efficient in terms of time and space. Spacewise, the data for the machine language routine is stored as a single symbol in the string rather than as a sequence of numerals and commas. Timewise, efficiency is achieved because string storage eliminates time-consuming READ and POKE sequences such as those in lines 50 to 90 of Box 14.

Our next program combines many of the ideas discussed so far in this chapter into one example. In this program (Box 22), machine language routines are used to speed up redefinition of the character set in program 14. The machine language routines that move and redefine the standard character set are stored as strings in lines 30 and 40. Also, the data used to redefine the characters as a cat is stored as a string in line 50. These simple changes in program 14 yield a fantastic increase in execution speed, as you will see when you type in and run the program.

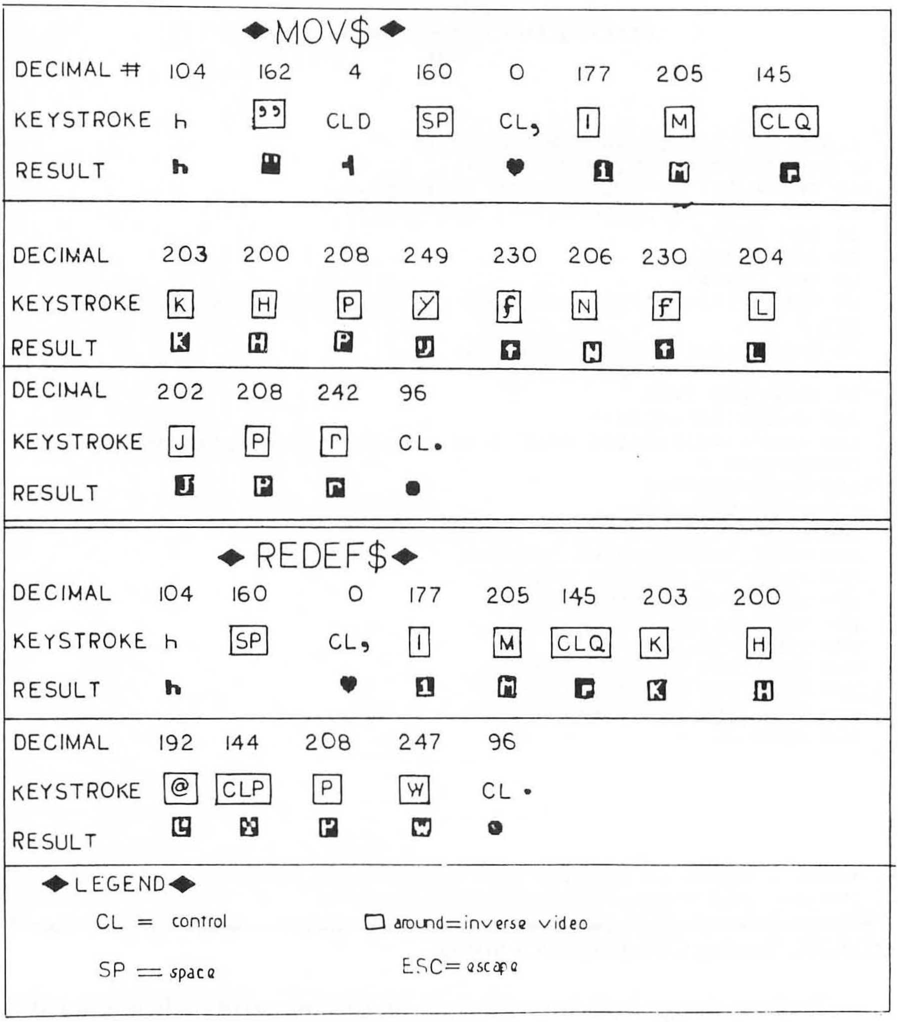
Storing subroutines in strings does entail some extra steps and occasionally a little inconvenience. Consider the subroutine MOV$ which consists of twenty decimal numbers. To store these as a string it is necessary to convert the decimal numbers into character symbols using Appendix E. As an example, for the first four numbers of MOV$:

DECIMAL #	104	162	4	160
KEYSTROKE	h	inverse "	CTL D	inverse space
RESULT	**h**	▣	┤	

```
                          BOX 22
              Storing Characters as a String
                            Cat


1 REM ** FAST CHARACTER CHANGE **
5 REM * COMPARE WITH BOX 14 *
10 REM * SET UP ROUTINES IN STRINGS *
20 DIM MOV$(20),REDEF$(14), CHAR$(104)
30 REM MOV$=
40 REM REDEF$=
50 REM CHAR$=
60 REM * INITIALIZE PAGE ZERO LOCATIONS AND MOVE CHARACTER
SET *
70 A=PEEK(106)-8:POKE 106,A+4
80 POKE 204,A+4:POKE 206,224
90 GRAPHICS 2+16
100 M=USR(ADR(MOV$))
110 REM * INITIALIZE PAGE ZERO LOCATIONS AND REDEFINE
CHARACTERS *
120 Q=ADR(CHAR$)
130 HIQ=INT(Q/256)
140 LOQ=Q-HIQ*256
150 POKE 205,LOQ:POKE 206,HIQ
160 POKE 203,24:POKE 204,A+4
170 R=USR(ADR(REDEF$))
180 REM * DISPLAY ON SCREEN *
190 POKE 756,A+4
200 POSITION 9,1:? #6;"#$%"
210 POSITION 8,2:? #6;"&'()*"
220 POSITION 8,3:? #6;"+,-./"
230 GOTO 230




**NOTE: CHAR$ is developed from the data numbers in Box 14 by
the procedure described in the text.
```

BOX 22. Storing Characters as a String

Storing things in strings can be a bit disconcerting when you list the program on a printer. Very often some of the symbols in the string will be interpreted by the printer as control codes, causing it to do very strange things. Even if the string doesn't cause printer pollution, you may find that not all of the characters print out. Our way around these problems is illustrated in Boxes 22 and 23. As you have discovered by now, we replace the strings with REM statements before attempting a listing and write a separate listing for the machine language routine using the notation defined in Box 23.



◆MOV$◆

DECIMAL #	104	162	4	160	0	177	205	145
KEYSTROKE	h	"	CLD	SP	CL,	I	M	CLQ
RESULT	h	"	┤		♥	1	M	┌

DECIMAL	203	200	208	249	230	206	230	204
KEYSTROKE	K	H	P	y	f	N	f	L
RESULT	K	H	P	y	f	N	f	L

DECIMAL	202	208	242	96
KEYSTROKE	J	P	r	CL.
RESULT	J	P	r	●

◆REDEF$◆

DECIMAL	104	160	0	177	205	145	203	200
KEYSTROKE	h	SP	CL,	I	M	CLQ	K	H
RESULT	h		♥	1	M	┌	K	H

DECIMAL	192	144	208	247	96
KEYSTROKE	@	CLP	P	W	CL.
RESULT	@	♣	P	w	●

◆LEGEND◆

CL = control

☐ around=inverse video

SP = space

ESC= escape

BOX 23. Notation used to depict strings

Let's use the subroutine MOVE$ to examine, in more detail, the process of moving data from one area of memory to another. In order to write a program that moves data in memory, you must have a way to locate each byte to be moved and a way to determine where each byte is going. This concept is presented schematically in Figure 4-1:

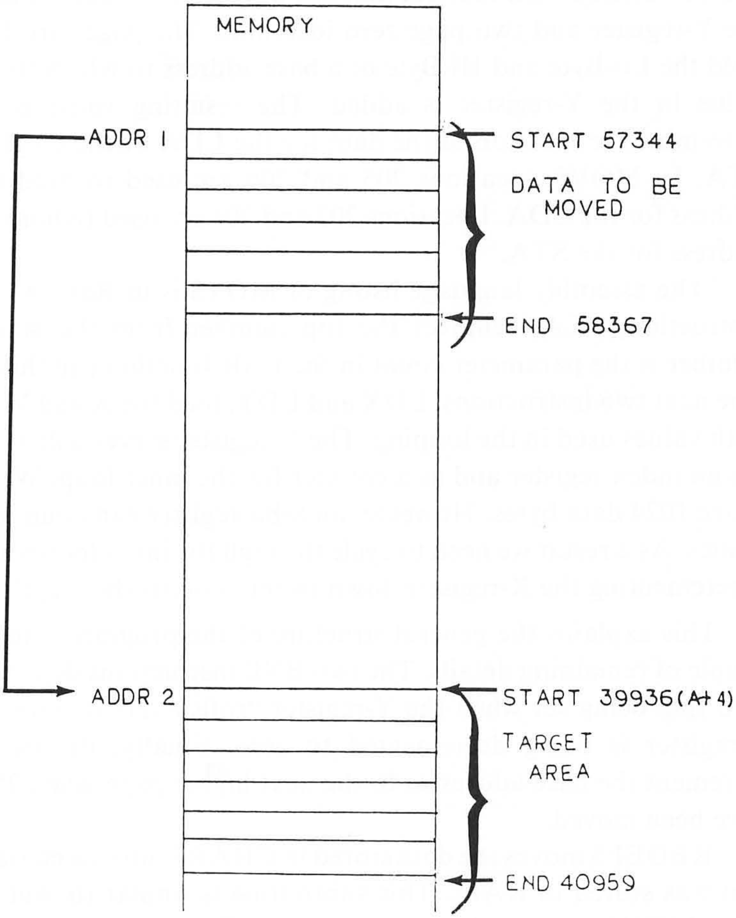


Figure 4-1. Subroutine MOVE$ presented schematically

You also need a way to determine when you have moved all your data. There are several ways to solve each of these programming problems and sometimes they can be handled simultaneously by use of indexed addressing. Such is the case here, where we want to move 1024 bytes of character data from ROM to RAM. The source and target addresses can be handled with indirect indexed addressing which makes use of the Y-register and two page zero locations. The page zero locations hold the Lo-Byte and Hi-Byte of a base address to which the current value in the Y-register is added. The resulting value is used to determine the location of the data for the LDA or the target for the STA. In MOV$, locations 205 and 206 are used to hold the base address for the LDA. Locations 203 and 204 are used to hold the base address for the STA.

The assembly language listing of MOV$ is in Box 24. The first instruction, PLA, removes the top number from the stack. This number is the parameter count in the USR function (in this case 0). The next two instructions, LDX and LDY, load the X and Y registers with values used in the looping. The Y-register serves a dual purpose -as an index register and as a counter for the inner loop. We have to move 1024 data bytes. However, an 8-bit register can count only 256 values. As a result we need to cycle through the inner loop four times. Decrementing the X-register down to zero counts these cycles.

This explains the general structure of the program. There are a couple of remaining details. The two BNE instructions depend on the zero flag being set when the Y-register "rolls over" to zero and the X-register is being decremented to zero. Finally, the two INC's increment the base addresses to the next higher page when 256 bytes have been moved.

REDEF$ moves the data stored in CHAR$ into the character set that was stored in RAM. This subroutine is similar to but simpler than MOV$ so we will leave it to you to figure out the details. The assembly language listings are in Boxes 25, 25A, and 25B.

Box 24

MOV$

203,206 Base Address (where to get it)
203,204 Base Address (where to put it)

LABEL	MNEMONIC	OPERAND	OPCODE NUMBER	OPERAND NUMBER	COMMENT
	PLA		104		Remove parameter count
	LDX	#4	162	4	Load X with number of inner loops
	LDY	#0			Load Y with initial value
LOOP	LDA	(ADDR1),Y	177	(205),Y	Load accumulator from base addr Locations 205,206 plus value in Y
	STA	(ADDR2),Y	145	(203),Y	Store accumulator from base locations 203,204 plus value in Y
	INY		200		Increment Y to point to next byte
ENDLP1	BNE	LOOP	208	249	If not done, then continue
	INC	206	230	206	Increment base address to next higher page when 256 bytes have moved
	INC	204	230	204	
	DEX		202		Decrement X, counting off one inner loop completed.
ENDLP2	BNE	LOOP	208	242	If not done, then continue
	RTS		96		Return from subroutine

BOX 24. MOV$

REDEF$

Less than 256 Bytes

LABEL	MNEMONIC	OPERAND	OPCODE NUMBER	OPERAND NUMBER	COMMENT
	PLA		104		Remove parameter count
	LDY	#0	169	0	Initialize Y-register
LOOP	LDA	(ADDR1),Y	177	205	Load accumulator from base + Y
	STA	(ADDR2),Y	145	203	Store accumulator at base + Y
	INY		200		Increment Y to point to next Byte
	CPY	#144	192	144	Compare contents of Y register to 144
	BNE	LOOP	208	247	If move not complete, then continue
	RTS		96		Otherwise return from subroutine.

BOX 25. Assembly Language Listings

REDEF$

More than 256 Bytes

LABEL	Mnemonic	OPERAND	OPCODE NUMBER	OPERAND NUMBER	COMMENT
	PLA				
	LDY	#0			
LOOP	LDA	(205),Y			
	STA	(203),Y			
	INY				
	BNE	LOOP			
	INC	204			
	INC	206			
LOOPB	LDA	(205),Y			
	STA	(203),Y			
	INY				
	CPY	REMAINDER			
	BNE	LOOPB			
	RTS				

The simplest way to redefine more than 256 bytes is to move a multiple of 256 first and then move the remainder. We have left the Opcodes, Operands, and Comments for you to fill in.

BOX 25A. REDEF$ (for more than 256 bytes)

REDEF$

More than 512 Bytes

LABEL	MNEMONIC	OPERAND	OPCODE NUMBER	OPERAND NUMBER	COMMENT
	PLA				
	LDY	#0			
	LDX	#MULTIPLE			
	LDA	(205),Y			
	STA	(203),Y			
	INY				
	BNE	LOOPA			
	INC	204			
	INC	206			
	DEX				
	BNE	LOOPA			
	LDA	(205),Y			
	STA	(203),Y			
	INY				
	CPY	REMAINDER			
	BNE	LOOPB			
	RTS				

The simplest way to redefine more than 512 bytes is to move multiples of 256 first and then move the remainder. We have left the Operands, Opcodes, Operands, and Comments for you to fill in.

Box 25B. REDEF$ (for 512 or more bytes)

More on Branching

If you have an assembler program then calculating the address used in a branch instruction is no problem. You identify the target instruction with a label and use that label as the operand of the branch. The assembler program then calculates the necessary relative address. On the other hand, calculating relative addresses without an assembler is a bit tricky the first few times you do it. So far we have seen a branch forward (in Box 21) and a couple of branches backward. Here is the branch forward section of Box 21:

```
CPX NUM        224  37
BNE END        208  5
LDA RESET      169  31 Start counting here
STA POINTER    141  31  6
END PLA        104←Target
```

The relative address for the BNE is 5. But, "5", from where? The target is the PLA. The number following the BNE is added to the program counter to get the address of the target instruction. A little counting tells us that the program counter was pointing to LDA when 5 was added to it. *Thus, the opcode following the branch operand is the starting point for counting in relative addressing.* Notice that all the numbers between this opcode and the target area are counted.

Consider a segment of the code for MOV$:

```
LABEL    MNEMONIC   OPERAND    OPCODE     OPERAND
                               NUMBER     NUMBER

LOOP       LDA       ADDR1       177        205
           STA       ADDR2       145        203
           INY                   200
ENDPL 1    BNE       LOOP ←      208        249
           INC        206        230        206
```

How do we calculate the operand of the BNE instruction? Starting at the INC (marked with an arrow) count backwards, to the LDA marked LOOP, the result is 7. Subtract 7 from 256 and you get 249, the operand of BNE. To branch forward you count upwards from zero. To branch backwards you count down from 256. Of course, the counting forward and backward will eventually meet. Consequently, you can branch forward a maximum of 127 bytes and backwards a maximum of 128 bytes.

Passing Parameters to Subroutines

We can use the program in Box 26 to understand the details of passing parameters from a USR instruction to a machine language subroutine. The machine language subroutine is in lines 80 - 90 and is used to move the lightbulb down the screen. A single parameter, the current Y position of the bottom of the lightbulb player, is passed to the subroutine in the USR command in line 135. Written out, the subroutine is:

LABEL	OPCODE	OPERAND	NUMERIC OPCODE	NUMERIC OPERAND	COMMENT
	PLA		104		Remove parameter count
	PLA		104		Get Hi-Byte and Lo-Byte of player's position and store in page zero
	STA	POS-1	133	204	
	PLA		104		
	STA	POS-2	133	203	
	LDA	#10	160	10	
LOOP	LDA	(POS-2),Y	177	203	Indirect indexed addr
	INY		200		INC Y in order to move the byte down
	STA	(POS-2),Y	145	203	
	DEY		136		DEC Y twice in order to fetch next player byte

```
DEY                 136
CPY    ENOUGH       192      255
BNE     LOOP        208      245
RTS                  96
```

You can see from the listing that the Hi-Byte and Lo-Byte of the Y-position are pulled off the stack and put into page zero locations 203 and 204 respectively. Page zero locations 203 to 209 are free to use in subroutines and probably will be sufficient in most cases. However, if neither the machine language routine nor the BASIC part of your program use the floating point package, then page zero locations from 212 to 255 are also free to use.

Study the structure of the loop portion of this subroutine because it will also appear in the next program example. The object is to move each byte, of a group, up one location in memory. This moves the player down on the screen. The loop starts with an indirect indexed load to get the byte at the base of the player. Incrementing the Y-register and using indirect indexed addressing with STA moves this byte up one memory location. Then the Y-register is decremented twice in order to get the next byte. However, before branching back to LDA a CPY instruction checks to see if all the bytes have been moved. If they have, the result of the CPY is zero, the Z flag is set, and the subroutine ends. The program will loop 256 times before control is returned to the BASIC program.

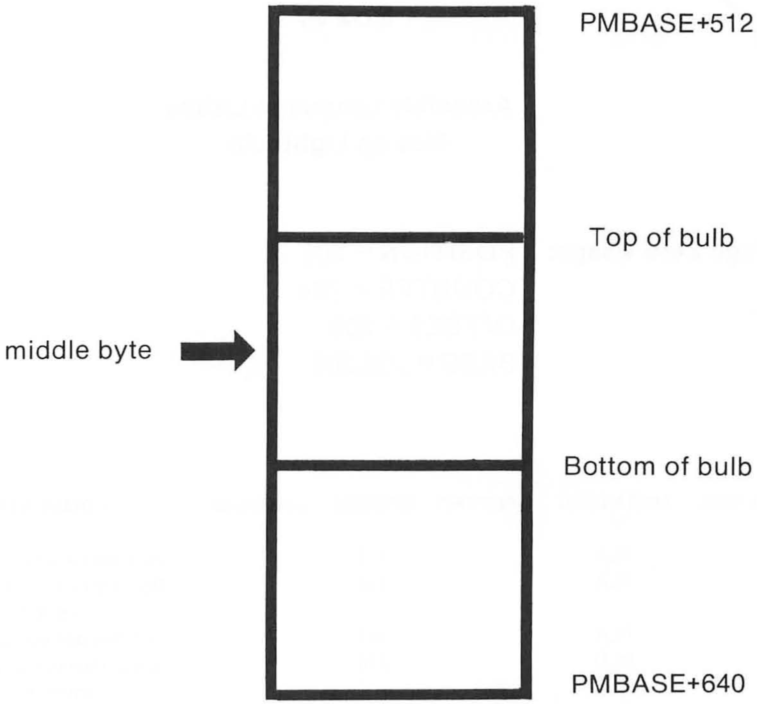
Because the next program (Box 26) is considerably more complicated than our previous program examples, we will discuss in general terms the program's structure before looking at the assembly language code. Pedagogically, the program illustrates the use of simple binary arithmetic which also involves two's complement arithmetic.

BOX 26
Player Movement

```
5 REM ** MOVING PLAYER **
10 A=PEEK(106)-8:POKE 106,A:REM * MOVE RAMTOP *
20 GRAPHICS 2
30 POKE 54279,A:REM * SET PMBASE *
40 PBASE=A*256
45 REM * CLEAR PLAYER MEMORY *
50 FOR I=PBASE+512 TO PBASE+640:POKE I,0:NEXT I
55 REM * READ PLAYER DATA INTO THE MIDDLE OF PLAYER 0 SECTION
*
60 FOR I=1 TO 9
70 READ D:POKE PBASE+562+I,D
80 NEXT I
90 DATA 60,126,255,255,126,60,24, 24,24
100 POKE 53248,120:REM SET HORIZONTAL POSITION
110 POKE 704,88:REM * SET THE COLOR *
115 REM * TURN ON THE PLAYER *
120 POKE 559,46:POKE 53277,3
130 FOR I=0 TO 89:READ ML:POKE 1536+I,ML:NEXT I
140 DATA 104,104,104,216,56,229,
203,240,43,16,42,73,255,24,105,1,133,204,165,203,56,233,4,133
,205,164,205,162,11
150 DATA 177,206,136,145,206,200,
200,202,208,246,198,205,198,204,208,236,165,205,24,105,4,133,
203,96,133,204
160 DATA 165,203,24,105,4,133,205,
164,205,162,11,177,206,200,145,206,136,136,202
170 DATA 208,246,230,205,198,204,
208,236,165,205,56,233,4,133,203,96
180 POKE 203,54:POKE 206,0:POKE 207,A+2
200 PRINT "CHOOSE A POSITION FROM 8 TO 120"
210 INPUT YPO
220 IF YPO<8 OR YPO>120 THEN GOTO 200
230 DUMMY=USR(1536,YPO)
240 GOTO 200
```

BOX 26. Player movement

In the program the user is asked to input a number between 8 and 120. This number defines a position to which the machine language in lines 140 to 175 will move the lightbulb player. The first task that the subroutine must do is determine whether to move the lightbulb up, down, or not at all. This means that the position number, input by the user, must be passed to the subroutine and compared with the bulb's current position. Physically, the lightbulb is a group of nine bytes located somewhere in the 128 bytes of Player 0.



The bulb's current position is kept track of by noting the location of the center byte. If the bulb is to be moved up the screen - which corresponds to bytes being moved to smaller values in memory - the movement routine will start at the top of the player. If the player is to be moved down the screen, the movement routine will begin at the bottom of the player.

The movement decision is made by subtracting the current position from the new position and looking at the result. If the result is equal to zero, then no change is called for and the subroutine ends. If the result is greater than zero, the player moves up. If the result is less than zero, the player moves down. The distance that the bulb moves is equal to the difference between the old and new position values. That figure conveniently serves as a loop counter. Since movement begins at either the top or the bottom of the player, the necessary value will have to be calculated from the current position before the player can be moved. Finally, after the movement is complete, the location of the center byte must be calculated and retained.

Box 27

Assembly Language Listing Moving Lightbulb

Page Zero Usage:

POSITION = 203
COUNTER = 204
OFFSET = 205
BASE = 206,207

LABEL	MNEMONIC	OPERAND	OPCODE	OPERAND	COMMENTS
	PLA		104		Remove parameter count
	PLA		104		Remove Position Hi-Byte, discard
	PLA		104		Remove position Lo-Byte
	CLD		216		Clear decimal for binary arithmetic
	SEC		56		Set carry bit for a
	SBC	POSITION	229	203	Subtract with borrow
	BEQ	END	240	43	If result is zero, don't move
	BPL	DOWN	16	42	If result positive move player down
	EOR	#255	73	255	Result must be negative. Change it to a positive number
	CLC		24		
	ADC	#1	105	1	
	STA	COUNTER	133	204	Save the distance to move player
	LDA	POSITION	165	203	Obtain current position
	SEC		56		Determine the location of the top of the player. This is The Y-OFFSET
	SBC	#4	233	4	
	STA	OFFSET	133	205	
OUTRLP	LDY	OFFSET	164	205	Load Y-register with the offset
	LDX	#11	162	11	Load X-register with the byte count

BOX 27. Assembly Language Listing Moving Lightbulb

INNRLP	LDA	BASE,Y	177	206	Load accumulator with player byte
	DEY		136		Change Y-register
	STA	BASE,Y	145	206	Store byte in new memory location
	INY		200		Go back for
	INY		200		next byte
	DEX		202		One byte of player was moved
	BNE	INNRLP	208	246	Are all bytes moved?
	DEC	OFFSET	198	205	If so, adjust offset to player top
	DEC	COUNTER	198	204	Check off whole player moved once
	BNE	OUTRLP	208	236	Is move complete?
	LDA	OFFSET	165	205	Move is complete
	CLC		24		So update the position register
	ADC	#4	105	4	
	STA	POSITION	133	203	
END	RTS		96		Return from move routine
DOWN	STA	COUNTER	133	204	Save distance to move player
	LDA	POSITION	165	203	Obtain current position
	CLC		24		Determine the location of the bottom of the player. This is the Y-OFFSET
	ADC	#4	105	4	
	STA	OFFSET	133	205	
OUTRLP	LDY	OFFSET	164	205	Load X-register with the offset
	LDX	#11	162	11	Load X-register with the byte count
INNRLP	LDA	BASE,Y	177	206	Load Accumulator with player byte
	INY		200		Change Y-register
	STA	BASE,Y	145	206	Store byte in new memory location
	DEY		136		GO back for
	DEY		136		next byte
	DEX		202		Check off one byte was moved
	BNE	INNRLP	208	246	Are all bytes moved?
	INC	OFFSET	230	205	If so, reset offset to player top
	DEC	COUNTER	198	204	Check off player was moved once
	BNE	OUTRLP	208	236	Is move Complete?
	LDA	OFFSET	165	205	Move is complete so
	SEC		56		update the
	SBC	#4	233	4	position
	STA	POSITION	133	203	register
	RTS		96		Return from move routine

Box 27. (Cont.)

Arithmetic Instructions

The arithmetic instructions used in this routine are simple single byte binary instructions. To see them in action, look at Box 27 which is the assembly language of the 'Moving Lightbulb' program. The first two PLA's remove the parameter count and Hi-Byte of the player position from the stack. This Hi-Byte will always be zero, so we don't need it. The third PLA removes the byte we need and holds it in the accumulator.

The 6502 CPU can perform two types of arithmetic, binary and binary coded decimal. Binary arithmetic can be treated as signed or unsigned. In this book we will only concern ourselves with binary arithmetic. Therefore, the first instruction we use is CLD (Clear Decimal mode). This instruction *must always precede* a binary arithmetic sequence. This done, we are ready to subtract the current position from the new position. The value of the bulb's current position is in memory location 203; the value of the new position is in the accumulator.

The 6502 subtract instruction is SBC (SuBtract with Carry) which actually means subtract using the carry flag as a borrow digit, if necessary. This allows for the possibility that the number being subtracted from the accumulator is bigger than the number in the accumulator. Since SBC needs the carry flag, it must be preceded by SEC in all cases. SBC, like LDA, has eight different addressing modes. In this program we use zero mode.

Having subtracted the current position from the new position, it is necessary to determine which direction to move the player. BEQ END takes care of the case in which there is no movement. BPL DOWN sends program control to the section of the routine that moves the player down. If neither of these conditions occur, then the routine goes about moving the player up.

Now we come to a somewhat technical topic. The 6502 performs its subtraction by addition! The method used is known as two's complement arithmetic. The practical consequence here is, that if the result of the subtraction is a negative number, then the number left in the accumulator will not be in ordinary binary form, but rather in two's compliment form. Accordingly, the result will have to be converted to its positive equivalent before we can use it as a loop counter.

Two's complement arithmetic works as follows. Suppose you are adding the negative of an ordinary decimal number to itself, say 2+(-2). The result is, of course, zero. Now, suppose that you add the binary number 1 and the binary number 255. The result in an eight bit register is zero. In binary form the result is:

$$\begin{array}{ll} & 0\,0\,0\,0\quad 0\,0\,0\,1 \\ & \underline{1\,1\,1\,1\quad 1\,1\,1\,1} \\ \underset{\text{a ninth bit}}{1} & \underbrace{0\,0\,0\,0\quad 0\,0\,0\,0}_{\text{8 bits, all zero stored in register}} \end{array}$$

You will get the same result if you add 2 and 254 in binary form. Or if you add 3 and 253. Or 4 and 252. By now you should see the pattern. Every positive number up to 128 has a corresponding number (its negative) that, when added to it, gives 256. Which, as far as an eight bit register like the accumulator is concerned, is really zero.

Now, what does this mean for the subroutine? Well, if we were to subtract an input position value of 80 from a current position value of 42, the result left in the accumulator would be the two's compliment equivalent of minus 38, or 218 (in binary from 1101 1010). To use this as a counter in the subroutine, it must be converted to a positive number.

The algorithm for changing a number into its two's compliment form is very simple. All you do is change every zero to a one and every one to a zero and then add one. The algorithm works both ways. Consider our example of 42 minus 80. The result, in decimal is -38. The result in 6502 subtraction is 11011010 which has the decimal equivalent of 218. Parenthetically, 218+38=256, as we would expect. By decimal arithmetic, we have concluded that 218 must be the negative of 38. Let's apply the algorithm:

1 1 0 1 1 0 1 0 (218)

0 0 1 0 0 1 0 1 Flip bits

+1 Add 1

0 0 1 0 0 1 1 0 = 32 + 6 = 38

Suppose we wanted to go the other way:

0 0 1 0 0 1 1 0 = 38

1 1 0 1 1 0 0 1 Flip bits

+1

1 1 0 1 1 0 1 0 = 218

The 6502 instruction set has a very handy command that allows you to change one's to zero and vice versa. It is the EOR (review Box 3) command. To flip all the digits of a number all you have to do is to EOR the number with 255. For example:

1 0 1 1 0 1 1 1 = 183

1 1 1 1 1 1 1 1 EOR with 255

0 1 0 0 1 0 0 0

Finally, to make it a two's complement, add 1:

0 1 0 0 1 0 0 0

+1

0 1 0 0 1 0 0 1 two's complement of 183

In conjunction with this discussion of two's complement arithmetic, remember that branch instructions can go backwards 128 steps. Backward branches use negative numbers written in two's complement form.

In the assembly listing, the EOR command follows the BPL test, which, if true, sends the program to the section which moves the player down. Look at the two instructions following EOR. These instructions are required since our program needs to add 1 to the result of the EORing. The 6502 add instruction is ADd with Carry (ADC) and as the name implies, any carry from the accumulator will go over into the carry bit. Consequently, before using an ADC, you must first clear the carry flag with CLC. Once this is done you are free to ADC.

Following the EOR-CLC-ADC sequence, the up-routine proper begins with STA counter. From this point on, the up and down routines are mirror images of each other. Therefore, we'll comment only on the up routine. Unlike the previous move routine, this one uses a double loop structure. The reason is it moves only eleven of the player's 128 bytes. Eleven bytes are moved rather than just the nine bytes that actually form the lightbulb player, so that blanks can be moved into the position where the player originated thereby preventing it from leaving a trail as it moves up the screen. Additional things to note are: that the offset used by the Y-register is decremented at the completion of the inner loop; and that at the completion of the move, the new position is calculated by adding four to the offset.

Up to this point we have illustrated a good portion of the 6502 instruction set, introduced several fundamental programming concepts, and discussed basic arithmetic operations. We shall conclude this chapter by presenting a subroutine that moves missiles and in the process demonstrates logic and shift instructions in action.

We have seen that XOR can be used to complement bits. The other two logic commands - AND and ORA - can be used to test, clear, or set specific bits in memory. AND and ORA make use of two bytes, one in memory and one in the accumulator. Bit by bit comparisions of the two numbers are carried out according to the logic rules described in chapter one. The result is stored back in the accumulator and the sign and zero flags are set, if appropriate.

Using the AND/ORA instructions in a missile move routine is dictated by the way missiles are represented in memory. The four missiles, M0 through M3, are located side by side in player-missile memory at PMBASE384 or PMBASE768:

Missile	M3		M2		M1		M0	
Bit	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Decimal	128	64	32	16	8	4	2	1

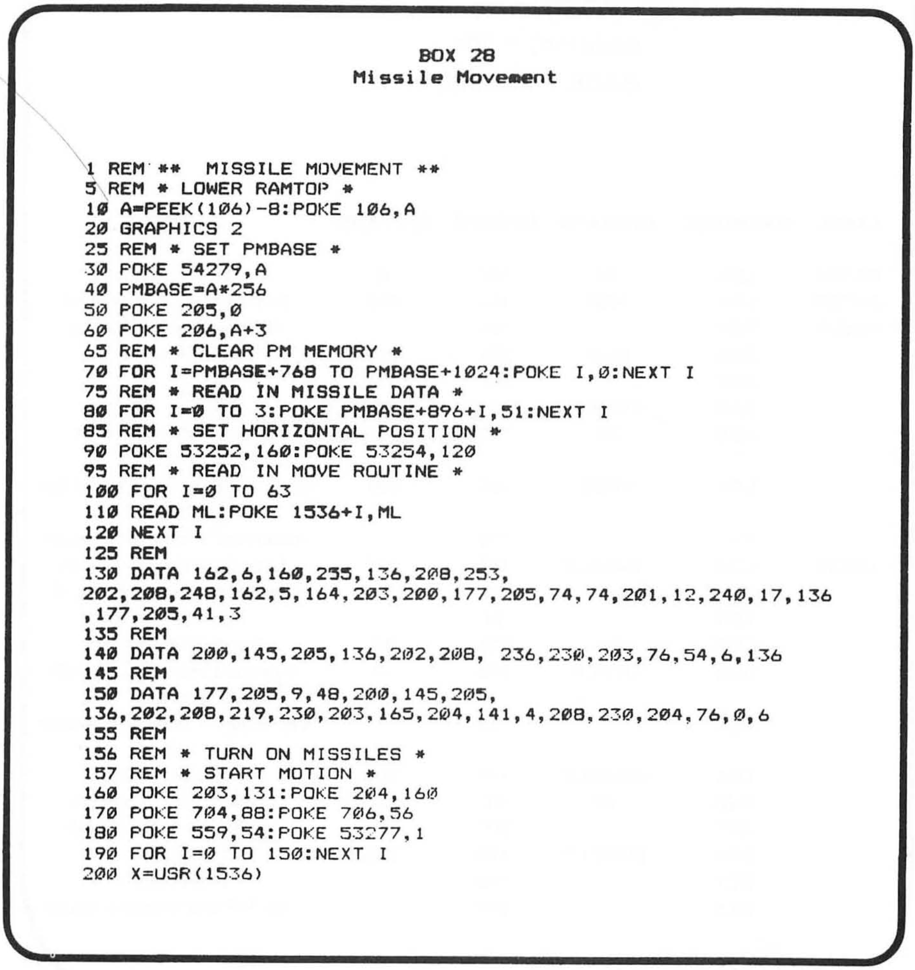
Now, if missiles M0 and M2 exist as:

	M2		M0
00	11	00	11
00	11	00	11
00	11	00	11
00	11	00	11

and you want to move M0 without disturbing M2, it is necessary to mask out, not erase, M2's bits. This masking is accomplished by ANDing at the appropriate time. On the other hand, if the missiles exist in memory as:

	M2		M0
00	00	00	11
00	00	00	11
00	00	00	11
00	00	00	11
00	11	00	00
00	11	00	00
00	11	00	00
00	11	00	00

and you want to move M0 next to M2, you have to take care that M2's bits are not wiped out in the process. This is accomplished by masking with AND and unmasking with OR. Of course the same comments hold true if it's M2 that is being moved rather than M0.



```
BOX 28
Missile Movement

1 REM ** MISSILE MOVEMENT **
5 REM * LOWER RAMTOP *
10 A=PEEK(106)-8:POKE 106,A
20 GRAPHICS 2
25 REM * SET PMBASE *
30 POKE 54279,A
40 PMBASE=A*256
50 POKE 205,0
60 POKE 206,A+3
65 REM * CLEAR PM MEMORY *
70 FOR I=PMBASE+768 TO PMBASE+1024:POKE I,0:NEXT I
75 REM * READ IN MISSILE DATA *
80 FOR I=0 TO 3:POKE PMBASE+896+I,51:NEXT I
85 REM * SET HORIZONTAL POSITION *
90 POKE 53252,160:POKE 53254,120
95 REM * READ IN MOVE ROUTINE *
100 FOR I=0 TO 63
110 READ ML:POKE 1536+I,ML
120 NEXT I
125 REM
130 DATA 162,6,160,255,136,208,253,
202,208,248,162,5,164,203,200,177,205,74,74,201,12,240,17,136
,177,205,41,3
135 REM
140 DATA 200,145,205,136,202,208, 236,230,203,76,54,6,136
145 REM
150 DATA 177,205,9,48,200,145,205,
136,202,208,219,230,203,165,204,141,4,208,230,204,76,0,6
155 REM
156 REM * TURN ON MISSILES *
157 REM * START MOTION *
160 POKE 203,131:POKE 204,160
170 POKE 704,88:POKE 706,56
180 POKE 559,54:POKE 53277,1
190 FOR I=0 TO 150:NEXT I
200 X=USR(1536)
```

Box 28. Missile movement

BOX 29

Assembly Language Listing
Moving Missiles

Register Usage: VPOS (Vertical position) = 203
HPOS(Horizontal position) = 204
BASE = 205,206

6

LABEL	MNEMONIC	OPERAND	OPCODE	OPERAND	COMMENT
BEGIN	LDX	#4	162	6	Delay action
OUTER	LDY	#255	160	255	to slow missile movement
INNER	DEY		136		to speed the human eye
	BNE	Inner	208	253	can perceive
	DEX		202		
	BNE	OUTER	208	248	
	LDX	#5	162	5	Load X with number of segments + 1
	LDY	VPOS	164	203	Load Y with location of bottom segments
	INY		200		Increment Y to target position
LOOP	LDA	(BASE),Y	177	205	Load from target position
	LSR		74		Shift out of missile 0 bits, if
	LSR		74		they are present
	CMP	#12	201	12	Test for missile 2 bits
	BEQ	OTHER	240	17	If present branch to OTHER routine
Go back	DEY		136		Decrement Y to pick up missile 0
	LDA	(BASE),Y	177	205	Pick up M0
	AND	#3	41	3	Mask out M2, if present
	INY		200		Increment Y to store M0
	STA	(BASE),Y	145	205	Store M0
	DEY		136		Decrement Y
	DEX		202		Check if one segment moved

page 1 of 2

```
         BNE     LOOP         208     236    Branch back to move another
                                                      segment
         INC     VPOS         230     203    If done, increment the position
                                                      register
         JMP     HORIZ         76    54,6    Jump to horizontal move routine
OTHER    DEY                  136                Decrement Y to pick up M0
Go back  LDA     (BASE),Y     177     205            Load M0 segment
         ORA     #48            9      48            Add in missile 2
         INY                  200            Increment Y to store M0 and M2
         STA     (BASE),Y     145     205            Store M0 and M2
         DEY                  136             Decrement Y for next test and
                                                      pickup
         DEX                  202               Check off segment moved
         BNE     LOOP         208     219       Go back to move another
                                                      segment
         INC     VPOS         230     203      If done increment position
                                                      register
HORIZ    LDA     HPOS         165     204        Load horizontal position
         STA     HPOSMO       141   4,208      Store in M0 position register
         INC     HPOS         230     204     Increment horizontal position
         JMP     BEGIN         76     0,6        Jump back to beginning
```

Box 29. Assembly language listing moving missiles

Box 28 is the program listing and Box 29 is the assembly language listing for missile movement. In addition to illustrating AND, ORA, and LSR instructions, machine language speed is displayed. In order to allow the eye to perceive the missile as one box traveling diagonally across the screen it was necessary to begin the program with a delay loop. Even with this delay loop the missile appears to have a tail, much the same as a comet, as it speeds across the screen.

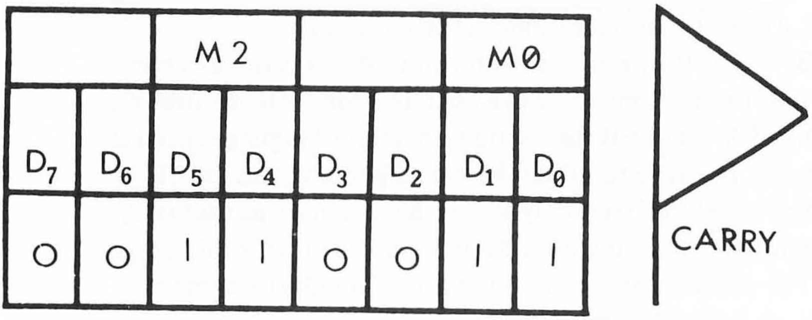
At the outset of the program the missiles are in memory as:

	M2		**M0**
00	11	00	11
00	11	00	11
00	11	00	11
00	11	00	11

Missile two will remain stationary while missile zero travels. What we must insure is that when we move M0, we don't wipe out M2. This involves shifting, testing, and masking. When making a comparision it is easier if there is only one element present. Therefore, prior to testing for M2 it makes sense to temporarily remove M0's bits from the accumulator. This is done with the LSR instruction.

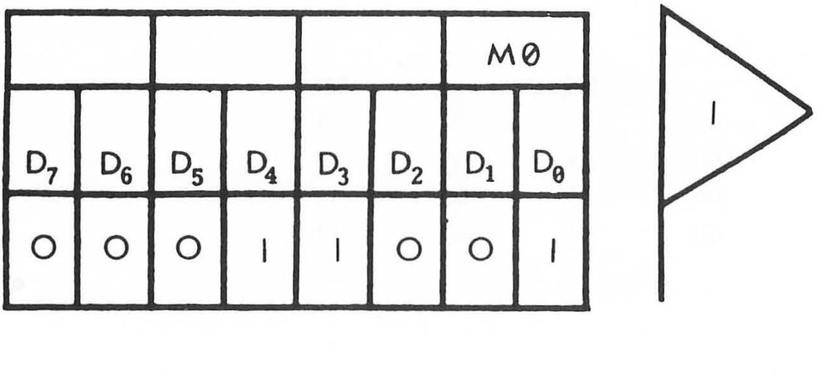
The LSR (Logical Shift Right) instruction has four addressing modes. In the present situation we use the accumulator addressing mode. In the LSR, a zero is shifted into bit D_7 and bit D_0 is shifted into the carry flag. At this point we are essentially not interested in the status of the carry flag, but rather in seeing that bits D_0 and D_1 go out of the accumulator. Pictorially the process can be represented as:

Accumulator before LSR



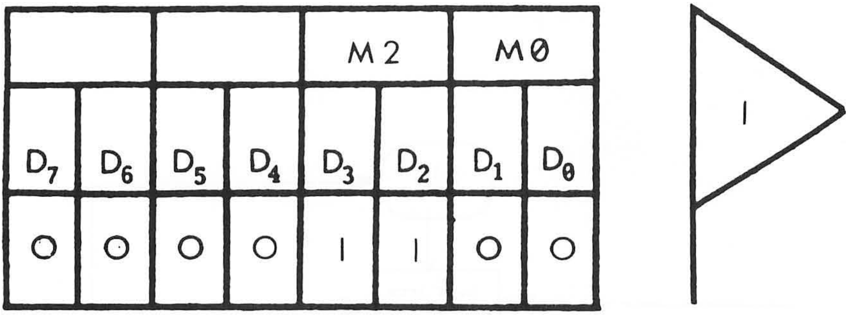
		M 2				M0	
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	1	1	0	0	1	1

After the first LSR



						M0	
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	1	1	0	0	1

After the second LSR



				M2		M0	
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	1	1	0	0

Note that the M0 bits are gone. The M2 bits have been shifted to the right so they now represent the decimal number 12. It is now a simple matter to test for their presence with a CMP #12.

There is something else we can learn from the LSR instruction. Each LSR will divide an even number by two. 48 divided by two twice is twelve. With odd numbers the presence of a one in the carry flag indicates the existence of a .5 remainder. What happens to a number if you shift the bits to the left? If you answered the number is multiplied by two, you are correct.

The comments in the machine language listing provide a detailed description of the missile movement. The following flowchart will help you to follow the logic.

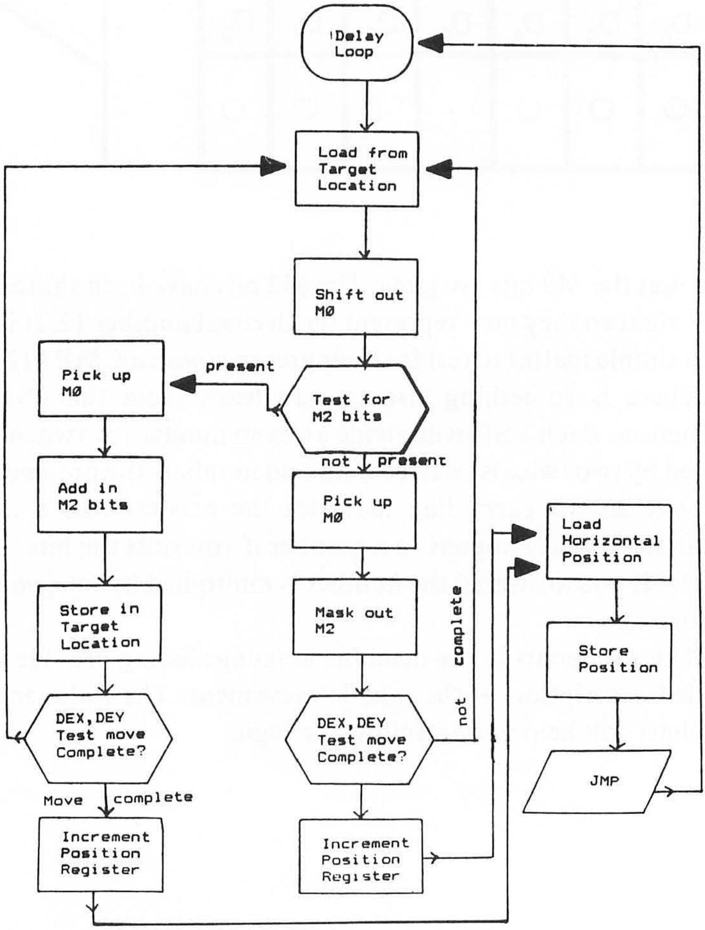


Figure 4.2. Missile Program Logic Flow

5

Sound

Introduction

Second generation computers such as the Atari Home Computer provide the programmer with the opportunity to use music and sounds within their programs as another means of communication. In an adventure game, music can set a mood, arouse emotions and complement the action. In a utility program, sounds can signal a keyboard entry error or warn that the disk is almost full. In addition to these common applications, the hardware capabilities of the Atari computers offer you the chance to try your hand at music synthesis via programs dedicated solely to sound generation. For maximum versatility and satisfaction, sound programs should be written in machine language since BASIC is too slow for generating complex sounds. Additionally, because of the nature of the 6502 processor, a music program written in BASIC cannot run simultaneously with the main program.

A complete analysis of music and sound synthesis is a field of study in and of itself and can be undertaken only by using advanced mathematics. Consequently, we will limit our discussion to the fundamentals of sound synthesis. This will be sufficient to suggest ways to use the sound generation hardware that we will describe. To aid you in exploring the sound capabilities of the Atari we have also included reference material and some utility programs.

A Bit of Theory

A sound or musical tone may be described by its intensity or loudness, its frequency or pitch, and its waveform or timbre. Sounds are created by devices such as tuning forks, TV speakers, or human vocal chords, that vibrate back and forth in a cyclic manner. These vibrations generate pressure changes in the surrounding air that are detected by the human ear as sound. What a human perceives as sound is a function of both the instrument generating the sound and the human ear - a piano sounds different from an oboe. Before we can understand complex tones, like those generated by a piano, it is necessary to understand simple tones or notes.

Sound transmission can be represented pictorially as waveforms with the simplest waveform being a sine wave. A waveform is usually drawn as a graph where the horizontal axis represents time and the vertical axis represents a parameter such as the displacement or pressure of the medium carrying the wave. The sine wave is referred to as a pure tone even though the aural perception of a pure tone may be impure.

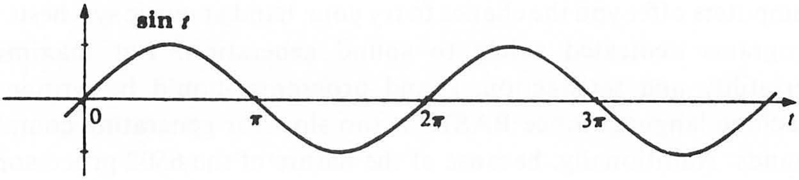


Figure 5-1. Sine wave

The fundamental parameters describing a pure tone are its frequency and amplitude. The *frequency* is the number of complete repetitions or cycles per second that the waveform displays. A related parameter is the *period* which is the time for one cycle. Mathematically, period and frequency are related by:

$$\text{frequency} = 1/\text{period}$$

Frequency (cycles per second) is measured in units called Hertz (Hz), after the 19th century physicist who discovered radio waves. Frequency is closely related to the perceived pitch. When frequency is increased, the perceived pitch also increases. However, pitch is a subjective parameter and the relationship between the two is not linear. For example, an increase from 100 Hz to 200 Hz results in the perception of a large increase in pitch upward, but an increase from 4000 Hz to 4200 Hz is a much less perceptible increase.

Likewise, *amplitude* is related to, but not equivalent to loudness. Here again the characteristics of the human ear enter into what is perceived. It has been found that the response of the ear is not proportional to the amplitude. Instead it is useful to use a logarithmic scale to measure loudness. Since the computer hardware puts limitations on the amplitude of the sounds that can be generated we will not have to worry about loudness scales. What will be important to us is the effect of loudness variations in sound generation.

A pure tone, such as would be produced by the waveform of figure 5-1, rapidly becomes dull listening. The essence of music is variation, variation in parameters such as amplitude, pitch and the rhythm with which notes are played. Too much variation, such as complete randomness, generates noise. Too little generates monotony.

One obvious way to introduce change into the sine waveform is to vary the amplitude. figure 5-2 shows a sine wave whose amplitude is modulated:

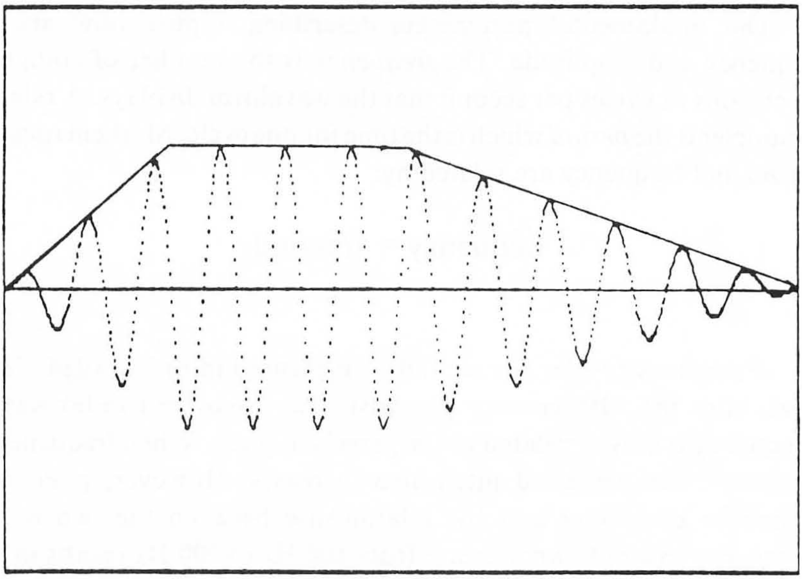


Figure 5-2. Sine wave with modulated amplitude

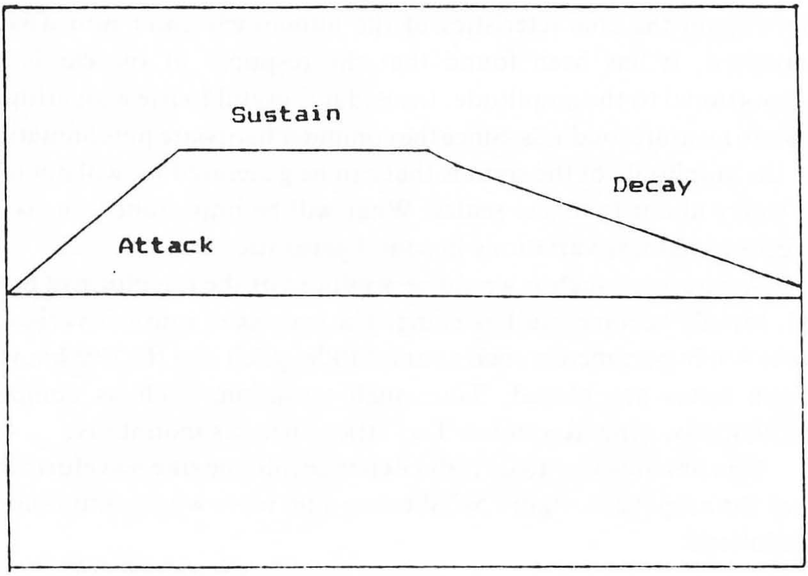


Figure 5-3. Sound envelope

If we connect the peaks with a line and throw away the sine wave the result is a picture of the sound envelope. This envelope is descriptive of a single note that sounds for a short period of time. The shape of the envelope is described by the risetime (attack), the sustain time, and the decay time.

Ignoring amplitude modulations of notes, the tones produced by a musical instrument are not pure tones characterized by a single frequency, but are composites of a fundamental frequency and overtones, or harmonics. Harmonics are waves whose frequency is an integral multiple of the fundamental frequency. What we call *timbre* is the result of different combinations of harmonics. One reason middle C on a piano and an oboe sound different is due to different combinations of harmonics.

Figure 5-4 illustrates a fundamental frequency, two overtones, and the result when these three waves are combined. Now as far as the ear is concerned the same result occurs if the final wave form of figure 5-4 is produced by a single instrument or if three separate instruments sound the pure tones simultaneously.

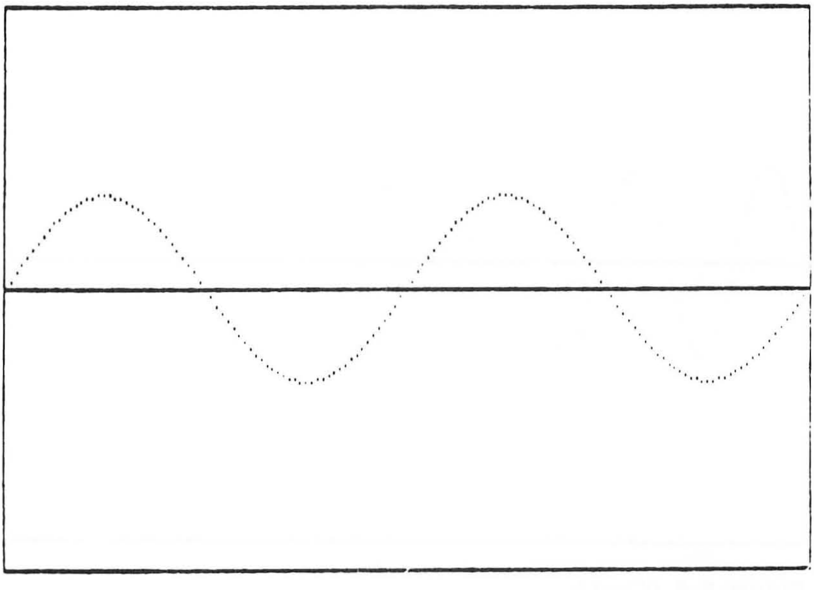
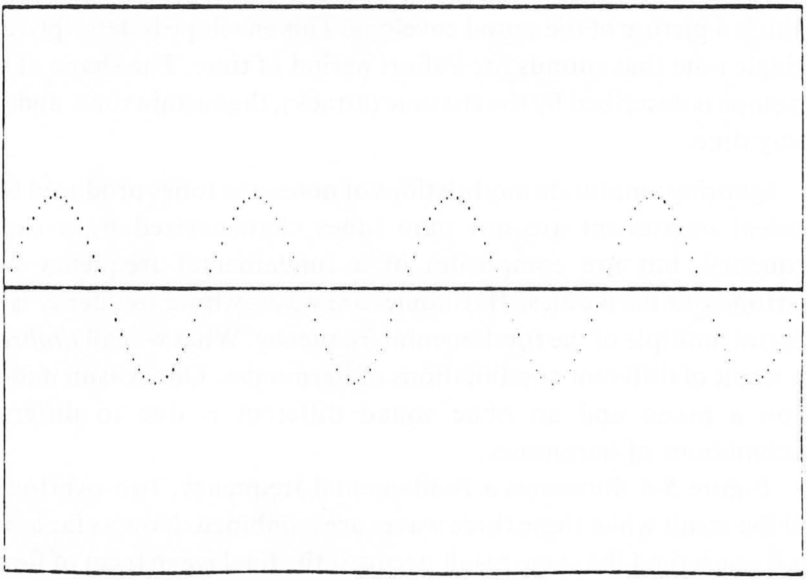


Figure 5-4. Fundamental frequency with two overtones



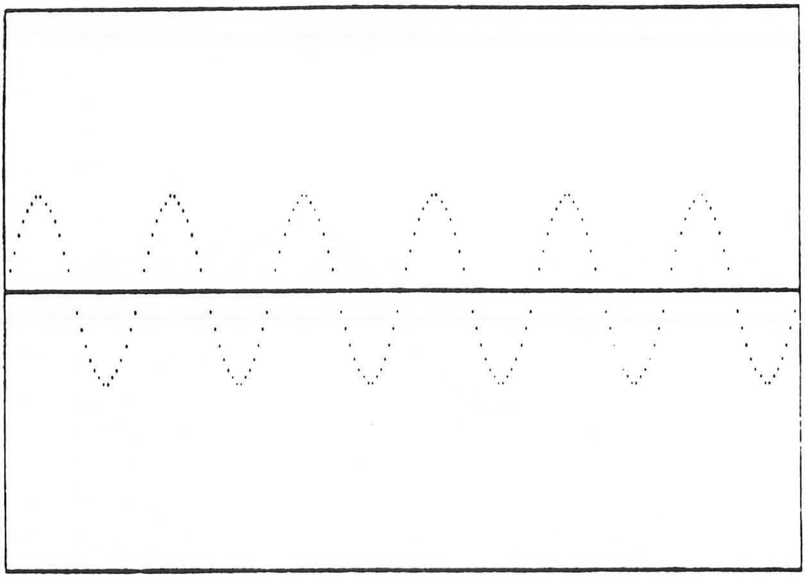


Figure 5-4. (Cont.)

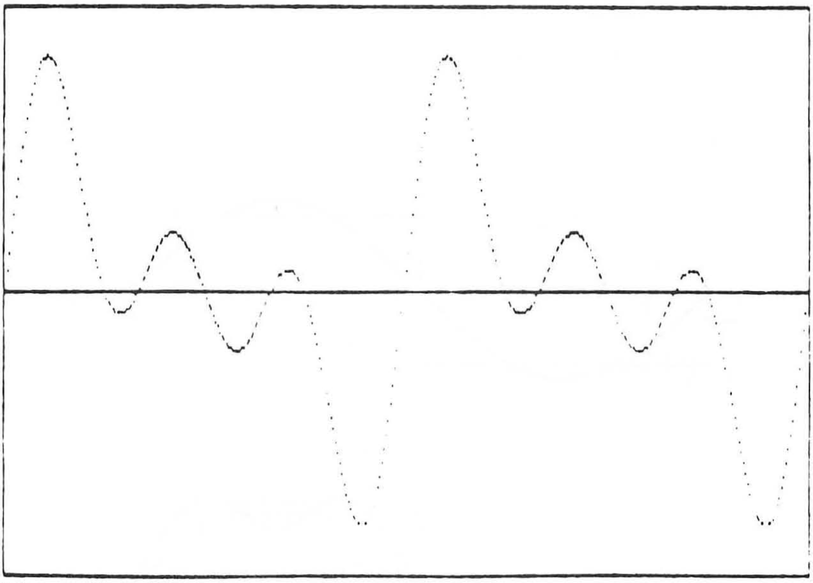


Figure 5-4. (Cont.)

The concept of adding sine waveforms of different frequency and amplitude together to form a new and different waveform is an extremely powerful technique in music synthesis. In fact, it's a powerful idea in mathematics as well. If one drops the restriction of figure 5-4, that the harmonics are integral multiples of the fundamental frequency, waves of almost any conceivable shape can be generated. Figures 5-5 and 5-6 show how sine waves can be added together to make a square wave and a triangle wave.

A triangle wave sounds very much like the sustained tone of an oboe, but the oboe's tone is much warmer and more interesting because of minute fluctations produced by the person playing the instrument. Such minute fluctuations are sometimes referred to as dynamic variation of the sound parameters. Of these, dynamic variation of frequency is perhaps the most basic. For example, in a simple one-voice melody if the frequency transition between notes is fairly long, the audible effect is that of a glide from note to note. Often with conventional instruments a small wavering of the frequency, called *vibrato* is added to the notes. Vibrato modifies the frequency six

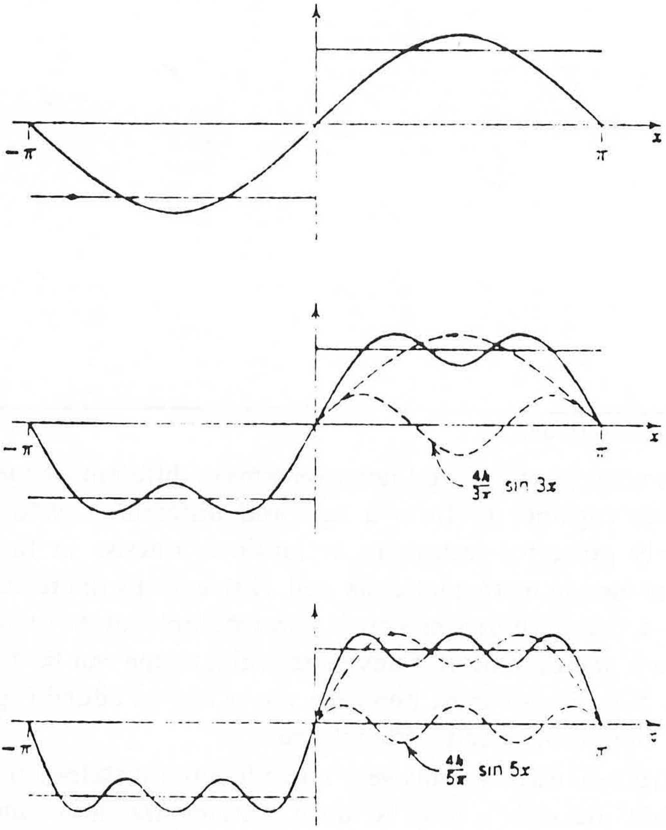


Figure 5-5. Square wave

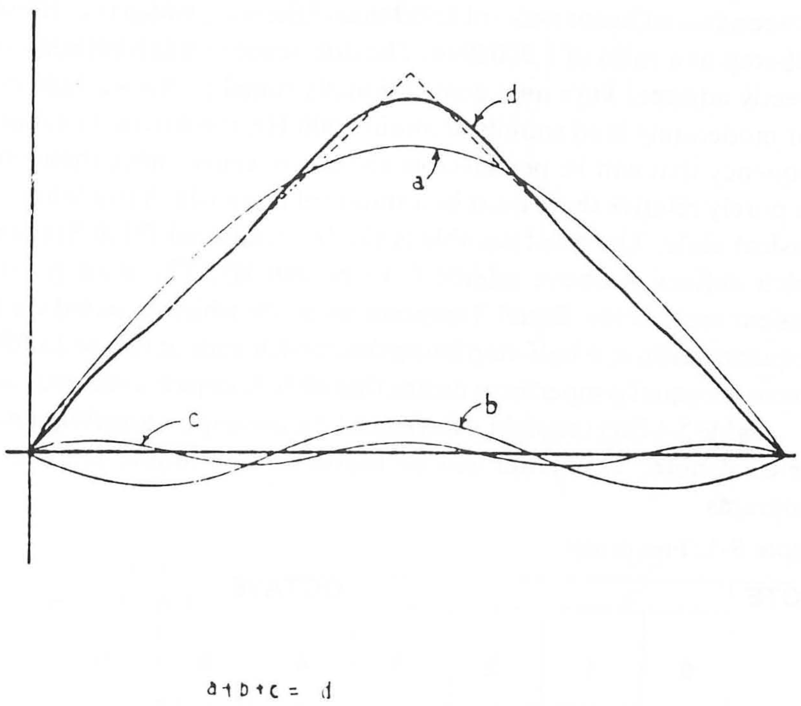


Figure 5-6. Triangle wave

to eight Hz with possibly a one percent amplitude variation. Actually, amplitude variation alone can be introduced and this is called *tremolo.* In conventional instruments both vibrato and tremolo will usually be present to some degree.

The discussion so far implies that there are several options available to the programmer. He can experiment with variations in amplitude or frequency and superimpose sounds of different frequencies. In addition to this, with the Atari Home Computer you can create sound effects by what might be called subtraction rather than superposition. Before going on to consideration of the hardware capabilities let's review the measurement of pitch.

The basic unit for measuring pitch is the *octave.* If tone 'A' is one octave higher than tone 'B', then its frequency is exactly twice as high and the sensation of pitch is twice as high. Other units of measurement are the *half-step,* which is 1/12 of an octave or a frequency ratio between two adjacent notes of 1.05946 and the *cent,* which is 1/100 of a half-step or a ratio of 1.0005946. The difference in pitch between two directly adjacent keys on a conventionally tuned piano is a half-step. For moderately loud sounds of about 1000 Hz, the smallest change in frequency that can be perceived is about five cents. Since these units are purely relative there must be a standard upon which to anchor any musical scale. The most notable is the International Pitch Standard which defines A above middle C to be 440 Hz. The most popular musical scale is the Equal Temperment Scale which is based on the frequency ratio of a half-step being the twelfth-root of two or 1.05946. The name equal temperment means that all half steps are the same size.

Table 5-1 lists the eight octaves and the corresponding frequencies for each note. This table will be useful in fine tuning your music programs.

Table 5-1. Frequency

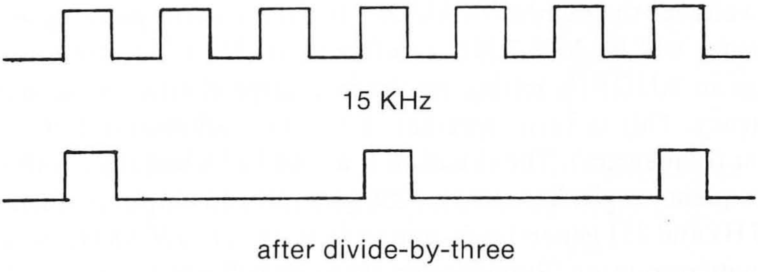
NOTE	OCTAVE							
	0	1	2	3	4	5	6	7
C	16.35	32.70	65.41	130.81	261.63*	523.25	1046.50	2093.00
C#	17.32	34.65	69.30	138.59	277.18	554.37	1108.75	2217.46
D	18.35	36.71	73.42	146.83	293.66	587.33	1174.66	2349.32
D#	19.45	38.89	77.78	155.56	311.13	622.25	1244.51	2489.02
E	20.60	41.20	82.41	164.81	329.63	659.26	1318.51	2637.02
F	21.83	43.65	87.31	174.61	349.23	698.46	1396.91	2793.83
F#	23.12	46.25	92.50	185.00	369.99	739.99	1479.98	2959.96
G	24.50	49.00	98.00	196.00	392.00	783.99	1567.98	3135.96
G#	25.96	51.91	103.83	207.65	415.30	830.64	1661.22	3322.44
A	27.50	55.00	110.00	220.00	440.00#	880.00	1760.00	3520.00
A#	29.41	58.27	116.54	233.08	466.16	932.33	1864.66	3729.31
B	30.87	61.74	123.47	246.94	493.88	987.77	1975.53	3951.07

* = middle C
# = concert A

Sound Hardware

The heart of sound generation in the Atari Home Computer is four programmable sound channels that can operate independently or in pairs. Associated with each sound channel is a frequency register, that is used to determine which note is played, and an audio control register. This is all handled by POKEY. In addition to sound generation, POKEY is an input/output chip that controls serial I/O and keyboard input. POKEY allows a sufficient number of control, frequency, and volume options so that the programmer can work with these parameters to synthesize music.

Frequency is the basis of music so we'll look at the frequency registers first. The frequency registers AUDF1 through AUDF4 are at memory locations 53760, 53762, 53764, and 53766 respectively. Numbers stored in these registers provide the "N" in divide by N counters that reduce one of the three basic clock frequencies to a desired sound frequency. The three basic clock frequencies are 15 KHz (kilohertz), 64 KHz and the system clock 1.79 MHz (millionhertz). Suppose you are working with the 15 KHz clock. This clock generates a signal consisting of 15,000 square pulses per second:



after divide-by-three

A simple divide by three operation allows every third pulse through and thus reduces the frequency. If the resulting signal is fed to the TV speaker, the speaker will vibrate in response to the pulses.

There are two formulas that are used to calculate the output frequency. If the clock frequency chosen is 64 KHz or 15 KHz, the formula is:

$$\text{frequency out} = \text{clock frequency}/2(\text{AUDF}+1)$$

where AUDF is the number in the frequency register.

If the system clock, 1.79 MHz is used then the formula is modified to:

$$\text{frequency out} = \text{clock frequency}/2(\text{AUDF}+\text{M})$$

Where M=4 if the frequency registers operate singly.
Where M=8 if the two sound channels are paired.

The option just mentioned, pairing sound channels, is provided to give the user the opportunity to match sound frequencies more closely than may be possible with single channels. The following numerical examples will make this concept clearer. Suppose you are using single channel sound. Then the numbers in the frequency registers are eight bit numbers that can have decimal values of 0 to 255. Using 10 as the number in AUDF1 with a clock frequency of 64000 Hz, according to the formula, the output frequency is

$$64000/2(10+1) = 2909.1 \text{ Hz}$$

If you change the number in AUDF1 to 11, the corresponding output frequency will be 2666.7 Hz, a difference of 242.1 Hz. Thus a small change in AUDF1's setting results in a large change in the output frequency. This in turn represents a loss in resolution (selectivity of output frequencies). The situation is not so bad when the numbers in the frequency registers are large: 250 generates an output frequency of 127.5 Hz and 251 generates an output frequency of 126.98 Hz, which is adequate resolution. Single sound channels will work satisfactorily in many cases. For cases in which they are not adequate, pairing two sound channels also pairs the frequency registers. Paired registers act as 16 bit numbers thereby giving N values of 0 to 65535. When sound channels are paired the clock frequency used is 1.79 MHz.

In the above discussion the values 15 KHz, 64 KHz and 1.79 MHz are all approximate. If you need the exact values, use 15.6999 KHz, 63.9210 KHz and 1.78979 MHz respectively.

Associated with each frequency register is a control register AUDC1 through AUDC4. These are at the memory location following the frequency register that they control - 53761, 53763, 53765, and 53767. There is also one general control register AUDCTL at memory location 53768 (see table 5-2).

The bit configuration of the control registers is:

D_7 D_6 D_5 Distortion control bits for sound effects

D_4 Sets a volume only mode that disables frequency registers

D_3 D_2 D_1 D_0 Volume control

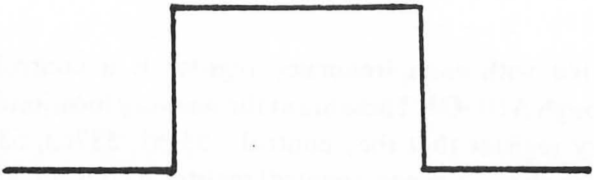
For a pure tone the upper four bits of a control register must be set as:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	X	1	0				

X means it doesn't matter.

The bottom four bits (D_3 - D_0) determine the volume with 0000 giving no sound, 1000 provides half volume and 1111 maximum volume. In any case, the sum of all volumes in all sound channels should not exceed decimal 32 or the sound quality will suffer.

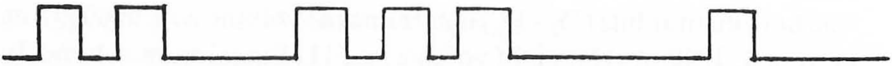
The volume only option, which is chosen when bit four of the AUDC registers is set, gives the programmer an opportunity to experiment with waveform synthesis. When this bit is set, the frequency registers are disconnected from the system. Then bits D_0 - D_3 determine the position of the TV speaker. A TV speaker consists of a paper cone that moves in and out in response to changing voltage values sent to it by the computer. A single pulse, consisting of a rising voltage followed by a falling voltage



would cause the speaker to move out with the rising portion and return to its rest position with the falling portion of the pulse.

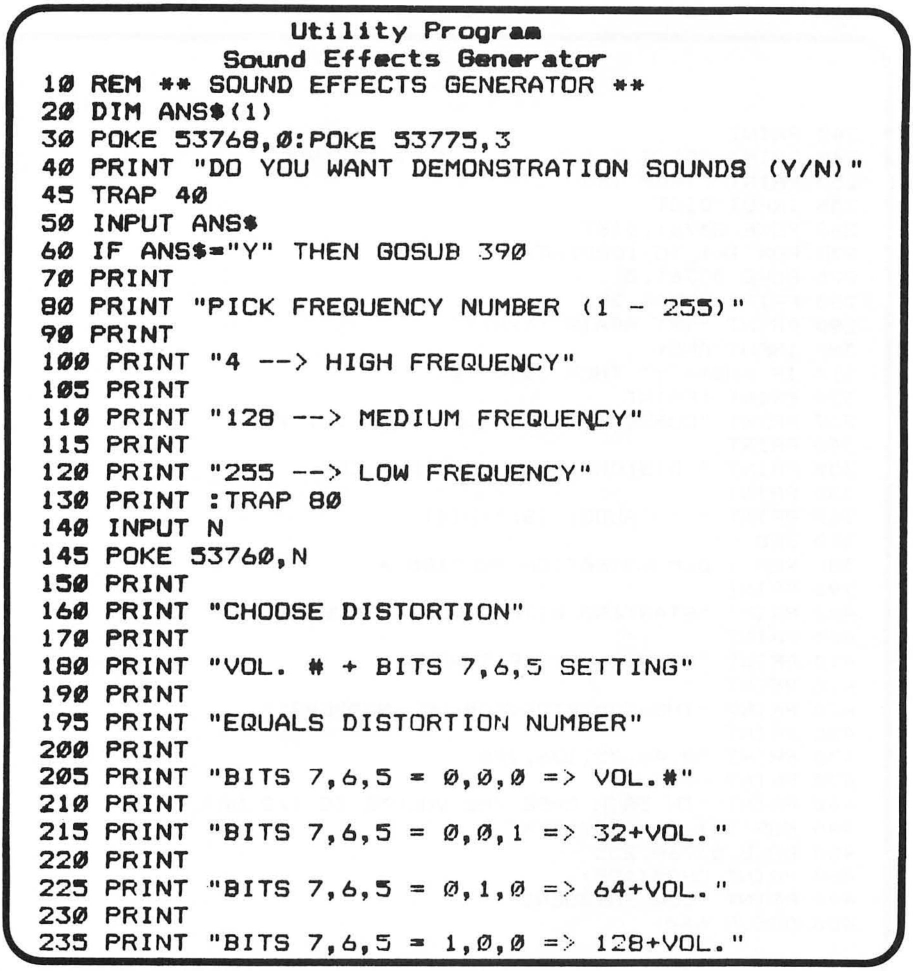
Using the volume only mode, the speakers moves to one of sixteen positions depending on the value stored in AUDC. In this case, however, the speaker does not automatically return to the zero position as with the pulse above, but remains set in a position until the program modifies it. In principle then, synthesizing a waveform becomes a matter of writing a program to move the speaker so that its successive positions match the amplitude of the waveform. The major limitation is that there is only sixteen position settings.

The upper three bits (D_7 - D_5) of the audio control registers are used to create sound effects where a pure tone is not wanted. These bits control polynomial counters, also called poly-counters, that are used to remove pulses from a normal pure tone's train of pulses. These pulses are removed in a semi-random manner and the bit pattern that results will repeat after a span of time. When bits are removed randomly, giving a pulse train such as,



the resulting sound can imitate anything from a lawn mower to a rocket blast off!

The repetition rate depends on the number of bits in the poly-counter. A *poly-counter* is a modified shift register whose internal operation needn't concern us. There are three poly-counters, two that are small, being four and five bits long, and a large one - 17 bits long. Optionally using AUDCTL, the 17 bit poly-counter can be reduced to a 9 bit length. The poly-counters can operate singly or in pairs. Table 5-2 gives the bit settings for choosing the allowable combinations. Keep in mind that the starting point is a train of pulses at a constant frequency which has been determined by the clock and the frequency register. Two factors enter into the creation of the sound heard: the particular combination of poly-counters and the frequency being used. For this reason it will take some experimentation to create a desired effect. Box 30 is a sound effects utility program to help you experiment.



Utility Program

Sound Effects Generator

```
10 REM ** SOUND EFFECTS GENERATOR **
20 DIM ANS$(1)
30 POKE 53768,0:POKE 53775,3
40 PRINT "DO YOU WANT DEMONSTRATION SOUNDS (Y/N)"
45 TRAP 40
50 INPUT ANS$
60 IF ANS$="Y" THEN GOSUB 390
70 PRINT
80 PRINT "PICK FREQUENCY NUMBER (1 - 255)"
90 PRINT
100 PRINT "4 --> HIGH FREQUENCY"
105 PRINT
110 PRINT "128 --> MEDIUM FREQUENCY"
115 PRINT
120 PRINT "255 --> LOW FREQUENCY"
130 PRINT :TRAP 80
140 INPUT N
145 POKE 53760,N
150 PRINT
160 PRINT "CHOOSE DISTORTION"
170 PRINT
180 PRINT "VOL. # + BITS 7,6,5 SETTING"
190 PRINT
195 PRINT "EQUALS DISTORTION NUMBER"
200 PRINT
205 PRINT "BITS 7,6,5 = 0,0,0 => VOL.#"
210 PRINT
215 PRINT "BITS 7,6,5 = 0,0,1 => 32+VOL."
220 PRINT
225 PRINT "BITS 7,6,5 = 0,1,0 => 64+VOL."
230 PRINT
235 PRINT "BITS 7,6,5 = 1,0,0 => 128+VOL."
```

Box 30. Sound effects generator

```
240 PRINT
245 PRINT "BITS 7,6,5 = 1,1,0 => 192+VOL."
250 PRINT :TRAP 160
255 INPUT DIST
260 POKE 53761,DIST
270 FOR I=1 TO 1000:NEXT I
275 POKE 53761,0
280 PRINT :TRAP 290
290 PRINT "TRY AGAIN (Y/N)?"
300 INPUT ANS$
310 IF ANS$="Y" THEN GOTO 70
320 PRINT :PRINT
330 PRINT "CURRENT NUMBER IN AUDF1 IS:";N
340 PRINT
350 PRINT " DISTORTION-VOLUME NUMBER"
355 PRINT
360 PRINT " IN AUDC1 IS:";DIST
370 END
380 REM * DEMONSTRATION SECTION *
390 PRINT
400 PRINT "STARTING WITH A LOW FREQUENCY"
405 PRINT
410 PRINT "WE WILL CYCLE THROUGH"
415 PRINT
420 PRINT "THESE DISTORTION-VOL NUMBERS:"
425 PRINT
430 PRINT "8,40,72,136,200"
435 PRINT
440 PRINT "IN EACH CASE THE VOLUME IS 1/2 MAX."
445 FOR I=1 TO 1000:NEXT I
450 POKE 53760,255
460 PRINT CHR$(125)
470 PRINT "LOW FREQUENCY"
480 GOSUB 640
```

Box 30. Cont.

```
490 PRINT :PRINT "CONTINUE (Y/N)?"
500 INPUT ANS$:TRAP 500
510 IF ANS$="N" THEN GOTO 80
520 PRINT CHR$(125):RESTORE 650
530 POKE 53760,128
540 PRINT "MEDIUM FREQUENCY"
550 GOSUB 640
560 PRINT :PRINT "CONTINUE (Y/N)?"
570 INPUT ANS$:TRAP 570
580 IF ANS$="N" THEN GOTO 80
590 PRINT CHR$(125):RESTORE 650
600 POKE 53760,4
610 PRINT "HIGH FREQUENCY"
620 GOSUB 640
625 TRAP 40000
630 RETURN
640 READ X:PRINT :PRINT "DISTORTION+VOL =";X
650 DATA 8,40,72,136,200,-1
660 IF X=-1 THEN POKE 53761,0:RETURN
670 POKE 53761,X
680 FOR I=1 TO 500:NEXT I
690 GOTO 640
```

BOX 30. (Cont.)

The frequency registers select the tones. The control registers AUDC give volume and sound effects option for each sound channel. There is one register that exerts overall control. This is AUDCTL at memory location 53768. The functions of the bits in this register are listed in table 5-2 which also summarizes the functions of all other sound hardware registers.

The high pass filters mentioned under AUDCTL in table 5-2 deserve some explanation. A *high pass filter* allows frequencies higher than some predetermined limit to pass through. Here the limit is determined by the frequency in another channel. When there is a filter in channel 2, channel 4 sets the frequency limit. When there is a filter in channel 1, channel 3 sets the limit.The filters can be useful for creating sound effects.

Table 5-2. Summary of sound registers

A. Frequency Registers at (AUDF)

53760, 53762, 53764, 53766

B. Audio Control Registers at (AUDC)

53761, 53763, 53765, 53767

Bit Functions

	Bit Functions
D_3 D_2 D_1 D_0	Set Volume in frequency mode Position (volume) in volume only mode
D_4	1 sets volume only mode

D_5	D_6	D_7	
0	0	0	Remove pulses using 5 and 17 bit polys, divide by 2
0	X	0	Remove pulses using 5 bit polys, divide by 2
0	1	0	Remove pulses using 4 and 5 bit polys, divide by 2
1	0	0	Remove pulses using 17 bit poly, divide by 2
1	X	1	Pure tone
1	1	1	Remove pulses using 4 bit poly, divide by 2

C. Audio Control (AUDCTL) at 53768

Bit Functions

D_0	0 gives 64 KHz clock; 1 gives 15Hz clock
D_1	when set, uses high-pass filter in channel 2
D_2	when set, uses high-pass filter in channel 1
D_3	when set, joins channels 3 and 4 (16 bit resolution)
D_4	when set, joins channels 2 and 1 (16 bit resolution)
D_5	when set, clocks channel 3 with 1.79 MHz
D_6	when set, clocks channel 1 with 1.79 MHz
D_7	when set, changes 17 bit poly to 9 bit poly

store a 0 in AUDCTL to initialize POKEY for sound

D. Serial Port Control (SKCTL) at 53775

Store a 3 here to initialize POKEY for sound

Program Examples

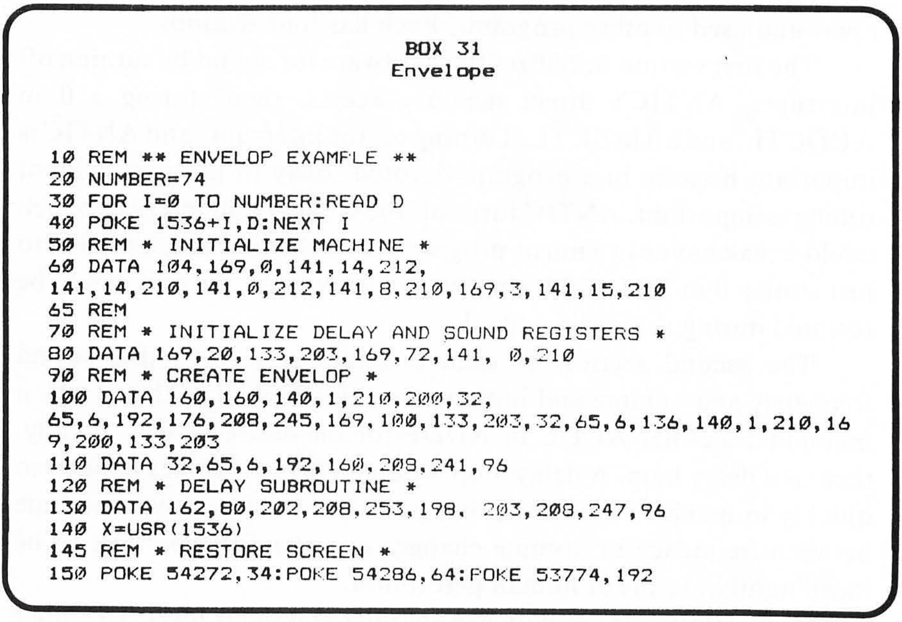
The next few programs (Boxes 31 - 33A) illustrate the theoretical concepts discussed earlier: Envelope, Tremolo, and Vibrato. The programs are all structured similarly and are intended to be taken apart and used in other programs. Each has four sections.

The first section initializes the hardware for sound by turning off interrupts, ANTIC's direct memory access, then storing a 0 in AUDCTL, and a 3 in SKTL. Turning off the interrupts and ANTIC is important because in a program devoted solely to music consistent timing is important. ANTIC turns off the CPU at odd intervals which could wreak havoc in a music program. Note that it is not sufficient to just store a 0 in DMACTL since it is shadowed at 559 and would be restored during the vertical blank.

The second section of each program sets the initial sound frequency and volume and initializes a delay loop. The third section manipulates either AUDC or AUDF for the desired effect. Finally, there is a delay loop. A delay loop is needed because things happen so quickly in machine language, that you have to slow down the time between frequency or volume changes in order for the effect to be meaningful in terms of human perception.

Each delay loop actually has an inner and outer loop. The inner loop takes some set amount of time say, for example, .225 milliseconds to execute and the outer loop determines how many of these units of time are used in the delay. The amount of time taken for delay loops can be estimated by counting the machine cycles needed for execution of each instruction. Using the approximate value of the CPU clock frequency, 1.79 MHz, the time for one machine cycle is:

.000000559 seconds



BOX 31
Envelope

```
10 REM ** ENVELOP EXAMFLE **
20 NUMBER=74
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,
141,14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM
70 REM * INITIALIZE DELAY AND SOUND REGISTERS *
80 DATA 169,20,133,203,169,72,141, 0,210
90 REM * CREATE ENVELOP *
100 DATA 160,160,140,1,210,200,32,
65,6,192,176,208,245,169.100,133,203,32,65,6,136,140,1,210,16
9,200,133,203
110 DATA 32,65,6,192,160.209.241,96
120 REM * DELAY SUBROUTINE *
130 DATA 162,80,202,208.253.198. 203,208,247,96
140 X=USR(1536)
145 REM * RESTORE SCREEN *
150 POKE 54272.34:POKE 54286.64:FOKE 53774,192
```

Box 31. Envelope

BOX 31A

Assembly Language Listing
for
Attack, Sustain, Delay

Section 1: *Initialize the machine*

```
PLA                 104           Remove parameter count
LDA #0              169,0
STA NMIEN           141,14,212    Turn off
STA IRQEN           141,14,210    interrupts
STA DMACTL          141,0,212     Turn off ANTIC
STA AUDCTL          141,8,210
LDA #3              169,3         Initialize POKEY
STA SKCTL           141,15,210
```

Section 2: *Initialize Delay and Sound*

```
LDA #20             169,20        Initialize
STA COUNT           133,203       delay loop
LDA #72             169,72        Intialize
STA AUDF1           141,0,212     frequency
```

BOX 31A. Assembly language listing for Attack, Sustain, and Decay

```
Section 3: Create envelop
          LDY #160        160,160     Start with
LOOPA STY AUDC1          140,1,210    zero volume
          INY               200       Increment for next volume
          JSR DELAY        32,65,6    Delay before changing volume
          CPY #176        192,176     Is volume its maximum?
          BNE LOOPA       208,245     If not continue
          LDA #100        169,100     Create delay for the
          STA COUNT       133,203     sustain portion of envelop
          JSR DELAY        32,65,6    Jump to delay
LOOPB DEY                   136       Start decay portion of
          STY AUDC1      140,1,210    the envelop
          LDA #200        169,200     create delay for the
          STA COUNT       133,203     decay portion
          JSR DELAY        32,65,6    Jump to delay
          CPY #160        192,160     Is volume zero?
          BNE LOOPB       208,241     If not, continue
          RTS                96       Return to basic
Section 4: Delay Subroutine
DELAY LDX #80             162,80      Inner loop counter
LOOPC DEX                   202       Inner
          BNE LOOPC       208,253     Loop
          DEC 203         198,203     Decrement delay counter
          BNE DELAY       208,247     If counter not 0, continue
          RTS                96       Return from delay
```

Box 31A. (Cont.)

```
                         BOX 32
                         Tremolo
10 REM ** TREMOLO EXAMPLE **
20 NUMBER=75
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,
141,14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM
70 REM * INITIALIZE DELAY AND SOUND REGISTERS *
80 DATA 169,2,133,203,160,72,
140,0,210,160,166,140,1,210,32,62,6
90 REM * CREATE TREMOLO *
100 DATA 200,140,1,210,32,62,6,
192,169,208,245,136,140,1,210,32,62,6,192,166,208,245,76,37,6
120 REM * DELAY SUBROUTINE *
130 DATA 162,60,202,208,253,198, 203,208,247,169,2,133,203,96
140 X=USR(1536)
```

Box 32. Tremolo

BOX 32A

Assembly Language Listing for Tremolo

Section 1: *Initialize the machine*

```
PLA                 104        Remove parameter count
LDA #0              169,0
STA NMIEN       141,14,212     Turn off
STA IRQEN       141,14,210     interrupts
STA DMACTL       141,0,212
STA AUDCTL       141,8,210
LDA #3              169,3      Initialize POKEY
STA SKCTL       141,15,210
```

Section 2: *Initialize Delay and Sound*

```
LDA #2              169,2      Initialize
STA 203           133,203      delay
LDY #72            160,72      Initialize
STY AUDF1        141,0,210     frequency
LDY #166          160,166      Initialize
STY AUDC1        140,1,210     volume
JSR DELAY          32,62,6     Jump to delay
```

BOX 32A. Assembly language listing for Tremolo

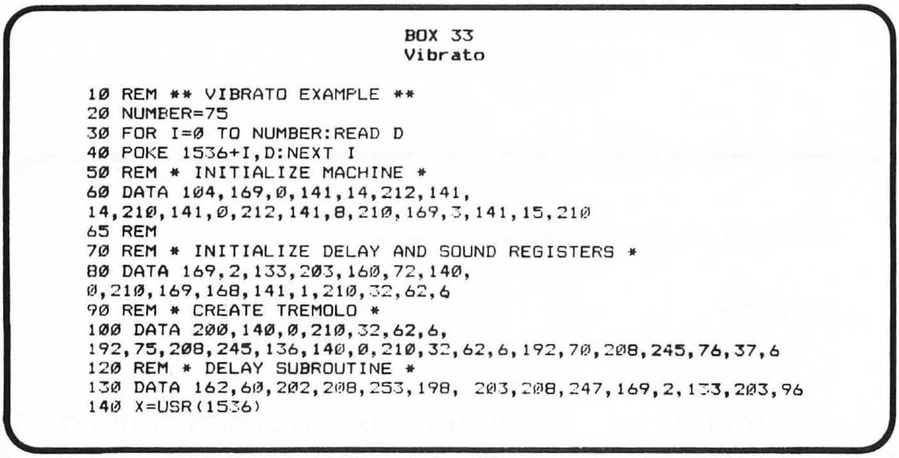
Section 3: *Volume increase and decay*

```
INCR INY              200        Increment
   STY AUDC1          140        volume
   JSR DELAY        32,62,6      Jump to delay
   CPY #169         192,169      Is volume increase done?
   BNE INCR         208,245      If not, continue
DECR DEY              136        Decrease
   STY AUDC1       140,1,210     volume
   JSR DELAY        32,62,6      Jump to delay
   CPY #166         192,166      Is volume done?
   BNE DECR         208,245      If not, continue
   JMP INCR         76,37,6      Jump to increase
```

Section 4: *Delay Subroutine*

```
DELAY LDX #80       162,80       Inner loop counter
LOOPC DEX             202        Inner
   BNE LOOPC        208,253      loop
   DEC 203          198,203      Decrement delay counter
   BNE DELAY        208,247      If counter not 0, continue
   RTS                96         Return from delay
```

BOX 32A. (Cont.)



BOX 33
Vibrato

```
10 REM ** VIBRATO EXAMPLE **
20 NUMBER=75
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,141,
14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM
70 REM * INITIALIZE DELAY AND SOUND REGISTERS *
80 DATA 169,2,133,203,160,72,140,
0,210,169,168,141,1,210,32,62,6
90 REM * CREATE TREMOLO *
100 DATA 200,140,0,210,32,62,6,
192,75,208,245,136,140,0,210,32,62,6,192,70,208,245,76,37,6
120 REM * DELAY SUBROUTINE *
130 DATA 162,60,202,208,253,198, 203,208,247,169,2,133,203,96
140 X=USR(1536)
```

Box 33. Vibrato

BOX 33A

Assembly Language Listing for Vibrato

Section 1: *Initialize the machine*

```
PLA            104            Remove parameter count
LDA #0         169,0
STA NMIEN      141,14,212     Turn off
STA IRQEN      141,14,210     interrupts
STA DMACTL     141,0,212      Turn off ANTIC
STA AUDCTL     141,8,210
LDA #3         169,3          Initialize POKEY
STA SKCTL      141,15,210
```

Section 2: *Initialize Delay and Sound*

```
LDA #2         169,2          Initialize
STA 203        133,203        delay loop
LDY #72        160,72         Initialize
STY AUDF1      140,0,210      frequency
LDA #168       169,168        Set volume
STA AUDC1      141,1,210      at ½ maximum
JSR DELAY      32,62,6        Jump to delay
```

BOX 33A. Assembly language listing for Vibrato

Section 3: *Create vibrato*

```
INCR INY          200            Increase
     STY AUDC1    140,0,210      frequency
     JSR DELAY    32,62,6        Jump to delay
     CPY #75      192,75         Is frequency increase done?
     BNE INCR     208,245        If not, continue
DECR DEY          136            Decrease
     STY AUDC1    140,1 210      frequency
     JSR DELAY    32,62,6        Jump to delay
     CPY #70      192,70         Is frequency done?
     BNE DECR     208,245        If not, continue
     JMP INCR     76,37,6        jump to increase
```

Section 4: *Delay Subroutine*

```
DELAY LDX #80     162,80         Inner loop counter
LOOPC DEX         202            Inner
     BNE LOOPC    208,253        loop
     DEC 203      198,203        Decrement delay counter
     BNE DELAY    208,247        If counter not 0, continue
     RTS          96             Return from delay
```

BOX 33A. (Cont.)

The number of cycles taken for each instruction depends on the instruction and its addressing mode. Some typical values are listed in table 5-3.

Table 5-3. Typical cycle times

Instruction	Addressing Mode				
	Immediate	Page 0	Absolute	Implied	Relative
LDA, LDX, LDY	2	2	4		
DEX DEY INX INY				2	
DEC		5	6		
BNE BEQ					3 (same page) 4 (different pg)
JSR RTS				6	

A complete listing of the number of cycles for all instructions can be found in more advanced books on 6502 programming. For a simple loop, the time calculation goes like this:

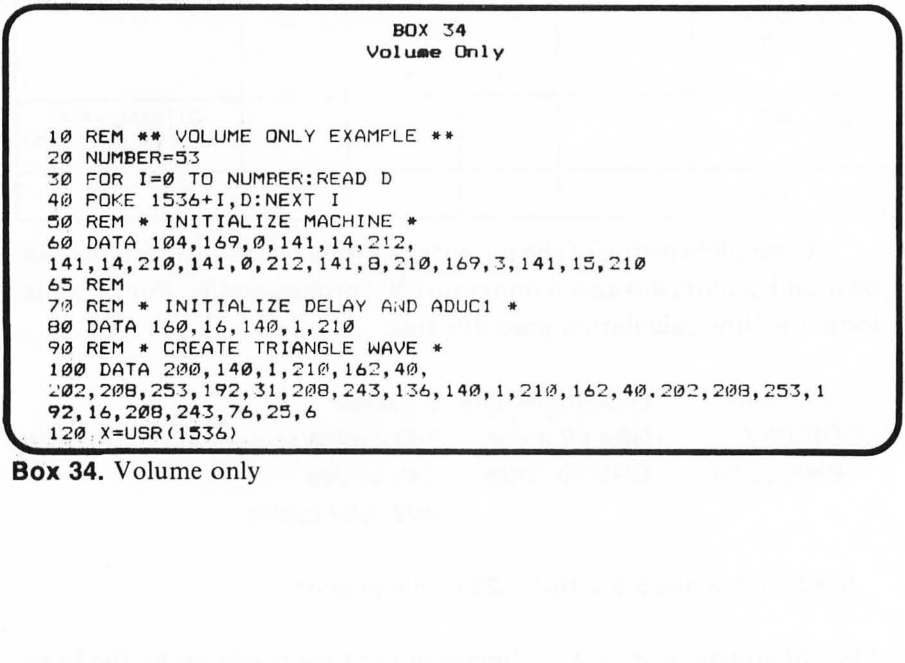
```
   LDX #80       LDX immediate  2 cycles
LOOP DEX         DEX 80 times   160 cycles
   BNE LOOP      BNE 80 times   240 cycles
                                402 total cycles
```

402 cycles times 5.6 x 10^{-7} =.225 milliseconds.

This, of course, is only an estimate of the time taken up by the loop because the jump and return instructions take CPU time as will any other instruction involving the outer loop.

Each of the machine language routines in Boxes 31, 32, and 33 is fully documented in an accompanying assembly language listing. So that you don't get lost in the details, keep in mind the objective of each program. In the envelop and tremolo programs, a note frequency is chosen and the routine manipulates the volume of the note by changing the values in the lower four bits of AUDC1. In the vibrato program it is the frequency value in AUDF1 that is changed.

The next two program examples illustrate the simplest type of volume only sound. Both programs generate triangle shaped waveforms by incrementing the volume bits of AUDC1 from 0000 to 1111 and then decrementing back to 0000. In these programs, the delay loop is placed within the straight line flow of the program, rather than in a subroutine, because the timing requirements of the program are more severe.



BOX 34
Volume Only

```
10 REM ** VOLUME ONLY EXAMPLE **
20 NUMBER=53
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,
141,14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM
70 REM * INITIALIZE DELAY AND ADUC1 *
80 DATA 160,16,140,1,210
90 REM * CREATE TRIANGLE WAVE *
100 DATA 200,140,1,210,162,40,
202,208,253,192,31,208,243,136,140,1,210,162,40.202,208,253,1
92,16,208,243,76,25,6
120 X=USR(1536)
```

Box 34. Volume only

BOX 34A

Assembly Language Listing for Volume Only - Triangle Wave

Section 1: *Initialize the machine*

```
PLA             104             Remove parameter count
LDA #0          169,0
STA NMIEN       141,14,212      Turn off
STA IRQEN       141,14,210      interrupts
STA DMACTL      141,0,212       Turn off ANTIC
STA AUDCTL      141,8,210
LDA #3          169,3           Initialize POKEY
STA SKCTL       141,15,210
```

Section 2: *Initialize Volume*

```
LDY VOLONLY     160,16          Set Bit D4 of AUDC1
STA AUDC1       140,1,210       for volume only
```

BOX 34A. Assembly language listing for Volume Only-Triangle Wave

Section 3: *Generate triangle waveform*

```
INCR INY             200           Increase
   STY AUDC1         140,1,210     volume
   LDX,40            162,40        Delay loop. Value in X
LOOP DEX             202           register determines
   BNE LOOP          208,253       frequency of the sound
   CPY MAXVOL        192,31        Is up-ramp complete?
   BNE INCR          208,243       If not, continue
DECR DEY             136           If volume is max,
   STY AUDC1         140,1,210     start down-ramp
   LDX,40            162,40        Delay loop. Value will
LOOP DEX             202           determine frequency of
   BNE LOOP          208,253       the sound
   CPY #16           192,16        Is down-ramp complete?
   BNE DECR          208,243       If not, continue
   JMP INCR          790,25,6      If yes, go to up ramp
```

BOX 34A. (Cont.)

```
10 REM ** VOLUME ONLY EXAMPLE WITH VARYING FREQUENCY **
20 NUMBER=79
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,141,
14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM
70 REM * INITIALIZE DELAY AND ADUC1 *
80 DATA 160,16,140,1,210
90 REM * CREATE TRIANGLE WAVE *
100 DATA 200,140,1,210,162,40,202,
208,253,192,31,208,243,136,140,1,210,162,40,202,208,253,192,1
6,208,243
110 REM
120 DATA 200,140,1,210,162,32,202,
208,253,192,31,208,243,136,141,1,210,162,32,202,208,253,192,1
6,208,243,76,25,6
140 X=USR(1536)
```

Box 35. Volume only with varying frequency

BOX 35A

Assembly Language Listing for Triangle Wave - Varying Frequency

Section 1: *Initialize the machine*

```
PLA           104          Remove parameter count
LDA #0        169,0
STA NMIEN     141,14,212   Turn off
STA IRQEN     141,14,210   interrupts
STA DMACTL    141,0,212    Turn off ANTIC
STA AUDCTL    141,8,210
LDA #3        169,3        Initialize POKEY
STA SKCTL     141,15,210
```

Section 2: *Initialize Volume*

```
LDY VOLONLY   160,16       Set Bit D4 of AUDC1
STA AUDC1     140,1,210    for volume only
```

BOX 35A. Assembly language listing for Triangle Wave-Varying Frequency

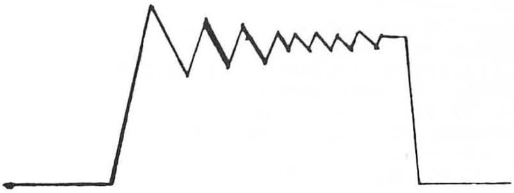
```
Section 3: Generate triangle waveform
INCR1          INY                 200
               STY AUDC1           140,1,210
               LDX,40              162,40
LOOP1          DEX                 202
               BNE LOOP1           208,253
               CPY #31             192,31
               BNE INCR1           208,243
DECR1          DEY                 136
               STY AUDC1           140,1,210
               LDX,40              162,40
LOOP2          DEX                 202
               BNE LOOP2           208,253
               CPY #16             192,16
               BNE DECR            208,243
INCR2          INY                 200
               STY AUDC1           140,1,210
               LDX #30             162,30
LOOP3          DEX                 202
               BNE LOOP3           208,253
               CPY #31             192,31
               BNE INCR2           208,243
DECR2          DEY                 136
               STY AUDC1           141,1,210
               LDX #30             162,30
               DEX                 202
               BNE LOOP            208,253
               CPY #16             192,16
               BNE DECR2           208,243
               JMP INCR1           76,25,6
```

Box 35A. (Cont.)

You can change the pitch of the sound by changing the delay value. A shorter delay yields a higher pitch. The two programs differ in that the second program (box 35) effectively generates triangle waves with two different frequencies. When you compare the two assembly listings you will see that the program in box 34 creates a triangle waveform with a delay value of 40 and then creates a triangle waveform with a delay value of 30. There is a very noticeable difference in the sound produced by this change.

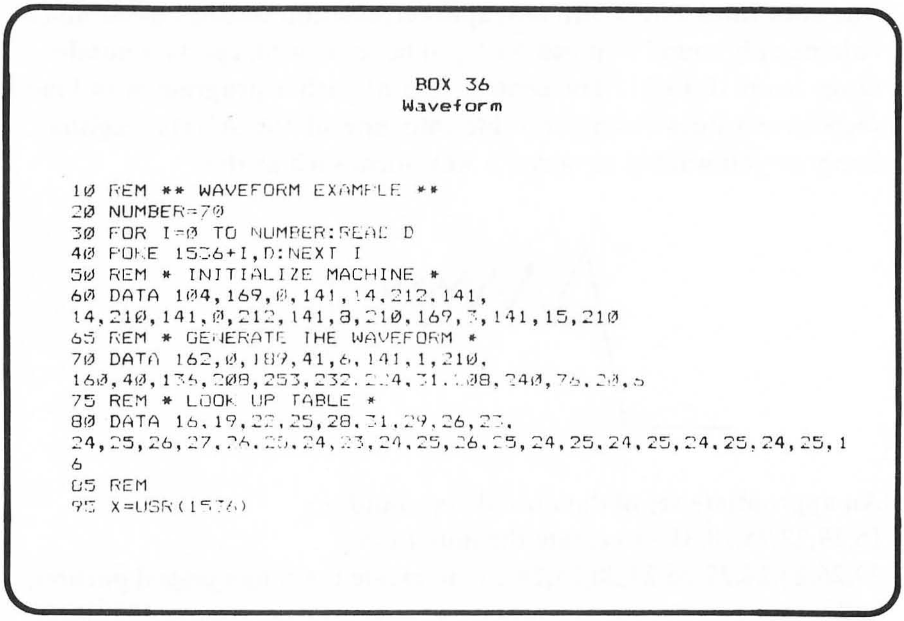
There are two simple exercises that you should do at this point. The first is to put comments into the assembly listing in box 35A. The second is to rewrite the program so that it carries out the same task but in a more efficient manner. The program as written is straightforward but repetitive. Anytime that you have a repetitive set of commands such as we have here, it should be possible to write the program code more efficiently.

The most versatile way to use the volume only mode is to generate sound waveforms from a set of data numbers stored in a look up table. The data table holds different speaker position settings 0-15. Since volume only sound requires bit D_4 to be set as well, the data numbers range from 16 to 31. The central idea of such a program is to load successive values from the table into one of the AUDC registers. Suppose you wanted to create a waveform such as this:



An appropriate set of data numbers would be:
16,19,22,25,28,31 - to create the initial ramp
29,26,23,24,25,26,27,26,25,24,23 - to create the rough jagged portion, and
24,25,26,25,24,25,24,25,24,25,24,25,16 - to create the fine jagged portion.

This waveform repeated over and over again will produce a note subtly different from those in previous programs. The reason is that if the waveform were synthesized by the addition of sine waves it would have a different fundamental frequency and different harmonics or overtones than the previous examples such as the triangle waveform. By programming different waveforms you can experiment with note timbre, or quality. Box 36 is a program that uses the data numbers given above to produce a continuous note. Writing music this way is a considerable task since the duration of the note and the frequency must be written into the program. The frequency is controlled by the delay portion (see the assembly listing in box 36A). For music, the delay portion must be modified to access a table of frequencies for each note. The frequencies must be calculated from knowledge of machine cycles or determined experimentally with the help of a piano or other musical instrument. The duration of the note can be handled by replacing the JMP START instruction with program lines that make the waveform repeat a suitable number of times.



BOX 36
Waveform

```
10 REM ** WAVEFORM EXAMPLE **
20 NUMBER=70
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,141,
14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM * GENERATE THE WAVEFORM *
70 DATA 162,0,189,41,6,141,1,210,
160,40,136,208,253,232,224,31,208,240,76,20,6
75 REM * LOOK UP TABLE *
80 DATA 16,19,22,25,28,31,29,26,23,
24,25,26,27,26,25,24,23,24,25,26,25,24,25,24,25,24,25,24,25,1
6
85 REM
95 X=USR(1536)
```

Box 36. Waveform

BOX 36A

Assembly Language Listing
for
Waveform Example

```
START  LDX #0       162,0      Initialize the index register
LOOPA  LDA TABLE,X  189,41,6   Load Accumulator from table
       STA AUDC1    141,1,210  Store speaker position in AUDC1
       LDY #40      160,40     DELAY section. Change value
LOOP B DEY          136        in Y-register to change
       BNE LOOPB    208,253    frequency
       INX          232        Increment X to point to next value
       CPX #31      224,31     Is table all read?
       BNE LOOPA    208,240    If not, continue
       JMP START    76,20,6    Go back to beginning.
```

Box 36A. Assembly language listing for waveform example

Because of the complexity of writing music programs this way, most programmers will probably want to start experimenting with music by using the pure tone option and loading the frequency registers with data numbers for each note. Box 37 is a BASIC program that calculates the data numbers for 8-bit music. This program is easy to use and will be adequate for most applications. As a point of reference, when the program asks "What octave?", the octave beginning with middle C is octave four.

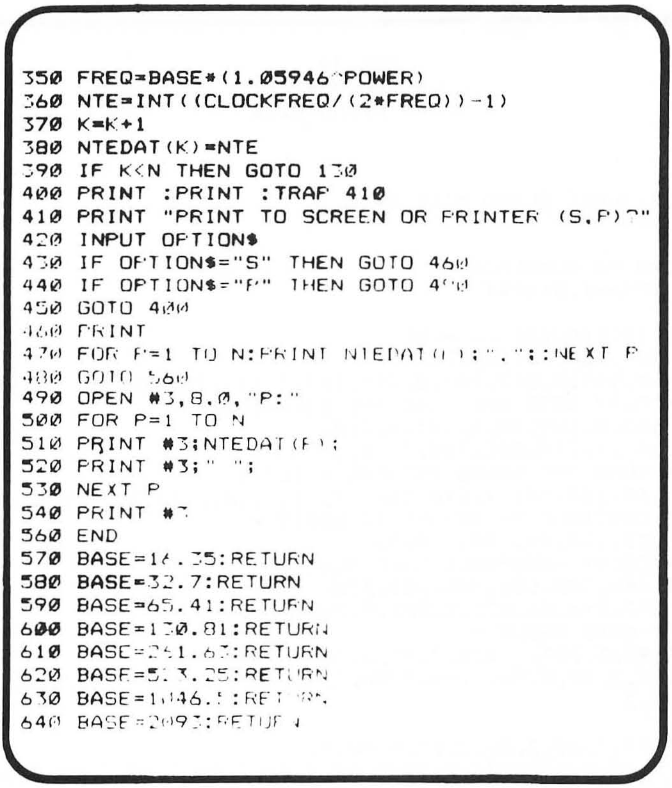
The next program, box 38, plays "Three Blind Mice". The notes are taken from a data table that holds frequency and duration values. When you study the assembly listing, you will gain further appreciation for how fast the computer operates. In order to slow the computer down to a human time frame, it was necessary to use three loops in the delay portion of the program that controls the note duration and two loops in a delay that separates one note from another. In the duration delay, one loop uses the Y-register, another uses a page 0 memory location that we call DREG, and the last uses a value loaded into the accumulator from the data table. This is the value that you change to produce a quarter note, half note, or whole note. Notice that while there are increment and decrement instructions for the index registers and memory locations, there aren't any such instructions for the accumulator. Consequently you must use addition or subtraction to increment or decrement the accumulator.

BOX 37

Utility Program
8-Bit Music Data Generator

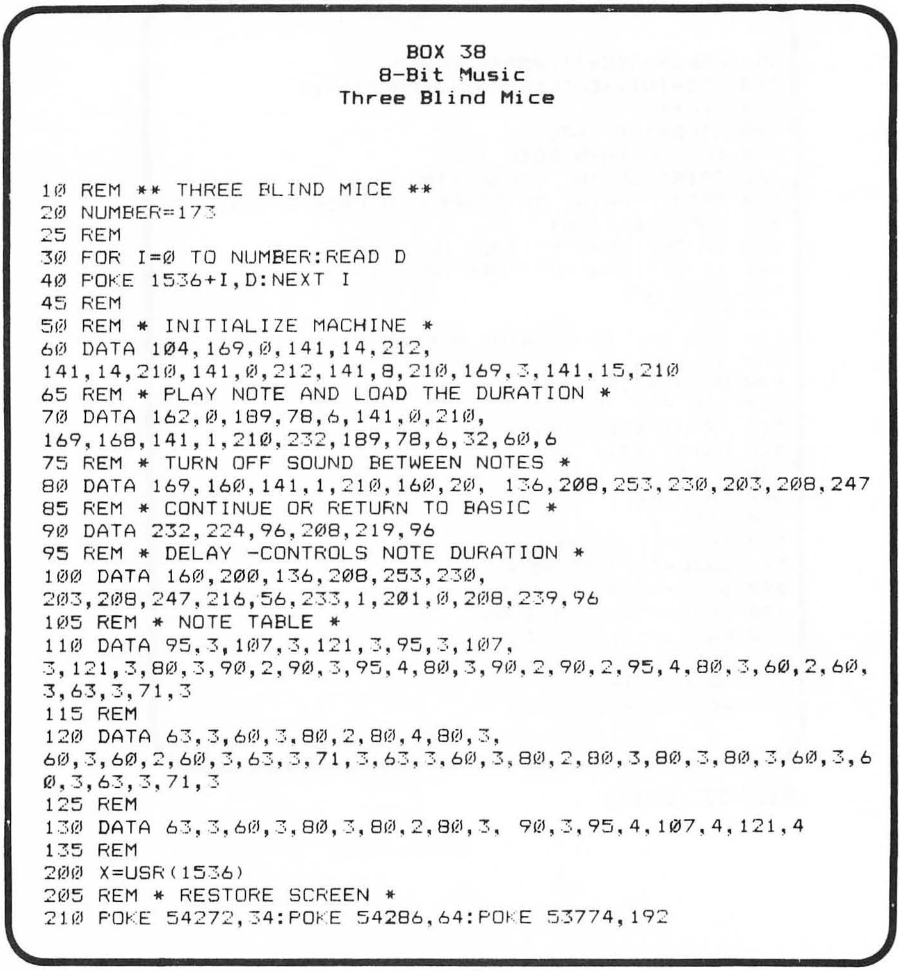
```
10 REM ** PROGRAM TO GENERATE 8-BIT MUSIC DATA NUMBERS **
20 PRINT
30 PRINT "HOW MANY NOTES?":TRAP 30
40 INPUT N
50 DIM NTEDAT(N),NTE$(1),NTETYPE$(1),FREQ$(3),OPTION$(1)
60 PRINT
70 PRINT "WHAT CLOCK FREQUENCY? ('15K' OR '64K')":TRAP 70
80 INPUT FREQ$
90 IF FREQ$="15K" THEN CLOCKFREQ=15699.9:GOTO 120
100 IF FREQ$="64K" THEN CLOCKFREQ=63921:GOTO 120
110 GOTO 60
120 PRINT
130 PRINT "WHAT OCTAVE (0 - 7)?":TRAP 130
140 INPUT OCTAVE
150 OCTAVE=OCTAVE+1
160 ON OCTAVE GOSUB 570,580,590,600,610,620,630,640
170 PRINT
180 PRINT "WHAT NOTE?":TRAP 180
190 INPUT NTE$
200 IF NTE$="C" THEN POWER=0:GOTO 280
210 IF NTE$="D" THEN POWER=2:GOTO 280
220 IF NTE$="E" THEN POWER=4:GOTO 280
230 IF NTE$="F" THEN POWER=5:GOTO 280
240 IF NTE$="G" THEN POWER=7:GOTO 280
250 IF NTE$="A" THEN POWER=9:GOTO 280
260 IF NTE$="B" THEN POWER=11:GOTO 280
270 GOTO 180
280 PRINT
290 PRINT "NATURAL (N), SHARP (S) OR FLAT (F)":TRAP 290
300 INPUT NTETYPE$
310 IF NTETYPE$="N" THEN GOTO 350
320 IF NTETYPE$="S" THEN POWER=POWER+1:GOTO 350
330 IF NTETYPE$="F" THEN POWER=POWER-1:GOTO 350
340 GOTO 290
```

BOX 37. Utility program--8-bit music data generator



```
350 FREQ=BASE*(1.05946^POWER)
360 NTE=INT((CLOCKFREQ/(2*FREQ))-1)
370 K=K+1
380 NTEDAT(K)=NTE
390 IF K<N THEN GOTO 130
400 PRINT :PRINT :TRAP 410
410 PRINT "PRINT TO SCREEN OR PRINTER (S,P)?"
420 INPUT OPTION$
430 IF OPTION$="S" THEN GOTO 460
440 IF OPTION$="P" THEN GOTO 490
450 GOTO 400
460 PRINT
470 FOR P=1 TO N:PRINT NTEDAT(P);",";:NEXT P
480 GOTO 560
490 OPEN #3,8,0,"P:"
500 FOR P=1 TO N
510 PRINT #3;NTEDAT(P);
520 PRINT #3;" ";
530 NEXT P
540 PRINT #3
560 END
570 BASE=16.35:RETURN
580 BASE=32.7:RETURN
590 BASE=65.41:RETURN
600 BASE=130.81:RETURN
610 BASE=261.63:RETURN
620 BASE=523.25:RETURN
630 BASE=1046.5:RETURN
640 BASE=2093:RETURN
```

BOX 37. (Cont.)



```
BOX 38
8-Bit Music
Three Blind Mice


10 REM ** THREE BLIND MICE **
20 NUMBER=173
25 REM
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
45 REM
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,
141,14,210,141,0,212,141,8,210,169,3,141,15,210
65 REM * PLAY NOTE AND LOAD THE DURATION *
70 DATA 162,0,189,78,6,141,0,210,
169,168,141,1,210,232,189,78,6,32,60,6
75 REM * TURN OFF SOUND BETWEEN NOTES *
80 DATA 169,160,141,1,210,160,20, 136,208,253,230,203,208,247
85 REM * CONTINUE OR RETURN TO BASIC *
90 DATA 232,224,96,208,219,96
95 REM * DELAY -CONTROLS NOTE DURATION *
100 DATA 160,200,136,208,253,230,
203,208,247,216,56,233,1,201,0,208,239,96
105 REM * NOTE TABLE *
110 DATA 95,3,107,3,121,3,95,3,107,
3,121,3,80,3,90,2,90,3,95,4,80,3,90,2,90,2,95,4,80,3,60,2,60,
3,63,3,71,3
115 REM
120 DATA 63,3,60,3,80,2,80,4,80,3,
60,3,60,2,60,3,63,3,71,3,63,3,60,3,80,2,80,3,80,3,80,3,60,3,6
0,3,63,3,71,3
125 REM
130 DATA 63,3,60,3,80,3,80,2,80,3, 90,3,95,4,107,4,121,4
135 REM
200 X=USR(1536)
205 REM * RESTORE SCREEN *
210 POKE 54272,34:POKE 54286,64:POKE 53774,192
```

Box 38. 8-bit music "Three Blind Mice"

BOX 38A
Assembly Language Listing for Three Blind Mice

Section 1: *Initialize the machine*

```
    PLA              104           Remove parameter count
    LDA #0           169,0
    STA NMIEN        141,14,212    Turn off
    STA IRQEN        141,14,210    interrupts
    STA DMACTL       141,0,212     Turn off ANTIC
    STA AUDCTL       141,8,210
    LDA #3           169,3         Initialize POKEY
    STA SKCTL        141,15,210
```

Section 2: *Play the notes and load the duration*

```
START LDX #0         162,0         Initialize the index register
LOOPA LDA TABLE,X    189,78,6      Load the note value
    STA AUDF1        141,0,210     Store in frequency register 1
    LDA #168         169,168       Load pure tone and ½ volume
    STA AUDC1        141,1,210     Store in AUDC1
    INX              232           Increment X to point to duration
    LDA TABLE,X      189,78,6      Load the duration value
    JSR DELAY        32,60,6       Jump to the delay
```

Box 38A. Assembly language listing for Three Blind Mice

```
Section 3: Turn off sound between notes
     LDA #160          169,160        Load pure tone 0 volume
     STA AUDC1         141,1,210      Store in AUDC1
LOOPB LDY#20           160,20         A two loop delay routine
LOOPC DEY              136            The value loaded in the Y
     BNE LOOPC         208,253        register helps to determine
     DEC DREG          230,203        the tempo by controlling
     BNE LOOPB         208,247        the pause between notes.

Section Four: Continue or return
     INX               232            Increment X to next note
     CPX TABLEND       224,96         Check if finished
     BNE LOOPA         208,233        If not, continue
     RTS               96             If yes, return to basic
Section 5: Delay routine
DELAY LDY #200         160,200        Load Y with delay value
LOOPD DEY              136            for first loop. Change this
     BNE LOOPD         208,253        for precise timing
     DEC DREG          230,203        Second delay
     BNE DELAY         208,247        loop
     CLD               216            Third delay loop using
     SEC               56             binary subtraction of 1
     SBC #1            233,1          from the duration value
     CMP #0            201,0          Loaded from the data table
     BNE DELAY         208,239
     RTS               96             Return from delay subroutine
```

BOX 38A. (Cont.)

The "Three Blind Mice" program can be used as a starting point for other music programs. To play another song, just change the data numbers in line 20 and the value of TABLEND in section four of the assembly listing. Of course there are other ways to modify the program. You can introduce attack, sustain and decay or vibrato to make the music richer. A straightforward way to modify the program is to play chords. So far all of the program examples have used only a single sound channel. The program in box 39 plays "Three Blind Mice" using two sound channels.

```
                         BOX 39
             Three Blind Mice with Chords


10 REM ** THREE BLIND MICE WITH CHORDS **
20 NUMBER=234
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA 104,169,0,141,14,212,
141,14,210,141,0,212,141,8.210,169,3.141,15.210
65 REM * PLAY NOTES AND LOAD DURATION *
70 DATA 162.0,189.91,6.141,0,210,
232,189,91,6,141,2,210.169.168,141.1,210,141.3,210,232,189,91
,6,32,73,6
75 REM * TURN OFF SOUND BETWEEN NOTES *
80 DATA 169,160,141,1,210,141,3,
210,160,20,136,208.253,230.203,208,247
85 REM * CONTINUE OR RETURN TO BASIC *
90 DATA 232,224.144,208,206,96
95 REM * DELAY -CONTROLS NOTE DURATION *
100 DATA 160,200,136,208.253,230,
203,208,247,216,56,233,1.201,0,208,239,96
105 REM * NOTE TABLE *
110 DATA 95,162,3,107,182,3,121,
192,3,95,162,3,107,182,3,121,192.3.80,128.3,90.144,2,90,144,3
,95,162,4,80,128
120 DATA 3,90.144,2,90.144.2,95.
162,4,80,162,3,60.121.2,60.121,3,63.107.3,71,95,3,63,107,3,60
,95,3,80.128,2
130 DATA 80,128,4,80.128.3,60,95.3.
60.95,2,60,95.3,63,107,3,71,95.3,63,107,3.60,95,3,80,95.2,80,
95,3,80,128
140 DATA 3,80.128.3,60,95.3,60,95.3,
63,107,3,71,121,3,63,107.3,60.95.3,80,121.3.80,121,2,80,121.3
,90,144,3
150 DATA 95,162.4,107.182.  4.121.192.4
200 X=USR(1536)
205 REM * RESTORE SCREEN *
210 POKE 54272,34:POKE 54286,64:POKE 53774,192
```

Box 39. "Three Blind Mice" with chords

Box 39A

Assembly Language Listing for Three Blind Mice with Chords

Section 1: *Initialize the machine*

```
    PLA             104         Remove parameter count
    LDA #0          169,0
    STA NMIEN       141,14,212  Turn off
    STA IRQEN       141,14,210  interrupts
    STA DMACTL      141,0,212   Turn off ANTIC
    STA AUDCTL      141,8,210
    LDA #3          169,3       Initialize POKEY
    STA SKCTL       141,15,210
```

Section 2: *Play the notes and load the duration*

```
START LDX #0        162,0
LOOPA LDA TABLE,X 189,91,6
    STA AUDF1       141,0,210
    INX             232
    LDA TABLE,X     189,91,6
    STA AUDF2       141,2,210
    LDA #168        169,168
    STA AUDC1       141,1,210
    STA AUDC2       141,2,210
    INX             232
    LDA TABLE,X     189,91,6
    JSR DELAY       32,73,6
```

Box 39A. Assembly language listing for Three Blind Mice with Chords

Section 3: *Turn off sound between notes*

```
          LDA #160        169,160
          STA AUDC1       141,1,210
          STA AUDC2       141,3,210
LOOPB LDY #20             160,20
          LOOPC DEY       136
          BNE LOOPC       208,253
          DEC DREG        230,203
          BNE LOOPB       208,247
```

Section 4: *Continue or return*

```
          INX             232
          CPX TABLEND     224,144
          BNE LOOPA       208,206
          RTS             96
```

Section 5: *Delay Routine*

```
          LDY #200        160,200
          DEY             136
          BNE LOOPC       208,253
          DEC DREG        230,203
          BNE DELAY       208,247
          CLD             216
          SEC             56
          SBC #1          233,1
          CMP #0          201,0
          BNE DELAY       208,239
          RTS             96
```

Box 39A. (Cont.)

Sixteen Bit Music

At times music generated by eight bit data numbers will be unsatisfactory because some of the notes will sound slightly flat or too sharp. As the example earlier in this chapter showed, this is because eight bits will not always give an adequate selection of frequencies. The problem can be remedied by joining the sound channels into pairs with channels 1 and 2 forming one pair and channels 3 and 4 the other. Because the bytes in the paired AUDF registers are used together, the numbers in the "divide by circuit" ranges from 0 to 65536.

The 16-bit music option is controlled by bits D_3 and D_4 of ADDCTL (53768). Bit D_4 joins channels 1 and 2; bit D_3 joins channels 3 and 4. Each note will have two data numbers - a Hi-Byte and a Lo-Byte. If channels 1 and 2 are paired, the Lo-Byte goes into AUDF1 and the Hi-Byte into AUDF2. When channels 3 and 4 are paired, the Lo-Byte goes into AUDF3 and the Hi-Byte into AUDF4. Data numbers for octave 0 - 8 are given in Appendix H. The data numbers are based on the system clock frequency, 1.79MHZ. This requires bit D_6 of AUDCT2 to be set.

A program to play "Three Blind Mice" using 16-bit music is given in box 40 and the assembly language listing in box 40A. The program pairs channels 1 and 2. While the program is similar to the previous program in Box 39, there are some differences to note. In the initialization section the value stored in AUDCTL sets bits D_6 and D_4. AUDC1 is set to zero since the choice of pure tone and volume are controlled by AUDC2. In the section that loads the note data values AUDF1 is the target for the Lo-Byte and AUDF2 receives the Hi-Byte.

To summarize, for 16-bit music you:

- Set bits 6 and 4 of AUDCTL to join channels 1 and 2 or
- Set bits 6 and 3 of AUDCTL to join channels 3 and 4
- Turn off AUDC of the lower numbered channel
- Store Lo-Byte of data number in lower channel

- Store Hi-Byte of data number in higher channel
- Store volume in AUDC3 or AUDC4 as required

```
10 REM ** THREE BLIND MICE -16 BIT MUSIC **
20 NUMBER=235
30 FOR I=0 TO NUMBER:READ D
40 POKE 1536+I,D:NEXT I
50 REM * INITIALIZE MACHINE *
60 DATA
104,169,0,141,14,212,141,14,210,141,0,212,169,160.141,1,210,1
69,80,141,8,210,169,3,141,15,210
65 REM * PLAY NOTES AND LOAD DURATION *
70 DATA
162,0,189,92,6,141,0,210,232,189,92,6,141,2,210,169,168,141.3
,210,232,189,92,6,32,74,6
75 REM * TURN OFF SOUND BETWEEN NOTES *
80 DATA 169,160,141,3,210,160,20,136,208,253,230,203,208,247
85 REM * CONTINUE OR RETURN TO BASIC *
90 DATA 232,224,144,208,212,96
95 REM * DELAY -CONTROLS NOTE DURATION *
100 DATA
160,150,136,208,253,230,203,208,247,216,56,233,1.201,0,208,23
9,96
105 REM * NOTE TABLE *
110 DATA
148,10,3,224,11,3,85,13,3,148,10,3,224,11,3,85,13,3,228,8,3,2
51,9,2,251,9,3,148,10,4,228,8,3,251,9,2
120 DATA
251,9,2,148,10,4,228,8,3,167,6,2,167,6,3,13,7,3,235,7,3,13,7.
3,167,6,3,228,8,2,228,8,4,228,8,3,167
130 DATA
6,3,167,6,2,167,6,3,13,7,3,235,7,3,13,7,3,167,6,3,228,8,2.228
,8,3,228,8,3,228,8,3,167,6,3,167,6,3
140 DATA
13,7,3,235,7,3,13,7,3,167,6,3,228,8,3,228,8,2,228,8,3,251,9,3
,148,10,4,224,11,4,85,13,4
150 DATA 95,162,4,107,182,4,121,192,4
200 X=USR(1536)
205 REM * RESTORE SCREEN *
210 POKE 54272,34:POKE 54286,64:POKE 53774,192
```

Box 40. 16-bit music "Three Blind Mice"

BOX 40A

16 Bit Music Program
Three Blind Mice

```
START LDX #0            162,0       Load counter
LOOPA LDA TABLE,X       189,92,6    Load Lo-Byte of music data number.
      STA AUDF1         141,0,210   Store it in AUDFI
      INX               232         Increment X to point to next value
      LDA TABLE,X       189,92,6    Load Hi-Byte of music data no.
      STA AUDF2         141,2,210   Store it in AUDF2
      LDA #168          169,168     Load volume number
      STA AUDC2         141,3,210   Store volume number in AUDC2
      INX               232         Increment X to point to delay value
      LDA TABLE,X       189,92,6    Load delay value
      JSR DELAY         32,74,6     Jump to delay (keep note playing)
      LDA #160          169,160     Load zero volume
      STA AUDC2         141,3,210   Store in AUDC2 to turn sound off
LOOPB LDY #20           160,20      Start delay loop for
LOOPC DEY               136         sound off
      BNE LOOPC         208,253
      DEC DREG          230,203
      BNE LOOPB         208,247     Is delay loop off? No, continue
```

Box 40A. 16 Bit music program "Three Blind Mice"

```
   INX              232       Increment X to point to next value
   CPX TABLEND      224,144   Is the table ended?
   BNE LOOPA        208,212   No, branch back to beginning
   RTS              96        Return from subroutine
   DELAY LDY #200   160,200   Load initial delay value
LOOPD DEY           136       First Delay
   BNE LOOPD        208,253   inner loop
   DEC DREG         230,203   Another inner
   BNE DELAY        208,247   delay loop
   CLD              216       Clear decimal mode
   SEC              56        set carry            \  outer delay
   SBC #1           233,1     subtract             |  determines
   CMP #0           201,0     is accumulator 0     {   length of
   BNE DELAY        208,239   if not, recycle      /     note
   RTS              96        Return from delay
```

Box 40A. (Cont.)

Summary

In this chapter we have presented the fundamentals of music but have only scratched the surface of what you can do. There are many options available. We have mentioned tremolo, vibrato, and chords. You can put in glides, attack, sustain, decay. The sound channels do not have to be turned on and off at the same time as we did in our simple programs. There is one option yet to go. That is to combine music with the vertical blank interrupt. This option is the subject of the next chapter.

6

Advanced Techniques

Introduction

As you now know, most of the sound and graphics features of the Atari Home Computer are, to some extent, accessible from BASIC. However, there are situations in which the only satisfactory implementation of sound or graphics comes through machine language subroutines. For example, you can use BASIC to play music, but if you want to play music as an integral part of your program, it can only be done in machine language during the TV's vertical blank. You can detect collisions, or write programs to go with a touch tablet in BASIC, but these tasks are also done more satisfactorily in the vertical blank with machine language.

The OS vertical blank routines are among the more powerful and versatile features of Atari computers. In this chapter we will focus on describing what happens during the vertical blank and how to integrate your own routine(s) into the regular OS protocols. We will illustrate the general procedures with examples that demonstrate scrolling, music, and input with a touch tablet.

The Vertical Blank Routines

When the electron beam that generates the TV display comes to the end of the last scan line, the display hardware sends a non-maskable interrupt to the CPU. In response, the CPU tests to see if the interrupt was caused by a DLI, a Reset, or a Vertical Blank Interrupt (VBI).

Figure 6-1 is a flowchart of the vertical blank routine. After determining that the interrupt is a VBI, the OS jumps to the location pointed to by VVBLKI (locations 546,547). Stored in these memory locations are the Lo-Byte and Hi-Byte of the address of a subroutine that is normally carried out during each vertical blank. In technical jargon, locations such as 546 and 547 which store addresses of subroutines are called pointers or vectors. Normally the 'vector' at 546,547 points to 58463. This is the first of three places in the VB routine where you can insert your own machine language program. You do this by 'stealing the vector', ie. POKE the starting address of your own routine into 546,547. Depending on your intent, your routine should end with a JMP back to the operating system's program at 58463, or to exit the vertical blank program by a JMP to XITVBL at 58466.

The routine at 58463 is called the Stage I or Vertical Blank Immediate routine. During the Stage I routine, the OS:

- increments the real time clock
- decrements system timer one
- performs color attracting

Once this is completed, the OS checks memory location 66 (CRITIC). If CRITIC has a non-zero value, then a time critical code section is being executed and the program jumps to XITVBL. If the code is not critical then Stage II of the vertical blank routine is processed. During Stage II the OS takes care of the following 'housekeeping' chores:

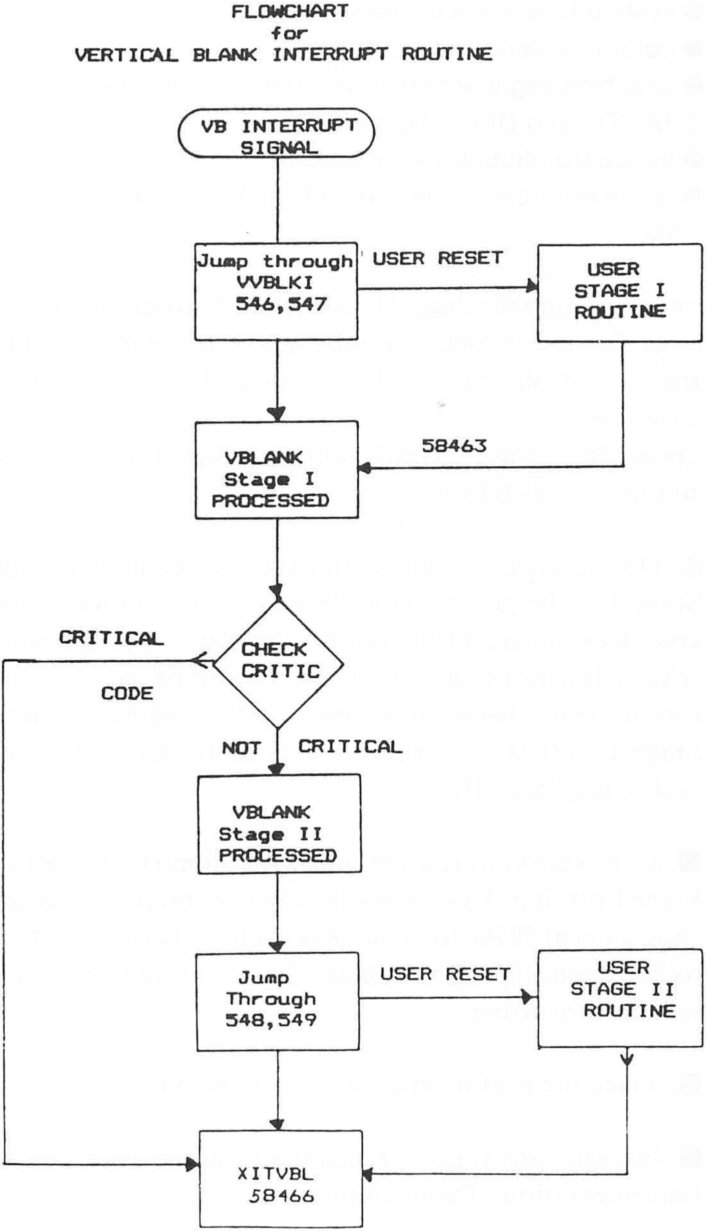


Figure 6-1. Flowchart for vertical blank interrupt routine

- system timers are decremented
- color registers are updated
- graphics registers such as CHBASE, PRIOR, CHACTL, and DMACTL are updated
- keyboard utilities are processed
- game controller data is read from hardware to RAM

After completing the Stage II tasks, the OS program jumps to the location specified in 548,549. Normally this vector points to XITVBL. This is the second place to steal a vector and make it point to an alternate routine.

The procedure to use vertical blank interrupts for your programs can be summarized as follows.

- Decide whether your routine is to be executed in Stage I or Stage II. The general considerations that should enter into your decision are: (1) if your routine must be mixed with time critical functions such as disk I/O, use Stage I. (2) If your routine reads from, or writes to, OS shadow registers, use Stage II. (3) If your routine uses more than 3000 machine cycles, use Stage II.

- Make sure your routine ends with a jump to: (a) 58463 for a Stage I rotuine. This allows the OS to continue its normal VB chores or (b) 58466 for a Stage II routine. This allows the CPU to exit from the vertical blank interrupt and go back to its regular processing.

- Place the routine in a safe area in memory.

- Initialize any timers or memory locations necessary for the implementation of your routines.

- Link your routine to the proper vertical blank stage by a short machine language program that stores the routine's starting address at 546,547 for Stage I or 548,549 for Stage II.

The linking mentioned in the last step is provided for in the operating system. The OS contains a subroutine called SETVBV (Set Vertical Blank Vector) which installs addresses in the Stage I and Stage II pointers. To link a subroutine to Stage I, use a USR command to call this short program.

BOX 41

Vertical Blank Linking Routine

```
PLA             104
LDY ADDRLO      160,ADDRLO Load Y with Lo-Byte of routines
                           address
LDX ADDRHI      162,ADDRHI Load X with Hi-Byte of routines
                           address
LDA #7          169,7      7 indicates Stage II link.
JSR SETVBV      32,92,228  Jump to SETVBV.
RTS             96         Return
```

ADDRLO and ADDRHI are the Lo-Byte/Hi-Byte of the address location where you store your routine.

Box 41. Vertical blank linking routine

In this program (Box 41) ADDRLO and ADDRHI of course depend on where you store your routine. The 7 loaded into the accumulator flags SETVBV that this is to be linked to Stage II. If you want to link a routine to Stage I, load the accumulator with 6 (LDA #6) instead.

The reason for the SETVBV subroutine is that it prevents system lockup. The vertical blank vectors are two byte critters. Setting these vectors with BASIC POKE statements runs the risk of an interrupt occuring before both locations are updated and... that would crash the program!

So far we have mentioned several memory locations in connection with vertical blank interrupts. Table 6-1 lists several more. Actually this just scratches the surface since all the shadow/hardware register combinations, collision registers, priority registers, etc. are potentially useful. In addition to the system registers that we have discussed, table 6-1 lists timer registers.

TABLE 6-1

Memory Locations Useful With VBIs

TIMERS

(a) 18,19,20	The real time clock. Incremented each VB Stage I
(b) 536,537	(CDTMV1) System timer 1. When it reaches 0 it sets a flag to jump through the addresses in 550,551. Used by OS.
(c) 538,539	(CDTMV2) System timer 2. Decremented every Stage II. Performs a JSR through location 552,553 when value counts down to 0. Very useful for VB routines.
(d) 540,541	(CDTMV3) System timer 3. Sets a flag at 554 when counts to 0. Used for cassette I/O.
(e) 542,543	(CDTMV4) System timer 4. Sets a flag when decremented to 0.
(f) 544,545	(CDTMV5) System timer 5. Sets a flag at 558 when decremented to 0.

SYSTEM REGISTERS:

(a) 546,547	Vertical Blank Stage I Vector explained in text.
(b) 548,549	Vertical Blank Stage II vector explained in text.
(c) 58460	SETBV routine to link in a user vertical blank routine.
(d) 58463	SYSVBV Stage I VB entry point. A user Stage I routine should end with a jump to this address.
(e) 58466	XITVBV Exit from vertical blank. A Stage II routine should end with a jump to this address.

Table 6-1. Memory locations useful with VBIs

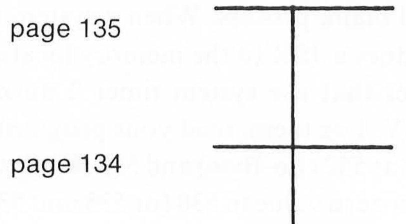
System timer 2 is the third place to add a machine language routine to the vertical blank process. When a system timer 2 counts down to zero the OS does a JSR to the memory location specified in 552 and 553. Routines that use system timer 2 do not have to be installed with SETVBV. For them, read your program into memory, put its starting address at 552 (Lo-Byte) and 553 (Hi-Byte). To start the program, POKE a non-zero value in 538 (or 538 and 539 if you need a long delay before it begins).

Scrolling

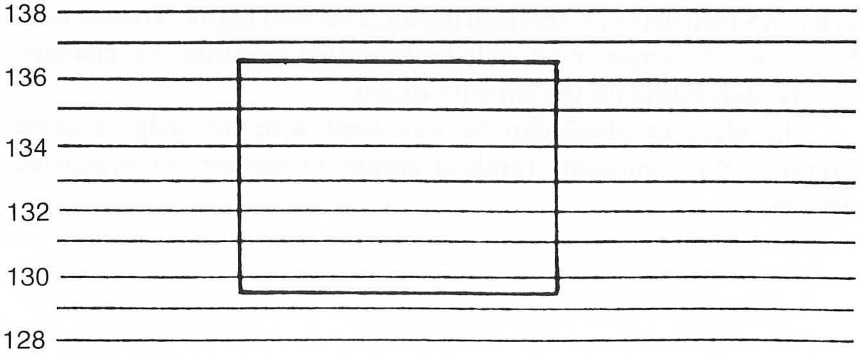
Our first programs that use the vertical blank will illustrate horizontal and vertical scrolling. Scrolling in the Atari Home Computer is best thought of as using the TV screen as a window on memory. Using ANTIC's LMS instruction and two fine scroll registers, the window can be moved smoothly from one part of memory to another. Scrolling done this way is more versatile than scrolling in machines that use a fixed area in memory for the screen display. In this case you have to move information (bytes) through the screen memory, sometimes moving thousands of bytes. With the Atari you can create a large picture and scroll across it as if it were a panaroma.

The heart of scrolling is manipulating ANTIC's display list and the two fine scroll registers, HSCROL (54276) and VSCROL (54277). Recall that ANTIC's display list is a microprocessor program and as such can be modified. Any mode line instruction can also have the load memory scan option (LMS instruction). Suppose that the address for each LMS instruction is changed during a vertical blank. Then what is shown on the screen next will have shifted position, or changed entirely, depending on the value(s) stored.

The ideas involved can be visualized with the help of some diagrams. We commonly think of computer memory as organized vertically:



For the purposes of discussion, it is better to think of memory as cut into pieces that are lined up horizontally:



The square superimposed on the horizontal lines represents the screen window. Move this window up or down and the display scrolls vertically. Move the window left or right and the display scrolls horizontally. Coarse movement requires proper configuration of the display list and manipulating LMS address bytes. Fine movement is accomplished by changing the value in HSCROL or VSCROL. Extended smooth motion requires using a combination of both fine and coarse scrolling during the vertical blank.

Now that you know the general ideas, let's see how to put them into action with a vertical scrolling program. The program in box 42 scrolls a yellow submarine up the screen. Since this program is more complicated than previous examples in the book we will describe it in detail.

The first consideration in designing the program is how to organize memory. The submarine, constructed with redefined characters, is displayed on a full Graphics 2 screen. Space must be put aside for the character set and screen memory. The screen memory area should be cleared and large enough so that if the display window is scrolled either above or below the submarine no unwanted characters appear. Since a full screen Graphics 2 uses 240 bytes, 1K of memory set aside is sufficient. Besides room for the character set and screen memory, we also need space for a custom display list and the machine language scrolling routine. To provide room for the above, the program starts by lowering RAMTOP 12 pages to page 148 (37888). The character set occupies 1K bytes beginning at page 154, leaving 512 bytes below it empty (pages 152,153). By starting the display list at page 157, two pages are left empty between screen memory and the next data in memory. Finally, the scrolling routine occupies the lower part of page 158. Certainly this is rather loose use of memory since the full character set is not needed, and neither the display list nor the scrolling routine need a full page. However, it was written this way so that you could use it as a skeleton for other, larger programs.

BOX 42
Vertical Scrolling
The Yellow Submarine

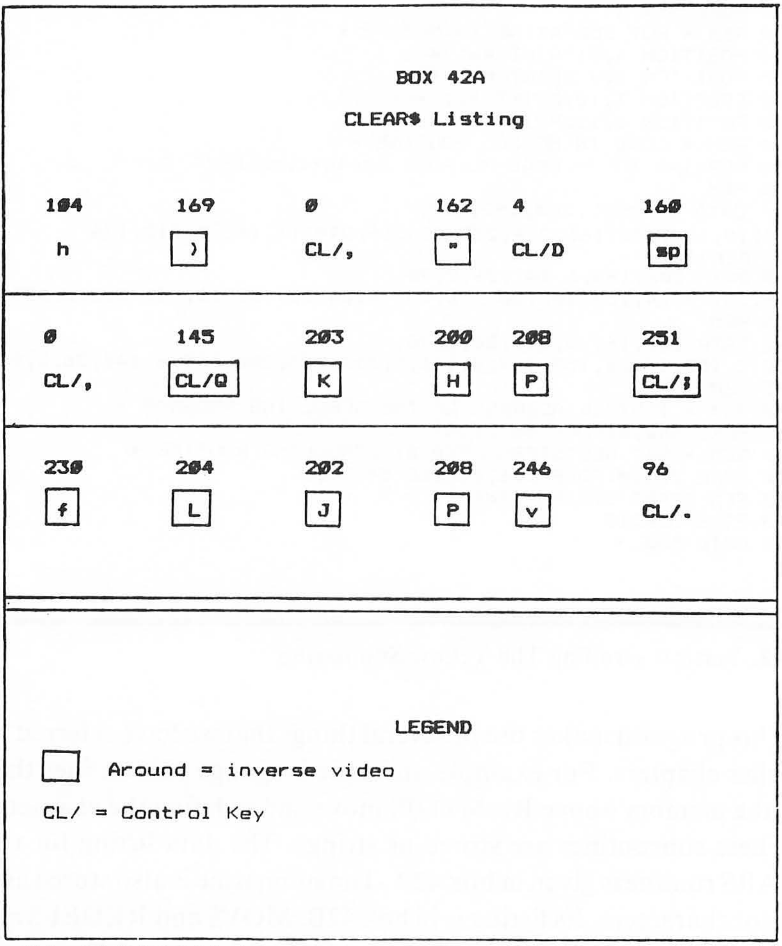
```
5 REM ** YELLOW SUBMARINE SCROLL **
10 REM * DIMENSION STRINGS THAT STORE ML ROUTINES *
20 REM * AND CHARACTER SET *
30 DIM CLEAR$(18),MOV$(20),REDEF$(14),SUB$(120),S$(32)
40 REM CLEAR$
50 REM MOV$
60 REM REDEF$
70 REM SUB$
80 REM S$
85 SUB$(LEN(SUB$)+1)=S$
90 REM * SET UP RESERVED SPACE AND CLEAR *
100 POKE 106,148:POKE 203,0:POKE 204,148
110 CLEAR=USR(ADR(CLEAR$))
120 REM * SET GRAPHICS MODE AND COLORS *
130 GRAPHICS 18:POKE 752,1:POKE 708,60:POKE 712,134
140 REM * MOVE STANDARD CHARACTERS/REDEFINE *
150 POKE 205,0:POKE 206,224
160 MOVE=USR(ADR(MOV$))
170 Q=ADR(SUB$)
180 HIQ=INT(Q/256)
190 LOQ=Q-HIQ*256
200 POKE 205,LOQ:POKE 206,HIQ
210 POKE 203,24:POKE 204,148
220 R=USR(ADR(REDEF$))
230 REM * SET UP CUSTOM DISPLAY LIST *
240 FOR I=0 TO 2:POKE 40192+I,112:NEXT I
250 POKE 40195,103
260 POKE 40196,0:POKE 40197,154
270 FOR I=0 TO 9:POKE 40198+I,39:NEXT I
280 POKE 40208,7
290 POKE 40209,65
300 POKE 40210,0:POKE 40211,157
```

Box 42. Vertical Scrolling The Yellow Submarine

```
310 REM * TELL ANTIC AND OS WHERE SCREEN MEMORY IS *
320 POKE 559,0
330 POKE 560,0:POKE 561,157
340 POKE 88,0:POKE 89,154
350 POKE 756,148
360 POKE 559,34
370 REM * PUT SUBMARINE IN MEMORY *
380 POSITION 6,8:PRINT #6;"#"
390 POSITION 3,9:PRINT #6;"$%&'()"
400 POSITION 3,10:PRINT #6;"*++++,"
410 POSITION 3,11:PRINT #6;"-.//01"
420 REM * LOAD IN SCROLL ROUTINE *
430 FOR I=0 TO 70:READ ML:POKE 40448+I,ML:NEXT I
435 REM
440 DATA 164,205,200,192,120,
240,19,132,205,166,206,232,224,16,240,11,142,5,212,134
445 REM
450 DATA 206,169,6,141,26,2,96,
216,24,173,4,157,105,20,176,16,141,4,157,169,0,141,5,212,133
455 REM
460 DATA 206,169,6,141,26,2,96,
238,5,157,141,4,157,169,0,141,5,212,133,206,169,6,141,26,2,96
465 REM
470 REM * INSTALL ADDRESS OF THE SCROLLING PROGRAM *
480 POKE 552,0:POKE 553,158
490 REM * SET REGISTERS USED BY SCROLLING ROUTINE *
500 POKE 205,0:POKE 206,0:POKE 54277,0
510 REM START SYSTEM TIMER 2
520 POKE 538,10
530 GOTO 530
```

Box 42. Vertical scrolling The Yellow Submarine

This program makes use of several things that we have referred to in earlier chapters. For example, machine language subroutines that clear the memory above RAMTOP, move and redefine the character set. These subroutines are stored as strings. The data listing for the CLEAR$ routine is given in box 42A. The submarine is also stored as a string of characters. Its listing is in box 42B. MOV$ and REDEF$ are presented in box 24 and box 25. Box 42C is the assembly language listing for the vertical scrolling routine.



```
                    BOX 42A
                CLEAR$ Listing

104     169     0       162   4       160
 h       )      CL/,     "    CL/D    sp

0       145     203     200   208     251
CL/,    CL/Q     K       H     P      CL/;

230     204     202     208   246     96
 f       L       J       P     v      CL/.

                    LEGEND
□   Around = inverse video
CL/ = Control Key
```

Box 42A. CLEAR$ Listing

BOX 42B

The Yellow Submarine
SUB$ Listing

1.	(#)	120	120	96	96	96	96	96	96
		x	x	CL/.					
2.	($)	0	0	0	0	0	127	127	127
		CL/,					ES/TAB		
3.	(%)	0	0	0	0	0	7	63	255
		CL/,					CL/G	?	$\boxed{\text{ES/CL>}}$
4.	(&)	15	17	49	63	127	255	255	255
		CL/O	CL/Q	1	?		$\boxed{\text{ES/CL>}}$		
5.	(’)	248	40	40	248	254	255	255	255
		$\boxed{\text{x}}$	(	(	$\boxed{\text{x}}$	$\boxed{\text{ES/BS}}$	$\boxed{\text{ES/CL>}}$		
6.	(‘)	0	0	0	0	0	224	254	255
		CL/,					CL/.	$\boxed{\text{ES/BS}}$	

(cont. on next page)

7.	(,)	0 CL/,	0	0	0	0	0	0	192 $\boxed{\text{@}}$
8.	(*)	127 ES/TB	127	63 ?	31 ES/CL*	15 CL/O	15	31	63
9.	(+)	255 $\boxed{\text{ES/CL>}}$	255	255	255	255	255	255	255
10.	(,)	224 $\boxed{\text{CL/}}$	248 $\boxed{\text{x}}$	252 $\boxed{\text{SH=}}$	252 $\boxed{\text{SH=}}$	254 $\boxed{\text{ES/CL/BS}}$	252 $\boxed{\text{SH=}}$	252 $\boxed{\text{SH=}}$	248 $\boxed{\text{x}}$
11.	(-)	255 $\boxed{\text{ES/CL>}}$	255	255	0 CL/,	0	0	0	0
12.	(.)	255	127 ES/TB	63 ?	15 CL/O	0 CL/,	0	0	0
13.	(/)	255 $\boxed{\text{ES/CL>}}$	255	255	255	0 CL/,	0	0	0
14.	(0)	255 $\boxed{\text{ES/CL>}}$	255	252 $\boxed{\text{SH=}}$	224 $\boxed{\text{CL/.}}$	0 CL/,	0	0	0
15.	(1)	248 $\boxed{\text{x}}$	224 $\boxed{\text{CL/.}}$	0 CL/,	0	0	0	0	0

LEGEND

box around = inverse video
ES = Escape Key
BS = Back space

SH = Shift
CL/ = Control Key
TB = Tab Key

Box 42B. The Yellow Submarine SUB$ Listing

BOX 42C
Vertical Scrolling Routine
The Yellow Submarine

205 (COUNT) keeps track of how far we've scrolled
206 (SCRLREG) keeps track of value to put in VSCROL.

```
      LDY COUNT    164,205   First check to see if submarine
      INY          200       has scrolled to top of screen
      CPY #120     192,120   Here, the value 120 limits
      BEQ END      240,19    the scrolling
      STY COUNT    132,205   Save Y for next pass thru routine
      LDX SCRLREG  166,206   Get value to put in VSCROL
      INX          232       Increase it by 1 scan line
      CPX #16      224,16    Have we reached the
                             maximum?
      BEQ COARSE   240,11    If so, do coarse scroll/reset
                             SCRLREG
      STX VSCROL   142,5,212If not store value in VSCROL
      STX SCRLREG  134,206   Save value for next time around
      LDA #6       169,6     Load and store a delay value. A
      STA TIMER    141,26,2  fine scroll each 6th VB.
END   RTS          96        Return from subroutine
```

(cont. on next page)

```
COARSE CLD            216          CLEAR decimal mode and carry
       CLC            24           flag for binary addition
       LDA SCNLO      173,4,157    Get Lo-Byte of screen memory
                                   address
       ADC #20        105,20       Add the no. of bytes in Gr 2
                                   mode line
       BCS ADDHI      176,16       If carry results, Increment
                                   Hi-Byte
       STA SCNLO      141,4,157    Get Lo-Byte of screen memory
                                   address
       LDA #0         169,0
       STA VSCROL     141,5,212    Reset VSCROL and SCRLREG
       STA SCRLREG    133,206
       LDA #6         169,6        Load and store a delay value
       STA TIMER      141,26,2     for a fine scroll each 6th VB
       RTS   SCNHI    96           Return from subroutine
ADDHI  INC SCNLO      238,5,157    Increment Hi-Byte of screen
                                   address
       STA SCNLO      141,4,157    update Lo-Byte of screen
                                   address
       LDA #0         169,0
       STA VSCROL     141,5,212    Reset VSCROL and SCRLREG
       STA SCRLREG    133,206
       LDA #6         169,6        Load and store a value for
       STA TIMER      141,26,2     a fine scroll every 6th VB
       RTS            96           Return from subroutine
```

Add 256 to screen

Box 42C. Vertical Scrolling Routine The Yellow Submarine

The sequence of events in the submarine scrolling program is as follows:

- The machine language strings are DIMensioned and defined.

- RAMTOP is lowered and 8 pages of memory above RAMTOP are cleared.

- The ROM character set is moved and redefined.

- A custom display list that includes setting the fine scroll option is constructed.

- The machine language scrolling routine is read into memory.

- The address of the scrolling routine is put into locations 552,553.

- Finally, the scrolling routine is started by POKEing a value into 538 so that when system timer 2 counts down to 0, the OS will jump to the scrolling routine.

Now let's look at the theory behind vertical scrolling and at the machine language scrolling routine. Refer to figure 6-2. Each horizontal line represents 20 bytes of screen memory - the number of pixels in a Graphics 2 mode line. To begin with, the address specified in the display list LMS instruction tells ANTIC to find the internal character code for the top left-hand pixel at memory location 39424 (154 Hi-Byte/0 Lo-Byte). Suppose that during a vertical blank the LMS address is changed to 154,20. This change moves the data that determined the top mode line off the screen, shifts all mode lines up one position and moves a new mode line in at the bottom. The effect produced is that of a course vertical scroll.

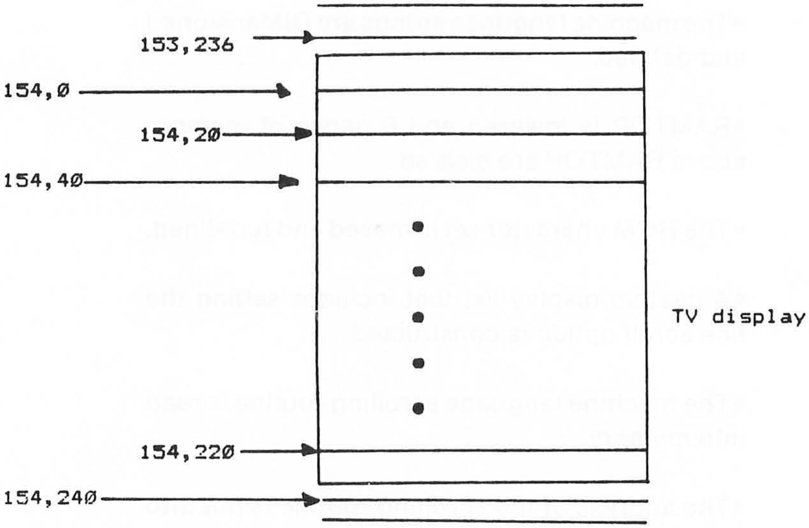


Figure 6-2. Vertical scrolling

Since the image moves one whole mode line at a time, coarse scrolling is noticeably jerky, especially in Graphics 2, where a single mode line has 16 scan lines. Using the vertical fine scroll option allows you to move a mode line, a group of mode lines, or the entire screen up or down in scan line increments. The number of scan lines to move the display is determined by the number in VSCROL. By incrementing VSCROL during the vertical blank, the display drawn is shifted up slightly from the previous one. Fine scrolling is limited by the fact that only the lower four bits of VSCROL are significant. Consequently, the largest number of scan lines that a display can be moved with fine scrolling alone is 15. To continue the scrolling beyond this limit involves invoking a coarse scroll and resetting the fine scroll register.

The process in flowchart form is shown in figure 6.3. Vertical fine scrolling can be done with the whole display or part of the display. All that is required is to set bit D5 in each display list instruction by adding 32 to the normal mode line number (see chapter 3).

Now we are in a position to look at the assembly language listing in box 42C. The program starts off with a check on how far the scrolling has progressed. This is done with a page zero register count called COUNT at location 206 which is incremented on each pass through the program. After 120 passes, a RTS is performed without resetting system timer 2, thus effectively ending program execution. Without the check on the scrolling, the screen window would scroll across the computer memory which would be interesting but not necessarily desirable. For an interesting experience, replace BEQ #19 with two NOP's.

Once the check on scrolling is done, the program loads the current value in VSCROL from a software register at 206, increments it, and compares it with 16. A natural question is "Why do we need a software register?" The answer is because VSCROL is a write only register. This means that there are some things you can't do with it such as load CPU registers from it or use the INC VSCROL instruction. If fine scrolling is not complete, which is determined by the results of CPX #16, the new value is stored in VSCROL and back into SCRLREG. Finally, a delay value of 6 is loaded into system timer 2. This value does

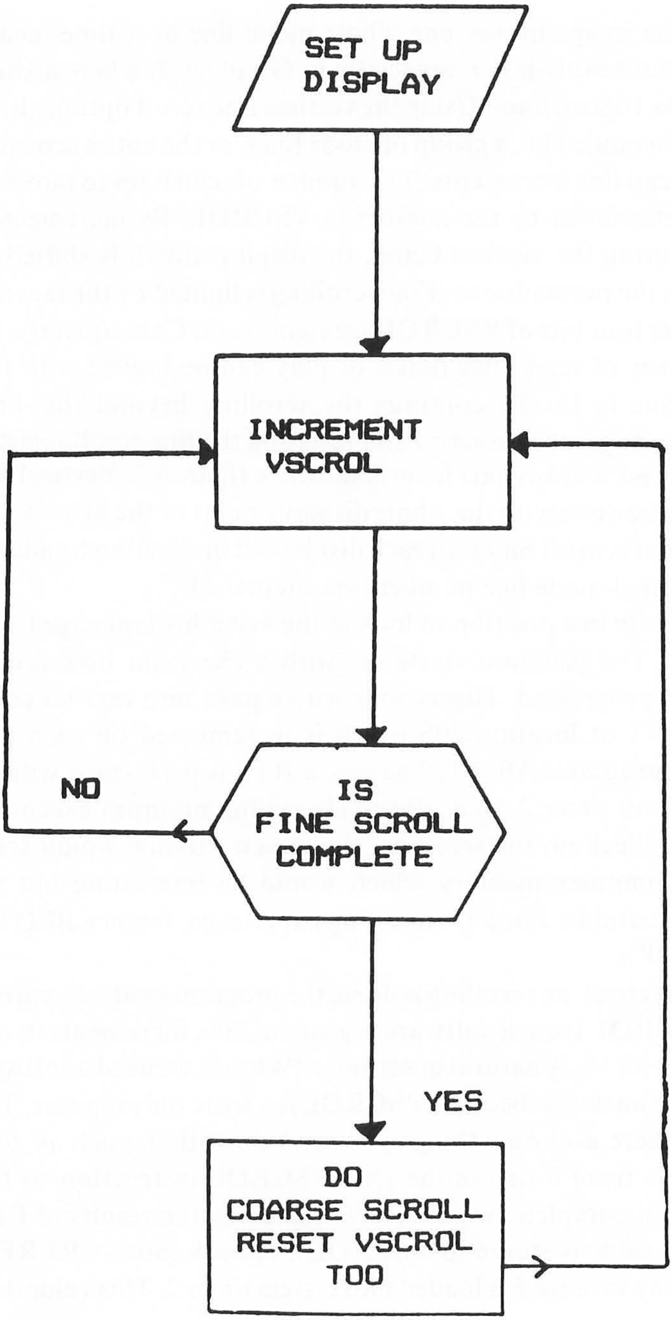


Figure 6-3. Vertical fine scrolling process

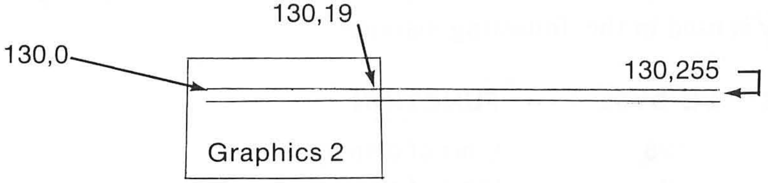
two things. It makes sure the subroutine will be called again when timer 2 counts down to 0 and it controls the speed of scrolling.

When SCRLREG increments to 16, it is necessary to shift the LMS instruction to point to the next lower screen mode line by adding 20 to the low byte of the LMS address. The program section beginning with the label COARSE does a binary add and resets VSCROL and SCRLREG. A new wrinkle in this addition routine is that sometimes we must perform two byte addition. That happens when adding 20 to SCNLO ;the Lo-Byte in the LMS address gives a result greater than 255. The procedure for doing this should be evident from the assembly listing.

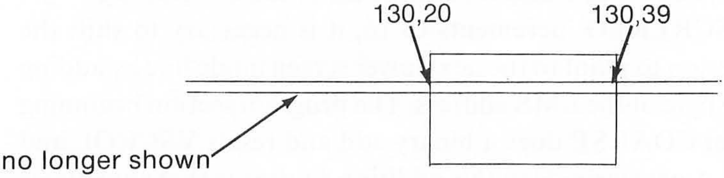
Before going further, test your understanding of the concepts covered by rewriting the program so that the submarine scrolls down the screen. To do this you will want to decrement the value in VSCROL. Since VSCROL does not use the upper four bits, one way to proceed is to load both hardware and software registers with 255 and decrement down until bits D_0—D_3 are clear. Now since you are scrolling down, you must subtract 20 from the Lo-Byte of the screen address. Don't forget it may be necessary to borrow from the Hi-Byte.

Horizontal Scrolling

The idea behind horizontal scrolling is to allocate a portion of screen memory for each mode line. If the memory allocated is larger than necessary, and if each mode line has the LMS option, then by changing the address in the instruction, the display can be moved left or right. This concept is illustrated with a single mode line thusly:



Change the Lo-Byte of the LMS from 0 to 20 and the display shifts:



The easiest way to organize memory for horizontal scrolling is to set aside one page for each mode line. Then you need only increment or decrement the Lo-Byte of the LMS instruction. Incrementing scrolls the display to the left. Decrementing scrolls the display to the right. The diagram above shows the effect of adding 20 bytes to the address, which is coarse scrolling in a big way! Normal coarse scrolling increments by one byte at a time. To smooth out the motion, the horizontal equivalent of a vertical fine scroll must be done. Horizontal fine scrolling is done in color clock units. The fine scrolling register HSCROL is nominally 8 bits wide but only the lower four bits are used. Thus, the maximum number of color clocks that an image can be fine scrolled before resetting is 16. The actual number to use depends in the number of color clocks in the characters of the graphics mode being used.

Box 43 is a horizontal scrolling example using the submarine of the previous program. This program and our succeeding example, in box 44, are similar, in order to reduce the amount of typing you need to do. In both programs the machine language routine is stored in a string for reasons of efficiency and economy. To make space for the display list, character set, and screen memory, RAMTOP is lowered 27 pages. Memory is used in the following manner:

PAGE	LO-BYTE	FUNCTION
133	128	Start of display list
134	0	Start of character set
135	0	Start of screen memory

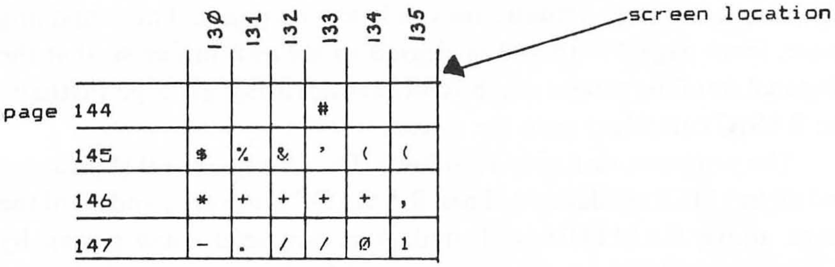
The screen memory actually uses only eleven pages. The remaining space, from page 149 to 160 is cleared to act as a buffer so that the diagonal scrolling program in box 44 does not bring 'garbage' (actually the BASIC cartridge) onto the screen.

The program outline is as follows. The strings are DIMensioned and all but ML$ are defined. Then RAMTOP is lowered and all of the space above RAMTOP is cleared. You can save some typing by starting out with the previous program and modifying or changing the appropriate lines. After the character set has been moved and redefined, a custom display list is constructed. The display list,

```
  112
  112
  112
  LMS
ADDRLO
ADDRHI
  LMS
ADDRLO
ADDRHI
   ●
   ●
   ●
```

has an LMS instruction at each mode line. The LMS instruction opcode is 87 (64+16+7) which sets bits D_6, D_4 and selects Graphics 2. The Lo-Byte of each instruction specifies that the memory scan counter starts at the half way point of each page. This allows the option of scrolling either way - left or right.

With a custom display list such as this, it is much easier to POKE the character codes for each of the redefined characters directly into memory. The easiest way to figure out where to POKE the numbers is to make a sketch such as:



		130	131	132	133	134	135
page	144				#		
	145	$	%	&	'	(	(
	146	*	+	+	+	+	,
	147	-	.	/	/	0	1

Lines 330-400 put the display in memory. Lines 420-450 tell ANTIC where to find the display list and the OS where to find the character set. Finally, lines 460-560 define the scrolling routine and start things going.

The assembly listing of the scrolling routine is in box 43A. The listing is fully documented. However, a couple of comments are in order. As in the previous program, a count value is used to keep from scrolling beyond the page boundary. There are, of course, other ways to do this check. One way is to load in a typical LMS address and test to see if it is within bounds. Also note that there is a curious asymmetry between what we do to the fine scroll register and the address bytes. To scroll right, HSCROL is incremented and the address is decremented. To scroll left, HSCROL is decremented and the address is incremented.

Once again, it would be a good idea for you to rewrite the program so that the submarine scrolls to the left across the screen. Never mind that submarines probably don't back up too well! Start your changes by repositioning the submarine in screen memory and decrement the screen address bytes. This time however, decrement the fine scrolling number down from 15 and make use of the fact that the Z flag is set when DEX results in a zero in the X-register. That will shorten the routine so watch your branch!

BOX 43
Horizontal Scrolling
The Yellow Submarine

```
5 REM ** YELLOW SUBMARINE SCROLL **
10 REM * DIMENSION STRINGS THAT STORE ML ROUTINES *
20 REM * AND CHARACTER SET *
30 DIM CLEAR$(18),MOV$(20),
REDEF$(14),SUB$(120),S$(32),ML$(52)
40 REM CLEAR$
50 REM MOV$
60 REM REDEF$
70 REM SUB$
80 REM S$ (the remainder of SUB$)
85 SUB$(LEN(SUB$)+1)=S$
90 REM * SET UP RESERVED SPACE AND CLEAR *
100 POKE 106,133:POKE 203,0:POKE 204,133
110 CLEAR=USR(ADR(CLEAR$))
120 REM * SET GRAPHICS MODE AND COLORS *
130 GRAPHICS 18:POKE 752,1:POKE 708,60:POKE 712,134
140 REM * MOVE STANDARD CHARACTERS/REDEFINE *
150 POKE 205,0:POKE 206,224
160 MOVE=USR(ADR(MOV$))
170 Q=ADR(SUB$)
180 HIQ=INT(Q/256)
190 LOQ=Q-HIQ*256
200 POKE 205,LOQ:POKE 206,HIQ
210 POKE 203,24:POKE 204,134
220 R=USR(ADR(REDEF$))
230 REM * SET UP CUSTOM DISPLAY LIST *
240 FOR I=0 TO 2:POKE 34176+I,112:NEXT I
250 FOR I=0 TO 10:POKE 34179+I*3,87:NEXT I
260 FOR I=0 TO 10:POKE 34180+I*3,128:NEXT I
270 FOR I=0 TO 10:POKE 34181+I*3,138+I:NEXT I
280 POKE 34212,65
290 POKE 34213,128
300 POKE 34214,143
```

continued on next page

```
310 REM * TELL ANTIC AND OS WHERE SCREEN MEMORY IS *
320 REM * PUT SUBMARINE IN MEMORY *
330 REM * POKE INTERNAL CHAR NUMBERS DIRECTLY IN MEMORY *
340 POKE 144*256+133,3
350 FOR I=1 TO 6:POKE 145*256+(129+I),(I+3):NEXT I
360 POKE 146*256+130,10
370 FOR I=1 TO 4:POKE 146*256+(130+I),11:NEXT I
380 POKE 146*256+135,12
390 FOR I=0 TO 2:POKE 147*256+(130+I),13+I:NEXT I
400 FOR I=0 TO 2:POKE 147*256+(133+I).15+I:NEXT I
410 REM * CHANGE CHARBAS *
420 POKE 559,0
430 POKE 560,128:POKE 561,133
440 POKE 756,134
450 POKE 559,34
460 ML$
470 REM * INSTALL ADDRESS OF THE SCROLLING ROUTINE *
480 Q=ADR(ML$)
490 HIQ=INT(Q/256)
500 LOQ=Q-HIQ*256
510 POKE 552,LOQ:POKE 553,HIQ
520 REM * SET REGISTERS USED BY SCROLLING ROUTINE *
530 POKE 205,0:POKE 206,0:POKE 54276,0
540 REM * START SYSTEM TIMER 2 *
550 POKE 538,10
560 GOTO 560
```

Box 43. Horizontal scrolling The Yellow Submarine

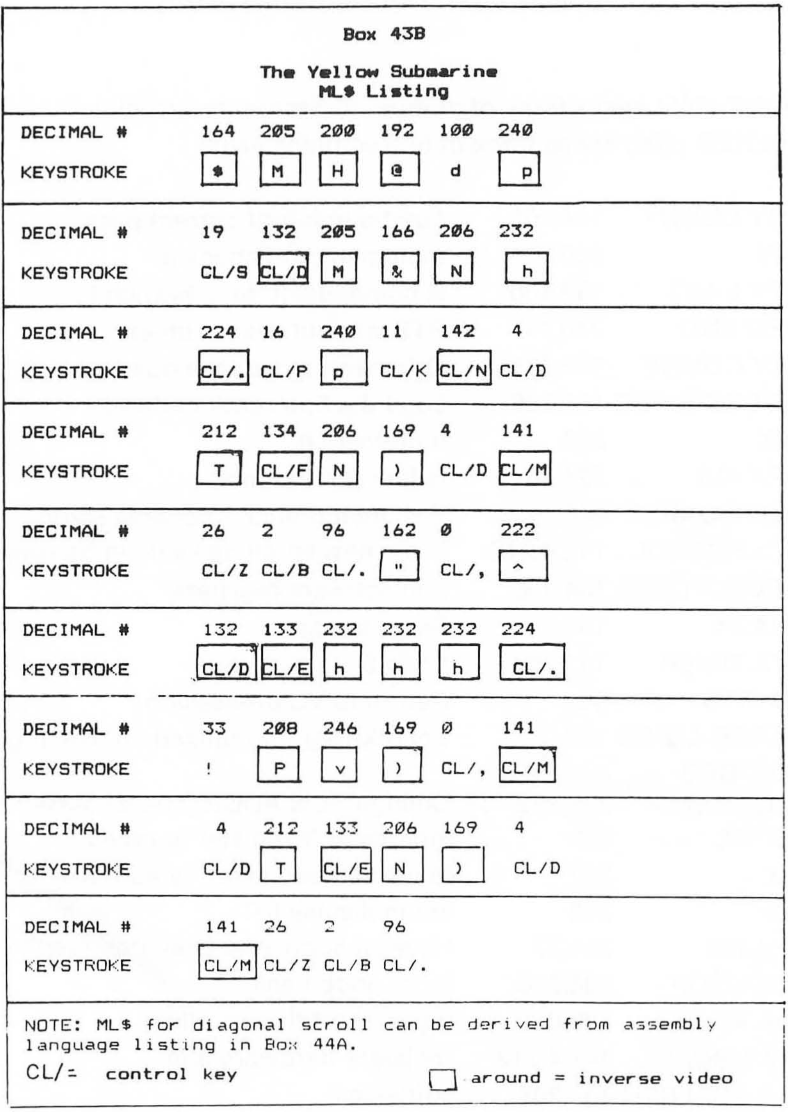
BOX 43A

The Yellow Submarine
Assembly Listing for Horizontal Scroll

COUNT (205) keeps track of number of passes thru routine.
SCRLREG (206) keeps track of fine scrolling value

```
  LDY COUNT     164,205      Load number of current pass
  INY           200          increment and check if
  CPY LIMIT     192,100      is complete. If done branch to
  BEQ END       240,19       RTS without setting timer
  STY COUNT     132,205      Otherwise store pass number
  LDX SCRLREG   166,206      Load the fine scroll number
  INX           232          Increment it
  CPX #16       224,16       Is fine scroll done?
  BEQ COARSE    240,11       Yes, then branch to coarse scroll
  STX HSCROL    142,4,212    Store new scrolling value in hardware
  STX SCRLREG   134,206      and software registers
  LDA #4        169,4        Reset system
  STA TIMER     141,26,2     timer 2
END  RTS        96           Return to VB processing
COARSE  LDX#0   162,0        Load X-Reg for indexed addressing
LOOP  DEC  Dec SCNMEM,X
~~LOSCNMEM,X~~   222,132,133  Change SCN ADDR (coarse scroll)
  INX ~~[illegible]~~  232          Increment X-register to point
  INX           232          to next screen memory address
  INX           232          in the display list
  CPX #33       224,33       Have all addresses been changed?
  BNE LOOP      208,246      If not, loop back
  LDA #0        169,0        If yes, reset fine scrolling
  STA HSCROL    141,4,212    registers-hardware and
  STA SCRLREG   133,206      software
  LDA #4        169,4        Reset system
  STA TIMER     141,26,2     timer 2
  RTS           96           Return to VB processing
```

Box 43A. The Yellow Submarine assembly listing for horizontal scroll



Box 43B

The Yellow Submarine
ML$ Listing

DECIMAL #	164	205	200	192	100	240
KEYSTROKE	$	M	H	@	d	p
DECIMAL #	19	132	205	166	206	232
KEYSTROKE	CL/S	CL/D	M	&	N	h
DECIMAL #	224	16	240	11	142	4
KEYSTROKE	CL/.	CL/P	p	CL/K	CL/N	CL/D
DECIMAL #	212	134	206	169	4	141
KEYSTROKE	T	CL/F	N	)	CL/D	CL/M
DECIMAL #	26	2	96	162	0	222
KEYSTROKE	CL/Z	CL/B	CL/.	"	CL/,	^
DECIMAL #	132	133	232	232	232	224
KEYSTROKE	CL/D	CL/E	h	h	h	CL/.
DECIMAL #	33	208	246	169	0	141
KEYSTROKE	!	P	v	)	CL/,	CL/M
DECIMAL #	4	212	133	206	169	4
KEYSTROKE	CL/D	T	CL/E	N	)	CL/D
DECIMAL #	141	26	2	96		
KEYSTROKE	CL/M	CL/Z	CL/B	CL/.		

NOTE: ML$ for diagonal scroll can be derived from assembly language listing in Box 44A.

CL/= control key

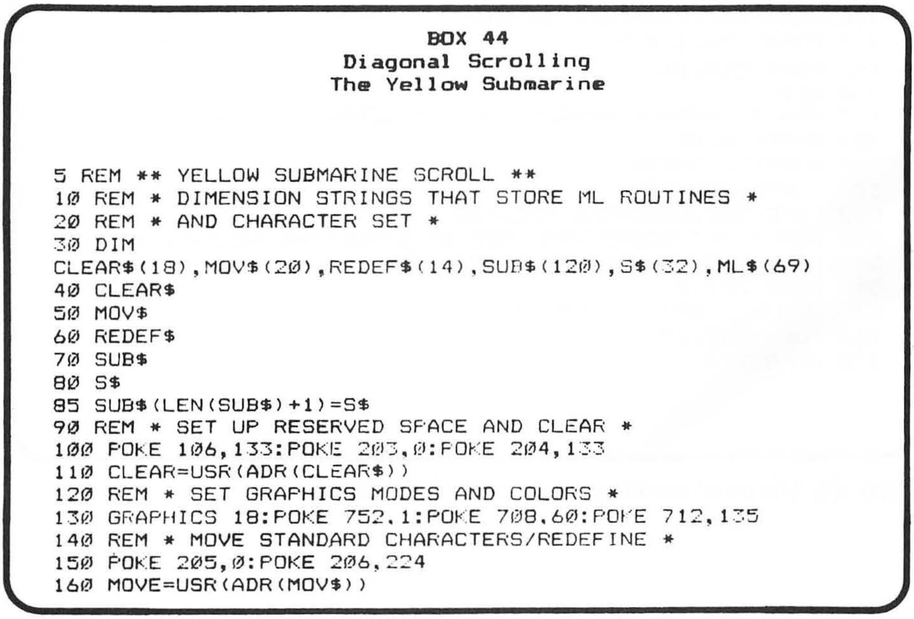
☐ around = inverse video

Box 43B. The Yellow Submarine ML$ listing

Diagonal Scrolling

The final scrolling example, in box 44, shows how to program a diagonal scroll. Except for the fact that both fine scrolling bits are set in the LMS instruction, all of the interesting differences between this program and the previous one occur in the scrolling routine which is written out in detail in box 44A.

Diagonal scrolling involves simultaneous manipulation of both fine scroll registers. Now, because we are using Graphics 2, the vertical fine scroll register goes from 0 to 15, while HSCROL is incremented from 0 to 7. To keep things simple, we increment VSCROL twice for each increment of HSCROL. Consequently, both registers reach their limits together and by testing only HSCROL it is possible to choose whether or not to branch to the coarse scroll segment. The vertical coarse scroll differs from the example in box 42 because screen memory is organized in one page per mode line. Here it is only necessary to increment the address Hi-Byte to scroll one line.



BOX 44
Diagonal Scrolling
The Yellow Submarine

```
5 REM ** YELLOW SUBMARINE SCROLL **
10 REM * DIMENSION STRINGS THAT STORE ML ROUTINES *
20 REM * AND CHARACTER SET *
30 DIM
CLEAR$(18),MOV$(20),REDEF$(14),SUB$(120),S$(32),ML$(69)
40 CLEAR$
50 MOV$
60 REDEF$
70 SUB$
80 S$
85 SUB$(LEN(SUB$)+1)=S$
90 REM * SET UP RESERVED SPACE AND CLEAR *
100 POKE 106,133:POKE 203,0:POKE 204,133
110 CLEAR=USR(ADR(CLEAR$))
120 REM * SET GRAPHICS MODES AND COLORS *
130 GRAPHICS 18:POKE 752.1:POKE 708.60:POKE 712,135
140 REM * MOVE STANDARD CHARACTERS/REDEFINE *
150 POKE 205,0:POKE 206.224
160 MOVE=USR(ADR(MOV$))
```

(cont. on next page)

```
170 Q=ADR(SUB$)
180 HIQ=INT(Q/256)
190 LOQ=Q-HIQ*256
200 POKE 205,LOQ:POKE 206,HIQ
210 POKE 203,24:POKE 204,134
220 R=USR(ADR(REDEF$))
230 REM * SET UP CUSTOM DISPLAY LIST *
240 FOR I=0 TO 2:POKE 34176+I,112:NEXT I
250 FOR I=0 TO 10:POKE 34179+I*3,119:NEXT I
260 FOR I=0 TO 10:POKE 34180+I*3,128:NEXT I
270 FOR I=0 TO 10:POKE 34181+I*3,138+I:NEXT I
280 POKE 34212,65
290 POKE 34213,128
300 POKE 34214,133
310 REM * TELL ANTIC AND OS WHERE SCREEN MEMORY IS *
320 REM * PUT SUBMARINE IN MEMORY *
330 REM * POKE INTERNAL CHAR NUMBERS DIRECTLY IN MEMORY *
340 POKE 144*256+133,3
350 FOR I=1 TO 6:POKE 145*256+(129+I),(I+3):NEXT I
360 POKE 146*256+130,10
370 FOR I=1 TO 4:POKE 146*256+(130+I),11:NEXT I
380 POKE 146*256+135,12
390 FOR I=0 TO 2:POKE 147*256+(130+I),13+I:NEXT I
400 FOR I=0 TO 2:POKE 147*256+(133+I),15+I:NEXT I
410 REM * CHANGE CHARBAS *
420 POKE 559,0
430 POKE 560,128:POKE 561,133
440 POKE 756,134
450 POKE 559,34
460 ML$
470 REM * INSTALL ADDRESS OF THE SCROLLING ROUTINE *
480 Q=ADR(ML$)
490 HIQ=INT(Q/256)
500 LOQ=Q-HIQ*256
510 POKE 552,LOQ:POKE 553,HIQ
520 REM * SET REGISTERS USED BY SCROLLING ROUTINE *
530 POKE 205,0:POKE 206,0:POKE 54276,0
540 POKE 207,0
550 REM * START SYSTEM TIMER 2 *
560 POKE 538,10
570 GOTO 570
```

Box 44. Diagonal scrolling The Yellow Submarine

BOX 44A

The Yellow Submarine
Assembly Listing for Diagonal Scroll

COUNT (205) keeps track of number of passes thru routine
HSREG (206) keeps track of horizontal fine scrolling value
VSREG (207) keeps track of vertical fine scrolling value

```
LDY COUNT     164,205     Load number of current pass
INY           200         increment and check if
CPY LIMIT     192,80      is complete. If done branch to
BEQ END       240,28      RTS without setting timer
STY COUNT     132,205     Otherwise store pass number
LDX HSREG     166,206     Load the fine scroll number
INX           232         Increment it
CPX #8        224,8       Is fine scroll done?
BEQ COARSE    240,20      Yes, then branch to coarse scroll
STX HSREG     134,206     No store horizontal scroll in hardware
STX HSCROL    142,4,212   and software registers
LDX VSREG     166,207     Load vertical fine scroll number
INX           232         Increment it twice (fine scroll
```

(cont. on next page)

```
JNX                 232            In 2 scan line increments)
STX VSREG           134,207        Store value in software and
STX VSCROL          142,5,212      hardware registers
LDA #6              169,6          Reset system
STA TIMER           141,26,2       timer 2
RTS                 96             Return to VB Processing
COARSE   LDX#0      162,0          Load X-reg for indexed addressing
LOOP   DEC
LOSCNMEM,X          222,132,133Change Lo-Byte of screen address
INX                 232            Inc X-reg to point to Hi-Byte of
                                   Screen Address
INC HISCNMEM,X254,132,133Increment Hi-Byte of screen address
INX                 232            Increment X-reg to point to next
INX                 232            Lo-Byte of the screen address
CPX#33              224,33         Are all screen addresses changed?
BNE LOOP            208,243        No? Then branch back to do next one
LDA #0              169,0          If yes, reset
STA HSREG           133,206        all software
STA VSREG           133,207        and
STA HSCROL          141,4,212      hardware
STA VSCROL          141,5,212      registers
LDA #6              169,6          Reset system
STA TIMER           141,26,2       timer 2
RTS                 96             Return to VB processing
```

Box 44A. The Yellow Submarine assembly listing for diagonal scroll

Vertical Blank Music

So far, all of the examples have used system timer 2 to link the subroutines into the vertical blank. This leaves the vector at 548,549 free to link in another program. Since the vertical blank is a fine place to play music, let's add the appropriate song to our previous examples! Up to this point the program in box 45 is the most ambitious program that we have presented. It scrolls the submarine around the screen while playing - you guessed it - The Yellow Submarine! Again, the program is structurally similar to the previous yellow submarine programs, so that your typing chore will be reduced. However, there are some changes. First, the scrolling routine is no longer in a string. Because of its length, it is read in from data numbers. So eliminate ML$ in line 30, but add VB$(11) in its place. VB$ is the short routine that links the music program to the regular VB processing by 'stealing' the vector at 548,549. In line 100, RAMTOP is lowered to page 130 to allow room for the scrolling routine and the music data numbers. In line 130, Graphics 18 is changed to Graphics 2 so we can print a message in the text window. Since ML$ is gone, lines 420 through 570 have been changed. These are the major changes. However, be sure to compare this program carefully with the previous one before you start typing.

Now that we have discussed the changes, let's see how the program is organized and how it works. Memory use is as follows:

- RAMTOP is repositioned to page 130.
- A one page buffer exists between RAMTOP and the scrolling routine, which starts at page 131.
- Since the scrolling routine is 241 bytes long, it takes up most of page 131.
- The music data is stored on page 132. The music program is only 93 bytes and fits into the first half of page 133.
- As before, the display list is in the second half of page 133.

When you plan out programs that are going to run during the vertical blank, one important thing to remember is to make sure that the machine language routines are in place before they are called up by the BASIC program. Otherwise the computer will lock up.

The program in box 45 makes sure everything is in place by going to a loading subroutine at line 420, right after the submarine has been put in memory. Since this is a rather lengthy process, a message is printed in the text window to let the user know what's happening. After the machine language routines are in place, the registers they use are set, the system timer is started, the music routine is linked, and the action begins. Speaking of linking a subroutine to the VB, here is the routine again: LDY with the Lo-Byte of the routine's address, LDX with the Hi-Byte, LDA with a 6 for an immediate, or with a 7 for a deferred vertical blank routine. Then JSR SETVBV. SETBV is at 92,228 (Lo-Byte, Hi-Byte).

In keeping with the format of the book, the entire assembly listing of the scrolling routine is in box 45A and the assembly listing of the music routine is in box 45B. However, because of the length and complex nature of the programs we offer the following explanation. Basically the scrolling consists of four sections:

- A vertical scroll up
- A horizontal scroll to the right
- A vertical scroll down and
- A horizontal scroll to the left.

The first thing the program must do is to decide in which direction to move. This is the purpose of section 1 of the program. Associated with each section is a COUNT register that keeps track of how far the scrolling has progressed. The program tests each of these registers in turn and will branch to the appropriate scrolling routine when it finds a nonzero value. There is an interesting wrinkle at the end of section 1. Because of the length of the program, it is not possible to rely on relative branching alone to go from the last COUNT test in section 1 to the horizontal scroll-left (HORZLFT)in section 6. Recall that one can branch forward only 127 bytes and backward 128 bytes. Consequently, we branch to a JMP instruction that's conveniently tucked in after the end of the vertical up (VERTUP) routine. The JMP then sends the program to (HORZLFT).

Look at Sections 3,4,and 5 (VERTUP,HORZRT,VERTDN). The first thing each of these segments do is to set up the COUNT and fine scroll registers for the segment following it. That way when their COUNT register is finally decremented to zero, the next one will be ready to go. However, the very last segment, horizontal left (HORZLFT), can't do this. Why? Because the horizontal left routine is executed 100 times before it gets down to zero, and after the first time through, COUNT1 would be set. The program would read that, branch to VERTUP, and never go back to HORZLFT. For every vertical scroll up after the first, VERTUP is executed as the default routine when section 1 finds only zero COUNT registers. The register values needed by VERTUP are provided in section 2.

Notice that the values stored in the COUNT registers for VERTUP and VERTDN are different. Also, the initial values of the software scrolling register, VSCLREG, are different in each case. The reason for this is that it is necessary to use these numbers in order to make sure that the scrolling rectangle closes and the submarine doesn't slowly sink into the mud at the bottom of the sea.

Other than this, the scrolling routines are similar to the ones in the previous programs. First, the program does a fine scroll until the limit has been reached. Then it branches to a coarse scroll. As in diagonal scrolling, in this program a coarse vertical scroll is done by incrementing, or decrementing the Hi-Byte of the screen address in the LMS instruction. This reflects the allocation of one page of memory per scan line. HORZRT is the same as before and HORZLFT is the answer to the exercise we posed earlier.

The music routine, in Box 45B has three parts:

- one which stores music frequency numbers in AUDF1 and AUDF2 (8 bit music)
- one which turns the notes off so they do not run together and
- a section at the beginning which chooses between the previous two.

The beginning section also controls the timing register at 1536. In order to keep track of whether we're on a music cycle or a silent cycle, there is a flag register called CYCLE at 1537. The program begins by loading in the current timer value and decrementing it. Then it tests to see if result is zero. If it is, the end of the current cycle has been reached and then the program branches to see which cycle comes next. If the cycle flag is 1, a new pair of frequencies is loaded in. If the cycle flag is zero the sound is turned off. Each segment of the program sets the flag to identify the next segment of the routine.

BOX 45
Finale
Scrolling and Music

```
2 REM ** FINALE PROGRAM **
5 REM ** YELLOW SUBMARINE SCROLL AND MUSIC **
10 REM * DIMENSION STRINGS THAT STORE ML ROUTINES *
20 REM * AND CHARACTER SET *
30 DIM CLEAR$(18),MOV$(20),
REDEF$(14),SUB$(120),S$(32),VB$(11)
40 CLEAR$
50 MOV$
60 REDEF$
70 SUB$
80 S$
85 SUB$(LEN(SUB$)+1)=S$
90 REM * SET UP RESERVED SPACE AND CLEAR *
100 POKE 106,130:POKE 203,0:POKE 204,130
110 CLEAR=USR(ADR(CLEAR$))
120 REM * SET GRAPHICS MODES AND COLORS *
130 GRAPHICS 2:POKE 752,1:POKE 708,60:POKE 712,134
140 REM * MOVE STANDARD CHARACTERS/REDEFINE *
150 POKE 205,0:POKE 206,224
160 MOVE=USR(ADR(MOV$))
170 Q=ADR(SUB$)
180 HIQ=INT(Q/256)
190 LOQ=Q-HIQ*256
200 POKE 205,LOQ:POKE 206,HIQ
210 POKE 203,24:POKE 204,134
220 R=USR(ADR(REDEF$))
230 REM * SET UP CUSTOM DISPLAY LIST *
240 FOR I=0 TO 2:POKE 34176+I,112:NEXT I
250 FOR I=0 TO 10:POKE 34179+I*3,119:NEXT I
260 FOR I=0 TO 10:POKE 34180+I*3.020001,128:NEXT I
270 FOR I=0 TO 10:POKE 34181+I*3,138+I:NEXT I
280 POKE 34212,65
290 POKE 34213,128
300 POKE 34214,133
320 REM * PUT SUBMARINE IN MEMORY *
330 REM * POKE INTERNAL CHAR NUMBERS DIRECTLY IN MEMORY *
340 POKE 144*256+133,3
350 FOR I=1 TO 6:POKE 145*256+(129+I),(I+3):NEXT I
360 POKE 146*256+130,10
370 FOR I=1 TO 4:POKE 146*256+(130+I),11:NEXT I
380 POKE 146*256+135,12
390 FOR I=0 TO 2:POKE 147*256+(130+I),13+I:NEXT I
400 FOR I=0 TO 2:POKE 147*256+(133+I),15+I:NEXT I
410 PRINT "LOADING MACHINE LANGUAGE ROUTINES AND MUSIC"
420 GOSUB 700
430 VB$
440 REM * CHANGE CHARBAS *
450 POKE 559,0
```

(cont. on next page)

```
460 POKE 560,128:POKE 561,133
470 POKE 756,134:POKE 559,34
480 REM * INSTALL ADDRESSES OF VERTICAL BLANK ROUTINES *
490 POKE 552,0:POKE 553,131
500 REM * SET REGISTERS USER BY ROUTINES *
510 POKE 209,1:POKE 1536,1:POKE 1537,0
520 POKE 203,80:POKE 204,0:POKE 205,0
530 POKE 206,0:POKE 207,0:POKE 208,0
550 REM * START TIMER AND MUSIC REGISTERS *
560 POKE 538,10
570 POKE 53768,0:POKE 53775,3
580 POKE 53761,168:POKE 53763,168:POKE 53765,168
590 X=USR(ADR(VB$))
600 GOTO 600
700 REM READ IN MACHINE LANGUAGE ROUTINE
710 FOR I=0 TO 92:READ MUS
720 POKE 34048+I,MUS:NEXT I
725 REM
730 DATA 174,0,6,202,142,0,6,224,
0,240,3,76,98,228,166,209,224,1,240,20,169,0,141,0,210,141,2,
210
735 REM
740 DATA 169,2,141,0,6,169,1,133,
209,76,98.228,174,1,6,189,0,132,141,0,210,232,189,0,132,141,2
,210
745 REM
750 DATA 232,189,0,132,141,0,6,
232,224,186,240,10,142,1,6,169,0,133,209,76,98,228,169,0,133,
209,141,1,6
755 REM
760 DATA 169,20,141,0,6,76,98,228
765 REM
770 REM READ IN MUSIC DATA
780 FOR I=0 TO 185:READ MUSDAT
790 POKE 33792+I,MUSDAT:NEXT I
795 REM
800 DATA 63,53,22,71,53,22,80,53,
22,5,53,30,47,0,12,85,71,30,85,71,8,85,71,30,85,71,8,85,71.50
,85,71,30
805 REM
810 DATA 85,71,8,85,71,30,85,71,8,
85,71,50,107,80,30,107,80,8,107,80,30,107,80,8,107,80,50,63,5
3,22
815 REM
820 DATA 71,53,22,80,53,22,85,53,
30,47,0,12,85,71,30,85,71,8,85,71,30,85,71,8,85,71,50,85,71,3
0,85,71,8
825 REM
830 DATA 85,71,30,85,71,8,85,71,
50,107,80,30,107,80,8,107,80,30,107,80,8,107,80,22,0,63,30,60
,0,8
835 REM
840 DATA 0,53,64,63,0,8,0,71,30,
63,0,8,0,80,64,63,80,30,0,63,8,71,121,30,0,80,8,95,121,48,80,
```

(cont. on next page)

```
63,30
845 REM
850 DATA 0,63,8,71,0,64,0,63,30,
60,0,8,0,53,64,63,0,12,0,71,30,63,0,8,0,80,72
860 REM
870 REM READ IN SCROLLING ROUTINE
880 FOR I=0 TO 240:READ ML
890 POKE 33536+I,ML:NEXT I
895 REM
900 DATA 164,203,192,0,208,24,
164,204,192,0,208,75,164,205,192,0,208,123,164,206,192,0,208,
60
905 REM
910 DATA 169,8,133,207,160,90,
169,100,133,204,169,4,133,208,136,132,203,166,207,232,224,16,
240,11
915 REM
920 DATA 142,5,212,134,207,169,6,
141,26,2,96,162,0,254,133,133,232,232,232,224,33,208,246,169,
0,133,207
925 REM
930 DATA 141,5,212,169,6,141,26,
2,96,76,195,131,169,85,133,205,169,2,133,207,136,132,204,166,
208,232,224
935 REM
940 DATA 8,240,11,142,4,212,134,
208,169,6,141,26,2,96,162,0,222,132,133,232,232,232,224,33,20
8,246,169,0
945 REM
950 DATA 141,4,212,133,208,169,6,
141,26,2,96,169,100,133,206,169,1,133,208,136,132,205,166,207
,202,224,0
955 REM
960 DATA 240,11,142,5,212,134,
207,169,6,141,26,2,96,162,0,222,133,133,232,232,232,224,33,20
8,246,169,15,133
965 REM
970 DATA 207,141,5,212,169,6,141,
26,2,96,136,132,206,166,208,202,224,0,240,11,142,4,212,134,20
8,169,6,141
975 REM
980 DATA 26,2,96,162,0,254,132,
133,232,232,232,224,33,208,246,169,8,141,4,212,133,208,169,6,
141,26,2,96
985 REM
990 RETURN
```

Box 45. Finale Scrolling and Music

BOX 45A

Assembly Language Listing for Yellow Submarine Scrolling

Register use:

COUNT1 = 203
COUNT2 = 204
COUNT3 = 205
COUNT4 = 206
VSCLREG = 207
HSCLREG = 208

SECTION 1: *Determine which move to make*

```
          LDY COUNT1            164,203
          CPY #0                192,0
          BNE VERTUP            208,24
          LDY COUNT2            164,204
          CPY #0                192,0
          BNE HORZRT            208,75
          LDY COUNT3            164,205
          CPY #0                192,0
          BNE VERTDN            208,123
          LDY COUNT 4           164,206
          CPY #0                192,0
          BNE HORZLFT           208,60
```

BOX 45A. Assembly language listing for Yellow Submarine Scrolling

SECTION 2: *Initialize counters for vertical up move* (ie next routine)

```
          LDA #8                    169,8
          STA VSCLREG               133,207
          LDY #90                   169,90
```

SECTION 3: *Vertical Scroll Up*

```
VERTUP        LDA #100              169,100
              STA COUNT2            133,204
              LDA #4                169,4
              STA HSCLREG           133,208
              DEY                   136
              STY COUNT1            132,203
              LDX VSCLREG           166,207
              INX                   232
              CPX #16               224,16
              BEQ COARSE1           240,11
              STX VSCROL            142,5,212
              STX VSCRLREG          135,207
              LDA #6                169,6
              STA TIMER             141,26,2
              RTS                   96
COARSE1       LDX #0                162,0
LOOP          INC HISCN,X           254,133,135
              INX                   232
              INX                   232
              INX                   232
              CPX #33               224,33
              BNE LOOP              208,246
              LDA #0                169,0
```

(cont. on next page)

```
            STA VSCLREG          133,207
            STA VSCROL           141,5,212
            LDA #6               169,6
            STA TIMER            141,26,2
            RTS                  96
            JMP HORZLFT          76,195,131
```

SECTION 4: *Horizontal Scroll to the Right*

```
HORZRT      LDA#85               169,85
            STA COUNT3           133,205
            LDA #15              169,2
            STA VSCRLREG         133,207
            DEY                  136
            STY COUNT2           132,204
            LDX HSCLREG          166,208
            INX                  232
            CPX #8               224,8
            BEQ COARSE2          240,11
            STX HSCROL           142,4,212
            STX HSCLREG          134,208
            LDA #6               169,6
            STA TIMER            141,26,2
            RTS                  96

COARSE2     LDA #0               162,0
LOOP        DEC LOSCN,X          222,132,133
            INX                  232
            INX                  232
            INX                  232
            CPX #33              224,33
            BNE LOOP             208,246
```

(cont. on next page)

```
          LDA #0                169,0
          STA HSCROL            141,4,212
          STA HSCREG            133,208
          LDA #6                169,6
          STA TIMER             141,26,2
          RTS                   96
```

SECTION 5: *Vertical Down Scroll*

```
VERTDN      LDA #100            169,100
            STA COUNT4          133,206
            LDA #1              169,1
            STA HSCLREG         133,208
            DEY                 136
            STY COUNT3          132,205
            LDX VCSCLREG        166,207
            DEX                 202
            CPX #0              224,0
            BEQ COARSE3         240,11
            STX VSCROL          142,5,212
            STX VSCLREG         135,207
            LDA #6              169,6
            STA TIMER           141,26,2
            RTS                 96
COARSE3     LDX #0              162,0
LOOP        DEC HISCN,X         222,133,133
            INX                 232
            INX                 232
            INX                 232
            CPX #33             224,33
            BNE LOOP            208,246
            LDA #15             169,15
```

(cont. on next page)

```
                 STA VSCLREG              133,207
                 STA VSCROL               141,5,212
                 LDA #6                   169,6
                 STA TIMER                141,26,2
                 RTS                      96
```

SECTION 6: *Horizontal Scroll Left*

```
HORZLFT          DEY                      136
                 STY COUNT4               132,206
                 LDX HSCLREG              166,208
                 DEX                      202
                 CPX #0                   224,0
                 BEQ COARSE4              240,11
                 STX HSCROL               142,4,212
                 STX HSCRLREG             134,208
                 LDA #6                   169,6
                 STA TIMER                141,26,2
                 RTS                      96
COARSE4          LDX #0                   162,0
LOOP             INC LOSCN,X              254,132,133
                 INX                      232
                 INX                      232
                 INX                      232
                 CPX #33                  224,33
                 BNE LOOP                 208,246
                 LDA #8                   169,8
                 STA HSCROL               141,4,212
                 STA HSCLREG              133,208
                 LDA #6                   169,6
                 STA TIMER                141,26,2
                 RTS                      96
```

(cont. on next page)

BOX 45B

Assenbly Language Listing
for
Yellow Submarine Music

```
Register use:            MUSTIMER = 1536
                         DATACNT = 1537
                         CYCLE = 209

Section 1: Timing and Cycle Test


       LDX MUSTIMR  174,0,6    Load duration of note or silence
       DEX          202        decrement and save. If done
       STX MUSTIMR  142,0,6    then go see what happens
       CPX #0       224,0      next
       BEQ CYTEST   240,3
       JMP XITVBL   76,98,228  If not done leave vertical blank

CYTESTLDX CYCLE     166,209    Test flag to see if we're on
       CPX #1       224,1      silence or sound
       BEQ SOUND    240,20
```

(cont. on next page)

Section 2: *Sound Off to Separate Notes*

```
        LDA #0          169,0      Turn off
        STA AUDF1       141,0,210  Sound
        STA AUDF2       141,2,210  registers
        LDA #2          169,2      Load duration of silence
        STA MUSTIMR     141,0,6    and store in timer
        LDA #1          169,1      Set cycle
        STA CYCLE       133,209    Flag
        JMP XITVBL      76,98,228  Leave vertical blank
```

Section 3: *Sound On*

```
SOUND   LDX DATACNT     174,1,6
        LDY TABLE,X     189,0,132  Load notes and
        STA AUDF1       141,0,210  duration from
        INX             232        the music data
        LDA TABLE,X     189,0,132  table. Store in sound
        STA AUDF2       141,2,210  Channels 1 and 2
        INX             232        and store duration in
        LDA TABLE,X     189,0,132  MUSTIMR
        STA MUSTIMR     141,0,6

        INX             232
        CPX TABLEND     224,186    Test for end of data table
        BEQ REPEAT      240,10
        STX DATACNT     142,1,6    Save last X-reg for next note
        LDA #0          169,0      Set flag for sound off
        STA CYCLE       132,209
        JMP XITVBL      76,98,228  Leave vertical blank

REPEAT  LDA #0          169,0
        STA CYCLE       133,209    Restore registers and
        STA DATACNT     141,1,6    set timer to hold
        LDA #20         169,20     the last note a little
        STA MUSTIMR     141,0,6    longer....
        JMP XITVBL      76,98,228
```

Box 45B. Assembly language listing for Yellow Submarine music

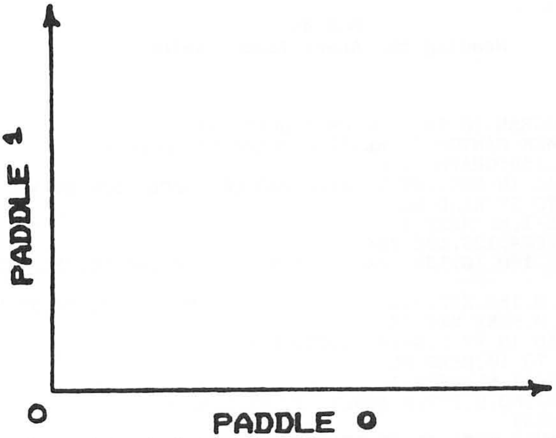
Alternate forms of Input/Output

There is an area of programming, about which very little has been written which has a lot of potential for new and innovative programming. This area is the use of alternate forms of input such as light pens, touch sensitive pads, or specialized switches designed for handicapped computer users. We will discuss touch sensitive pads because they are comparatively inexpensive, readily available, and quite versatile. When one thinks of a touch sensitive pad like the Atari Touch Tablet™ or Koala Pad™, one invariably thinks of drawing and artwork. But there is absolutely no reason that either of these pads be relegated to duty as a high-tech Etch-a-Sketch™. A touch pad can be just as useful for input as a mouse and uses a lot less desk space.

The typical touch pad can be connected into any of the joystick ports. As we shall see, which port you use depends on the program written for it. The pad has two or three switches and the touch sensitive surface. The touch sensitive surface works like a rectangular grid of variable resistors, usually referred to as potentiometers. In its action the pad mimics a pair of game paddles.

In the Atari Home Computer a pair of paddles connects into one game port: paddles 0 and 1 into port 1, paddles 2 and 3 into port 2, etc. A small current at +5 volts is sent from the computer through the potentiometer in the paddle. Turning the knob on the paddle varies the resistance and changes the voltage of the signal. The current returns to the computer at pins 5 and 9 of the joystick port. Here the signal is changed by an analog to digital converter into a number from 0 to 228. During each vertical blank, the OS reads the values from hardware registers 53760 - 53767 and stores the values in the shadow registers PADDL0 - PADDL7 at 624 - 631.

Now it is easy to see how a touch pad works. For example, suppose that the Atari Touch Tablet™ is plugged into game port 1 and you touch the stylus to the surface. The number read into PADDL0 (624) gives a measure of the horizontal distance from the left side of the tablet to the point of contact. The number in PADDL1 (625) gives a measure of the vertical distance from the bottom of the tablet to the point of contact.



Notice that the tablet is laid out the same as quadrant I of a normal X-Y coordinate system with the origin in the lower hand corner. This is different from how coordinates are laid out on the TV screen where the origin is at the upper left hand corner. The Koala Pad™, on the other hand uses the same layout as the screen display with the origin in the upper left hand corner and PADDL0 giving the X values and PADDL1 giving the Y values.

In order to make use of a touch pad (digitizer) for input you need to know how the PADDL0 and PADDL1 numbers are mapped on to the pad. The program in box 46 will help you to do this. Cut a piece of graph paper with¼inch squares the size of the sensitive surface and lay it on the pad. Plug your digitizer into Port 1. When run, the program reads PADDL0 and PADDL1 every 1/6th second during the vertical blank storing the values in page 158. It waits for a while at line 145 to allow the reading to be completed and then prints out the values on the screen. Reading the paddle values during the vertical blank is more accurate than doing it from BASIC. This is particularly true for the Atari Touch Tablet which has the quicker response and sensitivity of the two digitizers. By running the program several times and making contact with the pad at different places you can prepare a diagram with the paddle values obtained.

As an example of what can be done with a digitizer, Box 47 contains a program to play music by simply moving the stylus around the surface.

```
                    BOX 46
         Reading the Atari Touch Tablet


5 REM ** PROGRAM TO TEST TOUCH TABLETS **
10 REM * LOWER RAMTOP TO RESERVE SPACE FOR DATA *
20 POKE 106,158:GRAPHICS 0
30 REM * LOAD IN ROUTINE TO READ PADDL0/PADDL1 DURING VB *
40 FOR I=0 TO 37:READ ML
50 POKE 1536+I,ML:NEXT I
60 DATA 164,204,136,132,204,
192,0,208,26,160,10,132,204,166,203,224,254,240,16,232,173,11
2,2
70 DATA 157,0,158,232,173, 113,2,157,0,158,134.203.76,98,228
80 POKE 203,0:POKE 204,10
90 REM * LOAD IN VB LINKING ROUTINE *
100 FOR I=0 TO 10:READ ML
110 POKE 1664+I,ML:NEXT I
120 DATA 104,160,0,162,6,169,7, 32,92,228,96
130 X=USR(1664)
140 REM DELAY * DATA TO BE READ *
150 FOR I=0 TO 6000
160 NEXT I
170 REM * PRINT OUT PADDL0/PADDL1 DATA *
180 FOR I=1 TO 255
190 PRINT PEEK(40448+I);" ";
200 NEXT I
```

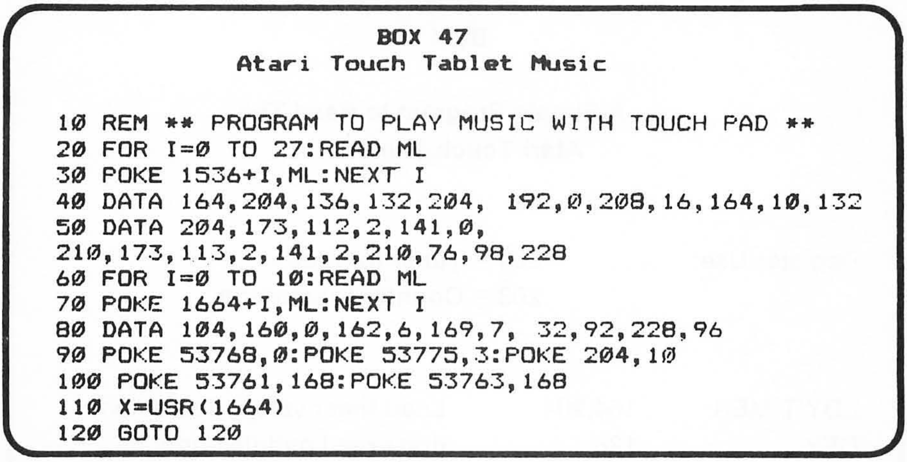
Box 46. Reading the Atari Touch Tablet

BOX 46A

A Simple Program to Read The Atari Touch Tablet

```
Register Use:           204 = Timer for read
                        203 = Counter for storage


LDY TIMER      164,204       Load timer value, decrement
DEY            136           don't read paddles until its
STY TIMER      132,204       down to zero. Thus, read
CPY #0         192,0         every 10th VB
BNE END        208,26
LDY TIMER      160,10        Reset
STA TIMER      132,204       Timer

LDX,COUNT      166,203       Load table offset called COUNT
CPX FINAL      224,254       Are we done?
BEQ END        240,16        If so, end

INX            232           Increment offset
LDA PADDL0     173,113,2     Get paddle 0 value
STA TABLE,X    157,0,158     Store in Table
INX            232           Increment offset
LDA PADDL1     173,112,2     Get paddle 1 value
STA TABLE,X    157,0,158     Store in table
STX,COUNT      134,203       Save Offset
JMP,XITVBL     76,98,228     Leave vertical blank processing
```

BOX 46A. A simple program to read the Atari Touch Tablet™



BOX 47
Atari Touch Tablet Music

```
10 REM ** PROGRAM TO PLAY MUSIC WITH TOUCH PAD **
20 FOR I=0 TO 27:READ ML
30 POKE 1536+I,ML:NEXT I
40 DATA 164,204,136,132,204, 192,0,208,16,164,10,132
50 DATA 204,173,112,2,141,0,
210,173,113,2,141,2,210,76,98,228
60 FOR I=0 TO 10:READ ML
70 POKE 1664+I,ML:NEXT I
80 DATA 104,160,0,162,6,169,7, 32,92,228,96
90 POKE 53768,0:POKE 53775,3:POKE 204,10
100 POKE 53761,168:POKE 53763,168
110 X=USR(1664)
120 GOTO 120
```

Box 47. Atari Touch Tablet Music

Once again the program reads PADDL0 and PADDL1 during the vertical blank. However, this time it stores the values in AUDF1 and AUDF2 which have been initialized for pure tones. This is a simple program which we have left for you to disassemble. It makes use of a timer register on page 0 (204) and reads the paddles every 10th vertical blank. As written, the program addresses the Atari Touch Tablet and needs to be modified for the Koala Pad by shortening the delay value used in the timer.

In addition to the touch sensitive surface which inputs paddle numbers, the digitizers have either two (Koala Pad) or three (Atari Touch Tablet) switches. The status of these switches can be monitored by reading the STICK or STRIG registers. Table 6-2 lists the ports to read and the values produced when each switch is closed. Additionally, the hardware and shadow registers for the STICK and STRIG ports are listed.

Table 6-2. Switch summary

	Kaola Pad	Value	Atari	Value
Left Switch	Stick(x)	11	Stick(x)	11
Right Switch	Stick(x)	7	Stick(x)	7
Probe Switch	--------	--	Stick(x)	14
Left switch and probe	--------	--	Stick(x)	10
Right switch and probe	--------	--	Stick(x)	6
Left and right switch	Stick(x)	3	Stick(x)	3

(x) = depends on the game port used

STRIG	Shadow Register	Hardware Register
STRIG(0)	644	53264
STRIG(1)	645	53265
STRIG(2)	646	53266
STRIG(3)	647	53267
STICK(0)	632	54016 Bits D_0 - D_3
STICK(1)	633	54016 Bits D_4 - D_7
STICK(2)	634	54017 Bits D_0 - D_3
STICK(3)	635	54018 Bits D_4 - D_7

When writing programs that use digitizer pads, there are several ways to regard them as input devices. One common way is to think of the pad surface as the primary input source, and use the switches to control program branching between different program segments that process this input. Another way to use them is to partition the surface into areas, each of which is associated with some program action. This is the sort of programming that is done when you use the stylus to pick an icon for paint or draw. Another way to look at them is to consider the geometry involved. The pad can be thought of as a grid of approximately 220 horizontal and 220 vertical lines. Where these lines meet defines a point. The total number of points is approximately 48400. Each point is labelled with two numbers and since order is important, each point can be distinguished from every other point. Thus, in principle a digitizer pad can be used to input 48400 separate pieces of information. Combine this with the switches and you can input up to 242,000 pieces of information. Granted this is a rather abstract way in which to regard the touch pads, but it is one with many possibilities that have not yet been explored in any depth.

Appendix A

6502 Microprocessor Instruction Set

ADC	Add memory to Accumulator with carry
AND	Logical "AND" of memory with Accumulator
ASL	Shift left one bit [Accumulator or memory]
BCC	Branch on carry clear
BCS	Branch on carry set
BEQ	Branch on result equal to zero
BIT	Test bits in memory with Accumulator
BMI	Branch on result minus
BNE	Branch on result not equal to zero
BPL	Branch on result plus
BRK	Force Break
BVC	Branch on overflow clear
BVS	Branch on overflow set
CLC	Clear the carry flag
CLD	Clear decimal mode
CLI	Clear the interrupt disable bit
CLV	Clear the overflow flag
CMP	Compare memory and Accumulator

CPX	Compare memory and X-Register
CPY	Compare memory and Y-Register
DEC	Decrement memory by one
DEX	Decrement X-Register by one
DEY	Decrement Y-Register by one
EOR	Logical "Exclusive-OR", memory with Accumulator
INC	Increment memory by one
INX	Increment X-Register by one
INY	Increment Y-Register by one
JMP	Jump to new location
JSR	Jump to subroutine
LDA	Load the Accumulator
LDX	Load the X-Register
LDY	Load the Y-Register
LSR	Shift right one bit [Accumulator or memory]
NOP	No operation
ORA	Logical "OR", Memory with Accumulator
PHA	Push Accumulator onto stack
PHP	Push Processor Status Register onto stack
PLA	Pull value from stack into Accumulator
PLP	Pull vlaue from stack into Processor Status
ROL	Rotate one bit left [Accumulator or Memory]
ROR	Rotate one bit right [Accumulator or Memory]
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract memory from Accumulator with borrow
SEC	Set carry flag
SED	Set decimal mode
SEI	Set interrupt disable
STA	Store Accumulator in memory
STX	Store X-Register in memory
STY	Store Y-Register in memory
TAX	Transfer Accumulator to X-Register
TAY	Transfer Accumulator to Y-Register
TSX	Transfer Stack Pointer to X-Register
TXA	Transfer X-Register to Accumulator
TXS	Transfer X-Register to Stack Pointer
TYA	Transfer Y-Register to Accumulator

Appendix B

Acknowledgement: We thank D. Kassabian for writing the disassembler and assembler programs that follow.

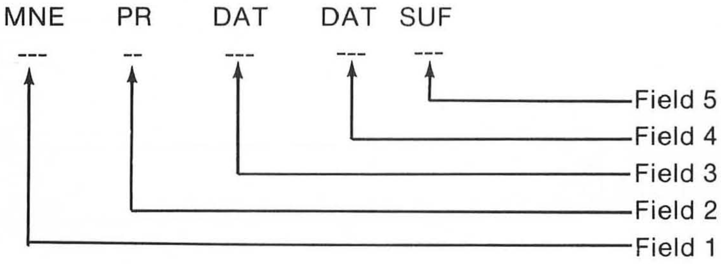
The assembler in Appendix B and the disassembler in Appendix C are included to make your machine language programming easier. They may be used as either stand alone utilities or in combination. Both programs are written in a straight forward manner so that you can modify them if you wish. They can be obtained by sending $20.00 to: Weber Systems Inc., Box 413, Gates Mills, OH 44040. Visa, Mastercard and American Express are accepted.

The assembler begins by asking for a starting address. The most common address to use is 1536 (page 6). However, for longer programs you might want to lower RAMTOP before RUNing the program by typing

```
POKE(106),PEEK(106)-8:GRAPHICS 0
```

and use pages 156 through 159 (39936-40960). Remember that you need a buffer above RAMTOP since each time the screen is cleared the OS clears beyond RAMTOP. The program will next give you the option to continue, quit, list or print. If you choose to continue, an input prompt is displayed on the screen. The input asked for is a string of characters organized into five fields: mnemonic, prefix, data, data, suffix. If the mnemonic doesn't require the full set of fields (for example, RTS), the program automatically pads out the input with blanks.

The prompt on the screen defines the fields as follows:



Spaces or any characters typed between the fields are ignored.

Once the input string is constructed it is broken down into its components. These components are used to select the proper OP CODE to go with the mnemonic. The program is relatively forgiving in that it allows you two chances to edit your input. The first is right after the initial input. The second opportunity to edit comes after the OP CODE is determined and a summary has been displayed on the screen. If the mnemonic entered does not correspond to a legal OP CODE, ??? is echoed back to the user and the input must be edited. Assembled code can be either listed on the screen (L) or printed out by the printer (P). This task can be done at any time and assembly continued afterwards.

This is a simple single pass assembler and no provision has been made for labels and relative branching. You can work around this by using the assembler in conjunction with the disassembler and doing the branching calculations yourself. The way to do this is to have both the assembler and disassembler on the same disk saved under convenient names such as ENCODE and DECODE respectively. Now suppose you are assembling some code and come to a branch instruction. Enter a 'dummy number' from 0 to 255 - one that you will easily recognize -for the branch data. List the assembled code on the screen and write down the address of the data number. Now continue on with assembly until you reach a convenient stopping point or have finished. Then "quit" the assembler and type RUN "D:DECODE". If you were careful and stored your assembled code in a protected area of memory such as page 6, the disassembler will load and run leaving your code intact. You may use the disassembler to print out the assembly listing from which you can quickly calculate the proper branching number. All that remains is to POKE the correct branching number into the address you wrote down. Just as you can run the disassembler from the assembler you can return to assembly by typing RUN "D:ENCODE". Just be sure to *keep track of what your starting address should be.*

Both the assembler and disassembler use standard notations to indicate the addressing modes. These are as follows:

MNEMONIC + no data = implied or accumulator addressing

MNEMONIC + 1 data number = 0 page or relative addressing

MNEMONIC + # data = immediate addressing

MNEMONIC + 2 data numbers = absolute addressing

MNEMONIC + 1 data,X = 0 page,X indexed addressing

MNEMONIC + 1 data,Y = 0 page,Y indexed addressing

MNEMONIC + 2 data nos,X = absolute,X indexed addressing

MNEMONIC + 2 data nos,Y = absolute,Y indexed addressing

MNEMONIC + (data,X) = indexed indirect addressing

MNEMONIC + (data),Y = indirect indexed

JSR + (   ) = absolute indirect

APPENDIX B
ASSEMBLER

```
1000 REM SIMPLE INTERPRETATIVE ASSEMBLER
1010 REM
1020 OPEN #1,4,0,"K:"
1030 OPEN #2,8,0,"S:"
1040 DIM PRE$(2),SUF$(3),CODE$(19),ML(128),A$(1)
1050 DIM S$(1),ANS$(1),MN$(3),DAT1$(3),DAT2$(3)
1070 PRINT :PRINT "ENTER STARTING ADDRESS"
1075 TRAP 1070
1080 INPUT ST
1090 PC=ST
1100 Z=0:CODE$="":DAT1=0:DAT2=0
1110 PRINT CHR$(125)
1120 PRINT "PRESS <RETURN> TO CONTINUE":PRINT "PRESS [Q] TO
QUIT":PRINT "PRESS [P] TO PRINT"
1125 PRINT "PRESS [L] TO LIST"
1130 INPUT A$
1140 IF A$="" THEN GOTO 1170
1150 IF A$="Q" THEN END
1160 IF A$="P" OR A$="L" THEN GOTO 1635
1165 GOTO 1120
1170 PRINT CHR$(125):POSITION 4,4:POKE 752,1
1180 PRINT "MNE PR DAT DAT SUF      PC=";PC
1190 POSITION 4,6:PRINT #2;"?"
1200 GET #1,S:S$=CHR$(S):Z=Z+1
1210 POSITION 3+Z,6:PRINT #2;S$
1220 CODE$(LEN(CODE$)+1)=S$
1230 IF Z=19 THEN GOTO 1290
1240 IF Z<19 AND S$=CHR$(155) THEN GOTO 1260
1250 POSITION 4+Z,6:PRINT #2;"?":GOTO 1200
1260 FOR I=LEN(CODE$) TO 18
1270 CODE$(I,I)=" "
1280 NEXT I
1290 POSITION 4,8:PRINT #2;"CHANGE ?? (Y/N)"
1300 INPUT ANS$
1310 IF ANS$="Y" THEN GOTO 1100
1320 IF ANS$="N" THEN GOTO 1340
1330 GOTO 1290
1340 MN$=CODE$(1,3):PRE$=CODE$(5,6)
1350 DAT1$=CODE$(8,10):DAT2$=CODE$(12,14)
1360 SUF$=CODE$(16,18)
1370 IF DAT1$="   " AND DAT2$="   " THEN DC=0:GOTO 1400
1380 IF DAT1$<>"   " AND DAT2$="   " THEN DC=1:GOTO 1400
1390 DC=2
1400 GOSUB 1900
1410 PRINT CHR$(125)
1420 PRINT :PRINT
1430 POKE 752,0
1440 PRINT "MNEMONIC:  ";MN$;"      ";"0P CODE:   ";OP
1450 PRINT "ADDR MODE: ";PRE$;"  ";SUF$
1460 PRINT "DATA CNT:  ";DC
```

```
1470 PRINT "DATA:       ";DAT1$;" ";DAT2$
1480 PRINT :PRINT
1490 PRINT "THIS WILL BE ENTERED AT: ";PC
1500 PRINT
1510 PRINT "PRESS RETURN TO CONTINUE, 'E' TO EDIT"
1520 INPUT A$
1530 IF A$="" THEN GOTO 1560
1540 IF A$="E" THEN GOTO 1100
1550 GOTO 1510
1555 REM **POKE PROGRAM INTO MEMORY**
1560 TRAP 1440
1570 IF DAT1$<>"   " THEN DAT1=VAL(DAT1$)
1575 IF DAT2$<>"   " THEN DAT2=VAL(DAT2$)
1580 POKE PC,OP
1590 IF DAT1$<>"   " THEN PC=PC+1:POKE PC,DAT1
1600 IF DAT2$<>"   " THEN PC=PC+1:POKE PC,DAT2
1610 PC=PC+1
1620 GOTO 1100
1630 REM **PRINT OUT**
1635 IF A$="P" THEN OPEN #3,8,0,"P:"
1640 FOR I=ST TO PC-1:X=PEEK(I)
1650 IF A$="P" THEN PRINT #3;X;:PRINT #3;",";
1655 IF A$="L" THEN PRINT I;" : ";X
1660 NEXT I
1665 IF A$="P" THEN CLOSE #3:LPRINT
1670 IF A$="L" THEN PRINT "PRESS [RETURN]":INPUT A$
1680 GOTO 1100
1900 REM **STANDARDIZE PRE$ AND SUF$**
1910 IF PRE$=" #" THEN PRE$="# "
1920 IF SUF$=" ,X" THEN SUF$=",X "
1930 IF SUF$=" ,Y" THEN SUF$=",Y "
1940 IF SUF$=" ) " OR SUF$="  )" THEN SUF$=")  "
2000 REM **LOOK-UP TABLE**
2010 IF MN$="STA" AND DC=1 AND SUF$=",X)" THEN OP=129:GOTO
3600
2020 IF MN$="STY" AND DC=1 AND SUF$="   " THEN OP=132:GOTO
3600
2030 IF MN$="STA" AND DC=1 AND SUF$="   " THEN OP=133:GOTO
3600
2040 IF MN$="STX" AND DC=1 AND SUF$="   " THEN OP=134:GOTO
3600
2050 IF MN$="DEY" AND DC=0 THEN OP=136:GOTO 3600
2060 IF MN$="TXA" AND DC=0 THEN OP=138:GOTO 3600
2070 IF MN$="STY" AND DC=2 THEN OP=140:GOTO 3600
2080 IF MN$="STA" AND DC=2 AND SUF$="   " THEN OP=141:GOTO
3600
2090 IF MN$="STX" AND DC=2 THEN OP=142:GOTO 3600
2100 IF MN$="BCC" THEN OP=144:GOTO 3600
2110 IF MN$="STA" AND DC=1 AND SUF$="),Y" THEN OP=145:GOTO
3600
2120 IF MN$="STY" AND DC=1 AND SUF$=",X " THEN OP=148:GOTO
3600
2130 IF MN$="STA" AND DC=1 AND SUF$=",X " THEN OP=149:GOTO
3600
```

```
2140 IF MN$="STX" AND DC=1 AND SUF$=",Y " THEN OP=150:GOTO
3600
2150 IF MN$="TYA" AND DC=0 THEN OP=152:GOTO 3600
2160 IF MN$="STA" AND DC=2 AND SUF$=",Y " THEN OP=153:GOTO
3600
2170 IF MN$="TXS" AND DC=0 THEN OP=154:GOTO 3600
2180 IF MN$="STA" AND DC=2 AND SUF$=",X " THEN OP=157:GOTO
3600
2190 IF MN$="LDY" AND DC=1 AND PRE$="# " THEN OP=160:GOTO
3600
2200 IF MN$="LDA" AND DC=1 AND SUF$=",X)" THEN OP=161:GOTO
3600
2210 IF MN$="LDX" AND PRE$="# " THEN OP=162:GOTO 3600
2220 IF MN$="LDY" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=164:GOTO 3600
2230 IF MN$="LDA" AND DC=1 AND PRE$="  " AND SUF$="   " THEN
OP=165:GOTO 3600
2240 IF MN$="LDX" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=166:GOTO 3600
2250 IF MN$="TAY" AND DC=0 THEN OP=168:GOTO 3600
2260 IF MN$="LDA" AND PRE$="# " THEN OP=169:GOTO 3600
2270 IF MN$="TAX" AND DC=0 THEN OP=170:GOTO 3600
2280 IF MN$="LDY" AND DC=2 AND SUF$="   " THEN OP=172:GOTO
3600
2290 IF MN$="LDA" AND DC=2 AND SUF$="   " THEN OP=173:GOTO
3600
2300 IF MN$="LDX" AND DC=2 AND SUF$="   " THEN OP=174:GOTO
3600
2310 IF MN$="BCS" AND DC=0 THEN OP=176:GOTO 3600
2320 IF MN$="LDA" AND DC=1 AND SUF$="),Y" THEN OP=177:GOTO
3600
2330 IF MN$="LDY" AND DC=1 AND SUF$=",X " THEN OP=180:GOTO
3600
2340 IF MN$="LDA" AND DC=1 AND SUF$=",X " THEN OP=181:GOTO
3600
2350 IF MN$="LDX" AND DC=1 AND SUF$=",Y " THEN OP=182:GOTO
3600
2360 IF MN$="CLV" AND DC=0 THEN OP=184:GOTO 3600
2370 IF MN$="LDA" AND DC=2 AND SUF$=",Y " THEN OP=185:GOTO
3600
2380 IF MN$="TSX" AND DC=0 THEN OP=186:GOTO 3600
2390 IF MN$="LDY" AND DC=2 AND SUF$=",X " THEN OP=188:GOTO
3600
2400 IF MN$="LDA" AND DC=2 AND SUF$=",X " THEN OP=189:GOTO
3600
2410 IF MN$="LDX" AND DC=2 AND SUF$=",Y " THEN OP=190:GOTO
3600
2420 IF MN$="CPY" AND PRE$="# " THEN OP=192:GOTO 3600
2430 IF MN$="CMP" AND DC=1 AND SUF$=",X)" THEN OP=193:GOTO
3600
2440 IF MN$="CPY" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=196:GOTO 3600
2450 IF MN$="CMP" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=197:GOTO 3600
```

```
2460 IF MN$="DEC" AND DC=1 AND SUF$="   " THEN OP=198:GOTO
3600
2470 IF MN$="INY" AND DC=0 THEN OP=200:GOTO 3600
2480 IF MN$="CMP" AND PRE$="# " THEN OP=201:GOTO 3600
2490 IF MN$="DEX" AND DC=0 THEN OP=202:GOTO 3600
2500 IF MN$="CPY" AND DC=2 THEN OP=204:GOTO 3600
2510 IF MN$="CMP" AND DC=2 AND SUF$="   " THEN OP=205:GOTO
3600
2520 IF MN$="DEC" AND DC=2 AND SUF$="   " THEN OP=206:GOTO
3600
2530 IF MN$="BNE" THEN OP=208:GOTO 3600
2540 IF MN$="CMP" AND DC=1 AND SUF$="),Y" THEN OP=209:GOTO
3600
2550 IF MN$="CMP" AND DC=1 AND SUF$=",X " THEN OP=213:GOTO
3600
2560 IF MN$="DEC" AND DC=1 AND SUF$=",X " THEN OP=214:GOTO
3600
2570 IF MN$="CLD" AND DC=0 THEN OP=216:GOTO 3600
2580 IF MN$="CMP" AND DC=2 AND SUF$=",Y " THEN OP=217:GOTO
3600
2590 IF MN$="CMP" AND DC=2 AND SUF$=",X " THEN OP=221:GOTO
3600
2600 IF MN$="DEC" AND DC=2 AND SUF$=",X " THEN OP=222:GOTO
3600
2610 IF MN$="CPX" AND PRE$="# " THEN OP=224:GOTO 3600
2620 IF MN$="SBC" AND DC=1 AND SUF$=",X)" THEN OP=225:GOTO
3600
2630 IF MN$="CPX" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=228:GOTO 3600
2640 IF MN$="SBC" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=229:GOTO 3600
2650 IF MN$="INC" AND DC=1 AND SUF$="   " THEN OP=230:GOTO
3600
2660 IF MN$="INX" AND DC=0 THEN OP=232:GOTO 3600
2670 IF MN$="SBC" AND PRE$="# " THEN OP=233:GOTO 3600
2680 IF MN$="NOP" AND DC=0 THEN OP=234:GOTO 3600
2690 IF MN$="CPX" AND DC=2 THEN OP=236:GOTO 3600
2700 IF MN$="SBC" AND DC=2 AND SUF$="   " THEN OP=237:GOTO
3600
2710 IF MN$="INC" AND DC=2 AND SUF$="   " THEN OP=238:GOTO
3600
2720 IF MN$="BEQ" AND DC=1 THEN OP=240:GOTO 3600
2730 IF MN$="SBC" AND DC=1 AND SUF$="),Y" THEN OP=241:GOTO
3600
2740 IF MN$="SBC" AND DC=1 AND SUF$=",X " THEN OP=245:GOTO
3600
2750 IF MN$="INC" AND DC=1 AND SUF$=",X " THEN OP=246:GOTO
3600
2760 IF MN$="SED" AND DC=0 THEN OP=248:GOTO 3600
2770 IF MN$="SBC" AND DC=2 AND SUF$=",Y " THEN OP=249:GOTO
3600
2780 IF MN$="SBC" AND DC=2 AND SUF$=",X " THEN OP=253:GOTO
3600
2790 IF MN$="INC" AND DC=2 AND SUF$=",X " THEN OP=254:GOTO
```

```
3600
2800 IF MN$="BRK" AND DC=0 THEN OP=0:GOTO 3600
2810 IF MN$="ORA" AND DC=1 AND SUF$=",X)" THEN OP=1:GOTO 3600
2820 IF MN$="ORA" AND DC=1 AND SUF$="   " THEN OP=5:GOTO 3600
2830 IF MN$="ASL" AND DC=1 AND SUF$="   " THEN OP=6:GOTO 3600
2840 IF MN$="PHP" AND DC=0 THEN OP=8:GOTO 3600
2850 IF MN$="ORA" AND PRE$="# " THEN OP=9:GOTO 3600
2860 IF MN$="ASL" AND DC=0 THEN OP=10:GOTO 3600
2870 IF MN$="ORA" AND DC=2 AND SUF$="   " THEN OP=13:GOTO
3600
2880 IF MN$="ASL" AND DC=2 AND SUF$="   " THEN OP=14:GOTO
3600
2890 IF MN$="BPL" AND DC=0 THEN OP=16:GOTO 3600
2900 IF MN$="ORA" AND DC=1 AND SUF$=")Y" THEN OP=17:GOTO 3600
2910 IF MN$="ORA" AND DC=1 AND SUF$=",X " THEN OP=21:GOTO
3600
2920 IF MN$="ASL" AND DC=1 AND SUF$=",X " THEN OP=22:GOTO
3600
2930 IF MN$="CLC" AND DC=0 THEN OP=24:GOTO 3600
2940 IF MN$="ORA" AND DC=2 AND SUF$=",Y " THEN OP=25:GOTO
3600
2950 IF MN$="ORA" AND DC=2 AND SUF$=",X " THEN OP=29:GOTO
3600
2960 IF MN$="ASL" AND DC=2 AND SUF$=",X " THEN OP=30:GOTO
3600
2970 IF MN$="JSR" AND DC=0 THEN OP=32:GOTO 3600
2980 IF MN$="AND" AND DC=1 AND SUF$=",X)" THEN OP=33:GOTO
3600
2990 IF MN$="BIT" AND DC=1 THEN OP=36:GOTO 3600
3000 IF MN$="AND" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=37:GOTO 3600
3010 IF MN$="ROL" AND DC=1 AND SUF$="   " THEN OP=38:GOTO
3600
3020 IF MN$="PLP" AND DC=0 THEN OP=40:GOTO 3600
3030 IF MN$="AND" AND PRE$="# " THEN OP=41:GOTO 3600
3040 IF MN$="ROL" AND DC=0 THEN OP=42:GOTO 3600
3050 IF MN$="BIT" AND DC=2 THEN OP=44:GOTO 3600
3060 IF MN$="AND" AND DC=2 AND SUF$="   " THEN OP=45:GOTO
3600
3070 IF MN$="ROL" AND DC=2 AND SUF$="   " THEN OP=46:GOTO
3600
3080 IF MN$="BMI" AND DC=0 THEN OP=48:GOTO 3600
3090 IF MN$="AND" AND DC=1 AND SUF$="),Y" THEN OP=49:GOTO
3600
3100 IF MN$="AND" AND DC=1 AND SUF$=",X " THEN OP=53:GOTO
3600
3110 IF MN$="ROL" AND DC=1 AND SUF$=",X " THEN OP=54:GOTO
3600
3120 IF MN$="SEC" AND DC=0 THEN OP=56:GOTO 3600
3130 IF MN$="AND" AND DC=2 AND SUF$=",Y " THEN OP=57:GOTO
3600
3140 IF MN$="AND" AND DC=2 AND SUF$=",X " THEN OP=61:GOTO
3600
3150 IF MN$="ROL" AND DC=2 AND SUF$=",X " THEN OP=62:GOTO
```

```
3600
3160 IF MN$="RTI" AND DC=0 THEN OP=64:GOTO 3600
3170 IF MN$="EOR" AND DC=1 AND SUF$=",X)" THEN OP=65:GOTO
3600
3180 IF MN$="EOR" AND DC=1 AND SUF$="   " THEN OP=69:GOTO
3600
3190 IF MN$="LSR" AND DC=1 AND SUF$="   " THEN OP=70:GOTO
3600
3200 IF MN$="PHA" AND DC=0 THEN OP=72:GOTO 3600
3210 IF MN$="EOR" AND PRE$="# " THEN OP=73:GOTO 3600
3220 IF MN$="LSR" AND DC=0 THEN OP=74:GOTO 3600
3230 IF MN$="JMP" AND DC=2 AND SUF$="   " THEN OP=76:GOTO
3600
3240 IF MN$="LSR" AND DC=2 AND SUF$="   " THEN OP=78:GOTO
3600
3250 IF MN$="BVC" AND DC=0 THEN OP=80:GOTO 3600
3260 IF MN$="EOR" AND DC=1 AND SUF$="),Y" THEN OP=81:GOTO
3600
3270 IF MN$="EOR" AND DC=1 AND SUF$=",X " THEN OP=85:GOTO
3600
3280 IF MN$="LSR" AND DC=1 AND SUF$=",X " THEN OP=86:GOTO
3600
3290 IF MN$="CLI" AND DC=0 THEN OP=88:GOTO 3600
3300 IF MN$="EOR" AND DC=2 AND SUF$=",Y " THEN OP=89:GOTO
3600
3310 IF MN$="EOR" AND DC=2 AND SUF$=",X " THEN OP=93:GOTO
3600
3320 IF MN$="LSR" AND DC=2 AND SUF$=",X " THEN OP=94:GOTO
3600
3340 IF MN$="RTS" AND DC=0 THEN OP=96:GOTO 3600
3350 IF MN$="ADC" AND DC=1 AND SUF$=",X)" THEN OP=97:GOTO
3600
3360 IF MN$="ADC" AND DC=1 AND PRE$="# " AND SUF$="   " THEN
OP=101:GOTO 3600
3370 IF MN$="ROR" AND DC=1 THEN OP=102:GOTO 3600
3380 IF MN$="PLA" AND DC=0 THEN OP=104:GOTO 3600
3390 IF MN$="ADC" AND PRE$="# " THEN OP=105:GOTO 3600
3400 IF MN$="ROR" AND DC=0 THEN OP=106:GOTO 3600
3410 IF MN$="JMP" AND DC=2 AND SUF$=")  " THEN OP=108:GOTO
3600
3420 IF MN$="ADC" AND DC=2 AND SUF$="   " THEN OP=109:GOTO
3600
3430 IF MN$="BVS" AND DC=0 THEN OP=112:GOTO 3600
3440 IF MN$="ADC" AND DC=1 AND SUF$="),Y" THEN OP=113:GOTO
3600
3450 IF MN$="ADC" AND DC=1 AND SUF$=",X " THEN OP=117:GOTO
3600
3460 IF MN$="ROR" AND DC=1 AND SUF$=",X " THEN 118:GOTO 3600
3470 IF MN$="SEI" AND DC=0 THEN OP=120:GOTO 3600
3480 IF MN$="ADC" AND DC=2 AND SUF$=",Y " THEN OP=121:GOTO
3600
3490 IF MN$="ADC" AND DC=2 AND SUF$=",X " THEN OP=125:GOTO
3600
3500 IF MN$="ROR" AND DC=2 AND SUF$=",X " THEN OP=126:GOTO
3600
3510 MN$="???":OP=666:GOTO 3600
3600 RETURN
```

Appendix C

BASIC Disassembler

The BASIC disassembler is an example of a simple look-up table translator. By PEEKing consecutive memory locations and comparing their contents to the OP CODES of the 6502 instruction set, it is able to generate a list of mnemonics and operands. Operand is a general term used to refer to the addresses or data numbers following an OP CODE. As mentioned in the previous appendix, the disassembler can be used in conjunction with the assembler or, to disassemble someone else's machine language routine. To use the disassembler in this manner you should add the following lines to the beginning of the program:

```
100 FOR I=10 TO NUMBER
120 READ ML:POKE MEMORY+I,ML
130 NEXT I
140 REM DATA NUMBERS HERE
150 REM DATA NUMBERS HERE
160 REM DATA NUMBERS HERE
          ■
          ■
          ■
```

NUMBER in line 100 is 1 less than the total data numbers in the program you are disassembling. MEMORY in line 120 is the place where you want to store the data numbers. This is usually page 6 (1536), but you could store them above RAMTOP. If that is your choice add lines:

```
80 POKE 106,PEEK(106)-8
90 GRAPHICS 0
```

to lower RAMTOP eight pages and set NUMBER accordingly. If you do this keep in mind that each clear screen call wipes out memory above RAMTOP, so leave a buffer.

The program allows you to print out the disassembled code that has been displayed on the screen. The line that allows this is:

```
10 OPEN#2,5,0,"E:"
```

This is a forced read command and must be used with some care. *It should precede all other lines in the program.*

When you run this program you will find that the addresses are listed in hexadecimal. In this book we have not made use of hex numbers beyond the introduction to number systems in chapter one. That was a deliberate choice on our part since all of the machine language routines we have discussed were meant to be called by BASIC, and BASIC requires decimal numbers. However, many programmers use hexadecimal numbers and so it is useful to become familiar with them. By exposing you to hex numbers here we believe that you can begin to get used to them without being forced into innumerable conversions between number systems. Of course, if you wish, you can modify the program to output the addresses in decimal.

APPENDIX C

```
4000 OPEN #2,5,0,"E:"
4010 REM DISASSEMBLER VERSION 1.0
4020 REM
4030 PRINT CHR$(125)
4040 REM
4050 DIM HEX$(16),A$(5),ADDR$(5),PRE$(2),SUF$(3),LINE$(120)
4060 DIM MN$(3),OP$(3),B$(1),HEX2$(1),HEX3$(1),HEX4$(1)
4070 HEX$="0123456789ABCDEF"
4080 PRINT
4090 PRINT "6502 IN-MEMORY DISASSEMBLER"
4100 PRINT
4110 PRINT
4120 REM
4130 PRINT "ENTER STARTING ADDRESS IN DECIMAL"
4140 INPUT A$
4150 IF A$="" THEN END
4160 A=VAL(A$)
4170 PRINT CHR$(125)
4180 PRINT
4190 PRINT "  ADDR   MNEM   OP CODE   DATA"
4200 PRINT
4210 REM DISPLAY ONLY 18 LINES AT A TIME
4220 CNT=-1:LINE=-1
4230 CNT=CNT+1:LINE=LINE+1
4240 IF LINE=18 THEN GOTO 4470
4250 ADDR=CNT+A
4260 GOSUB 4550
4270 REM READ MEMORY
4280 P=PEEK(ADDR)
4290 OP$=STR$(P)
4300 MN$="???":PRE$="  ":SUF$="   "
4310 DC=0:R=0:RL=0
4320 GOSUB 4670
4330 REM PRINTOUT SECTION
4340 IF DC=1 THEN GOTO 4380
4350 IF DC=2 THEN GOTO 4410
4360 PRINT "  ";ADDR$;"   ";MN$;"     ";OP$
4370 ADDR$="":GOTO 4230
4380 PRINT "  ";ADDR$;"   ";MN$;"     ";OP$;"   ";PRE$;
4390 CNT=CNT+1:ADDR=CNT+A:DTA=PEEK(ADDR):PRINT DTA;SUF$
4400 ADDR$="":GOTO 4230
4410 PRINT "  ";ADDR$;"   ";MN$;"     ";OP$;"   ";PRE$;
4420 CNT=CNT+1:ADDR=CNT+A:DTA=PEEK(ADDR)
4430 PRINT DTA;
4440 CNT=CNT+1:ADDR=CNT+A:DTA=PEEK(ADDR)
4450 PRINT ", ";DTA;SUF$
4460 ADDR$="":GOTO 4230
4470 REM PAUSE SECTION
4480 PRINT "<RETURN> TO END : 'N' FOR NEXT : P FOR PRINT"
4490 TRAP 4500
4500 INPUT B$
4510 IF B$="" THEN END
```

```
4520 IF B$="N" THEN A=ADDR+1:GOTO 4170
4530 IF B$="P" THEN GOTO 6190
4540 GOTO 4480
4550 HEX1=INT(ADDR/4096)
4560 ADDR$=HEX$(HEX1+1,HEX1+1)
4570 HEX2=INT((ADDR-HEX1*4096)/256)
4580 HEX2$=HEX$(HEX2+1,HEX2+1)
4590 ADDR$(LEN(ADDR$)+1)=HEX2$
4600 HEX3=INT((ADDR-HEX1*4096-HEX2*256)/16)
4610 HEX3$=HEX$(HEX3+1,HEX3+1)
4620 ADDR$(LEN(ADDR$)+1)=HEX3$
4630 HEX4=INT(ADDR-HEX1*4096-HEX2*256-HEX3*16)
4640 HEX4$=HEX$(HEX4+1,HEX4+1)
4650 ADDR$(LEN(ADDR$)+1)=HEX4$
4660 RETURN
4670 REM LOOK-UP TABLE
4680 IF P=129 THEN MN$="STA":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
4690 IF P=132 THEN MN$="STY":DC=1:GOTO 6180
4700 IF P=133 THEN MN$="STA":DC=1:GOTO 6180
4710 IF P=134 THEN MN$="STX":DC=1:GOTO 6180
4720 IF P=136 THEN MN$="DEY":GOTO 6180
4730 IF P=138 THEN MN$="TXA":GOTO 6180
4740 IF P=140 THEN MN$="STY":DC=2:GOTO 6180
4750 IF P=141 THEN MN$="STA":DC=2:GOTO 6180
4760 IF P=142 THEN MN$="STX":DC=2:GOTO 6180
4770 IF P=144 THEN MN$="BCC":DC=1:GOTO 6180
4780 IF P=145 THEN MN$="STA":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
4790 IF P=148 THEN MN$="STY":DC=1:SUF$=",X ":GOTO 6180
4800 IF P=149 THEN MN$="STA":DC=1:SUF$=",X ":GOTO 6180
4810 IF P=150 THEN MN$="STX":DC=1:SUF$=",Y ":GOTO 6180
4820 IF P=152 THEN MN$="TYA":GOTO 6180
4830 IF P=153 THEN MN$="STA":DC=2:SUF$=",Y "
4840 IF P=154 THEN MN$="TXS":GOTO 6180
4850 IF P=157 THEN MN$="STA":DC=2:SUF$=",X ":GOTO 6180
4860 IF P=160 THEN MN$="LDY":DC=1:PRE$="# ":GOTO 6180
4870 IF P=161 THEN MN$="LDA":DC=1:PRE$=" (":SUF$=",X) ":GOTO
6180
4880 IF P=162 THEN MN$="LDX":DC=1:PRE$="# ":GOTO 6180
4890 IF P=164 THEN MN$="LDY":DC=1:GOTO 6180
4900 IF P=165 THEN MN$="LDA":DC=1:GOTO 6180
4910 IF P=166 THEN MN$="LDX":DC=1:GOTO 6180
4920 IF P=168 THEN MN$="TAY":GOTO 6180
4930 IF P=169 THEN MN$="LDA":DC=1:PRE$="# ":GOTO 6180
4940 IF P=170 THEN MN$="TAX":GOTO 6180
4950 IF P=172 THEN MN$="LDY":DC=2:GOTO 6180
4960 IF P=173 THEN MN$="LDA":DC=2:GOTO 6180
4970 IF P=174 THEN MN$="LDX":DC=2:GOTO 6180
4980 IF P=176 THEN MN$="BCS":DC=1:GOTO 6180
4990 IF P=177 THEN MN$="LDA":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5000 IF P=180 THEN MN$="LDY":DC=1:SUF$=",X ":GOTO 6180
5010 IF P=181 THEN MN$="LDA":DC=1:SUF$=",X ":GOTO 6180
```

```
5020 IF P=182 THEN MN$="LDX":DC=1:SUF$=",Y ":GOTO 6180
5030 IF P=184 THEN MN$="CLV":GOTO 6180
5040 IF P=185 THEN MN$="LDA":DC=2:SUF$=",Y ":GOTO 6180
5050 IF P=186 THEN MN$="TSX":GOTO 6180
5060 IF P=188 THEN MN$="LDY":DC=2:SUF$=",X ":GOTO 6180
5070 IF P=189 THEN MN$="LDA":DC=2:SUF$=",X ":GOTO 6180
5080 IF P=190 THEN MN$="LDX":DC=2:SUF$=",Y ":GOTO 6180
5090 IF P=192 THEN MN$="CPY":DC=1:PRE$="# ":GOTO 6180
5100 IF P=193 THEN MN$="CMP":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
5110 IF P=196 THEN MN$="CPY":DC=1:GOTO 6180
5120 IF P=197 THEN MN$="CMP":DC=1:GOTO 6180
5130 IF P=198 THEN MN$="DEC":DC=1:GOTO 6180
5140 IF P=200 THEN MN$="INY":GOTO 6180
5150 IF P=201 THEN MN$="CMP":DC=1:PRE$="# ":GOTO 6180
5160 IF P=202 THEN MN$="DEX":GOTO 6180
5170 IF P=204 THEN MN$="CPY":DC=2:GOTO 6180
5180 IF P=205 THEN MN$="CMP":DC=2:GOTO 6180
5190 IF P=206 THEN MN$="DEC":DC=2:GOTO 6180
5200 IF P=208 THEN MN$="BNE":DC=1:GOTO 6180
5210 IF P=209 THEN MN$="CMP":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5220 IF P=213 THEN MN$="CMP":DC=1:SUF$=",X ":GOTO 6180
5230 IF P=214 THEN MN$="DEC":DC=1:SUF$=",X ":GOTO 6180
5240 IF P=216 THEN MN$="CLD":GOTO 6180
5250 IF P=217 THEN MN$="CMP":DC=2:SUF$=",Y ":GOTO 6180
5260 IF P=221 THEN MN$="CMP":DC=2:SUF$=",X ":GOTO 6180
5270 IF P=222 THEN MN$="DEC":DC=2:SUF$=",X ":GOTO 6180
5280 IF P=224 THEN MN$="CPX":DC=1:PRE$="# ":GOTO 6180
5290 IF P=225 THEN MN$="SBC":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
5300 IF P=228 THEN MN$="CPX":DC=1:GOTO 6180
5310 IF P=229 THEN MN$="SBC":DC=1:GOTO 6180
5320 IF P=230 THEN MN$="INC":DC=1:GOTO 6180
5330 IF P=232 THEN MN$="INX":GOTO 6180
5340 IF P=233 THEN MN$="SBC":DC=1:PRE$="# ":GOTO 6180
5350 IF P=234 THEN MN$="NOP":GOTO 6180
5360 IF P=236 THEN MN$="CPX":DC=2:GOTO 6180
5370 IF P=237 THEN MN$="SBC":DC=2:GOTO 6180
5380 IF P=238 THEN MN$="INC":DC=2:GOTO 6180
5390 IF P=240 THEN MN$="BEQ":DC=1:GOTO 6180
5400 IF P=241 THEN MN$="SBC":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5410 IF P=245 THEN MN$="SBC":DC=1:SUF$=",X ":GOTO 6180
5420 IF P=246 THEN MN$="INC":DC=1:SUF$=",X ":GOTO 6180
5430 IF P=248 THEN MN$="SED":GOTO 6180
5440 IF P=249 THEN MN$="SBC":DC=2:SUF$=",Y ":GOTO 6180
5450 IF P=253 THEN MN$="SBC":DC=2:SUF$=",X ":GOTO 6180
5460 IF P=254 THEN MN$="INC":DC=2:SUF$=",X ":GOTO 6180
5470 IF P=0 THEN MN$="BRK":GOTO 6180
5480 IF P=1 THEN MN$="ORA":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
5490 IF P=5 THEN MN$="ORA":DC=1:GOTO 6180
5500 IF P=6 THEN MN$="ASL":DC=1:GOTO 6180
```

```
5510 IF P=8 THEN MN$="PHP:GOTO 4000"
5520 IF P=9 THEN MN$="ORA":DC=1:PRE$="# ":GOTO 6180
5530 IF P=10 THEN MN$="ASL:GOTO 4000"
5540 IF P=13 THEN MN$="ORA":DC=2:GOTO 6180
5550 IF P=14 THEN MN$="ASL":DC=2:GOTO 6180
5560 IF P=16 THEN MN$="BPL":DC=1:GOTO 6180
5570 IF P=17 THEN MN$="ORA":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5580 IF P=21 THEN MN$="ORA":DC=1:SUF$=",X ":GOTO 6180
5590 IF P=22 THEN MN$="ASL":DC=1:SUF$=",X ":GOTO 6180
5600 IF P=24 THEN MN$="CLC":GOTO 6180
5610 IF P=25 THEN MN$="ORA":DC=2:SUF$=",Y ":GOTO 6180
5620 IF P=29 THEN MN$="ORA":DC=2:SUF$=",X ":GOTO 6180
5630 IF P=30 THEN MN$="ASL":DC=2:SUF$=",X ":GOTO 6180
5640 IF P=32 THEN MN$="JSR":DC=2:GOTO 6180
5650 IF P=33 THEN MN$="AND":DC=1:PRE$=" (":SUF$=",X) ":GOTO
6180
5660 IF P=36 THEN MN$="BIT":DC=1:GOTO 6180
5670 IF P=37 THEN MN$="AND":DC=1:GOTO 6180
5680 IF P=38 THEN MN$="ROL":DC=1:GOTO 6180
5690 IF P=40 THEN MN$="PLP":GOTO 6180
5700 IF P=41 THEN MN$="AND":DC=1:PRE$="# ":GOTO 6180
5710 IF P=42 THEN MN$="ROL":GOTO 6180
5720 IF P=44 THEN MN$="BIT":DC=2:GOTO 6180
5730 IF P=45 THEN MN$="AND":DC=2:GOTO 6180
5740 IF P=46 THEN MN$="ROL":DC=2:GOTO 6180
5750 IF P=48 THEN MN$="BMI":DC=1:GOTO 6180
5760 IF P=49 THEN MN$="AND":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5770 IF P=53 THEN MN$="AND":DC=1:SUF$=",X ":GOTO 6180
5780 IF P=54 THEN MN$="ROL":DC=1:SUF$=",X":GOTO 6180
5790 IF P=56 THEN MN$="SEC":GOTO 6180
5800 IF P=57 THEN MN$="AND";DC=2:SUF$=",Y ":GOTO 6180
5810 IF P=61 THEN MN$="AND":DC=2:SUF$=",X ":GOTO 6180
5820 IF P=62 THEN MN$="ROL":DC=2:SUF$=",X ":GOTO 6180
5830 IF P=64 THEN MN$="RTI":GOTO 6180
5840 IF P=65 THEN MN$="EOR":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
5850 IF P=69 THEN MN$="EOR":DC=1:GOTO 6180
5860 IF P=70 THEN MN$="LSR":DC=1:GOTO 6180
5870 IF P=72 THEN MN$="PHA":GOTO 6180
5880 IF P=73 THEN MN$="EOR":DC=1:PRE$="# ":GOTO 6180
5890 IF P=74 THEN MN$="LSR:GOTO 4000"
5900 IF P=76 THEN MN$="JMP":DC=2:GOTO 6180
5910 IF P=78 THEN MN$="LSR":DC=2:GOTO 6180
5920 IF P=80 THEN MN$="BVC":DC=1:GOTO 6180
5930 IF P=81 THEN MN$="EOR":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
5940 IF P=85 THEN MN$="EOR":DC=1:SUF$=",X ":GOTO 6180
5950 IF P=86 THEN MN$="LSR":DC=1:SUF$=",X ":GOTO 6180
5960 IF P=88 THEN MN$="CLI":GOTO 6180
5970 IF P=89 THEN MN$="EOR":DC=2:SUF$=",Y ":GOTO 6180
5980 IF P=93 THEN MN$="EOR":DC=2:SUF$=",X ":GOTO 6180
5990 IF P=94 THEN MN$="LSR":DC=2:SUF$=",X ":GOTO 6180
```

```
6000 IF P=96 THEN MN$="RTS":GOTO 6180
6010 IF P=97 THEN MN$="ADC":DC=1:PRE$=" (":SUF$=",X)":GOTO
6180
6020 IF P=101 THEN MN$="ADC":DC=1:GOTO 6180
6030 IF P=102 THEN MN$="ROR":DC=1:GOTO 6180
6040 IF P=104 THEN MN$="PLA":GOTO 6180
6050 IF P=105 THEN MN$="ADC":DC=1:PRE$="# ":GOTO 6180
6060 IF P=106 THEN MN$="ROR":GOTO 6180
6070 IF P=108 THEN MN$="JMP":DC=1:PRE$=" (":SUF$=")  ":GOTO
6180
6080 IF P=109 THEN MN$="ADC":DC=2:GOTO 6180
6090 IF P=110 THEN MN$="ROR":DC=2:GOTO 6180
6100 IF P=112 THEN MN$="BVS":DC=1:GOTO 6180
6110 IF P=113 THEN MN$="ADC":DC=1:PRE$=" (":SUF$="),Y":GOTO
6180
6120 IF P=117 THEN MN$="ADC":DC=1:SUF$=",X ":GOTO 6180
6130 IF P=118 THEN MN$="ROR":DC=1:SUF$=",X ":GOTO 6180
6140 IF P=120 THEN MN$="SEI":GOTO 6180
6150 IF P=121 THEN MN$="ADC":DC=2:SUF$=",Y ":GOTO 6180
6160 IF P=125 THEN MN$="ADC":DC=2:SUF$=",X ":GOTO 6180
6170 IF P=126 THEN MN$="ROR":DC=2:SUF$=",X ":GOTO 6180
6180 RETURN
6190 POSITION PEEK(82),0
6200 FOR I=1 TO 20
6210 INPUT #2,LINE$
6220 LPRINT ,LINE$
6230 NEXT I
6240 GOTO 4480
```

Appendix D

Memory Map

This memory map is arranged to give you an overview of the organization of Atari memory. We have given emphasis to the specific memory locations that are directly useful in terms of sound and graphics as explained in the text of this book. See the end of the memory map for sources of the complete memory allocations.

<table>
<tr><th>LABEL</th><th>DECIMAL</th><th>HEX</th><th>FUNCTION</th></tr>
<tr><td colspan="4">Page zero is found at locations zero to 255 ($0-$FF). These locations are accessed faster and easier by the machine. On page 0 locations 0 to 127 are observed for the OS, while locations 128 to 255 are for BASIC and the programmer's use.</td></tr>
<tr><td colspan="4">Locations 2 to 7 are not cleared out by any of the startup routines. Locations 16 - 127 are cleared on warmstart and coldstart.</td></tr>
</table>

POKMSK	16	10	POKEY interrupts
BRKKEY	17	11	Indicates status of the BREAK key
RTCLOCK	18,19,20	12,13,14	Realtime clock
Locations 32 to 47 are the Page 0 Input/Output Control Block.			
CRITIC	66	42	Critical I/O region flag used in ML Vertical Blank routines
ATRACT	77	4D	Attract mode timer and flag
DRKMSK	78	4E	Dark attract mode
COLRSH	79	4F	Attract color shifter
LMARGN	82	52	Column of left margin of text, GR 0 or text window
RMARGN	83	53	Right margin of text screen
ROWCRS	84	54	Current cursor row, 0-191
COLCRS		55,56	Current cursor column 0-319
DINDEX	87	57	Current screen mode
SAVMSC	88,89	58,59	Lo-Byte/Hi-Byte of the start of screen memory
LDROW	90	5A	Previous graphics cursor row
OLDCOL	91,92	5B,5C	previous graphics cursor column
OLDCHR	93	5D	Data under graphics window cursor
ADRESS	100,101	64,65	Temporary storage holds contents of SAVMSC
RAMTOP	106	6A	Top of RAM memory

Locations 128-210 are for programmer's use and BASIC page 0 RAM. Locations 128-145 are the site of BASIC program pointers; 146 to 202 is for BASIC RAM; 203-209 are not used by BASIC. Locations 146 to 202 are reserved for use by 8K BASIC. Locations 176 to 207 are reserved for Assembler Editor Cartridge. Locations 212 to 255 are reserved for the floating point package.

LOMEM	128,129	80,81	BASIC low memory pointer token output address
MEMTOP	144,145	90,91	BASIC top of memory pointer
STOPLN	186,187	BA,BB	Line number where a STOP or TRAP occured
ERRSAVE	195	C3	Error number causing the STOP or TRAP to occur
------	203-207	CB-CF	Unused by BASIC or Assembler Cartridge
-----	208-209	D0-D1	Unused by BASIC
-----	210-211	D2-D3	Reserved for BASIC or other cartridge

Page 1 - The Stack are at locations 256 - 511. Machine language, JSR, PHA and interrupts result in data being written to page 1; RTS, PLA, and RTI instructions read data from page 1.

VDSLST	512,513	200,201	NMI DLI vector
VPRCED	514,515	202,203	Serial proceed
VINTER	516,517	204,205	Serial interrupt
VBREAK	518,519	206,207	Break instruction vector
VKEYBD	520,521	208,209	Keyboard interrupt vector
VTIMR1	528,529	210,211	POKEY timer 1 vector
VTIMR2	530,531	212,213	POKEY timer 2 vector
VTIMR3	532,533	214,215	POKEY timer 3 vector
VIMIRQ	534,535	216,217	General IRQ immediate vector

Locations 536 - 558 are used for the system software timers and are accessed by assembly language. These timers count backwards every sixtieth or thirtieth of a second until they reach zero.

CDTMV1	536,537	218,219	System timer 1 value
CDTMV2	538,539	21A,21B	System timer 2 value
CDTMV3	540,541	21C,21D	System timer 3 value
CDTMV4	542,543	21E,21F	System timer 4 value
CDTMV5	544,545	220,221	System timer 5 value
VVBLKI	546,547	222,223	VB immediate jump address
VVBLKD	548,549	224,225	VB deferred jump address
CDTMA1	550,551	226,227	System timer 1 jump addr
CDTMA2	552,553	228,229	System timer 2 jump addr
CDTMF3	554	22A	System time 3 flag
SRTIMR	555	22B	Software repeat timer
CDTMF4	556	22C	System timer 4 flag
INTEMP	557	22D	Temporary register used by SETVBL
CTMF5	558	22E	System timer 5 flag
SDMCTL	559	22F	DMA enable
SDLSTL	560,561	230,231	Hi-Byte/Lo-Byte of the starting address of the display list
SSKCTL	562	232	Serial port control
SPARE	563	233	No OS use
LPENH	564	234	Horizontal value of light pen
LPENV	565	235	Vertical value of light pen
BRKKY	566,567	236,237	Break key interrupt vector
	568,569	238,239	Two spare bytes
------	581	245	Spare byte buffer
GPRIOR	623	26F	Priority selection register - shadow of 53275

Locations 624 - 647 are paddles, joysticks and lightpen controls.			
PADDL0	624	270	Value of paddle 0
PADDL1	625	271	Value of paddle 1
PADDL2	626	272	Value of paddle 2
PADDL3	627	273	Value of paddle 3
PADDL4	628	274	Value of paddle 4
PADDL5	629	275	Value of paddle 5
PADDL6	630	276	Value of paddle 6
PADDL7	631	277	Value of paddle 7
STICK0	632	278	Value of joystick 0
STICK1	633	279	Value of joystick 1
STICK2	634	27A	Value of joystick 2
STICK3	635	27B	Value of joystick 3
PTRIG0	636	27C	Determines if trigger or button on paddle has been pressed
PTRIG1	637	27D	
PTRIG2	638	27E	
PTRIG3	639	27F	
PTRIG4	640	280	
PTRIG5	641	281	
PTRIG6	642	282	
PTRIG7	643	283	
STRIG0	644	284	Determines if the stick button has been pressed
STRIG1	645	285	
STRIG2	646	286	
STRIG3	647	287	

<table>
<tr><td>------</td><td>651</td><td>28B</td><td>Spare byte</td></tr>
<tr><td>------</td><td>652,653</td><td>28C,28D</td><td>OS ROM interrupt handler</td></tr>
<tr><td>------</td><td>654,655</td><td>28E,28F</td><td>Spare bytes</td></tr>
<tr><td colspan="4">Locations 656 - 703 are used for screen RAM display handler and depends on the Graphics Mode.</td></tr>
<tr><td>TXTROW</td><td>656</td><td>290</td><td>Text cursor row 0-3</td></tr>
<tr><td>TXTCOL</td><td>657,658</td><td>291,292</td><td>Text cursor column 0-39</td></tr>
<tr><td>TINDEX</td><td>659</td><td>293</td><td>Current split-screen text window</td></tr>
<tr><td>TXTMSC</td><td>660,661</td><td>294,295</td><td>Upper left corner of the text window</td></tr>
<tr><td colspan="4">Locations 704-712 are color registers for playfield, players, and missiles. They are the RAM shadow registers for 53266-53274.</td></tr>
<tr><td>PCOLR0</td><td>704</td><td>2C0</td><td>Color of player/missile 0</td></tr>
<tr><td>PCOLR1</td><td>405</td><td>2C1</td><td>color of player/missile 1</td></tr>
<tr><td>PCOLR2</td><td>406</td><td>2C2</td><td>Color of player/missile 2</td></tr>
<tr><td>PCOLR3</td><td>707</td><td>2C3</td><td>Color of player/missile 3</td></tr>
<tr><td>COLOR0</td><td>708</td><td>2C4</td><td>Color register 0
Playfield 0</td></tr>
<tr><td>COLOR1</td><td>709</td><td>2C5</td><td>Color register 1
Playfield 1</td></tr>
<tr><td>COLOR2</td><td>710</td><td>2C6</td><td>Color register 2
Playfield 2</td></tr>
<tr><td>COLOR3</td><td>711</td><td>2C7</td><td>Color register 3
Playfield 3</td></tr>
<tr><td>COLOR4</td><td>712</td><td>2C8</td><td>Color register 4
Background/Border</td></tr>
</table>

------	713-735	2C9-2DF	Spare bytes
------	736-739	2E0-2E3	Miscellaneous use
RAMSIZ	740	2E4	Top of RAM address
MEMTOP	741,742	2E5,2E6	OS top of memory pointer
------	745	2E9	Spare byte
CRSINH	752	2F0	Cursor inhibit
KEYDEL	753	2F1	Key debounce counter
CH1	754	2F2	Prior keyboard character code
CHAT	755	2F3	Character mode register
CHABAS	756	2F4	Character base register
------	757-761	2F5-2F9	Spare bytes
CHAR	762	2FA	Internal value for last character written or read
ATACHR	763	2FB	Last ATASCII character read/write and DRAWTO value
CH	764	2FC	Internal code for last key pressed
FILDAT	765	2FD	Color for XIO
DSPFLG	766	2FE	Display flag used for control characters
SSFLAG	767	2FF	Start/stop flag to halt scrolling and CNTL2

Page three: The locations 768 to 831 are the device handler and vectors to the handler routines.

Locations 832 - 959 are used for the eight Input/Output control blocks. These are the channels for the transfer of data into and out of the computer, or between devices.

<table>
<tr><td>PRNBUF</td><td>960-999</td><td>3C0-3E7 Printer buffer</td></tr>
<tr><td colspan="3">Locations 1000 - 1020 are a reserved spare buffer area.</td></tr>
<tr><td>CASBUF</td><td>1021-1151</td><td>3FD-47F cassette buffer</td></tr>
<tr><td colspan="3">Locations 1152 - 1791 are for user RAM requirements. There are 128 spare bytes available at location 1152 - 1279 if you don't use the floating point package at locations 1406 - 1535.</td></tr>
<tr><td colspan="3">Locations 1536 - 1791 are on page six and are not used by the OS. These locations may be used safely for machine language routines, redefined character sets and whatever the programmer can fit into this space.</td></tr>
<tr><td colspan="3">Locations 1792 - 2047 (page 7) are the user boot area. Locations 1792 to the address of LOMEM at 128,129 are also used by DOS and the File Management System.</td></tr>
<tr><td>CTRDGE. B</td><td>32768</td><td>8000 Used by right slot on 800</td></tr>
<tr><td>CTRDGE. A</td><td>40960</td><td>A000 Used by the left slot</td></tr>
<tr><td colspan="3">Locations 49152 - 53247 unused; reserved for future expansion. This 4K area of memory can NOT be written to by the programmer.</td></tr>
</table>

GTIA	53248-53505	D000 D0FF	processes the video signal
HPOSP0	53248	D000	(W) horizontal position of player 0
M0PF			(R) missile-playfield collision
HPOS1	53249	D001	(W) horizontal position of player 1
M1PF			(R) missile-playfield collision
HPOSP2	53250	D002	(W) horizontal position of player 2
M2PF			(R) missile-playfield collision
HPOSP3	53251	D003	(W) horizontal position of player 3
M3PF			(R) missile-playfield collision
HPOSM0	53252	D004	(W) horizontal position of missile 0
P0PF			(R) player to playfield collision
HPOSM1	53253	D005	(W) horizontal position of missile 1
P1PF			(R) player to playfield collision
HPOSM2	53254	D006	(W) horizontal position of missile 2
P2PF			(R) player to playfield collision
HPOSM3	53255	D007	(W) horizontal position of missile 3
P3PF			(R) player to playfield collision
M0PL	53256	D008	(R) missile 0 to player collision

SIZEP0			(W) size of player 0
M1PL	53257	D007	(R) missile 1 to player collision
SIZEP1			(W) size of player 1
M2PL	53258	D00A	(R) missile 2 to player collision
SIZEP2			(W) size of player 2
M3PL	53259	D00B	(R) missile 3 to player collision
SIZEP3			(W) size of player 3
P0PL	53260	D00C	(R) player 0 to player collision
SIZEM			(W) size for all missiles
GRAFP0	53261	D00D	(W) graphics shape for player 0
P1PL			(R) player 1 to player collisions
GRAPFP1	53262	D00E	(W) graphics shape for player 1
P2PL			(R) player 2 to player collisions
GRAPFP2	53263	D00F	(W) graphics shape for player 2
P3PL			(R) player 3 to player collisions
GRAPFP3	53264	D010	(W) graphics shape for player 3
TRIG0 (644)			(R) joystick trigger 0
GRAPFPM	53265	D011	(W) graphics for all missiles
TRIG1			(R) joystick trigger 1
COLPM0	53266	D012	(704) color/luminance of player/missile 0
TRIG2 (646)			(R) joystick trigger 2

COLPM1	53267	D013	(705) color/luminance of player/missile 1
TRIG3 (647)			(R) joystick 3 trigger
COLPM2	53268	D014	(706) color/luminance of player/missile 2
COLPM3	53269	D015	(707) color/luminance of player/missile 3
COLPF0	53270	D016	(708) color/luminance of playfield 0
COLPF1	53271	D017	(709) color/luminance of playfield 1
COLPF2	53272	D018	(710) color/luminance of playfield 2
COLPF3	53273	D019	(711) color/luminance of playfield 3/missile 4
COLBK	53274	D01A	(712) color/luminance of background
PRIOR	53275	D01B	Priority selection
VDELAY	53276	D01C	(W) vertical delay
GRACTL	53277	D01D	(W) use with DMACTL: latch triggers turn on players turn on missiles
HITCLR	53278	D01E	(W) clear collision registers
CONSOL	53279	D01F	(R/W) check for OPTION-SELECT-START buttons pressed

Locations 53280 - 53503 are repeats of locations 53248 - 53279. Programmers cannot use these locations.

Locations 53504 - 53759 are unused. However they are not empty. If you are interested you can PEEK these locations to see their contents.

POKEY: Locations 53760 - 54015 are the I/O chip that controls audio frequency registers and audio control registers, frequency dividers, polynoise counters, paddle controllers, random number generation, keyboard scan, serial port I/O and IRQ interrupts.

AUDF1	53760	D200	(W) Audio channel 1 freq.
POT0 (624)			(R) POT 0
AUDC1	53761	D201	(W) Audio channel 1 control
POT1 (625)			POT 1
AUDF2	53762	D202	(W) Audio channel 2 freq.
POT2 (626)			(R) Pot 2
AUDC2	53763	D203	(W) Audio channel 2 control
POT3 (627)			(R) Pot 3
AUDF3	53764	D204	(W) Audio channel 3 freq.
POT4 (628)			(R) Pot 4
AUDC3	53765	D205	(W) Audio channel 3 control
POT5 (629)			(R) Pot 5
AUDF4	53766	D206	(W) Audio channel 4 freq.
POT6 (630)			(R) Pot 6
AUDC4	53767	D207	(W) Audio channel 4 control
POT7 (631)			(R) Pot 7
AUDCTL	53768	D208	(W) Audio control
ALLPOT			(R) 8 line pot port state
STIMER	53769	D209	(W) Start POKEY timers
KBCODE (764)			(R) Keyboard code
SKREST	53770	D20A	(W) Reset serial port status
RANDOM			Random number generator
POTGO	53771	D20B	(W) Start pot scan sequence
------	53722	D20C	unused
SEROUT	53773	D20D	(W) Serial port output

SERIN			(R) Read serial port status
IRQEN	53774	D20E	(W) Interrupt request enable
IRQST			(R) Interrupt request status
SKCTL	53775	D20F	(W) Serial port control
SKSTAT			(R) Reads serial port status

Locations 53776 to 54015 are a repeat of locations 53760 to 53775. As of this writing these locations have no use.

PIA: 6520 Chip is located at addresses 54016 to 54271. These locations are used for control ports, controller jacks one through four and to process VINTER and VPRCED.

PORTA	54016	D300	R/W data from controller jacks one and two
PORTB	54017	D301	R/W data to/from jacks three and four
PACTL	54018	D302	(W/R) Port A controller
PBCTL	54019	D303	(W/R) Port B controller

Locations 54020 to 54271 are a repeat of locations 54016 to 54019.

ANTIC resides at locations 54727 to 54783 and controls GTIA, screen display and other screen functions. NMI interrupts are also processed here.

DMACTL	54272	D400	(W) DMA control (559)
CHACTL	54273	D401	(W) Character mode control (755)
DLISTL/H	54274,5	D402,3	Display list pointer (560,561)
HSCROL	54276	D404	(W) Horizontal scroll enable
VSCROL	54277	D405	(W) Vertical scroll enable
------	54278	D406	Unused
PMBASE	54279	D407	(W) Player/missile base address
------	54280	D408	Unused
CHBASE	54281	D409	(W) Character base addr.
WSYNC	54282	D40A	(W) Wait for horizontal synchronization
VCOUNT	54283	D40B	(R) Vertical line counter
PENH	54284	D40C	(R) Horizontal position of lightpen
PENV	54285	D40D	(R) Vertical position of lightpen
NMIEN	54286	D40E	(W) Non-Maskable interrupt enable
NMIRES	54286	D40F	(W) Reset for NMIST: clears interrupt request register
NMIST			(R) NMI status

Locations 54288 to 54303 are a repeat of locations 54272 to 54287. Locations 54784 to 55295 are unused. However they are not empty and therefore are not programmable. If you are interested you can PEEK into these locations and find out what is there.

Locations 59310 to 59628 are the Operating System ROM.

Locations 55296 to 57343 are used for the ROM's Floating Point Mathematics Package. The FP Package also uses page 0 locations 212-245 and page 5 locations 1406 - 1535. BASIC Trigonometric functions which use the FP routines are located at 48549 to 49145.

Locations 57344 to 58367 contain the standard Atari character set.

57344	E000	Special characters
57600	E100	Punctuation, numbers
57856	E200	Capital letters@
58112	E300	Lowercase letters

Locations 58368 to 58477 are vector tables. These are base addresses which are used by the resident handlers (screen editor, display handler, keyboard handler, printer and cassette handler) and are stored in Lo-Byte/ Hi-Byte form.

Locations 58448 to 58495 are jump vectors. Locations 58496 to 58533 are the initial RAM vectors.

SETVBV	58460	E45C	Set system timers routine
SYSVBV	58463	E45F	Stage 1 VBLANK entry
XITVBV	58466	E462	Exit VBLANK entry

Locations 58534 to 59092 are the addresses for the Central Input/ Output routines

Locations 59093 to 59715 are the addresses for the interrupt handler routines

PIRQ	59123	E6F3	Start of IRQ interrupt service routine
SYSVBL	59310	E7AE	Start of the VBLANK routines
PNMI	59316	E7B4	(JSR) interrupt service routine
SETVBL	59629	E8ED	Subroutines to set the VBLANK timers/vectors

Locations 59716 - 60905 are the addresses for the serial Input/Output routines.

Locations 61667 - 62435 are the addresses for the monitor handler routines.

Sources of memory locations:

(1) DE RE ATARI
(2) Technical Users Notes
(3) ANTIC Magazine
(4) Master Memory Map, Santa Cruz Educational Software (1981)
(5) Mapping The Atari, COMPUTE! Publications (1983)

Appendix E

CHARACTER CODES

ATASCII DECIMAL	KEYSTROKE	CHARACTER	INTERNAL CODE
0	CNTRL ,	♥	64
1	CNTRL A	├	65
2	CNTRL B	│	66
3	CNTRL C	┘	67
4	CNTRL D	┤	68
5	CNTRL E	┐	69
6	CNTRL F	╱	70
7	CNTRL G	╲	71
8	CNTRL H	◢	72
9	CNTRL I	▗	73
10	CNTRL J	◣	74
11	CNTRL K	▝	75
12	CNTRL L	▘	76
13	CNTRL M	▔	77
14	CNTRL N	▁	78
15	CNTRL O	▖	79
16	CNTRL P	♣	80
17	CNTRL Q	┌	81
18	CNTRL R	─	82
19	CNTRL S	┼	83
20	CNTRL T	●	84

21	CNTRL U	▄	85
22	CNTRL V	▌	86
23	CNTRL W	┬	87
24	CNTRL X	┴	88
25	CNTRL Y	▌	89
26	CNTRL Z	▙	90
27	ESC ESC	␛	91
28	ESC CNTRL -	↑	92
29	ESC CNTRL =	↓	93
30	ESC CNTRL +	←	94
31	ESC CNTRL *	→	95
32	SPACE BAR		0
33	SHIFT 1	!	1
34	SHIFT 2	"	2
35	SHIFT 3	#	3
36	SHIFT 4	$	4
37	SHIFT 5	%	5
38	SHIFT 6	&	6
39	SHIFT 7	'	7
40	SHIFT 9	(	8
41	SHIFT 0	)	9
42	*	*	10
43	+	+	11
44	,	,	12
45	-	-	13
46	.	.	14

47	/	/	15
48	0	0	16
49	1	1	17
50	2	2	18
51	3	3	19
52	4	4	20
53	5	5	21
54	6	6	22
55	7	7	23
56	8	8	24
57	9	9	25
58	SHIFT;	:	26
59	;	;	27
60	<	<	28
61	=	=	29
62	>	>	30
63	SHIFT /	?	31
64	SHIFT 8	@	32
65	A	A	33
66	B	B	34
67	C	C	35
68	D	D	36
69	E	E	37
70	F	F	38
71	G	G	39
72	H	H	40

73	I	I	41
74	J	J	42
75	K	K	43
76	L	L	44
77	M	M	45
78	N	N	46
79	O	O	47
80	P	P	48
81	Q	Q	49
82	R	R	50
83	S	S	51
84	T	T	52
85	U	U	53
86	V	V	54
87	W	W	55
88	X	X	56
89	Y	Y	57
90	Z	Z	58
91	SHIFT ,	[	59
92	SHIFT +	\	60
93	SHIFT .	]	61
94	SHIFT *	^	62
95	SHIFT -	_	63
96	CNTRL .	◆	96
97	a	a	97
98	b	b	98

99	c	c	99
100	d	d	00
101	e	e	101
102	f	f	102
103	g	g	103
104	h	h	104
105	i	i	105
106	j	j	106
107	k	k	107
108	l	l	108
109	m	m	109
110	n	n	110
111	o	o	111
112	p	p	112
113	q	q	113
114	r	r	114
115	s	s	115
116	t	t	116
117	u	u	117
118	v	v	118
119	w	w	119
120	x	x	120
121	y	y	121
122	z	z	122
123	CNTRL ;	♠ |	123
124	SHIFT =	|	124

125	ESC CNTRL <	↖	125
126	ESC BACK S	◀	126
127	ESC TAB	▶	127

[⅄] = Inverse Video

128	[⅄]CNTRL,	♥	192
129	[⅄]CNTRL A	├	193
130	[⅄]CNTRL B	│	194
131	[⅄]CNTRL C	┘	195
132	[⅄]CNTRL D	┤	196
133	[⅄]CNTRL E	┐	197
134	[⅄]CNTRL F	╱	198
135	[⅄]CNTRL G	╲	199
136	[⅄]CNTRL H	◢	200
137	[⅄]CNTRL I	▗	201
138	[⅄]CNTRL J	◣	202
139	[⅄]CNTRL K	▝	203
140	[⅄]CNTRL L	▘	204
141	[⅄]CNTRL M	▔	205
142	[⅄]CNTRL N	▁	206
143	[⅄]CNTRL O	▖	207
144	[⅄]CNTRL P	♣	208
145	[⅄]CNTRL Q	┌	209
146	[⅄]CNTRL R	─	210
147	[⅄]CNTRL S	┼	211
148	[⅄]CNTRL T	●	212
149	[⅄]CNTRL U	▄	213

150	[Λ]CNTRL V	▐	214
151	[Λ]CNTRL W	┬	215
152	[Λ]CNTRL X	┴	216
153	[Λ]CNTRL Y	▐	217
154	[Λ]CNTRL Z	└	218
155	[Λ]RETURN		219
156	ESC SHIFT BACK S	↑	220
157	ESC SHIFT >	↓	221
158	ESC CNTRL TAB	←	222
159	ESC SHIFT TAB	→	223
160	[Λ]SPACE BAR		128
161	[Λ] SHIFT 1	!	129
162	[Λ] SHIFT 2	"	130
163	[Λ] SHIFT 3	#	131
164	[Λ] SHIFT 4	$	132
165	[Λ] SHIFT 5	%	133
166	[Λ] SHIFT 6	&	134
167	[Λ] SHIFT 7	'	135
168	[Λ] SHIFT 9	(	136
169	[Λ] SHIFT 0	)	137
170	[Λ] *	*	138
171	[Λ] +	+	139
172	[Λ] ,	,	140
173	[Λ] -	-	141
174	[Λ] .	.	142
175	[Λ] /	/	143

176	[^] 0	0	144
177	[^] 1	1	145
178	[^] 2	2	146
179	[^] 3	3	147
180	[^] 4	4	148
181	[^] 5	5	149
182	[^] 6	6	150
183	[^] 7	7	151
184	[^] 8	8	152
185	[^] 9	9	153
186	[^] SHIFT;	:	154
187	[^] ;	;	155
188	[^] <	<	156
189	[^] =	=	157
190	[^] >	>	158
191	[^] SHIFT /	?	159
192	[^] SHIFT 8	@	160
193	[^] A	A	161
193	[^] B	B	162
195	[^] C	C	163
196	[^] D	D	164
197	[^] E	E	165
198	[^] F	F	166
199	[^] G	G	167
200	[^] H	H	168
201	[^] I	I	169

202	[^] J	J	170
203	[^] K	K	171
204	[^] L	L	172
205	[^] M	M	173
206	[^] N	N	174
207	[^] O	O	175
208	[^] P	P	176
209	[^] Q	Q	177
210	[^] R	R	178
211	[^] S	S	179
212	[^] T	T	180
213	[^] U	U	181
214	[^] V	V	182
215	[^] W	W	183
216	[^] X	X	184
217	[^] Y	Y	185
218	[^] Z	Z	186
219	[^] SHIFT ,	[	187
220	[^] SHIFT +	\	188
221	[^] SHIFT .	]	189
222	[^] SHIFT *		190
223	[^] SHIFT -	_	191
224	[^] CNTRL .	♦	224
225	[^] a	a	225
226	[^] b	b	226
227	[^] c	c	227
228	[^] d	d	228

229	[∧] e	e	229
230	[∧] f	f	230
231	[∧] g	g	231
232	[∧] h	h	232
233	[∧] i	i	233
234	[∧] j	j	234
235	[∧] k	k	235
236	[∧] l	l	236
237	[∧] m	m	237
238	[∧] n	n	238
239	[∧] o	o	239
240	[∧] p	p	240
241	[∧] q	q	241
242	[∧] r	r	242
243	[∧] s	s	243
244	[∧] t	t	244
245	[∧] u	u	245
246	[∧] v	v	246
247	[∧] w	w	247
248	[∧] x	x	248
249	[∧] y	y	249
250	[∧] z	z	250
251	[∧] CNTRL ;	♠	251
252	[∧] SHIFT =	|	252
253	ESC CNTRL 2	↰	253
254	[∧] ESC CNTRL BACK S	◀	254
255	[∧] ESC CNTRL >	▶	255

Appendix F

Instructions and Flags

Instruction	Flag						
	B	**N**	**Z**	**C**	**I**	**D**	**V**
ADC	-	x	x	x	-	-	x
AND	-	x	x	-	-	-	-
ASL	-	x	x	x	-	-	-
BIT	-	-	x	-	-	-	-
BRK	1	-	-	-	1	-	-
CLC	-	-	-	0	-	-	-
CLD	-	-	-	-	-	0	-
CLI	-	-	-	-	0	-	-
CLV	-	-	-	-	-	-	0
CMP	-	x	x	x	-	-	-
CPX	-	x	x	x	-	-	-
CPY	-	x	x	x	-	-	-
DEC	-	x	x	x	-	-	-
DEX	-	x	x	-	-	-	-
DEY	-	x	x	-	-	-	-

EOR	-	x	x	-	-	-	-
INC		x	x	-	-	-	
INX		x	x	-	-	-	
INY		x	x	-	-	-	
LDA		x	x	-	-	-	
LDX		x	x	-	-	-	-
LDY		x	x	-	-	-	-
LSR		0	x	x	-	-	-
ORA		x	x	-	-	-	-
PLA		x	x	-	-	-	-
PLP	From Stack						
ROL		x	x	x	-	-	-
ROR		x	x	x	-	-	-
RTI	From Stack						
SBC		x	x	x	-	-	-
SEC		-	-	1	-	-	-
SED		-	-	-	-	1	-
SEI					1		
TAX		x	x	-	-	-	-
TAY		x	x	-	-	-	-
TSX		x	x	-	-	-	-
TXA		x	x	-	-	-	-
TYA		x	x	-	-	-	-

Appendix G

Decimal Values for 6502 Instructions

Mnemonic Code	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	Indexed Indirect	Indirect Indexed	Indirect
ADC		105	101	117		109	125	121			97	113	
AND		41	37	53		45	61	57			33	49	
ASL	10		6	22		14	30						
BCC										144			
BCS										176			
BEQ										240			
BIT			36			44							
BMI										48			
BNE										208			
BPL										16			
BRK									00				
BVC										80			
BVS										112			
CLC									24				
CLD									216				
CLI									88				
CLV									184				

	Acc	Imm	0 page	0 page, X	0 page, Y	Abs	Abs, X	Abs, Y	Implied	Relative	Indexed Indirect	Indexed Indirect	Indirect
CMP		201	197	213		205	221	217			193	209	
CPX		224	228			236							
CPY		192	196			204							
DEC			198	214		206	222						
DEX									202				
DEY									136				
EOR		73	69	85		77	93	89			65	81	
INC			230	246		238	254						
INX									232				
INY									200				
JMP						76							108
JSR						32							
LDA		169	165	181		173	189	185			161	177	
LDX		162	166		182	174		190					
LDY		160	164	180		172	188						
LSR	74		70	86		78	94						
NOP									234				
ORA		9	5	21		13	29	25			1	17	
PHA									72				
PHP									08				
PLA									104				
PLP									40				
ROL	42		38	54		46	62						
ROR	106		102	118		110	126						
RTI									64				
RTS									96				

SBC		233	229	245		237	253	249			225	291	
SEC									56				
SED									248				
SEI									120				
STA			133	149		141	157	153			129	145	
STX			134		150	142							
STY			132	148		140							
TAX									170				
TAY										168			
TSX										186			
TXA										138			
TYA										152			

Appendix H

FREQUENCY VALUES

To Generate Notes Using Registers as Pairs

NOTE	AUDF	LOBYTE	HIBYTE
OCTAVE 0			
C	54720	192	213
C#	51649	193	201
D	48750	110	190
D#	46015	191	179
E	43430	166	169
F	40992	32	160
F#	38691	35	151
G	36519	167	142
G#	34469	165	134
A	32535	23	127
A#	30708	244	119
B	28984	56	113
OCTAVE 1			
C	27357	221	106
C#	25821	221	100
D	24372	52	95
D#	23003	219	89
E	21712	208	84
F	20493	13	80
F#	19342	142	75
G	18257	81	71
G#	17231	79	67
A	16264	136	63
A#	15351	247	59
B	14489	153	56

OCTAVE 2

```
C.................13675..............107............53
C#................12907..............107............50
D.................12182..............150............47
D#................11498..............234............44
E.................10852..............100............42
F.................10243................3............40
F#.................9668..............196............37
G..................9125..............165............35
G#.................8612..............164............33
A..................8128..............192............31
A#.................7672..............248............29
B..................7241...............73............28
```

OCTAVE 3

```
c..................6834..............178............26
C#.................6450...............50............25
D..................6088..............200............23
D#.................5746..............114............22
E..................5423...............47............21
F..................5118..............254............19
F#.................4830..............222............18
G..................4559..............207............17
G#.................4303..............207............16
A..................4061..............221............15
A#.................3832..............248............14
B..................3617...............33............14
```

OCTAVE 4

```
C..................3413...............85............13
C#.................3222..............150............12
D..................3040..............224............11
D#.................2869...............53............11
E..................2708..............148............10
F..................2555..............251.............9
F#.................2412..............108.............9
G..................2276..............228.............8
G#.................2148..............100.............8
A..................2027..............235.............7
A#.................1913..............121.............7
B..................1805...............13.............7
```

OCTAVE 5

C	1703	167	6
C#	1607	71	6
D	1517	237	5
D#	1431	151	5
E	1350	70	5
F	1274	250	4
F#	1202	178	4
G	1134	110	4
G#	1070	46	4
A	1010	242	3
A#	962	194	3
B	899	131	3

OCTAVE 6

C	848	80	3
C#	800	32	3
D	755	243	2
D#	712	200	2
E	672	160	2
F	634	122	2
F#	598	86	2
G	564	52	2
G#	532	20	2
A	501	245	1
A#	473	217	1
B	446	190	1

OCTAVE 7

C	421	165	1
C#	397	141	1
D	374	118	1
D#	353	97	1
E	332	76	1
F	313	57	1
F#	295	39	1
G	278	22	1
G#	262	6	1
A	247	247	0
A#	233	233	0
B	219	219	0

```
OCTAVE 8

C.....................207..............207..............0
C#....................195..............195..............0
D.....................183..............183..............0
D#....................173..............173..............0
E.....................163..............163..............0
F.....................153..............153..............0
F#....................144..............144..............0
G.....................136..............136..............0
G#....................128..............128..............0
A.....................120..............120..............0
A#....................113..............113..............0
B.....................106..............106..............0
```

Index

More Computer Books from Weber Systems Inc.

Instant Lotus 1-2-3	$ 7.95
Turbo Pascal: Programming Ex. & Subroutines	$15.95
Turbo Prolog Programmer's Guide	$17.95
Amiga Programmer's Guide	$19.95
Atari ST Programmer's Guide	$19.95
Excel: Using Macros	$19.95
Working With Q & A	$19.95
Commodore 128 Programmer's Guide	$17.95
dBASE III Plus: Things the Manual Didn't Tell You	$19.95
Working with MS-DOS v. 3-2	$19.95
Working with Reflex	$19.95
Microsoft Basic 3rd Edition	$17.95
Atari XE User's Handbook	$15.95
Atari XL User's Handbook	$17.95
IBM PC AT User's Handbook	$ 9.95
Okidata Printer User's Handbook	$15.95
Sanyo MBC 550/555 User's Handbook	$17.95
AppleWorks User's Handbook	$ 9.95
Working With Multimate	$17.95
The Applewriter II Handbook	$19.50
The WordStar Handbook	$19.95
Lotus 1-2-3 Models	$15.95
Symphony Command Language Programmer Guide	$17.95
Supercalc^3 Models	$15.95
All About 1-2-3	$ 5.95
1-2-3 Managerial Worksheets	$ 7.95
Controlling Financial Performance for Higher Profits	$17.50
Planning and Budgeting for Higher Profits: 1-2-3	$17.50
Business Problem Solving: 1-2-3	$19.95
Analyzing Your Financial Statements with Symphony: 1-2-3	$19.95
Business Decision Making for Higher Profits: 1-2-3	$19.50
Business Graphics for Analysis & Presentation: 1-2-3	$19.95
How to Get Started with MS-DOS	$ 7.95
CP/M Simplified	$13.95
Working with PICK	$19.95
The World of PC-DOS	$ 7.95

Inventory Management for Small Computers	$ 5.95
Programs for Electronic Circuit Design	$14.95
The Model 100 Program Book	$12.95
IBM PC Business Software	$17.95
Sanyo MBC Business Software	$11.95
32 BASIC Programs for the Apple	$ 5.95
More Than 32 BASIC Programs for the IBM PC	$ 5.95
32 BASIC Programs for Atari 800XL and 130XE	$ 5.95
Computers, Teaching and Learning	$12.95
Electronic Notebook	$ 9.95
Computers for Everybody 3rd Edition	$ 4.95
Bit, Bytes, and Buzzwords	$ 3.95
Atari Assembly Language Programmer's Guide	$14.95
6502, 65C02, and 65816 Handbook	$14.95
Inside the TRS-80 Model 100	$19.95
Baron's Used Computer Price Guide	$ 7.95
Speech Systems for Your Microcomputer	$14.95
Logo and More for the Commodore 64/128	$ 7.95
The Programmer's Toolbox	$ 9.95
Sanyo BASIC User's Handbook	$17.95
Nailing Jelly to a Tree	$ 7.95
Shaping Logo on Your Apple	$ 7.95
How to Build a Program	$ 9.95
Beyond Beginning BASIC	$ 7.95
Beginning FORTRAN	$15.95
BASIC: Step-by-Step Programming	$15.95
Beginning FORTH	$16.95
Beginning COBOL	$ 9.95
Introduction to Structured FORTRAN	$19.95
Lisp	$15.95
Pascal: Step-by-Step Programming	$16.95
Introduction to C	$19.95
Introduction to Modula 2	$19.95
Introduction to Ada	$19.95
Turbo Pascal Programmer's Guide	$17.95
Xenix User's Handbook	$17.95
THEOS/OASIS User's Handbook	$17.95

Title	**Price**	**Quantity**	**Total**
Catalogue			NC

Book Subtotal ______

Shipping & Handling ($2.50 per item) ______

Ohio Residents Add 6½% Sales Tax ______

GRAND TOTAL ______

Method of Payment: ☐ Check ***Charge my:*** ☐ VISA ☐ Master Card ☐ American Express

Card Number ______ Exp. Date ______

Cardholder Name ______

Ship To ______

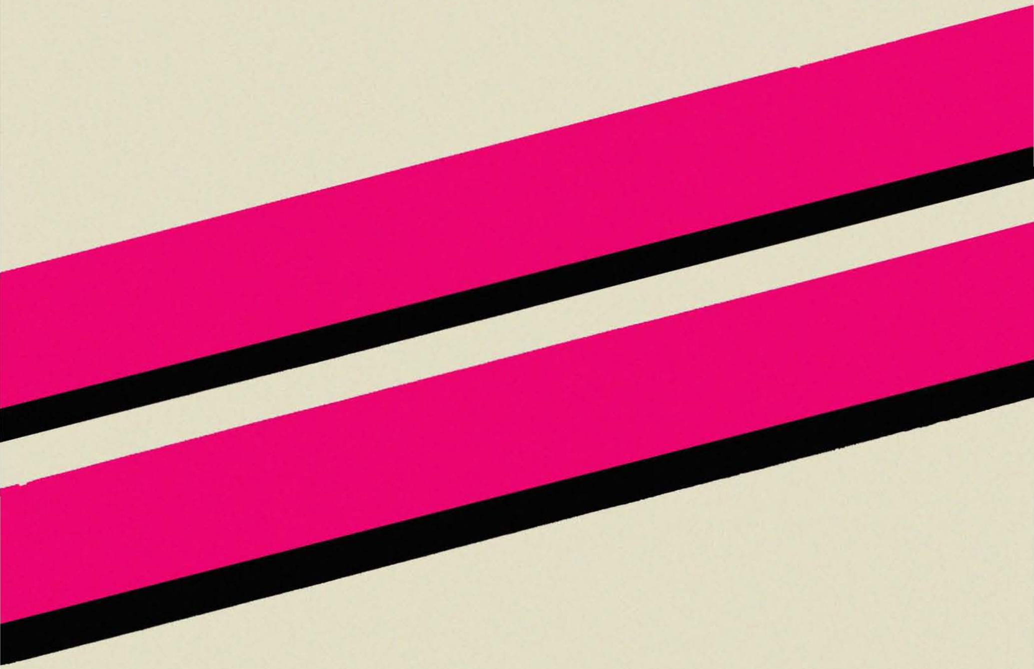
Name ______

Address ______

City ______ State ______ Zip ______

Send to: Weber Systems Inc., 8437 Mayfield Road, Chesterland, OH 44026
Toll-free order number 800-851-6018

$19.95 US
$28.95 Canada



ISBN 0-938862-54-5

WEBER SYSTEMS, INC.
8437 Mayfield Road, Chesterland, Ohio 44026