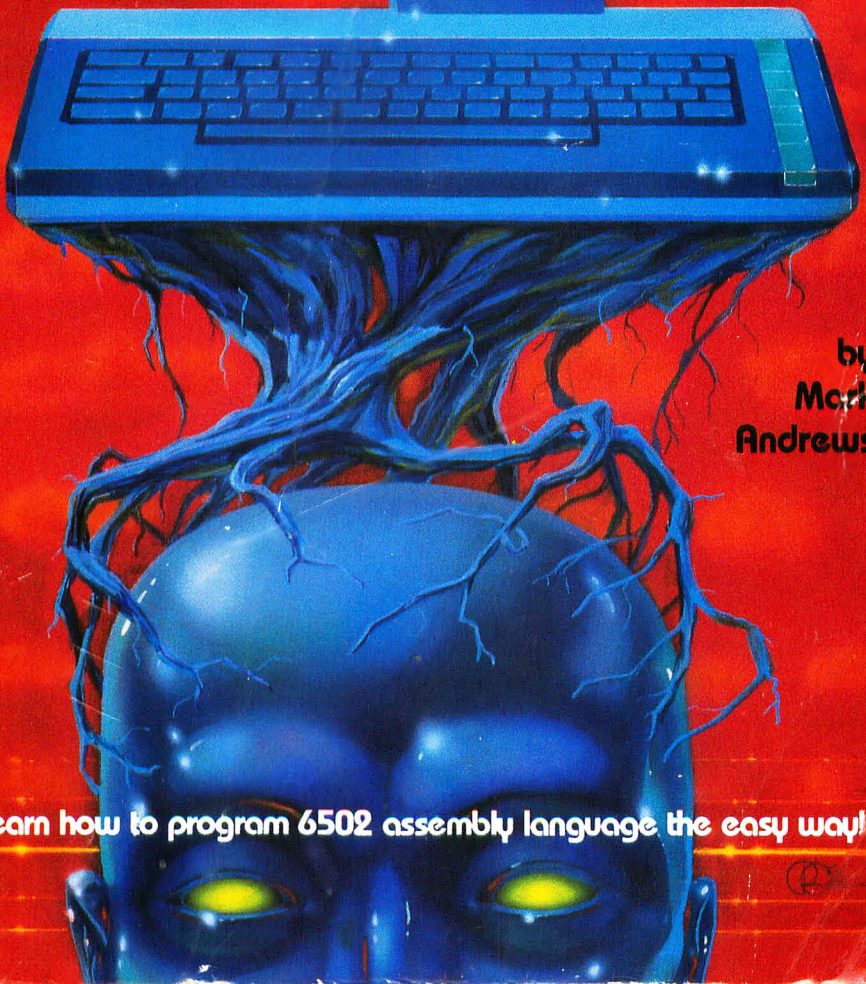
Reston Publishing Company Inc

$14.95

ATARI ROOTS

A GUIDE TO ATARI ASSEMBLY LANGUAGE



by Mark Andrews

Learn how to program 6502 assembly language the easy way!

ATARI ROOTS

ATARI ROOTS

by

Mark Andrews

Illustrated by

Tes Bonnarith

DATAMOST INC.

20660 Nordhoff Street, Chatsworth, CA 91311-6152

(818) 709-1202

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

ISBN 0-8359-0130-0

Printed in U.S.A.

To Maureen

Table of Contents

Preface

Many books have been written about 6502 assembly language. But this book is different in several important ways. Until now, there have been two main kinds of books about 6502 assembly language. There are generic books that make no mention of Atari computers, and there are reference books that are full of information about Atari computers, but are so technical that only experts can understand them. But there was a distinct shortage of books designed to teach Atari users who knew a little BASIC how to start programming in assembly language.

This book was written to fill that void. It's written in English, not computerese. It's written for Atari users, not for professional programmers (though they might find it useful). Every major topic that it covers is illustrated with at least one (and often more) simple, informative, useful programs. This book is also unique in several other ways. It's the first assembly language guidebook that has been user-tested on Atari's new XL series computers (and every program in it will work on Atari's older computers, too). It's the first book to cover the operation of the OSS MAC/65 assembler, one of the most popular Atari assemblers on the market, as well as the operation of the Atari Assembler Editor cartridge. And no matter what kind of computer or assembler you own, the book you're now reading is probably the easiest to understand assembly language textbook you'll ever own. In this book you'll find — for the first time between book covers — everything you'll need to know to start programming in Atari assembly language, and to start running your programs on an Atari computer system. Best of all, this book will have you writing assembly language programs before you know it, and by the time you finish, you'll be well on your way to becoming an expert assembly language programmer.

All you'll need is this book, an Atari computer (any Atari computer), and a few other supplies. They are:

- A machine language assembler and debugger. The programs in this book will work without any changes on either a MAC/65

assembler from Optimized Systems Software (OSS) of San Jose, CA, or the Atari Assembler Editor cartridge manufactured by Atari. If you own another kind of assembler, you can probably use it without too much difficulty, since there is a standard instruction set in 6502 assembly language.

There are differences in assemblers, though, just as there are in the dialects of BASIC used by various BASIC interpreters. So if you do use an assembler other than the two that were used to write the programs in this book, you may have to make some alterations in the way the programs have been written, assembled, debugged, loaded, saved and run. And I don't recommend this unless you already know how to program in assembly language.

Once you own this book and an assembler, you'll need only a few other items to start programming in Atari assembly language. These items are:

- An Atari BASIC cartridge and BASIC Reference Manual.
- An Atari or Atari-compatible 5¼ inch floppy disk drive (or, better still, two disk drives).
- An Atari or Atari-compatible line printer (any kind that works — 40-column or 80-column, thermal or impact, dot matrix or letter quality; it doesn't matter).

As you learn more about assembly language, you may also want to buy more books about Atari computers and Atari assembly language programming. You'll find a few of the best books on both of those subjects listed in a short bibliography at the end of this volume. Good luck, and happy programming!

Mark Andrews
New York, NY
February, 1984

Introduction

If your Atari doesn't understand you, maybe it's because you don't speak its language. Together, we're going to break that language barrier. This book will teach you how to write programs in assembly language — the fastest running, most memory efficient of all programming languages. This book will also give you a good working knowledge of machine language, your computer's native tongue. It will enable you to create programs that would be impossible to write in BASIC or other less advanced languages. And it will prove to you that programming in assembly language is not nearly as difficult as you may have thought it would be.

What's in Store

If you know BASIC — even a little BASIC — you can learn to program in assembly language; and once you know assembly language, you'll be able to do many other things, such as:

- Write programs that will run 10 to 1000 times faster than programs written in BASIC.
- Use up to 128 colors on your video screen simultaneously.
- Custom design your own screen displays, mixing text and graphics in any way you like.

You'll also be able to:

- Create your own customized character sets.
- Design animated screen displays using both player-missile graphics and character animation techniques.
- Use both fine and coarse horizontal and vertical scrolling in your programs.

And you'll even discover how to:

- Create sound effects that are too complex to be programmed in BASIC.

- Use graphics modes that BASIC does not support.
- Write programs that will boot from a disk and run automatically when you turn your computer on.

In other words, once you learn how to program in assembly language, you'll be able to start writing programs using the same kinds of techniques that professional Atari programmers use. Many of those techniques are downright impossible without a knowledge of assembly language. Finally, and even more important, as you learn assembly language, you'll also be learning what makes computers tick. And that will make you a better programmer in any language.

Assembly Language Demystified

This book has been carefully tailored to take the drudgery out of learning assembly language. It's packed with sample programs and routines. It even contains a selection of interactive tutorial programs, written in Atari BASIC, that are especially designed to help you learn assembly language.

Chapter 1 will introduce you to assembly language and explain the differences between assembly language and other programming languages.

In Chapter 2 you'll start learning about bits, bytes and binary numbers; the building blocks that program designers use to create assembly language programs. You'll even find some easy to use BASIC programs that will automatically perform hexadecimal and binary conversions, and will help take the mystery out of hex and binary numbers.

In Chapter 3 we'll start probing the mysteries of the 6502 microprocessor chip, the heart (or, perhaps more accurately, the brain) of your Atari computer.

In Chapter 4 you'll start writing assembly language programs. And by the time you finish this book, you'll be well on your way to becoming an accomplished assembly language programmer.

Chapter One

Introducing Assembly Language

Start programming immediately in machine language! Turn on your Atari computer and type in this program. Then run it, type a few words, and you'll see something very interesting on your computer screen.

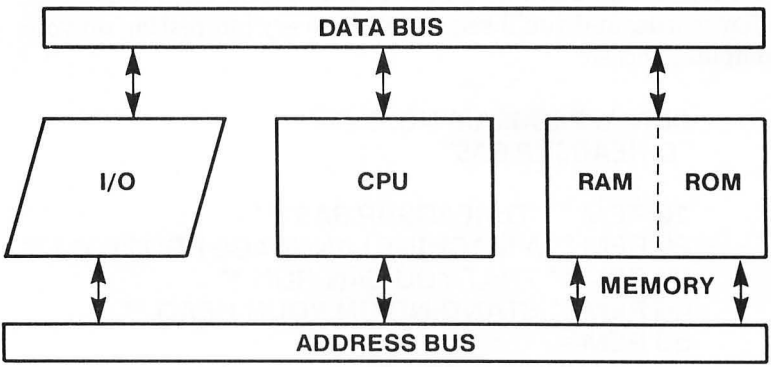
BONUS PROGRAM NO. 1
"D:HEADSUP.BAS"

```
10 REM ** "D:HEADSUP.BAS" **
20 REM ** A MACHINE LANGUAGE PROGRAM **
30 REM ** THAT YOU CAN RUN **
40 REM ** STANDING ON YOUR HEAD **
50 REM
60 GRAPHICS 0:PRINT
100 POKE 755,4
110 OPEN #1,4,0,"K:"
120 GET #1,K
130 PRINT CHR$(K);
140 GOTO 120
```

This is, of course, a BASIC program. Line 60 clears your computer screen with a GRAPHICS 0 command. Line 110 opens the Atari keyboard as an input device. Then, in lines 120 through 140, there is a loop that prints typed-in characters on your screen. But the most important line in this program, the line that makes it do what it's supposed to do, is line 100. The active ingredient of line 100, the instruction POKE 755,4, is actually a machine language instruction. In fact, all POKE commands in BASIC are machine language instructions. When you use a POKE command in BASIC, what you're actually doing is storing a number in a specific memory location in your computer. And when you store a number in a specific memory location in your computer, what you're doing is using machine language.

Under the Hood of Your Atari

Every computer has three main parts: a *Central Processing Unit (CPU)*, *memory* (usually divided into two blocks called *Random Access Memory [RAM]* and *Read Only Memory [ROM])*, and *Input/Output (I/O) devices.*



Your Atari's main input device is its *keyboard.* Its main output device is its *video monitor.* Other I/O devices that an Atari computer can be connected to (or *interfaced* with) include *telephone modems, graphics tablets, cassette data recorders,* and *disk drives.* In a microcomputer, all of the functions of a central processing unit are contained in a *MicroProcessor Unit* (or *MPU*). Your Atari computer's MPU, as well as its CPU (*Central Processing Unit*), is a circuit using *Large Scale Integration (LSI)* called a *6502 microprocessor.*

The 6502 Family

The 6502 microprocessor, your computer's command center, was developed by MOS Technology, Inc. Several companies are now licensed to manufacture 6502 chips, and a number of computer

manufacturers use the 6502 processor in their machines. The 6502 chip and several updated models, such as the 6502A and the 6510, are used not only in Atari computers, but also in personal computers manufactured by Apple, Commodore, and Ohio Scientific. That means, of course, that 6502 assembly language can also be used to program many different personal computers — including the Apple II, Apple II+, Apple //e and Apple ///; all Ohio Scientific computers; the Commodore PET computer, and the Commodore 64. And that's not all; the *principles* used in Atari assembly language programming are universal; they're the same principles that assembly language programmers use, no matter what kind of computers they're writing programs for. Once you learn 6502 assembly language, it will be easy to learn to program other kinds of chips, such as the Z-80 chip used in Radio Shack and CP/M based computers, and even the powerful newer chips that are used in 16-bit microcomputers such as the IBM-PC.

The Fountains of ROM

Your computer has two kinds of memory: Random Access Memory (RAM) and Read Only Memory (ROM). ROM is your Atari's long-term memory. It was installed in your computer at the factory, and it's as permanent as your keyboard. Your computer's ROM is permanently etched into a certain group of chips, so it never gets erased, even when the power is turned off. For most home computer owners, that's a good thing. Without its ROM, your Atari wouldn't be an Atari. In fact, it wouldn't be much more than an expensive, high tech doorstop. The biggest block of memory in ROM is the block that holds your computer's *Operating System*, or *OS*. Your Atari's operating system is what enables it to do all of those wonderful things that Ataris are supposed to do, such as accepting inputs from the keyboard, displaying characters on the screen, and so on. ROM is also what enables your computer to communicate with peripherals such as disk drives, cassette recorders, and telephone modems. If you own one of Atari's XL series of computers, your unit's ROM package also contains a number of added features, such as a built-in self-diagnostic system, a built-in foreign language character set, and built-in BASIC.

RAM is Fleeting

ROM, as you can imagine, was not built in a day. Your Atari's ROM package is the result of a lot of work by a lot of assembly language programmers. RAM, on the other hand, can be written by anybody — even you. RAM is your computer's main memory. It has a lot more memory cells than ROM does, but RAM, unlike ROM, is fleeting. The trouble with RAM is that it's erasable, or, as a computer engineer might put it, *volatile*. When you turn your computer on, the block of memory inside it that's reserved for RAM is as empty as a blank sheet of paper. And when you turn your computer off, anything you may have in RAM disappears. That's why most computer programs have to be loaded into RAM from mass storage devices such as cassette data recorders and disk drives. After you've written a program, you have to store it somewhere so it won't be erased when the power goes off and erases your RAM.

Your computer's RAM, or main memory, can be visualized as a huge grid made up of thousands of compartments, or cells, something like tiers upon tiers of post office boxes along a wall. Each cell in this vast memory matrix is called a memory location, and each *memory location*, like each box in a post office, has an individual and unique *memory address*. The analogy between computers and post office boxes doesn't end there. A computer program, like an expert postal worker putting mail in post office boxes, can get to any location in its memory about as quickly as it can get to any other. In other words, it can access any location in its memory at random. And that's why user-addressable memory in a computer is known as *random access memory*.

Its "Letters" are Numbers

Our post office analogy isn't absolutely perfect, however. A post office box can be stuffed full of letters, but each memory location in a computer's memory can hold only one number. And that number can represent only one of three things:

1. The stored number itself;
2. A code representing a typed character; or
3. A machine language instruction.

What Next?

When a computer goes to a memory location and finds a number, it must be told what to do with the number it finds. If the number equates to just a number, then the computer must be told why the number is there. If the number is a code representing a typed character, then the computer must be told how the character is to be used. And if the number is to be interpreted as a machine language instruction, the computer must be told that, too.

Its Instructions are Programs

The instructions that computers are given so that they can find and interpret the numbers stored in their memories are called *computer programs*. People who write programs are, of course, called *programmers*. The languages that programs are written in are called *programming languages*. Of all the programming languages assembly language is the most comprehensive.

Running a Machine Language Program

When your computer runs a program, the first thing it has to be told is where the program has been stored in its memory. Once it has that information, it can go to the memory address where the program begins and take a look at what's there. If the computer finds an instruction that it's programmed to understand, then it will carry out that instruction. The computer will then move on to the next address in its memory. After it follows the instruction it finds there, it will move on to the next address, and so on. The computer will repeat this process of carrying out an instruction and moving on to the next one until it reaches the end of whatever program has been stored in its memory. Then, unless it encounters an instruction to return to an address within the program or to jump to a new address, it will simply sit there, patiently waiting to receive another instruction.

Computer Languages

As you know, programs can be written in dozens of computer languages such as BASIC, COBOL, Pascal, LOGO, and so on. Languages like these are called *high level languages*, not because they're particularly esoteric or profound, but because they're written at too high a level for a computer to understand. A computer can actually understand only one language, *machine language*, which is written entirely in numbers. So before a computer can run a program written in a high level language, the program must somehow be translated into machine language.

Programs written in high level languages are usually translated into machine language using software packages called *interpreters* and *compilers*. An interpreter is a piece of software that can convert a program into machine language as it is being written. Your Atari BASIC interpreter is a high level language interpreter. Interpreters can also be used to convert a few other high level languages, such as LOGO and Pilot, into machine language. A compiler is a software package designed to convert high level languages into machine language *after* they are written. COBOL, Pascal and most other high level languages are usually translated into machine language with the help of compilers.

Machine Language Assemblers

Interpreters and compilers are not used in writing assembly language programs. Assembly language programs are almost always written with the aid of software packages called *assemblers*. A number of other assemblers for Atari computers are available, including Atari's very advanced Macro Assembler and Text Editor package. An assembler doesn't work like an interpreter, or like a compiler. That's because assembly language is not a high level language. One could say, in fact, that assembly language is not really a programming language at all. Actually, assembly language is nothing more than a *notation system* used for writing machine language programs using alphabetical symbols that human programmers can understand.

What we're trying to get across here is the fact that assembly language is totally different from every other programming language. When a high level language is translated into machine language by an interpreter or a compiler, one instruction in the original programming language can easily equate to dozens — sometimes even hundreds — of machine language instructions. When you write a program in assembly language, however, *every assembly language instruction that you use equates to just one machine language instruction with exactly the same meaning.* In other words, *there is an exact one-to-one relationship between assembly language instructions and machine language instructions.* Because of this one-to-one correspondence, machine language assemblers have a much easier job than interpreters and compilers have.

Since assembly language programs (often called *source code*) can be converted directly into machine language programs (often known as *object code*), an assembler can just zip right along, turning source code listings into object code without having to struggle through any of the tortuous translation contortions that interpreters have to face each time they carry out their appointed rounds. Assemblers also have one other advantage over compilers. The programs that they produce tend to be more straightforward and less repetitious. Assembled programs are more memory efficient and run faster than interpreted and compiled programs.

The Programmer's Plight

Unfortunately, a price has to be paid for all of this efficiency and speed; and the individual who pays that price is, sadly enough, the assembly language programmer. Ironically, even though assembly language programs run much faster than programs written in high level languages, they require many more instructions and take much longer to write. One widely quoted estimate is that it takes an expert programmer about ten times as long to write an assembly language program than it would take him (or her) to write the same program in a high level language such as BASIC, COBOL, or Pascal. On the other hand, assembly language programs run 10 to 1000 times faster than BASIC programs,

and can do things that BASIC programs can't do at any speed. So if you want to become an expert programmer, you really have no choice but to learn assembly language.

How Machine Language Works

Machine language, like every other computer language, is made up of *instructions*. As we have pointed out, however, every instruction used in machine language is a number. The numbers that computers understand are not the kind that we're accustomed to using. Computers think in *binary numbers* — numbers that are nothing but strings of ones and zeros. Here, for example, is part of an actual computer program written in binary numbers (the kind of numbers that a computer understands):

```
00011000
11011000
10101001
00000010
01101001
00000010
10000101
11001011
01100000
```

It doesn't take much imagination to see that you'd be in for quite a struggle if you had to write long programs, which typically contain thousands of instructions, in binary style machine language. With an assembler, however, the job of writing a machine language program is considerably easier. Here, for example, is the above program as it would appear if you wrote it in assembly language:

```
CLC
CLD
LDA
#02
ADC
#02
STA
$CB
RTS
```

You may not understand all of that yet, but you'll have to admit that it at least *looks* more comprehensible. What this program does, by the way, is add 2 and 2. Then it stores the result of its calculation in a certain memory location in your computer — specifically, memory address 203. Later on we'll come back to this program and take a closer look at it. Then you'll get a chance to see exactly how it works. First, though, we're going to go into a little more detail about assemblers and assembly language.

Assembly Language and BASIC Compared

Assembly language is written using three-letter instructions called *mnemonics*. Some mnemonics are quite similar to BASIC instructions. One assembly language instruction that's much like a BASIC instruction is RTS, the last instruction in the sample routine we just looked at. RTS (written 0110 0000 in machine language) means "ReTurn from Subroutine." It's used much like the RETURN instruction in BASIC. There's also an assembly language mnemonic that's similar to BASIC's GOSUB instruction. It's written JSR, and means "Jump to SuBroutine." Its equivalent in binary coded machine language is 0010 0000.

Not all assembly language instructions bear such a close resemblance to BASIC instructions, however. An assembly language instruction never tells a computer to do something as complex as draw a line or print a letter on a screen, for example. Instead, most assembly language mnemonics instruct computers to carry out very elementary tasks such as adding two numbers, comparing two pieces of data, or (as we have seen) jumping to a subroutine. That's why it often takes vast numbers of assembly language instructions to equal just one or two words in a high level language.

Source Code and Object Code

When you write an assembly language program, the listing that you produce is called *source code*, since it's the source from which a machine language program will be produced. Once you've written an assembly language program in source code, you can

run it through an assembler. The assembler will then convert it into *object code*, which is just another name for a machine language program produced by an assembler.

The Speed and Efficiency of Machine Language

Since assembly language instructions are so specific (you might even say primitive) it obviously takes lots of them to make up a complete program; many, many more instructions than it would take to write the same program in a high level language. Ironically, machine language programs still take up less memory space than programs written in high level languages do. That's because when a program written in a high level language is interpreted or compiled into machine language, big blocks of machine code must be repeated every time they are used. But in a well-written assembly language program, a routine that's used over and over can be written just once, and then addressed as many times as needed with JSR, RTS, and similar commands. Many other kinds of techniques can also be used to conserve memory in assembly language programs.

Chapter Two

Bits, Bytes, and Binary

A one or a zero might not mean much to you, but to a computer it means quite a bit. Binary numbers, as you may already know, are numbers made up solely of ones and zeros. And they're the only kind of numbers that a computer can understand. A computer, or at any rate, a digital computer, which is what your Atari is, has great difficulty conceiving of concepts in shades of gray. To a computer, a switch is on or it's off. An electrical signal is there or it isn't. Everything in a microcomputer's small mind is black or white, plus or minus, off or on.

Call it what you like. It doesn't matter. It's male and female, Shiva and Shakti, yin and yang. Mathematicians sometimes call it Boolean algebra. Computer designers sometimes call it two-state logic. And programmers often refer to it as the binary system. In the binary system, the digit 1 symbolizes the positive, a current that's flowing, for instance, or a switch that's on. The digit 0 represents the negative, a current that's not flowing, or a switch that's off. But there is no digit for the number 2. The only way to represent the number 2 in binary is to take a 1, move it one space to the left, and follow it with a 0, like this: 10. That's right. In binary notation, "10" means 2, not 10, "11" means 3, "100" is 4, "101" is 5, and "110" is 6, and so on.

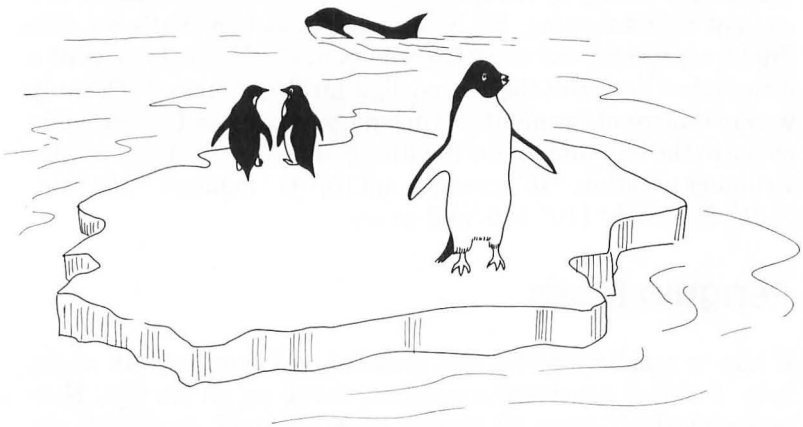
Penguin Math

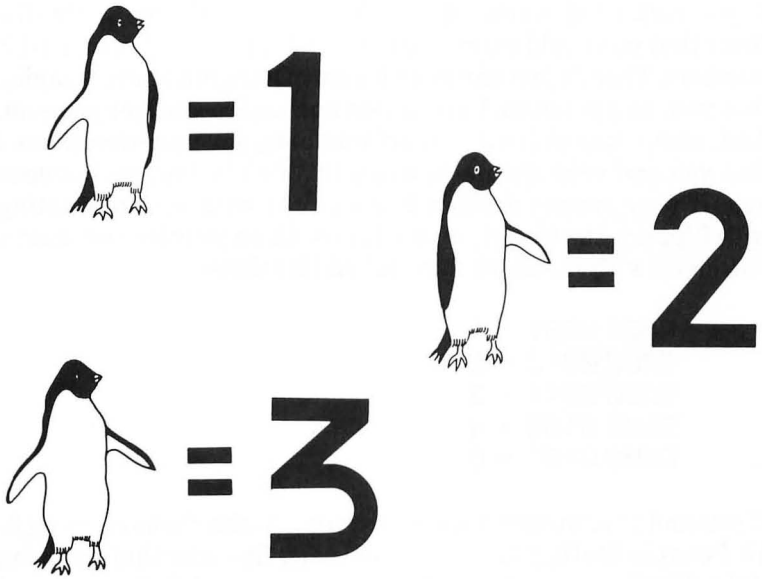
If binary numbers baffle you, a course in Penguin Math might help. Imagine you were a penguin, living on an ice floe. Now penguins don't have 10 fingers on each hand, as people do. Instead, they have two flippers. So if you were a penguin, and counted on your flippers like some people count on their fingers, you'd be able to count only to 2. If you were a very bright penguin, however, you might one day figure out how to use your flippers to count past 2. Suppose, for example, that you decided to equate a

raised right flipper to 1, and a raised left flipper to 2. Then you could let *two raised flippers* be equal to 3. Now suppose you were an extraordinarily bright penguin, and devised a notation system to express in writing what you had done. You could use a 0 to represent an unraised flipper, and a 1 to represent a raised one. And then you could scratch these equations in the ice:

Penguin Numbers

00 = 0
01 = 1
10 = 2
11 = 3





Those, of course, are binary numbers. And what they show, in Penguin Math, is that you can express four values — 0 through 3 — as a 2-bit (two-digit) binary number. Now let's suppose that you, as a penguin, wanted to learn to count past 3. Let's imagine that you looked down at your feet and noticed that you had two more flippers. Voila, bigger numbers! If you sat down on your ice floe so that you could raise both arms and both legs at the same time, you could count as follows:

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

. . . and so on.

If you continued counting like this you would eventually discover that you could express 16 values, 0 through 15, using 4-bit numbers. There's just one more lesson in Penguin Math. Imagine that you, as a penguin, have gotten married to another penguin. And, using your skill with binary numbers, you have determined that you and your spouse have a total of eight flippers between you. If your spouse decided to cooperate with you in counting with flippers, the two of you could now sit on your ice and start a floe chart with numbers that looked like these:

0000 0001 = 1
0000 0010 = 2
0000 0011 = 3
0000 0100 = 4
0000 0101 = 5

If you and your spouse kept on counting in this fashion, using 8-bit Penguin Math, you would eventually discover that by using eight flippers you could could count from 0 to 255, for a total of 256 values. That completes our brief course in Penguin Math. What it has taught us is that it is possible to express 256 values, from 0 through 255, using 8-bit binary numbers.

Bits, Bytes and Nybbles

As was pointed out a few paragraphs back, when ones and zeros are used to express binary numbers, they are known as bits. A group of eight bits is called a *byte*. And a group of four bytes is called a *nibble* (sometimes spelled "*nybble*"). And a group of 16 bits is called a *word*. Now we're going to take another look at a series of 8-bit bytes. Observe them closely, and you'll see that every binary number that ends in zero is twice as large as the previous round number; or, in other words, is the square of the previous round number:

0000 0001 = 1
0000 0010 = 2
0000 0100 = 4
0000 1000 = 8
0001 0000 = 16

0010 0000 = 32
0100 0000 = 64
1000 0000 = 128

Here are two more binary numbers that assembly language programmers often find worth memorizing:

11111111 = 255
11111111 11111111 = 65,535

The number 255 is worthy of note because it's the largest 8-bit number. The number 65,535 is the largest 16-bit number. Now the plot thickens. As we have mentioned, Atari computers are called 8-bit computers because they're built around an 8-bit microprocessor, a computer processor chip that handles binary numbers up to eight places long, but no longer. Because of this limitation, your Atari can't perform calculations on numbers larger than 255, in fact it can't even perform a calculation with a *result* that's greater than 255!

Obviously, this 8-bit limitation places severe restrictions on the ability of Atari computers to perform calculations on large numbers. In effect, the 6502's *Arithmetic Logic Unit* (ALU) is like a calculator that can't handle a number larger than 255. There are ways to get around that limitation, of course, but it isn't easy. To work with numbers larger than 255, an 8-bit computer has to perform a rather complex series of operations. If a number is greater than 255, an 8-bit computer has to break it down into 8-bit chunks, and perform each required calculation on each 8-bit number. Then the computer has to patch all of these 8-bit numbers back together again.

If the result of a calculation is more than eight bits long, things get even more complicated. That's because each *memory location* in an 8-bit computer, each cell in its Random Access Memory (RAM) as well as its Read Only Memory (ROM), is an 8-bit memory register. So if you want to store a number larger than 255 in an 8-bit computer's memory, you have to break it up into two or more 8-bit numbers, and then store each of those numbers in a separate memory location. And then, if you ever want to use the original number again, you have to patch it back together from the 8-bit pieces it was split into.

8-bit Versus 16-bit Computers

Now that you know all that, you can understand why 16-bit computers, such as the IBM Personal Computer and its many imitators, can run faster than 8-bit computers can. A 16-bit computer can handle binary numbers up to 16 bits long without doing any mathematical cutting and pasting, and can therefore process numbers ranging up to 65,535 in single chunks, a 16-bit word at a time. So 16-bit computers are considerably faster than 8-bit computers are, at least when they're dealing with large numbers. 16-bit computers also have another advantage over 8-bit computers. They can keep track of much more data at one time than 8-bit computers can. And they can therefore be equipped with much larger memories.

Your Computer's Memory Map

A computer's memory can be visualized as a huge grid containing thousands of pigeonholes, or memory locations. In an Atari computer, each of those locations can hold one 8-bit number. Earlier, we spoke of an analogy between a computer's memory and tiers upon tiers of of post office boxes. Now we can extend that analogy a little further.

As we've mentioned, each memory location in your computer has an individual and unique memory address. And each of these addresses is actually made up of two index numbers. The first of these numbers could be called the X axis of the location of the address on your computer's memory grid. The second number could be called the location's Y axis. By using this kind of X,Y coordinate system for locating addresses in its memory, your Atari computer can keep track of addresses that are up to 16 bits long, even though it's only an 8-bit computer. All it has to do is keep the X axis in one 8-bit register and the Y axis in another. So when you want your Atari to fetch a piece of data from its memory, all you have to do is provide it with the X axis and the Y axis of the address of the data. The computer can then immediately get the data you're looking for.

Your Computer's Address Book

But there's still a limit to the number of address locations that an 8-bit computer can keep track of. Since 255 is the largest number that an 8-bit register can hold, the X axis of an address can range only from 0 to 255, a total of 256 numbers. The same limit applies to the Y axis. So unless certain fancy memory expansion tricks are used, the maximum memory capacity of an 8-bit computer is 256 times 256, or 65,536; in other words, 64K. The reason for the odd number, incidentally, is that 64K equates to a binary number, not a decimal number. 64K is not equal to decimal 64 times decimal 1,000, but is equivalent instead to the product of 64 and 1024. Those two numbers look like they were pulled out of a hat when they're written in decimal notation, but in binary they're both nice round numbers: 0100 0000 and 0100 0000 0000, respectively.

What a Difference 8 Bits Make

If a 16-bit computer kept track of the addresses in its memory the same way an 8-bit computer does, by using the X axis and the Y axis of each location as reference points, then a 16-bit computer could address more than 4 million memory locations (65,536 cells by 65,536 cells). But 16-bit computers don't usually keep track of the addresses in their memories that way. In the IBM-PC for example, each memory location is assigned what amounts to a 20-bit address. So an IBM-PC can address 1,048,576 address locations; not quite the 4 million plus locations that it could address with an X,Y matrix system, but still enough locations to hold more than a million bytes (a *megabyte*) of memory.

Now you can understand what all of the fuss over 16-bit computers is all about. 16-bit computers can address more memory than 8-bit computers can, and can also process information faster. In addition, they're easier to program in assembly language than 8-bit computers are, since they can digest chunks of data that are 16 bits long. Since Atari computers are 8-bit computers, none of this talk about 16-bit computers is likely to help you much in your quest for knowledge about Atari assembly language. Fortunately, however, a knowledge of Atari assembly language *will* help you a great deal if you ever decide to study a 16-bit assembly

language. If you can learn to juggle bits well enough to become a good Atari assembly language programmer, you'll probably find that 16-bit assembly language, which doesn't require any splicing together of 8-bit numbers, is a snap to learn.

The Hexadecimal Number System

What's the sum of D plus F? Well, it's 1C if you're working in hexadecimal numbers.

Hexadecimal numbers, as you may know if you've done much programming, are strange looking combinations of letters and numbers that are often used in assembly language programs. The hexadecimal notation system uses not only the digits 0 through 9, but also the letters A through F. So weird looking letter and number combinations like FC3B, 4A5D and even ABCD are perfectly good numbers in the hexadecimal system. Hex numbers are often used by assembly language programmers because they're closely related to binary numbers. It's that close relationship that we're going to take a look at now.

16-Finger Math

Remember how we used Penguin Math to explain the concept of binary numbers? Well, if you can imagine that you lived in a society where everyone had 16 fingers instead of 10 fingers like us, or two flippers like a penguin, then you'll be able to grasp the concept of hexadecimal numbers quite easily. Binary numbers, as we have pointed out, have a base of 2. Decimal numbers, the kind we're used to, have a base of 10. And hexadecimal numbers have a base of 16. Hexadecimal numbers are used in assembly language programming because they're quite similar to binary numbers; which, as we pointed out in Chapter 1, are the kind of numbers that computers understand. At first glance it may be difficult to see how binary numbers and hexadecimal numbers have anything in common. But you can see very clearly how binary and hex numbers relate to each other simply by looking at this chart:

Decimal	Hexadecimal	Binary
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

As you can see from this list, the decimal number 16 is written "10" in hex and "00010000" in binary, and is thus a round number in both systems. And the hexidecimal digit F, which comes just before hex 10 (or 16 in decimal), is written 00001111 in binary. As you become more familiar with the binary and hexadecimal systems, you will begin to notice many other similarities between them. For example, the decimal number 255 (the largest 8-bit number) is 11111111 in binary and FF in hex. The decimal number 65,535 (the highest memory address in a 64K computer) is written FFFF in hex and 11111111 11111111 in binary. And so on. The point of all this is that it's much easier to convert back and forth between binary and hexadecimal numbers than it is to convert back and forth between binary and decimal numbers.

1011	1000	binary
B	8	hexadecimal
184		decimal
0010	1110	binary
2	E	hexadecimal
46		decimal

1111	1100	binary
F	C	hexadecimal
252		decimal
0001	1100	binary
1	C	hexadecimal
28		decimal

As you can see, a nybble (four bits) in binary notation always equates to one digit in hexadecimal notation. But there is no clear relationship between the length of a binary number and the length of the same number written in decimal notation. This same principle can be extended to binary, hexadecimal and decimal conversions of 16-bit numbers. For example:

1111	1100	0001	1100	binary
F	C	1	C	hexadecimal
64540				decimal

Some Illustrative Programs

Now we'll look at some BASIC programs that perform operations involving binary, decimal and hexadecimal numbers. Since all Atari computers are 8-bit computers (at this writing, anyway), the only way to store a 16-bit number in an Atari is to put it into two memory registers. And 6502 based computers use the odd (to some people) convention of storing 16-bit numbers with the lower (least significant) byte first and the higher (most significant) byte second. For example, if the hexadecimal number FC1C were stored in hexadecimal memory addresses 0600 and 0601, FC (*most signficant byte*) would be stored in address 0601, and 1C (the *least signficant byte*) would be stored in address 0600. (In assembly language programs, incidentally, hexadecimal numbers are usually preceded with dollar signs so that they can be distinguished from decimal numbers. The hexadecimal addresses 0600 and 0601, therefore, would ordinarily be written $0600 and $0601 in an assembly language program.

Now let's suppose, just for illustration purposes, that you wanted to store a 16-bit number in two 8-byte addresses in your com-

puter's RAM ($0600 and $0601 for example), while running a BASIC program. Since BASIC programs are written using ordinary decimal numbers, the first thing you'd have to do is convert the addresses you were going to be working with, $0600 and $0601, into decimal numbers. You could do that in a number of different ways. You could look up the decimal equivalents of $0600 and $0601 on a decimal/hexadecimal conversion chart. Or you could carry out the necessary conversions by hand. Or you could perform them with the help of a computer program. No matter how you managed to make the conversions, however, what you would wind up discovering is that the hexadecimal number $0600 is equal to the decimal number 1536, and that the hexadecimal number $0601 is equivalent to decimal 1537. Once you found this out, you could store the 16-bit value in $0600 and $0601 using the following BASIC routine (or some variation of the same theme):

Routine for Storing a 16-Bit Number in RAM

```
10 PRINT "TYPE A POSITIVE INTEGER"
20 PRINT "RANGING FROM 0 TO 65,535"
30 INPUT X
40 LET HIBYTE=INT(X/256)
50 LET LOBYTE=X-HIBYTE*256
60 POKE 1536,LOBYTE
70 POKE 1537,HIBYTE
80 END
```

Now let's assume that that you wanted to retrieve a 16-bit number stored in your computer's RAM; for example, the number stored in memory addresses $0600 and $0601 in the above program. And let's suppose, once again, that you wanted to do that while you were running a BASIC program. Your BASIC routine might look something like this:

Retrieving a 16-Bit Number from RAM

```
10 LET X=PEEK(1537)*256+PEEK(1536)
20 PRINT X
```

Converting Binary Numbers to Decimal Numbers

Since assembly language programmers work with three different kinds of numbers, decimal, hexadecimal and binary numbers, they often find it necessary to perform conversions between one number base and another. It isn't very difficult to convert a binary number to a decimal number. In a binary number, the bit farthest to the right represents 2 to the power 0. The next bit to the left represents 2 to the power 1, the next represents 2 to the power 2, and so on. The digits in an 8-bit binary number are therefore numbered 0 to 7, starting from the rightmost digit. The rightmost bit — often referred to as the *least significant bit*, or *LSB* — represents 2 to the 0th power, or the number 1. And the leftmost bit — often called the *most significant bit*, or *MSB* — is equal to 2 to the 7th power, or 128. Here's a list of simple equations that illustrate what each bit in an 8-bit binary number means:

$$\text{Bit } 0 = 2^0 = 1$$
$$\text{Bit } 1 = 2^1 = 2$$
$$\text{Bit } 2 = 2^2 = 4$$
$$\text{Bit } 3 = 2^3 = 8$$
$$\text{Bit } 4 = 2^4 = 16$$
$$\text{Bit } 5 = 2^5 = 32$$
$$\text{Bit } 6 = 2^6 = 64$$
$$\text{Bit } 7 = 2^7 = 128$$

Using the above chart, it's easy to convert any 8-bit binary number into its decimal equivalent. Instead of writing the number down from left to right, write it instead in a vertical column, with bit 0 at the top of the column and bit 7 at the bottom. Then multiply each bit in the binary number by the decimal number that it represents. Then add up the results of all of these multiplications. The total you get will be the decimal value of the binary number. Suppose, for example, that you wanted to convert the binary number 00101001 into a decimal number. Here's how you'd do it:

$$
\begin{array}{rrr}
1 \times & 1 = & 1 \\
0 \times & 2 = & 0 \\
0 \times & 4 = & 0 \\
1 \times & 8 = & 8 \\
0 \times & 16 = & 0 \\
1 \times & 32 = & 32 \\
0 \times & 64 = & 0 \\
0 \times & 128 = & 0 \\
\hline
\multicolumn{3}{l}{\text{TOTAL} = 41}
\end{array}
$$

According to the results of this calculation, the binary number 00101001 is equivalent to the decimal number 41. Look up either 00101001 or 41 on a binary-to-decimal or decimal-to-binary conversion chart, and you'll see that the calculation was accurate. And this conversion technique will work with any other binary number. Now we'll go in the other direction, and convert a decimal number to a binary number. And here's how we'll do that: We'll divide the number by 2, and write down both the quotient and the remainder. Since we'll be dividing by 2, the quotient will be either a 1 or 0. So we'll write down either a 1 or a 0. Then we'll take the quotient we got, divide it by two, and write that quotient down. If there's a remainder (a 1 or a 0), we'll write that down, too, right underneath the first remainder. When there are no more numbers left to divide, we'll write down all of the *remainders* we got, reading from the bottom to the top. What we'll have then, of course, is a binary number, a number made up of ones and zeros. And that number will be the binary equivalent of the decimal number we started out with. Now let's try this conversion technique on the decimal number 117:

$$
\begin{array}{rl}
117/2 = & 58 \text{ with a remainder of } 1 \\
58/2 = & 29 \text{ with a remainder of } 0 \\
29/2 = & 14 \text{ with a remainder of } 1 \\
14/2 = & 7 \text{ with a remainder of } 0 \\
7/2 = & 3 \text{ with a remainder of } 1 \\
3/2 = & 1 \text{ with a remainder of } 1 \\
1/2 = & 0 \text{ with a remainder of } 1
\end{array}
$$

According to the results of this calculation, the binary equivalent of the decimal number 117 is 01110101. And this result, as a check of a decimal-to-binary conversion chart would confirm, is also accurate.

Binary-to-Hex and Hex-to-Binary Conversions

It's easy to convert binary numbers to their decimal equivalents. Just use this chart:

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

To convert a multiple digit hex number to binary, just string the hex digits together and convert each one separately. For example, the binary equivalent of the hexadecimal number C0 is 1100 0000. The binary equivalent of the hex number 8F2 is 1000 1111 0010. The binary equivalent of the hex number 7A1B is 0111 1010 0001 1011. And so on. To convert binary numbers to hexadecimal numbers, just use the chart in reverse. The binary number 1101 0110 1110 0101, for example, is equivalent to the hexadecimal number D6E5.

Doing it the Easy Way

Even though it isn't difficult to convert binary numbers to hexadecimal and vice versa, it is time consuming to do it by hand, and when you program in assembly language, you have to do a lot of

binary-to-decimal and decimal-to-binary converting. So there are a lot of BASIC programs around for converting numbers back and forth between the binary and decimal notation systems. And now I'm going to give two of them to you, absolutely free. Here's one for converting binary numbers to decimal numbers:

Converting Binary Numbers to Decimal Numbers

```
10 DIM BN$(9),BIT(8),T$(1)
20 GRAPHICS 0
25 ? :? "BINARY-DECIMAL CONVERSION"
30 ? :? "ENTER AN 8-BIT BINARY NUMBER:":?
   :INPUT BN$
35 IF LEN(BN$)< >8 THEN 30
40 FOR L=1 TO 8
50 T$=BN$(L)
55 IF T$< >"0" AND T$< >"1" THEN 30
60 BIT(L)=VAL(T$)
70 NEXT L
75 ANS=0
80 M=256
90 FOR X=1 TO 8
100 M=M/2:ANS=ANS+BIT(X)*M
110 NEXT X
140 ? "DECIMAL: ";ANS
150 GOTO 30
```

And here's a program for converting decimal numbers to binary numbers:

Converting Decimal Numbers to Binary Numbers

```
10 DIM BIN$(8),TEMP$(8),R$(1)
20 GRAPHICS 0
30 ? :? "DECIMAL-BINARY CONVERSION"
40 ? :? "ENTER A POSITIVE INTEGER (0 TO
   255):":? :TRAP 40:INPUT NR
50 IF NR-INT(NR)< >0 THEN 40
```

```
60 IF NR>255 OR NR<0 THEN 40
70 FOR L=1 TO 8
80 Q=NR/2
90 R=Q-INT(Q)
100 IF R=0 THEN R$="0":GOTO 120
110 R$="1"
120 TEMP$(1)=R$:TEMP$(2)=BIN$
    :BIN$=TEMP$
130 NR=INT(Q)
140 NEXT L
150 ? "BINARY: ";BIN$
160 TRAP 40000
170 GOTO 40
```

Decimal/Hexadecimal Conversion

Decimal/hexadecimal conversion is a complex process best done on a computer or a special calculator. Texas Instruments makes a calculator called the Programmer that can perform decimal/hexadecimal conversions in a flash, and can also add, subtract, multiply and divide both decimal and hexadecimal numbers. Many assembly language program designers use the TI Programmer, or some similar calculator, and would have a hard time getting along without it. In case you can't get your hands on a programmer's calculator right away, here's an Atari BASIC program that will convert decimal numbers to hexadecimal numbers and vice versa. Another program is available on page H-18 of the *BASIC Reference Manual* that comes with the Atari BASIC cartridge.

Dec-Hex and Hex-Dec Conversion Program

```
10 REM
20 REM HEX TO DEC CONVERSION PROGRAM
30 REM
40 REM THE FAST WAY
50 REM DON'T USE MATH
60 REM
70 DIM H$(40),A$(40)
80 REM
```

```
90 PRINT
100 PRINT
110 PRINT
120 PRINT "HEX TO DECIMAL CONVERSION"
130 PRINT
140 PRINT "H) HEX TO DEC"
150 PRINT "D) DEC TO HEX"
160 PRINT
180 INPUT A$
190 IF A$="H" THEN 220
200 IF A$="D" THEN 400
210 GOTO 100
220 REM
230 REM CONVERT HEX TO DECIMAL
240 REM
250 PRINT "ENTER HEX NUMBER";
260 INPUT H$
270 REM
280 D=0
290 S=1:REM POSITIONAL MULTIPLIER
295 REM GO THROUGH THE STRING
300 FOR L=LEN(H$) TO 1 STEP -1
310 A$=H$(L,L)
320 REM
330 REM CONVERT "0"-"F" TO 0-15
340 N=ASC(A$)-48:IF N>9 THEN N=N-7
345 IF N<0 OR N>15 THEN 90
350 D=D+N*S
360 S=S*16
370 NEXT L
380 PRINT "HEX ";H$;" = DEC ";D
390 GOTO 90
400 REM
410 REM CONVERT DECIMAL TO HEX
420 REM
430 PRINT "ENTER DECIMAL TO CONVERT"
440 INPUT D
450 REM
460 REM FIRST FIND THE HIGHEST DIGIT
470 REM
480 S=16
```

```
490 X=2
500 IF S<D THEN X=X+1:S=S*16:GOTO 500
505 PRINT "DECIMAL ";D;" = HEX ";
510 T=D
520 FOR L=X TO 1 STEP -1
530 N=INT (T/S)
540 PRINT CHR$(48+N+7*(N>9));:REM
    CONVERT DEC TO HEX 0-F
550 T=T-N*S:S=S/16
560 NEXT L
570 PRINT
580 GOTO 90
```

That concludes our crash course into bits, bytes, and binary. This was an important chapter because it's a prerequisite to Chapter 3, which in turn is a prerequisite to Chapter 4, where you'll get a chance to actually start writing some assembly language programs.

Something to Tide You Over

Meanwhile, here's another BASIC program that might help you feel that you're getting through to your Atari. It uses an infinite loop to cycle through all of the colors and hues that your computer can generate, loading each of them in turn into the memory register that controls the border area of your video screen. In Chapter 9, Programming Bit by Bit, you'll learn how to do this same trick using assembly language.

BONUS PROGRAM NO. 2
THE ATARI RAINBOW

```
10 REM ** THE ATARI RAINBOW **
20 REM ** "D:RAINBOW.BAS" **
30 REM
40 FOR L=2 TO 254 STEP 2:REM ALL VALID
   COLORS HAVE EVEN NUMBERS
50 POKE 712,L:REM 712 IS ADDRESS OF
   BORDER AREA COLOR REGISTER
```

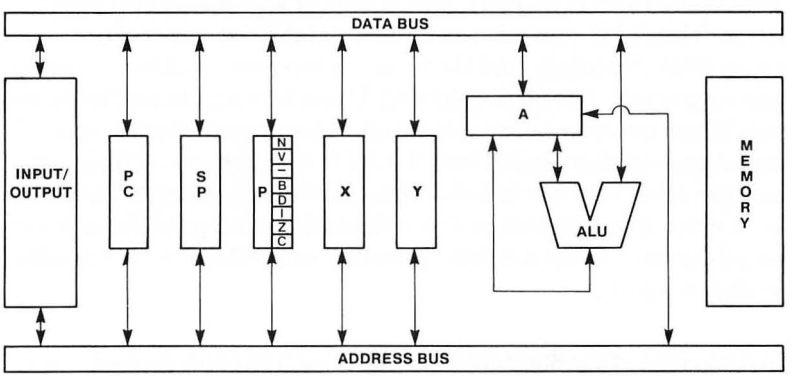
```
60 FOR WAIT=1 TO 10:NEXT WAIT:REM JUST
   A DELAY LOOP TO USE UP TIME
70 NEXT L
80 GOTO 40:REM AN INFINITE LOOP
```

Type this program into your computer, run it, and watch the show! Then we'll continue on to Chapter 3.

Chapter Three

Inside the 6502

In this chapter we're going to get under the hood of your Atari computer and see how it works. Then you'll be able to find your way around inside your computer and at last start doing some assembly language programming. As we explained in Chapter 1, every computer has three main parts: a Central Processing Unit (CPU), memory (divided into RAM and ROM), and input and output devices (such as keyboards, video monitors, cassette recorders, and disk drives). In a microcomputer, all of the functions of a CPU are contained in a microprocessor unit (sometimes abbreviated MPU). Your Atari's MPU is a 6502 microprocessor.



The 6502 microprocessor contains seven main parts: an *Arithmetic Logical Unit (ALU)* and six addressable registers. Data is moved around inside the 6502 chip and between the 6502 and other components in your computer over transmission lines called

buses. There are two kinds of buses in an Atari computer: an 8-bit *data bus* and a 16-bit *address bus*. The data bus is used for passing 8-bit data and instruction bytes from one 6502 register to another, and also as for passing data and instructions back and forth between the 6502 and your computer's memory. The address bus is used to keep track of your computer's 16-bit memory addresses: the addresses that instructions and data are coming from, and the addresses that instructions and data are being sent to.

The Arithmetic Logical Unit

The most important component in your computer is the 6502 chip. And the most important part of the 6502 chip is its Arithmetic Logical Unit (ALU). Every time your computer performs a calculation or a logical operation, the ALU is where all of the work is done. The ALU can actually perform only two kinds of calculations: addition operations and subtraction operations. Division and multiplication problems can also be solved by the ALU, but only in the form of sequences of addition and subtraction operations. The ALU can also compare values, by subtracting one value from the other, and then noting the results of the subtraction operation. The 6502 chip's ALU has two inputs and one output. When two numbers are to be added, subtracted or compared, one of the numbers is fed into the ALU through one of its inputs, and the other number is fed in through the other input. The ALU then carries out the requested calculation, and puts the answer on a data bus so that it can be transported to wherever it's needed in the program.

In diagrams of the 6502 chip, the ALU is often represented as a V-shaped hopper. The arms of the V are the ALU's inputs, and the bottom of the V is the ALU's output. When a calculation or a logical operation is to be carried out by the ALU, one piece of data and an operand (an addition or a subtraction) instruction are deposited into one of the ALU's inputs (one arm of the V). The other piece of data is deposited into the other input (the other arm of the V). When the calculation has been performed, its result is ejected through the ALU's output (the bottom of the V).

The Accumulator

The ALU never works alone; it carries out all of its operations with the help of a 6502 register called the accumulator (abbreviated "A"). When the ALU is called upon to add or subtract two numbers, one of the numbers is put on a data bus and then sent to one of the ALU's inputs, along with an operand. The other number is in the accumulator. When the bus carrying a number and an operand to the ALU discharges its cargo into the ALU, the accumulator puts the number it is holding on the data bus and sends that number to the ALU. When the ALU has carried out the requested calculation, it deposits the result of the calculation in the accumulator.

An Example

Suppose, for example, that you wanted your computer to add 2 and 2, and then place the result of its calculation into a certain memory location. You could use an assembly language routine like this one:

```
LDA #Ø2
ADC #Ø2
STA $CB
```

The first instruction in this routine, "LDA," means "LoaD the Accumulator" (with the value that follows). In this case, that value is 2. The "#" sign in front of the 2 means that the 2 is to be interpreted as a literal number, not as the address of a memory location in your computer.

The second instruction in the routine, ADC, means "ADd with Carry." In this addition problem there is no number to be carried, the "carry" part of the instruction has no effect here, and all the ADC instruction does is add 2 and 2.

The third and last instruction in our routine, STA, means "STore the contents of the Accumulator" (in the memory address that follows).

As you can see, the memory address that follows the instruction STA is $CB, the hexadecimal equivalent of the decimal number 203. Since there is no "#" sign in front of the hex number $CB, your assembler will not interpret $CB as a literal number. Instead, $CB will be interpreted as a memory address, which is what a number has to be in assembly language if it is not a literal number. (Incidentally, if you did want your assembler to interpret $CB as a literal number, you would have to write it "#$CB." When a "#" symbol and a dollar sign both appear before a number it is interpreted as a literal *hexadecimal number.*) If the third line of our sample routine read STA #$CB, however, that would be a syntax error. That's because STA (store the contents of the accumulator in . . .) is an instruction that has to be followed by a value that can be interpreted as a memory address, not by a literal number.

Five Other Registers

Besides the accumulator, the 6502 processor has five other registers. They are the *X Register*, the *Y Register*, the *Program Counter*, the *Stack Pointer*, and the *Processor Status Register*. Here is a brief summation of the functions of each of these registers.

The 6502's Other Registers

- The *X Register* (abbreviated "X") is an 8-bit register that is often used for temporary storage of data during a program. But the X register has a special feature, too; it can be incremented and decremented with a pair of one-byte assembly language instructions (INX and DEX), and it is therefore often used as an index register, or counter, during loops and read/data-type instructions in programs.

- The *Y Register* (abbreviated "Y") is also an 8-bit register, and can also be incremented and decremented with a pair of one-byte instructions (INY and DEY). So the Y register, like the X register, is used both for data storage and as a counter.

- The *Program Counter* (abbreviated "PC") is a pair of 8-bit registers that are used together as one 16-bit register. The two 8-bit registers that are combined to make up the Program Counter are sometimes referred to as *"Program Counter-Low (PCL)"* and *"Program Counter-High (PCH)"*. The program counter always contains the 16-bit memory address of the next instruction to be executed by the 6502 processor. When that instruction has been carried out, the address of the next instruction is loaded into the program counter.

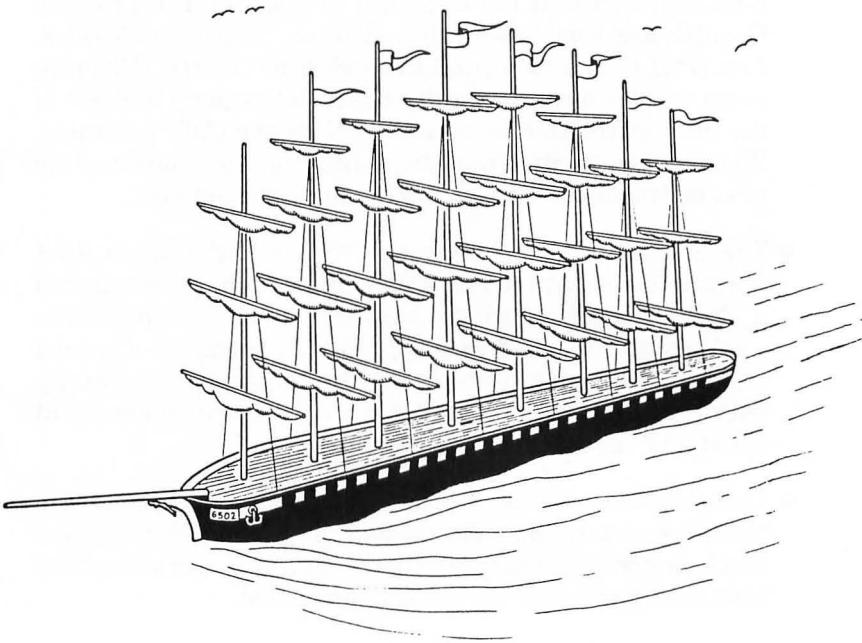
- The *Stack Pointer* (abbreviated "S" or "SP") is an 8-bit register that always contains the address of the top element in a block of RAM called the hardware stack. The *hardware stack*, usually referred to simply as "the stack," is a special segment of memory in which data is often stored temporarily during the execution of a program. We'll go into more detail about how the stack works later on.

- The *Processor Status Register* (usually called simply the "status register," but abbreviated "P") is an 8-bit register that keeps track of the results of the results of operations that have been performed by the 6502 processor.

The Processor Status Register

The Processor Status Register (P) is a little different from the other registers in the 6502 microprocessor. It isn't used for storing ordinary 8-bit numbers, as the 6502's other registers are. Instead, this register's bits are flags that keep track of several kinds of important information.

Four of the status register's bits are called *status flags*. They are the *carry flag (C)*, the *overflow flag (V)*, the *negative flag (N)*, and the *zero flag (Z)*. These four flags are used to keep track of the results of operations being carried out by the other registers inside the 6502 processor. Three of the P register's other bits, called *condition flags*, are used to determine whether certain conditions exist in a program. These three bits are the *interrupt disable flag (I)*, the *break flag (B)*, and the *decimal mode flag (D)*. The eighth bit in the status register is not used.



Layout of the Processor Status Register

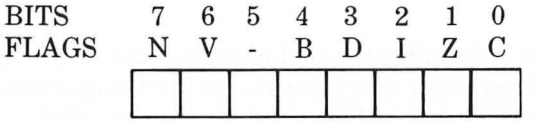
The status register can be visualized as a rectangular box containing six square compartments. Each "compartment" in the box is actually a bit, and each bit is used as a flag.

If a given bit is a "1" instead of a "0," then it is said to be a flag that is set.

If a given bit is a "0" instead of a "1," then it is said to be a flag that is cleared.

The bits in the 6502 status register, like the bits in all 8-bit registers, are customarily numbered from 0 to 7. The rightmost bit is bit 0. The leftmost bit is bit 7.

An Illustration of the Processor Status Register



Here's a complete list of the flags in the 6502's processor status register, and an explanation of what each one means.

Bit 0 (The Rightmost Bit) The Carry Flag (C)

As we saw in Chapter 2, it isn't easy to do 16-bit arithmetic with an 8-bit chip like the 6502. When the 6502 chip is required to perform an addition operation on a number greater than 255, or if the result of a calculation might be greater than 255, a program has to be written that will break each number down into 8-bit segments for processing, and will then patch all of the numbers back together again. This kind of mathematical cutting and pasting, as you can probably imagine, involves a lot of carrying (if addition problems are being performed) and borrowing (when the 6502 is performing subtraction). The carry flag of the 6502 P register is the flag that keeps up with all of this carrying and borrowing.

If an addition operation results in a carry, the carry flag is automatically set. And if a subtraction operation requires a borrow, the carry flag keeps track of that too. Since the carry flag is almost constantly being set and cleared as a result of carries and borrows in addition and subtraction, it's always a good idea to *clear* it before an addition operation is to be carried

out, and to *set* it before a subtraction operation takes place. Otherwise, there's a chance that your calculations will be messed up by the leftover results of previous addition and subtraction operations.

In addition to keeping track of carrying and borrowing operations, the P register's carry flag is also used in operations involving comparisons of values, and in certain *shift* and *rotate* operations to check, compare and manipulate specific bits in binary numbers. We'll discuss number comparisons and bit operations in later chapters. For now, it's more important to remember that the assembly language instruction to clear the P register's carry bit is *CLC*, which stands for *"CLear Carry,"* and that the instruction to set the carry bit is *SEC*, which stands for *"SEt Carry."*

Bit 1 (The Second Bit from the Right) The Zero Flag (Z)

When the result of an arithmetical or logical operation is zero, the status register's zero flag is automatically set. Addition, subtraction and logical operations can all result in changes in the status of the zero flag. If a memory location or an index register is decremented to zero, that will also result in a set zero flag. An ironic 6502 convention is that when the result of an operation is zero, the zero flag is set to 1, and that when the result of an operation is not zero, the zero flag is *cleared* to 0. It's important to understand this concept, since it would be easy to assume that the zero flag operates in just the opposite manner. There are no assembly language instructions to clear or set the zero flag. It's strictly a "read" bit, so instructions to write to it are not provided.

Bit 2 (The Third Bit from the Right) The Interrupt Disable Flag (I)

Some Atari programs contain *interrupts*; instructions that halt operations temporarily so that other operations can take place. Some interrupts are called *maskable* interrupts because you can prevent them from taking place by including "masking" instruc-

tions in a program. Others are called *nonmaskable* because you can't stop them from taking place, no matter what you do. When you want to disable a maskable interrupt, you can do it with the P register's interrupt disable flag. When the flag is set, maskable interrupts are not permitted. When it is clear, they are allowed. The assembly language instruction to clear the interrupt flag is CLI. The instruction to set the interrupt flag is SEI.

Bit 3 (The Fourth Bit from the Right) The Decimal Mode Flag (D)

The 6502 processor normally operates in *binary mode*, using standard binary numbers of the type that were discussed in Chapter 2. But the 6502 can also operate in what is known as a binary coded decimal, or BCD mode. To put the 6502 into *BCD* mode, you have to set the decimal flag of the 6502 status register. BCD arithmetic is slower than plain binary arithmetic, and it also consumes more memory. But its results, unlike those of plain binary arithmetic, are always 100 percent accurate. Therefore it is often used in programs and routines in which accuracy is more important than speed or memory efficiency.

One example of a program that uses BCD arithmetic is your Atari BASIC interpreter. In Atari BASIC all numbers are stored as 6-byte BCD numbers, and all arithmetic is performed as BCD arithmetic. Because of this feature, Atari BASIC runs somewhat slowly. But its calculations yield accurate results. Another advantage of BCD numbers is that they're easier to convert into decimal numbers than plain binary numbers are. So BCD numbers are sometimes used in programs that call for the instant display of numbers on a video monitor as well.

We'll discuss BCD at greater length in Chapter 10, Assembly Language Math. For now, it's sufficient to say that when the status register's decimal mode flag is set, the 6502 chip will perform all of its arithmetic using BCD numbers. BCD arithmetic is rarely what you want when you use an Atari computer, so you'll usually want to make sure that the decimal flag is clear when your computer is doing arithmetic. The assembly language instruction that clears the decimal flag is CLD. The instruction that sets the flag is SED.

Bit 4 (The Fifth Bit from the Right) The Break Flag (B)

The break flag is set by a special assembly language instruction, BRK. Program designers often use the break instruction in assembly language programs during the debugging phase. When the instruction is used and the break flag is set, certain error flagging operations take place and control of the computer returns to the programmer. The break instruction is a highly complex, sophisticated debugging tool, and we won't go into much detail about it in this volume. But you can learn more about it in some of the advanced 6502 programming texts listed in the bibliography.

Bit 5 (The Sixth Bit from the Right) [Unused Bit]

For some reason, the microprogrammers who designed the 6502 status register left one bit unused. This is the one.

Bit 6 (The Second Bit from the Left) The Overflow Flag (V)

The overflow flag is used to detect an overflow from bit 6 (the next to rightmost bit) in a binary number. If you don't know what that means yet, don't be concerned about it; the overflow flag is used primarily in advanced 6502 arithmetic, specifically to keep track of changes in the plus and minus signs of signed numbers when signed binary arithmetic is being performed. As an Atari assembly language programmer, you'll rarely, if ever, have an occasion to use the overflow flag. Nevertheless, we'll discuss it at greater length in Chapter 10, Assembly Language Math. For now, all we'll add (just for the record) is that the assembly language instruction that clears the overflow flag is CLV. There is no instruction to set the flag.

Bit 7 (The Leftmost Bit)
The Negative Flag

The negative flag is set when the result of an operation is negative, and cleared when the result of an operation is zero. It is often used in operations involving signed numbers, and it also has other uses that will be discussed in later chapters. There are no instructions to set or clear the negative flag. There is no need for any, since the flag is used for test purposes only.

Your Big Chance

Now that you know what goes on inside your computer's 6502 processor, you're ready to write, and run, your first assembly language program. You will have a chance to do just that in the next chapter. But first, here's a chance to run another bonus program containing some machine language values that can be POKEd in from a BASIC program. This routine, Bonus Program No. 3, will turn your computer keyboard into a musical keyboard. Type it and run it, and then we'll take a look at how it works.

BONUS PROGRAM NO. 3
"D:SOUNDOFF.BAS"

```
10 REM "D:SOUNDOFF.BAS"
20 INKEY=35:FREQ=53760
30 SOUND 0,0,0,0:GRAPHICS 0:OPEN #1,4,0,"K:"
35 GET #1,K:IF K=32 THEN SOUND 0,0,0,0:PRINT
   CHR$(K):GOTO INKEY:REM 32 IS A SPACE
40 IF K<64 OR K>122 THEN GOTO INKEY:REM
   NOT A LETTER
50 IF K>90 AND K<97 THEN GOTO INKEY:REM
   NOT A LETTER
60 IF K>90 THEN K=K-32:REM CHANGE LOWER
   CASE TO UPPER CASE
70 TONE=K-64:IF TONE<1 OR TONE>7 THEN
   GOTO INKEY: REM NOT A,B,C,D,E,F OR G
80 PRINT CHR$(K);:GOSUB 1000
90 ON TONE GOTO 100,200,300,400,500,600,700
100 POKE FREQ,145:GOTO INKEY:REM A
```

```
200 POKE FREQ,134:GOTO INKEY:REM B
300 POKE FREQ,121:GOTO INKEY:REM C
400 POKE FREQ,108:GOTO INKEY:REM D
500 POKE FREQ,96:GOTO INKEY:REM E
600 POKE FREQ,91:GOTO INKEY:REM F
700 POKE FREQ,81:GOTO INKEY:REM G
1000 SOUND 0,255,10,8:RETURN:REM SET UP
     AUDIO REGISTERS TO PLAY NOTES
```

How it Works

In lines 30 and 35 of this program the audio registers of your computer are cleared to zero and a loop is set up to print characters on your screen. In lines 40 through 70 some checks are carried out to see if any characters that have been typed are valid notes, A, B, C, D, E, F or G. If a character is typed in lower case, it is automatically converted to upper case in line 60 to make the program work smoother. In the subroutine at line 1000 the audio registers in your computer are reset to play the notes A through G when the corresponding letters are typed, unless the character is a space. If the character is a space, all audio registers are turned off, a space is printed on the screen, and the computer is instructed to wait for the next typed character (that happens in line 35).

The machine language instructions in the program are in lines 100 through 700. In those lines a certain memory register in your computer (called FREQ in this program) is stuffed with a value that equates to a musical note. Each time that value changes, the note being played changes accordingly. This is a very simple program that doesn't even begin to explore the complex sound capabilities of Atari computers. Still, the ways in which it can be expanded are limited only by the user's own imagination. By making the program a little more complicated, you could add the capability of reproducing sharps and flats (with the control key, for example), and you could also add more octaves (for instance with the shift key, the numbers keys, or shift/control key combinations). You could make the screen change colors as the notes change or you could save melodies in your computer's memory and save them on a disk and you might even be able to figure out a way to play chords! You could do all of those things in BASIC, if

you wished or, if you were more ambitious, you could move beyond BASIC and continue to learn more about how to put your Atari through its paces using assembly language. Which brings us to Chapter 4. Read on.

Chapter Four

Writing an Assembly Language Program

Here it is, at long last, the assembly language program we promised you, a program you can type, assemble and execute without having to rely on any BASIC commands. The program was written on an old, battle-scarred Atari 800 computer, using an Atari Assembler Editor cartridge almost as ancient. But the program will run on any Atari computer and, like all of the programs in this book, it is totally compatible with both the Atari Assembler Editor cartridge and the newer and faster MAC/65 assembler manufactured by OSS (Optimized Systems Software, Inc., of San Jose, CA).

With a few minor but critical revisions such as eliminating line numbers and changing the "*=$0600" directive in line 40 to "ORG $0600", this program, and all of the others in this book, can also be converted into programs that will work with the Atari Macro Assembler and Text Editor. But unless you're already an assembly language programmer, I strongly recommend that you type, assemble, debug and edit the programs in this book using an assembler more like the ones they were written on: the MAC/65 assembler and the Atari Assembler Editor cartridge.

Here's the program we'll be working with in this chapter. As you can see, it's a very simple program for adding 2 and 2. Let's take a close look at it, and see how it does what it's supposed to do.

AN 8-BIT ADDITION PROGRAM (ADDNRS.SRC)

```
10 ;
20 ;ADDNRS.SRC
30 ;
40 *=$0600
```

```
50 ;
60  CLD
70 ADDNRS CLC
80  LDA #2
90  ADC #2
100  STA $CB
110  RTS
120  .END
```

Look closely at this program and you'll see that the numbers it's supposed to add, 2 and 2, are in lines 80 and 90. After the program adds 2 and 2, it stores the result of its calculation in memory address $CB (or 203 in decimal notation). That happens in line 100. Some of the instructions in the program may look familiar; we've touched on most of them in the preceding chapters. But there are a few items in the program that are encountered here for the first time. These include the semicolons in the first few lines of the program, the "*=" directive in line 40, and the ".END" directive in line 120.

Spacing Out

As you can see more clearly in the "explosion diagram" that follows, the program's source code listing is divided into four fields, or columns. If each field had a heading, and if the program listing weren't all jammed together, here's what you would see:

AN 8-BIT ADDITION PROGRAM (ADDNRS.SRS) (Column-by-Column Listing)

```
FIELD  LINE              OP
NAMES  NO.  LABEL        CODE  OPERAND  REMARKS

       10   ;
       20   ;ADDNRS.SRC
       30   ;
       40                *=    $0600
       50   ;
       60   ADDNRS       CLD
       70                CLC
```

```
80              LDA   #2
90              ADC   #2
0100            STA   $CB
0110            RTS
0120            .END
```

Only an Example

The above listing is provided only as an illustration of what a source code listing would look like if its four fields; line numbers, labels, op codes and remarks, were clearly separated into columns. Actually, no one writes source code listings this way.

When you write a source code listing using MAC/65 or the Atari Assembler Editor cartridge, the usual way to do it is to type your fields in a format that is rather jammed together; crammed together so tightly, in fact, that the columns don't line up at all. Once you you get a little practice writing code in this format, though, it practically becomes second nature.

These are the rules:

Line Numbers

When you write a source code listing using MAC/65 or the Atari Assembler Editor cartridge, each statement, or line, must be assigned a line number. The line numbers of the program are typed flush left, just as they are in BASIC programs. Line numbers aren't really necessary in assembly language programs, and are not required by some assemblers. The Atari Macro Assembler and Text Editor package, for example, doesn't require the use of line numbers and won't assemble a program that includes them. But MAC/65 and the Atari Assembler Editor cartridge do use line numbers, so we'll use them too, at least for now.

The line numbers in the first column of our addition program progress in increments of 10, just like the numbers in a typical BASIC program. They don't have to be written that way, but, as is also the case in BASIC programs, they usually are. Line numbers can range from 0 through 65535.

Labels

Labels, if they are used, occupy the second field in assembly language statements. Exactly one space, not two, must be left between a line number and any label that follows it. If you start a label two or more spaces after a line number, or if you use your tab key to get to your label field, you may clobber your program. In assembly language programs, labels are used to identify the first lines of routines and subroutines. Our program for adding 2 and 2, for example, is labeled ADDNRS in line 60.

Since our program has a label, the program could be used as a subroutine in a longer program, and could easily be accessed by its label. Several kinds of assembly language instructions could be used to call it; for example, JSR ADDNRS (Jump to SubRoutine at label ADDNRS), BCS ADDNRS (Branch on Carry Set to label ADDNRS), or JMP ADDNRS (JuMP to label ADDNRS). In the first example, ADDNRS would end with the RTS (ReTurn from Subroutine) instruction in line 110. RTS performs the same kind of job in assembly lanugage that the RETURN instruction performs in BASIC; it signals the end of a subroutine, and returns control back to the main body of the program. In the latter examples, a second branch or jump instruction would be used to transfer control instead of the RTS.

We'll discuss jumping, branching instructions, and subroutines at greater length in later chapters. For now, what's most important to remember is that when you type a source code listing on a MAC/65 assembler or Atari Assembler Editor cartridge, one space and only one space must be used to separate each label you use from its line number. A label can be as short as one character and as long as the length of a statement permits (106 characters minus the number of characters in the statement's line number). Most programmers use labels three to six characters long.

Operation Code Mnemonics

An *operation code* (or *op code*) mnemonic is just a fancy name for an assembly language instruction. There are 56 op code mnemonics in the 6502 instruction set, and they are the only ones that can be used in Atari assembly language instructions. Op code mnemon-

ics such as CLC, CLD, LDA, ADC, STA and RTS are typed in the op code field of assembly language source code listings. When you write a program using a MAC/65 or the Atari Assembler Editor, each op code mnemonic you use must start *at least two spaces* after a line number, or *one space* after a label. An op code mnemonic placed in the wrong field will not be flagged as an error when you type your program, but will be flagged as an error when your program is assembled.

The op code field in a source code listing is also used for *directives* and *pseudo ops*; words and symbols that are entered into a program like mnemonics but are not actually members of the 6502 instruction set. The asterisk in line 40 of our program is a directive, and the ".END" statement in line 120 is a pseudo op. The "*" directive is used to tell your computer where an assembly language program is to be stored in memory after it is assembled. The ".END" directive is used to tell the assembler where to stop assembling, and to end an assembly language program.

Operands

The *operand* field in a MAC/65 Atari Assembler program starts at least one space (or a tab) after an op code mnemonic. Operands are used to expand op code mnemonics into complete instructions. Some mnemonics, such as CLC, CLD and RTS, do not require operands. Others, such as LDA, STA and ADC, do require operands. We'll be providing much more information about operands later on.

Comments

Comments in assembly language programs are like remarks in BASIC programs; they don't affect a program in any way, but are used to explain programming procedures and to provide eye-saving space in program listings. There are two ways to write comments in source code listings written on the MAC/65 and the Atari Assembler Editor. One method is to put comments in the label field of a listing, preceded by semicolons. The other method is to put them in a special "Comments" field that occupies what's

left of each line following the instruction fields (the op code and operand fields). If you use the "Comments" field at the end of a line and don't have room there for the comment you want to write, you can continue your remarks on the next line by simply typing a space, a semicolon, and the rest of your comments.

Examining the Program

Now that we've looked at our sample program field by field, let's examine it line by line.

Lines 10 Through 30

(Comments)

Lines 10 through 30 are comments. Line 20 explains what the program does and lines 10 and 30 set off the explanatory line with white space. It's good programming practice to use remarks liberally in assembly language, as well as in most other programming languages, so we've used quite a few comments in the programs in this volume.

Line 40

("*=" Directive)

This is the *origin* line of our sample program. Every assembly language program must start with an origin line. As you may remember from Chapter 1, the first thing a computer does when it runs a machine language program is go to a predetermined memory location and see what it finds there. So when you write an assembly language program, the first thing *you* have to do is tell your computer where to start looking for the program in its memory. When your assembler encounters an origin directive, it will set the program counter of your computer's 6502 processor to the address given in the directive. The first instruction in the program will then be loaded into that memory location, and the rest of the program will follow in sequence.

The origin directive looks like a simple line to write, but deciding what number to put in this line can be a very tricky job, especially for the beginning assembly language programmer. There are many blocks of memory in your computer that you can't use for assembly language programs because they're reserved for other uses (for example, to hold your computer's operating system, disk operating system, BASIC interpreter, and so on). Even the assembler that you'll use to write this program takes up a block of memory space that is forbidden territory for you, at least until you become familiar with certain landmarks you can use to find your way through the jungle of your computer's memory.

Until then, there does happen to be one small block of memory that's reserved, under ordinary circumstances, for just the kind of short, user-written assembly language programs that we're going to be working with in the next few chapters. The block of memory is called page 6 because it runs from memory address $0600 to $06FF (1536 to 1791 in decimal notation). Page 6 is only 256 bytes long, but that's more than enough space for the program we're going to be working with in this chapter, and for the other programs that we're going to be working on for the next few chapters. Later on, as your programs grow longer, you'll learn how to move on to larger blocks of memory. Line 40 in our sample addition program tells your computer that the program you're going to write will start at memory address $0600 (1536 in decimal notation).

Line 50

(Blank Line)

This "Comment" line just separates the origin line from the other lines in our program. The white space looks nice, doesn't it?

Line 60
ADDNRS CLD

(Label: ADDNRS)
(Mnemonic: "Clear Decimal Mode")

ADDNRS: We've used the label field in this line to name our program ADDNRS. So if we ever decide to use our program as a routine or a subroutine in a larger program, it will have a name. Then we can address it by its name, if we wish, instead of by its memory location. It is usually good programming practice to give labels to important routines and subroutines. A label not only makes a routine easier to locate and use, it also serves as a reminder of what the routine does (or, until your program is debugged, what it's supposed to do).

CLD: We're using plain binary numbers in this program, not Binary Coded Decimal (BCD) numbers. So in this line we'll clear the decimal mode flag of the 6502 processor status register. The decimal flag need not be cleared before every arithmetical operation in a program, but it's a good idea to clear it before the *first* addition or subtraction operation in a program, since it just may have been set during a previous program.

Line 70
CLC

("CLear Carry")

The status register's carry flag is affected by so many kinds of operations that it's considered good programming practice to clear this flag before every addition operation. It takes only one half a millionth of a second, and just one byte of RAM. Compared to the time and energy that debugging can cost, that's a bargain.

Line 80
LDA #2

("LoaD Accumulator with the Number 2")

This is a very straightforward instruction. The first step in an addition operation is always to load the accumulator with one of the numbers that is to be added. The "#" sign in front of the number 2 means that it's a literal number, not an address. If the instruction were "LDA 2," then the accumulator would be loaded with the contents of memory address 0002, not the number 2.

Line 90
ADC #2

("ADd with Carry the number 2 to the accumulator")

This is also a straightforward instruction. "ADC #2" means that the literal number 2 is to be added to the number that's in the accumulator; in this case, another 2. As we've mentioned, there is no 6502 assembly language instruction that means "add without carry." So the only way that an addition operation can be performed without a carry is to clear the status register's carry flag and then perform an "add with carry" operation.

Line 100
STA $CB

("STore Accumulator in Memory Address $CB")

This line completes our addition operation. It stores the contents of the accumulator in memory address $CB (decimal 203). Note that the symbol "#" is not used before the operand (CB) in this instruction, since the operand in this case is a memory address, not a literal number.

Line 110
RTS

("ReTurn from Subroutine")

If the mnemonic RTS is used at the end of a subroutine, it works like the RETURN instruction in BASIC; it ends the subroutine and returns to the main body of a program, beginning at the line following the line in which the RTS instruction appears. But if the RTS is used at the end of the main body of a program, as it is here, it has a different function. Then, instead of passing control of the program to a different line, it terminates the whole program and returns control of the computer to the input device that was in control before the program began, usually a cartridge, an operating system, a keyboard screen editor, or a machine language monitor.

Line 120
".END" Directive

Just as the "*=" directive begins an assembly language program, the "*.END*" directive ends it. The .END directive tells the assembler to stop assembling, and that's exactly what the assembler does, even if there's more source code after the .END directive. The .END directive can therefore be used as a powerful debugging tool. You can put it wherever you want in a program you're debugging, and that's where the assembler will always stop assembling, until you remove the ".END" directive. When you've finished debugging your program you can use the .END directive to bring the routine neatly to an end. Before you can do that, of course, you must remove any leftover .END directives that may still be hanging around, that is, if you want your final program completely assembled. When debugging is complete and your program is finished, it should contain only one .END directive, where it belongs — at the very end of your program.

Assembling an Assembly Language Program

OK. Are you ready to write and run your first assembly language program? Good. Then sit down at your computer, turn on your disk drive and your video monitor, and boot up your assembler (if it's on a disk), or slip your assembler cartridge (if that's the kind of assembler you've got) into the cartridge slot in your Atari. If you're using a MAC/65 assembler or an Atari Assembler Editor cartridge, your assembler will be ready to go when the word "EDIT", the assembly language equivalent to BASIC's "READY" prompt, appears on your video monitor. If that doesn't happen, then check all of the connections on your computer components and repeat the start-up process. Until you get an "EDIT" prompt, you can't do any assembly language programming.

When you have your assembler up and running you can put a blank, formatted disk into your disk drive. This disk should have a set of DOS files recorded on it so that your data disk will boot automatically when you turn your computer on, without any

need for a special master disk. If you put every program in this book on your data disk, you'll still have room for a set of DOS files, and they'll save you quite a bit of time.

Entering Your Program

When the "EDIT" prompt comes up on your screen, you can type the addition program (source code version) at the beginning of this chapter into your computer. As you type the program, be very careful about the spacing you use. MAC/65 and the Atari Assembler Editor cartridge are a bit fussy about spacing. In the lines that contain semicolons, remember that there should be only one space between the line number and the semicolon. In line 40, however, there should be at least *two* spaces between the line number and the asterisk, since "*" is a *directive*, and since directives appear in the op code field of MAC/65 and Atari Assembler Editor programs.

In line 60, there should be one space between the line number and the "ADDNRS" label, and one space between "ADDNRS" and the mnemonic "CLD." In lines 70 through 110 there should be at least two spaces between each line number and the op code that follows. And in line 120 there should be at least two spaces between the line number and the ".END" directive. If you make a mistake while typing a line, you can move back and correct it, using the cursor control (arrow) keys on your keyboard.

Listing Your Program

All right now. After you've typed your source listing of Program 1 into your computer, type the word LIST and you should see a screen display that looks like this:

```
EDIT
LIST

10 ;
20 ;ADDNRS.SRC
30 ;
```

```
40          *= $0600
50 ;
60 ADDNRS CLD
70          CLC
80          LDA #2
90          ADC #2
0100        STA $CB
0110        RTS
0120        .END

EDIT
```

If you have a printer you can now print your program out on paper. Just type:

```
LIST #P:
```

Easy enough, right? No sooner typed than done! Now, if your listing looks all right, you can save your program on a disk. Just make sure that a formatted disk (preferably a blank one) is in your disk drive and the ready light is on. Then type:

```
LIST #D:ADDNRS.SRC
```

The top red light on your disk drive should now go on, and the disk you're storing your program on should start to spin. When your disk drive's "busy" light goes off, your source code should be safely recorded on a disk under the file name #D:ADDNRS.SRC. We suggest you keep your ADDNRS.SRC source code in a safe place, since we'll be working some more with this program in later chapters. Right now, in fact, you can use that very source code to assemble your program. To do that, just keep your source code disk in your disk drive, keep your Assembler Editor cartridge in your computer, and type the command ASM. As soon as you've done that, your computer should present you with a screen display that looks something like this:

```
EDIT
ASM
PAGE 1
```

```
                        10 ;
                        20 ;ADDNRS.SRC
                        30 ;
0000                    40          *= $0600
                        50 ;
0600 D8                 60 ADDNRS CLD
0601 18                 70          CLC
0602 A902               80          LDA #2
0604 6902               90          ADC #2
0606 85CB               0100        STA $CB
0608 60                 0110        RTS
0609                    0120        .END

*** ASSEMBLY ERRORS: 0      23202 BYTES F[1]
REE
  PAGE 2
SYMBOLS

 0600 ADDNRS

EDIT
```

If you've made any typing errors in your program, this is where you'll probably find out about them. If your assembler finds an error in a line, it will sound a beep and display an error message. It may not be able to spot *every* error you make, but when it does catch one, it will print an error number on your screen (just like a BASIC interpreter does), and you can find out what the number means by consulting your MAC/65 or Atari Assembler Editor user's manual. If your assembler finds any errors in your program you can now type LIST and go back and correct them. Then you can type ASM again and try once more to assemble your program. Once your object code listing has been printed out on your screen without any error messages, you'll know that your program has been assembled correctly; and that you have just written and assembled your first assembly language program!

[1]This depends on the version of DOS that you have.

A Difference Between Assemblers

Now we've come to an important difference between the Atari Assembler Editor cartridge and the MAC/65 assembler. When you assemble a source code program using the Atari Assembler Editor cartridge, the object code generated by the assembly process is automatically entered into your computer's memory. When you assemble a program using the MAC/65 assembler, however, the object code is not automatically stored in memory unless you use a special directive. That directive is .OPT (for "OPTion"). The option directive is used in this format:

```
Ø5.OPT OBJ
```

If you have a MAC/65 assembler, you can insert that line into your ADDNRS.SRC program, and the program will automatically be stored in RAM as it is assembled.

What Next?

Once you've stored a program in RAM, you can do just about anything with your program you like; run it, print it on paper, store it on a disk, or store it in your computer's memory. Before you do any of those things, however, it might be a good idea to take a closer look at the ADD.NRS program, the object code listing of the program in its final assembled form. In *column 1* of your object code listing you'll see the memory addresses in which your addition will be stored after it has been loaded into your computer's memory. *Column 2* is the actual object code listing of your program. It's this column that will show you, in hexadecimal notation, the actual machine code version of the program. This is what the numbers in column 2 mean:

SOURCE CODE	MACHINE CODE	MEANING
CLD	D8	Clear status register's decimal mode flag
CLC	18	Clear status register's carry flag
LDA #2	A902	Load accumulator with the number 2

SOURCE CODE	MACHINE CODE	MEANING
ADC #2	6902	Add 2, with carry
STA $CB	85CB	Store result in memory address CB (decimal 203)
RTS	60	Return from subroutine

Now, if you like, you can print your assembly listing out on paper. Simply type ASM,#P: and you'll get a hard copy listing of your assembled program.

Saving Your Program

You've now written, assembled and printed your first assembly language program. And that means that we're almost ready to end this programming session. But before you turn off your computer, it wouldn't be a bad idea to save the object code of our program on a disk. So let's do that right now.

"LIST" or "SAVE"?

A few paragraphs back you saved the source code of your addition program with the command LIST. But to save the object code of an assembly language code, there are two other commands. One is SAVE. The other is BSAVE. If you're using a MAC/65 assembler, the command to use at this point to to save the object code version of ADDNRS is BSAVE. If you're using the Atari Assembler Editor, however, the command to use is SAVE.

Saving an Object Code Program

The SAVE and BSAVE commands are used in exactly the same way, however. Here's how it's done: First, type the command ASM, and your computer will assemble the source code of your

addition program. (If you've already done that, there's no harm in doing it again). Next, look at column 1 of the object code listing on your screen and note the memory addresses that your program has been assembled into. The program should start at memory address $0600 and should end at memory address $0609. So this is the line to type if you have a MAC/65 assembler:

```
BSAVE #D:ADDNRS.OBJ<0600,0609
```

And this is the line to type if you have an Atari Assembler Editor cartridge:

```
SAVE #D:ADDNRS.OBJ<0600,0609
```

As soon as you type that line and hit your RETURN key, the "busy" light on your disk drive should light up, and your disk should start to spin. When the "busy" light goes off and your disk drive stops, the object code listing of our addition program should be safely stored on a disk under the file name #D:ADDNRS.OBJ.

Did it Work?

Now let's check to see whether all of that LISTing and SAVEing worked. First type the word NEW and press the RETURN key to clear your computer's memory. Then type "ENTER #D:ADDNRS.SRC." Your disk drive should spin, and the word EDIT should appear on your screen. Now, if you type the word LIST, the source code listing for the ADDNRS program should come right up on your video screen. Now, if your assembler is a MAC/65, type the line "BLOAD #D:ADDNRS.OBJ," and hit the RETURN key. If you have an Atari Assembler Editor cartridge, type "LOAD #D:ADDNRS.OBJ." Your disk drive should now load the object code of the ADDNRS program into your computer, and the word EDIT should appear once more. Then, if you type the command ASM again, the assembly code listing of the ADDNRS program should appear on your video screen.

Moving Along

You've accomplished quite a bit in this chapter. You've written and assembled your first assembly language program and, hopefully, you have a pretty good understanding of how it was all done. You've saved both your source code listing and your assembly code listing on a disk. If you have a printer, you've also printed out both listings on paper. And now, in Chapter 5, you're going to learn how to *run* an assembly language program.

Chapter Five

Running an Assembly Language Program

There are several ways to execute a machine language program on an Atari computer system. For example, you can run a machine language program by:

- Using a special debugging command (the "G" command) provided by both the MAC/65 assembler and the Atari Assembler Editor cartridge.
- Running the program using the Atari disk operating system (DOS) or (if you have a MAC/65 assembler) the OS/A+ operating system.
- Using the AUTORUN.SYS utility of Atari DOS (or a STARTUP.EXC file if you're using the OS/A+ operating system).
- Calling your machine language program from a BASIC program.

In this chapter, we'll cover the first three of these methods of running machine language programs. The fourth technique, calling assembly language programs from BASIC, will be covered in Chapter 8. First we'll discuss the technique for running a program with the "G" command offered by the MAC/65 assembler and the Atari Assembler Editor cartridge.

Your Assembler's Built-in Monitor

To use the "G" command, you'll need the help of a handy tool that's provided free with both the MAC/65 assembler and the Atari Assembler Editor cartridge. That tool is called a debug-

ging utility. If you've just finished Chapter 4 and *still* have your computer turned on, you can start using your assembler's debugging facility in just a few moments, as soon as readers who've turned their computers off between chapters have had a chance to get their machines back into action again.

If you've turned off your computer since the end of Chapter 4, please get it up and running again. You'll need your data disk in place and your assembler turned on. When the EDIT prompt appears on your video screen, you can load the source code listing of the program that you wrote in Chapter 4 into your computer's memory. Just type in the word "NEW", a good habit to get into when you want to load a program, just in case there may already be a program in memory. Then type

```
ENTER #D:ADDNRS.SRC
```

— just as you did when you loaded the ADDNRS.SRC program at the end of Chapter 4. When your disk drive stops spinning, you can check to see whether the program has been loaded correctly by simply typing the command:

```
LIST
```

You should then see the program listed on your screen. Now let's assemble the program. (Actually, if you did the exercises in Chapter 4, our ADDNRS program is already assembled, and stored your data disk in its assembled form. But the program is so short that it would take more time to load its object code into memory from a disk than it would take to assemble it again. So we're going to assemble it again right now.) If you're using a MAC/65 assembler, take a look at the source code listing of your program and make sure it contains the line

```
Ø5  .OPT OBJ
```

— so that it will be loaded into your computer's memory as it is assembled. (If you're using an Atari Assembler Editor cartridge, line 5 should not be in your program! The .OPT directive means nothing to the Atari Assembler Editor cartridge, and will be flagged as an error!)

Once line 5 is either in your program or out of it, depending on what kind of assembler you have, you can assemble the ADDNRS program. To do this, simply type:

```
ASM
```

Your assembler will present you with an object code listing of the ADDNRS program.

Then you can use the debugging facility built into your MAC/65 or Atari assembler to debug your program and save it on a disk in its final form. The debugging facilities of the MAC/65 assembler and the Atari Assembler Editor are quite similar. But there are a few differences.

Using the MAC/65 Debugger

The debugger built into the MAC/65 assembler is called BUG/65. To use it you must first make sure that your ADDNRS source code has been properly assembled using the ".OPT OBJ" directive. Then, while your assembler is in its EDIT mode, type the command "CP". That will return you to your assembler's OS/A+ operating system. Now, in response to the OS/A+ prompt "D1:", type "BUG65". Your disk drive should start spinning, and when it stops, the yellow bordered BUG/65 screen should be displayed on your computer monitor.

Using the Atari Assembler's Debugger

If you're using an Atari Assembler Editor cartridge, putting your assembler into its DEBUG mode is even easier. Just type

```
BUG
```

— (*not* DEBUG), followed by a carriage return. This will present you with a screen display that says

```
DEBUG
```

— and when that command appears on your screen, you can then debug assembly language programs using a whole host of commands.

In this chapter, we'll be discussing only a few of the many capabilities of your assembler's debugging package. Your assembler's debugger is a very special kind of software package. With it, you can PEEK into your computer's memory registers, and display the contents of those registers in many different ways. You can even run programs using your assembler's debugger, which can alert you to many kinds of programming errors, both while the program is running and after it has run.

Using Your Debugging Package

We're now going to show you how the monitor built into your assembler can help you examine the contents of your Atari's RAM. Then you'll get a chance to run a machine language program using your assembler's built-in monitor.

Listing the Contents of Memory Locations

As we've pointed out, all of the capabilities of your assembler's built-in monitor are accessible from the assembler's DEBUG mode. When you're in DEBUG mode, for example, you can take a look at the contents of any memory locations you like by using the instruction "D," which stands for "display memory." The "D" command is similar to the PEEK command in BASIC. By using the "D" command, you can peek into your Atari's memory registers and see what their contents are. To use the "D" command, all you have to tell your assembler is what memory locations you want it to peer into. If you type a "D" followed by a memory address (expressed as a hex number, of course), you'll get an onscreen listing of the requested location and the next seven locations that follow it. If you have assembled your ADDNRS source code listing, you can take a look at how your monitor's "D" command works right now. Simply type

```
D600
```

— and you should see a screen display that looks something like this:

```
0600 D8 18 A9 02 69 02 85 CB
```

(If you have a MAC/65 assembler, there'll be a few extra characters after the letters "CB". Don't pay attention to them, they're the printed forms of characters that could be represented by the numbers in this line under certain circumstances, but mean nothing in the context of what we're doing right now.) The rest of that line, as you can see by glancing at one of the object code listings we've created in other exercises, is nothing but a stripped-down machine code listing of the first eight bytes of your ADDNRS.OBJ program. You can also use your monitor's "D" command to look at more than eight consecutive locations in your computer's memory. Just type two addresses after the "D", using the format

```
D5000 500F
```

— if you have a MAC/65 assembler, and the format

```
D5000,500F
```

— if you have an Atari Assembler Editor cartridge.

Your assembler's debugger will then provide you with a list of the contents of all registers from the first address listed to the second address. To see how the "D" command works when you use it optional second parameter, just type:

```
D0600 0608
```
(or D0600,0608 if you have an Atari Assembler)

You should get a listing something like this (with some extra symbols tacked on if your assembler is a MAC/65):

```
0600 D8 18 A9 02 69 02 85 CB
0608 60 00
```

That, of course, is a disassembled listing of your complete addition program, all the way down to its last mnemonic, the RTS instruction.

The Atari Assembler Editor's 'L' Command

The Atari Assembler Editor is also equipped with an "L" (List Memory With Disassembly) command. (The MAC/65 assembler also has an "L" command, but it's entirely different. The MAC's "L" command is used for locating hexadecimal strings.) But the Atari assembler's "L" command can be used to display *disassembled* listings of machine language programs. The Atari Assembler Editor's "L" command is similar to the "D" command, but there are some differences. The Atari "L" command, like the "D" command, can be used with either one or two addresses. When you use one address, "L" will list the contents of 20 consecutive memory locations in your computer (not just the contents of eight locations, as the "D" command does).

Whether you use that optional second address or not, "L" will *disassemble* the machine code at the addresses it lists; alongside each hex number listed, it will also list the assembly language instruction that the number equates to, if any. To get a look at how the Atari "L" command works, type the following on an Atari Assembler.

```
L0600,0608
```

If you try to enter that line on a MAC/65 assembler, you'll get nothing but an angry beep and a "COMMAND ERROR!" message. But if have an Atari assembler, this is what you should see:

```
0600    D8      CLD
0601    18      CLC
0602    A9 02   LDA     #$02
0604    69 02   ADC     #$02
0606    85 CB   STA     $CB
0608    60      RTS
```

This is a listing of the actual contents of memory addresses $0600 through $0608 of your computer, after your ADDNRS.SRC source code listing has been assembled and stored in RAM. By

merely looking at this listing, you can see that your program has been assembled and loaded into RAM correctly, and is now just sitting in RAM and waiting to be run. Now we've come to a debugging command that can actually be used to run a program, the "G" (for "Go," or "Execute") command.

Golly, G

Fortunately, the "G" command can be used on both the MAC/65 assembler and the Atari Assembler Editor cartridge. By using the "G" command, you can instruct your computer to execute machine language code that begins at any specified memory location. If you have an Atari Assembler Editor cartridge, it's very easy to use the "G" command. While your assembler is in its DEBUG mode, just type the letter G, immediately followed by the memory address at which a program or routine starts. Your computer will then run the program or routine that starts at the specified address. If you're using an Atari Assembler Editor, you can use the "G" command to run your ADDNRS program right now. Just type

```
G0600
```

The program should then run.

If you have a MAC/65 assembler, the "G" command is a little more powerful and, in this case, requires an additional parameter. When you use the MAC's "G" command, you can (and in this instance should) use both an initial address and a termination (or "breakpoint") address for the routine you want to run. And when you type your programs breakpoint parameter, you must flag it with the prefix "@". For example, here's how to use the "G" command to execute the ADDNRS program using the MAC/65 debugger:

```
G 0600 @0608
```

A number of other ways in which the "G" command can be used are listed in the MAC/65 and BUG/65 user's manual.

No Bells and Whistles

If your ADDNRS program runs without any hitches using the MAC/65 or Atari "G" command, you won't see much action on your computer screen. The program will just quietly do its job, which is adding 2 and 2, and then, quick as a wink, it will return control of your computer to you. At that point, what you'll see on the screen is a display of your Atari's internal registers. If you have an Atari Assembler Editor, your screen display will look like this:

```
A=1C X=00 Y=00 P=30 S=04
```

If you have a MAC/65 assembler, the display will be a little more complicated. It will look more like this:

```
 A   X   Y  SP  NV_BDIZC    PC   INSTR
04  00  00  00  00100000  0608   RTS
```

Both of the above displays have the same general function. Both tell us that your program has finished running and has left some values in five of the 6502 chip's status registers: the accumulator, the X register, the Y register, and processor status register, and the stack pointer. The line displayed by the MAC/65 debugger also lists the address that was in your computer's program counter (PC) when the program reached its breakpoint. The last item in the line tells what the instruction (INSTR) at that address was.

All of this information can be useful in some debugging applications, since it's sometimes helpful to know what condition the 6502 chip has been left in after a program has been run. But it doesn't mean a great deal to us right now, since our addition program has finished running and we don't really care what the 6502's registers now contain. What's more important to us at the moment is whether or not our program did the job it was supposed to do. The only way we can find that out is to look and see whether the program did what we instructed it to do, namely, whether it added 2 and 2, and stored the result of that calculation in memory address $CB.

While your assembler is in its "debug" mode, you can easily

PEEK inside memory address $CB and see if the sum of 2 and 2 is stored there. Simply type:

```
DCB
```

You should see one or two lines something like these on your video screen (again, with a few minor and not especially significant differences if our assembler is a MAC/65):

```
00CB 04 00 00 00 00
00D0 00 79 00
```

Success! The number 4, the sum of 2 and 2, is indeed stored in memory address $CB!

Saving a Machine Language Program

Back in Chapter 4, you learned how to save both source code listings and object code listings of assembly language programs. In Atari assembly language, source code listings are loaded into memory using the ENTER command, and are saved to disk using the LIST command. If you have a MAC/65 assembler, you can also load source code into memory using command LOAD, and you can save source code listings to disk using the command SAVE. When you use a MAC/65 assembler, ENTER and LIST are used to save and load source code programs in their "untokenized," or unabbreviated form, LOAD and SAVE are used to load and save source code listings in their "tokenized," or abbreviated, form. This is the same system used for loading and saving programs written in Atari BASIC.

If you have an Atari Assembler Editor cartridge, the commands LOAD and SAVE are used for a completely different purpose. In programs written with the Atari Assembler editor, the LOAD and SAVE commmands are reserved for loading and saving object code, so they can't be used at all for loading and saving source code listings. Object code listings are loaded into memory using the commands LOAD (with the Atari Assembler) or BLOAD (with MAC/65), and are saved to disk using the commands SAVE (with the Atari cartridge) or BSAVE (with MAC/65). Source code

programs must be loaded into memory and saved to disk while your assembler is active and in its edit mode. But object code programs can be loaded and saved in two different ways.

You can load and save object code programs while your assembler is active and in its edit mode. You can also load and store object code programs using either the Atari or OS/A+ disk operating systems. To save an object code program using the Atari DOS menu, all you have to do is select Menu Option K, the "Binary Load" command. To save an object code program using the OS/A+ operating system, the correct format is:

```
[D1:]SAVE ADDNRS.OBJ 0600 0608
```

I know this is all quite confusing, but at least you have this chapter to guide you, which is more than I had when I was trying to learn Atari assembly language!

Writing Programs That Will Run When Loaded

You can use both the Atari DOS menu and the OS/A+ operating system to save machine language programs in such a way that they will automatically run as soon as they have been loaded into your computer's memory. When you select option "K" on the Atari DOS menu, your computer responds with this prompt:

```
SAVE — GIVE FILE, START, END, INIT, RUN
```

If you wish, you can respond to this prompt by typing only two addresses: START and END. But the prompt also allows you to use two more addresses: INIT and RUN. These two addresses are *optional parameters*. When you save a program without using these parameters, the program you save will load, but not run when you retrieve it from a disk. To run a program that has been saved without using the INIT or RUN parameters, a special "execute" command must be used: either Option M (RUN AT ADDRESS) on the Atari DOS menu or certain special commands (such as the "G" command) that are available on various assemblers, debuggers, and operating systems.

If you like, you can use Atari DOS to save an object code program in such a way that it will automatically run when it is loaded into memory. In fact, if you wish, you can even save an object code program in such a way that it will run as soon as a disk on which it is stored is booted. To flag an object code program so that it will automatically run when it's loaded into memory, you can save it from DOS using the INIT parameter, the RUN parameter, or both. The INIT and RUN parameters do slightly different things so, not surprisingly, they are used for slightly different purposes.

When you use INIT, your program will start running at its INIT address *as soon as that address is loaded into memory*. When you use the RUN parameter, your program will start running at that address, but *not until the entire program has been loaded into memory*. The INIT parameter is usually used for running short routines within a program while the program is being loaded. For example, converting text strings from one kind of character code to another, so that the conversions will all be complete by the time the program runs. The RUN parameter is used to run the entire program after it has been loaded into memory.

Using the Run Parameter

This is how you would store the ADDNRS.OBJ program with an Atari Assembler Editor cartridge using the RUN parameter but not the INIT parameter:

```
ADDNRS.OBJ,Ø6ØØ,Ø6Ø8,,Ø6ØØ
```

Notice the two commas between the numbers 0608 and 0600 in this example. They mean that the INIT instruction has been left blank, and thus has not been used. If it had been used, it would be the third number typed, right between the commas. Instead, the line has been typed using this format:

```
ADDNRS.OBJ,START,END,,RUN
```

The program will therefore run automatically, beginning at $0600 (the address entered under the RUN parameter), as soon as the entire program is loaded into memory. If you saved your program

using this format, and then loaded the program into your computer's memory, it would start loading at address $0600 and would stop loading at address $0608. It would then start running at address $0600.

Using the 'INIT' Parameter

The "INIT" parameter of the Atari Assembler Editor's Binary Load command can be used either by itself or in conjunction with the "RUN" parameter. You can use the "INIT" command as many times as you like in a program, for each portion of the program that you want to run as it is being assembled. The "RUN" command may be used only once, to run the entire program. Detailed instructions for using the RUN and INIT commands can be found in your *Atari Disk Operating System II Reference Manual.* For our purposes, all you really need to know right now is that you can save object code programs in such a way that they'll run after they are loaded by using the optional "RUN" parameter of the binary load command.

Running Machine Language Programs Using OS/A+

OS/A+ doesn't provide any INIT or LOAD parameters for running machine language programs because it doesn't require any. To run a machine language program using OS/A+, all you have to do is use the "RUN" command. To use the OS/A "RUN" command, just respond to the "D1:" prompt by typing the word RUN, followed by the starting address of the program you want to run. For example:

```
[D1:] RUN Ø6ØØ
```

The binary file stored at that address will then run.

Writing Self-booting Programs

Have you ever wanted to write a program that will boot itself and then run itself automatically, as soon as you turn your computer on? Well, you can do that very easily, if you know how to use

assembly language! All you have to do is save the program using the AUTORUN.SYS utility built into Atari DOS (or the STARTUP.EXC utility provided by OS/A+, which we will discuss in a moment). Use either of these utilities, and your program will run automatically each time the disk it is saved on is booted, just like a piece of professionally written software!

Two Ways to do It

There are a number of ways to use Atari's AUTORUN.SYS. One way simply is to save your program as a self-booting program, using the INIT parameter, the RUN parameter or both, under the file name AUTORUN.SYS. Another way is to take a program that you've saved as an automatically running program, and simply change its name to AUTORUN.SYS. Here's how to change the name of a file to AUTORUN.SYS using the Atari Assembler Editor cartridge. First, make sure that the program you want to convert is already a self-running program. In other words, make sure that it was saved using either the RUN or INIT parameter, or both. Then call up your DOS menu and type "E" for RENAME FILE. You should then see this prompt:

```
RENAME — GIVE OLD NAME, NEW
```

In response to this prompt, type:

```
ADDNRS.OBJ,AUTORUN.SYS
```

That's all there is to it! From now on, each time you boot the disk that Program 1 is on, the program will run automatically (you can use your Assembler Editor cartridge's monitor, if you like, to assure yourself that it's true).

Using the OS/A+ STARTUP.EXC Utility

The STARTUP.EXC utility provided by OS/A+ is very similar to the AUTORUN.SYS offered by Atari DOS. To use the OS/A+ STARTUP.EXC utility, just boot OS/A+ and then replace your OS/A+ master disk with a data disk on which the object code of

your ADDNRS program is stored. When your data disk is in place, respond to the OS/A+ "D1:" prompt by typing:

```
TYPE E: D1:STARTUP.EXC
```

This is the line you should then see on your screen:

```
D1: TYPE E: D1:STARTUP.EXC
```

When you type this line, be sure to use the exact spacing that we used in these examples. In particular, make sure that there's a space between "E:" and "D1:STARTUP.EXC." OS/A+, like the MAC assembler and the Atari Assembler Editor, is rather fussy about spacing. When you're sure you've typed the line correctly, hit your RETURN key and your computer screen will go blank for a moment. When the lights come back on, you'll see a blank screen with a cursor in the upper left-hand corner. When the blank screen and cursor appear, simply type the word "LOAD", followed by the name of the file that you want to convert into a self-booting file. For example:

```
LOAD ADDNRS.OBJ
```

Next, type the word "RUN", followed by the address (in hexadecimal notation) at which the first instruction in your program is located. For example:

```
RUN 0600
```

At this point, these two lines should be all you see on your screen:

```
LOAD ADDNRS.OBJ
RUN 0600
```

Now type RETURN, followed by [CONTROL] 3 (which is typed, of course, by pressing the CONTROL key and the "3" key simultaneously). When you've done that, your data disk should start to spin. When it stops, the ADDNRS.OBJ program should be stored on your disk as a self-booting file. When you've finished creating your STARTUP.EXC file, you can check to see if it's really on

your disk by typing the command "DIR" to get a disk directory. Then, if the STARTUP.EXC file is there, you can check to see if it works by turning your computer off and then turning it on again. When your computer is up and running again, load your debugger into memory by typing the command BUG65. Then you can use your debugger's "D" command to check memory registers $600 to $608 to see if your program loaded, and memory register $CB to see if it executed properly.

Calling Machine Language Programs from BASIC

You can also run machine language programs by calling them from BASIC programs. But it's a complex process, requiring an understanding of some fairly sophisticated programming techniques. So we're going to save our explanation of calling machine language programs from BASIC for Chapter 8, which will be completely dedicated to mixing assembly language and BASIC programs.

Chapter Six

The Right Address

We've covered a lot of ground in the first five chapters of this book. You now have a pretty good idea of how your computer works, and you know what goes on inside your Atari's 6502 chip when a program is running. You now know the principles of the binary and hexadecimal number systems, and you know how to write, debug, load and save assembly language programs. But we've really just begun to explore the capabilities of 6502 assembly language.

The 6502 processor in your Atari computer is an incredibly versatile device. It has only seven registers, and it understands only 56 instructions. But with those limited facilities, it can do some amazing things. One reason the 6502 chip is so versatile is because it can access the memory locations in a computer in 13 different ways. In other words, the 6502 processor has 13 different *addressing modes.* In the world of assembly language, an *addressing mode* is a technique for locating and using information stored in a computer's memory.

In the programs presented in this book so far, we've used three addressing modes: implicit addressing, immediate addressing, and zero page addressing. In this chapter, we'll be examining all three of those addressing modes, along with the ten others that are available.

Every 6502 instruction must be using one of the addressing modes. Not one instruction is capable of using all of the addressing modes, and no addressing mode can be used by all the instructions. The instruction tells the processor what to do and the addressing mode tells the processor what to do it with. On the following page is a complete list of addressing modes and the format of the operation so you can tell what mode you are using. These formats are standard for the 6502 microchip so they should be understood by most 6502 assemblers.

The 6502's Addressing Modes

The 13 addressing modes of the 6502 processor are:

ADDRESSING MODE	FORMAT
1. Implicit (Implied)*	RTS
2. Accumulator	ASL A
3. Immediate*	LDA #2
4. Absolute	LDA $5000
5. Zero Page*	STA $CB
6. Relative	BCC LABEL
7. Absolute Indexed,X	LDA $5000,X
8. Absolute Indexed,Y	LDA $5000,Y
9. Zero Page, X	LDA $CB,X
10. Zero Page, Y	STX $CB,Y
11. Indexed Indirect	LDA ($B0,X)
12. Indirect Indexed	LDA ($B0),Y
13. Indirect	JMP ($5000)

"ADDNRS.SRC" Revisited

The three instructions marked by * asterisks are the ones we've used in this book so far. All three appear in "ADDNRS.SRC," the 8-bit addition program introduced a few chapters back, which we'll take another look at now:

THE "ADDNRS" SOURCE PROGRAM

```
10 ;
20 ;8-BIT ADDITION PROGRAM
30 ;
40  *=$0600
50 ;
60 ADDNRS CLD ;IMPLIED ADDRESS
70  CLC ;IMPLIED ADDRESS
80  LDA #02 ;IMMEDIATE ADDRESS
90  ADC #02 ;IMMEDIATE ADDRESS
100  STA $CB ;ZERO PAGE ADDRESS
110  RTS ;IMPLIED ADDRESS
```

In this example, the three address modes used in the program are identified in the comments column. Let's look now at each of these three address modes.

Implicit (or Implied) Addressing

(Lines 60, 70 and 110)
Format: CLD, CLC, RTS, etc.

When you use *implicit addressing*, all you have to type is a three letter assembly language instruction; implicit addressing does not require (in fact does not allow) the use of an operand.

The instruction in an implied address is thus similar to an intransitive verb in English; it has no object. The address it refers to (if it refers to an address at all) is not specified, but merely *implied* by the the mnemonic itself. So no operand is required or allowed in implicit addressing. Op code mnemonics that can be used in the implicit addressing mode are BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

Immediate Addressing

(Lines 80 and 90)
Format: LDA #02, ADC #02, etc.

When *immediate addressing* is used in an assembly language instruction, the operand that follows the op code mnemonic is a literal number, not the address of a memory location. So in a statement that uses immediate addressing, a "#" sign, the symbol for a literal number, always appears in front of the operand. When an immediate address is used in an assembly language statement, the assembler does not have to peek into a memory location to find a value. Instead, the value itself is stuffed directly into the accumulator. Then whatever operation the statement calls for can be immediately performed. Instructions that can be used in the immediate address mode are ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA and SBC.

Zero Page Addressing

(Line 100)
Format: STA $CB,etc.

It isn't difficult to distinguish between a statement that uses immediate addressing and one that uses *zero page addressing*. In a statement that uses zero page addressing, the operand always consists of just one byte, a number ranging from $00 to $FF. And that number equates to an address in a block of RAM called *page zero*.

The "#" symbol is not used in zero page addressing because the operand in a statement that employs zero page addressing is always a memory location, never a literal number. So the operation called for in the statement is performed on the *contents* of the specified memory location, not on the operand itself. Zero page addresses use one-byte operands because that's all they need. As we just said, the memory locations they refer to are in a block of your computer's memory that's called, logically enough, page zero. And to address a memory location on page zero, a one-byte operand is all that's necessary.

Specifically, the memory block in your computer known as page zero extends from memory address $00 through memory address $FF. You could just as easily (and just as correctly) say that page zero extends $0000 to $00FF. But it isn't really necessary to use those extra pairs of zeros when you want to refer to a zero page address. When you follow an assembly language instruction with a one-byte address, your computer knows that the address is on page zero. Since zero page addresses use memory saving one-byte operands, page zero is the high rent district in your Atari's RAM; it's such a desirable piece of real estate, in fact, that the people who designed your computer took most of it for themselves. Most of page zero is used up by your computer's operating system and other essential routines, and not much space has been left there for user written programs.

Later on in this book, in a chapter dedicated to memory management, we'll discuss the memory space available on page zero in more detail. For now, the most important fact to remember about page zero is that it's an address mode that uses a memory

address on page zero as a one-byte operand. Instructions that can be used with zero page addressing are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

New Addressing Modes

Now we'll describe the five 6502 address modes we haven't covered so far:

Accumulator Addressing

Format: ASL A

The *accumulator addressing mode* is used to perform an operation on a value stored in the 6502 processor's accumulator. The command ASL A, for example, is used to shift each bit in the accumulator by one bit position, with the leftmost bit (bit 7) dropping into the carry bit of the processor status (P) register. Other instructions that can be used in the accumulator addressing mode are LSR, ROL, and ROR.

Absolute Addressing

Format: STA $5000

Absolute addressing is similar to zero page addressing. In a statement that uses absolute addressing, the operand is a memory location, not a literal number. The operation called for in an absolute address statement is always performed on the value stored in the specififed memory location, not on the operand itself. The difference between an absolute address and a zero page address that an absolute address statement doesn't have to be on page zero; it can be anywhere in free RAM. So an absolute address statement requires a two-byte operand, not a one-byte operand, which is all that a zero page address requires.

This is what our ADDNRS.SRC program would look like if absolute addressing, instead of zero page addressing, were used:

The "ADDNRS" Source Program

(with absolute addressing in line 100)

```
10 ;
20 ;8-BIT ADDITION PROGRAM
30 ;
40  *=$0600
50 ;
60 ADDNRS CLD ;IMPLIED ADDRESS
70  CLC ;IMPLIED ADDRESS
80  LDA #02 ;IMMEDIATE ADDRESS
90  ADC #02 ; IMMEDIATE ADDRESS
100  STA $5000 ;ABSOLUTE ADDRESS
110  RTS ;IMPLIED ADDRESS
```

The only change that has been made in this program is the one in line 100. The operand in that line is now a two-byte operand, and that change makes the program one byte longer. But now the address in line 100 no longer has to be on page zero. Now it can be the address of any free byte in RAM.

Mnemonics that can be used in the absolute addressing mode are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

Relative Addressing

Format: BCC NEXT

Relative addressing is an address mode used for a technique called *conditional branching*, a method for instructing a program to jump to a given routine under certain specific conditions. There are eight conditional branching instructions, or relative address mnemonics, in 6502 assembly language. All eight begin with "B," which stands for "branch to." Examples of the conditional branching instructions that use relative addressing are:

BCC (Branch to a specified address if the Carry flag is Clear.)

BCS (Branch to a specified address if the Carry flag is Set.)

BEQ (Branch to a specified address if the Zero flag is Set.)

BNE (Branch to a specified address if the Zero flag is Clear.)

All eight of the conditional branching instructions will be described later on in this book in a chapter devoted to looping and branching.

What Comparison Instructions do

The eight conditional branching mnemonics are often used with three other instructions called *comparison instructions.* Typically, a comparison instruction is used to compare two values with each other, and a conditional branch instruction is then used to determine what should be done if the comparison turns out in a certain way. The three comparison instructions are:

CMP ("compare the number in the accumulator with...")
CPX ("compare the value in the X register with . . . ")
CPY ("compare the value in the Y register with . . . ")

Conditional branching instructions can also follow arithmetic or logical operations, and various kinds of testing of bits and bytes. Usually, a branch instruction causes a program to branch off to a specified address if certain conditions are met or not met. A branch might be made, for example, if one number is larger than another, if two numbers are equal, or if a certain operation results in a positive, negative, or zero value.

An Example of Conditional Branching

Here's an example of an assembly language routine that uses conditional branching:

```
AN 8-BIT ADDITION PROGRAM WITH ERROR
CHECKING

10 ;
20 ;8-BIT ADDITION WITH ERROR CHECKING
```

```
30 ;
40   *=$0600
50 ;
60 ADD8BTS CLD
70   CLC
80   LDA $5000
90   ADC $5001
100   BCS ERROR
110   STA $5002
120   RTS
130 ERROR RTS
```

This is an 8-bit addition program with a simple error checking utility built-in. It adds two 8-bit values, using absolute addressing. If this calculation results in a 16-bit value (a number larger than 255), there will be an overflow error in addition, and the carry bit of the processor status register will be set. If the carry bit is not set, then the sum of the values in $5000 and $5001 will be stored in $5002. If the carry bit is set, however, this condition will be detected in line 100, and the program will branch to the line labeled ERROR — line 130. At line 100, you could begin any kind of routine you wanted to: you might choose, for example, to write a routine that would print an error message on the screen. In this sample program, however, an error results only in an RTS instruction.

Absolute Indexed Addressing

Format: LDA $0500,X or LDA $0500,Y

An *indexed address*, like a relative address, is calculated by using an offset. But in an indexed address, the offset is determined by the current content of the 6502's X register or Y register. A statement containing an indexed address can be written using either of these formats:

```
LDA $5000,X
or
LDA $5000,Y
```

How Absolute Indexed Addressing Works

When *indexed addressing* is used in an assembly language statement , the contents of either the X register or the Y register (depending upon which index register is being used) are added to the address given in the instruction to determine the final address. Here's an example of a routine that makes use of indexed addressing. The routine is designed to move byte by byte through a string of ATASCII (Atari ASCII) characters, storing the string in a text buffer. When the string has been stored in the buffer, the routine will end. The text to be moved is labeled TEXT, and the buffer to be filled with text is labeled TXTBUF. The starting address of TXTBUF, and the ATASCII code number for a carriage return are defined in a symbol table that precedes the program.

Routine for Moving a Block of Text
(An Example of Indexed Addressing)

```
10 ;
20 ;ROUTINE FOR MOVING A BLOCK OF TEXT
30 ;
40 TXTBUF=$5000
50 EOL=$9B
70 ;
80   *=$600
90 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,
     $20
110   .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120   .BYTE $4C,$45,$41,$44,$45,$52,$21,$9B
130 ;
140 DATMOV
150 ;
160   LDX #0
170 LOOP LDA TEXT,X
180   STA TXTBUF,X
190   CMP #EOL
200   BEQ FINI
210   INX
```

```
220   JMP LOOP
230 FINI RTS
250   .END
```

Testing for a Carriage Return

When the program begins, we know that the string ends with a carriage return (ATASCII $9B), as strings often do in Atari programs. As the program proceeds through the string, it tests each character to see whether it is a carriage return or not. If the character is not a carriage return, the program moves on to the next character. If the character is a carriage return, that means there are no more characters in the string, and the routine ends.

Zero Page, X Addressing

Format: LDA $CB,X

Zero page,x addressing is used just like absolute indexed,x addressing. However, the address used in the zero page,x addressing mode must (logically enough) be located on page zero. Instructions that can be used in the zero page,x addressing mode are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, and STY.

Zero Page, Y Addressing

Format: STX $CB,Y

Zero page,y addressing works just like zero page,x addressing, but can be used with only two mnemonics: LDX and STX. If it weren't for the zero page,y addressing mode, it wouldn't be possible to use absolute indexed addressing with the instructions LDX and STX — that's the only reason that this addressing mode exists at all.

Indirect Addressing

There are two subcategories of indexed addressing: indexed indirect addressing, and indirect indexed addressing. Both indexed indirect addressing and indirect indexed addressing are

used primarily to look up data stored in tables. If you think the names of the two addressing modes are confusing, you're not the first one with that complaint. I never could keep them sorted out myself until I dreamed up a little memory trick to help eliminate the confusion.

Here's the trick: *Indexed indirect addressing*, which has an "x" in the first word of its name, is an addressing mode that makes use of the 6502 chip's X register. *Indirect indexed addressing*, which *doesn't* have an "x" in the first word of its name, uses the 6502's Y register. Now we'll look at each of your Atari's two indirect addressing modes, beginning with indexed indirect addressing.

Indexed Indirect Addressing

Format: ADC ($C0,X)

Indexed indirect addressing works in several steps. First, the contents of the X register are added to a zero page address — not to the contents of the address, but to the address itself. The result of this calculation must always be another zero page address. When this second address has been calculated, the value that it contains, together with the contents of the next address, make up a third address. That third address is (at last) the address that will finally be interpreted as the operand of the statement in question.

An Example of Indexed Indirect Addressing

An example might help clarify this process.

Let's suppose that memory address $B0 in your computer held the number $00, that memory address $B1 held the number $06, and that the X register held the number 0. Here are those equates in an easier to read form:

```
$BØ = #$ØØ
$B1 = #$Ø6

X = #$ØØ
```

Now let's suppose you were running a program that contained the indexed indirect instruction LDA ($B0,X). If all of those conditions existed when your computer encountered the instruction LDA ($B0,X), your computer would add the contents of the X register (a 0) to the number $B0. The sum of $B0 and 0 would, of course, be $B0. So your computer would go to memory address $B0 and $B1. It would find the number $00 in memory address $B0, and the number $06 in address $B1.

Since 6502 based computers store 16-bit numbers in reverse order, low byte first, your computer would interpret the number found in $B0 and $B1 as $0600. So it would load the accumulator with the number $0600, the 16-bit value stored in $B0 and $B1. Now let's imagine that when your computer encountered the statement LDA ($B0,X), 6502's X register held the number 04, instead of the number 00. Here is a chart illustrating those values, plus a few more equates that we'll be using shortly:

```
$B0 = #$00
$B1 = #$06
$B2 = #$9B
$B3 = #$FF
$B4 = #$FC
$B5 = #$1C

 X = #$04
```

If *these* conditions existed when your computer encountered the instruction LDA ($B0,X), your computer would add the number $04 (the value in the X register) to the number $B0, and would then go to memory addresses $B4 and $B5. In those two addresses, it would find the final address (low byte first, of course) of the data it was looking for, in this case, $1CFC.

A Rarely Used Mode

Indexed indirect addressing is not used in many assembly language programs. When it *is* used, its purpose is to locate a 16-bit address stored in a table of addresses stored on page zero. Since space on page zero is so hard to find, it's not very likely that you'll

ever be able to store many data tables there. So it's not too likely that you'll ever find much use for indexed indirect addressing.

Indirect Indexed Addressing

Format: ADC ($C0),Y

Indirect indexed addressing is not nearly as rare as indexed indirect addressing. In fact, it is quite often used in assembly language programs. Indirect indexed addressing uses the Y register (never the X register) as an offset to calculate the *base address* of the start of a table. The starting address of the table has to be stored on page zero, but the table itself doesn't have to be. When an assembler encounters an indirect indexed address in a program, the first thing it does is peek into the page zero address that is enclosed in the parentheses that precede the "Y." The 16-bit value stored in that address and the following address are then added to the contents of the Y register. The value that results is a 16-bit address, the address the statement is looking for.

An Example of Indirect Indexed Addressing

Here's an example of indirect indexed addressing:

Your computer is running a program and comes to the instruction ADC ($B0),Y. It then looks into memory address $B0 and $B1. In $B0, it finds the number $00. In $B1, it finds the number $50. And the Y register contains a 4. Here is a chart that illustrates those conditions:

```
$B0 = #$00
$B1 = #$50

 Y = #$04
```

If these states existed when your computer encountered the instruction ADC ($B0),Y, then your computer would combine the numbers $00 and $50, and would come up (in the 6502 chip's peculiar low byte first fashion) with the address $5000. It would

then add the contents of the Y register (4 in this case) to the number $5000, and would wind up with a total of $5004. That number, $5004, would be the final value of the operand ($B0),Y. So the contents of the accumulator would be added to whatever number was stored in memory address $5004.

Once you understand indirect indexed addressing, it can become a very valuable tool in assembly language programming. Only one address, the starting address of a table, has to be stored on page zero, where space is always scarce. Yet that address, added to the contents of the Y register, can be used as a pointer to locate any other address in your computer's memory. As you become more familiar with assembly language, you'll have many opportunities to see how indirect addressing works. You'll find a few examples of the technique in programs in this book, and you'll run across many more examples in other assembly language programs.

Indirect Addressing

Format: JMP ($5000)

In 6502 assembly language, unindexed indirect addressing can be used with only one mnemonic: JMP. One example of unindexed indirect addressing is the instruction JMP ($5000), which means, "Jump to the memory location stored in memory addresses $5000 and $5001."

The 'LIFO' Concept

The stack is what programmers sometimes call a "LIFO" (last in, first out) block of memory. It works like a spring loaded stack of plates in a diner; when you put a number in the memory location on top of the stack, it covers up the number that was previously on top. So the number on top of the stack must be removed before the number under it, which was previously on top, can be accessed.

How 6502 Uses the Stack

The 6502 processor often uses the stack for temporary data storage during the operation of a program. When a program jumps to a subroutine, for example, the 6502 chip takes the memory address that the program will later have to return to, and pushes that address onto the top of the the stack. Then, when the subroutine ends with an RTS instruction, the return address is pulled from the top of the stack and loaded into the 6502's program counter. Then the program can return to the proper address, and normal processing can resume. The stack is also used quite often in user written programs. Here is an example of a routine that makes use of the stack. You may recognize it as a variation on the 8-bit addition program that we've been using.

[1]**AN ADDITION ROUTINE THAT MAKES USE OF THE STACK**

```
10 ;
20 ;STACKADD
30 ;
40   *=$0600
50 ;
60 ;WHEN THIS PROGRAM BEGINS, TWO
70 ;8-BIT NUMBERS ARE ON THE STACK
80 ;
90 STKADD
100   CLD
105   CLC
110   PLA
120   STA $B0
130   PLA
140   ADC $B0
150   STA $C0
160   RTS
170    .END
```

This program is a simple, straightforward addition routine that shows how easy and convenient it can be to use the stack in assembly language programs. In line 110, a value is pulled from the stack and stored in the accumulator. Then in line 120, the

[1]Don't try to run this program until you understand the stack and how to prevent the program from crashing.

value is stored in memory address $B0. In lines 130 and 140, another value is pulled from the stack, and added to the value now stored in $B0. The result of this calculation is then stored in $C0, and the routine ends. That's only one short example of many ways in which the stack can be used.

You'll find other ways to use the stack in later chapters of this volume. If you take care to manage the stack properly, in other words, if you clear the stack after each use, it can be a very powerful programming tool. But, if you mess up the stack while you're using it, you're surely bound for trouble!

Mnemonics that make use of the stack are:

PHA ("push the contents of the accumulator onto the stack")

PLA ("pull the top value off the stack and deposit it in the accumulator")

PHP ("push the contents of the P register onto the stack")

PLP ("pull the top value off the stack and deposit it into the P register")

JSR ("put the current PC on the stack and jump to address")

RTS ("pull the return address off the stack and put it in the PC and increment it by one." This will cause execution to continue where it left off.)

The PHP and PLP operations are often included in assembly language subroutines so that the contents of the P register won't be wiped out during subroutines. When you jump to a subroutine that may change the status of the P register, it's always a good idea to start the subroutine by pushing the contents of the P register onto the stack. Then, just before the subroutine ends, you can restore the P register's previous state with a PHP instruction. That way, the P register's contents won't be destroyed during the course of the subroutine.

Chapter Seven

Looping Around and Branching Out

Now we're going to start having some fun with Atari assembly language. In this chapter, you'll learn how to print messages on the screen, how to encode and decode ATASCII (Atari ASCII) characters, and how to perform other neat tricks in assembly language. We're going to accomplish these feats with some advanced assembly language programming techniques that we haven't tried out so far, along with some new variations on techniques covered in earlier chapters. These are some of the programming techniques we're going to cover in this chapter:

- Using the assembly language .BYTE directive.
- Incrementing and decrementing the X and Y registers.
- Using comparison and branching instructions together.
- Advanced looping and branching.
- Writing relocatable assembly language instructions.

Before we get started, though, I'm going to pull a very sneaky trick. I'm going to ask you to type up and store on a disk a program that you haven't yet been introduced to. You probably won't understand it unless you've had previous experience in assembly language programming. I'm asking you to type this program because it contains a couple of subroutines that are needed to run two other programs, programs that will be introduced and explained in this chapter. The program you may not understand is one that will be explained in Chapter 12, I/O and You.

Two Good Reasons

There are two rather sophisticated but extremely useful subroutines in this program. One is a routine that will open your screen as an output device and put your Atari into its screen edit-

disk, you'll already have that job done the next time you encounter them, in I/O and You. So I hope you'll look ahead to greener pastures in the last chapter of this book, and not be too angry at me for asking you to type this program now.

PROGRAM FOR PRINTING ON THE SCREEN

```
10 ;
20  .TITLE "PRNTSC ROUTINE"
30  .PAGE "ROUTINES FOR PRINTING ON THE
      SCREEN"
40 ;
50   *=$5000
60 ;
70 BUFLEN=23
80 ;
100 EOL=$9B
105 ;
110 OPEN=$03
120 OWRIT=$08
130 PUTCHR=$0B
135 ;
140 IOCB2=$20
170 ICCOM=$342
180 ICBAL=$344
190 ICBAH=$345
200 ICBLL=$348
210 ICBLH=$349
220 ICAX1=$34A
230 ICAX2=$34B
235 ;
240 CIOV=$E456
250 ;
260 SCRNAM .BYTE "E:",EOL
270 ;
280 OSCR ;OPEN SCREEN ROUTINE
290   LDX #IOCB2
300   LDA #OPEN
310   STA ICCOM,X
320 ;
330   LDA #SCRNAM&255
```

```
340  STA ICBAL,X
350  LDA #SCRNAM/256
360  STA ICBAH,X
370 ;
380  LDA #OWRIT
390  STA ICAX1,X
400  LDA #0
410  STA ICAX2,X
420  JSR CIOV
430 ;
440  LDA #PUTCHR
450  STA ICCOM,X
460 ;
470  LDA #TXTBUF&255
480  STA ICBAL,X
490  LDA #TXTBUF/256
500  STA ICBAH,X
510  RTS
520 ;
530 PRNT
540  LDX #IOCB2
550  LDA #BUFLEN&255
560  STA ICBLL,X
570  LDA #BUFLEN/256
580  STA ICBLH,X
590  JSR CIOV
600  RTS
610 ;
620 TXTBUF=*
630 ;
640    *=*+BUFLEN
650 ;
660  .END
```

Now Save It!

When you have this program typed, you can assemble its object code and save it on a disk under the filename PRNTSC.OBJ. Then there's one other program I'd like for you to type and assemble. It's the one we'll be working with for the rest of this chapter.

```
10 ;
20 ;THE VISITOR
30 ;
35 TXTBUF=$5041
40 OPNSCR=$5003
50 PRNTLN=$5031
70 ;
80   *=$600
90 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,
     $20
110   .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120   .BYTE $4C,$45,$41,$44,$45,$52,$21
130 ;
140 VIZTOR
150 ;
160   LDX #0
170 LOOP LDA TEXT,X
180   STA TXTBUF,X
190   INX
200   CPX #23
210   BNE LOOP
220   JSR OPNSCR
230   JSR PRNTLN
240 INFIN JMP INFIN
```

This program is called (for reasons you'll soon discover) The Visitor. It's a program that's designed to print a cryptic message on your video screen.

Running 'The Visitor'

When you've finished typing The Visitor, you can run it immediately. Just assemble it, and then load the object code of the PRNTSC program into your computer. Then you can run The Visitor either by putting your computer into its DEBUG mode and typing G617, or by getting into DOS mode and running The Visitor with a DOS command. When you've finished trying out

The Visitor program, it wouldn't be a bad idea to save it, too, on a disk. The suggested file name for the program is VISITOR.SRC. After you've run and saved the program, you'll know exactly what it does. So now we can explain how it does what it has done. We'll start with an explanation of the assembly language .BYTE directive, which you'll see in lines 100 to 120.

The .BYTE Directive

The .BYTE directive is sometimes called a pseudo operation code, or pseudo op, because it appears in the op code column of assembly language source code listing but is not actually a part of the 6502 assembly language instruction set. Instead, it's a specialized directive designed that can be used with some assemblers, but not with others. For example, .BYTE works with both the MAC/65 assembler and the Atari Assembler Editor, but does not work with the Atari Macro Assembler and Text Editor. When

you write a program with the Atari Macro Assembler and Text Editor, you have to use the letters DB in place of the .BYTE directive. Other pseudo ops also differ from assembler to assembler. There are no generally accepted standards for writing pseudo op directives, so pseudo op codes designed for one assembler often won't work with another.

What the .BYTE Directive Does

When the .BYTE directive is used in a program created with the MAC/65 assembler or the Atari Assembler Editor, the bytes that follow the directive are assembled into consecutive locations in RAM. In the program called The Visitor, the bytes that follow the label TEXT are ATASCII (Atari ASCII) codes for a series of text characters.

Looping the Loop

As we explained in Chapter 6, the X and Y registers in the 6502 chip can be progressively incremented and decremented during loops in a program. In the Visitor program, the X register is incremented from 0 to 23 during a loop in which characters in a text string are read. The characters that to be read are written as ATASCII codes in lines 100 through 120 of the program. In line 160, the statement LDX #0 is used to load the X register with a zero. Then, in line 170, the loop begins.

Incrementing the X Register

The first statement in the loop is LDA TEXT,X. Each time the loop cycles, this statement will use indexed addressing to load the accumulator with an ATASCII code for a text character. Then, in line 180, the indexed addressing mode will be used again, this time to store the character in a text buffer. When the loop ends, all of the characters in the text buffer will be printed on the screen. The first time the program hits line 170, there will be a 0 in the X register (since a 0 has just been loaded into the X register in the previous line). So the first time the program encounters

the statement LDA TEXT,X, it loads the accumulator with the hexadecimal number $54 — what programmers sometimes call the "0th" byte after the label TEXT. (There's no need for a "#" symbol in front of the number $54, incidentially, since numbers that follow the .BYTE directive are always interpreted by the MAC/65 assembler and the Atari Assembler Editor as literal numbers.)

Incrementing and Decrementing the X and Y Registers

Now let's move on to line 190. The mnemonic you see there — INX — means "increment the X register." Since the X register currently holds a 0, this instruction will now increment that 0 to a 1. Next, in line 200, we see the instruction CPX #23. That means "compare the value in the X register to the literal number 23." The reason we want this comparison to be performed is so we can determine whether 23 characters have been printed on the screen yet. There are 23 characters in the text string that we're printing, and when we've printed all of them, we'll want to print a carriage return and end our program.

Comparing Values in Assembly Language

There are three comparison instructions in 6502 assembly language: CMP, CPX, and CPY. CMP means "compare to a value in the accumulator." When the instruction CMP is used, followed by an operand, the value expressed by the operand is subtracted from the value in the accumulator. This subtraction operation is not performed to determine the exact difference between these two values, but merely to test whether or not they are equal, and if they are not equal, to determine which one is larger than the other. If the value in the accumulator is equal to the tested value, the zero (Z) flag of the processor status (P) register will be set to 1. If the value in the accumulator is not equal to the tested value, the Z flag will be left in a cleared state.

If the value in the accumulator is less than the tested value, then the carry (C) flag of the P register will be left in a clear state. And if the value in the accumulator is greater than or equal to the tested

value, then the Z flag will be set to 1 and the carry flag will also be set. CPX and CPY work exactly like CMP, except that they are used to compare values with the contents of the X and Y registers. They have the same effects that CMP has on the status flags of the P register.

Using Comparison and Branching Instructions Together

The three comparison instructions in Atari assembly language are usually used in conjunction with eight other assembly language instructions — the eight conditional branching instructions that we mentioned in Chapter 6. The sample program that we have called The Visitor contains a conditional branching instruction in line 210. That instruction is BNE LOOP, which means "branch to the statement labeled LOOP if the zero flag (of the processor status register) is set." This instruction uses what can be a confusing convention of the 6502 chip. In the 6502's processor status register, the zero flag is set (equals 1) if the result of an operation that has just been performed is 0. And the zero flag is cleared (equals 0) if the result of an operation that has just been performed is not zero.

It Really Doesn't Matter

This is all quite academic, however, as far as the result of the statement BNE LOOP is concerned. When your computer encounters line 210, it will keep branching back to line 170 (the line labeled LOOP) as long as the value of the X register has not yet been decremented to zero. Once the value of the X register has been decremented to zero, the statement BNE LOOP in line 210 will be ignored, and the program will move on to line 220, the next line. In line 220, the program will jump to the subroutine OPNSCR — which currently resides in RAM beginning at memory address $5041, provided the object code for both PRNTSC and VISITOR have been loaded into your computer and are ready to run.

Conditional Branching Instructions

As we pointed out in the previous chapter, there are eight conditional branching instructions in 6502 assembly language. They all begin with the letter B, and they're also called relative addressing, or branching instructions. These eight instructions, and their meanings, are:

BCC — Branch if the carry (C) flag of the processor status (P) register is clear. (If the carry flag is set, the operation will have no effect.)

BCS — Branch if the carry (C) flag is set. (If the carry flag is clear, the operation will have no effect.)

BEQ — Branch if the result of an operation is zero (if the zero [Z] flag is set).

BMI — Branch on minus (if an operation results in a set negative [N] flag.

BNE — Branch if not equal to zero (if the zero [Z] flag isn't set).

BPL — Branch on plus (if an operation results in a cleared negative [N] flag).

BVC — Branch if the overflow (V) flag is clear.

BVS — Branch if overflow (V) flag is set.

How Conditional Branching Instructions are Used

To use a conditional branching instruction in 6502 assembly language, the usual method is to load the X or Y register with a zero or some other value, and then to load the A register (or a memory register) with a value to be used for a comparison. After that is done, a conditional branching instruction is used to tell the

computer what P register flags to test, and what to do if these tests succeed or fail. This all sounds very complicated — and it is. But once you understand the general concept of conditional branching, you can use a simple table for writing conditional branching instructions. Here's one such table.

TO TEST FOR:	DO THIS:	AND THEN THIS:
A = VALUE	CMP #VALUE	BEQ
A <> VALUE	CMP #VALUE	BNE
A >= VALUE	CMP #VALUE	BCS
A > VALUE	CMP #VALUE	BEQ and then BCS
A < VALUE	CMP #VALUE	BCC
A = (ADDR)	CMP $ADDR	BEQ
A <> (ADDR)	CMP $ADDR	BNE
A >= (ADDR)	CMP $ADDR	BCS
A > (ADDR)	CMP $ADDR	BEQ and then BCS
A < (ADDR)	CMP $ADDR	BCC
X = VALUE	CPX #VALUE	BEQ
X <> VALUE	CPX #VALUE	BNE
X >= VALUE	CPX #VALUE	BCS
X > VALUE	CPX #VALUE	BEQ and then BCS
X < VALUE	CPX #VALUE	BCC
X = (ADDR)	CPX $ADDR	BEQ
X <> (ADDR)	CPX $ADDR	BNE
X >= (ADDR)	CPX $ADDR	BCS
X > (ADDR)	CPX $ADDR	BEQ and then BCS
X < (ADDR)	CPX $ADDR	BCC
Y = VALUE	CPY #VALUE	BEQ
Y <> VALUE	CPY #VALUE	BNE
Y >= VALUE	CPY #VALUE	BCS
Y > VALUE	CPY #VALUE	BEQ and then BCS
Y < VALUE	CPY #VALUE	BCC
Y = (ADDR)	CPY $ADDR	BEQ
Y <> (ADDR)	CPY $ADDR	BNE
Y >= (ADDR)	CPY $ADDR	BCS
Y > (ADDR)	CPY $ADDR	BEQ and then BCS
Y < (ADDR)	CPY $ADDR	BCC

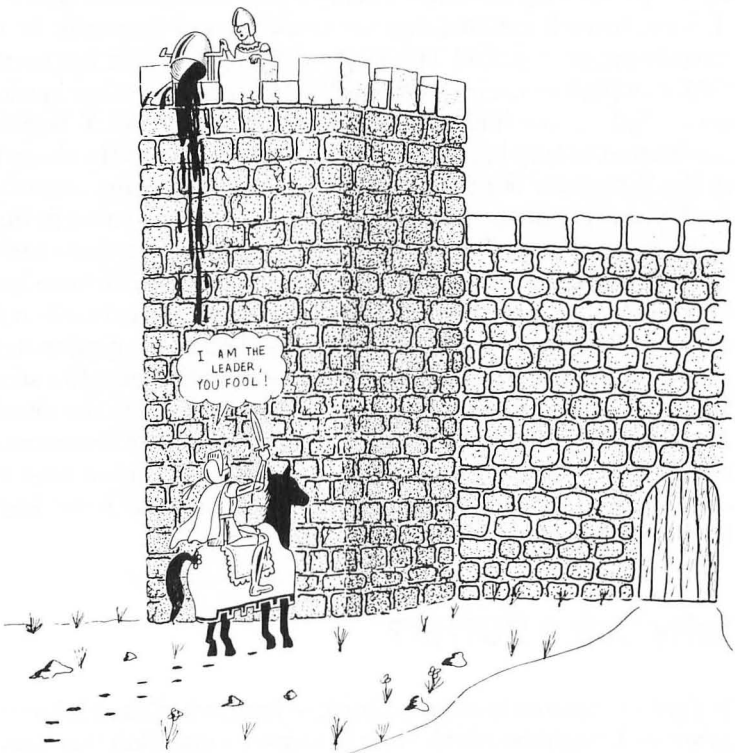
Assembly Language Loops

In 6502 assembly language, comparison instructions and conditional branch instructions are usually used together. In the sample program called The Visitor, the comparison instruction CPX and the branch instruction BNE are used together in a loop controlled by the incrementation of a value in the X register. Each time the loop in the program goes through a cycle, the value in the X register is progressively incremented or decremented. And each time the program comes to line 200, the value in the X register is compared to the literal number 23. When that number is reached, the loop ends. The program will therefore keep looping back to line 170 until 23 characters have been printed on the screen. Then, in lines 220 and 230, it will open your computer's screen — clearing it in the process — and will print the string that has been transferred into the text buffer on the screen. Finally, at line 240, the program will go into what's known as an infinite loop — cycling back to the same JMP instruction over and over again, and doing nothing else until you push the break key or in some other way halt the program.

Why Use a Buffer?

Before we move on to our next topic — improving The Visitor program — it might be worthwhile to answer a question that may or may not have occurred to you. The question is: Why use a text buffer? Why not just print the text in lines 100 to 120 directly onto the screen, without moving it first into a buffer and then out again?

Here is the answer to that question: Text can be loaded into a buffer in many ways: from a keyboard or from a telephone modem, for example, as well as being loaded in as data directly from a program. And, once a string is in a buffer, it can be removed from the buffer in just as many different ways. Another advantage of a text buffer is that you can load it into RAM, note its address, and use it from then on whenever you like. A buffer can therefore serve as a central repository for text strings, which will then be accessible with great ease and in many different ways.



Improving the Visitor Program

Now we're ready to make some improvements in the program called The Visitor. Not that the program doesn't work; it does, but it has certain limitations. And some of those limitations could be removed quite easily — as they have been in this new program, which I've called Response.

Response is quite similar to The Visitor — but, as you will soon see, significantly better in several ways:

RESPONSE

```
10 ;
20 ;RESPONSE
30 ;
40 TXTBUF=$5041
50 OPNSCR=$5003
60 PRNTLN=$5031
70 ;
80 EOL=$9B
90 ;
100  *=$650
110 ;
120 TEXT .BYTE "I AM the leader, you fool!",EOL
130 ;
140 RSPONS
150 ;
160  LDX #0
170 LOOP
180  LDA TEXT,X
190  STA TXTBUF,X
200  CMP #$9B
210  BEQ FINI
220  INX
230  JMP LOOP
240 FINI
250  JSR OPNSCR
260  JSR PRNTLN
270 INFIN
280  JMP INFIN
```

If you want to run the Response program — and I hope you do — you can type it into your computer's memory right now. Since it calls the same subroutines that its predecessor did, you can assemble it and run it as soon as it's typed, provided you still have the program called PRNTSC is loaded into RAM.

To run the Response program, you can either call your debugger program and use a G command, or go into DOS mode and use a DOS command. Whichever mode you decide to use, you should be able to run the program using a run address of $066B,

provided that you've typed it and assembled it in accordance with the suggestions that I've provided. Even if you've followed the instructions, though, you'll still encounter one small problem. Only the first 23 characters of the string "I AM the leader, you fool!" will print out on your computer screen. That's because the text buffer that we created in the program PRNTSC is only 23 characters long.

That flaw is easy to remedy. But before we fix it, it might be a good idea to save Response on a disk, in both its source code and object code versions. The program's suggested file names are RESPONSE.SRC and RESPONSE.OBJ.

Fixing the PRNTSC Program

Now we're ready to lengthen the text buffer used in the PRNTSC program, so that it will print its complete message on your computer screen. To lengthen the print buffer, just put your assembler into its editor mode and load the program's source code into your computer's memory. Then you can change line 70 from BUFLEN=23 to BUFLEN=40. When you've made this change, you can save the amended source code under the file name PRNTSC.SR2, assemble the program, and save the object code under the name PRNTSC.OB2 (to distinguish it from PRNTSC.SRC and PRNTSC.OBJ, you original PRNTSC programs).

When you've saved your PRNTSC.SR2 and PRNTSC.OB2 programs, you can reload the Response program and run it with PRNTSC.OB2 instead of PRNTSC.OBJ. This time you should see the full string that the Response program calls for on your video screen — but, unfortunately, you'll now notice something else wrong. After the line, "I AM the leader, you fool!" you'll see a line of little hearts. How did they get there? I'll answer that question in just a moment. But first, let's take a look at some of the differences between the program called The Visitor and the one called Response.

A Better Routine

From a technical point of view, the Response program is better than The Visitor — for several reasons. The most obvious difference between the two programs is the way they handle text strings. In the program called The Visitor, we used a text string made up of ATASCII codes. In Response, we've used a string made up of actual characters. That made the program much easier to write — and it makes it much easier to read, too.

Another important difference between our newest program and its predecessor is the way the loop is written. In the program called The Visitor, the loop counted the number of characters that had been printed on the screen, and ended when the count hit 23. Now that's a perfectly good system — for printing text strings that are 23 characters long. Unfortunately, it isn't so great for printing strings of other lengths. So it isn't a very versatile routine for printing characters on a screen.

Testing for a Carriage Return

The Response program is much more versatile than The Visitor because is can print strings of almost any length on a screen. That's because it doesn't keep track of the number of characters it has printed by maintaining a running count of how many letters have been printed on the screen. Instead, each time the program encounters a character, its tests the character to see whether its value is $9B — the ATASCII code for a carriage return, or end of line (EOL) character. If the character is not an EOL, the computer prints it on the screen and goes on to the next character in the string. If the character is an EOL, the EOL is printed on the screen and the routine ends. And that's that — except those pesky little hearts that we encountered when we ran the Response program. Now we'll take another look at those hearts, and see what we can do about them.

A String of Hearts

The hearts are there because the text buffer we have set up for our Visitor and Response message is now 40 characters long — longer than either of our messages are. And the part of the buffer that's left over is filled with zeros, as empty memory locations in a computer usually are. Then why the hearts? Well, in the ATASCII character code that your Atari uses, a zero does not equate to a space; instead, it equates to a heart-shaped graphics character. The ASCII code for a space is $20, or 32 in decimal notation. Just look at the ASCII character string in The Visitor and Response programs, and you'll see that the spaces in the message "TAKE ME TO YOUR LEADER!" are indeed represented by the value $20.

It is possible, of course, to print messages on your Atari's screen without strings of hearts appearing after them. What you have to do to keep the hearts from appearing is clear your text buffer — or, more accurately, stuff it with spaces — before a program runs.

Clearing a Text Buffer

Here's a short routine that will clear a text buffer — or any block of memory — and will stuff it with spaces, zeros, or any other value you choose. By incorporating this routine into our Visitor

and Response programs, you can replace the zeros in your onscreen messages with ASCII spaces, and can therefore make spaces appear as spaces, rather than as hearts, on your computer screen. As you continue to work with assembly language, you'll find that memory clearing routines such as this one can come in very handy in many different kinds of programs. Word processors, telecommunications programs, and many other kinds of software packages make extensive use of routines that can clear values from blocks of memory and replace them with other values.

PROGRAM TO CLEAR A BLOCK OF MEMORY

```
1300 FILL
1310  LDA #FILLCH
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

This program is quite straightforward. Using indirect addressing and an X register countdown, it will fill each memory address in a text buffer (TXTBUF) with a designated fill character (FILLCH). Then the program ends. This routine will work with any 8-bit fill character, and with any buffer length (BUFLEN) up to 255 characters. Later on in this book, you'll find some 16-bit routines that can stuff values into longer blocks of RAM. You can use this routine by incorporating it into both your Visitor program and your Response program. Let's append it to both of those programs now, starting with The Visitor. With your assembler in its editing mode, load The Visitor from a disk and add these lines to it:

```
55 BUFLEN = 40
60 FILLCH = $20

150  JSR FILL

1300 FILL
1310  LDA #FILLCH
```

```
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

When these changes have been made, your VISITOR.SRC program should look like this:

THE VISITOR

```
10 ;
20 ;THE VISITOR
30 ;
35 TXTBUF = $5041
40 OPNSCR = $5003
50 PRNTLN = $5031
55 BUFLEN = 40
60 FILLCH = $20
70 ;
80  *=$600
90 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,
     $20
110  .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120  .BYTE $4C,$45,$41,$44,$45,$52,$21
130 ;
140 VIZTOR
150  JSR FILL
160  LDX #0
170 LOOP LDA TEXT,X
180  STA TXTBUF,X
190  INX
200  CPX #23
210  BNE LOOP
220  JSR OPNSCR
230  JSR PRNTLN
240 INFIN JMP INFIN
1300 FILL
1310  LDA #FILLCH
```

```
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

When your program looks just like that, you can save it on a disk in its improved version. Then you can make exactly the same kinds of changes in your Response program, and resave that program, too. When you've finished with the Response program, it should look something like this:

RESPONSE

```
10 ;
20 ;RESPONSE
30 ;
40 TXTBUF = $5041
50 OPNSCR = $5003
60 PRNTLN = $5031
65 BUFLEN = 40
70 FILLCH = $20
75 ;
80 EOL = $9B
90 ;
100  *=$650
110 ;
120 TEXT .BYTE "I AM the leader, you fool!",EOL
130 ;
140 RSPONS
150 ;
160  LDX #0
170 LOOP
180  LDA TEXT,X
190  STA TXTBUF,X
200  CMP #$9B
210  BEQ FINI
220  INX
230  JMP LOOP
240  FINI
```

```
250  JSR OPNSCR
260  JSR PRNTLN
270 INFIN
280  JMP INFIN
1300 FILL
1310  LDA #FILLCH
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

Doing It

When you have both of these improved programs safely stored on a disk — in both their source code and object code versions — you can run them and see that all of our difficulties with our text buffers have now been resolved. And that brings us to our next chapter, in which you will learn how to call assembly language routines from BASIC programs.

Chapter Eight

Calling Assembly Language Programs from BASIC

Sometimes it's hard to decide whether to write a program in BASIC or in assembly language. But it isn't always necessary to make that choice. In many cases, you can combine the simplicity of BASIC with the speed and versatility of assembly language by simply writing assembly language routines that can be called from BASIC. In this chapter, that's what you'll be learning to do.

The first two programs we'll be working with are the ones that were introduced in Chapter 7: the programs called The Visitor and Response. Before we can call these two programs from BASIC, however, we'll have to make a couple of changes in the way the two programs are written. First we'll have to delete the infinite loop at the end of each program, and replace it with an RTS instruction, so that each program will return to BASIC when it's done instead of looping around endlessly. We'll also have to add a PLA instruction at the beginning of each program so that the programs won't mess up the stack when they're called from BASIC. Later on in this chapter, we'll explain exactly why these PLA instructions are needed. If you've just finished Chapter 7, and still have your computer on, you can load The Visitor and Response programs into your computer and make the necessary changes in them now.

Starting with The Visitor

Let's start with the program called The Visitor. With your assembler active and in its EDIT mode, load the source code of The Visitor into your computer and type LIST. This is what you should see:

THE VISITOR

```
10 ;
20 ;THE VISITOR
30 ;
35 TXTBUF=$5041
40 OPNSCR=$5003
50 PRNTLN=$5031
55 BUFLEN=40
60 FILLCH=$20
70 ;
80  *=$600
90 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,
    $20
110  .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120  .BYTE $4C,$45,$41,$44,$45,$52,$21
130 ;
140 VIZTOR
150  JSR FILL
160  LDX #0
170 LOOP LDA TEXT,X
180  STA TXTBUF,X
190  INX
200  CPX #23
210  BNE LOOP
220  JSR OPNSCR
230  JSR PRNTLN
240 INFIN JMP INFIN
1300 FILL
1310  LDA #FILLCH
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

Now you can amend the program so it can be called from BASIC. First, insert a line containing a PLA instruction by typing

```
145  PLA
```

Then change the infinite loop in line 240 to an RTS instruction by typing

```
240  RTS
```

When that's done, I suggest that you change the name of the program to VISITOR.SR2 to distinguish it from the original program. To do that, just type

```
20 ;VISITOR.SR2
```

and hit the carriage return.

All Done

When you've made all of those line changes in the VISITOR.SRC program, type LIST again and here's what you should see:

```
THE VISITOR

10 ;
20 ;VISITOR.SR2
30 ;
35 TXTBUF=$5041
40 OPNSCR=$5003
50 PRNTLN=$5031
55 BUFLEN=40
60 FILLCH=$20
70 ;
80  *=$600
90 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,
    $20
110  .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120  .BYTE $4C,$45,$41,$44,$45,$52,$21
130 ;
140 VIZTOR
145  PLA
```

```
150  JSR FILL
160  LDX #0
170 LOOP LDA TEXT,X
180  STA TXTBUF,X
190  INX
200  CPX #23
210  BNE LOOP
220  JSR OPNSCR
230  JSR PRNTLN
240  RTS
1300 FILL
1310  LDA #FILLCH
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

Saving your Amended Program

When you've made all of the necessary changes in your Visitor program, you can save it on a disk once again — in both its source code and object code versions — under the file names VISITOR.SR2 and VISITOR.OB2. Then you'll be ready to amend the other program introduced in Chapter 7 — Response — so that it, too, can be called from BASIC.

Amending Response

To fix up RESPONSE.SRC so that it is accessible from BASIC, just load its source code into your computer and make the same three kinds of line changes that you made in your Visitor program. When you've made the necessary changes, this is how Response should look:

RESPONSE

```
10 ;
20 ;RESPONSE.SR2
30 ;
40 TXTBUF=$5041
50 OPNSCR=$5003
60 PRNTLN=$5031
65 BUFLEN=40
70 FILLCH=$20
75 ;
80 EOL=$9B
90 ;
100  *=$650
110 ;
120 TEXT .BYTE "I AM the leader, you fool!",EOL
130 ;
140 RSPONS
145  PLA
150  JSR FILL
160  LDX #0
170 LOOP
180  LDA TEXT,X
190  STA TXTBUF,X
200  CMP #$9B
210  BEQ FINI
220  INX
230  JMP LOOP
240 FINI
250  JSR OPNSCR
260  JSR PRNTLN
270  RTS
1300 FILL
1310  LDA #FILLCH
1320  LDX #BUFLEN
1330 START
1340  DEX
1350  STA TXTBUF,X
1360  BNE START
1370  RTS
```

Doing It

When you've made the necessary changes in the Response program, you can save it in its new version under the file names RESPONSE.SR2 and RESPONSE.OB2. Then we'll be ready to call both The Visitor program and the Response program from BASIC. To do that, you'll need a BASIC cartridge for your computer if it's an Atari 400, 800 or 1200XL. (You won't need a BASIC cartridge if you have a late model Atari, since the newer Atari models have BASIC built-in.) Anyway, when you have your computer up and running again, in BASIC now, and with your data disk in your disk drive, the first thing you'll have to do is call up the Atari DOS menu. When your BASIC interpreter's READY prompt appears, type the command DOS. Then, when the DOS menu appears, select menu option L (BINARY LOAD) and load VISITOR.OB2, RESPONSE.OB2 and PRNTSC.OB2 into your computer's memory.

When you have all three programs loaded, type B to return control of your computer to your BASIC interpreter. Then, when your BASIC interpreter's READY prompt appears on your screen, you can type in this BASIC command:

```
X=USR(1559)
```

If you've typed and assembled your VISITOR.SR2 program just the way we did, and if its object code is now stored in your computer's memory, then it should run as soon as you type X=USR(1559) and hit a carriage return, since its starting address is $0617, or 1559 in decimal notation.

Similarly, you can run RESPONSE.OB2 by simply typing

```
X=USR(1643)
```

and hitting your carriage return.

Now that you've called two programs from BASIC, we're ready to talk about how you did it: specifically, how the Atari BASIC

USR function works, and how it's used in assembly language programming.

The USR Function

Machine language programs, as we have just observed, are called from BASIC with a special function called a USR statement. The USR function can be written in two ways: either with or without one or more optional arguments. A call that does not include arguments is written using the format we used to call our Visitor and Response programs:

```
X=USR(1643)
```

When a call is written using this format, the number in parentheses equates to the starting address (expressed as a decimal number) of the machine language program being called. When the machine language program ends, control of the computer will be returned to BASIC. If a program is running when the USR function is used, the program will resume at the first instruction following the USR fuction when control is returned to BASIC.

Works Like "GOSUB"

In this respect, a USR statement works just like an ordinary GOSUB instruction. Like a subroutine written in BASIC, a machine language program called from BASIC must end with a return instruction. In BASIC, the return instruction is, logically enough, RETURN. In assembly language, the return instruction is RTS, which stands for "ReTurn from Subroutine." A call, in which arguments are used, looks like this

```
X=USR(1536,ADR(A$),ADR(B$))
```

or like this:

```
X=USR(1536,X,Y)
```

When arguments are used in a USR function, each argument can be any value that equates to a 16-bit number. Machine language operations can be performed on the values referred to in the arguments, and the results of those operations can be passed back to BASIC when the machine language operations are completed. In this way, operations that take place at machine language speed can be used in BASIC programs. Whether arguments are used in a USR function or not, the machine language program called by the USR function can also *return* a 16-bit value to BASIC when it passes control back to a BASIC program.

Returning to BASIC

To return a value to BASIC when a machine language program has ended, all you have to do is store the value that is to be returned in two special 8-bit memory locations before control is returned to BASIC. Those two locations are $D4 and $D5 (212 and 213 in decimal notation). When a value to be returned to BASIC is stored in these two locations, it should be stored with the low byte in $D4 and the high byte in $D5. When control returns to BASIC, the 16-bit value in those two locations will automatically be converted into a decimal value ranging from 0 to 65535. Then that value will be returned to BASIC as the value of the variable in the USR statement, for example, the "X" in X=USR(1536,X,Y).

How the USR Function Works

When a BASIC program enounters a USR statement, the memory address of the current instruction in the BASIC program being run is placed on top of the hardware stack. Next an 8-bit number, the *number* of arguments that appear in the USR statement, is pushed onto the stack. If there are no arguments in the USR statement, then a zero is placed on top of the stack. When a USR statement calls a machine language subroutine, the machine language subroutine's return address is always covered up on

the stack by another number and therefore, even if that number is a zero, it must be cleared from the stack before control can be returned to BASIC.

Clearing the Stack

And that is why the first mnemonic in most machine language programs designed to be called from BASIC is a "clear the stack" instruction: PLA.

More Stack Operations

If arguments are included in a USR function, then more operations involving the stack take place when a machine language program is called. Before the *number* of arguments is placed on the stack, *the arguments themselves* are pushed onto the stack. Then, when the machine language program begins, the arguments can be removed from the stack and processed by the machine language program in whatever way the programmer wishes. Finally, when the machine language program ends and control of the computer is passed back to BASIC, the results of any machine language operations that may have been carried out using the arguments in the USR statement can be stored in memory addresses $D4 and $D5. The values that have been stored in this pair of locations can then be returned to BASIC as the value of the variable in the original USR statement.

A Sample Program

That may sound like a mouthful, but another sample program that you can type into your computer and call from BASIC right now should clarify what we're getting at. Type the following program in your computer.

A 16-BIT ADDITION PROGRAM

```
10 ;
20 NUM1=$CB
30 NUM2=$CE
```

```
40 SUM=$D4
50 ;
60   *=$0600
70 ;
90   CLD
100    PLA ;CLEAR # OF ARGS. FROM ACC.
105    PLA
110    STA NUM1+1 ;HIGH BYTE OF 1ST ARG.
120    PLA
130    STA NUM1 ;LOW BYTE OF 1ST ARG.
140    PLA
150    STA NUM2 +1 ;HIGH BYTE OF 2ND ARG.
160    PLA
170    STA NUM2;LOW BYTE OF 2ND ARG.
180 ;
190    CLC
210    LDA NUM1 ;LOW BYTE OF NUM1
220    ADC NUM2 ;LOW BYTE OF NUM2
230    STA SUM ;LOW BYTE OF SUM
240    LDA NUM1+1 ;HIGH BYTE OF NUM1
250    ADC NUM2+1 ;HIGH BYTE OF NUM2
260    STA SUM+1 ;HIGH BYTE OF SUM
270    RTS
```

When you've finished typing the program, you can assemble it and save it in both its source code and object code versions. The suggested file names for the listings are "ADD16B.SRC" and "ADD16B.OBJ".

An Adding Machine Program

When you have your programs saved, you can remove your assembler from your computer, and type in this BASIC program:

THE WORLD'S MOST EXPENSIVE ADDING MACHINE

```
10 GRAPHICS 0:PRINT:PRINT "WORLD'S MOST
   EXPENSIVE ADDING MACHINE"
20 PRINT:PRINT "X=";
30 INPUT X
40 PRINT "Y=";
```

```
50 INPUT Y
60 SUM=USR(1536,X,Y)
70 PRINT "X+Y=";SUM
80 PRINT
90 GOTO 20
```

When you have finished typing in the program, you can save it under the file name "ADD16B.BAS". That will complete our preparations for calling our newest addition program from BASIC. You can now call your ADD16B.OBJ from BASIC the same way you called your Response program. Just call up your DOS menu and load the binary file ADD16B.OBJ. Type "B" to get back into your BASIC editing mode, load your ADD16B.BAS file, and type "RUN". You can then start using "The World's Most Expensive Adding Machine."

As you can see by running a few numbers through your new adding machine, it can perform 16-bit addition. Although we haven't covered 16-bit addition in this book yet, you can probably figure out how the program works with very little trouble.

First, in lines 20 through 40, the program reserves memory space for three 16-bit numbers, two numbers that are added, and their sum. The two numbers to be added are labeled NUM1 and NUM2, and the address where their sum will be store is labeled SUM. In line 100, the program clears the hardware stack with a PLA instruction. That gets rid of the 8-bit "number of arguments" value that our BASIC program has placed on top of the stack. Then, in lines 105 through 170, the program removes two 16-bit arguments from the stack — the 16-bit numbers that have been included in the USR statement in our BASIC program. These are the two numbers to be added, and our machine language program now places them in the two pairs of 8-bit memory locations that have been labeled NUM1 and NUM2.

The Heart of the Program

Now we've come to the main part of our assembly language program. In line 200, the program clears the carry flag, as every good addition program should. Then the actual addition begins.

The program adds the low bytes of NUM1 and NUM2, and stores the result of this calculation into the low byte of SUM. It then adds the high bytes of NUM1 and NUM2 (along with a carry, if there is one) and stores the result of *this* calculation into the *high* byte of SUM.

That's all there is to it. As it turns out, and this is no accident, the memory addresses that have been reserved for the the value SUM are $D4 and $D5, the two special memory locations that are always returned to BASIC as the value of the variable in a USR command. That's quite a mouthful, but it really isn't hard to understand. What it means is that our machine language program has now solved the equation it was presented with when we handed it the USR instruction in our BASIC program. That USR statement, as you may remember, looked like this:

```
SUM=USR(1536,X,Y)
```

the values have now been assigned to all of its variables.

The "X" and "Y" in the equations were assigned values when you typed them in during the BASIC part of the program. Then, when control of your computer was transferred to machine language, the values of X and Y were converted to 8-bit numbers and pushed onto the hardware stack. In the machine language program which you wrote, the values of X and Y were pulled from the stack. They were then stored in two pairs of 8-bit memory locations (labeled NUM1 and NUM2, to avoid any confusion with the 6502's X and Y registers). Next, the values of X and Y (now called NUM1 and NUM2) were added. Their sum was stored in $D4 and $D5. When control was returned to BASIC, your BASIC interpreter converted the values in $D4 and $D5 into a 16-bit value and assigned that value to the the BASIC variable SUM, the variable used in the USR statement that called the machine language program. Then, as instructed in the BASIC program you wrote, your BASIC interpreter printed the value stored in the BASIC variable sum on your video screen.

This was quite a complex series of operations, as most sequences of machine language operations are. Unfortunately, we still haven't written a very useful assembly language addition pro-

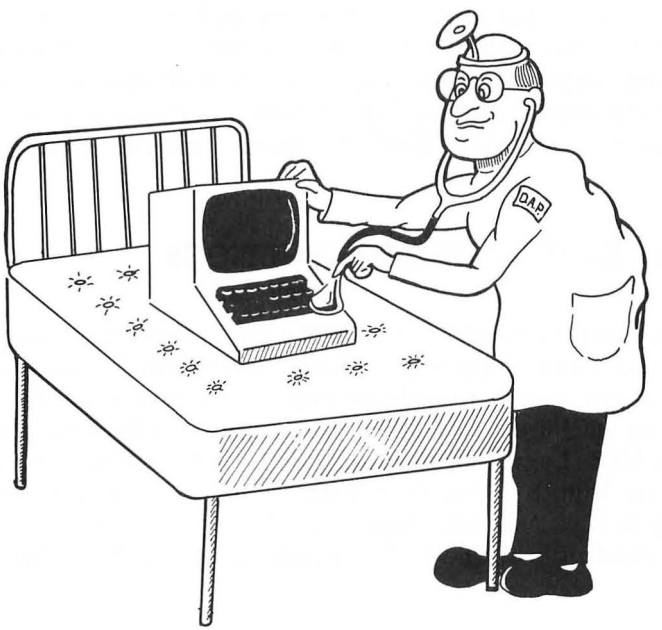
gram. While the adding machine we have just created may be a very expensive, it isn't a very useful one in real world applications. It *can* add 16-bit numbers and print out 16-bit results. That's a definite improvement over the 8-bit addition program we created a few chapters back, but it still has some serious deficiencies. It can't handle numbers or results that are longer than 16 bits. It can't work with floating point decimal numbers, or with signed numbers. If you type in a number that's too big for it to handle, it won't let you know, it will simply "roll over" past zero and add numbers without any carrying, and give incorrect results.

Obviously, we have not yet managed to write an addition program that will work as well as a good addition program should. We haven't even looked at any subtraction, multiplication or division programs. Very shortly, though, we shall. We'll also be discussing many other topics including signed numbers, BCD numbers, bit manipulations and more in Chapter 10, Assembly Language Math. Before we get to Chapter 10, however, there's another bit of ground to cover in Programming Bit by Bit, the topic of Chapter 9.

Chapter Nine

Programming Bit by Bit

In the world of computer programming, being able to perform operations on single binary bits is somewhat akin to being able to perform microsurgery. If you can test bits, shift bits and generally manipulate bits skillfully, you're a real D.A.P. (Doctor of Assembly Language Programming). Nonetheless, bit manipulation, like most facets of assembly language programming, is not nearly as difficult as it appears at first glance. An understanding of a few basic principles will remove much of the mystery from bit-shifting, bit-testing, and other single bit operations in assembly language. We've already touched on many of the concepts you'll need to know to become an expert bit surgeon.



For example, consider what you've already learned about using the carry bit of the 6502 processor status register. Using the carry bit is one of the most important bit manipulation techniques in 6502 assembly language. You've already had some experience in using the carry bit in addition programs. In this chapter, you'll have an opportunity to teach your computer how to perform some new tricks using the carry bit of its 6502 processor status (P) register.

Using the Carry Bit in Bit-Shifting Operations

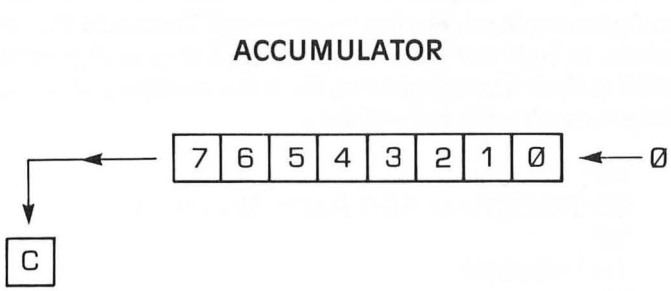
As we have pointed out a number of times now, the 6502 microprocessor in your Atari computer is an 8-bit chip; it cannot perform operations on numbers larger than 255 without putting them through some fairly tortuous contortions. In order to process numbers that are larger than 255, the 6502 must split them up into 8-bit chunks, and then perform the requested operations on each piece of a number. Then each number that has been ripped apart must be put back together and made whole again. Once you're familiar with how this is done, it isn't nearly as difficult as it sounds. In fact the electronic scissors that are used in all of this electronic cutting and pasting are actually contained in one tiny bit, the carry bit in the 6502's processor status (P) register.

Four Bit-Shifting Instructions

You've seen how carry operations work in several programs in this book. But in order to get a clearer look at how the carry works in 6502 arithmetic, it would be useful to examine four very specialized machine language instructions: ASL (Arithmetic Shift Left), LSR (Logical Shift Right), ROL (ROtate Left), and ROR (ROtate Right). These four instructions are used very extensively in 6502 assembly language. We'll look at them one at a time, starting with the ASL (Arithmetic Shift Left) instruction.

ASL (Arithmetic Shift Left)

As we pointed out back in our chapter on binary arithmetic, every round number in binary notation is equal to the square of the preceding round binary number. In other words, 1000 0000 ($80) is double the number 0100 0000 ($40), which is double the number 0010 0000 ($20), which is double the number 0001 0000 ($10), and so on. It is therefore extremely easy to multiply a binary number by 2. All you have to do is shift every bit in the number left one space, and place a zero in the bit that has been emptied by this shift, bit 0, or the rightmost bit of the number. If the leftmost bit, bit 7, of the number to be doubled is a 1, then provision must be made for a carry. The entire operation we have just described, shifting a byte left with a carry, can be performed by a single instruction in 6502 assembly language. That instruction is ASL, which stands for "Arithmetic Shift Left." Here's an illustration of how the ASL instruction works:



As you can see from this illustration, the instruction ASL moves each bit in an 8-bit number one space to the left, each bit except Bit 7. Bit 7 drops into the carry bit of the processor status (P) register. The ASL instruction is used for many purposes in 6502 assembly language. You could use it as an easy way of doubling a number.

```
10 ;
20  *=$0600
30 ;
40  LDA #$40 ;REM 0100 0000
```

```
50  ASL A ;SHIFT VALUE IN ACCUMULATOR
    TO LEFT
60  STA $CB
70  .END
```

If you run this program, and then use your debugger's "D" (display) command to examine the contents of memory address $CB, you'll see that the number $40 (0100 0000) has been doubled to $80 (1000 0000) before being stored in memory address $CB.

Packing Data Using ASL

Another use for the ASL instruction is to "pack" data, and thus to increase a computer's effective memory capacity. To get an idea of how data packing works, suppose you had a series of 4-byte values stored in a block of memory in your computer. These values could be ASCII characters, BCD numbers, (more about those later) or any other kind of 4-bit values. Using the ASL instruction, you could pack two such values into every byte of the block of memory in which they were stored. You could thus store the values in half the memory space that they had previously occupied in their unpacked form. Here is a routine you could use in a loop to pack each byte of data:

```
10 ;
20 ;PROGRAM FOR PACKING DATA
30 ;
40 *=$0600
50 ;
60 NYB1=$C0
70 NYB2=$C1
80 PKDBYT=$C2
90 ;
100   LDA #$04
110   STA NYB1
120   LDA #$06
130   STA NYB2
140 ;
150   CLC
160   LDA NYB1
170   ASL A
```

```
180   ASL A
190   ASL A
200   ASL A
210   ADC NYB2
220   STA PKDBYT
230   .END
```

How the Routine Works

This routine will load a 4-bit value into the accumulator, shift that value to the high nybble in the accumulator, and then use the instruction ADC to place another 4-bit value in the low nybble of the accumulator. The accumulator is thus "packed" with two 4-bit values, and those two values are then stored in a single 8-byte memory register.

Testing the Results

Type the program into your computer, and you can then use MAC/65 or Atari debugger's G (GOTO) command to run it. Then, if it executes correctly, you can use your debugger's "D" (display) command to see exactly what has been done. With your assembler in its DEBUG mode, type "DC0" and you should see this line:

```
00C0  04 06 46 00 00 00 00 00
```

As you can see from this line, the program has stored the number $04 in memory address $C0, and $06 in memory address $C1. Both of these values have been packed into memory address $C1. It doesn't take much imagination to see how this technique can double your computer's capacity to store 4-bit numbers in 8-bit memory locations.

Unpacking Data

It wouldn't do any good to pack data if it couldn't later be *unpacked*. It so happens that data packed using ASL can be unpacked using a complementary instruction, LSR (Logical Shift Right). We'll discuss the LSR instruction later on in this chapter.

Loading a Color Register Using ASL

In Atari assembly language, the ASL command can also be used to control the colors on the screen. Here's how that's done. In an Atari computer the colors you can use in screen graphics are stored in five color registers. Tables listing the colors and luminance values that can be stored in these registers are printed in Part 9 of the *Atari BASIC Reference Manual.* The upper nybble of each Atari color register holds a hue value, which is the same number as the second parameter used in the SETCOLOR command in Atari BASIC. Bits 1, 2 and 3 in each color register hold the luminance value of the color, the same number as the third parameter in the BASIC SETCOLOR command. It doesn't matter what bit 0 is in a color register, since that bit is not used. By using the instruction ASL, you can easily control the onscreen colors in an Atari assembly language program.

How it's Done

Color Register 2 holds the background color in Graphics 0, the standard Atari text mode. Suppose you wanted to load this register with its standard color, which is light blue. In your Atari, the memory address of Color Register 2 is $2C6. The Atari code number for blue is 9, and the code number for the luminance of the light blue used in the Graphics 0 screen display is 4. The ASL command could therefore be used to store light blue in Color Register 2 in the following manner:

```
10 ;
20 ;SETCLR PROGRAM
30 ;
40   *=$0600
50 ;
60   CLC
70   CLD
80   LDA #$09 ;REM LIGHT BLUE
90   ASL A
100   ASL A
110   ASL A
120   ASL A
```

```
130   STA $2C6 ;REM COLOR REGISTER 2
140   LDA #$04 ;HUE NO. 4
150   ASL A
160   ADC $2C6
170   STA $2C6
180   .END
```

As you can see, this program loads Color Register 2 (address $2C6) with Color #$09, Luminance #$04, the shade of light blue that Atari uses for the background of its standard Graphics 0 screen. If you assemble the program and run it, these are the values that will wind up in each bit of Color Register 2 (memory address $2C6) in your computer.

COLOR REGISTER 2
(memory address $2C6)

Bit No.	7	6	5	4	3	2	1	0
Contents	1	0	0	1	1	0	0	*
	Color ($09)				Lum. ($04)			

* Bit 0 is not used.

Testing the Program

Type the program and execute it, you can then use the "D" command of your MAC/65 or Atari debugger to see if it worked. When you type "D2C6" take a look at the contents of Color Register 2, your display line should tell you that memory address $2C6 (Color Register 2) contains the value $98. Convert the hex number $98 to a binary number, and you'll see that it equals 10011000, the exact binary number illustrated above in our bit-by-bit breakdown of Color Register 2. This same technique could also be used, to load any other register with any other color and luminance in an assembly language program. All you'd have to do is substitute a few variables for literal numbers. Here's one way the program could be rewritten to make it more versatile:

A BETTER PROGRAM FOR SETTING COLORS

```
10 ;
20 ;SETCLR PROGRAM
30 ;
40 CLRNR=$C0 ;COLOR NUMBER
50 HUENR=$C1 ;HUE NUMBER
60 CLREG=$2C6 ;COLOR REGISTER NR.
70 ;
80   *=$0600
90 ;
100   LDA #$09 ;LIGHT BLUE
110   STA CLRNR
120   LDA #$04 ;HUE #4
130   STA HUENR
140 ;
150   CLC
160   CLD
170   LDA CLRNR
180   ASL A
190   ASL A
200   ASL A
210   ASL A
220   STA CLREG
230   LDA HUENR
240   ASL A
250   ADC CLREG
260   STA CLREG
270   .END
```

This is actually two programs in one. In lines 100 to 130, you can stuff values you want to use into variables that represent a color, a luminance, and a color register. Then the main body of the program, lines 150 through 260, can be used to load any color and luminance values into any color register. So why not try it? Change the variables we used in lines 40 through 60, run the program a few times, and watch the colors on your screen change!

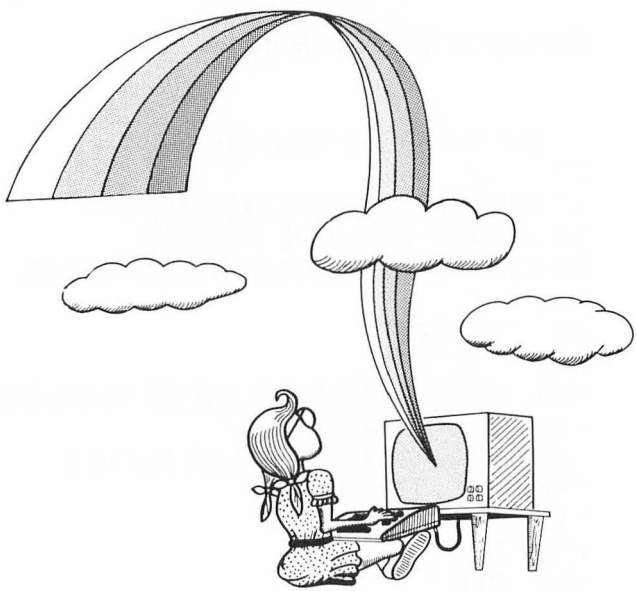
An Easier Way

You can also change the screen colors generated by your Atari without going to the trouble of using a lot of ASL commands. If you wish, you can perform all of the necessary ASL operations in your head, before writing the program. For example, if you multiply one of Atari's color numbers by $10 (or 16 in decimal notation), you'll get the same result that you would if you performed four ASL operations on the number. Multiply $09 by $10, and you'll get $90, the same number you'd get by performing four ASL operations on the number $09. Similarly, you can perform one ASL operation on a binary number by simply multiplying it by 2 (or, if you prefer, by $02). Perform one ASL operation on the number $04 (binary 0100), and you get $08 (binary 1000); the same number you'd get if you multiplied $04 by 2. If you wanted to write an easier SETCLR program, you could do it this way:

AN EASIER SETCLR PROGRAM

```
10 ;
20 ;AN EASIER SETCLR PROGRAM
30 ;
40 CLRNR=$C0 ;COLOR NUMBER
50 HUENR=$C1 ;HUE NUMBER
60 CLREG=$2C6 ;COLOR REGISTER NO.
70 ;
80   *=$0600
90 ;
100   LDA #$90 ;COLOR NO. 09 TIMES $10
110   STA CLRNR
120   LDA #$08 ;HUE NO. 04 TIMES 2
130   STA HUENR
140 ;
150   CLC
160   CLD
170   LDA CLRNR
220   STA CLREG
230   LDA HUENR
250   ADC CLREG
260   STA CLREG
270   .END
```

By adding a couple of loops to a program like this, plus an infinite loop at the end, it would be possible to stuff a color register with a constantly changing rainbow of colors. You could then make the Atari computer cycle over and over again through all of its screen colors, not stopping until someone hit the BREAK or SYSTEM RESET key, or turned off the machine or pulled the plug. Here's a program that will do just that. It will loop endlessly through all of the colors and hue combinations that your Atari can generate, displaying each of them in turn on the border area around your computer screen. (If the program looks familiar, that's because it is. As promised, it's an assembly language version of the BASIC color rotation program that was presented back in Chapter 2.)

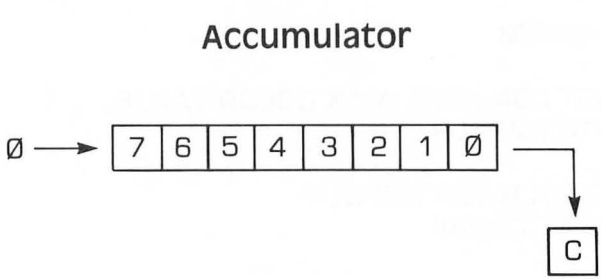


```
10 ;
20 ;RAINBOW.SRC
30 ;
40 COLRBK=$2C8 ;THE GRAPHICS 0 BORDER COLOR
   REGISTER
50 TMPCLR=$C0 ;A PLACE TO STORE COLORS
   TEMPORARILY
60 ;
70   *=$0600
80 ;
90 START LDA #$FE ;MAX COLOR VALUE
100   STA TMPCLR
110 ;
120 NEWCLR LDA TMPCLR
130   STA COLRBK
140 ;
150   LDX #$FF
160 LOOPA ;JUST A DELAY LOOP
170 ;
180   LDY #$30
190 LOOPB ;ANOTHER DELAY LOOP
200   DEY
210   BNE LOOPB
220 ;
230   DEX
240   BNE LOOPA
250 ;
260   DEC TMPCLR ;DECREMENT TMPCLR
270   DEC TMPCLR ;SUBTRACT 2 FOR NEXT COLOR
280   BNE NEWCLR ;IF NOT ZERO, CHANGE COLORS
      AGAIN
290 ;
300   JMP START ;ALL COLORS DISPLAYED — NOW
      DO 'EM ALL AGAIN
```

LSR (Logical Shift Right)

The instruction LSR (Logical Shift Right) is the exact opposite of the instruction ASL, as you can see from this illustration:

An Illustration of the "LSR" Mnemonic



How the LSR Instruction Works

LSR, like ASL, works on whatever binary number is in the 6502's accumulator. But it will shift each bit in the number one position to the *right*. Bit 7 of the new number, left empty by the LSR instruction, will be filled in with a zero. The LSB (Least Significant Bit) will be dumped into the carry flag of the P register. The LSR instruction can be used to divide any even 8-bit number by 2, as follows:

DIVIDING A NUMBER BY 2 WITH THE "LSR" INSTRUCTION

```
10 ;
20 ;DIVIDING BY 2 USING LSR
30 ;
40 VALUE1=$C0
50 VALUE2=$C1
60 ;
70   *=$0600
```

```
80 ;
90  LDA #6
100  STA VALUE1
110 ;
120  LDA VALUE1
130  LSR A
140  STA VALUE2
150  .END
```

This routine can also be used for another purpose. If you run it, and then check the carry flag, you can tell whether the number in VALUE1 is odd or even. If the routine leaves the carry bit clear, the number that was just divided is odd. If the carry bit is set, the value is even!

Next is a program you can type, execute, and check using your debugger to see whether a number is even or odd. If the program leaves the number $FF in memory address $C2, labeled FLGADR, then the number divided by 2 in line 160 is odd. If the program leaves a 0 in FLGADR, then the number that was divided is even:

```
10 ;
20 ;ODD OR EVEN?
30 ;
40 VALUE1=$C0
50 VALUE2=$C1
60 FLGADR=$C2
70 ;
80  *=$0600
90 ;
100  LDA #7 ;(ODD)
110  STA VALUE1
120  LDA #0
130  STA FLGADR ;CLEARING FLGADR
140 ;
150  LDA VALUE1
160  LSR A ;PERFORM THE DIVISION
170  STA VALUE2 ;DONE
180 ;
190  BCS FLAG
```

```
200   RTS ;END ROUTINE IF CARRY CLEAR...
210 ;
220 FLAG
230   LDA #$FF ;OTHERWISE, SET FLAG...
240   STA FLGADR
250   RTS ;... AND END THE PROGRAM
```

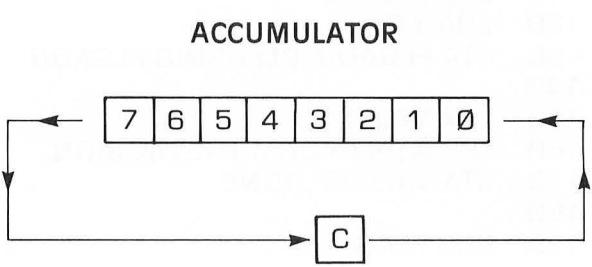
Unpacking Data

As we've mentioned, you can also use LSR to unpack data that has been packed using ASL. But to unpack data, you also have to use another type of assembly language function, called a logical operator. We'll discuss logical operators and present a sample routine for unpacking data later in this chapter. Meanwhile, let's take a look at two more bit-shifting operators: ROL (which stands for "ROtate Left") and ROR (which means "ROtate Right").

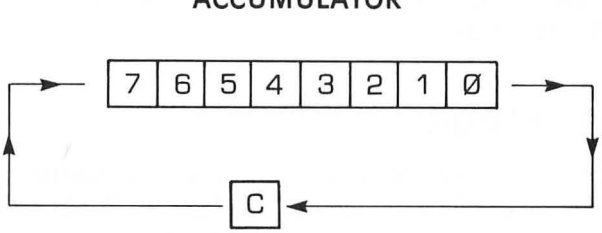
ROL (Rotate Left) and ROR (Rotate Right)

The instructions ROL (rotate left) and ROR (rotate right) are also used to shift bits in binary numbers. But they don't make use of the carry bit. Instead, they work this way:

The ROL ("Rotate Left") Instruction



The ROR ("Rotate Right") Instruction



How "ROL" and "ROR" Work

As you can see, ROL and ROR work much like ASL and LSR, except that the carry bit is shifted into the end bit left empty by the rotation instead of a zero. ROL, like ASL, shifts the contents of a byte one place to the left. But ROL does not place a zero into bit 0. Instead, it moves the carry bit into bit 0 of the number being shifted, which has been left empty by the shift rotation, and places bit 7 into the carry bit. ROR works just like ROL, but in the opposite direction. It moves each bit of a byte right one position, placing the carry bit into bit 7 and bit 0 into the carry bit.

The Logical Operators

Before we move on to conventional binary arithmetic, let's take a brief glance at four important assembly language mnemonics called *logical operators*. These instructions are AND ("and"), ORA ("or"), EOR ("exclusive or"), and BIT ("bit"). The four 6502 logical operators look very mysterious at first glance. But, in typical assembly language fashion, they lose much of their mystery (and most of their scare value) once you understand how they work.

AND, ORA, EOR and BIT are all used to compare values. But they work differently from the comparison operators CMP, CPX and CPY. The instructions CMP, CPX and CPY all yield very

general results. All they can determine is whether two values are equal and, if the values aren't equal, which one is larger than the other. AND, ORA, EOR and BIT are much more specific instructions. They're used to compare *single bits* of numbers, and hence have all sorts of uses.

Boolean Logic

The four logical operators in assembly language use principles of a mathematical science called *Boolean logic*. In Boolean logic, the binary numbers 0 and 1 are used not to express values, but to indicate whether a statement is true or false. If a statement is proved true, its value in Boolean logic is said to be 1. If it is false, its value is said to be 0. In 6502 assembly language, the operator AND has the same meaning that the word "and" has in English. If one bit AND another bit have a value of 1, and are thus "true," then the AND operator also yields a value of 1. But if any other condition exists, if one bit is true and the other is false, or if both bits are false, then the AND operator returns a result of 0, or false.

The results of logical operators are often illustrated with diagrams called *truth tables*. Here's a truth table for the AND operator:

Truth Table for "AND"

```
  0        0        1        1
AND 0    AND 1    AND 0    AND 1
-----    -----    -----    -----
  0        0        0        1
```

In 6502 assembly language, the AND instruction is often used in an operation called *bit masking*. The purpose of bit masking is to clear or set specific bits of a number. The AND operator can be used, for example, to clear any number of bits by placing a zero in each bit that is to be cleared. This is how that kind of bit masking operation could work:

```
100   LDA $AA ;BINARY 1010 1010
110   AND $F0 ;BINARY 1111 0000
```

If your computer encountered this routine in a program, the following AND operation would take place:

```
    1010 1010 (contents of accumulator)
AND 1111 0000
-------------
    1010 0000 (new value in accumulator)
```

As you can see, this operation would clear the low nybble of $AA to $0 (with a result of $A0). The same technique would work with any other 8-bit number. No matter what the number being passed through the mask 1111 0000 might be, its lower nybble would always be cleared to $0, and its upper nybble would always emerge from the AND operation unchanged.

Unpacking Data Using the "AND" Operator

The AND operator, together with the bit-shifting instruction LSR, can be used to unpack data that was packed using the instruction ASL. Here is a sample routine for unpacking data.

```
10 ;
20 ;UNPACKING DATA
30 ;
40 PKDBYT=$C0
50 LONYB=$C1
60 HINYB=$C2
70 ;
80   *=$0600
90 ;
100   LDA #$45 ;OR ANYTHING ELSE
110   STA PKDBYT
120   LDA #0 ;CLEAR LONYB AND HINYB
130   STA LONYB
140   STA HINYB
150 ;
160   LDA PKDBYT
165   PHA ;SAVE IT   ON THE STACK
170   AND #$0F ;BINARY 0000 1111
180   STA LONYB
```

```
190  PLA ;PULL PKDBYT OFF THE STACK
200  LSR A
210  LSR A
220  LSR A
230  LSR A
240  STA HINYB
250  RTS
```

The "ORA" Operator

When the instruction ORA ("or") is used to compare a pair of bits, the result of the comparison is 1 (true) if the value of *either* bit is 1. Here is the truth table for ORA:

Truth Table for "ORA"

0	0	1	1
ORA 0	ORA 1	ORA 0	ORA 1
0	1	1	1

ORA is also used in bit masking operations. Here is an example of a masking routine using ORA:

```
LDA #VALUE
ORA $0F
STA DEST
```

Suppose that the number in VALUE were $22 (binary 0010 0010). The following is the masking operation that would then take place.

```
        0010 0010 (in accumulator)
ORA     0000 1111
-------------------
        0010 1111  (new value in accumulator)
```

The "EOR" Operator

The instruction EOR ("exclusive or") will return a true value (1) if one, and *only* one, of the bits in the pair being tested is a 1. The following turth table is for the EOR operator.

Truth Table for "EOR"

Ø	Ø	1	1
EOR Ø	EOR 1	EOR Ø	EOR 1
Ø	1	1	Ø

The EOR instruction is often used for comparing bytes to determine if they are identical, since if any bit in two bytes is different, the result of a comparison will be non-zero. Here is an illustration of that comparison.

Example 1		Example 2	
	1011 0110		1011 0110
EOR	1011 0110	But: EOR	1011 0111
	0000 0000		0000 0001

In Example 1, the bytes being compared are identical, so the result of the comparison is zero. In Example 2, one bit is different, so the result of the comparison is non-zero. The EOR operator is also used to *complement* values. If an 8-bit value is used with $FF, every bit in it that's a 1 will be complemented to a 0, and every bit that's a 0 will be complemented to a 1.

```
      1110 0101 (in accumulator)
EOR   1111 1111
-------------------
      0001 1010 (new value in accumulator)
```

Still another useful characteristic of the EOR instruction is that when it is performed twice on a number using the same operand, the number will be first be changed to another number, and then restored to its original value. This is shown in the following example.

```
      1110 0101 (in accumulator)
EOR   0101 0011
-------------------
      1011 0110 (new value in accumulator)
EOR   0101 0011 (same operand as above)
-------------------
      1110 0101 (original value in accumulator restored)
```

This capability of the EOR instruction is often used in high resolution graphics to put one image over another without destroying the one underneath. (Yes, that's how its done!)

The "BIT" Operator

That brings us to the BIT operator, an instruction even more esoteric than AND, ORA, or EOR. The BIT instruction is used to determine the state of a specific bit — or specific bits — of a binary value stored in memory. When the BIT instruction is used in a program, bits 6 and 7 of the value being tested are transferred directly to bits 6 and 7 (the sign and overflow bits) of the processor status register. Then an AND operation is performed with the accumulator and the value in memory. The result of this AND operation is stored in the Z (zero) flag of the P register. If there is a 1 in both the accumulator and the value in memory at the same bit position, the result is non-zero and the Z flag is cleared. If the bits are different or both zero, the result is zero and the Z flag is set. The most important aspect here is that after all of this takes place, the values in the accumulator and the memory location remain unchanged.

Chapter Ten

Assembly Language Math

As an Atari assembly language programmer, you probably won't ever have to write many, if any, ultrasophisticated, multiprecision arithmetical programs. If you ever have to write a program that includes a lot of multiprecision math, your Atari can help you. It has a pretty powerful set of arithmetical programs, called *Floating Point*, or *FP* routines, built right into its operating system. The folks at Atari have taken care to provide you with the means of using these OS routines in your own assembly language programs. They've provided instructions on how to use the Atari FP package in a number of publications, including *De Re Atari*, a manual published by Atari for assembly language programmers.

Even if you don't want to use the FP package built into your Atari (and there are reasons not to; the routines are slow), you can find prewritten code for most kinds of sophisticated arithmetical operations, often called *multiple precision* binary operations, in a number of manuals on 6502 assembly language programming. One text that's packed with multiple precision programs that you can simply type into your computer and use is *6502 Assembly Language Subroutines*, written by Lance A. Leventhal and Winthrop Saville and published by Osborne/McGraw Hill.

Then Why Bother?

You may ask why we are bothering to include a chapter on advanced 6502 arithmetic in this volume. The answer: no matter how much help is available, you still have to know the principles of advanced 6502 arithmetic if you want to become a good assembly language programmer. So even though you may never have to write an assembly language routine that will perform long division on signed numbers, accurate to 17 decimal places, chances are pretty good that you'll eventually have to use *some* arithmetic operations in *some* programs.

Most assembly language programmers occasionally have to write an addition or subtraction routine, or a routine that will multiply or divide a pair of numbers, or a program that will deal with signed or BCD (Binary Coded Decimal) numbers. Logical operations, which are extensively used in 6502 programs, also fall under the heading of assembly language math. In this chapter, therefore, we'll be reviewing 8-bit and 16-bit binary addition, subtraction and multiplication, and also saying a few words about binary long division. We'll wind up the chapter with brief introductions to signed numbers and the BCD (Binary Coded Decimal) number system.

In the addition problem that you called from BASIC in Chapter 8, you saw how the carry bit of the processor status works in 16-bit addition operations. Now we're going to review the use of the carry bit in addition problems, and we're also going to take a look at how carries work in subtraction, muliplication and division problems.

A Close Look at the Carry Bit

The best way to get a close look at how the carry bit works is to look at it through an "electronic microscope" at the bit level. Look at these two simple 4-bit hexadecimal and binary addition problems in their binary and hexadecimal forms, and you'll see clearly how neither addition operation generates a carry in either binary or hexadecimal notation.

HEXADECIMAL	BINARY
04	0100
+01	+0001
05	0101
08	1000
+03	+0011
0B	1011

Now let's look at a couple of problems that use larger (8-bit) numbers. The first of these two problems doesn't generate a carry, but the second one does.

HEXADECIMAL		BINARY
8E		1000 1110
+23		+ 0010 0011
B1		1011 0001
8D		1000 1101
+FF		+ 1111 1111
18C	(1)	1000 1100

Note that the sum in the second problem is a 9-bit number — 1 1000 1100 in binary, or 18C in hexadecimal notation. Here's an assembly language program that will perform that very same addition problem. Type it into your computer and run it, and you'll be able to see how the carry flag in your computer works:

8-BIT Addition With a Carry

```
10  *=$0600
20  CLD
30  CLC
40  LDA  #$8D
50  ADC  #$FF
60  STA  $CB
70  RTS
```

When you've typed this program, assemble it and then run it by activating your assembler's debugger and using the "G" command. When the program has been executed, and while your debugger is still turned on, type the command "DCB" (for "Display memory location $CB"). You should then see this kind of line displayed on your video screen:

```
00CB 8C 00 00 00 00
```

That line shows us that memory address $CB now holds the number $8C, the correct sum of the numbers we added, except for the carry. So where's the carry? Well, if what you've read in this book about the carry bit is true, it must be in the carry bit of your computer's P register. As our program is written now, there's no easy way to find out whether the carry bit from our addition operation has been dumped into the carry bit of the P register. But by adding a couple of lines to the program, and running it again, we can find out. Here's how to rewrite the program so we can check the carry bit:

```
10  *=$0600
20  CLD
30  CLC
40  LDA #$8D
50  ADC #$FF
60  PHP
70  STA $CB
80  PLA
90  AND #01
100   STA $CC
110   RTS
```

In this rewrite of our original program, we've used one new stack manipulation instruction: PHP.PHP means "PusH Processor status (P register) or stack." We've also used the AND operator introduced in Chapter 9. In addition, we've used one stack manipulation instruction that was introduced a couple of chapters ago: PLA, which means "PulL Accumulator from stack."

The instruction PHP is used in line 60 of our rewritten program. It appears there because we want to save the contents of the P register as soon as the numbers #$8D and #$FF have been added. We can use the instruction PHP without any fear that it will do anything terrible to our program, since it is an instruction that doesn't affect the contents of either the P register or the accumulator, but it does affect the stack pointer.

When you run this program, the first thing it will do is add the literal numbers $8D and $FF. Before it stores the result of this

calculation anywhere, however, it pushes the contents of the status register onto the stack, using the instruction PHP. When that operation is complete, the value in the accumulator (still the sum of $8D and $FF, with no carry), is stored in memory address $CB. Next, in line 80, when almost everything else in the program has been done, the value that was pushed onto the stack by the PHP instruction back in line 60 is removed from the stack. Then, since the only flag in the P register that we're interested in is the carry flag (Bit 0), we have used an AND operation to mask out every bit of the number just pulled from the stack except Bit 0. Finally, the resulting number — which should be $01 if our operations up to now have worked — is stored in memory address $CC.

Now, at any time we like, we can peek into the memory address and see what the result of the calculation in the program was (without a carry). Then we can peer into memory address $CC and take a look at just what the status of the P register was just after we added the numbers $8D and $FF. So let's do it! Assemble the program, execute it using your debugger's "G" comand, and then use the command "DCB" to take a look at the contents of memory address $CB and the memory locations that follow. Here's what you should see:

```
ØØCB 8C Ø1 ØØ ØØ ØØ
```

That line tells us two things: that memory address $CB does hold the number $8C, the result of our calculation, without a carry and that our addition of #$8D and #$FF did indeed set the carry bit of the processor's status register.

16-Bit Addition

We will now take a look at a program that will add two 16-bit numbers. The same principles used in this program can also be used to write programs that will add numbers having 24 bits, 32 bits, and more. Here's the program:

```
A MULTIPLE PRECISION ADDITION PROGRAM

1Ø ;
2Ø ;THIS PROGRAM ADDS A 16-BIT NUMBER
    IN $BØ AND $B1
```

```
30 ;TO A 16-BIT NUMBER IN $C0 AND $C1
40 ;AND DEPOSITS THE RESULTS IN $C2
   AND $C3
50 ;
60   *=$0600
65 ;
70   CLD
80   CLC
90   LDA $B0;LOW HALF OF 16-BIT NUMBER IN
     $B0 AND $B1
100   ADC $C0;LOW HALF OF 16-BIT NUMBER
      IN $C0 AND $C1
110   STA $C2
120   LDA $B1;HIGH HALF OF 16-BIT NUMBER
      IN $B0 AND $B1
130   ADC $C1 ; HIGH HALF OF 16-BIT
      NUMBER IN $C0 AND $C1
140   STA $C3
150   RTS
```

When you look at this program, remember that your Atari computer stores 16-bit numbers in reverse order — high byte in the second address, and low byte in the first address. Once you understand that fluke, 16-bit binary addition isn't hard to comprehend. In this program, we first clear the carry flag of the P register. Then we add the low byte of a 16-bit number in $B0 and $B1 to the low byte of a 16-bit number in $C0 and $C1. The result of this calculation is then placed in memory address $C2. If there is a carry, the P register's carry bit will be set automatically.

In the second half of the program, the high byte of the number in $B0 and $B1 is added to the high byte of the number in $C0 and $C1. If the P register's carry bit has been set as a result of the preceding addition operation, then a carry will also be added to the high bytes of the two numbers being added. Then the result of this half of our program will be deposited into memory address $C3. When that operation is completed, the results of our addition problem will be stored, low byte first, in memory addresses $C2 and $C3.

16-Bit Subtraction

Here's a 16-bit subtraction program:

```
10 ;
20 ;THIS PROGRAM SUBTRACTS A 16-BIT
   NUMBER IN $B0 AND $B1
30 ;FROM A 16-BIT NUMBER IN $C0 AND $C1
40 ;AND DEPOSITS THE RESULTS IN $C2
   AND $C3
50 ;
60   *=$0600
65 ;
70   CLD
80   SEC ;SET CARRY
90   LDA $C0 ;LOW HALF OF 16-BIT NUMBER
     IN $C0 AND $C1
100   SBC $B0 ;LOW HALF OF 16-BIT NUMBER
      IN $B0 AND $B1
110   STA $C2
120   LDA $C1 ;HIGH HALF OF 16-BIT NUMBER
      IN $C0 AND $C1
130   SBC $B1 ;HIGH HALF OF 16-BIT NUMBER
      IN $B0 AND $B1
140   STA $C3
150   RTS
```

Since subtraction is the exact opposite of addition, the carry flag is set, not cleared, before a subtraction operation is performed in 6502 binary arithmetic. In subtraction, the carry flag is treated as a borrow, not a carry, and it must therefore be set, not cleared, so that if a borrow is necessary, there'll be a value to borrow from. After the carry bit is set, a 6502 subtraction problem is quite straightforward. In our sample problem, the 16-bit number in $B0 and $B1 is subtracted, low byte first, from the 16-bit number in $C0 and $C1. The result of our subtraction problem (including a borrow from the high byte, if one was necessary) is then stored in memory addresses $C2 and $C3.

Binary Multiplication

There are no 6502 assembly language instructions for multiplication or division. To multiply a pair of numbers using 6502 assembly language, you have to perform a series of addition operations. To divide numbers, you have to perform subtraction sequences. Here is an example of how two 4-bit binary numbers can be multiplied using the principles of addition:

```
  Ø11Ø    [$Ø6]
X Ø1Ø1    [$Ø5]
--------------
  Ø11Ø
 ØØØØ
Ø11Ø
ØØØØ
--------------
ØØ1111Ø   [$1E]
```

Notice what happens when you work this problem. First, 0110 is multiplied by 1. The result of this operation, also 0110, is written down.

What Happens Next

Next, 0110 is multiplied by 0. The result of that operation, a string of zeros, is shifted one space to the left and written down. Then 0110 is multiplied by 1 again, and the result is once again shifted left and written down. Finally, another multiplication by zero results in another string of zeros, which are also shifted left and duly noted. Once that's done, all of the partial products of our problem are added up, just as they would be in a conventional multiplication problem. The result of this addition, as you can see, is the final product $1E.

This multiplication technique works fine, but it's really quite arbitrary. Why, for example, did we shift each partial product in this problem to the left before writing it down? We could have accomplished the same result by shifting the partial product above it to the right before adding. In 6502 multiplication, that's

exactly what's often done; instead of shifting each partial product to the left before storing it in memory, many 6502 muliplication algorithms shift the preceding partial product to the left before adding it to the new one.

Multiple Precision Multiplication

We're now going to present a program that will show you how that works.

A MULTIPLE PRECISION MULTIPLICATION PROGRAM

```
10 MPR=$C0  ;MULTIPLIER
20 MPD1=$C1  ;MULTIPLICAND
30 MPD2=$C2  ;NEW MULTIPLICAND AFTER
   8 SHIFTS
40 PRODL=$C3 ;LOW BYTE OF PRODUCT
50 PRODH=$C4 ;HIGH BYTE OF PRODUCT
60 ;
70  *=$0600
80 ;
85 ;THESE ARE THE NUMBERS WE WILL
   MULTIPLY
87 ;
90   LDA #250
100   STA MPR
110   LDA #2
120   STA MPD1
130 ;
140 MULT CLD
150    CLC
160    LDA #0 ;CLEAR ACCUMULATOR
170    STA MPD2 ;CLEAR ADDRESS FOR SHIFTED
       MULTIPLICAND
180    STA PRODL ;CLEAR LOW BYTE OF
       PRODUCT ADDRESS
190    STA PRODH ;CLEAR HIGH BYTE OF
       PRODUCT ADDRESS
200    LDX #8 ;WE WILL USE THE X REGISTER
       AS A COUNTER
```

```
210 LOOP LSR MPR ;SHIFT MULTIPLIER RIGHT;
      LSB DROPS INTO CARRY BIT
220   BCC NOADD ;TEST CARRY BIT; IF ZERO,
      BRANCH TO NOADD
230   CLC
240   LDA PRODL
250   ADC MPD1 ;ADD LOW BYTE OF PRODUCT
      TO MULTIPLICAND
260   STA PRODL ;RESULT IS NEW LOW BYTE
      OF PRODUCT
270   LDA PRODH ;LOAD ACCUMULATOR WITH
      HIGH BYTE OF PRODUCT
280   ADC MPD2 ;ADD HIGH PART OF
      MULTIPLICAND
290   STA PRODH ;RESULT IS NEW HIGH BYTE
      OF PRODUCT
300 NOADD ASL MPD1 ;SHIFT MULIPLICAND
      LEFT; BIT 7 DROPS INTO CARRY
310   ROL MPD2 ;ROTATE CARRY BIT INTO BIT
      7 OF MPD2
320   DEX ;DECREMENT CONTENTS OF X
      REGISTER
330   BNE LOOP ;IF RESULT ISN'T ZERO, JUMP
      BACK TO LOOP
340   RTS
350   .END
```

A Complex Procedure

As you can see, 8-bit binary multiplication isn't exactly a snap. There's a lot of left and right bit-shifting involved, and it's hard to keep track of. In the above program, the most difficult manipulation to follow is probably the one involving the multiplicand (MPD1 and MPD2). The multiplicand is only an 8-bit value, but it's treated as a 16-bit value because it keeps getting shifted to the left, and while it is moving, it takes a 16-bit address (actually two 8-bit addresses) to hold it.

To see for yourself how the program works, type it out on your keyboard and assemble it. Then use the "G" command of your debugger to execute it. Then, while you're still in the DEBUG

mode, you can type "DC3" (for "Display $C3), and take a look at the contents of memory addresses $C3 and $C4, which should hold the 16-bit product of the decimal number 2 and the decimal number 250, which the program is supposed to multiply. The value in $C3 and $C4 should be $01F4, displayed low byte first, the hex equivalent of decimal 500, the correct product.

Not the Ultimate Multiplication Program

Although the program we've just outlined works fine, there are many algorithms for binary multiplication, and some of them are shorter and more efficient than the one just presented. The following program, for example, is much shorter than our first example, and therefore more memory efficient and faster running. One of its neatest tricks is that is uses the 6502's accumulator, rather than a memory address, for temporary storage of the problem's results.

AN IMPROVED MULTIPLICATION PROGRAM

```
10 ;
20 PRODL=$C0
25 PRODH=$C1
30 MPR=$C2
40 MPD=$C3
50 ;
60   *=$0600
70 ;
80 VALUES LDA #10
90   STA MPR
100   LDA #10
110   STA MPD
120 ;
130   LDA #0
140   STA PRODL
150   LDX #8
160 LOOP LSR MPR
170   BCC NOADD
180   CLC
190   ADC MPD
200 NOADD ROR A
```

```
210  ROR PRODL
220  DEX
230  BNE LOOP
235  STA PRODH
240  RTS
250  .END
```

Another Test

If you wish, you can test out this improved multiplication program the same way you tested the previous one: by executing it using your debugger's "G" command, and then taking a look at its result using the "D" command.

A Different Command

You should type "DC0" this time, since the product in this problem is stored in $C0 and $C1. The 16-bit value in $C0 and $C1 should be $0064 (stored low byte first), the hexadecimal equivalent of decimal 100, and the answer to this problem.

Feel Free to Play

You can play around with these two multiplication problems as much as you like, trying out different values and perhaps even calling the programs up from BASIC, the way we did our 16-bit addition problem a few chapters ago. The best way to become intimately familiar with how binary multiplication works, though, is to do a few problems by hand, using those two tools of our forefathers, a pencil and a piece of paper. Work enough binary multiplication problems on paper, and you'll soon begin to understand the principles of 6502 multiplication.

Multiprecision Binary Division

It's unlikely that you'll ever have an occasion to write a multiprecision binary long division program. And even if the need should arise, you'd probably have no use for the limited program and explanation we could publish here.

Nevertheless . . .

Still, this chapter would not be complete without an example of a binary long division program. So here is a simple (but tricky) program for dividing a 16-bit dividend by an 8-bit divisor. The result is an 8-bit quotient.

A Tricky Program

This program is even more subtly designed than the multiplication program we presented a few paragraphs ago. During the execution of the program, the high part of the dividend is stored in the accumulator and the low part of the dividend is stored in a variable called DVDL. The program contains a lot of shifting, rotating, subtracting, and decrementing of the X register. When it ends, the quotient is in a variable labeled QUOT and the remainder is in the accumulator. That's true until line 380 when the remainder is moved out of the accumulator and into a variable called RMDR. Then, finally, an RTS instruction ends the program.

A SIMPLE DIVISION PROGRAM

```
10 ;
20 ;DIVISION.SRC
30 ;
40   *=$0600
50 ;
60 DVDL=$C0 ;LOW PART OF DIVIDEND
70 DVDH=$C1 ;HIGH PART OF DIVIDEND
80 QUOT=$C2 ;QUOTIENT
90 DIVS=$C3 ;DIVISOR
100 RMDR=$C4 ;REMAINDER
110 ;
120   LDA #$1C ;JUST A SAMPLE VALUE
130   STA DVDL
140   LDA #$02 ;THE DIVIDEND IS NOW $021C
150   STA DVDH
160   LDA #$05 ;ANOTHER SAMPLE VALUE
170   STA DIVS ;WE'RE DIVIDING BY 5
180   ;
```

```
190   LDA DVDH ;ACCUMULATOR WILL HOLD
      DVDH
200   LDX #08 ;FOR AN 8-BIT DIVISOR
210   SEC
220   SBC DIVS
230 DLOOP PHP ;THE LOOP THAT DIVIDES
240   ROL QUOT
250   ASL DVDL
260   ROL A
270   PLP
280   BCC ADDIT
290   SBC DIVS
300   JMP NEXT
310 ADDIT ADC DIVS
320 NEXT DEX
330   BNE DLOOP
340   BCS FINI
350   ADC DIVS
360   CLC
370 FINI ROL QUOT
380   STA RMDR
390   RTS ;END IT
```

Not the Ultimate Division Program

As complex as this program appears, it is not by any means the world's best binary long division routine. It isn't the most accurate division program you'll ever see, and it won't handle fractions, decimal points, very long numbers, or signed numbers. If a versatile, accurate multiprecision division program is what you need, you'll have to look toward the floating point package built into your Atari's operating system.

The Atari floating point package is not easy to use, but more or less complete instructions on how to use it can be found in the Atari programmer's guidebook *De Re Atari*. If you decide not to use your computer's FP package, you can take a look at the many division and other arithmetic routines that are included in many 6502 assembly language manuals and "cookbooks." Quite a few arithmetic routines that are yours for the asking are

published in manuals such as the excellent text *6502 Assembly Language Subroutines* by Leventhal and Saville (Berkeley: Osborne/McGraw-Hill, 1982).

Signed Numbers

Before we move on to the next chapter, there are two more topics that we should briefly cover: signed numbers and BCD (Binary Coded Decimal) numbers. First we'll talk about signed numbers. Arithmetic operations cannot be performed on signed numbers using the techniques that have been described so far in this chapter. However, if some slight modifications are made in those techniques, the 6502 chip in your Atari computer is capable of adding, subtracting, multiplying and dividing signed numbers. If you want to perform arithmetic operations on signed numbers, the first thing you'll have to know is how to represent their signs. Fortunately, that isn't difficult to do. To represent a signed number in binary arithmetic, all you have to do is let the leftmost bit (bit 7) represent a positive or negative sign. In signed binary arithmetic, if bit 7 of a number is zero, the number is positive. If bit 7 is a 1, the number is negative.

Obviously, if you use one bit of an 8-bit number to represent its sign, you no longer have an 8-bit number. What you then have is a 7-bit number or, if you want to express it another way, you have a signed number that can represent values from –128 to +127 instead of from 0 to 255. It should also be obvious that it takes more than the redesignation of a bit to turn unsigned binary arithmetic operations into signed binary arithmetic operations. Consider, for example, what we would get if we tried to add the numbers +5 and –4 by doing nothing more than using bit 7 as a sign:

```
  0000 0101  (+5)
+ 1000 0100  (–4)
-------------------
  1000 1001  (–9)
```

That answer is wrong. The answer should be 1.The reason we arrived at the wrong answer is that we tried to solve the problem

without using a concept that is fundamental to the use of signed binary arithmetic: the concept of *complements*.

Complements are used in signed binary arithmetic because negative numbers are *complements* of positive numbers. And complements of numbers are very easy to calculate in binary arithmetic. In binary math, the complement of a 0 is a 1, and the complement of a 1 is a 0. It might be reasonable to assume, therefore that the negative complement of a positive binary number could be arrived at by complementing each 0 in the number to a 1, and each 1 to a 0 (except for bit 7, of course, which must be used for the purpose of representing the number's sign). This technique of calculating the complement of a number by flipping its bits from 0 to 1 and from 1 to 0 has a name in assembly language circles. It's called *one's complement.*

To see if the one's complement technique works, let's try using it to add two signed numbers, say +8 and –5.

```
  0000 1000 (+8)
+ 1111 1010 (–5) (one's complement)
-------------------
  0000 0010 (+2) (plus carry)
```

Oops! That's wrong, too! The answer should be plus 3. Well, that takes us back to the drawing board. One's complement arithmetic doesn't work.

But there's another technique, which comes very close to one's complement, that does work. It's called *two's complement*, and it works like this: first calculate the one's complement of a positive number. Then simply add one. That will give you the two's complement, the true complement, of the number. Then you can use the conventional rules of binary math on signed numbers — and, if you don't make any mistakes, they'll work every time. Here's how:

```
  0000 0101 (+5)
+ 1111 1000 (–8) (two's complement)
-------------------
  1111 1101 (–3)
```

Here's another two's complement addition problem:

```
  1111 1011 (–5) (two's complement)
+ 0000 1000 (+8)
-----------------
  0000 0011 (+3) (plus carry)
```

As we said, it works every time. Unfortunately, it's not easy to explain why. There are some lovely mathematical proofs, and if you're interested in what they are, you can find them in numerous textbooks on the theory of binary numbers. At the moment, though, the most important thing to know about two's complement arithmetic is how to use it, should the need ever arise.

Using the Overflow Flag

There's one more important fact to remember about signed binary arithmetic: when you add signed numbers, you use the overflow (V) flag rather than the carry flag to carry numbers from one byte to another. The reason for this is as follows: The carry flag of the P register is set when there's an overflow from bit 7 of a binary number. But when the number is a signed number, bit 7 is the sign bit — not part of the number! So the carry flag cannot be used to detect a carry in an operation that involves signed numbers. You can solve this problem by using the overflow bit of the processor status register. The overflow bit is set when there is an overflow from bit 6, not bit 7. So it can be used as a carry bit in arithmetic operations on signed numbers.

BCD (Binary Coded Decimal) Numbers

Another variety of binary arithmetic that it might be helpful to know something about is the BCD (Binary Coded Decimal) system. In BCD notation, the digits 0 through 9 are expressed just as they are in conventional binary notation, but the hexadecimal digits A through F (1010 through 1111 in binary) are not used. Long numbers must therefore be represented differently in BCD notation than they are in conventional binary notation. The

decimal number 1258, for example, would be written in BCD notation as:

```
     1         2         5         8

0000 0001 0000 0010 0000 0101 0000 1000
```

In conventional binary notation, the same number would be written as:

```
    $0    $4    $E    $A

0000 0100 1110 1010
```

This which equates to $04EA, or the hexadecimal equivalent of 1258. BCD notation is often used in bookkeeping and accounting programs because BCD arithmetic, unlike straight binary arithmetic, is 100% accurate. BCD numbers are also sometimes used when it is desirable to print them out instantly, digit by digit as they are being used — for example, when numbers are being used for on screen scorekeeping in a game program.

The main disadvantage of BCD numbers is that they tend to be difficult to work with. When you use BCD numbers, you must be extremely careful with signs, decimal points and carry operations, or chaos can result. You must also decide whether you want to use an 8-bit byte for each digit, which wastes memory, since it really only takes 4 bits to encode a BCD digit, or whether to "pack" two digit into each byte, which saves memory but consumes processing time.

Fortunately, as we have pointed out, you'll probably never have to use most of the programming techniques described in this chapter, but an understanding of how they work will definitely make you a better Atari assembly language programmer.

Chapter Eleven

Beyond Page 6

Most beginning assembly language programmers write short routines that will fit easily in short blocks of memory. That's why the engineers who designed your Atari set aside page 6, a block of memory extending from $0600 to $06FF, as an area for user written assembly language programs. As you become more and more skilled at assembly language programming, however, it's more than likely that you'll eventually start writing programs that consume more than 256 bytes of memory available on page 6. Figuring out where to put long assembly language programs in an Atari computer can be a tricky problem.

The main problem is not usually the amount of free memory that's available, but rather where it is situated in your computer's RAM. In most computers, most of the memory available for user written programs is almost all in one place. It usually extends from a low address, just above where the computer's operating system ends, to a high address, just below the place where a block of RAM called screen memory begins.

The memory organization of an Atari computer is not quite that simple. In an Atari computer, the space available for user written programs is scattered all over the memory map, and learning how to discover little corners where you can stash object code without clobbering your Atari's operating system can become quite a challenging, if sometimes frustrating, task. This situation exists because there are several different kinds of Atari computers, and because all of them are very versatile machines. Atari computers have RAM capacities that range from 16K to 64K, and they can be used in many different graphics modes and with one to four disk drives. Yet they are all software compatible and keeping them software compatible has led to some interesting tricks that have been pulled by Atari in memory design.

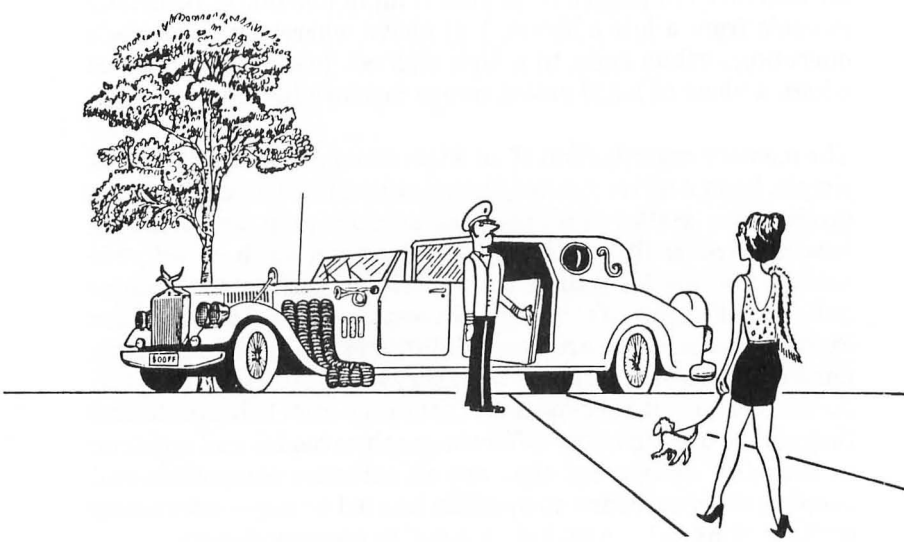
Unfortunately, the people who design Atari computers haven't yet found a way — and it's doubtful that anyone could — to keep all Atari computers compatible and to keep their memory organization simple at the same time. Nevertheless, there are a few safe places to store your assembly language programs in your Atari's memory. To help you find them, here's a simple *memory map* of your computer's RAM:

The High Rent District

Page Zero

From $0000 to $00FF

Page zero, the block of memory that extends from $0000 to $00FF, is the high rent district in your computer's RAM. Memory space there is so valuable that very little of it is available for short term use by user written programs. If you want to write high performance programs, particularly programs that use indexed addressing, then you'll have to find at least a few free memory locations on page zero. If you look around carefully, you'll find a few free locations there.



Your Atari's operating system consumes most of page zero. There are a couple of small memory blocks that aren't used by the OS, but they aren't always available for user written programs, since they are often dedicated to other uses. When you write a program using an assembler, for example, your assembler always uses some of page zero. If your program is designed to be called from BASIC, then the BASIC interpreter that you'll have to use will use up more of page zero.

The floating point math routines in your Atari's operating system also consume a block of memory on page zero. However, if you write programs that don't use the Atari FP package, then the block of memory reserved for that package will be free. Specifically, these are the memory locations on page zero that you can use, and the conditions under which you can use them:

Memory Map for Page Zero

$00 - $AF — Reserved for use by operating system.

$B0 - $CF — Bytes left free by Assembler Editor cartridge.

$CB - $D1 — Bytes left free by BASIC cartridge.

$D4 - $FF — Free if you don't use your OS floating point package, the Atari Assembler Editor cartridge, or a BASIC program that uses the Atari OS floating point routines.

Zero Page Locations You Can Use

In the programs presented in this book, all of the page zero locations that have been used have fallen into the block of memory extending from $B0 to $CF. Look at the last chart, and you'll see why. When you write a program using the Atari Assembler Editor cartridge, the $B0 to $CF block is the only part of page zero that's not used either by your computer's operating system or your Assembler Editor cartridge. (The MAC/65 assembler uses less of page zero, but the programs in this book were written to be compatible with both assemblers.) If you're writing a pro-

gram designed to be called from BASIC, the portion of page zero that you can use is even smaller. Then your free space will extend only from memory address $CB through memory address $D1. That's seven whole bytes of page zero that you can use for your program! There are two easy ways to get around this limitation. Either write programs that use very little of page zero, or write programs that don't have to be called from BASIC!

Memory Addresses $100 - $6FF

Operating System RAM

Memory addresses $100 to $5FF in your computer are reserved for operating system RAM. Here's how this block of memory is divided up:

$100 - $1FF — Your computer's hardware stack. You can use this block of memory, but only for stack manipulation operations. You remember those: PLA, PHA, PLP, PHP, JSR and RTS.

$200 - $3BF — IOCB's (input/output control blocks) and miscellaneous OS variables. Your computer uses this section of memory mainly for communicating with input and output devices. It's not available for use by user written programs.

$3C0 - $3E7 — Printer buffer, where data is held while it's on its way to your printer.

$3E8 - $3FC — Reserved for OS; not available to you.

$3FD - $47F — Cassette buffer, a holding area for data between your computer and data cassette recorder.

$480 - $57D — Reserved for use by BASIC cartridge. May be used by assembly language programs not called from BASIC.

$57E - $5FF — OS floating point package. Used by BASIC. May also be used by user written assembly language programs. Free for other uses in assembly language programs if floating point routines and BASIC cartridge are not used.

$600 - $6FF — "Page 6". This is usually available for use by user written assembly language programs. There is one important exception, however. When the INPUT statement is used in a BASIC program, and more than 128 characters are input via the keyboard, the characters in excess of 128 are stored on page 6. In this case, object code stored from $600 to $67F might be erased. However, addresses $680 to $6FF are unconditionally available for user written programs.

Memory Addresses $700 - "MEMLO"

DOS Dedicated RAM

Beginning at memory address $700, there's a block of memory that's reserved for use by your computer's disk operating system. The size of this block of memory is affected by a number of factors, including how many disk drives you use, and whether or not you're currently using the disk utility programs listed on your computer's DOS menu. Because the size of this memory block varies so widely from computer to computer, and from application to application, your Atari has been equipped with a special 16-bit variable that can tell you at a glance how large its DOS and DOS related block of memory is. That variable is called MEMLO, and it's stored in memory addresses $2E7 and $2E8 (743 and 744 in decimal notation). The 16-bit value that MEMLO contains is a very important number. It's not only the address where your computer's DOS routines end; it's also the address where the biggest block of user addressable memory in your computer begins. Once you know what the value of MEMLO is, you'll know exactly where to start the object code for your own machine language programs.

From "MEMLO" to "MEMTOP"

Free RAM

The RAM that you can use freely extends from the variable called MEMLO ($2E7 AND $2E8) to another 16-bit variable named, logically enough, MEMTOP. You can see what value MEMTOP contains by peering into the contents of memory addresses $2E5

and $2E6 (741 and 742 in decimal notation). Once you know what the value of MEMTOP is, you'll know the upper limit of the block of memory in which you can store your assembly language programs.

Above "MEMTOP"

Screen Memory

The memory block extending upward from MEMTOP is your computer's screen display area, an area reserved for the data it uses to create its screen display. Programs that create their own custom screen displays can overwrite this block of RAM, but if you use your computer's built-in screen displays, you'll have to stay out of this section of memory, because that's where they are located.

$8000 to $9FFF

"Cartridge Slot B"

When the Atari 800 was designed, this block of memory was dedicated to "Cartridge B," the right-hand slot in a pair of cartridge slots. As it turned out, the Cartridge B slot was utilized by only one or two programs written for the Atari. So newer Atari computers, the 1200XL and subsequent models, have been designed with only one cartridge slot. And that means that memory addresses $8000 to $9FFF, originally designed for "Cartridge B," are now available for use in user written programs.

$8000 to $BFFF

Cartridge Slot A

Cartridge Slot A is the slot that holds most Atari cartridges, the Atari Assembler Editor cartridge, and all other kinds of cartridge based programs. Many disk based programs also use this block of memory. When you write programs using cartridges or utility programs that occupy this memory block, there's no easy way for you to use that space for your programs. If you get good at writing relocatable code, however, there's no reason you

can't use this memory block; after all, other Atari assembly language programmers do! Usually this slot goes from $A000 to $BFFF, but some can go from $8000 to $BFFF.

$C000 to $CFFF

Not used in Atari 400 and 800 because they do not contain this RAM location. Used by OS in the newer models. Enter at your own risk; not recommended for use in user written programs.

$D000 to $D7FF

Atari hardware Read/Write registers

Not available for user written programs.

$D800 to $DFFF

Floating Point ROM

Available for use by user written programs if the OS floating point package is not used, and if BASIC routines that call FP routines are not used. This is only available in the XL line, the Atari 400 and 800 do not have RAM available here.

$E000 to $FFFF

Operating System ROM

Not available for use by user written programs.

The Problem of Allocating Memory

Once you know your Atari's memory map like the palm of your hand, you'll almost be ready to start allocating memory to assembly language programs. Almost, but not quite. First you'll have to learn how to solve two big problems that can be a real pain in the neck to Atari assembly language programmers. These two problems are:

- Making sure that your source code programs and your object code programs don't overwrite each other.

- Keeping BASIC and machine language programs away from each other in the computer's memory.

Actually, these two problems are not difficult to solve. But they seem to be more complicated than they really are because of a confusing system that has been developed by Atari for keeping track of the lowest address of free memory.

In your computer's operating system, there are two variables, or *pointers*, that are designed to help you figure out where you can start machine language programs in your computer's memory. One of these variables is called LOMEM, and the other is called MEMLO. If you think that's confusing, that's only the beginning. Sometimes LOMEM and MEMLO are interchangeable, and sometimes they aren't. While their abbreviations are merely confusing, their full names are downright misleading. On pages D-1 and D-2 of your *Atari BASIC Reference Manual*, *MEMLO* is identified as your computer's *operating system low memory pointer*, and *LOMEM* is identified as your Atari's *BASIC low memory pointer*. Unless you want to wind up totally baffled, don't pay any attention to either of these names. Here's how your computer's LOMEM and MEMLO pointers *really* work, and what they can really tell you.

The LOMEM Pointer

Your Atari's LOMEM pointer is a 16-bit variable stored in memory addresses $80 and $81 (or 128 and 129 in decimal notation). LOMEM always contains the beginning address of a block of memory in your computer called the *edit text buffer*. The edit text buffer is a special buffer designed to hold ATASCII text while that text is being written and edited. When you write or edit a BASIC program, the edit text buffer is where your program is stored until you're ready to run or solve it. The edit text buffer is also used to store assembly language source code programs.

When you turn your computer on, the address in LOMEM is the lowest address of your computer's free RAM, the lowest address (not including page 6) at which user written programs can safely begin. If you don't have any disk drives connected to your computer, then the value of LOMEM will be $0700 when you turn on your computer system. If you do have one disk drive or more hooked-up to your computer, and they're turned on, then the block of RAM that lies just below LOMEM will be the memory block where your computer's disk operating system (DOS) is stored.

Changing the LOMEM Pointer

Even though the value of LOMEM is automatically set to a predetermined value when you turn your computer on, you can change it any time you like. When the value of LOMEM changes, the starting address of your computer's edit text buffer will automatically shift to the address that has been loaded into the LOMEM pointer. What this means, is that you can change the location of your computer's edit text buffer at any time you like by merely poking (or loading) a new 16-bit address into the LOMEM pointer.

Why would you want to change the location of the edit text buffer? The most common purpose for doing it is to keep source code programs and object code programs away from each other while source code is being written, edited and assembled. To understand how LOMEM can be manipulated to keep source code and object code away from each other, it helps to understand how LOMEM and MEMLO are related.

The MEMLO Pointer

MEMLO is also a 16-bit value, but is stored in memory addresses $2E7 and $2E8 of your computer (or 743 and 744 in decimal notation). When you turn your computer on, without a cartridge or a disk in it, your computer's MEMLO pointer always contains the lowest free address in RAM, the lowest address at which user

written programs can begin. That means, that when you turn your computer on, MEMLO and LOMEM contain exactly the same address, the lowest free address in RAM. That address is also the starting address of your computer's edit text buffer.

Now let's suppose that you're sitting at your computer, and that your computer, assembler, your disk drives and your program data disk are all up and running and ready to go. Let's now suppose that your assembler's EDIT prompt has just come on, and that you're ready to start typing in some source code. Since you've just turned your computer on, your LOMEM and MEMLO pointers will contain the same address when you begin your editing session, the lowest free address in your computer's RAM. Since you haven't changed any default values, that address will also be the starting address of your computer's edit text buffer.

When you start typing in source code, therefore, your source code will always start at the lowest free address in your computer's memory, which brings us to an unfortunate but obvious conclusion: When you write an assembly language program using the MAC/65 assembler or the Atari Assembler Editor cartridge, you can't start your object code program at the address contained in your LOMEM and MEMLO pointers; if you try to do that, your source code and your object code will attempt to overwrite each other, and your computer will reward you with an error message!

Solving the Problem

So what's a poor programmer to do? Well, you can do a couple of things. For example, you could use a special command called SIZE that's provided as a special bonus with both the MAC/65 assembler and the Atari Assembler Editor cartridge. It's easy to use the SIZE command. All you have to do is put your assembler into its EDIT mode, type the word SIZE, and hit your RETURN key. Your computer will then print a line on your screen that looks something like this:

```
1CFC    2062    9C1F
```

What the "SIZE" Line Means

The first of these three numbers, 1CFC, is the current value of your computer's LOMEM pointer — the lowest usable address in your computer's RAM. The second number, 2062 in our example, is the value of your computer's MEMLO pointer —the address at which your edit text buffer currently ends. (We used the word "currently," because the length of your Atari's edit text buffer can vary; as you type in source code, your edit text buffer will get longer. When you delete source code, it will get shorter).

The third number in your assembler's SIZE line (9C1F in our sample line) is the value of another important pointer — MEMTOP, the highest memory address that can be used safely in a user written program. (Just above MEMTOP is where your computer's screen display memory begins.)

Three Basic Facts

Your assembler's SIZE command, then, can provide you with three important facts that can help you with your memory allocation problems. It can tell you:

- Where your source code program begins.
- Where your source code program ends.
- How much free RAM there is between the end of your source code program and the start of your computer's screen display memory.

A Word of Caution

If you use the SIZE command to decide where to store your object code, however, we do have one more warning: Most assembly language programs produce a symbol table, a list of labels used within a program and their corresponding memory addresses. When you write a program containing labels using an Atari Assembler Editor cartridge, your assembler will automatically store a symbol table just above your computer's edit text buffer. So when you use the Atari Assembler Editor to write a program

that produces a symbol table, you must always leave some space between the end of your edit text buffer (the second number in the SIZE line) and the beginning of your object code program. Otherwise, your symbol table and your object code program may overwrite each other, with potentially disastrous results.

No Need to Guess

Fortunately, you don't have to guess how much room you'll need for a symbol table; you can figure that out. You'll need three bytes for each label in your program, plus one byte for each typed character in each label. That sounds like a lot of calculating, and it is; but if you do it long enough, you'll eventually become very proficient at guessing the lengths of symbol tables.

There's an Easier Way!

Now that you know all that, here's some good news. There's another command that can be used with both MAC/65 and the Atari Assembler Editor, a command called LOMEM, that can make this whole business of allocating memory much easier. Here's how to use the LOMEM command: When you've loaded your assembler into RAM — or have slipped your assembler cartridge into your computer and turned your computer on, just type the word LOMEM followed by a hexadecimal number — like this:

```
LOMEM $5000
```

Then hit your RETURN key. That simple procedure will automatically reset your computer's LOMEM pointer, and will place any source code you subsequently *write* above the object code that it will generate, instead of below it. You can then store your source code anywhere you like, in the wide open spaces *above* your machine code, instead of in the cramped space beneath it.

With the LOMEM command, you'll never have to worry about what the current value of MEMLO is, and you'll never have to count the number of typed characters in a symbol table. There's

one fact to remember, though; if you want to use the LOMEM command, you must use it *before* you start writing a program. If you use the LOMEM command with a MAC/65 assembler, it will wipe out everything stored in RAM, just like the command NEW. When you're writing programs using the Atari Assembler Editor cartridge, LOMEM must be the very first command you use when you you turn your computer on. Otherwise, it simply won't work. If you forget that and still want to use LOMEM, you'll have to turn your computer off and then back on again.

Another Memory Management Problem

You can also run into memory allocation problems when you mix BASIC and assembly language, that is when you write an assembly language program that's designed to be called from a BASIC program. When you want to call a machine language program from BASIC, it's obviously necessary for both programs to be present in your computer at the same time. It's also obvious, unfortunately, that the two programs can't start at the same address. If they did, one program would overwrite the other, and chaos would result. Fortunately, there are ways to solve this problem.

Changing Your MEMLO Pointer

When you load a BASIC program into your computer's memory, your computer uses the value in MEMLO to determine where the program should be stored. If MEMLO points to the lowest free RAM address in your computer when the BASIC program is loaded, then the program will be loaded into your computer's memory starting at that address. But if MEMLO points to a higher address when a BASIC program is loaded into memory, then the program will be loaded into RAM starting at that address.

Obviously, then, the way to keep a BASIC program from overwriting a machine language program stored in low memory is to change MEMLO to a higher address before the BASIC program is loaded. It's easy to change MEMLO to a higher value before a

BASIC program is loaded. All you have to do is use a routine like this:

CHANGING THE VALUE OF THE MEMLO POINTER

```
10 ;
20 ;NEWMEMLO.SRC
30 ;
40  *=$0600
50 ;
60 NEWMLO=$5000 ;NEW MEMLO ADDRESS
65 MEMLO=$2E7 ;ADDRESS OF MEMLO
   POINTER
70 ;
80  LDA #NEWMLO&255
90  STA MEMLO
100  LDA #NEWMLO/256
110  STA MEMLO+1
120  RTS
```

What Have You Done?

When you assemble and run this routine, it will store a new address, in this case $5000, into your computer's MEMLO pointer. If you then load a BASIC program into your computer's memory, the starting address of that program will not be the lowest address in free RAM, as it would ordinarily be. Instead, the BASIC program will start at memory address $5000. That will reserve a big block of memory for user written machine language programs: the block extending from the lowest byte in free RAM (the value of LOMEM) to memory address $4FFF.

A Better Way

Although the routine we've just presented will work fine in programs you run yourself, it may not be adequate in programs designed to be run by other people. That's because the MEMLO pointer is set not only at power-up time, but also when the Atari SYSTEM RESET button is pushed. So if the SYSTEM RESET

button is pressed accidentally by a program user, MEMLO will be reset to its default value. A more complex MEMLO setting program that is immune to accidents such as the hitting of SYSTEM RESET can be found on pages 8-11 of *De Re Atari*, the assembly language programmer's guide published by Atari. That program, yours for the typing, can even be run as an AUTORUN.SYS routine, and is just about as user transparent as such a program can be.

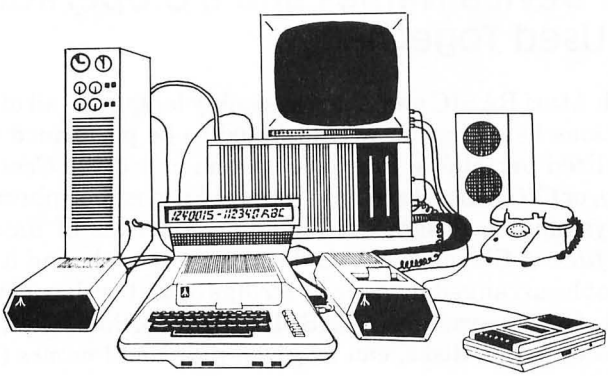
Chapter Twelve

I/O and You

Types of I/O Devices

Many kinds of I/O devices can be connected to your Atari computer. But there are seven specific kinds of devices that can be addressed in both Atari BASIC and Atari assembly language using specific procedures and specific commands. Each of these seven types of devices has a unique one letter abbreviation, or device name, by which it can be addressed in both Atari BASIC and Atari assembly languages. These seven types of devices, and their corresponding device names in both BASIC and assembly language, are:

- Keyboard (K:).
- Line Printer (P:).
- Program (Cassette) Recorder (C:).
- Disk Drives (D:) (or, if more than one disk drive is used, D1:, D2:, D3:, and D4:).
- Screen Editor (E:).
- TV Monitor (Screen) (S:).
- RS-232 Serial Interface (R:).



Note the colon following the letter in each of these abbreviations. The colon is an integral part of each device name, and may not be omitted.

The Eight Atari I/O Operations

In both Atari BASIC and Atari assembly language, there are eight I/O operations that can be performed using the seven abbreviations, or device names, listed above. These eight I/O operations are:

- OPEN (to open a specified device).
- CLOSE (to close a specified device).
- GET CHARACTER (to read one character from a specified device or file).
- PUT CHARACTER (to write one character to a specified device or file).
- READ RECORD (to read the next record, a string which must end with a return character [$9B] from a specified device or file).
- WRITE RECORD (to write a record, a string, which must end with a return character [$9B] to a specified device or file).
- STATUS (to get the status of a specified device).
- SPECIAL (to perform a specified special operation on specified device used primarily in file management and RS-232 serial operations).

How Device Names and I/O Operations are Used Together

In both Atari BASIC and Atari assembly language, all of the I/O operations listed earlier are designed to be performed using a centralized peripheral interface system called the *Central I/O Utility*, or *CIO*. The Atari CIO system, like most peripheral interface systems, is designed to handle sequences of data bytes called *files*. A file may contain data, text, or both, and it may or may not be arranged by *records*, strings of text or data separated by end of line characters (ATASCII code $9B). Some files, such as files recorded on disks, can be given individual *names* (such as

"D1:TESTIT.SRC). Other files, such as those used with the Atari screen editor or line printer, do not have individual names, but are addressed simply by the name of the device on which they appear, for example, "E:" or "P:".

Both Atari BASIC and Atari assembly language allow programmers to access up to eight different devices and/or files at the same time. In both BASIC and and assembly language, this access is provided via eight dedicated blocks of memory that are called *Input/Output Control Blocks*, or *IOCBs*. In Atari assembly language, just as in Atari BASIC, the eight IOCBs are numbered from 0 through 7. In both assembly language and BASIC, any free IOCB number can be assigned to any I/O device, although IOCB #0 is always assigned to the screen editor when an Atari computer is first turned on, and is the screen editor's default IOCB number.

Opening a Device

In both Atari BASIC and Atari assembly language, I/O devices are assigned IOCB numbers when they are first addressed, or *opened*. When a device is first opened for either read or write operations, an IOCB number must be assigned to it. Once an IOCB number has been assigned to a device, the device can be referred to by that number until a command to close the device is issued. Once a device is closed, the IOCB number that was assigned to it becomes free again, and can be used to open any other device in your computer system.

Assembly Language Lacks IOCB Commands

In Atari BASIC, specific commands are provided to open, close, read from and write to any I/O devices that may be connected to a computer. No such commands exist in 6502 assembly language. The IOCB system used in Atari computers does provide the assembly language programmer with a means of handling all of the I/O devices that can be connected to an Atari computer. It can handle it in a way that is relatively easy to manage and easy to understand.

Opening a Device Using Atari BASIC

It is not difficult to open a device or a file using Atari BASIC. To open a device or a file, all a BASIC programmer has to do is write a line using the following formula.

```
10 OPEN #n,n1,n2,filespec
```

The following is an example of an Atari BASIC statement written using the standard IOCB formula.

```
10 OPEN #2,8,0,"D1:TESTIT.BAS"
```

As you can see, there are five components in an OPEN statement in Atari BASIC: The OPEN command itself, a series of three parameters separated by commas, and a device name plus a file name, if applicable. A mandatory "#" mark appears before the first parameter after the OPEN statement and the device name is followed by a mandatory colon. In addition, the device name and the file name, if applicable, are enclosed in mandatory quotation marks. The meanings of the five components of an OPEN statement are explained below.

1. "OPEN" — the OPEN command.

2. "#n" (#2 in the sample statement above) — The IOCB number. This number, as we have pointed out, ranges from 0 through 7. "#2" in this position means "IOCB #2."

3. "n1" (8 in our example) — A code number for a specific type of input or output operation. In our sample OPEN statement, the "8" in this position is the code number for an output (open for write) operation.

4. "n2" (0 in our sample statement) — A device dependent auxiliary code sometimes used for various purposes (in this case, though, not used).

5. "filespec" — A device name plus a file name, if applicable. In our example, "D1:TESTIT.BAS" refers to a file called TESTIT.BAS which our computer will expect to find stored on a disk in disk drive 1.

How BASIC Processes an "OPEN" Command

When your computer encounters an OPEN command while processing a BASIC program, it carries out a series of standardized operations using the values in each of the four parameters of the OPEN statement. When all of those operations are completed, BASIC jumps to a special OS subroutine called the *CIO vector*, or *CIOV*. The CIOV subroutine then automatically opens the device in question, referring to the parameters that were contained in the OPEN statement (and are now stored in certain memory locations) in order to make sure that the proper device is opened for the kind of access called for in the OPEN statement.

Advantages of Assembly Language I/O Operations

To understand how a device is opened using Atari assembly language, it's helpful to know how devices are opened using Atari BASIC. That's because BASIC programs and assembly language programs open devices in exactly the same way. The only difference is that when you open a device using BASIC, your BASIC interpreter does most of the work for you. When you use assembly language, you have to do all of the work yourself. Fortunately, there's a payoff for doing all of this extra work. When you control your system's CIO system using assembly language, you have a lot more control over the system than you do when you allow BASIC to do all the work.

Opening a Device Using Assembly Language

Now let's take a look at exactly how devices are opened, read from, written to and closed, in both Atari BASIC and Atari assembly language.

Another Look at IOCBs

As we've pointed out, the I/O operations of an Atari computer are controlled using a series of eight I/O control blocks, or IOCBs. Each of these I/O control blocks is an actual block of memory in your computer. Each IOCB is 16 bytes long, and each byte in each IOCB has a specific name and a specific function. *Moreover, each byte in each IOCB has the same name, and performs the same kind of function, as the corresponding byte in every other IOCB.* That's important, so let's say it again in a different way: Each byte in each IOCB in your computer has the same name, and performs the same kind of function, as the byte *with the same offset* in each other IOCB.

Indirect Addressing in IOCB Operations

The reason this fact is important is that indirect addressing is used quite often in IOCB operations. Indirect addressing, as we've explained several times in this book, is an addressing technique in which a memory location is sought out by means of an offset value stored in the 6502 processor's X or Y register. Since the offsets of all of the bytes in all Atari IOCBs correspond to each other, that makes the indirect addressing mode very easy to use in Atari IOCB operations.

The 16 Bytes of an IOCB

This concept is much easier to understand when examples are given. So here is an actual assembly language program that will now be used to explain the Atari I/O system. If this program looks familiar, that's because it's almost exactly like the one you were asked to type in back in Chapter 7. It is the same one we used to print messages on the screen. If you still have that program stored on a disk, you can load it into your computer, and with just a few changes, you can turn it into an exact replica of the program below. That way you won't have to type it all over again.

PROGRAM FOR PRINTING ON THE SCREEN

```
10 ;
20  .TITLE "PRNTSC ROUTINE"
30  .PAGE "ROUTINES FOR PRINTING ON THE
    SCREEN"
40 ;
50  *=$5000
60 ;
70 BUFLEN=255 ;(EXPANDED BEYOND PREVIOUS
   LIMITS)
80 ;
90 EOL=$9B ; ATASCII CODE FOR END OF LINE
   CHARACTER
100 ;
110 OPEN=$03 ;TOKEN FOR OPENING A DEVICE
    OR FILE
120 OWRIT=$08 ;TOKEN FOR "OPEN FOR WRITE
    OPERATIONS"
130 PUTCHR=$0B ;TOKEN FOR "PUT CHARACTER"
140 CLOSE=$0C ;TOKEN FOR CLOSING A DEVICE
    OR FILE
150 ;
160 IOCB2=$20 ;OFFSET FOR IOCB NO. 2
170 ICCOM=$342 ;COMMAND BYTE (CONTROLS
    CIO OPERATIONS)
180 ICBAL=$344 ;BUFFER ADDRESS (LOW BYTE)
190 ICBAH=$345 ;BUFFER ADDRESS (HIGH BYTE)
200 ICBLL=$348 ;BUFFER LENGTH (LOW BYTE)
210 ICBLH=$349 ;BUFFER LENGTH (HIGH BYTE)
220 ICAX1=$34A ;AUXILIARY BYTE NO. 1
230 ICAX2=$34B ;AUXILIARY BYTE NO. 2
235 ;
240 CIOV=$E456 ;CIO VECTOR
250 ;
260 DEVNAM .BYTE "E:",EOL
270 ;
280 OSCR ;OPEN SCREEN ROUTINE
290   LDX #IOCB2
300   LDA #OPEN
310   STA ICCOM,X
```

```
320 ;
330   LDA #DEVNAM&255
340   STA ICBAL,X
350   LDA #DEVNAM/256
360   STA ICBAH,X
370 ;
380   LDA #OWRIT
390   STA ICAX1,X
400   LDA #0
410   STA ICAX2,X
420   JSR CIOV
430 ;
440   LDA #PUTCHR
450   STA ICCOM,X
460 ;
470   LDA #TXTBUF&255
480   STA ICBAL,X
490   LDA #TXTBUF/256
500   STA ICBAH,X
510   RTS
520 ;
530 PRNT
540   LDX #IOCB2
550   LDA #BUFLEN&255
560   STA ICBLL,X
570   LDA #BUFLEN/256
580   STA ICBLH,X
590   JSR CIOV
600   RTS
605 ;
610 CLOSED
620   LDX #IOCB2
630   LDA #CLOSE
640   STA ICCOM,X
650   JSR CIOV
660   RTS
670 ;
680 TXTBUF=*
690 ;
700   *=*+BUFLEN
710 ;
720   .END
```

"PRNTSC.SRC," Line by Line

Now, at last, we'll take a good close look at this program and see how it works, line by line. We'll start with the first three lines of the program, lines 290 through 310.

Initializing a Device for "OPEN"

```
290  LDX #IOCB2
300  LDA #OPEN
310  STA ICCOM,X
```

Substitute literal numbers for the variables in these three lines, and this is how they will read.

```
290  LDX #$20
300  LDA #$03
310  STA $342,X
```

These three instructions are all it takes to open a device in Atari assembly language. In order to understand what they do, you have to know something about the structure of an Atari IOCB. As we've pointed out, there are eight IOCBs in your Atari's operating system, and each one contains 16 bytes (or $10 bytes in hexadecimal notation). That means that to address IOCB #1, you have to add 16 (or $10) bytes to the address of IOCB #0 and to address IOCB #2, you have to add 32 (or $20) bytes to the address of IOCB #0. In other words, when you use the address of IOCB #0 as a reference point (as the Atari CIO system does), the *offset* you have to use is 32 in decimal notation, or $20 using the hexadecimal system. Here are all of the IOCB offsets used in the Atari CIO system:

The Eight Atari IOCB Offsets

IOCB0=$00	IOCB4=$40
IOCB1=$10	IOCB5=$50
IOCB2=$20	IOCB6=$60
IOCB3=$30	IOCB7=$70

Now let's take another look at our literal value version of the first three lines of the PRNTSC.SRC program:

```
290  LDX #$20
300  LDA #$03
310  STA $342,X
```

Now you can begin to see why the number $20 has been loaded into the X register in line 290. Obviously, it's going to be used as an offset in line 310, but before we move on to line 310, let's take a look at line 300, the line in between. In line 300, the accumulator is loaded with the number $03 — which has been identified back in line 110 of the program as the "token for opening a device." Now what does that mean?

I/O Tokens

Well, in the Atari CIO system, each of the eight I/O operations described at the beginning of this chapter can be identified by a one-digit (hex) code, or *token*. Here is a complete list of those tokens, and the operations for which they stand.

Token	Name	Function
$03	OPEN	Open a specified device or file.
$04	OREAD	Open a device or file for read operations.
$08	OWRITE	Open a device or file for write operations.
$05	GETREC	Read a record from a specified device or file.
$07	GETCHR	Read character from specified device or file.
$09	PUTREC	Write a record to a specified device or file.
$0B	PUTCHR	Write character to a specified device or file.
$0C	CLOSE	Close a specified device or file.

Line 300 Explained

Now you can see what happens in line 300 of the program PRNTSC.SRC. The accumulator is loaded with the number $03, the token for "OPEN". In line 310, the OPEN token is stored in the indirect address ICCOM,X (or $342,X). Just what is this address?

ICCOM is the name of one of the 16 bytes that every IOCB contains. Specifically, ICCOM is the first byte (the zero offset byte) in every IOCB. Look at line 170 of the PRNTSC.SRC program and you'll see that ICCOM is located at memory address $342, and is identified as the "command byte" in the Atari CIO system. It is called the command byte because it is the byte that must be addressed when devices are to be initialized, opened or closed. ICCOM is the byte that points to a set of subroutines in your computer's operating system that perform all of those functions.

IOCB Addresses

Since we have listed all of the Atari I/O devices, I/O commands, I/O offsets and I/O operation codes in this chapter so far, we might as well provide a list now of ICCOM and the rest of the 16 bytes in each of your computer's IOCBs. Here is a complete list of the bytes in each IOCB.

Byte	Adrs	Name	Function
ICHID	$0340	Handler I.D.	Preset by OS
ICDNO	$0341	Device Number	Preset by OS
ICCOM	$0342	Command Byte	Controls CIO operations
ICSTA	$0343	Status Byte	Returns status of operations
ICBAL	$0344	Buffer Address, Low	Holds address of text buffer
ICBAH	$0345	Buffer Address, High	Holds address of text buffer
ICPTL	$0346	Unused Pointer	Not used in programming

Byte	Adrs	Name	Function
ICPTH	$0347	Unused Pointer	Not used in programming
ICBLL	$0348	Buffer Length, Low	Holds length of text buffer
ICBLH	$0349	Buffer Length, High	Holds length of text buffer
ICAX1	$034A	Auxiliary Byte No. 1	Picks write or read operation
ICAX2	$034B	Auxiliary Byte No. 2	Used for various purposes
ICAX3	$034C	Auxiliary Byte No. 3	Used by OS only
ICAX4	$034D	Auxiliary Byte No. 4	Used by OS only
ICAX5	$034E	Auxiliary Byte No. 5	Used by OS only
ICAX6	$034F	Auxiliary Byte No. 6	Used by OS only

Now you can understand the operation performed in lines 290 through 310 of the PRNTSC.SRC program:

```
290  LDX #IOCB2
300  LDA #OPEN
310  STA ICCOM,X
```

In line 290, the X register is loaded with the offset for IOCB #2: the number $20. In line 300, the accumulator is loaded with the token for the OPEN operation: the number $03. In line 310, the token of the OPEN operation (the number $03) is stored in ICCOM,X: the command byte of IOCB #2. After a few more operations, we're going to issue a "JSR CIOV" statement, so our Atari will jump to the CIO vector and open IOCB #2, as we have instructed. But first, we're going to have to set a few more parameters, so our computer will know exactly what kind of operations to open IOCB #2 for. So let's zip right through the rest of this "OPEN" operation now.

Completing the "OPEN" Operation

```
330  LDA #DEVNAM&255
340  STA ICBAL,X
350  LDA #DEVNAM/256
360  STA ICBAH,X
```

```
370 ;
380  LDA #OWRIT
390  STA ICAX1,X
400  LDA #0
410  STA ICAX2,X
420  JSR CIOV
```

In lines 330 through 360, the text buffer address in IOCB #2 is loaded with the address of a variable defined in line 260 as DEVNAM. The variable DEVNAM, as you can see by looking at line 260, contains the ATASCII code for the character string "E:" — the device name for the Atari screen editor. We could have opened IOCB #2 for any other I/O device in exactly the same way. If we wanted to use IOCB #2 as a printer IOCB, for example, we could have written line 260 this way:

```
260  DEVNAM .BYTE "P:",EOL
```

Then, in lines 330 through 360, the address of the ATASCII string "P:",EOL would be loaded in ICBAL,X. With that tiny change, the PRNTSC program, instead of opening your computer screen as an output device, would open your printer! You can also use this same programming procedure to open a specific file on a disk so that you can read from it or write to it, on either a character-by-character or a record-by-record basis. In the PRNTSC program, we could open a disk file instead of the screen editor by changing line 260 to read something like this:

```
260  DEVNAM .BYTE "D1:TESTIT.BAS",EOL
```

Then, instead of opening the screen editor, our program would open the disk file TESTIT.BAS (provided, of course, that there was a disk drive connected to our computer and that all other necessary conditions for opening such a file existed). We have just seen two examples of the tremendous power of the Atari CIO system. While the system may seem complex at first glance, its incredible versatility is a real testament to the programming know how of Atari's computer designers.

Moving Along

Let's continue on now with our "OPEN" operation. In lines 380 and 390, we load the number $08 the token for "open a device for a write operation" into Auxiliary Byte No. 1 of IOCB #2. We could make our program do something completely different if we stored the value $04, the token for "open read," in ICAX1,X instead of the value $08, the token for "open write." That's another demonstration of the versatility of the Atari CIO system.

We have now reached lines 400 and 410, in which we clear Auxiliary Byte No. 2 of IOCB #2 (a byte that is not used in this routine) by stuffing it with a zero. Finally, in line 420, we jump to the Atari CIO vector at memory address $E456. With that operation, we have opened IOCB #2 for a write operation to the Atari screen editor. In other words, we have opened IOCB #2 to print on the screen.

Printing a Character

We have not yet actually printed a character on the screen, however. To do that, we must carry out two more sequences of I/O operations. Now that you understand how the Atari CIO system works, that will be a snap. Here are lines 440 through 600 of the PRNTSC.SRC program.

```
430 ;
440   LDA #PUTCHR
450   STA ICCOM,X
460 ;
470   LDA #TXTBUF&255
480   STA ICBAL,X
490   LDA #TXTBUF/256
500   STA ICBAH,X
510   RTS
520 ;
530 PRNT
540   LDX #IOCB2
550   LDA #BUFLEN&255
```

```
560  STA ICBLL,X
570  LDA #BUFLEN/256
580  STA ICBLH,X
590  JSR CIOV
600  RTS
```

In lines 440 and 450, we store the number $0B, the token for a "put character" operation, into the command byte of IOCB #2. In lines 470 through 510, the address of the text buffer we have created especially for this program is stored in the buffer address bytes of IOCBC #2. That prepares us for the PRNT routine that starts at line 530. In the PRNT routine, which extends from line 530 to line 600, the length of our specially created text buffer is stored in the buffer length bytes of IOCB #2. Then there is another jump to the CIO vector, which automatically takes care of printing the text in the PRNTSC text buffer on your computer screen.

Closing a Device

When I presented the original version of this program in Chapter 7, I left out one very important routine, the routine for closing a device. There was no need for such a routine in Chapter 7, since the PRNTSC program was not presented as a program in its own right, but as an adjunct to two other programs that ended in infinite loops. Still, I must admit that it was a bad programming practice for me not to close the IOCB I was using when I was finished with it. When you open a device in assembly language (as in Atari BASIC), you must close it when you're finished with it. Otherwise, you'll cause an IOCB error, and that could cause some serious problems.

Forgetting to carry out such tasks as closing IOCBs (at the time they should be closed) can lead to program crashes and long and agonizing debugging sessions. Anyway, IOCB #2 is closed in this new and improved version of the PRNTSC program. In lines 610 through 660, the value $0C — the token for closing a file — is loaded into ICCOM,X. Then there's a jump to CIOV, and the Atari OS closes the IOCB.

That ends our brief glimpse into the intricacies of the central input/output system of Atari computers. But we have by no means exhausted that topic; much more information on Atari I/O is available in more advanced books than this one, and in technical reference manuals.

Chapter Thirteen

Atari Graphics

This chapter and the one that follows are the real payoffs in this book on Atari assembly language. In this chapter and the next, you'll learn how to use assembly language to:

- Custom design your own screen displays, intermixing text, graphics and colors in any way you choose.
- Scroll text and graphics on your computer screen.
- Create and use custom designed character sets.
- Use game controllers in assembly language programs.
- Use Atari player-missile graphics to create arcade style action on your computer screen.

To accomplish all of these things, you're going to have to write some fairly complex assembly language routines, but once you've typed and saved them, you'll find that there are many, many ways to use them. By the time you've finished this book, you'll discover (I hope) that you've become a pretty advanced assembly language programmer.

The first program in this chapter will enable you to create a title screen that you can use with any homemade program you like, and it will be a real eye catcher, too! It will display three different sizes of type on your computer screen, with each size of type displayed in a different color, against a background of still another color. And, as they say in those TV mail order ads, there's more. In the next (and last) chapter of this book, you'll learn how to use fine scrolling to animate your title screen. Then, as an extra bonus, you'll learn how to use game controllers and player-missile graphics in Atari assembly language programs. First, though, we'll have to take a brief look at some of the graphics related features of your Atari computer, and at the way it generates its screen display.

The Antic Chip

Atari computers use a much more sophisticated, and much more powerful technique for generating screen displays than most microcomputers do. In most microcomputers, you'll find just one block of RAM in which you can store data that is to appear on the computer's screen, in other words, one block of RAM that's dedicated to screen memory. Within that block of RAM, each letter or symbol that appears on the screen will be assigned one memory location. When the value of that memory location is changed, the text or graphics display in the screen location that corresponds to that memory location will also change. And that's about all you have to know to understand the graphics and text displays of most computer systems.

Atari graphics, as we just said, are more sophisticated than that and just a bit more complicated as well. Atari computers use two special chips, an ANTIC chip and a GTIA chip, to generate their graphics displays. One of these chips, the ANTIC, is a real microprocessor; it is designed to be used with a special instruction set, and a special kind of program called a *display list.* So, to create graphics using the ANTIC chip, you have to know how to use the instruction set to design display lists for your Atari. It also helps to have a rudimentary knowledge about how a television set works. So here goes:

Scan Lines

The picture on a television screen, as you may know, is made up of tiny horizontal lines — 262 lines, to be exact. And each of these horizontal lines is called a *scan line.*

As you may also know, these scan lines are produced by an electron gun behind your television set's picture tube. This electronic pistol fires electrons at the phosphor coating inside the TV picture tube in what is known as a *raster scan* pattern, a zigzag pattern that begins at the upper left-hand corner of the screen and ends in the bottom right-hand corner.

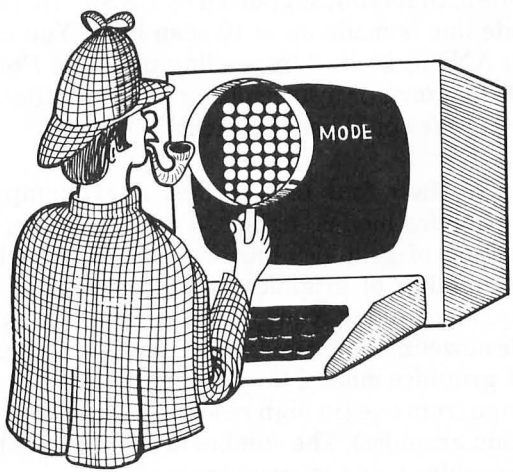
Since there are 262 horizontal scan lines on a video tube, the complete 262 line display on your TV screen is replaced by a com-

pletely new display 60 times each second. Between each of these lightning fast scenery changes, there is an extremely brief interval called a *vertical blank* period, in which the whole screen goes blank.

Because of a picture tube design technique called *overscan*, however, not all of the 262 scan lines that are available for a TV picture appear on the screen; some fall off the edges and are therefore never seen. So programs used to generate video displays for computers don't usually make use of all of those lines. Your Atari, for example, uses only 192 of the 262 scan lines that are available.

Dot Matrix Characters

If you look at a computer generated text display on a TV screen, you may also notice that each text character on the screen is made up of tiny dots. And if you could look closely enough at the text screen generated by your Atari while your computer is in its normal 40 column by 24 line text mode, you'd be able to see that each letter on the screen is made up of 64 dots, arranged in a matrix 8 dots wide and 8 dots high.



Mode Lines

Your Atari computer has four different text modes. Each of these modes produces letters of a different size. But no matter how big the letters on your screen are, each line of text in an Atari display is called a *mode line*. In your Atari's normal 40 column by 24 line text mode, the mode referred to in Atari BASIC as Graphics 0, each letter in a *mode line* is eight dots high, and each of those dots equates to one *scan line*. In BASIC's Graphics 0 mode, therefore, one *mode line* is equal to eight scan lines.

Atari BASIC supports two other text modes: the Graphics 1 mode, in which the characters on the screen are the same height as Graphics 0 characters but twice as wide, and the Graphics 2 mode, in which the characters are twice as high and twice as wide as standard Graphics 0 characters. When your computer is in its Graphics 1 mode, each mode line is made up of eight scan lines, the same number of scan lines used in a mode line in Graphics 0. When your Atari is in its Graphics 2 mode however, each mode line equals 16 scan lines.

Antic Mode

In assembly language, there is also another text mode, called ANTIC Mode 3, that is not supported by BASIC. In ANTIC Mode 3, each mode line is made up of 10 scan lines. You can find out more about ANTIC Mode 3 by reading the *Atari Programmer's Manual, De Re Atari,* or by consulting *The Atari 400/800 Technical Reference Notes* published by Atari.

In addition to their four text modes, Atari computers have numerous graphics modes; either 10 or 13 of them, depending upon what kind of graphics hardware came installed in your Atari. (The number of graphics modes offered by Atari computers vary, since older Ataris have a graphics chip called a CTIA, while newer models come with a new GTIA chip installed.) In non-text graphics modes, the number of scan lines per mode line can range from one (in high resolution graphics) to eight (in low resolution graphics). The number of colors available also differs from graphics mode to graphics mode.

In the Mode

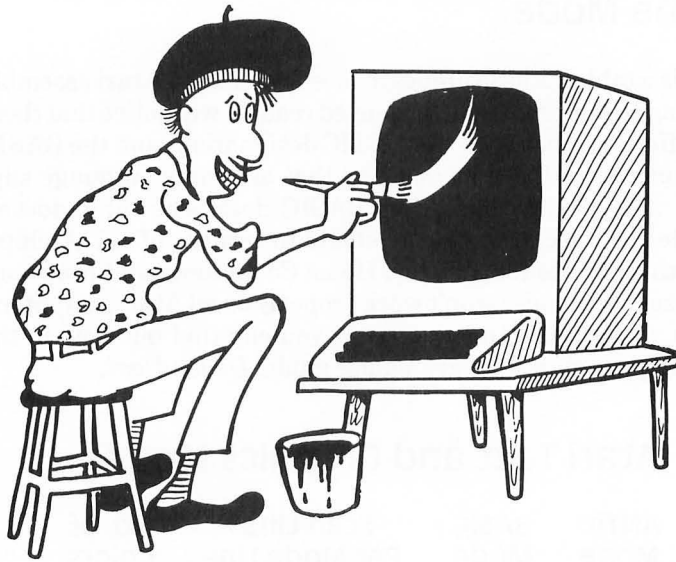
Here is a table of the graphics modes available to Atari assembly language programmers. Sharp-eyed readers will notice that there are differences between the ANTIC designations and the BASIC designations of these modes, and that assembly language supports more modes than Atari BASIC does. The table doesn't include the special modes available to owners of GTIA chips, since this book is for all Atari Home Computers, and programs that use those modes won't work properly on all Atari computers. If you want to use them anyway, you can find out how in the assembly language programming guide, *De Re Atari.*

Atari Text and Graphics Modes

ANTIC Mode	BASIC Mode	Scan Lines Per Mode Line	No. of Colors
2	0	8	2
3	None	10	2
4	None	8	4
5	None	16	4
6	1	8	5
7	2	16	5
8	3	8	4
9	4	4	2
A	5	4	4
B	6	2	2
C	None	1	2
D	7	2	4
E	None	1	4
F	8	1	2

Customizing Your Atari's Screen Display

Two steps are needed to custom design an Atari screen display. First you have to create a special kind of program called a *display list.* Then you have to write a program that will tell your computer how to use the display list you have designed. While doing



this be aware that a display list may *not* cross a 1K boundary and a screen display memory may not cross a 4K boundary without special handling.

In a moment, we'll talk about how to write a display list program. First, though, let's take a look at the display list that the ANTIC program will refer to. A display list is made up of a series of one-byte instructions that can be placed almost anywhere in your computer's available RAM. To get an idea of what a display list looks like, you can use your assembler's debugging utility to examine the display list that your computer uses when it's in its Graphics 0 text mode.

When you turn on your computer, it automatically goes into Graphics 0 mode, and the address of the display list which it uses to generate that mode is always stored in two locations: memory addresses $230 and $231. Memory address $230 always holds the low byte of the starting address of the display list that your computer is using, and memory address $231 always holds the high byte of the display list's starting address. So, once you know the contents of these two addresses, you'll be able to locate the dis-

play list that your computer is currently using. Once you locate your computer's Graphics 0 display list, you'll find that it looks something like this:

```
70 70 70 42 20 7C 02 02
02 02 02 02 02 02 02 02
02 02 02 02 02 02 02 02
02 02 02 02 02 41 E0 7B
```

As you can see, a display list is just that: a list, not a program. To use a display list, a separate program is needed. You'll get a chance to take a look at such a program in a moment. But first, let's examine this sample display list, byte by byte:

Bytes 1 - 3

$70 $70 $70

Each byte in a display list has a specific meaning to the Atari ANTIC chip. And within each byte, each nybble (that is each hexadecimal digit) also has a specific meaning. For example, each of the first three bytes — each of the three $70s at the beginning of the list — tells the ANTIC chip to display one blank mode line (in BASIC Graphics 0, eight blank scan lines). A standard Graphics 0 display always begins with three "skip mode line" instructions (in ANTIC language, three $70s) in order to overcome the overscan characteristics of TV tubes and make sure that the whole screen display called for in the display list winds up visible on the screen.

Bytes 4 - 6

$42 $20 $7C

The first actual display byte in our sample display list ($42) is what is known as a *Load Memory Scan (LMS)* command. The first display byte in a display list, that is, the first byte after all necessary blank lines have been taken care of, is always an LMS command. And a load memory scan command is always a three-byte instruction. In the display list we are now examining, the load memory scan instruction is made up of the three bytes: "$42

$20 $7C." The first nybble in this instruction, the digit 4, alerts ANTIC that what follows is going to be an LMS instruction. The second nybble in the LMS instruction, the digit 2, tells ANTIC to display an ANTIC Mode 2 line. Consult the table on graphics modes presented a few paragraphs back, and you'll see that in ANTIC language, Mode 2 is the same as BASIC Mode 0. The next two bytes of the LMS command, the bytes $20 $7C, provide ANTIC with the address at which screen memory will begin. ANTIC interprets these two bytes low byte first, in standard 6502 fashion. When ANTIC encounters the LMS instruction $42 $20 $7C, therefore, the first byte displayed on your Atari's video screen will be whatever byte is stored in memory address $7C20.

When you write a display list, you can put your screen memory in just about any convenient and available block of RAM. And you can fill that RAM up with whatever you like: ATASCII codes that equate to text, display screens drawn with the help of a graphics program, or character graphics created with a graphics generator program. Once you have a display created, address in the two bytes that follow your display list's LMS command. You'll have an opportunity to see how this technique works in the sample program at the end of this chapter.

Bytes 7 - 29

The byte $02, repeated 23 times

As explained above, the first LMS command in a display list tells ANTIC two things: the address at which screen memory begins, and the graphics mode to use to display the first mode line of text or data that will be found starting at that address. After ANTIC has been presented with this information, it must be told what graphics mode to use to display each subsequent mode line that will displayed on the screen. In the display list which we are now examining, every mode line on the screen is an ANTIC Mode 2 line. Therefore the next 23 instructions in this display list are *all* the same: Each tells ANTIC that the next line on the screen will be an ANTIC Mode 2 line.

What would happen, you may ask, if all of these 23 instructions were not the same? Well, if they were not the same, then more

than one graphics mode could be displayed on the screen simultaneously. Text of various sizes could be displayed on the same screen, and text and graphics modes could be intermixed as desired. This is a very powerful and quite unusual capability of Atari computers. You'll get a chance to see exactly how it works before we finish this chapter.

Bytes 30 - 32

$41 $E0 $7B

Every display list must end with a three-byte command called a JVB (Jump on Vertical Blank) instruction. The first byte in a JVB instruction is always the value $41. The next two bytes always combine to form a jump address. The destination of the jump is always the beginning of the display list in which the jump is contained. As it happens, the display list we're now looking at starts at memory address $7BE0. So that's the address that follows (low byte first) the JVB instruction $41. When ANTIC encounters the JVB instruction $41 in a display list, it jumps back to the beginning of the display list, waits for the next vertical blank period between raster scan displays, and then jumps to the address that follows the JVB instruction. Since this address is the address of the beginning of the display list, what the JVB instruction really does is generate the display list again.

Running a Display List

As we've pointed out, a display list can be placed in almost any convenient and available spot in your computer's memory. Screen memory can be placed just about anywhere in RAM, too. Once you've created a display list and a block of data to be used as screen memory, all you have to do to put your custom designed display on your TV screen is write a simple little assembly language program that tells your computer's operating system where your display list is. To direct your computer to your custom display list, all you have to do is store new values into a pair of OS memory locations known as "shadow" locations. Shadow addresses are used often in Atari programming, so I might as well explain right now what they are.

In your computer's memory, there are some very useful hardware registers that cannot normally be accessed by user-written programs. But sixty times per second, the data in each of these memory locations is updated. During this updating process, the value stored in each of these registers is replaced by data that has been stored in a corresponding *shadow register*. And shadow registers are in user-accessible RAM. So, by changing the value in a shadow register, you can also change the value of its corresponding hardware register. For most intents and purposes, therefore, a shadow register works just about like any other OS register that's situated in RAM. Three shadow addresses that are often used in display list programs are $22F, $230, and $231. Address $22F is an Atari OS memory location called SDMCTL (Shadow, Direct Memory Access Control). Addresses $230 and $231 are OS locations called SDLSTL (Shadow, Display List Pointer — Low) and SDLSTH (Shadow, Display List Pointer — High). To write a program that will put a custom display list on your Atari's screen, all you have to do is follow these three steps:

1. Turn your computer's ANTIC chip off by storing a zero in $22F (SDMCTL).

2. Store the starting address of your custom display list in $230 and $231 (SDLSTL and SDLSTH).

3. Turn your computer's ANTIC chip on again by storing the value $22 in $22F (SDMCTL).

Doing It

Now that you know how all of those things are done, we're ready to do them. Here is a customized display list, along with a program that will run it. If you wish, you can type it into your Atari computer, save it on a disk, and run it right now:

```
A CUSTOMIZED SCREEN DISPLAY

10 ;
20 ;HELLO SCREEN
30 ;
```

```
40  *=$3000
50  JMP INIT
60 ;
70 SDMCTL = $022F
80 ;
90 SDLSTL = $0230
100 SDLSTH = $0231
110 ;
120 COLOR0 = $02C4 ;OS COLOR REGISTERS
130 COLOR1 = $02C5
140 COLOR2 = $02C6
150 COLOR3 = $02C7
160 COLOR4 = $02C8
170 ;
180 ;DISPLAY LIST DATA
190 ;
200 START
210 ;
220 LINE1 .SBYTE "        PRESENTING        "
230 LINE2 .SBYTE "      the big program     "
240 LINE3 .SBYTE "                      By (You"
250   .SBYTE "r Name)                       "
260 LINE4 .SBYTE "    PLEASE STAND BY     "
270 ;
280 ;DISPLAY LIST
290 ;
300 HLIST
310   .BYTE $70,$70,$70 ;3 BLANK LINES
320   .BYTE $70,$70,$70,$70,$70 ;MORE
      BLANK LINES
330   .BYTE $46 ;LMS, ANTIC MODE 6 (BASIC
      MODE 2)
340   .WORD LINE1 ;(TEXT LINE:
      "PRESENTING . . . ")
350   .BYTE $70,$70,$70,$70,$47 ;LMS, ANTIC
      MODE 7
360   .WORD LINE2 ;(TEXT LINE: "THE BIG
      PROGRAM")
370   .BYTE $70,$42 ;(LMS, ANTIC MODE 2
      (GR. 0])
380   .WORD LINE3 ;(TEXT LINE: "By (Your Name]")
```

```
390   .BYTE $70,$70,$70,$70,$46 ;LMS, ANTIC
      MODE 6
400   .WORD LINE4 ;[TEXT LINE: "PLEASE
      STAND BY"]
410   .BYTE $70,$70,$70,$70,$70 ;5 BLANK
      LINES
420   .BYTE $41 ;JVB INSTRUCTION . . .
430   .WORD HLIST ;TO JUMP BACK TO START
      OF LIST
440 ;
450 ;RUN PROGRAM
460 ;
470 INIT ;SWITCHING COLOR REGISTERS FOR
    NICELY COLORED DISPLAY
480   LDA COLOR3
490   STA COLOR1
500   LDA COLOR4
510   STA COLOR2
520 ;NOW WE'LL RUN THE PROGRAM
530   LDA #0
540   STA SDMCTL ;TURN ANTIC OFF FOR A
      MOMENT . . .
550   LDA #HLIST&255 ;WHILE WE STORE OUR
      NEW LIST'S ADDRESS
560   STA SDLSTL ;IN THE OS DISPLAY LIST
      POINTER.
570   LDA #HLIST/256 ;NOW FOR THE HIGH
      BYTE.
580   STA SDLSTH ;NOW ANTIC WILL KNOW
      OUR NEW LIST'S ADDRESS
590   LDA #$22
600   STA SDMCTL ; . . . SO WE'LL TURN ANTIC
      BACK ON NOW
610 ;
620 FINI
630   RTS
```

How it Works

We've already covered just about everything in this program, so it's probably fairly self-explanatory. However, if you're using an Atari Assembler cartridge or an Atari Macro Assembler and Text Editor package, you'll have one problem with the program: it won't work! That's because Atari's two assemblers, as sophisticated as they may be, they are not equipped with one handy little directive that the MAC/65 assembler does have, the .SBYTE directive used in lines 220 through 260 of of our display list program.

The MAC/65's .SBYTE directive was designed to convert ATASCII code, which is what your computer uses to store text in its memory, to a completely different code that's used to display characters on the screen. This latter code is called, appropriately enough, *screen code*. It wouldn't be difficult to convert ATASCII to screen code if there were a direct one-to-one correlation between the two codes. Unfortunately, there is no such one-to-one relationship. To translate from ATASCII code to screen code, you have to add 64 to some character codes, subtract 32 from others, and leave still others alone. Here's an ATASCII to screen code conversion table that shows just what the translation process involves:

CONVERTING ATASCII CODE TO SCREEN CODE

ATASCII VALUES	OPERATION NEEDED FOR CONVERSION
0 to 31	Add 64
32 to 95	Subtract 32
96 to 127	None
128 to 159	Add 64
160 to 223	Subtract 32
224 to 255	None

An Automatic Conversion Routine

If your assembler doesn't have an .SBYTE function, there is an assembly language routine you can use to make the above conversions. I wrote it before there was any such thing as a MAC/65 assembler or an .SBYTE directive. It's a "quick and dirty" routine that was dreamed up in a hurry, and if you're a good assembly language programmer, you can probably write a routine that will do the same job faster, more efficiently, or both. But this one works just fine, and it will work with a block of text of any size. This little conversion subroutine simply performs the calculations in the above table automatically. Before you can use it, however, you'll have to make a few minor alterations in the main display list program you typed a few moments ago. Here's what you'll have to do:

Three Modifications

- Convert the .SBYTE directives in lines 220 to 260 to .BYTE directives (or to DB directives, if you're using an Atari Macro Assembler).

- Add one variable and one constant to the symbol table in your display list program. The variable, which I'll call TEMPTR (for "temporary pointer)," will have to be located on page zero, since it will be used with indirect indexed addressing, an addressing mode that demands two zero page locations. The constant you'll be using, called EOF, will have the literal value $88, which is the ATASCII code for an end of file character. You can add these symbols to the symbol table in your title screen program with these lines:

```
65 TEMPTR = $CC
66 EOF = $88
67 ;
```

- Add this line to your program to mark the end of the block of text to be converted to screen code:

```
265  .BYTE EOF
```

Now Let's Get Started

The conversion routine starts off by storing the starting address of the data for our new display list into a pair of pointers called TEMPTR and TEMPTR+1. Then, using indirect indexed addressing, it moves through the text to be converted character by character. Before it performs each conversion, however, it checks to see if the character in question is an end of file character ($88). If the character in question is an EOF character, the subroutine ends, since that means that all necessary conversions have been performed and that it's time to end the conversion routine. If the character is not an EOF character, the program goes ahead and performs the necessary conversion (if one is needed). Then it moves on to the next character.

Faking the .SBYTE Directive

If your assembler isn't equipped with an .SBYTE directive, or if, for some reason, you don't want to use the .SBYTE directive that the MAC/65 assembler provides, then you can use this conversion routine. Just type it into RAM, and then jump to it after your display list is loaded into memory but before the code that initilizes your display list begins. You can do that with this line:

```
475  JSR FIX
```

Here's the conversion program:

Note: Before assembling this program type SIZE *to verify that MEMLO (the second number) is less than the starting address of the object code in line 40. If it isn't, you should either change the starting address (e.g., 40*=5000), change LOMEM (e.g., LOMEM 4000), or remove comments from the source code so that MEMLO is less than the starting address.*

AN ATASCII-ASCII CONVERSION SUBROUTINE

```
2000 ;
2010 ; FIX DATA
2020 ;
```

```
2030 FIX
2040   LDA #START&255 ;STARTING ADDRESS
       OF NEW DISPLAY LIST (LOW BYTE)
2050   STA TEMPTR
2060   LDA #START/256 ;NEW DISPLAY LIST
       ADDRESS (HIGH BYTE)
2070   STA TEMPTR+1
2080 ;
2090 ;"FIX DATA" SUBROUTINE
2100 ;
2110   LDY #0;LOAD Y REGISTER WITH DUMMY
       0 FOR INDIRECT INDEXED ADDRESSING
2120 FXDT
2130   LDA (TEMPTR),Y ;START WITH FIRST
       CHARACTER IN BLOCK
2140   CMP #EOF ;IS IT AN END OF FILE
       CHARACTER ($88)?
2150   BEQ DONE ;IF SO, WE'RE DONE--EXIT
       SUBROUTINE
2160   JSR FXCH ;ELSE JUMP TO "FIX
       CHARACTER" SUBROUTINE
2170   STA (TEMPTR), Y ;THEN REPLACE OLD
       CHARACTER WITH NEW ONE
2180   INC TEMPTR ; . . . AND INCREMENT
       TEMPTR (LOW BYTE)
2190   BNE FXDT ;IF NO CARRY TO HIGH BYTE,
       START AGAIN
2200   INC TEMPTR+1 ;ELSE INCREMENT
       TEMPTR'S HIGH BYTE
2210   JMP FXDT ;. . . AND THEN GO BACK TO
       FXDT AND START AGAIN
2220 ;
2230 DONE ; MAIN SUBROUTINE DONE--ALL
     CHARACTERS CONVERTED
2240   RTS ;SO NOW WE RETURN TO THE
       PROGRAM IN PROGRESS
2250 ;
2260 ;"FIX CHARACTER" ROUTINE
2270 ;
2280 FXCH
2290   CMP #32 ;IS CHARACTER CODE < 32?
```

```
2300  BCC ADD64 ;IF SO, JUMP TO "ADD64"
      ROUTINE
2310 ;
2320  CMP #96 ;IS IT < 96?
2330  BCC SUB32 ;IF SO, JUMP TO "SUBTRACT
      32" ROUTINE
2340 ;
2350  CMP #128 ; < 128?
2360  BCC FIXT ;IF SO, JUMP TO "FIXT"
      ROUTINE (NO ACTION)
2370 ;
2380  CMP #160 ; < 160?
2390  BCC ADD64 ;IF SO, JUMP TO ADD64
2400 ;
2410  CMP #224 ; < 224?
2420  BCC SUB32 ;IF SO, JUMP TO SUB32
2430 ;
2440  JMP FIXT ;IF BETWEEN 224 AND 255,
      NO ACTION NEEDED
2450 ;
2460 ADD64
2470  CLC ;CLEAR CARRY TO ADD
2480  ADC #64 ;THEN ADD 64
2490  JMP FIXT ;RETURN TO MAIN (FXDT)
      SUBROUTINE
2500 ;
2510 SUB32
2520  SEC ;SET CARRY FOR SUBTRACTION
2530  SBC #32 ;AND SUBTRACT 32
2540 FIXT
2550  RTS ;RETURN TO MAIN (FXDT)
      SUBROUTINE
```

Coarse Scrolling

As soon as you've typed this program and saved it on a disk, you can run it, and if you've typed it exactly as written, you should have no problems. As soon as you have it up and running, you can start fixing it up so that it will be even fancier, with a technique known as scrolling. Before I start discussing scrolling, though, I'd like to suggest that you make one more small modification in

the display list program that we've been working on. We'll be scrolling just one line in the program — the line that says "THE BIG PROGRAM." And we'll need to insert some blank spaces before and after that line so that it will scroll across the screen properly. With these spaces inserted, the screen will start off with a blank space where the words "THE BIG PROGRAM" should be. Then these three words will come scrolling across the screen, like those ticker tape style signs you sometimes see in store windows and on TV. To provide this new spacing, you'll have to change one line of your original display list program, and then add two more lines. Here are the three lines we'll need (230 is the amended line, and 225 and 235 are the new ones):

```
225 LINE2 .SBYTE "                              "
230   .SBYTE "      the big program             "
235   .SBYTE "                                  "
```

Once these lines are modified, it isn't difficult to implement a primitive sort of scrolling (called coarse scrolling) in our display list program. To implement a coarse scroll, all you have to do is set up a loop that keeps incrementing or decrementing certain addresses in a program — specifically, the addresses that follow the LMS instructions in a display listing. If your scrolling program is written in assembly language, it will also have to include some sort of delay loop, since machine language is so fast that without some sort of delay, it will cause a scroll line to zoom by so rapidly that it turns into a blur.

Here is a display list and an accompanying display list implementation routine will add coarse scrolling to your title screen program. Make the following changes in the program, and the line that reads "THE BIG PROGRAM" will scroll across the screen over and over again in an endless loop. The text will move in a very jerky manner, one full letter at a time. But don't worry; in the next chapter, we'll smooth out that action with an assembly language technique known as fine scrolling.

AN EXAMPLE OF COARSE SCROLLING

```
10 ;
20 ;HELLO SCREEN (COARSE)
```

```
30 ;
40  *=$3000
50  JMP INIT
60 ;
70 TCKPTR = $2000
80 ;
90 SDMCTL = $022F
100 ;
110 SDLSTL=$0230
120 SDLSTH=$0231
130 ;
140 COLOR0 = $02C4 ;OS COLOR REGISTERS
150 COLOR1=$02C5
160 COLOR2=$02C6
170 COLOR3=$02C7
180 COLOR4=$02C8
190 ;
200 ;DISPLAY LIST DATA
210 ;
220 START
230 LINE1 .SBYTE "          PRESENTING          "
240 LINE2 .SBYTE "                              "
250  .SBYTE"      the big program          "
260  .SBYTE "                               "
270 LINE3 .SBYTE "                      By   (You"
280  .SBYTE "r Name)                        "
290 LINE4 .SBYTE "     PLEASE STAND BY      "
300 ;
310 ;HELLO DISPLAY LIST
320 ;
330 HLIST
340  .BYTE $70,$70,$70
350  .BYTE $70,$70,$70,$70,$70
360  .BYTE $46
370  .WORD LINE1
380  .BYTE $70,$70,$70,$70,$47
390 SCROLN
400  .WORD $00
410  .BYTE $70,$42
420  .WORD LINE3
430  .BYTE $70,$70,$70,$70,$46
```

```
440  .WORD LINE4
450  .BYTE $70,$70,$70,$70,$70
460  .BYTE $41
470  .WORD HLIST
480 ;
490 ;RUN PROGRAM
500 ;
510 INIT
520   LDA COLOR3
530   STA COLOR1
540   LDA COLOR4
550   STA COLOR2
560 ;
570   LDA #0
580   STA SDMCTL
590   LDA #HLIST&255
600   STA SDLSTL
610   LDA #HLIST/256
620   STA SDLSTH
630   LDA #$22
640   STA SDMCTL
650 ;
660 ;COARSE SCROLLING ROUTINE
670 ;
680   LDA #40
690   STA TCKPTR
700   JSR TCKSET
710 ;
720 COARSE
730   LDY TCKPTR ;40 TO START
740   DEY
750   BNE SCORSE
760   LDY #40
770   JSR TCKSET
780 SCORSE
790   STY TCKPTR
800   INC SCROLN
810   BNE LEAP
820   INC SCROLN+1
830 ;
```

```
840 ;DELAY LOOP
850 ;
860 LEAP
870   TYA
880   PHA ;SAVE Y REGISTER
890   LDX #$FF
900 XLOOP
910   LDY #$80
920 YLOOP
930   DEY
940   BNE YLOOP
950 ;
960   DEX
970   BNE XLOOP
980   PLA
990   TAY ;RESTORE Y REG
1000 ;
1010   JMP COARSE
1020 ;
1030 TCKSET
1040   LDA #LINE2&255
1050   STA SCROLN
1060   LDA #LINE2/256
1070   STA SCROLN+1
1080 ENDIT
1090   RTS
```

If you type the above modifications into your display list program and run it, what you'll see is an example of coarse scrolling. The line that reads "THE BIG PROGRAM" will come jumping across the screen, a letter at a time, in a jerky kind of way that could get rather disconcerting if you had to look at it for very long. Coarse scrolling is bad enough when it's used to move just one line across a screen. But it becomes even more nerve shattering when it's used to scroll an entire screen display.

To scroll more than one line on your screen, up to and including the maximum number of lines in a full screen display, all you have to do is go to your display list and insert an LMS instruction before every line you want scrolled. If you want to scroll your entire screen, you can precede *every* line with an LMS instruction! Then, to scroll your screen horizontally, all you'll have to do is set up a loop that progressively increments or decrements the starting address of the data that appears on every line. If you increment those addresses, your display will scroll from right to left. If you decrement them, it will scroll from left to right.

It's just as easy to do coarse *vertical* scrolling as it is to do coarse horizontal scrolling. To scroll a screen display vertically, all you have to do is increment or decrement the LMS address of each line *by the number of characters in the lines being scrolled*, instead of by just one character at a time. When you set up this kind of scrolling action, you have to count the number of characters in each line very carefully, so your characters won't move back and forth on the screen as they scroll up or down. Coarse vertical scrolling, like coarse horizontal scrolling, can be used to scroll

any number of lines on a monitor screen. Whether you're aware of it or not, in fact you've probably enountered coarse vertical scrolling in action many times. It's the kind scrolling you see when you're writing a BASIC or assembly language program, reach the bottom line on your monitor screen, and hit a carriage return. When you do that, your entire screen display moves up one line, using a coarse vertical scrolling routine.

Coarse scrolling is fine when it's used that way, just one line at a time, but it doesn't work very well in more demanding applications, such as moving screen displays around in arcade style games. Unfortunately for BASIC programmers, coarse scrolling is the only kind that Atari BASIC supports. To do smooth scrolling, you have to use — you guessed it! — assembly language. In the next (and last) chapter of this book, you'll learn how to use smooth scrolling in assembly language programs. And, as if that weren't enough, you'll also learn how to create and animate character sets, how to use player-missile graphics, and how to write assembly language programs that call for the use of game controllers.

Chapter Fourteen

Advanced Atari Graphics

No matter how good a BASIC programmer you may be, there are some things that you just can't do in a BASIC program. BASIC is simply not fast enough to handle such tasks as fine scrolling, high-speed character animation, and player-missile graphics. Assembly language can handle all three of these tasks quite easily, and in this chapter, you'll see exactly how. Let's start with fine scrolling.

Fine Scrolling

To demonstrate fine scrolling, I'm going to use an expanded version of the title screen program you typed into your computer in the preceding chapter. If you've saved that program on a disk, you can load it into your computer right now. Then, with a little editing and a few additions, you can modify it until it looks like the listing below. After you've modified it, you can save it on a disk, run it, and take a look at it in action. Then I'll explain how it works, and you'll have an eye-catching assembly language program that you can use from now on to help you create customized title screens for your own programs.

A DEMONSTRATION OF FINE SCROLLING

```
10  ;
20  ;HELLO SCREEN (FINE)
30  ;
40    *= $3000
50    JMP INIT
60  ;
70  TCKPTR = $2000
80  FSCPTR = TCKPTR+1
90  ;
100 SDMCTL = $022F
110 ;
120 SDLSTL = $0230
130 SDLSTH = $0231
140 ;
150 COLOR0 = $02C4 ;OS COLOR REGISTERS
160 COLOR1 = $02C5
170 COLOR2 = $02C6
180 COLOR3 = $02C7
190 COLOR4 = $02C8
200 ;
210 HSCROL = $D404
220 ;
230 VVBLKI = $0222 ;OS INTERRUPT VECTOR
240 SYSVBV = $E45F ;INTERRUPT ENABLE
      VECTOR
250 ;
```

```
260  SETVBV = $E45C ;SET VERTICAL BLANK
     INTERRUPT (VBI) VECTOR
270  XITVBV = $E462 ;EXIT VBI VECTOR
280  ;
290  ;DISPLAY LIST DATA
300  ;
310  START
320  LINE1 .SBYTE "      PRESENTING      "
330  LINE2 .SBYTE "                      "
340    .SBYTE "    the big program    "
350    .SBYTE "                       "
360  LINE3 .SBYTE "                 By  (You"
370    .SBYTE "r Name)                 "
380  LINE4 .SBYTE " PLEASE STAND BY    "
390  ;
400  ;DISPLAY LIST WITH SCROLLING LINE
410  ;
420  HLIST ; ('HELLO' LIST)
430    .BYTE $70,$70,$70
440    .BYTE $70,$70,$70,$70,$70
450    .BYTE $46
460    .WORD LINE1
470  ;NOTE THAT THE LAST BYTE IN THE
480  ;NEXT LINE IS $57, NOT $47 AS IT
490  ;WAS IN THE PRECEDING CHAPTER
500    .BYTE $70,$70,$70,$70,$57
510  SCROLN ;(THIS IS THE LINE WE'LL SCROLL)
520    .WORD $00 ;A BLANK TO BE FILLED IN
       LATER
530    .BYTE $70,$42
540    .WORD LINE3
550    .BYTE $70,$70,$70,$70,$46
560    .WORD LINE4
570    .BYTE $70,$70,$70,$70,$70
580    .BYTE $41
590    .WORD HLIST
600  ;
610  ;RUN PROGRAM
620  ;
630  INIT ; PREPARE TO RUN PROGRAM
640    LDA COLOR3 ;SET COLOR REGISTERS
```

```
650   STA COLOR1
660   LDA COLOR4
670   STA COLOR2
680 ;
690   LDA #0
700   STA SDMCTL
710   LDA #HLIST&255
720   STA SDLSTL
730   LDA #HLIST/256
740   STA SDLSTH
750   LDA #$22
760   STA SDMCTL
770 ;
780   JSR TCKSET ; INITIALIZE TICKER ADDRESS
790 ;
800   LDA #40 ;NUMBER OF CHARACTERS IN
      SCROLL LINE
810   STA TCKPTR
820   LDA #8
830   STA FSCPTR ;NUMBER OF COLOR CLOCKS
      TO FINE SCROLL
840 ;
850 ;ENABLE INTERRUPT
860 ;
870   LDY #TCKINT&255
880   LDX #TCKINT/256
890   LDA #6
900   JSR SETVBV
910 ;
920 ; TICKER INTERRUPT
930 ;
940 TCKINT
950   LDA #SCROLL&255
960   STA VVBLKI
970   LDA #SCROLL/256
980   STA VVBLKI+1
990 ;
1000 INFIN
1010   JMP INFIN ;INFINITE LOOP
1020 ;
1030 SCROLL
```

```
1040   LDX FSCPTR ;8 TO START
1050   DEX
1060   STX HSCROL
1070   BNE CONT
1080   LDX #8
1090 CONT ; (CONTINUE)
1100   STX FSCPTR
1110   CPX #7
1120   BEQ COARSE
1130   JMP SYSVBV
1140 COARSE
1150   LDY TCKPTR ;NUMBER OF CHARACTERS
       TO SCROLL
1160   DEY
1170   BNE SCORSE ;LOOP BACK TILL FULL LINE
       IS SCROLLED
1180   LDY #40
1190   JSR TCKSET ;RESET TICKER LINE
1200 SCORSE ; DO COARSE SCROLL
1210   STY TCKPTR
1220   INC SCROLN ;LOW BYTE OF ADDRESS
1230   BNE RETURN
1240   INC SCROLN+1 ;HIGH BYTE OF ADDRESS
1250 RETURN
1260   JMP SYSVBV
1270 ;
1280 TCKSET
1290   LDA #LINE2&255
1300   STA SCROLN
1310   LDA #LINE2/256
1320   STA SCROLN+1
1330 ENDIT
1340   RTS
```

As soon as you've typed this program, be sure save it on a disk immediately. Then you can run it, debug it if necessary, and save it again. Once you have it up and running properly, I think you'll agree with me that it's quite a nice display, and it certainly is an example of very smooth scrolling! Now for an explanation of how the program works.

How to Implement Fine Scrolling

The reason fine scrolling works so smoothly is that it has eight times the resolution of coarse scrolling. When coarse scrolling is employed in a program, it causes lines of text to jump across (or up or down) the screen one full character at a time. But when fine scrolling is used, text can be moved around the screen one-eighth of a character at a time. Here's how that works: Look closely at a text character on your video screen, and you'll see that it's made up of a matrix of tiny dots. If you use a magnifying glass, you will be able to see that there are exactly 64 dots in each character. Every character on your screen is eight rows of dots (or *scan lines*) high, and eight rows of dots (or *color clocks*) across. And these rows of dots, scan lines and color clocks, are the increments used in fine scrolling.

To create and implement a fine scrolling routine, several steps are required. First, you must go to your display list and *enable* fine scrolling by setting certain bits in the LMS instruction that appears before every line you want to scroll. When bit 4 of an LMS instruction is set, the line that follows the LMS instruction can be scrolled horizontally. When bit 5 of an LMS instruction is set, the line that follows the instruction can be scrolled vertically. If both bit 4 and bit 5 of an LMS instruction are set, then the line that follows the instruction can be scrolled both horizontally and vertically.

Take a look at lines 500 through 520 in the program you just typed, and you'll see that the LMS instruction preceding the line which I've labeled SCROLN (the line that scrolls) is $57. Look at the program in its previous incarnation, the version in Chapter 13, in which fine scrolling was not enabled. You will see that this LMS instruction has been changed from $47 to $57.

The number $47, expressed in binary notation, is 0100 0111. As any assembly language programmer can plainly see, bit 4 of that binary number (the fifth bit from the right, since the first bit of a binary number is bit 0), is a zero. In other words, bit 0 is not set. When we set bit 4, the number we're looking at becomes 0101

0111, or $57. Therefore, when horizontal fine scrolling of the line labeled SCROLN is enabled, lines 500 through 520 of our program become:

```
500   .BYTE $70,$70,$70,$70,$57
510 SCROLN ;(THIS IS THE LINE WE'LL SCROLL)
520   .WORD $00 ;A BLANK TO BE FILLED IN
        LATER
```

Now suppose you wanted to scroll SCROLN vertically instead of horizontally. What would you do? Well, you'd simply set bit 5 of the LMS instruction in line 500. Then the $57 that you see in that line would become $67 or, in binary notation, 0110 0111. If you wanted to enable both horizontal and vertical scrolling of the line, you'd simply change the LMS instruction $77 (0111 0111).

Fine scrolling, just like coarse scrolling, can be performed on any number of lines of text on your screen. Just set the proper bit (or bits) in the proper LMS instruction (or instructions), and the desired type of scrolling can be implemented for each selected line. But, you may ask, what if there is no LMS instruction for a line you want to scroll? Well, in that case, you could simply write one. There's absolutely no reason that a display list can't have an LMS instruction for every line on the screen. If you want to scroll an entire screen, you *must*, in fact, put an LMS instruction in front of every line. So far, all we've talked about is how to enable fine scrolling. But now that you know how to enable it, how do you actually do it?

Good question.

When fine scrolling of a line is enabled, control of the line is handed over to one of two *scrolling registers* that reside in your Atari's operating system. If you have authorized a horizontal scroll on a given line of a display, then that line becomes subject to control of a *horizontal scroll register*, which is abbreviated HSCROL and is situated at memory address $D404. When a vertical scroll has been enabled for a given given display list line, then that line becomes subject to the control of a *vertical scroll register*, or VSCROL, situated at address $D405. If both horizontal and vertical scrolling of a given line are enabled, then that line becomes

subject to the control of both the HSCROL and the VSCROL registers. Once control of a line has been turned over to HSCROL, VSCROL or both, then you can implement a fine scroll by simply loading a value into the appropriate scrolling register (or registers). When you load a number into the HSCROL register, every display list line that has been put under the control of that register will be shifted to the right by the number of color clocks loaded into HSCROL. Load a number into the VSCROL register, and every line for which a vertical scroll has been enabled will be scrolled upward by the number of scan lines you have specified.

Combining Fine Scrolling and Coarse Scrolling

There is one hitch, though. The scrolling registers in your computer are 8-bit registers, and only four of these 8-bits in each register are ever used. That means that fine scrolling can be taken only so far. To work properly, fine scrolling must be combined with coarse scrolling, which can handle as much scrolling data as you can program. Generally speaking, the best way to combine fine scrolling with coarse scrolling is to fine scroll a line or column of characters by seven color clocks or scan lines, and then to reset the appropriate fine scrolling register to its initial value and implement one coarse scroll. Loop through this kind of procedure over and over, and the result will be a smooth fine scroll. You can see how this procedure works by studying and experimenting with the fine scrolling routine in the title screen program we've been examining.

Smoothing Out Your Scrolling Action

That's about all there is to fine scrolling if you don't mind putting up with a jerk, a jump or a smear every now and then on your video display. If those kinds of messy situations don't appeal to you, and I'm sure they don't, then we might as well go ahead and talk about how to make a fine scrolling operation perfect: smooth, smear proof and jerk free.

As you may recall from the preceding chapter, the display on your computer monitor is redrawn by an electron gun 60 times

every second, and between each screen refresh there's a split second total screen blackout that takes place too rapidly for you to see. Well, when you write a fine scrolling routine in assembly language and don't take a few special precautions, your display may (in fact it virtually always will) smear a bit and jump around a little from time to time. That's because some of the scrolling action you've programmed will sometimes take place while a display is being drawn on your screen by the electron gun inside your video tube. But there is a way to keep that from happening. The folks who designed your Atari have provided you with something called a Vertical Blank Interrupt (VBI) vector, and once you learn how to use that vector, you can perform all kinds of graphics tricks on your computer screen, in real time and without any danger whatsoever of messing up your computer's screen display.

A vector, as you may know, is a pointer in your computer's operating system that contains the address of a specific routine. The primary purpose of a vector is to give you an easy method for implemeting an often used routine. When you jump to an OS vector during the course of a program, your program will automatically jump to the OS routine that the vector points to, and you can thus implement that routine without having to write from scratch all of the code that it contains. Vectors can sometimes be used in another way, too. Sometimes you can "steal" a vector; that is, you can change its value so that it will point to some routine you've written yourself, rather than the OS routine that it originally pointed to. That means that you can sometimes use a vector as an easy method for controlling the behavior of your computer's operating system. And that brings us the the point at hand: vertical blank interrupt vectors.

Actually, there are two VBI vectors in your computer, and each one has a corresponding pointer in your Atari's operating system. One of these vectors is called VVBLKI ("I" for "Immediate"), and the pointer that can be used to access it is situated at memory address $0222. The other VBI pointer is labeled VVBLKD ("D" for Deferred) and resides at memory address $0224.

Every time your Atari starts a vertical blank interrupt, it takes a look at the contents of the VVBLKI pointer. If the program that is

being processed does not make use of the VVBLKI pointer, then that pointer will contain nothing but an instruction to jump to a predetermined memory address: specifically, memory address $E45F (which Atari has labeled SYSVBV). Memory address $E45F usually doesn't contain anything exciting, either. All that it ordinarily contains is an instruction for your computer to continue its normal processing. By stealing the VVBLKI vector, however, you can make it point to any routine you like — usually one you yourself have written. Then, 60 times every second (every time your computer begins its 60 Hertz vertical blank interrupt) it will automatically process the routine whose address you have stored in the VVBLKI pointer. When your routine is finished, your computer will resume its normal processing.

Your computer's other VBI vector, VVBLKD, also points directly to an exit point unless it has been stolen for a software application. The VVBLKD vector's normal exit point is memory address $E462, which Atari calls XITVBV, it works just like SYSVBV. It merely terminates your computer's vertical blank interrupt period and allows your computer to resume normal processing. Once you understand how the VVBLKI and VVBLKD interrupts work, it isn't difficult to steal them. Here's all you have to do:

- Write a routine that you would like to see take place during a vertical blank interrupt.
- Make sure that your routine ends with a jump to SYSVBV or XITVBV (depending upon whether the vector you use is immediate or deferred).
- Store the address of your routine at VVBLKI for an immediate interrupt, or at VVBLKD for a deferred interrupt.

Once those steps are taken, your computer will process your new routine 60 times every second, just before it begins each VBI interrupt if you've used the immediate vector, or just before it returns from each VBI interrupt if you've used the deferred vector. That makes vector stealing a very valuable technique for writing programs involving high-speed, high-performance processing, such as the processing of routines involving graphics and sound.

At this point, you may be wondering why there are two vectors that you can steal, and what the differences between them are.

Well, the reason is simply that certain user-written routines are best performed at the start of a vertical blank period, while others should not be performed until a VBI has ended. More information on this point can be found in *De Re Atari* and in *The Atari 400/800 Technical Reference Notes.*

One More Thing . . .

There's just one more important fact that you should remember about VBI vector stealing. After you've stolen a vector, there's a small chance that an interrupt will begin after the first byte of the pointer you're using has been updated, but before the second byte has been changed. If that happens, it could crash your program. But this possibility can be easily avoided. All you have to do is use an operating system routine that the good people who designed your computer thoughtfully provided. This routine is called SETVBV, and it begins at memory address $E45C.

Here's how to use the SETVBV routine: First, load the 6502 Y register with the low byte of the address of the routine that instructs your computer to begin a vector changing routine. Then load the X register with the high byte of the address. Next, load the accumulator with a 6 for an immediate VBI or a 7 for a deferred VBI. Then do a JSR SETVBV, and your interrupt will be safely enabled. That's just about the whole story on how to use vertical blank in assembly language fine scrolling programs.

Customizing a Character Set

Your Atari computer has a very fine built-in character set. If you have a late model Atari, it may even have two sets of characters built into its ROM. But how would you like to be able to create your own character sets, including not only letters, numbers and special text symbols, but graphics characters, too?

Well, you can do that quite easily if you know assembly language. You can do it at lightning speed, too — not at the snail's pace you may have agonized over if you've ever tried to alter a character set using a BASIC program. It's actually quite easy to design a character set on an Atari computer. As I pointed out earlier in

this chapter, each character that your computer prints on your video monitor is made up of an 8 by 8 matrix of dots. This 8 by 8 grid is stored in your computer's RAM as eight bytes of data. The letter A, for example, is stored in your computer's memory as a string of binary digits that could be represented in this fashion:

Binary Notation		Hexadecimal Notation	Appearance
0000	0000	00	
0001	1000	18	XX
0011	1100	3C	XXXX
0110	0110	66	XX XX
0110	0110	66	XX XX
0111	1110	7E	XXXXXX
0110	0110	66	XX XX
0000	0000	00	

The primary character set in your computer is composed of 128 letters, each made up of 64 dots that could be arranged in the same 8 by 8 format as the letter "A." In fact, that is precisely the format in which the letters are arranged when they're displayed on your computer screen. Since there 128 characters in a set, and since each character is made up of eight bytes of data, a full character set occupies 1,024 bytes of RAM. Put all of that data together, and you have quite a lengthy table. You also have a table which must start on a 1K boundary because of your computer's architecture. Character sets can be stored almost anywhere in an Atari computer's memory, but the address of the character set currently in use is always stored in a specific pointer in the computer's operating system. That pointer, labeled CHBAS by the engineers who designed your Atari, is situated at memory address $2F4.

To locate a character in your computer, you only have to know two things: the current value of CHBAS, and the ATASCII code for the character you're seeking. Add the character's ATASCII code number to the value of CHBAS, and that will be the memory address of the character you're looking for. If your computer is an Atari 400 or an Atari 800, then its character set is built into ROM and thus, cannot be modified. If you have a later model Atari, your character set has a RAM address and can therefore be accessed somewhat more easily. But, if you want to alter your computer's built-in character set, you have to do it in a rather indirect way, no matter what kind of Atari you own.

The best way to modify a character set is to copy your computer's built-in character to some free and easily accessed block of RAM. You can then modify the contents of CHBAS so that it points to the starting address of your own block of characters rather than your computer's built-in character set. Then you can use either set of characters you like, either the one built into your Atari at the factory or the one you have created on your own.

Once you've defined a new character set and stored it in RAM, all you have to do to change your screen display from one character set to another is change the contents of CHBAS. That makes it easy to write routines calling for character animation. All you have to do to animate a character set is draw several different

character sets that vary slightly, and then switch back and forth among them by simply changing the contents of the CHBAS pointer. BASIC is too slow to handle a job like that very well, but when you know assembly language, you can animate character sets in real time, at lightning fast speeds. In a moment, I'll show a program that you can use to custom design your own character sets. The program has two distinct parts. The first part copies the entire Atari character table to a spot in memory selected by the programmer. The second part alters just one character — the character "A." But the same technique can be used to alter any other character or, if you wish, all of them!

I'd like to make two more observations before you type the next program. First, I'd like to point out that the first half of the program, the part that moves the character set, can be used to move any block of data from any location to any other location in your computer's RAM. That's a useful utility, since data often has to be moved from one block of memory to another in assembly language programs. The data moving portion of the program that follows is a particularly good one, since it it is designed to move blocks of memory a full page at a time, using 8-bit pointers instead of 16-bit pointers and thus saving a considerable amount of processing time. The second point I'd like to make is that there are ways to create character sets without having to go through the drudgery of drawing character sets on graph paper and then punching them into memory a byte at a time. There are several excellent programs on the market that can help you create your own character sets for Atari computers right on the screen, using a cursor, a set of menus, and keyboard commands. So if you're interested in computer graphics, it might be to your advantage to take a look at some professionally produced character generator programs.

But that's enough from me. Here's your program:

MOVING AND MODIFYING A CHARACTER SET

```
10 ;
20 ;ALTERING GRAPHICS CHARACTERS
30 ;
50  *=$0600
```

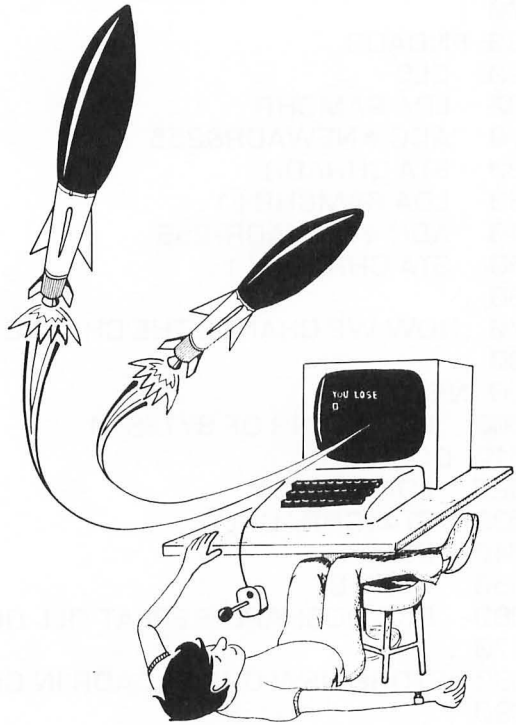
```
60   JMP MOVDAT
70  ;
80  CHBAS=$02F4
90  NEWADR=$5000
100  TABLEN=1024
110  ;
120  MVSRCE=$B0 ;PAGE ZERO PTR
130  MVDEST=MVSRCE+2 ;DITTO
140  CHRADR=MVDEST+2 ;DITTO
150  ;
160  LENPTR=$6000 ;ANOTHER POINTER
170  RAMCHR=LENPTR+2 ;DITTO
180  ;
190  SHAPE .BYTE $18,$DB,$42,$7E,$18,$7E,
     $66,$E7 ;A MAN
200  ;
210  START=MOVDAT
220  ;
230  MOVDAT
240  ;
250  ;STORE VALUES IN POINTERS
260  ;
270    LDA #0
280    STA MVSRCE ;LOW BYTE
290    LDA CHBAS ;HIGH BYTE
300    STA MVSRCE+1 ;HIGH BYTE
310  ;
320    LDA #NEWADR&255
330    STA MVDEST
340    LDA #NEWADR/256
350    STA MVDEST+1
360  ;
370    LDA #TABLEN&255
380    STA LENPTR
390    LDA #TABLEN/256
400    STA LENPTR+1
410  ;
420  ;START MOVE
430  ;
440    LDY #0
450    LDX LENPTR+1
```

```
460   BEQ MVPART
470 MVPAGE
480   LDA (MVSRCE),Y
490   STA (MVDEST),Y
500   INY
510   BNE MVPAGE
520   INC MVSRCE+1
530   INC MVDEST+1
540   DEX
550   BNE MVPAGE
560 MVPART
570   LDX LENPTR
580   BEQ MVEXIT
590 MVLAST
600   LDA (MVSRCE),Y
610   STA (MVDEST),Y
620   INY
630   DEX
640   BNE MVLAST
650 MVEXIT
660 ;
670 ;PART II: CHANGE CHARACTER
680 ;
690 ;WE'LL ALTER THE CHARACTER "A"
700 ;
710   LDA #33 ;RAM CODE: "A"
720   STA RAMCHR
730 ;
740 ;NOW WE CALCULATE RAMCHR'S ADR
750 ;
760   LDA #0
770   STA RAMCHR+1 ;CLEARING IT
780   LDA RAMCHR ;#33: AN "A"
790   CLC
800   ASL A ;MULT BY 2
810   ROL RAMCHR+1 ;GET CARRY
820   ASL A ;AGAIN
830   ROL RAMCHR+1
840   ASL A ;AND AGAIN
850   ROL RAMCHR+1
860   STA RAMCHR ;MULT BY 8 DONE
```

```
870 ;
880 FNDADR
890   CLC
900   LDA RAMCHR
910   ADC #NEWADR&255
920   STA CHRADR
930   LDA RAMCHR+1
940   ADC #NEWADR/256
950   STA CHRADR+1
960 ;
970 ;NOW WE CHANGE THE CHARACTER
980 ;
990 NEWCHR
1000    LDY #0 ;NR OF BYTES+1
1010  DOSHAPE
1020    LDA SHAPE,Y
1030    STA (CHRADR),Y
1040    INY
1050    CPY #9
1060    BCC DOSHAPE ;REPEAT TILL DONE
1070 ;
1080 ;STORE NEW CHR SET ADR IN CHBAS
1090 ;
1100    LDA #NEWADR/256 ;HIGH BYTE
1110    STA CHBAS
1120 ;
1130  FINI
1140    RTS
```

Player-Missile Graphics

Although animation can be programmed by flipping through alternate character sets, a far easier way to animate characters in an Atari assembly language program is to take advantage of a special graphics feature of Atari computers called *player-missile graphics*. Player-missile graphics is a technique for programming animation using graphics characters called (not surprisingly) players and missiles. Your Atari computer is equipped with four players, numbered 0 through 3, and four missiles, one for each player. If you write a program that doesn't call for mis-



siles, you can combine your four missiles to form a fifth player. Players and missiles can be used in any graphics mode, and can be drawn and moved around on a video display completely independently of anything else on the screen. They can pass over or under other objects on the screen, and over or under each other. Alternatively, they can be programmed to come to a halt when they run into things, or even to explode upon impact with onscreen objects or with each other!

Each of your Atari's four players is 8-bits wide, just like an ordinary graphics character. But players can be much taller than they are wide. Their maximum height is either 128 or 256 bytes high, depending on their vertical resolution. That means a player can be as tall as the full height of your video screen. Each player has its own color register. So by using players, you can add more color to a program than would ordinarily be available. Players

are usually single color entities. It is possible, however, to merge two or more players into a multicolored player by placing one player on top of another.

Players can have a vertical resolution of either one or two scan lines. The maximum horizontal resolution of a player is eight pixels, but the width of each horizontal pixel is variable; each pixel can be 8 color clocks, 16 color clocks or 32 color clocks wide. This choice is up to the programmer. The missiles used in player-missile graphics are 2-bit wide spots of light that can be used as bullets, stars, or other small graphics objects. Or, as previously mentioned, they can be combined to form a fifth full-size player.

Players are made up of grids of dots, just as standard text characters are. They can therefore be designed on graph paper, in the same way that ordinary text characters are created. Before you start blocking out a player on graph paper, however, it might be a good idea to remember what a player looks like on the screen when all of its bits are filled in. When a player is completely filled in, it looks like a ribbon extending from the top of a video screen to the bottom. You won't need nearly all of that height for most programming needs. Before you start drawing a player, it's usually a good idea to "erase" that entire ribbon by filling it in with zeros. The whole ribbon will thus become invisible. Then you can draw your player in exactly the same way you'd draw a conventional text character, by filling in its shape with "on" bits, or binary ones. When you've finished drawing your players in this way, you can store them almost anywhere in RAM. You can tell your computer where your players and missiles are by simply storing the starting address of the RAM in which they appear in an OS pointer called PMBASE. The address of PMBASE is $D407.

Since players and missiles are graphic objects, it is considered good programming practice to store them in high RAM, just below your computer's screen display. Atari's in-house programmers, who generally know what they're talking about, recommend that you store your player-missile display RAM about 2 K bytes below the top of your Atari computer's memory. Once you've figured out where your player-missile RAM is going to be stored, and consequently, what address PMBASE will point to, you can start storing the data for your players and missiles right

into RAM. Here's a chart showing where the RAM for each player and missile will start, in respect to the address stored in PMBASE:

DOUBLE-LINE RESOLUTION

OFFSET FROM PMBASE	CONTENTS
+ 0 - 383	Unused
+384 - 511	Missiles
+512 - 639	Player 0
+640 - 767	Player 1
+768 - 895	Player 2
+896 - 1023	Player 3
+1023	End of P/M RAM

Must start on a 1K address boundary.

SINGLE-LINE RESOLUTION

OFFSET FROM PMBASE	CONTENTS
+ 0 - 767	Unused
+ 768 - 1023	Missiles
+1024 - 1279	Player 0
+1280 - 1535	Player 1
+1536 - 1791	Player 2
+1792 - 2047	Player 3
+2047	End of P/M RAM

Must start on a 2K address boundary.

When your players and missiles are drawn and stored in RAM, it's fairly simple to move them around on your screen. In your computer's operating system there's a set of memory registers that are used to keep track of the horizontal positions of all players and missiles. These registers are labeled HPOSP0 through HPOSP3 (for players) and HPOSM0 through HPOSM3 (for missiles). The addresses of these registers are:

HORIZONTAL REGISTER FOR:	LABEL	ADDRESS
Player 0	HPOSP0	D000
Player 1	HPOSP1	D001
Player 2	HPOSP2	D002
Player 3	HPOSP3	D003
Missile 0	HPOSM0	D004
Missile 1	HPOSM1	D005
Missile 2	HPOSM2	D006
Missile 3	HPOSM3	D007

When you want to move a player or a missile from one horizontal position to another, all you have to do is change the value of the appropriate horizontal register.

Changing the vertical position of a player or a missile is a little more difficult. To move a player up or down, you have to "erase" the player from the vertical ribbon on which it appears by filling that space in with zeros. Then you have to redraw it in a position higher or lower in the block of RAM that makes up the ribbon.

The utility program below, the final program in this book, is a demonstration of how to use player-missile graphics. As a special bonus, it will also show you how to use a joystick in assembly language programs.

The joystick reading portion of the program begins with a set of instructions setting bit 2 of a register called PACTL (for "Port A ConTroL") located at memory address $D302. When bit 2 of PACTL is set, the reading of game controllers is enabled. The direction switches of a joystick plugged into port A can then be read, in much the same way as they are read in BASIC programs. When you type this program in and run it, you'll be able to move a little pink heart around on your screen using a joystick controller. When the heart disappears from the screen in any direction, keep your joystick switch pressed, and it will soon "wrap around" and reappear on the opposite edge of the screen, just where you would expect it to, moving in the same direction.

This program uses vertical blank interrupts, just like the smooth scrolling routine presented at the beginning of this chapter. As you'll see when you type the program and run it, assembly language is the best possible language to use when you're working with player-missile graphics. If you've ever worked with player-missile graphics using BASIC, you'll soon see that PM/G action is much faster and much smoother when it's programmed in assembly language.

```
10  ;
20  ;PLAYER-MISSILE GRAPHICS ROUTINE
30  ;
50    *=$5000
60    JMP START
70  ;
80  RAMTOP=$6A ;TOP OF RAM PTR
90  VVBLKD=$0224 ;INTERRUPT RTN
100  SDMCTL=$022F ;DMA CNT. SHADOW
110  SDLSTL=$0230 ;SDLST, LOW BYTE
120  STICK0=$0278
130  PCOLR0=$02C0 ;PLAYER COLOR
140  COLOR2=$02C6 ;BKG COLOR
150  ;
160  HPOSP0=$D000 ;PLAYER HORZ PSN
170  GRACTL=$D01D
180  PACTL=$D302 ;JS PORT CNTRL
190  PMBASE=$D407 ;PM BASE ADR
200  SETVBV=$E45C ;ENABLE INTRPT
210  XITVBV=$E462 ;EXIT INTERRUPT
220  ;
230  HRZPTR=$0600 ;HORIZ PSN PTR
240  VRTPTR=HRZPTR+1 ;VRT PSN PTR
250  OURBAS=VRTPTR+1 ;OUR PMBASE
260  TABSIZ=OURBAS+2 ;TABLE SIZE
270  FILVAL=TABSIZ+2 ;BLKFIL VALUE
280  ;
290  TABPTR=$B0 ;TABLE ADDR PTR
300  TABADR=TABPTR+2 ;TABLE ADDRESS
310  ;
320  PLBOFS=512 ;PLAYER BAS OFFS
330  PLTOFS=640 ;PLAYER TOP OFFS
340  ;
350  SHAPE .BYTE $00,$6C,$FE,$FE,$7C,$38,
     $10,$00
360  ;
370  START
380  ;
390  ;CLEAR SCREEN
```

```
400 ;
410   LDA #0
420   STA FILVAL
430   LDA SDLSTL
440   STA TABPTR
450   LDA SDLSTL+1
460   STA TABPTR+1
470   LDA #960&255 ;BYTES PER SCRN
480   STA TABSIZ
490   LDA #960/256
500   STA TABSIZ+1
510   JSR BLKFIL
520 ;
530 ;DEFINE PMG VARIABLES
540 ;
550   LDA #0
560   STA COLOR2 ;BLACK BKG
570   LDA #$58
580   STA PCOLR0 ;PINK PLAYER
590 ;
600   LDA #100 ;SET HORIZ PSN
610   STA HRZPTR
620   STA HPOSP0
630 ;
640   LDA #48 ;SET VERT PSN
650   STA VRTPTR
660 ;
670   LDA #0 ;CLEAR OURBASE
680   STA OURBAS
690   STA OURBAS+1
700 ;
710   SEC
720   LDA RAMTOP
730   SBC #8
740   STA PMBASE ;BASE=RAMTOP-2K
750   STA OURBAS+1 ;SAVE BASE ADR
760 ;
770   LDA #46
780   STA SDMCTL ;ENABLE PM DMA
790 ;
800   LDA #3
```

```
810   STA GRACTL ;ENABLE PM DSPLY
820  ;
830  ;FILL PM RAM W/ZEROS TO CLEAR
840  ;
850   CLC
860   LDA OURBAS
870   ADC #PLBOFS&255
880   STA TABADR
890   STA TABPTR
900   LDA OURBAS+1
910   ADC #PLBOFS/256
920   STA TABADR+1
930   STA TABPTR+1
940  ;
950   SEC
960   LDA #PLTOFS&255
970   SBC #PLBOFS&255
980   STA TABSIZ
990   LDA #PLTOFS/256
1000   SBC #PLBOFS/256
1010   STA TABSIZ+1
1020  ;
1030   LDA #0
1040   STA FILVAL
1050   JSR BLKFIL
1060  ;
1070  ;DEFINE PLAYER
1080  ;
1090  PLAYER
1100  ;
1110  ;DRAW PLAYER
1120  ;
1130   JSR DRAWPL
1140  ;
1150  ;ENABLE INTERRUPT
1160  ;
1170   LDY #INTRPT&255
1180   LDX #INTRPT/256
1190   LDA #7
1200   JSR SETVBV
1210  ;
```

```
1220 INTRPT
1230   LDA #RDSTIK&255
1240   STA VVBLKD
1250   LDA #RDSTIK/256
1260   STA VVBLKD+1
1270 ;
1280 ;INFINITE LOOP
1290 ;
1300 INFIN
1310   JMP INFIN
1320 ;
1330 ;READ JOYSTICK
1340 ;
1350 RDSTIK
1360   LDA #4
1370   ORA PACTL ;SET BIT #2
1380 ;
1390   LDA STICKØ
1400   CMP #$FF ;JS STRAIGHT UP?
1410   BEQ RETURN ;YES, NO ACTION
1420 ;
1430 TRYAGN
1440   CMP #$Ø7 ;RIGHT MOVE
1450   BNE TRYAG2
1460   LDX HRZPTR
1470   INX
1480   STX HRZPTR
1490   STX HPOSPØ
1500   JMP RETURN
1510 ;
1520 TRYAG2
1530   CMP #$ØB ;LEFT MOVE
1540   BNE TRYAG3
1550 ;
1560   LDX HRZPTR
1570   DEX
1580   STX HRZPTR
1590   STX HPOSPØ
1600   JMP RETURN
1610 ;
1620 TRYAG3
```

```
1630   CMP #$ØD ;DOWN MOVE
1640   BNE TRYAG4
1650  ;
1660   INC VRTPTR
1670   JSR DRAWPL
1680   JMP RETURN
1690  ;
1700  TRYAG4
1710   CMP #$ØE ;UP MOVE
1720   BNE RETURN
1730  ;
1740   DEC VRTPTR
1750   JSR DRAWPL
1760   JMP RETURN
1770  ;
1780  RETURN
1790   JMP XITVBV
1800  ;
1810  ;BLOCK FILL ROUTINE
1820  ;
1830  BLKFIL
1840  ;
1850  ;DO FULL PAGES FIRST
1860  ;
1870   LDA FILVAL
1880   LDX TABSIZ+1
1890   BEQ PARTPG
1900   LDY #Ø
1910  FULLPG
1920   STA (TABPTR),Y
1930   INY
1940   BNE FULLPG
1950   INC TABPTR+1
1960   DEX
1970   BNE FULLPG
1980  ;
1990  ;DO REMAINING PARTIAL PAGE
2000  ;
2010  PARTPG
2020   LDX TABSIZ
2030   BEQ FINI
```

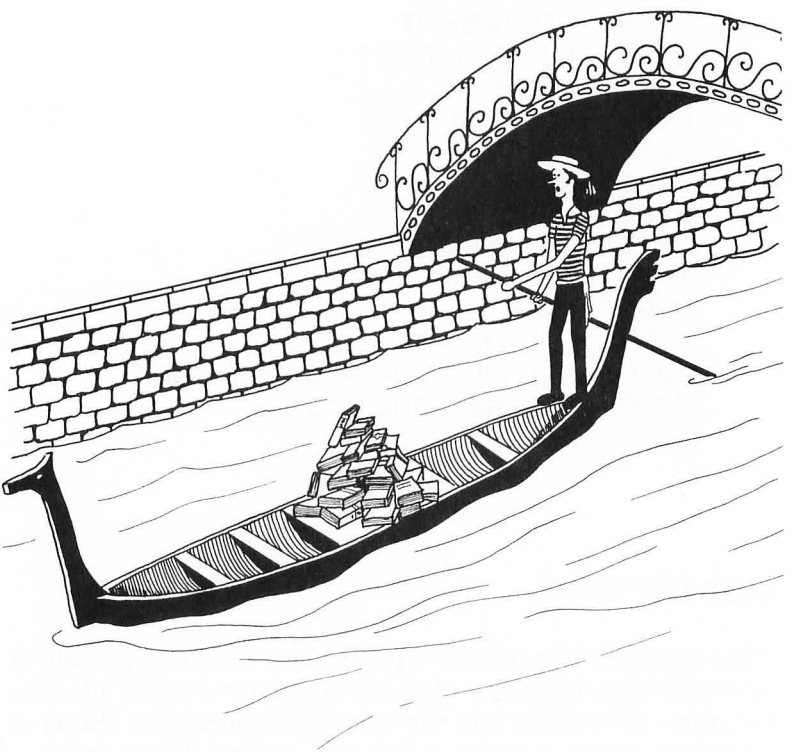
```
2040   LDY #0
2050  ;
2060  PARTLP
2070   STA (TABPTR),Y
2080   INY
2090   DEX
2100   BNE PARTLP
2110  ;
2120  FINI
2130   RTS
2140  ;
2150  DRAWPL
2160   PHA ;SAVE ACC VALUE
2170   CLC
2180   LDA TABADR
2190   ADC VRTPTR
2200   STA TABPTR
2210   LDA TABADR+1
2220   ADC #0
2230   STA TABPTR+1
2240  ;
2250   LDY #0
2260  FILLPL
2270   LDA SHAPE,Y
2280   STA (TABPTR),Y
2290   INY
2300   CPY #8
2310   BCC FILLPL ;REPEAT TILL DONE
2320   PLA ;RESTORE ACC VAL
2330   RTS
```

Not the End

Thus concludes this traveler's guide to the fascinating world of Atari assembly language. If you have typed and saved all of the programs in this book, you now have a fairly extensive library of assembly language routines that you can (no doubt) improve upon, and use in your own programs. If you have absorbed the

material that surrounds the routines in this volume, you now know just about all you need to know to start writing some pretty sophisticated programs in assembly language.

I have just one more suggestion. If you're interested in doing more programming in Atari assembly language — and I certainly hope you are — then there are two other books which you should definitely own. They are (in case you haven't already guessed) *De Re Atari* and *The Atari 400/800 Technical Reference Notes* both published by Atari. With those two books, and the one you have just finished, you should be able to do just about anything you want to do from now on in Atari assembly language.



Appendix A

The 6502 Instruction Set

The following is a complete listing of the 6502 microprocessor instruction set — all of the instruction mnemonics used in Atari assembly language programming. It does not include pseudo operations (also known as pseudo ops, or directives), which vary from assembler to assembler. Here are the meanings of the abbreviations used in this appendix:

Processor Status (P) Register Flags

N — Negative (sign) flag.
V — Overflow flag.
B — Break flag.
D — Decimal flag.
I — Interrupt flag.
Z — Zero flag.
C — Carry flag.

The 6502 Memory Registers

A — Accumulator.
X — X register.
Y — Y register.
M — Memory.

Addressing Modes

A — Absolute addressing.
AC — Accumulator addressing.
Z — Zero page addressing.
IMM — Immediate addressing.

IND — Indirect addressing.
IMP — Implied addressing.
AX — Absolute,X (X-indexed) addressing.
AY — Absolute,Y (Y-indexed) addressing.
IX — Indexed indirect (Indirect,X) addressing.
IY — Indirect indexed (Indirect,Y) addressing.
R — Relative addressing.
ZX — Zero page X-indexed (Zero page,X) addressing.
ZY — Zero page Y-indexed (Zero page,Y) addressing.

The 6502 Instruction Set (6502 Mnemonics)

ADC (Add with carry): Adds the contents of the accumulator to the contents of a specified memory location or literal value. If the P register's carry flag is set, a carry is also added. The result of the addition operation is then stored in the accumulator.

Flags affected: N, V, Z, C.
Registers affected: A.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

AND (Logical AND): Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location or an immediate value. The result of the operation is stored in the accumulator.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

ASL (Arithmetic Shift Left): Moves each bit in the accumulator or a specified memory location one position to the left. A zero is deposited into the the Bit 0 position, and Bit 7 is forced into the carry bit of the P register. The result of the operation is left in the accumulator or the affected memory register.

Flags affected: N, Z, C.
Registers affected: A, M.
Addressing modes: AC, A, Z, AX, ZX.

BCC (Branch if carry clear): Executes a branch if the carry flag is clear, results in no operation if the carry flag is set. Destination of branch must be within a range of –128 to +127 memory addresses from the BCC instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BCS (Branch if carry set): Executes a branch if the carry flag is set, results in no operation if the carry flag is clear. Destination of branch must be within a range of –128 to +127 memory addresses from the BCS instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BEQ (Branch if equal): Executes a branch if the zero flag is set, results in no operation if the zero flag is clear. Can be used to jump to cause a branch if the result of a calculation is zero, or if two numbers are equal. Destination of branch must be within a range of –128 to +127 memory addresses from the BEQ instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BIT (Compare bits in accumulator with bits in a specified memory register): Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location. The content of the accumulator is not affected, but three flags in the P register are. The result of the AND operation is stored in the Z flag. If there is a 1 in both the accumulator and the value in memory at the same bit position, the result is non-zero and the Z flag is cleared. If the bits are different or both zero, the result is zero and the Z flag is set. In addition, bits 6 and 7 of the value in memory being tested are transferred directly into the V and N bits of the status register. This feature of the BIT instruction is often used in signed binary arithmetic. If a BIT operation results in the setting of the N flag, then the value being

tested is negative. If the operation results in the setting of the V flag, that indicates a carry in signed-number math.

Flags affected: N, V, Z.
Registers affected: None.
Addressing modes: A, Z.

BMI (Branch on minus): Executes a branch if the N flag is set, results in no operation if the N flag is clear. Destination of branch must be within a range of –128 to +127 memory addresses from the BMI instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BNE (Branch if not equal): Executes a branch if the zero flag is clear (that is, if the result of an operation is non-zero). Results in no operation if the zero flag is set. Can be used to jump to cause a branch if the result of a calculation is not zero, or if two numbers are not equal. Destination of branch must be within a range of –128 to +127 memory addresses from the BNE instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BPL (Branch on plus): Executes a branch if the N flag is clear (that is, if the result of a calculation is positive). Results in no operation if the N flag is set. Destination of branch must be within a range of –128 to +127 memory addresses from the BPL instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BRK (Break): Halts the execution of a program, much like an interrupt would, and stores the value of the program counter, plus two, on the hardware stack, along with the contents of the P register (which now has the B flag set). BRK is often used in

debugging, and affects various debuggers in various ways. For more details, see your assembler and debugger's instruction manual.

Flags affected: B.
Registers affected: None.
Addressing modes: IMP.

BVC (Branch if overflow clear): Executes a branch if the P register's overflow (V) flag is clear. Results in no operation if the overflow flag is set. This instruction is used primarily in operations involving signed numbers. Destination of the branch must be within a range of –128 to +127 memory addresses from the BVC instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

BVS (Branch if overflow set): Executes a branch if the P register's overflow (V) flag is set. Results in no operation if the overflow flag is clear. This instruction is used primarily in operations involving signed numbers. Destination of the branch must be within a range of –128 to +127 memory addresses from the BVS instruction.

Flags affected: None.
Registers affected: None.
Addressing modes: R.

CLC (Clear carry): Clears the carry bit of the processor status register.

Flags affected: C.
Registers affected: None.
Addressing modes: IMP.

CLD (Clear decimal mode): Puts the computer into binary (its default) mode so that binary operations (the kind most often used) can be carried out properly.

Flags affected: D.
Registers affected: None.
Addressing modes: IMP.

CLI (Clear interrupt mask): Enables interrupts. Used in advanced assembly language programming. For more details, see advanced 6502 assembly language texts and manuals.

Flags affected: I.
Registers affected: None.
Addressing modes: IMP.

CLV (Clear overflow flag): Clears the P register's overflow flag by setting it to zero. This instruction is used primarily in operations involving signed numbers.

Flags affected: V.
Registers affected: None.
Addressing modes: IMP.

CMP (Compare with accumulator): Compares a specified literal number, or the contents of a specified memory location, with the contents of the accumulator. The N, Z and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the accumulator is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C.
Registers affected: None.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

CPX (Compare with X register): Compares a specified literal number, or the contents of a specified memory location, with the contents of the X register. The N, Z and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the X register is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C.
Registers affected: None.
Addressing modes: A, IMM, Z.

CPY (Compare with Y register): Compares a specified literal number, or the contents of a specified memory location, with the contents of the Y register. The N, Z and C flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used, and whether the value in the Y register is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C.
Registers affected: None.
Addressing modes: A, IMM, Z.

DEC (Decrement a memory location): Decrements the contents of a specified memory location by one. If the value in the location is $00, the result of a DEC operation will be $FF, since there is no carry.

Flags affected: N, Z.
Registers affected: M.
Addressing modes: A, Z, AX, ZX.

DEX (Decrement X register): Decrements the X register by one. If the value in the location is $00, the result of the DEX operation will be $FF, since there is no carry.

Flags affected: N, Z.
Registers affected: X.
Addressing modes: IMP.

DEY (Decrement Y register): Decrements the Y register by one. If the value in the location is $00, the result of the DEY operation will be $FF, since there is no carry.

Flags affected: N, Z.
Registers affected: Y.
Addressing modes: IMP.

EOR (Exclusive-OR with accumulator): Performs an Exclusive-OR operation on the contents of the accumulator and a specified literal value or memory location. The N and Z flags are

conditioned in accordance with the result of the operation, and the result is stored in the accumulator.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: A, Z, I, AX, AY, IX, IY, ZX.

INC (Increment memory): The contents of a specified memory location are incremented by one. If the value in the location is $FF, the result of the INC operation will be $00, since there is no carry.

Flags affected: N, Z.
Registers affected: M.
Addressing modes: A, Z, AX, ZX.

INX (Increment X register): The contents of the X register are incremented by one. If the value of the X register is $FF, the result of the INX operation will be $00, since there is no carry.

Flags affected: N, Z.
Registers affected: X.
Addressing modes: IMP.

INY (Increment Y register): The contents of the Y register are incremented by one. If the value of the Y register is $FF, the result of the INY operation will be $00, since there is no carry.

Flags affected: N, Z.
Registers affected: Y.
Addressing modes: IMP.

JMP (Jump to address): Causes program execution to jump to the address specified. The JMP instruction can be used with absolute addressing, and it is the only 6502 instruction that can be used with indirect addressing. A statement that uses indirect addressing is written using the format

```
JMP ($0600)
```

If this statement were used in a program, the JMP instruction would cause program execution to jump to the value stored in memory address $0600, not to that address.

Flags affected: None.
Registers affected: None.
Addressing modes: A, IND.

JSR (Jump to subroutine): Causes program execution to jump to the address that follows the instruction. That address should be the starting address of a subroutine that ends with the instruction RTS. When the program reaches that RTS instruction, execution of the program returns to the next instruction after the JSR instruction that caused the jump to the subroutine.

Flags affected: None.
Registers affected: None.
Addressing modes: A.

LDA (Load the accumulator): Loads the accumulator with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the accumulator, and the Z flag is set if the value loaded into the accumulator is zero.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

LDX (Load the X register): Loads the X register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the X register, and the Z flag is set if the value loaded into the X register is zero.

Flags affected: N, Z.
Registers affected: X.
Addressing modes: A, Z, IMM, AY, ZY.

LDY (Load the Y register): Loads the Y register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the Y register, and the Z flag is set if the value loaded into the Y register is zero.

Flags affected: N, Z.
Registers affected: Y.
Addressing modes: A, Z, IMM, AX, ZX.

LSR (Logical shift right): Each bit in the accumulator is moved one position to the right. A zero is deposited into the bit 7 position, and bit 0 is deposited into the carry. The result is left in the accumulator or in the affected memory register.

Flags affected: N, Z, C.
Registers affected: A, M.
Addressing modes: AC, A, Z, AX, ZX.

NOP (No operation): Causes the computer to do nothing for two cycles. Used in delay loops and to synchronize the timing of computer operations.

Flags affected: None.
Registers affected: None.
Addressing modes: IMP.

ORA (Inclusive-OR with the accumulator): Performs a binary inclusive-OR operation on the value in the accumulator and a literal value or the contents of a specified memory location. The N and Z flags are conditioned in accordance with the result of the operation, and the result of the operation is deposited in the accumulator.

Flags affected: N, Z.
Registers affected: A, M.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

PHA (Push accumulator): The contents of the accumulator are pushed on the stack. The accumulator and the P register are not affacted.

Flags affected: None.
Registers affected: None.
Addressing modes: IMP.

PHP (Push processor status). The contents of the P register are pushed on the stack. The P register itself is left unchanged, and no other registers are affected.

Flags affected: None.
Registers affected: None.
Addressing modes: IMP.

PLA (Pull accumulator): One byte is removed from the stack and deposited in the accumulator. The N and Z flags are conditioned, just as if an LDA operation had been carried out.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: IMP.

PLP (Pull processor status): One byte is removed from the stack and deposited in the P register. This instruction is used to retrieve the status of the P register after it has been saved by pushing it onto the stack. All of the flags are thus conditioned to reflect the original status of the P register.

Flags affected: N, V, B, D, I, Z, C.
Registers affected: None.
Addressing modes: IMP.

ROL (Rotate left): Each bit in the accumulator or a specified memory location is moved one position to the left. The carry bit is deposited into the bit 0 location, and is replaced by bit 7 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C.
Registers affected: A, M.
Addressing modes: AC, A, Z, AX, ZX.

ROR (Rotate right): Each bit in the accumulator or a specified memory location is moved one position to the right. The carry bit is deposited into the bit 7 location, and is replaced by bit 0 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C.
Registers affected: A, M.
Addressing modes: AC, A, Z, AX, ZX.

RTI (Return from interrupt): The status of both the program counter and the P register are restored in preparation for resuming the routine that was in progress when an interrupt occurred. All flags of the P register are restored to their original values. Interrupts are used in advanced assembly language programs, and detailed information on interrupts is available in advanced assembly language texts and Atari reference manuals.

Flags affected: N, V, B, D, I, Z, C.
Registers affected: None.
Addressing modes: IMP.

RTS (Return from subroutine): At the end of a subroutine, returns execution of a program to the next address after the JSR (jump to subroutine) instruction that caused the program to jump to the subroutine.

Flags affected: None.
Registers affected: None.
Address modes: IMP.

SBC (Subtract with carry): Subtracts a literal value or the contents of a specified memory location from the contents of the accumulator. The opposite of the carry is also subtracted — in other words, there is a borrow. The N, V, Z and C flags are all conditioned by this operation, and the result of the operation is deposited in the accumulator.

Flags affected: N, V, Z, C.
Registers affected: A.
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX.

SEC (Set carry): The carry flag is set. This instruction usually precedes an SBC instruction. Its primary purpose is to set the carry flag so that there can be a borrow.

Flags affected: C.
Registers affected: None.
Addressing modes: IMP.

SED (Set decimal mode): Prepares the computer for operations using BCD (binary coded decimal) numbers. BCD arithmetic is more accurate than binary arithmetic — the usual type of 6502 arithmetic — but is slower and more difficult to use, and consumes more memory. BCD arithmetic is usually used in accounting and bookkeeping programs, and in floating point arithmetic.

Flags affected: D.
Registers affected: None.
Addressing modes: IMP.

SEI (Set interrupt disable): Disables the interrupt response to an IRQ (maskable interrupt). Does not disable the response to an NMI (non-maskable interrupt). Interrupts are used in advanced assembly language programming, and are described in advanced assembly language texts and Atari reference manuals.

Flags affected: I.
Registers affected: None.
Addressing modes: IMP.

STA (Store accumulator): Stores the contents of the accumulator in a specified memory location. The contents of the accumulator are not affected.

Flags affected: None.
Registers affected: M.
Addressing modes: A, Z, AX, AY, IX, IY, ZX.

STX (Store X register): Stores the contents of the X register in a specified memory location. The contents of the X register are not affected.

Flags affected: None.
Registers affected: M.
Addressing modes: A, Z, ZY.

STY (Store Y register): Stores the contents of the Y register in a specified memory location. The contents of the Y register are not affected.

Flags affected: None.
Registers affected: M.
Addressing modes: A, Z, ZX.

TAX (Transfer accumulator to X register): The value in the accumulator is deposited in the X register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z.
Registers affected: X.
Addressing modes: IMP.

TAY (Transfer accumulator to Y register): The value in the accumulator is deposited in the Y register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z.
Registers affected: Y.
Addressing modes: IMP.

TSX (Transfer stack to X register): The value of the stack pointer is deposited in the X register. The N and Z flags are conditioned in accordance with the result of this operation. The value of the stack pointer is not changed.

Flags affected: N, Z.
Registers affected: X.
Addressing modes: IMP.

TXA (Transfer X register to accumulator): The value in the X register is deposited in the accumulator. The N and Z flags are conditioned in accordance with the result of this operation. The value of the X register is not changed.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: IMP.

TXS (Transfer X register to stack): The value in the X register is deposited to the stack pointer. No flags are conditioned by this operation. The value of the X register is not changed.

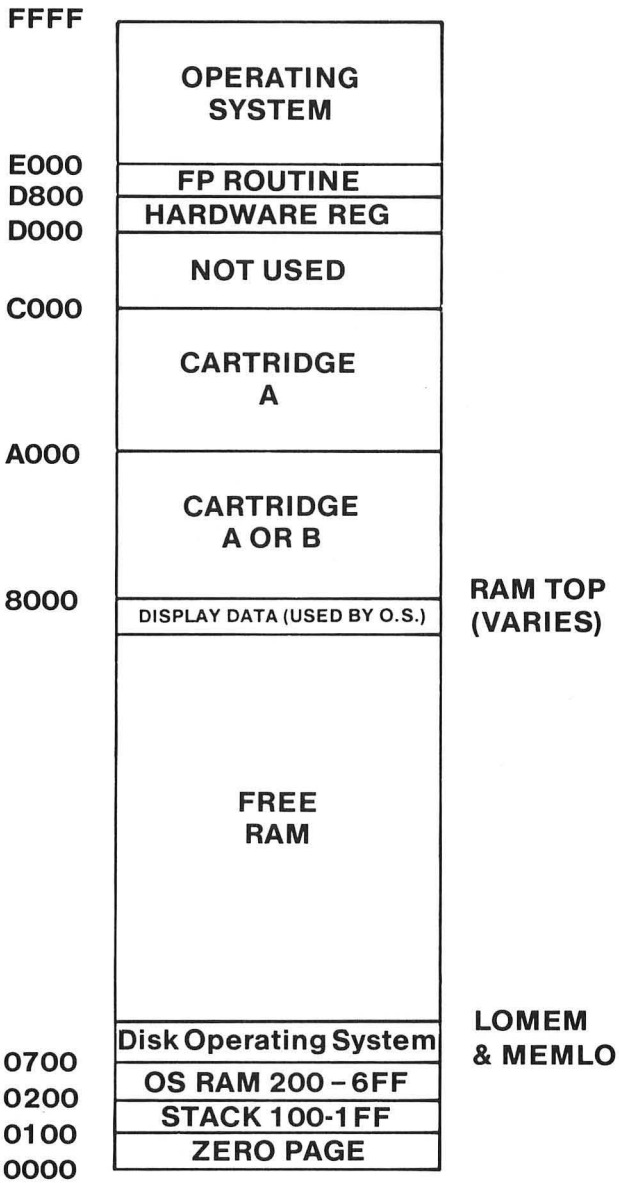
Flags affected: None.
Registers affected: None.
Addressing modes: IMP.

TYA (Transfer Y register to accumulator): The value in the Y register is deposited in the accumulator. The N and Z flags are conditioned by this operation. The value of the Y register is not changed.

Flags affected: N, Z.
Registers affected: A.
Addressing modes: IMP.

Appendix B

Atari Memory Map



Appendix C

For More Information

You can't say everything there is to say about Atari assembly language in just one volume. So here's a list of books you may want to check out, browse through, and perhaps even buy as you continue your study of assembly language programming. I found many of these books to be quite helpful when I was learning assembly language, and some of them still come in handy today. Two books that are practically indispensible to Atari assembly language programmers–although they can sometimes be impossibly difficult to understand — are *De Re Atari* and *The Atari 400/800 Technical Reference Manual*, both published by Atari.

Other useful Atari produced books include the Atari *BASIC Reference Manual*, the Atari *Assembler Editor Reference Manual*, and *The Atari 400/800 Operating System Source Listing*. There's also a very good general user's guide to Atari computers called, logically enough, *Your Atari Computer*. It was written by Ian Poole with Martin McNiff and Steven Cook. It's mostly about programming in BASIC, but it also contains a wealth of information that's useful, if not indispensible, to Atari assembly language programmers. For owners of Atari XL series computers, there's also *User's Guide to the Atari 600XL and 800XL* (New York: Macmillan, 1983).

Generic Books

There are several good generic books on 6502 assembly language programming. Two of the best are *Programming the 6502* by Rodnay Zaks (Berkeley: Sybex, 1980) and *6502 Assembly Language Programming* by Lance A. Leventhal (Berkeley: Osborne/McGraw-Hill, 1979). Leventhal is also the co-author (with Winthrop Saville) of *6502 Assembly Language Routines*, an

excellent book that contains many assembly language subroutines you can use in your own programs. Another generic book that contains some interesting routines is the *6502 Software Gourmet Guide & Cookbook* by Robert Findley (Rochelle Park, NJ: Hayden, 1979).

Atari-specific Guides

There are also a few small books that deal specifically with Atari assembly language programming. They have their flaws, but you may be able to find something useful in them. Two Atari specific books are *The Atari Assembler* by Don and Kurt Inman (Reston, VA: Reston Publishing Co., Inc., 1981) and *How to Program Your Atari in 6502 Machine Language* by Sam D. Roberts (Pomona, CA: Elcomp Publishing, Inc., 1982).

There are a couple of other Atari specific books that have limited aims, but fulfill them very nicely. One is the *Master Memory Map for Atari 400/800 Computers* by Robin Alan Sherer, published in 1982 by Educational Software Inc. The other is *Mapping the Atari* by Ian Chadwick (COMPUTE! Books, Greensboro, NC, 1983).

There are also a few assembly language texts that were written for 6502 based computers other than Ataris, but still contain information that can be very helpful to Atari programmers. These works include *Using 6502 Assembly Language* by Randy Hyde (Chatsworth, CA: DATAMOST, Inc., 1982), and *Assembly Lines: The Book* by Roger Wagner (North Hollywood, CA: Softalk Publishing, 1982).

INDEX

D

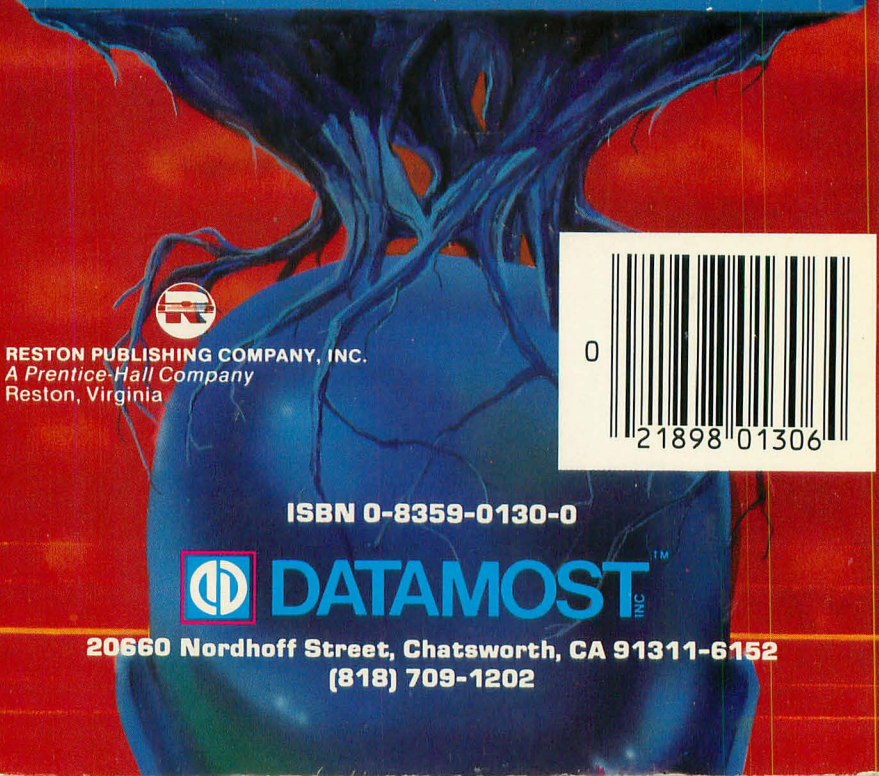
E

F

ATARI ROOTS

- Write programs that would be impossible to write in BASIC, and that run 10 to 1,000 times faster.
- Create your own customized character sets.
- Design screens using text, player-missile graphics and character animation.
- Create complex sound effects and use graphics modes that BASIC does not support.
- All programs will work without changes on either the MAC/65 assembler or the Atari Assembler Editor.

Written in plain English, ATARI ROOTS will teach you how to program your Atari Home Computer in 6502 assembly language — the fastest, most memory-efficient programming language.



RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0 21898 01306

ISBN 0-8359-0130-0

DATAMOST™

20660 Nordhoff Street, Chatsworth, CA 91311-6152
(818) 709-1202